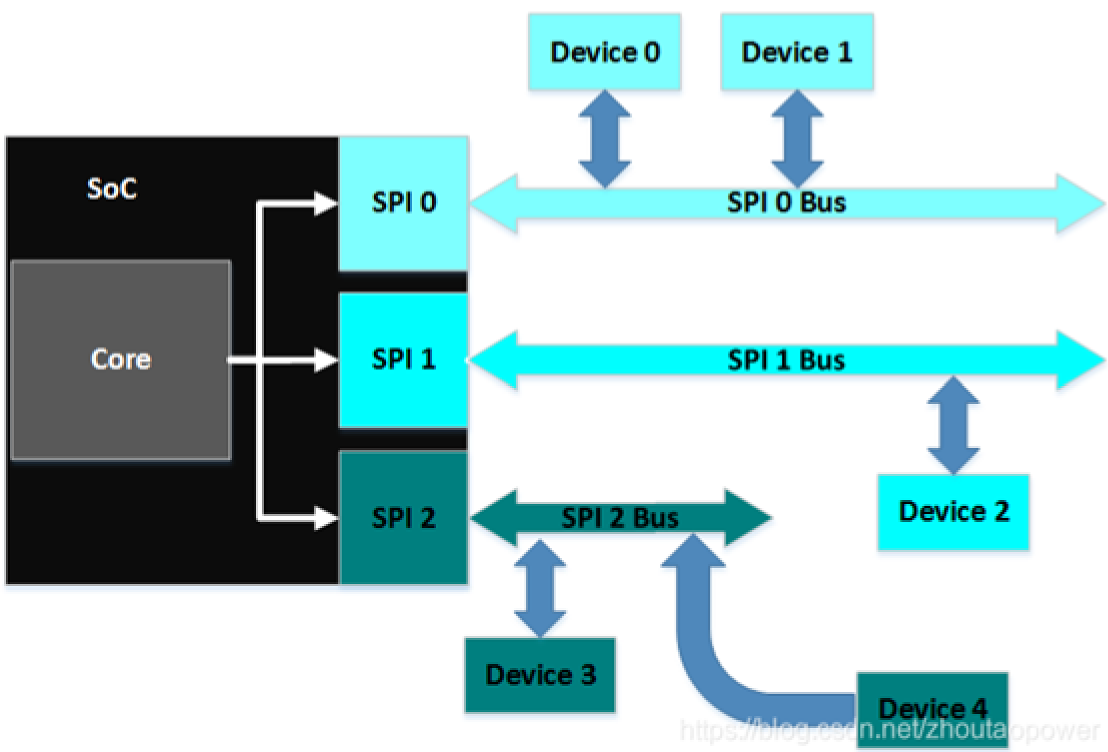
SPI(serial prepheral interface)

SPI是一种高速的串行全双工的接口，在SoC中被广泛的使用。在linux内核中，SPI和我们较为熟悉的platform一样，属于总线的一类，并且拥有自己的一套框架，该框架大致将SPI的驱动分为了以下几个部分：

1. **SPI主机：即SPI controller，老版本中叫做SPI master**
2. **SPI外设驱动：外设指的是SPI controller上的从设备**
3. **SPI 从设备描述**
4. **SPI传输层的动作描述**

SPI硬件拓扑



一般来说SoC上都会提供多个SPI控制器，每个SPI控制器上又可以挂载多个从设备。

内核SPI硬件描述数据结构

根据SPI的硬件拓扑结构，linux内核中的SPI框架将其抽象为主设备（spi_controller），从设备（spi_device）和对应的驱动（spi_driver）。

