Serial Peripheral Interface (SPI)

SPI is the "Serial Peripheral Interface", widely used with embedded systems because it is a simple and efficient interface: basically a multiplexed shift register. Its three signal wires hold a clock (SCK, often in the range of 1-20 MHz), a "Master Out, Slave In" (MOSI) data line, and a "Master In, Slave Out" (MISO) data line. SPI is a full duplex protocol; for each bit shifted out the MOSI line (one per clock) another is shifted in on the MISO line. Those bits are assembled into words of various sizes on the way to and from system memory. An additional chipselect line is usually active-low (nCS); four signals are normally used for each peripheral, plus sometimes an interrupt.

The SPI bus facilities listed here provide a generalized interface to declare SPI busses and devices, manage them according to the standard Linux driver model, and perform input/output operations. At this time, only "master" side interfaces are supported, where Linux talks to SPI peripherals and does not implement such a peripheral itself. (Interfaces to support implementing SPI slaves would necessarily look different.)

The programming interface is structured around two kinds of driver, and two kinds of device. A "Controller Driver" abstracts the controller hardware, which may be as simple as a set of GPIO pins or as complex as a pair of FIFOs connected to dual DMA engines on the other side of the SPI shift register (maximizing throughput). Such drivers bridge between whatever bus they sit on (often the platform bus) and SPI, and expose the SPI side of their device as a `struct spi_master`. SPI devices are children of that master, represented as a `struct spi_device` and manufactured from `struct spi_board_info` descriptors which are usually provided by board-specific initialization code. A `struct spi_driver` is called a "Protocol Driver", and is bound to a spi_device using normal driver model calls.

The I/O model is a set of queued messages. Protocol drivers submit one or more `struct spi_message` objects, which are processed and completed asynchronously. (There are synchronous wrappers, however.) Messages are built from one or more `struct spi_transfer` objects, each of which wraps a full duplex SPI transfer. A variety of protocol tweaking options are needed, because different chips adopt very different policies for how they use the bits transferred with SPI.

struct `spi_statistics`

statistics for spi transfers

Definition

```
struct spi_statistics {
  spinlock_t lock;
  unsigned long           messages;
  unsigned long           transfers;
  unsigned long           errors;
  unsigned long           timedout;
  unsigned long           spi_sync;
  unsigned long           spi_sync_immediate;
  unsigned long           spi_async;
  unsigned long long      bytes;
  unsigned long long      bytes_rx;
  unsigned long long      bytes_tx;
#define SPI_STATISTICS_HISTO_SIZE 17;
  unsigned long transfer_bytes_histo[SPI_STATISTICS_HISTO_SIZE];
  unsigned long transfers_split_maxsize;
};
```

Members

`lock`
: lock protecting this structure

`messages`
: number of spi-messages handled

`transfers`
: number of spi_transfers handled

`errors`
: number of errors during spi_transfer

`timedout`
: number of timeouts during spi_transfer

`spi_sync`
: number of times spi_sync is used

`spi_sync_immediate`
: number of times spi_sync is executed immediately in calling context without queuing and scheduling

`spi_async`
: number of times spi_async is used

`bytes`
: number of bytes transferred to/from device

`bytes_rx`
: number of bytes received from device

`bytes_tx`
: number of bytes sent to device

`transfer_bytes_histo`
: transfer bytes histogramm

`transfers_split_maxsize`
: number of transfers that have been split because of maxsize limit

struct `spi_device`

Controller side proxy for an SPI slave device

Definition

```
struct spi_device {
  struct device           dev;
  struct spi_controller   *controller;
  struct spi_controller   *master;
  u32 max_speed_hz;
  u8 chip_select;
  u8 bits_per_word;
  u32 mode;
#define SPI_CPHA        0x01                    ;
#define SPI_CPOL        0x02                    ;
#define SPI_MODE_0      (0|0)                   ;
#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                    ;
#define SPI_LSB_FIRST   0x08                    ;
#define SPI_3WIRE       0x10                    ;
#define SPI_LOOP        0x20                    ;
#define SPI_NO_CS       0x40                    ;
#define SPI_READY       0x80                    ;
#define SPI_TX_DUAL     0x100                   ;
#define SPI_TX_QUAD     0x200                   ;
#define SPI_RX_DUAL     0x400                   ;
#define SPI_RX_QUAD     0x800                   ;
#define SPI_CS_WORD     0x1000                  ;
#define SPI_TX_OCTAL    0x2000                  ;
#define SPI_RX_OCTAL    0x4000                  ;
#define SPI_3WIRE_HIZ   0x8000                  ;
  int irq;
  void *controller_state;
  void *controller_data;
  char modalias[SPI_NAME_SIZE];
  const char              *driver_override;
  int cs_gpio;
  struct gpio_desc        *cs_gpiod;
  uint8_t word_delay_usecs;
  struct spi_statistics   statistics;
};
```

Members

`dev`
: Driver model representation of the device.

`controller`
: SPI controller used with the device.

`master`
: Copy of controller, for backwards compatibility.

`max_speed_hz`
: Maximum clock rate to be used with this chip (on this board); may be changed by the device's driver. The spi_transfer.speed_hz can override this for each transfer.

`chip_select`
: Chipselect, distinguishing chips handled by **controller**.

`bits_per_word`
: Data transfers involve one or more words; word sizes like eight or 12 bits are common. In-memory wordsizes are powers of two bytes (e.g. 20 bit samples use 32 bits). This may be changed by the device's driver, or left at the default (0) indicating protocol words are eight bit bytes. The spi_transfer.bits_per_word can override this for each transfer.

