# PXA2xx SPI on SSP driver HOWTO

This a mini HOWTO on the pxa2xx_spi driver. The driver turns a PXA2xx synchronous serial port into an SPI master controller (see Documentation/spi/spi-summary.rst). The driver has the following features

- Support for any PXA2xx and compatible 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 &struct spi_message FIFO serviced by kernel thread. The kernel thread, spi_pump_messages(), drives message FIFO and is responsible for queuing SPI transactions and setting up and launching the DMA or interrupt driven transfers.

## Declaring PXA2xx Master Controllers

Typically, for a legacy platform, an 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 include/linux/spi/pxa2xx_spi.h:

```
struct pxa2xx_spi_controller {
      u16 num_chipselect;
      u8 enable_dma;
      ...
};
```

The "pxa2xx_spi_controller.num_chipselect" field is used to determine the number of slave device (chips) attached to this SPI master.

The "pxa2xx_spi_controller.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 than the Tx channel. See the "PXA2xx Developer Manual" section "DMA Controller".

For the new platforms the description of the controller and peripheral devices comes from Device Tree or ACPI.

## NSSP MASTER SAMPLE

Below is a sample configuration using the PXA255 NSSP for a legacy platform:

```
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_controller pxa_nssp_master_info = {
      .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, for a legacy platform, 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.rst" for additional information.

Each slave device attached to the PXA must provide slave specific configuration information via the structure "pxa2xx_spi_chip" found in "include/linux/spi/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;
};
```

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.

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 a descriptor) to accommodate these chips.

## NSSP SLAVE SAMPLE

For a legacy platform or in some other cases, 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.

```
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 */
};

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 */
};

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_controller" structure. For the newer platforms, that are known to support DMA, the driver will enable it automatically and try it first with a possible fallback to PIO. 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.