spi_controller

```
struct spi_controller {
        struct device        dev;

        struct list_head list;

        /* other than negative (== assign one dynamically), bus_num is fully
         * board-specific.  usually that simplifies to being SOC-specific.
         * example:  one SOC has three SPI controllers, numbered 0..2,
         * and one board's schematics might show it using SPI-2.  software
         * would normally use bus_num=2 for that controller.
         */
        s16                        bus_num;

        /* chipselects will be integral to many controllers; some others
         * might use board-specific GPIOs.
         */
        u16                        num_chipselect;

        ...

        /* flag indicating this is an SPI slave controller */
        bool                        slave;

        ...
        int                        (*setup)(struct spi_device *spi);

        /*
         * set_cs_timing() method is for SPI controllers that supports
         * configuring CS timing.
         *
         * This hook allows SPI client drivers to request SPI controllers
         * to configure specific CS timing through spi_set_cs_timing() after
         * spi_setup().
         */
        int (*set_cs_timing)(struct spi_device *spi, struct spi_delay *setup,
                             struct spi_delay *hold, struct spi_delay *inactive);

        /* bidirectional bulk transfers
         *
         * + The transfer() method may not sleep; its main role is
         *   just to add the message to the queue.
         * + For now there's no remove-from-queue operation, or
         *   any other request management
         * + To a given spi_device, message queueing is pure fifo
         *
         * + The controller's main job is to process its message queue,
         *   selecting a chip (for masters), then transferring data
         * + If there are multiple spi_device children, the i/o queue
         *   arbitration algorithm is unspecified (round robin, fifo,
         *   priority, reservations, preemption, etc)
         *
         * + Chipselect stays active during the entire message
         *   (unless modified by spi_transfer.cs_change != 0).
         * + The message transfers use clock and SPI mode parameters
         *   previously established by setup() for this device
         */
        int                        (*transfer)(struct spi_device *spi,
```

```
                                                struct spi_message *mesg);

        /* called on release() to free memory provided by spi_controller */
        void                        (*cleanup)(struct spi_device *spi);

        /*
         * Used to enable core support for DMA handling, if can_dma()
         * exists and returns true then the transfer will be mapped
         * prior to transfer_one() being called.  The driver should
         * not modify or store xfer and dma_tx and dma_rx must be set
         * while the device is prepared.
         */
         ...
        bool                                queued;
        struct kthread_worker                *kworker;
        struct kthread_work                pump_messages;
        spinlock_t                        queue_lock;
        struct list_head                queue;
        ...
        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);

        /*
         * These hooks are for drivers that use a generic implementation
         * of transfer_one_message() provied by the core.
         */
        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);

        ...
};
```

该结构体用来表示一个spi控制器，完整的数据结构可以在/include/linux/spi/spi.h中查看，这里保留了比较重要的成员，包括：

```
        bus_numspispi bus
        num_chipselect:spi
        mode_bitsSPIslave mode
        slaveslavespi
        probesetuptransferset_cs
        kworkerSPI
```

spi_deivce

```
struct spi_device {
        struct device                   dev;
        struct spi_controller           *controller;
        struct spi_controller           *master;        /* compatibility layer */
        u32                            max_speed_hz;
        u8                             chip_select;
        u8                             bits_per_word;
        bool                             rt;
        u32                            mode;
#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 */
#define        SPI_3WIRE        0x10                        /* SI/SO signals shared */
#define        SPI_LOOP        0x20                        /* loopback mode */
#define        SPI_NO_CS        0x40                        /* 1 dev/bus, no chipselect */
#define        SPI_READY        0x80                        /* slave pulls low to pause */
#define        SPI_TX_DUAL        0x100                        /* transmit with 2 wires */
#define        SPI_TX_QUAD        0x200                        /* transmit with 4 wires */
#define        SPI_RX_DUAL        0x400                        /* receive with 2 wires */
#define        SPI_RX_QUAD        0x800                        /* receive with 4 wires */
#define        SPI_CS_WORD        0x1000                        /* toggle cs after each word */
#define        SPI_TX_OCTAL        0x2000                        /* transmit with 8 wires */
#define        SPI_RX_OCTAL        0x4000                        /* receive with 8 wires */
#define        SPI_3WIRE_HIZ        0x8000                        /* high impedance turnaround */
        int                            irq;
        void                            *controller_state;
        void                            *controller_data;
        char                            modalias[SPI_NAME_SIZE];
        const char                     *driver_override;
        int                            cs_gpio;        /* LEGACY: chip select gpio */
        struct gpio_desc        *cs_gpiod;        /* chip select gpio desc */
        struct spi_delay        word_delay; /* inter-word delay */

        /* the statistics */
        struct spi_statistics        statistics;

        /*
         * likely need more hooks for more protocol options affecting how
         * the controller talks to each chip, like:
         *  - memory packing (12 bit samples into low bits, others zeroed)
         *  - priority
         *  - chipselect delays
         *  - ...
         */
};
```