`mode`
The spi mode defines how data is clocked out and in. This may be changed by the device's driver. The "active low" default for chipselect mode can be overridden (by specifying SPI_CS_HIGH) as can the "MSB first" default for each word in a transfer (by specifying SPI_LSB_FIRST).

`irq`
Negative, or the number passed to `request_irq()` to receive interrupts from this device.

`controller_state`
Controller's runtime state

`controller_data`
Board-specific definitions for controller, such as FIFO initialization parameters; from board_info.controller_data

`modalias`
Name of the driver to use with this device, or an alias for that name. This appears in the sysfs "modalias" attribute for driver coldplugging, and in uevents used for hotplugging

`cs_gpio`
LEGACY: gpio number of the chipselect line (optional, -ENOENT when not using a GPIO line) use cs_gpiod in new drivers by opting in on the spi_master.

`cs_gpiod`
gpio descriptor of the chipselect line (optional, NULL when not using a GPIO line)

`word_delay_usecs`
microsecond delay to be inserted between consecutive words of a transfer

`statistics`
statistics for the spi_device

Description

A **spi_device** is used to interchange data between an **SPI** slave (usually a discrete chip) and CPU memory.

In **dev**, the platform_data is used to hold information about this device that's meaningful to the device's protocol driver, but not to its controller. One example might be an identifier for a chip variant with slightly different functionality; another might be information about how this particular board wires the chip's pins.

struct `spi_driver`

Host side "protocol" driver

Definition

```
struct spi_driver {
  const struct spi_device_id *id_table;
  int (*probe)(struct spi_device *spi);
  int (*remove)(struct spi_device *spi);
  void (*shutdown)(struct spi_device *spi);
  struct device_driver    driver;
};
```

Members

`id_table`
: List of SPI devices supported by this driver

`probe`
: Binds this driver to the spi device. Drivers can verify that the device is actually present, and may need to configure characteristics (such as bits_per_word) which weren't needed for the initial configuration done during system setup.

`remove`
: Unbinds this driver from the spi device

`shutdown`
: Standard shutdown callback used during system state transitions such as powerdown/halt and kexec

`driver`
: SPI device drivers should initialize the name and owner field of this structure.

Description

This represents the kind of device driver that uses SPI messages to interact with the hardware at the other end of a SPI link. It's called a "protocol" driver because it works through messages rather than talking directly to SPI hardware (which is what the underlying SPI controller driver does to pass those messages). These protocols are defined in the specification for the device(s) supported by the driver.

As a rule, those device protocols represent the lowest level interface supported by a driver, and it will support upper level interfaces too. Examples of such upper levels include frameworks like MTD, networking, MMC, RTC, filesystem character device nodes, and hardware monitoring.

void **spi_unregister_driver**(struct spi_driver * *sdrv*)

reverse effect of spi_register_driver

Parameters

`struct spi_driver * sdrv`
: the driver to unregister

Context

can sleep

module_spi_driver(*__spi_driver*)

Helper macro for registering a SPI driver

Parameters

`__spi_driver`

spi_driver struct

Description

Helper macro for SPI drivers which do not do anything special in module init/exit. This eliminates a lot of boilerplate. Each module may only use this macro once, and calling it replaces `module_init()` and `module_exit()`

struct spi_controller

interface to SPI master or slave controller

Definition

```
struct spi_controller {
  struct device   dev;
  struct list_head list;
  s16 bus_num;
  u16 num_chipselect;
  u16 dma_alignment;
  u32 mode_bits;
  u32 bits_per_word_mask;
#define SPI_BPW_MASK(bits) BIT((bits) - 1);
#define SPI_BPW_RANGE_MASK(min, max) GENMASK((max) - 1, (min) - 1);
  u32 min_speed_hz;
  u32 max_speed_hz;
  u16 flags;
#define SPI_CONTROLLER_HALF_DUPLEX      BIT(0)  ;
#define SPI_CONTROLLER_NO_RX            BIT(1)  ;
#define SPI_CONTROLLER_NO_TX            BIT(2)  ;
#define SPI_CONTROLLER_MUST_RX          BIT(3)  ;
#define SPI_CONTROLLER_MUST_TX          BIT(4)  ;
#define SPI_MASTER_GPIO_SS              BIT(5)  ;
  bool slave;
  size_t (*max_transfer_size)(struct spi_device *spi);
  size_t (*max_message_size)(struct spi_device *spi);
  struct mutex            io_mutex;
  spinlock_t bus_lock_spinlock;
  struct mutex            bus_lock_mutex;
  bool bus_lock_flag;
  int (*setup)(struct spi_device *spi);
  void (*set_cs_timing)(struct spi_device *spi, u8 setup_clk_cycles, u8
hold_clk_cycles, u8 inactive_clk_cycles);
  int (*transfer)(struct spi_device *spi, struct spi_message *mesg);
  void (*cleanup)(struct spi_device *spi);
  bool (*can_dma)(struct spi_controller *ctlr,struct spi_device *spi, struct
spi_transfer *xfer);
  bool queued;
  struct kthread_worker           kworker;
  struct task_struct              *kworker_task;
  struct kthread_work             pump_messages;
  spinlock_t queue_lock;
  struct list_head                queue;
  struct spi_message              *cur_msg;
  bool idling;
  bool busy;
  bool running;
  bool rt;
  bool auto_runtime_pm;
  bool cur_msg_prepared;
  bool cur_msg_mapped;
  struct completion               xfer_completion;
  size_t max_dma_len;
  int (*prepare_transfer_hardware)(struct spi_controller *ctlr);
  int (*transfer_one_message)(struct spi_controller *ctlr, struct spi_message
*mesg);
  int (*unprepare_transfer_hardware)(struct spi_controller *ctlr);
  int (*prepare_message)(struct spi_controller *ctlr, struct spi_message
*message);
  int (*unprepare_message)(struct spi_controller *ctlr, struct spi_message
*message);
  int (*slave_abort)(struct spi_controller *ctlr);
  void (*set_cs)(struct spi_device *spi, bool enable);
  int (*transfer_one)(struct spi_controller *ctlr, struct spi_device *spi,
struct spi_transfer *transfer);
  void (*handle_err)(struct spi_controller *ctlr, struct spi_message *message);
  const struct spi_controller_mem_ops *mem_ops;
  int *cs_gpios;
  struct gpio_desc        **cs_gpiods;
  bool use_gpio_descriptors;
  struct spi_statistics   statistics;
  struct dma_chan         *dma_tx;
  struct dma_chan         *dma_rx;
  void *dummy_rx;
  void *dummy_tx;
```

```
  int (*fw_translate_cs)(struct spi_controller *ctlr, unsigned cs);
};
```

