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

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

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

Declaring PXA2xx Master Controllers

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

```
struct pxa2xx_spi_master {
        enum pxa_ssp_type ssp_type;
        u32 clock_enable;
        u16 num_chipselect;
        u8 enable_dma;
};
```

The "pxa2xx_spi_master.ssp_type" field must have a value between 1 and 3 and
informs the driver which features a particular SSP supports.

The "pxa2xx_spi_master.clock_enable" field is used to enable/disable the
corresponding SSP peripheral block in the "Clock Enable Register (CKEN"). See
the "PXA2xx Developer Manual" section "Clocks and Power Management".

The "pxa2xx_spi_master.num_chipselect" field is used to determine the number of
slave device (chips) attached to this SPI master.

The "pxa2xx_spi_master.enable_dma" field informs the driver that SSP DMA should
be used. This caused the driver to acquire two DMA channels: rx_channel and
tx_channel. The rx_channel has a higher DMA service priority the tx_channel.
See the "PXA2xx Developer Manual" section "DMA Controller".

NSSP MASTER SAMPLE

Below is a sample configuration using the PXA255 NSSP.

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

```
                .start  = IRQ_NSSP, /* NSSP IRQ */
                .end    = IRQ_NSSP,
                .flags  = IORESOURCE_IRQ,
        },
};

static struct pxa2xx_spi_master pxa_nssp_master_info = {
        .ssp_type = PXA25x_NSSP, /* Type of SSP */
        .clock_enable = CKEN_NSSP, /* NSSP Peripheral clock */
        .num_chipselect = 1, /* Matches the number of chips attached to NSSP */
        .enable_dma = 1, /* Enables NSSP DMA */
};

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

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

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

Declaring Slave Devices

Typically each SPI slave (chip) is defined in the arch/.../mach-*/board-*.c using the "spi_board_info" structure found in "linux/spi/spi.h". See "Documentation/spi/spi_summary" for additional information.

Each slave device attached to the PXA must provide slave specific configuration information via the structure "pxa2xx_spi_chip" found in "arch/arm/mach-pxa/include/mach/pxa2xx_spi.h". The pxa2xx_spi master controller driver
will uses the configuration whenever the driver communicates with the slave device. All fields are optional.

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

The "pxa2xx_spi_chip.tx_threshold" and "pxa2xx_spi_chip.rx_threshold" fields are used to configure the SSP hardware fifo. These fields are critical to the

performance of pxa2xx_spi driver and misconfiguration will result in rx
fifo overruns (especially in PIO mode transfers). Good default values are

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

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

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

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

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

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

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

NSSP SLAVE SAMPLE

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

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

/* Chip Select control for the CS8405A SPI slave device */
static void cs8405a_cs_control(u32 command)
{
        if (command & PXA2XX_CS_ASSERT)
```

```
                GPCR(3) = GPIO_bit(3);
        else
                GPSR(3) = GPIO_bit(3);
}

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

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

static struct spi_board_info streetracer_spi_board_info[] __initdata = {
        {
                .modalias = "cs8415a", /* Name of spi_driver for this device */
                .max_speed_hz = 3686400, /* Run SSP as fast a possbile */
                .bus_num = 2, /* Framework bus number */
                .chip_select = 0, /* Framework chip select */
                .platform_data = NULL; /* No spi_driver specific config */
                .controller_data = &cs8415a_chip_info, /* Master chip config */
                .irq = STREETRACER_APCI_IRQ, /* Slave device interrupt */
        },
        {
                .modalias = "cs8405a", /* Name of spi_driver for this device */
                .max_speed_hz = 3686400, /* Run SSP as fast a possbile */
                .bus_num = 2, /* Framework bus number */
                .chip_select = 1, /* Framework chip select */
                .controller_data = &cs8405a_chip_info, /* Master chip config */
                .irq = STREETRACER_APCI_IRQ, /* Slave device interrupt */
        },
};

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

DMA and PIO I/O Support

The pxa2xx_spi driver supports both DMA and interrupt driven PIO message transfers. The driver defaults to PIO mode and DMA transfers must be enabled by setting the "enable_dma" flag in the "pxa2xx_spi_master" structure. The DMA mode supports both coherent and stream based DMA mappings.

The following logic is used to determine the type of I/O to be used on

a per "spi_transfer" basis:

```
if !enable_dma then
        always use PIO transfers

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

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

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

otherwise
        use PIO transfer
```

THANKS TO

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