从设备的信息较少，主要包括了：

```
        controller:
        chip_selectnum_chipselect
        modeSPI
        modalias
```

spi_driver

```
/**
 * struct spi_driver - Host side "protocol" driver
 * @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.
 *
 * 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.
 */
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;
};
```

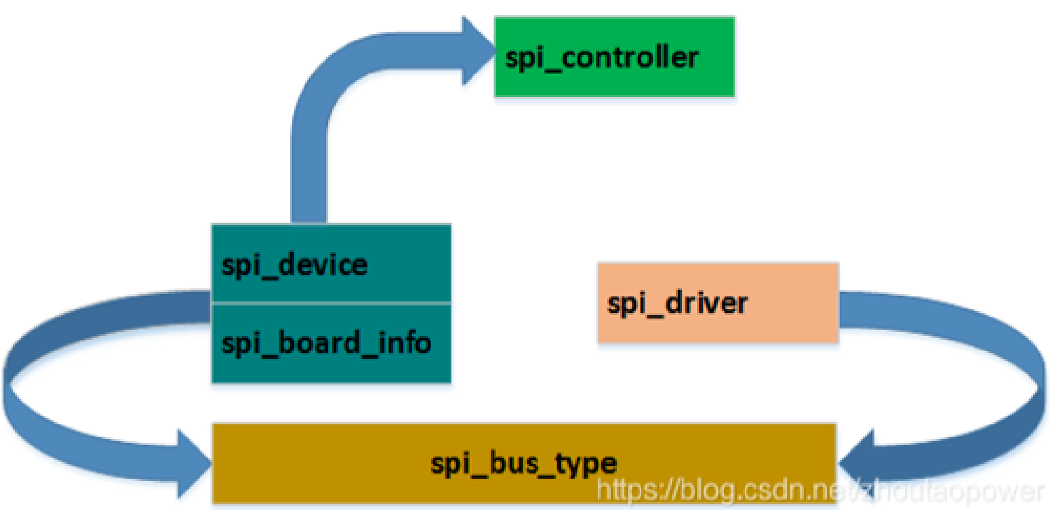
spi_driver中只有三个函数指针，其注释已经说明，该驱动作用于从设备。

spi_board_info

在内核的spi框架中，同样提供了板级启动时使用的从设备结构体，既然是描述从设备的，那么该结构体中的内容和spi_deivce几乎没有差别。

```
struct spi_board_info {
        /* the device name and module name are coupled, like platform_bus;
         * "modalias" is normally the driver name.
         *
         * platform_data goes to spi_device.dev.platform_data,
         * controller_data goes to spi_device.controller_data,
         * device properties are copied and attached to spi_device,
         * irq is copied too
         */
        char                modalias[SPI_NAME_SIZE];
        const void        *platform_data;
        const struct property_entry *properties;
        void                *controller_data;
        int                irq;

        /* slower signaling on noisy or low voltage boards */
        u32                max_speed_hz;


        /* bus_num is board specific and matches the bus_num of some
         * spi_controller that will probably be registered later.
         *
         * chip_select reflects how this chip is wired to that master;
         * it's less than num_chipselect.
         */
        u16                bus_num;
        u16                chip_select;

        /* mode becomes spi_device.mode, and is essential for chips
         * where the default of SPI_CS_HIGH = 0 is wrong.
         */
        u32                mode;

        /* ... may need additional spi_device chip config data here.
         * avoid stuff protocol drivers can set; but include stuff
         * needed to behave without being bound to a driver:
         *  - quirks like clock rate mattering when not selected
         */
};
```