Members

`dev`
device interface to this driver

`list`
link with the global spi_controller list

`bus_num`
board-specific (and often SOC-specific) identifier for a given SPI controller.

`num_chipselect`
chipselects are used to distinguish individual SPI slaves, and are numbered from zero to num_chipselects. each slave has a chipselect signal, but it's common that not every chipselect is connected to a slave.

`dma_alignment`
SPI controller constraint on DMA buffers alignment.

`mode_bits`
flags understood by this controller driver

`bits_per_word_mask`
A mask indicating which values of bits_per_word are supported by the driver. Bit n indicates that a bits_per_word n+1 is supported. If set, the SPI core will reject any transfer with an unsupported bits_per_word. If not set, this value is simply ignored, and it's up to the individual driver to perform any validation.

`min_speed_hz`
Lowest supported transfer speed

`max_speed_hz`
Highest supported transfer speed

`flags`
other constraints relevant to this driver

`slave`
indicates that this is an SPI slave controller

`max_transfer_size`
function that returns the max transfer size for a `spi_device`; may be `NULL`, so the default `SIZE_MAX` will be used.

`max_message_size`
function that returns the max message size for a `spi_device`; may be `NULL`, so the default `SIZE_MAX` will be used.

`io_mutex`
mutex for physical bus access

`bus_lock_spinlock`
spinlock for SPI bus locking

`bus_lock_mutex`
mutex for exclusion of multiple callers

`bus_lock_flag`
indicates that the SPI bus is locked for exclusive use

`setup`
: updates the device mode and clocking records used by a device's **SPI** controller; protocol code may call this. This must fail if an unrecognized or unsupported mode is requested. It's always safe to call this unless transfers are pending on the device whose settings are being modified.

`set_cs_timing`
: optional hook for **SPI** devices to request SPI master controller for configuring specific CS setup time, hold time and inactive delay interms of clock counts

`transfer`
: adds a message to the controller's transfer queue.

`cleanup`
: frees controller-specific state

`can_dma`
: determine whether this controller supports DMA

`queued`
: whether this controller is providing an internal message queue

`kworker`
: thread struct for message pump

`kworker_task`
: pointer to task for message pump kworker thread

`pump_messages`
: work struct for scheduling work to the message pump

`queue_lock`
: **spi** nlock to syncronise access to message queue

`queue`
: message queue

`cur_msg`
: the currently in-flight message

`idling`
: the device is entering idle state

`busy`
: message pump is busy

`running`
: message pump is running

`rt`
: whether this queue is set to run as a realtime task

`auto_runtime_pm`
: the core should ensure a runtime PM reference is held while the hardware is prepared, using the parent device for the **spi** dev

`cur_msg_prepared`
: **spi** _prepare_message was called for the currently in-flight message

`cur_msg_mapped`
: message has been mapped for DMA

`xfer_completion`
: used by core `transfer_one_message()`

`max_dma_len`
: Maximum length of a DMA transfer for the device.

`prepare_transfer_hardware`
: a message will soon arrive from the queue so the subsystem requests the driver to prepare the transfer hardware by issuing this call

`transfer_one_message`
: 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

`unprepare_transfer_hardware`
: there are currently no more messages on the queue so the subsystem notifies the driver that it may relax the hardware by issuing this call

`prepare_message`
: set up the controller to transfer a single message, for example doing DMA mapping. Called from threaded context.

`unprepare_message`
: undo any work done by `prepare_message()`.

`slave_abort`
: abort the ongoing transfer request on an SPI slave controller

`set_cs`
: set the logic level of the chip select line. May be called from interrupt context.

`transfer_one`
: transfer a single spi_transfer. - return 0 if the transfer is finished, - return 1 if the transfer is still in progress. When

    the driver is finished with this transfer it must call `spi_finalize_current_transfer()` so the subsystem can issue the next transfer. Note: transfer_one and transfer_one_message are mutually exclusive; when both are set, the generic subsystem does not call your transfer_one callback.

`handle_err`
: the subsystem calls the driver to handle an error that occurs in the generic implementation of `transfer_one_message()`.

`mem_ops`
: optimized/dedicated operations for interactions with SPI memory. This field is optional and should only be implemented if the controller has native support for memory like operations.

`cs_gpios`
: LEGACY: array of GPIO descs to use as chip select lines; one per CS number. Any individual value may be -ENOENT for CS lines that are not GPIOs (driven by the SPI controller itself). Use the cs_gpiods in new drivers.

`cs_gpiods`
: Array of GPIO descs to use as chip select lines; one per CS number. Any individual value may be NULL for CS lines that are not GPIOs (driven by the SPI controller itself).

