September 12, 2014

# Chapter 1

# REVISION

## 1.1 CHANGES

| Version | Description of change | Date |
|---|---|---|
| 0.1 | First Draft | 2014/08/25 |

# Chapter 2

# Analog to Digital Convertor

## 2.1 USER SPACE ACCESS

The ADC device driver file /dev/adc0 is used to allow control of the Analog to Digital convertor (ADC). One can use standard echo and cat command on the driver to write or read from it.

## 2.2 DEVICE DRIVER

TBD.

## 2.3 KERNEL LOGS

The way to log the chip is to use minicom, connect to /dev/ttyUSB0(115200 8N1 No Flow Control) and run commands on it to see the printk messages. When the module is installed to the kernel the init(check) function is called. Inorder to install the module insmod is used.

```
# insmod adc.ko
adc_mod_init
```

When the module is read its corresponding read(check) function is called in the driver.

```
# cat /dev/adc0
adc_open 0
adc_read 0 file=a0311ca0, buf=a1c01000, count=4096, off=0
251
adc_read 0 file=a0311ca0, buf=a1c01000, count=4096, off=0
adc_close 0
```

When the module is written by some new value the write(check) function of the driver is called.

```
# echo 1 > /dev/adc0
adc_open 0
adc_close 0
```

Finally at the removal of the module the remove(check) function is called. The *rmmod* binary is used here.

```
# rmmod adc.ko
adc_mod_exit
```

# Chapter 3

# Serial Peripheral Interface

## 3.1 SYSFS

The SPI driver is not meant to be used from the userspace. There are other slave drivers that may interact with it to use it. Even then the sysfs entry is */sys/devices/platform/lpc2xxx-spi*.

```
lrwxrwxrwx 1 root 0 0 Aug 27 15:17 driver -> \
../../../bus/platform/drivers/lpc2xxx-spi
-r--r--r-- 1 root 0 4096 Aug 27 15:17 modalias
drwxr-xr-x 2 root 0 0 Aug 27 15:17 spi0.0
lrwxrwxrwx 1 root 0 0 Aug 27 15:17 subsystem -> \
../../../bus/platform
--w------- 1 root 0 4096 Aug 27 15:17 uevent
```

## 3.2 BOARD SUPPORT

Also called platform initialization, Linux wants to keep board support information in a separate place. In *arch/arm/mach-lpc22xx/lpc2468_ea_board.c* the SPI is setup as below

```
/* -------------------------------------
*  SPI
*------------------------------------- */
...
/* p0.16 is used as chip select pin for touch panel */
#define TP_CS_PIN 0x00010000
...
...
/* SPI devices */
static struct spi_board_info lpc2468_spi_board_info[] \
__initdata = {
  /* Touch Controller */
  {
   .
    .max_speed_hz   = 2500000,
  },
};
```

```
/* Handle chip-select for SPI bus */
static void lpc2468_spi_set_cs\
(struct lpc2xxx_spi_info* spi, int cs, int pol) {
  switch (cs) {
    case CS_TP:
      if (pol)
        IOSET0 = TP_CS_PIN;
      else
        IOCLR0 = TP_CS_PIN;
      break;
    default:
      printk("%s: ERROR: unknown cs=%d\n", \
      __FUNCTION__, cs);
  }
}

/*
* Board specific data used by SPI.
* The device file (e.g lpc24xx_devices.c)
* will use this data when registering the SPI device.
*/
struct lpc2xxx_spi_info lpc2xxx_spi_pdata = {
 .board_size = ARRAY_SIZE(lpc2468_spi_board_info),
 .board_info = lpc2468_spi_board_info,
 .set_cs     = lpc2468_spi_set_cs,
};
```

A structure containing information about the SPI device in lpc24xxx is setup with some board information structure and information to set CHIP SELECT. The board info specifies that the max speed of SPI interface is 2.5MHz. Inorder to set the CS, Pin 0.16 is set, which actually brings it low in hardware. CS_ is the actual signal.

```
static void __init lpc2468_ea_init_spi_pins(void)
{
#if defined(CONFIG_SPI_LPC2XXX)
...
  lpc22xx_set_periph(LPC22XX_PIN_P0_15, 3, 0);        // SCK
  lpc22xx_set_periph(LPC22XX_PIN_P0_16, 0, 0);        // GPIO, used as CS
  lpc22xx_set_periph(LPC22XX_PIN_P0_17, 3, 0);        // MISO
  lpc22xx_set_periph(LPC22XX_PIN_P0_18, 3, 0);        // MOSI
#endif
}
```

The remaining code configures, P0.15, 0.17 and 0.18 as SCK, MISO and MOSI respectively. Note this is a __init function is called once and is removed later on from memory. A UML diagram is given below with some of the initialization. The peripherals are initialized in a seperate file called arch/arm/mach-lpc22xx/lpc24xx_devices.c.

```
/* ---------------------------------------------------------------------
* SPI (Chapter 19) *
--------------------------------------------------------------------- */
/*
* lpc2xxx_spi_pdata must be created and initialized in the
```