内核中硬件描述关系总结：



SPI信息交换机制分析

在介绍完linux内核中关于SPI硬件部分的抽象后，接下来就到了描述SPI信息交换的环节。这部分主要涉及了两个结构体，spi_transfer和spi_message。

spi_transfer

```
struct spi_transfer {
        /* it's ok if tx_buf == rx_buf (right?)
         * for MicroWire, one buffer must be null
         * buffers must work with dma_*map_single() calls, unless
         *   spi_message.is_dma_mapped reports a pre-existing mapping
         */
        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 /* 1bit transfer */
#define        SPI_NBITS_DUAL                0x02 /* 2bits transfer */
#define        SPI_NBITS_QUAD                0x04 /* 4bits transfer */
        u8                bits_per_word;
        u16                delay_usecs;
        struct spi_delay        delay;
        struct spi_delay        cs_change_delay;
        struct spi_delay        word_delay;
        u32                speed_hz;

        u32                effective_speed_hz;

        unsigned int        ptp_sts_word_pre;
        unsigned int        ptp_sts_word_post;

        struct ptp_system_timestamp *ptp_sts;

        bool                timestamped;

        struct list_head transfer_list;

#define SPI_TRANS_FAIL_NO_START        BIT(0)
        u16                error;
};
```

spi_transfer是spi消息传递的最小单位，在spi_transfer中，定义了这一个传输中所需的发送与接收的内存buffer（rx/tx_buf），速率，传输长度，和所需的delay等。其他的则是在启用dma功能时所需要的信息。

多个spi_transfer构成了spi_message，在注释中，spi_message被描述为一个完整的多片的spi传输。

spi_message

```
struct spi_message {
        struct list_head        transfers;

        struct spi_device        *spi;

        unsigned                is_dma_mapped:1;

        /* REVISIT:  we might want a flag affecting the behavior of the
         * last transfer ... allowing things like "read 16 bit length L"
         * immediately followed by "read L bytes".  Basically imposing
         * a specific message scheduling algorithm.
         *
         * Some controller drivers (message-at-a-time queue processing)
         * could provide that as their default scheduling algorithm.  But
         * others (with multi-message pipelines) could need a flag to
         * tell them about such special cases.
         */

        /* completion is reported through a callback */
        void                        (*complete)(void *context);
        void                        *context;
        unsigned                frame_length;
        unsigned                actual_length;
        int                        status;

        /* for optional use by whatever driver currently owns the
         * spi_message ...  between calls to spi_async and then later
         * complete(), that's the spi_controller controller driver.
         */
        struct list_head        queue;
        void                        *state;

        /* list of spi_res reources when the spi message is processed */
        struct list_head        resources;
};
```

其成员变量主要就由一个spi_transfer构成的list，和spi异步消息交换时需要的回调。

接下来我们关注一下spi消息交换的大致流程。

spi消息交换流程分析

我们只关注spi内核框架中关于消息交换相关的函数，首先最重要的就是spi控制器初始化时需要适配的底层最终的transfer函数，该函数共有三个变种，在最初的spi框架中只有transfer，而如今的spi控制器则较多的去实现了后面两个变种，transfer_one和transfer_one_message。

transfer_one函数指挥硬件完成一个spi_transfer消息的交换，而transfer_one_message可以指挥硬件完成一个spi_message的交换。

观察spi_register_controller函数中关于transfer钩子相关的部分：

spi_register_controller

```
	/*
	 * If we're using a queued driver, start the queue. Note that we don't
	 * need the queueing logic if the driver is only supporting high-level
	 * memory operations.
	 */
	if (ctlr->transfer) {
		dev_info(dev, "controller is unqueued, this is deprecated\n");
	} else if (ctlr->transfer_one || ctlr->transfer_one_message) {
		status = spi_controller_initialize_queue(ctlr);
		if (status) {
			device_del(&ctlr->dev);
			goto free_bus_id;
		}
	}
	/* add statistics */
```

可以看到，transfer作为老式的spi控制器如今只为了兼容老机器所使用，新的spi控制器都会选择实现transfer_one或者transfer_one_message（c2000中的spi控制器实现了transfer_one）。之后内核会初始化spi控制器中的队列，跟进该函数：

spi_controller_initialize_queue

```
static int spi_controller_initialize_queue(struct spi_controller *ctlr)
{
	int ret;

	ctlr->transfer = spi_queued_transfer; //
	if (!ctlr->transfer_one_message)
		ctlr->transfer_one_message = spi_transfer_one_message; //

	/* Initialize and start queue */
	ret = spi_init_queue(ctlr); //
	if (ret) {
		dev_err(&ctlr->dev, "problem initializing queue\n");
		goto err_init_queue;
	}
	ctlr->queued = true;
	ret = spi_start_queue(ctlr); //
	if (ret) {
		dev_err(&ctlr->dev, "problem starting queue\n");
		goto err_start_queue;
	}

	return 0;

err_start_queue:
	spi_destroy_queue(ctlr);
err_init_queue:
	return ret;
}
```