`use_gpio_descriptors`
: Turns on the code in the SPI core to parse and grab GPIO descriptors rather than using global GPIO numbers grabbed by the driver. This will fill in **cs_gpiods** and **cs_gpios** should not be used, and SPI devices will have the cs_gpiod assigned rather than cs_gpio.

`statistics`
: statistics for the spi_controller

`dma_tx`
: DMA transmit channel

`dma_rx`
DMA receive channel

`dummy_rx`
dummy receive buffer for full-duplex devices

`dummy_tx`
dummy transmit buffer for full-duplex devices

`fw_translate_cs`
If the boot firmware uses different numbering scheme what Linux expects, this optional hook can be used to translate between the two.

Description

Each SPI controller can communicate with one or more **spi_device** children. These make a small bus, sharing MOSI, MISO and SCK signals but not chip select signals. Each device may be configured to use a different clock rate, since those shared signals are ignored unless the chip is selected.

The driver for an SPI controller manages access to those devices through a queue of spi_message transactions, copying data between CPU memory and an SPI slave device. For each such message it queues, it calls the message's completion function when the transaction completes.

struct spi_res

spi resource management structure

Definition

```
struct spi_res {
  struct list_head        entry;
  spi_res_release_t release;
  unsigned long long      data[];
};
```

Members

`entry`
list entry

`release`
release code called prior to freeing this resource

`data`
extra data allocated for the specific use-case

Description

this is based on ideas from devres, but focused on life-cycle management during spi_message processing

struct spi_transfer

a read/write buffer pair

Definition

```
struct spi_transfer {
  const void      *tx_buf;
  void *rx_buf;
  unsigned len;
  dma_addr_t tx_dma;
  dma_addr_t rx_dma;
  struct sg_table tx_sg;
  struct sg_table rx_sg;
  unsigned cs_change:1;
  unsigned tx_nbits:3;
  unsigned rx_nbits:3;
#define SPI_NBITS_SINGLE        0x01 ;
#define SPI_NBITS_DUAL          0x02 ;
#define SPI_NBITS_QUAD          0x04 ;
  u8 bits_per_word;
  u8 word_delay_usecs;
  u16 delay_usecs;
  u32 speed_hz;
  u16 word_delay;
  struct list_head transfer_list;
};
```

Members

`tx_buf`
data to be written (dma-safe memory), or NULL

`rx_buf`
data to be read (dma-safe memory), or NULL

`len`
size of rx and tx buffers (in bytes)

`tx_dma`
DMA address of tx_buf, if **spi_message.is_dma_mapped**

`rx_dma`
DMA address of rx_buf, if **spi_message.is_dma_mapped**

`tx_sg`
Scatterlist for transmit, currently not for client use

`rx_sg`
Scatterlist for receive, currently not for client use

`cs_change`
affects chipselect after this transfer completes

`tx_nbits`
number of bits used for writing. If 0 the default (SPI_NBITS_SINGLE) is used.

`rx_nbits`
number of bits used for reading. If 0 the default (SPI_NBITS_SINGLE) is used.

`bits_per_word`

select a bits_per_word other than the device default for this transfer. If 0 the default (from **spi_device**) is used.

`word_delay_usecs`
: microseconds to inter word delay after each word size (set by bits_per_word) transmission.

`delay_usecs`
: microseconds to delay after this transfer before (optionally) changing the chipselect status, then starting the next transfer or completing this **spi_message**.

`speed_hz`
: Select a speed other than the device default for this transfer. If 0 the default (from **spi_device**) is used.

`word_delay`
: clock cycles to inter word delay after each word size (set by bits_per_word) transmission.

`transfer_list`
: transfers are sequenced through **spi_message.transfers**

Description

SPI transfers always write the same number of bytes as they read. Protocol drivers should always provide **rx_buf** and/or **tx_buf**. In some cases, they may also want to provide DMA addresses for the data being transferred; that may reduce overhead, when the underlying driver uses dma.

If the transmit buffer is null, zeroes will be shifted out while filling **rx_buf**. If the receive buffer is null, the data shifted in will be discarded. Only "len" bytes shift out (or in). It's an error to try to shift out a partial word. (For example, by shifting out three bytes with word size of sixteen or twenty bits; the former uses two bytes per word, the latter uses four bytes.)

In-memory data values are always in native CPU byte order, translated from the wire byte order (big-endian except with SPI_LSB_FIRST). So for example when bits_per_word is sixteen, buffers are 2N bytes long (**len** = 2N) and hold N sixteen bit words in CPU byte order.

When the word size of the SPI transfer is not a power-of-two multiple of eight bits, those in-memory words include extra bits. In-memory words are always seen by protocol drivers as right-justified, so the undefined (rx) or unused (tx) bits are always the most significant bits.

All SPI transfers start with the relevant chipselect active. Normally it stays selected until after the last transfer in a message. Drivers can affect the chipselect signal using cs_change.

(i) If the transfer isn't the last one in the message, this flag is used to make the chipselect briefly go inactive in the middle of the message. Toggling chipselect in this way may be needed to terminate a chip command, letting a single spi_message perform all of group of chip transactions together.

(ii) When the transfer is the last one in the message, the chip may stay selected until the next transfer. On multi-device SPI busses with nothing blocking messages going to other devices, this is just a performance hint; starting a message to another device deselects this one. But in other cases, this can be used to ensure correctness. Some devices need protocol transactions to be built from a series of spi_message submissions, where the content of one message is determined by the results of previous messages and where the whole transaction ends when the chipselect goes intactive.