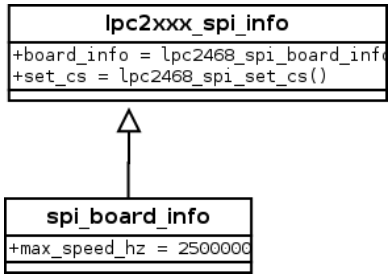


Figure 3.1: UML spi_info

```
* board specific file (see e.g. ea_lpc2478_board.c). */
extern struct lpc2xxx_spi_info lpc2xxx_spi_pdata;

#if defined(CONFIG_SPI_LPC2XXX)
static struct resource lpc2xxx_spi_resource[] = {
  {
    .name = "lpc2xxx-spi",
    .start = APB_SPI_BASE,
    .end = APB_SPI_BASE + APB_SPI_SIZE - 1,
    .flags = IORESOURCE_MEM,
  },
  {
    .name = "lpc2xxx-spi",
    .start = LPC2xxx_INTERRUPT_SPI0,
    .flags = IORESOURCE_IRQ,
  },
};

static struct platform_device lpc2xxx_spi_device = {
  .name = "lpc2xxx-spi",
  .id = -1,
  .num_resources = ARRAY_SIZE(lpc2xxx_spi_resource),
  .resource = lpc2xxx_spi_resource,
  .dev = {
    .platform_data = &lpc2xxx_spi_pdata,
  },
};

static void __init lpc2xxx_add_spi_device(void) {
  platform_device_register(&lpc2xxx_spi_device);
}
#else
static void __init lpc2xxx_add_spi_device(void) {}
#endif
```

Also note that some definitions of registers, structures etc are given in include/asm/arch-lpc22xx folder as well.

```
board.h        entry-macro.S  irq.h       lpc2xxx_pmc.h  serial.h  uncompress.h
clocks.h       gpio.h         irqs.h      memory.h       system.h
debug-macro.S  hardware.h     keyboard.h  nvram.h        time.h
dma.h          io.h           lpc22xx.h   param.h        timex.h
```

## 3.3 DRIVER INITIALIZATION

The static platform driver structure `lp2xx_spidrv` is initialized with a `spi_probe` function. The driver's name is set to `lpc2xxx-spi` which matches with the device's name. Because of that when the `platform_driver_register` is called with this driver structure, the binding is done between the driver and device. `lpc2xxx_spi_device` above is defined as the platform device with the same name

```
static struct platform_driver lpc2xxx_spidrv = {
        .probe                = lpc2xxx_spi_probe,
....
        .driver                = {
                .name        = "lpc2xxx-spi",
                .owner        = THIS_MODULE,
        },
};

static int __init lpc2xxx_spi_init(void)
{
        return platform_driver_register(&lpc2xxx_spidrv);
}

module_init(lpc2xxx_spi_init);
```

spi_probe starts by pointers to may different structures. Most notably spi_master and lpc2xxx_spi. The goal in the init routine is to set these to valid memory locations. The spi device acts as the spi-master. It starts by call to allocate memory for spi master. The size of the private data for the driver is given and the newly allocated memory pointer to a master struct is returned. The private data (lpc2xxx_spi) pointer is assigned a valid memory by spi_master_get_devdata. The memory is in the class private data. Now the private data has few member that are initialized to master, platform_data, platform_device. The private data consists of bitbang structure since lpc2468 supports that as of this writing. Its master, setup_transfer, chipselect, txrx_bufs, setup are setup appropriately. A UML diagram is given. A resource object is retrieved from the platform data. Memory, IO Area, and Interrupts are set for the private data. When the memory is requested, it actually creates a busy region with interlocks. Memory is remapped and the IRQ is requested to the kernel. When IRQ is requested, the IRQ line, the handler, etc.. are sent to the kernel. From that point on interrupts can be generated. So a service routine should be able to handle it. IO remap maps arbitrary physical memory to kernel virtual address space so that the kernel can execute code in external memory. The SPI control register is enabled for master mode and for interrupt generation when transfer is completed. Bitbang driver is started and any further requested are assumed to be handled by the driver. Finally a call is done to add the new spi device giving the master and board information object

In short, a spi_master obj is created. A private object that contains specific information about our driver is initialized. Memory, IRQ, hardware registers are setup. Finally the new device is added as a spi device to kernel core.

## 3.4 TRANSFER

SPI communication happens as transfers. Transfers are made up of 8 bit words or other sizes depending on the configuration. LPC24xx is configured to use the bit bang driver.

### 3.4.1 SETUP

First the driver is setup to do transfers and then `spi_bitbang_transfer()` function can be called to do the actual transfer. bitbang object is created in the spi_device private data. It's setup_transfer function is initialized to `lpc2xxx_spi_setupxfer` which does the real work. A user interacting driver code, such as one that of a SPI based ADC, will call `spi_bitbang_setup()` first which will dereference the function pointer for `bitbang->setup_transfer()` The dereference will result in `lpc2xxx_spi_setupxfer` being called. In this function, a prescalar is set up in SPI0 Control register and the chipselect set to be in INACTIVE mode. In LPC24xx the bits per word are 8.

| Object | Function Ptr | Value | Call |
|---|---|---|---|
| spi_lpc2xxx | setup_transfer() | lpc2xxx_spi_setupxfer() | spi_bitbang_setup |

### 3.4.2 TRANSMIT/RECEIPT

The actual transfer happens when the spi_bitbang_transfer() function is called. It deferences the function pointer txrx_bufs which results in lpc2xxx_spi_txrx being called. Of course txrx_bufs is initialized by the driver code to lpc2xxx_spi_txrx. Internally spi_bitbang_transfer function puts the transfer message in a kernel work queue and a separate function processes the queue. spin_locks are used during the work queue processing When the driver transfer function is called, the pointer to transmit buffer, the length of transfer etc are setup and writeb() is called to write the data byte by byte to the SPI0 data register. The routine waits for the completion of the transfer using wait queue. As in the init routine interrupt is generated when a transfer is completed. Upon completion it clears the wait queue so that another transaction can be started. The whole transfer happens till every data is completely transfered.

#### INTERRUPT SERVICE ROUTINE

The interrupt service routine is registered when a new spi_device is added as in the probe routine. Its called whenever a tranfer is completed. In it it reads the status register of SPI0, and checks for write collision or spi device not ready for transfer. it then increments the count and if needed reads byte from the data register, Work queue is cleared to start another transaction. ¡completion.h¿ describes how to work with wait queues.