可以看到，该函数将transfer和transfer_one_message替换为了内核的函数，然后初始化并启动了该spi控制器的消息队列。

init&start queue

```
static int spi_init_queue(struct spi_controller *ctlr)
{
        ctlr->running = false;
        ctlr->busy = false;

        ctlr->kworker = kthread_create_worker(0, dev_name(&ctlr->dev)); // worker
        if (IS_ERR(ctlr->kworker)) {
                dev_err(&ctlr->dev, "failed to create message pump kworker\n");
                return PTR_ERR(ctlr->kworker);
        }

        kthread_init_work(&ctlr->pump_messages, spi_pump_messages); // workspi_pump_messagesspi

        /*
         * Controller config will indicate if this controller should run the
         * message pump with high (realtime) priority to reduce the transfer
         * latency on the bus by minimising the delay between a transfer
         * request and the scheduling of the message pump thread. Without this
         * setting the message pump thread will remain at default priority.
         */
        if (ctlr->rt)
                spi_set_thread_rt(ctlr);

        return 0;
}

static int spi_start_queue(struct spi_controller *ctlr)
{
        unsigned long flags;

        spin_lock_irqsave(&ctlr->queue_lock, flags);

        if (ctlr->running || ctlr->busy) {
                spin_unlock_irqrestore(&ctlr->queue_lock, flags);
                return -EBUSY;
        }

        ctlr->running = true;
        ctlr->cur_msg = NULL;
        spin_unlock_irqrestore(&ctlr->queue_lock, flags);

        kthread_queue_work(ctlr->kworker, &ctlr->pump_messages); //workerworkspi_pump_messages

        return 0;
}
```

记住spi_pump_messages这个函数。我们介绍过spi在linux内核中支持消息的同步（spi_sync）和异步（spi_async）传输，其差异在于同步传输会等到spi_message传输完成才返回，我们接下来看一看spi异步传输的机制，看下spi_pump_messages这个函数是如何在其中发挥作用的。

__spi_async

```
static int __spi_async(struct spi_device *spi, struct spi_message *message)
{
        struct spi_controller *ctlr = spi->controller;
        struct spi_transfer *xfer;

        /*
         * Some controllers do not support doing regular SPI transfers. Return
         * ENOTSUPP when this is the case.
         */
        if (!ctlr->transfer)
                return -ENOTSUPP;

        message->spi = spi; // messagespi

        SPI_STATISTICS_INCREMENT_FIELD(&ctlr->statistics, spi_async);
        SPI_STATISTICS_INCREMENT_FIELD(&spi->statistics, spi_async);

        trace_spi_message_submit(message);

        if (!ctlr->ptp_sts_supported) {
                list_for_each_entry(xfer, &message->transfers, transfer_list) {
                        xfer->ptp_sts_word_pre = 0;
                        ptp_read_system_prets(xfer->ptp_sts);
                }
        }

        return ctlr->transfer(spi, message); // controllertransferspi
}
```

__spi_queued_transfer

```
static int __spi_queued_transfer(struct spi_device *spi,
                                 struct spi_message *msg,
                                 bool need_pump)
{
        struct spi_controller *ctlr = spi->controller;
        unsigned long flags;

        spin_lock_irqsave(&ctlr->queue_lock, flags);

        if (!ctlr->running) {
                spin_unlock_irqrestore(&ctlr->queue_lock, flags);
                return -ESHUTDOWN;
        }
        msg->actual_length = 0;
        msg->status = -EINPROGRESS;

        list_add_tail(&msg->queue, &ctlr->queue); // controller
        if (!ctlr->busy && need_pump)
                kthread_queue_work(ctlr->kworker, &ctlr->pump_messages); // workerspi_pump_messages

        spin_unlock_irqrestore(&ctlr->queue_lock, flags);
        return 0;
}
```