When SPI can transfer in 1x,2x or 4x. It can get this transfer information from device through **tx_nbits** and **rx_nbits**. In Bi-direction, these two should both be set. User can set transfer mode with SPI_NBITS_SINGLE(1x) SPI_NBITS_DUAL(2x) and SPI_NBITS_QUAD(4x) to support these three transfer.

The code that submits an spi_message (and its spi_transfers) to the lower layers is responsible for managing its memory. Zero-initialize every field you don't set up explicitly, to insulate against future API updates. After you submit a message and its transfers, ignore them until its completion callback.

struct spi_message

one multi-segment SPI transaction

Definition

```
struct spi_message {
  struct list_head        transfers;
  struct spi_device       *spi;
  unsigned is_dma_mapped:1;
  void (*complete)(void *context);
  void *context;
  unsigned frame_length;
  unsigned actual_length;
  int status;
  struct list_head        queue;
  void *state;
  struct list_head        resources;
};
```

Members

`transfers`
list of transfer segments in this transaction

`spi`
SPI device to which the transaction is queued

`is_dma_mapped`
if true, the caller provided both dma and cpu virtual addresses for each transfer buffer

`complete`
called to report transaction completions

`context`
the argument to `complete()` when it's called

`frame_length`
the total number of bytes in the message

`actual_length`
the total number of bytes that were transferred in all successful segments

`status`
: zero for success, else negative errno

`queue`
: for use by whichever driver currently owns the message

`state`
: for use by whichever driver currently owns the message

`resources`
: for resource management when the spi message is processed

Description

A **spi_message** is used to execute 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 always executed in FIFO order.

The code that submits an spi_message (and its spi_transfers) to the lower layers is responsible for managing its memory. Zero-initialize every field you don't set up explicitly, to insulate against future API updates. After you submit a message and its transfers, ignore them until its completion callback.

void **spi_message_init_with_transfers**(struct spi_message * *m*, struct spi_transfer * *xfers*, unsigned int *num_xfers*)

Initialize spi_message and append transfers

Parameters

`struct spi_message * m`
: spi_message to be initialized

`struct spi_transfer * xfers`
: An array of spi transfers

`unsigned int num_xfers`
: Number of items in the xfer array

Description

This function initializes the given spi_message and adds each spi_transfer in the given array to the message.

struct **spi_replaced_transfers**

structure describing the spi_transfer replacements that have occurred so that they can get reverted

Definition

```
struct spi_replaced_transfers {
  spi_replaced_release_t release;
  void *extradata;
  struct list_head replaced_transfers;
  struct list_head *replaced_after;
  size_t inserted;
  struct spi_transfer inserted_transfers[];
};
```

Members

`release`
: some extra release code to get executed prior to relasing this structure

`extradata`
: pointer to some extra data if requested or NULL

`replaced_transfers`
: transfers that have been replaced and which need to get restored

`replaced_after`
: the transfer after which the **replaced_transfers** are to get re-inserted

`inserted`
: number of transfers inserted

`inserted_transfers`
: array of spi_transfers of array-size **inserted**, that have been replacing replaced_transfers

note

that **extradata** will point to **inserted_transfers**[**inserted**] if some extra allocation is requested, so alignment will be the same as for spi_transfers

int spi_sync_transfer(struct spi_device * *spi*, struct spi_transfer * *xfers*, unsigned int *num_xfers*)

synchronous SPI data transfer

Parameters

`struct spi_device * spi`
: device with which data will be exchanged

`struct spi_transfer * xfers`
: An array of spi_transfers

`unsigned int num_xfers`
: Number of items in the xfer array

Context

can sleep

Description

Does a synchronous SPI data transfer of the given spi_transfer array.

For more specific semantics see `spi_sync()`.

Return

Return: zero on success, else a negative error code.

`int spi_write(struct spi_device * spi, const void * buf, size_t len)`

SPI synchronous write

Parameters

`struct spi_device * spi`
device to which data will be written

`const void * buf`
data buffer

`size_t len`
data buffer size

Context

can sleep

Description

This function writes the buffer **buf**. Callable only from contexts that can sleep.

Return

zero on success, else a negative error code.

`int spi_read(struct spi_device * spi, void * buf, size_t len)`

SPI synchronous read

Parameters

`struct spi_device * spi`
device from which data will be read

`void * buf`
data buffer

`size_t len`
data buffer size

Context

can sleep

Description

This function reads the buffer **buf**. Callable only from contexts that can sleep.

Return

zero on success, else a negative error code.

ssize_t `spi_w8r8`(struct spi_device * *spi*, u8 *cmd*)

SPI synchronous 8 bit write followed by 8 bit read

Parameters

`struct spi_device * spi`
device with which data will be exchanged

`u8 cmd`
command to be written before data is read back

Context

can sleep

Description

Callable only from contexts that can sleep.

Return

the (unsigned) eight bit number returned by the device, or else a negative error code.

ssize_t `spi_w8r16`(struct spi_device * *spi*, u8 *cmd*)

SPI synchronous 8 bit write followed by 16 bit read

Parameters

`struct spi_device * spi`
device with which data will be exchanged

`u8 cmd`
command to be written before data is read back

Context

can sleep

Description

The number is returned in wire-order, which is at least sometimes big-endian.

Callable only from contexts that can sleep.

Return

the (unsigned) sixteen bit number returned by the device, or else a negative error code.

`ssize_t spi_w8r16be(struct spi_device * spi, u8 cmd)`

SPI synchronous 8 bit write followed by 16 bit big-endian read

Parameters

`struct spi_device * spi`
device with which data will be exchanged

`u8 cmd`
command to be written before data is read back

Context

can sleep

Description

This function is similar to spi_w8r16, with the exception that it will convert the read 16 bit data word from big-endian to native endianness.

Callable only from contexts that can sleep.

Return

the (unsigned) sixteen bit number returned by the device in cpu endianness, or else a negative error code.

`struct spi_board_info`

board-specific template for a SPI device

Definition

```
struct spi_board_info {
  char modalias[SPI_NAME_SIZE];
  const void      *platform_data;
  const struct property_entry *properties;
  void *controller_data;
  int irq;
  u32 max_speed_hz;
  u16 bus_num;
  u16 chip_select;
  u32 mode;
};
```

Members

`modalias`
: Initializes spi_device.modalias; identifies the driver.

`platform_data`
: Initializes spi_device.platform_data; the particular data stored there is driver-specific.

`properties`
: Additional device properties for the device.

`controller_data`
: Initializes spi_device.controller_data; some controllers need hints about hardware setup, e.g. for DMA.

`irq`
: Initializes spi_device.irq; depends on how the board is wired.

`max_speed_hz`
: Initializes spi_device.max_speed_hz; based on limits from the chip datasheet and board-specific signal quality issues.

`bus_num`
: Identifies which spi_controller parents the spi_device; unused by `spi_new_device()`, and otherwise depends on board wiring.

`chip_select`
: Initializes spi_device.chip_select; depends on how the board is wired.

`mode`
: Initializes spi_device.mode; based on the chip datasheet, board wiring (some devices support both 3WIRE and standard modes), and possibly presence of an inverter in the chipselect path.

Description

When adding new SPI devices to the device tree, these structures serve as a partial device template. They hold information which can't always be determined by drivers. Information that `probe()` can establish (such as the default transfer wordsize) is not included here.

These structures are used in two places. Their primary role is to be stored in tables of board-specific device descriptors, which are declared early in board initialization and then used (much later) to populate a controller's device tree after the that controller's driver initializes. A secondary (and atypical) role is as a parameter to `spi_new_device()` call, which happens after those controller drivers are active in some dynamic board configuration models.

int spi_register_board_info(struct spi_board_info const * *info*, unsigned *n*)

register SPI devices for a given board

Parameters

`struct spi_board_info const * info`
array of chip descriptors

`unsigned n`
how many descriptors are provided

Context

can sleep

Description

Board-specific early init code calls this (probably during arch_initcall) with segments of the SPI device table. Any device nodes are created later, after the relevant parent SPI controller (bus_num) is defined. We keep this table of devices forever, so that reloading a controller driver will not make Linux forget about these hard-wired devices.

Other code can also call this, e.g. a particular add-on board might provide SPI devices through its expansion connector, so code initializing that board would naturally declare its SPI devices.

The board info passed can safely be __initdata ... but be careful of any embedded pointers (platform_data, etc), they're copied as-is. Device properties are deep-copied though.

Return

zero on success, else a negative error code.

int __spi_register_driver(struct module * *owner*, struct spi_driver * *sdrv*)

register a SPI driver

Parameters

`struct module * owner`
owner module of the driver to register

`struct spi_driver * sdrv`
the driver to register

Context

can sleep

Return

zero on success, else a negative error code.

struct spi_device * **spi_alloc_device**(struct spi_controller * *ctlr*)

Allocate a new SPI device

Parameters

`struct spi_controller * ctlr`
Controller to which device is connected

Context

can sleep

Description

Allows a driver to allocate and initialize a spi_device without registering it immediately. This allows a driver to directly fill the spi_device with device parameters before calling `spi_add_device()` on it.

Caller is responsible to call `spi_add_device()` on the returned spi_device structure to add it to the SPI controller. If the caller needs to discard the spi_device without adding it, then it should call `spi_dev_put()` on it.

Return

a pointer to the new device, or NULL.

int **spi_add_device**(struct spi_device * *spi*)

Add spi_device allocated with spi_alloc_device

Parameters

`struct spi_device * spi`
spi_device to register

Description

Companion function to spi_alloc_device. Devices allocated with spi_alloc_device can be added onto the spi bus with this function.

Return

0 on success; negative errno on failure

struct spi_device * spi_new_device(struct spi_controller * *ctlr*, struct spi_board_info * *chip*)

instantiate one new SPI device

Parameters

`struct spi_controller * ctlr`
Controller to which device is connected

`struct spi_board_info * chip`
Describes the SPI device

Context

can sleep

Description

On typical mainboards, this is purely internal; and it's not needed after board init creates the hard-wired devices. Some development platforms may not be able to use spi_register_board_info though, and this is exported so that for example a USB or parport based adapter driver could add devices (which it would learn about out-of-band).

Return

the new device, or NULL.

void spi_unregister_device(struct spi_device * *spi*)

unregister a single SPI device

Parameters

`struct spi_device * spi`
spi_device to unregister

Description

Start making the passed SPI device vanish. Normally this would be handled by `spi_unregister_controller()`.

void spi_finalize_current_transfer(struct spi_controller * *ctlr*)

report completion of a transfer

Parameters

`struct spi_controller * ctlr`
: the controller reporting completion

Description

Called by SPI drivers using the core `transfer_one_message()` implementation to notify it that the current interrupt driven transfer has finished and the next one may be scheduled.

struct spi_message * **spi_get_next_queued_message**(struct spi_controller * *ctlr*)

called by driver to check for queued messages

Parameters

`struct spi_controller * ctlr`
: the controller to check for queued messages

Description

If there are more messages in the queue, the next message is returned from this call.