可以看到，spi_message的传输最终都会在spi_pump_message中进行。spi_pump_message的函数整体比较长，大致的流程包括：

1. 检查当前是否有msg在传输

2. 检查当前controller是否为繁忙状态

3. 调用底层的prepare_message函数，进行一些传输前的设置

4. 如果开启了dma功能则进行一些相应的设置

4. 最后调用controller上的函数transfer_one_message，如果spi控制器没有实现该函数，则该函数在之前的流程中被设置为了

spi_transfer_one_message

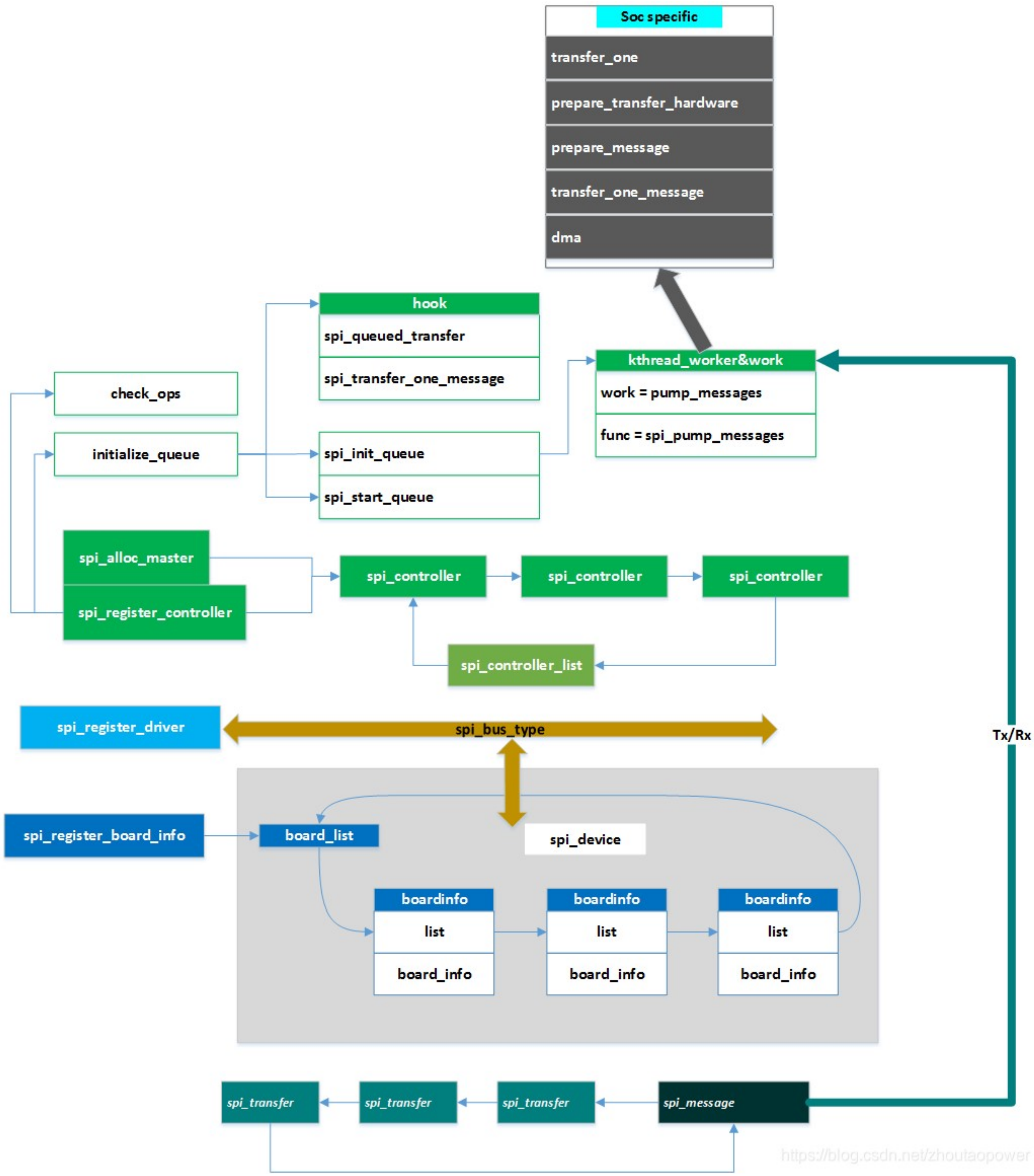
spi_transfer_one_message

```
static int spi_transfer_one_message(struct spi_controller *ctlr,
                                    struct spi_message *msg)
{
        struct spi_transfer *xfer;
        bool keep_cs = false;
        int ret = 0;
        struct spi_statistics *statm = &ctlr->statistics;
        struct spi_statistics *stats = &msg->spi->statistics;

        spi_set_cs(msg->spi, true, false);

        SPI_STATISTICS_INCREMENT_FIELD(statm, messages);
        SPI_STATISTICS_INCREMENT_FIELD(stats, messages);

        list_for_each_entry(xfer, &msg->transfers, transfer_list) { // msgtransferstransfermsg
                trace_spi_transfer_start(msg, xfer);

                spi_statistics_add_transfer_stats(statm, xfer, ctlr);
                spi_statistics_add_transfer_stats(stats, xfer, ctlr);

                if (!ctlr->ptp_sts_supported) {
                        xfer->ptp_sts_word_pre = 0;
                        ptp_read_system_prets(xfer->ptp_sts);
                }

                if ((xfer->tx_buf || xfer->rx_buf) && xfer->len) {
                        reinit_completion(&ctlr->xfer_completion);

fallback_pio:
                        ret = ctlr->transfer_one(ctlr, msg->spi, xfer); // transfer_one
                        if (ret < 0) {
                                if (ctlr->cur_msg_mapped &&
                                   (xfer->error & SPI_TRANS_FAIL_NO_START)) {
                                        __spi_unmap_msg(ctlr, msg);
                                        ctlr->fallback = true;
                                        xfer->error &= ~SPI_TRANS_FAIL_NO_START;
                                        goto fallback_pio;
                                }

                                SPI_STATISTICS_INCREMENT_FIELD(statm,
                                                               errors);