Return

the next message in the queue, else NULL if the queue is empty.

void **spi_finalize_current_message**(struct spi_controller * *ctlr*)

the current message is complete

Parameters

`struct spi_controller * ctlr`
: the controller to return the message to

Description

Called by the driver to notify the core that the message in the front of the queue is complete and can be removed from the queue.

int **spi_slave_abort**(struct spi_device * *spi*)

abort the ongoing transfer request on an SPI slave controller

Parameters

`struct spi_device * spi`
: device used for the current transfer

struct spi_controller * `__spi_alloc_controller`(struct device * *dev*, unsigned int *size*, bool *slave*)

allocate an SPI master or slave controller

Parameters

`struct device * dev`
: the controller, possibly using the platform_bus

`unsigned int size`
: how much zeroed driver-private data to allocate; the pointer to this memory is in the driver_data field of the returned device, accessible with `spi_controller_get_devdata()`.

`bool slave`
: flag indicating whether to allocate an SPI master (false) or SPI slave (true) controller

Context

can sleep

Description

This call is used only by SPI controller drivers, which are the only ones directly touching chip registers. It's how they allocate an spi_controller structure, prior to calling `spi_register_controller()`.

This must be called from context that can sleep.

The caller is responsible for assigning the bus number and initializing the controller's methods before calling `spi_register_controller()`; and (after errors adding the device) calling `spi_controller_put()` to prevent a memory leak.

Return

the SPI controller structure on success, else NULL.

int `spi_register_controller`(struct spi_controller * *ctlr*)

register SPI master or slave controller

Parameters

`struct spi_controller * ctlr`
: initialized master, originally from `spi_alloc_master()` or `spi_alloc_slave()`

Context

can sleep

Description

SPI controllers connect to their drivers using some non-SPI bus, such as the platform bus. The final stage of `probe()` in that code includes calling `spi_register_controller()` to hook up to this SPI bus glue.

SPI controllers use board specific (often SOC specific) bus numbers, and board-specific addressing for SPI devices combines those numbers with chip select numbers. Since SPI does not directly support dynamic device identification, boards need configuration tables telling which chip is at which address.

This must be called from context that can sleep. It returns zero on success, else a negative error code (dropping the controller's refcount). After a successful return, the caller is responsible for calling `spi_unregister_controller()`.

Return

zero on success, else a negative error code.

int **devm_spi_register_controller**(struct device * *dev*, struct spi_controller * *ctlr*)

register managed SPI master or slave controller

Parameters

`struct device * dev`
device managing SPI controller

`struct spi_controller * ctlr`
initialized controller, originally from `spi_alloc_master()` or `spi_alloc_slave()`

Context

can sleep

Description

Register a SPI device as with `spi_register_controller()` which will automatically be unregistered and freed.

Return

zero on success, else a negative error code.

void **spi_unregister_controller**(struct spi_controller * *ctlr*)

unregister SPI master or slave controller

Parameters

`struct spi_controller * ctlr`
the controller being unregistered

Context

can sleep

Description

This call is used only by **SPI** controller drivers, which are the only ones directly touching chip registers.

This must be called from context that can sleep.

Note that this function also drops a reference to the controller.

```
struct spi_controller * spi_busnum_to_master(u16 bus_num)
```

look up master associated with bus_num

Parameters

`u16 bus_num`
the master's bus number

Context

can sleep

Description

This call may be used with devices that are registered after arch init time. It returns a refcounted pointer to the relevant **spi**_controller (which the caller must release), or NULL if there is no such master registered.

Return

the **SPI** master structure on success, else NULL.

```
void * spi_res_alloc(struct spi_device * spi, spi_res_release_t release, size_t size, gfp_t gfp)
```

allocate a **spi** resource that is life-cycle managed during the processing of a spi_message while using spi_transfer_one

Parameters

`struct spi_device * spi`
: the **spi** device for which we allocate memory

`spi_res_release_t release`
: the release code to execute for this resource

`size_t size`
: size to alloc and return

`gfp_t gfp`
: GFP allocation flags

Return

the pointer to the allocated data

This may get enhanced in the future to allocate from a memory pool of the **spi_device** or **spi_controller** to avoid repeated allocations.

void spi_res_free(void * *res*)

free an **spi** resource

Parameters

`void * res`
: pointer to the custom data of a resource

void spi_res_add(struct spi_message * *message*, void * *res*)

add a **spi**_res to the spi_message

Parameters

`struct spi_message * message`
: the **spi** message

`void * res`
: the **spi**_resource

void spi_res_release(struct spi_controller * *ctlr*, struct spi_message * *message*)

release all **spi** resources for this message

Parameters

`struct spi_controller * ctlr`
: the **spi_controller**

`struct spi_message * message`
: the **spi_message**

struct spi_replaced_transfers * **spi_replace_transfers**(struct spi_message * *msg*, struct spi_transfer * *xfer_first*, size_t *remove*, size_t *insert*, spi_replaced_release_t *release*, size_t *extradatasize*, gfp_t *gfp*)

replace transfers with several transfers and register change with spi_message.resources

Parameters

`struct spi_message * msg`
: the spi_message we work upon

`struct spi_transfer * xfer_first`
: the first spi_transfer we want to replace

`size_t remove`
: number of transfers to remove

`size_t insert`
: the number of transfers we want to insert instead

`spi_replaced_release_t release`
: extra release code necessary in some circumstances

`size_t extradatasize`
: extra data to allocate (with alignment guarantees of struct **spi_transfer**)

`gfp_t gfp`
: gfp flags

Return

pointer to spi_replaced_transfers,
: PTR_ERR(...) in case of errors.

int **spi_split_transfers_maxsize**(struct spi_controller * *ctlr*, struct spi_message * *msg*, size_t *maxsize*, gfp_t *gfp*)

split spi transfers into multiple transfers when an individual transfer exceeds a certain size

Parameters

`struct spi_controller * ctlr`
: the **spi_controller** for this transfer

`struct spi_message * msg`
: the **spi_message** to transform

`size_t maxsize`
: the maximum when to apply this

`gfp_t gfp`
: GFP allocation flags

Return

status of transformation

int **spi_setup**(struct spi_device * *spi*)

setup **SPI** mode and clock rate

Parameters

`struct spi_device * spi`
the device whose settings are being modified

Context

can sleep, and no requests are queued to the device

Description

SPI protocol drivers may need to update the transfer mode if the device doesn't work with its 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. Except for SPI_CS_HIGH, which takes effect immediately, the changes take effect the next time the device is selected and data is transferred to or from it. When this function returns, the spi device is deselected.

Note that this call will fail if the protocol driver specifies an option that the underlying controller or its driver does not support. For example, not all hardware supports wire transfers using nine bit words, LSB-first wire encoding, or active-high chipselects.

Return

zero on success, else a negative error code.

void **spi_set_cs_timing**(struct spi_device * *spi*, u8 *setup*, u8 *hold*, u8 *inactive_dly*)

configure CS setup, hold, and inactive delays

Parameters

`struct spi_device * spi`
the device that requires specific CS timing configuration

`u8 setup`
CS setup time in terms of clock count

`u8 hold`
CS hold time in terms of clock count

`u8 inactive_dly`
CS inactive delay between transfers in terms of clock count

int **spi_async**(struct spi_device * *spi*, struct spi_message * *message*)

asynchronous **SPI** transfer

Parameters

`struct spi_device * spi`
device with which data will be exchanged

`struct spi_message * message`
describes the data transfers, including completion callback

Context

any (irqs may be blocked, etc)

Description

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 deselected. 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.)

Return

zero on success, else a negative error code.

int **spi_async_locked**(struct spi_device * *spi*, struct spi_message * *message*)

version of **spi** _async with exclusive bus usage

Parameters

`struct spi_device * spi`
device with which data will be exchanged

`struct spi_message * message`
describes the data transfers, including completion callback

Context

any (irqs may be blocked, etc)

Description

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 deselected. 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.)

Return

zero on success, else a negative error code.

int spi_sync(struct spi_device * *spi*, struct spi_message * *message*)

blocking/synchronous **SPI** data transfers

Parameters

`struct spi_device * spi`
device with which data will be exchanged

`struct spi_message * message`
describes the data transfers

Context

can sleep

Description

This call may only be used from a context that may sleep. The sleep is non-interruptible, and has no timeout. Low-overhead controller drivers may DMA directly into and out of the message buffers.

Note that the **SPI** device's chip select is active during the message, and then is normally disabled between messages. Drivers for some frequently-used devices may want to minimize costs of selecting a chip, by leaving it selected in anticipation that the next message will go to the same chip. (That may increase power usage.)

Also, the caller is guaranteeing that the memory associated with the message will not be freed before this call returns.

Return

zero on success, else a negative error code.

int spi_sync_locked(struct spi_device * *spi*, struct spi_message * *message*)

version of **spi**_sync with exclusive bus usage

Parameters

`struct spi_device * spi`
device with which data will be exchanged

`struct spi_message * message`
describes the data transfers

Context

can sleep

Description

This call may only be used from a context that may sleep. The sleep is non-interruptible, and has no timeout. Low-overhead controller drivers may DMA directly into and out of the message buffers.

This call should be used by drivers that require exclusive access to the **SPI** bus. It has to be preceded by a spi_bus_lock call. The SPI bus must be released by a spi_bus_unlock call when the exclusive access is over.

Return

zero on success, else a negative error code.

int spi_bus_lock(struct spi_controller * *ctlr*)

obtain a lock for exclusive **SPI** bus usage

Parameters

`struct spi_controller * ctlr`
SPI bus master that should be locked for exclusive bus access

Context

can sleep

Description

This call may only be used from a context that may sleep. The sleep is non-interruptible, and has no timeout.

This call should be used by drivers that require exclusive access to the SPI bus. The SPI bus must be released by a spi_bus_unlock call when the exclusive access is over. Data transfer must be done by spi_sync_locked and spi_async_locked calls when the SPI bus lock is held.

Return

always zero.

int **`spi_bus_unlock`**(struct spi_controller * *ctlr*)

release the lock for exclusive SPI bus usage

Parameters

`struct spi_controller * ctlr`
SPI bus master that was locked for exclusive bus access

Context

can sleep

Description

This call may only be used from a context that may sleep. The sleep is non-interruptible, and has no timeout.

This call releases an SPI bus lock previously obtained by an spi_bus_lock call.

Return

always zero.

int spi_write_then_read(struct spi_device * *spi*, const void * *txbuf*, unsigned *n_tx*, void * *rxbuf*, unsigned *n_rx*)

SPI synchronous write followed by read

Parameters

`struct spi_device * spi`
device with which data will be exchanged

`const void * txbuf`
data to be written (need not be dma-safe)

`unsigned n_tx`
size of txbuf, in bytes

`void * rxbuf`
buffer into which data will be read (need not be dma-safe)

`unsigned n_rx`
size of rxbuf, in bytes

Context

can sleep

Description

This performs a half duplex MicroWire style transaction with the device, sending txbuf and then reading rxbuf. The return value is zero for success, else a negative errno status code. This call may only be used from a context that may sleep.

Parameters to this routine are always copied using a small buffer; portable code should never use this for more than 32 bytes. Performance-sensitive or bulk transfer code should instead use **spi** _{async,sync}() calls with dma-safe buffers.

Return

zero on success, else a negative error code.