                                SPI_STATISTICS_INCREMENT_FIELD(stats,
                                                               errors);
                                dev_err(&msg->spi->dev,
                                        "SPI transfer failed: %d\n", ret);
                                goto out;
                        }

                        if (ret > 0) {
                                ret = spi_transfer_wait(ctlr, msg, xfer); // spi
                                if (ret < 0)
                                        msg->status = ret;
                        }
```

```
		} else {
			if (xfer->len)
				dev_err(&msg->spi->dev,
					"Bufferless transfer has length %u\n",
					xfer->len);
		}

		if (!ctlr->ptp_sts_supported) {
			ptp_read_system_postts(xfer->ptp_sts);
			xfer->ptp_sts_word_post = xfer->len;
		}

		trace_spi_transfer_stop(msg, xfer);

		if (msg->status != -EINPROGRESS)
			goto out;

		spi_transfer_delay_exec(xfer);

		if (xfer->cs_change) {
			if (list_is_last(&xfer->transfer_list,
					 &msg->transfers)) {
				keep_cs = true;
			} else {
				spi_set_cs(msg->spi, false, false);
				_spi_transfer_cs_change_delay(msg, xfer);
				spi_set_cs(msg->spi, true, false);
			}
		}

		msg->actual_length += xfer->len;
	}

out:
	if (ret != 0 || !keep_cs)
		spi_set_cs(msg->spi, false, false);

	if (msg->status == -EINPROGRESS)
		msg->status = ret;

	if (msg->status && ctlr->handle_err)
		ctlr->handle_err(ctlr, msg);

	spi_finalize_current_message(ctlr); // controller

	return ret;
}
```

至此，linux内核驱动控制spi控制器成功的完成了一次消息的交换。

linux内核中SPI驱动框架的大体结构如下：



Soc specific
transfer_one
prepare_transfer_hardware
prepare_message
transfer_one_message
dma
hook
spi_queued_transfer
spi_transfer_one_message
kthread_worker&work
work = pump_messages
func = spi_pump_messages
check_ops
initialize_queue
spi_init_queue
spi_start_queue
spi_alloc_master
spi_register_controller
spi_controller
spi_controller
spi_controller
spi_controller_list
spi_register_driver
spi_bus_type
Tx/Rx
spi_register_board_info
board_list
spi_device
boardinfo
list
board_info
boardinfo
list
board_info
boardinfo
list
board_info
spi_transfer
spi_transfer
spi_transfer
spi_message
https://blog.csdn.net/zhoutaopower