目录

[1 触摸屏驱动 3](#_Toc530571677)

[1.1触摸屏介绍 3](#_Toc530571678)

[1.1.1触摸屏的基本原理是欧姆定律 3](#_Toc530571679)

[1.1.2 触摸屏结构 4](#_Toc530571680)

[1.1.3 触摸屏等效电路 4](#_Toc530571681)

[1.1.4 如何测量触摸屏坐标 4](#_Toc530571682)

[1.1.4 触摸屏驱动思路 5](#_Toc530571683)

[1.1.5 触摸屏ADC介绍 5](#_Toc530571684)

[1.2触摸屏驱动框架 5](#_Toc530571685)

[1.2.1 设备注册 5](#_Toc530571686)

[1.2.2 硬件相关设置 6](#_Toc530571687)

[1.2.4 触摸屏按下松开中断处理函数 8](#_Toc530571688)

[1.2.4 ADC中断处理函数中得到测量电压值。 9](#_Toc530571689)

[1.2.5 测量准确度改进 9](#_Toc530571690)

[1.3 触摸屏驱动完整程序 12](#_Toc530571691)

[1.4 触摸屏测试 17](#_Toc530571692)

[1.5 内核自带驱动 18](#_Toc530571693)

[1.5.1 设备注册 18](#_Toc530571694)

[1.5.2 设备驱动匹配方式采用id\_table[] 19](#_Toc530571695)

[2 LCD驱动程序 20](#_Toc530571696)

[2.1 lcd驱动程序框架 20](#_Toc530571697)

[2.2 LCD 硬件原理 23](#_Toc530571698)

[2.3 LCD 驱动程序编写 25](#_Toc530571699)

[2.3.1 lcd 初始化 25](#_Toc530571700)

[2.3.2 lcd参数设置 26](#_Toc530571701)

[2.4 lcd 驱动完整程序 33](#_Toc530571702)

[2.5 编译测试 39](#_Toc530571703)

[2.6 修改源码使用内核自带的lcd驱动 39](#_Toc530571704)

[3 I2C驱动程序 42](#_Toc530571705)

[3.1 i2c 总线协议原理 42](#_Toc530571706)

[3.1.1 i2c总线接口 42](#_Toc530571707)

[3.1.2 i2c总线信号传输时序 42](#_Toc530571708)

[3.1.3 i2c总线的数据传输格式 43](#_Toc530571709)

[3.2 i2c裸板程序 46](#_Toc530571710)

[3.2.1 i2c控制器初始化 46](#_Toc530571711)

[3.2.2 at24cxx读写 47](#_Toc530571712)

[3.2.3 i2c读写 47](#_Toc530571713)

[3.3 2.6内核下的i2c驱动程序 50](#_Toc530571714)

[3.3.1 i2c驱动框架 50](#_Toc530571715)

[3.3.2 i2c驱动编写 51](#_Toc530571716)

[3.4 Linux 3.4.2内核下的i2c驱动程序设计 55](#_Toc530571717)

[3.4.1 I2C驱动架构 55](#_Toc530571718)

[3.4.2 I2C clent的四种注册方法 57](#_Toc530571719)

[3.4.3 采用第二种方法编写完整的驱动 63](#_Toc530571720)

[3.4.4 使用内核自带驱动程序 68](#_Toc530571721)

[3.5 i2c总线驱动程序 69](#_Toc530571722)

[4 网卡驱动程序 71](#_Toc530571723)

[4.1 网卡驱动框架 71](#_Toc530571724)

[4.2 虚拟网卡驱动 72](#_Toc530571725)

[4.3 DM9K驱动移植 74](#_Toc530571726)

[4.4 网卡驱动测试 78](#_Toc530571727)

[5 usb驱动开发 79](#_Toc530571728)

[5.1 usb 基础知识 79](#_Toc530571729)

[5.2 usb 总线驱动程序 80](#_Toc530571730)

[5.3 编写一个USB设备驱动 参考usb\_mouse.c 81](#_Toc530571731)

[6 字符设备驱动程序 87](#_Toc530571732)

[6.1 如何找到设备的file\_operations结构体 87](#_Toc530571733)

[6.1.1 2.4内核中如何找到设备的file\_operations结构体 87](#_Toc530571734)

[6.1.2 2.6内核中如何找到设备的file\_operations结构体 87](#_Toc530571735)

[7 RTC 设备驱动 90](#_Toc530571736)

[7.1 RTC驱动程序框架 90](#_Toc530571737)

[8 块设备驱动 93](#_Toc530571738)

[8.1 为何块设备需要优化读写过程 93](#_Toc530571739)

[8.1.1 磁盘设备为何要优化读写过程？ 93](#_Toc530571740)

[8.1.2 flash设备为何要优化读写过程？ 93](#_Toc530571741)

[8.1.3 字符设备和块设备区别 94](#_Toc530571742)

[8.2 块设备驱动程序框架 94](#_Toc530571743)

[8.3 编写一个ramdisk驱动 95](#_Toc530571744)

[9 Nand Flash 驱动 99](#_Toc530571745)

[9.1 Nand 硬件原理 99](#_Toc530571746)

[9.2 nandfalsh 驱动框架 101](#_Toc530571747)

[9.2 nandfalsh 驱动程序 104](#_Toc530571748)

[10 NorFlash驱动 110](#_Toc530571749)

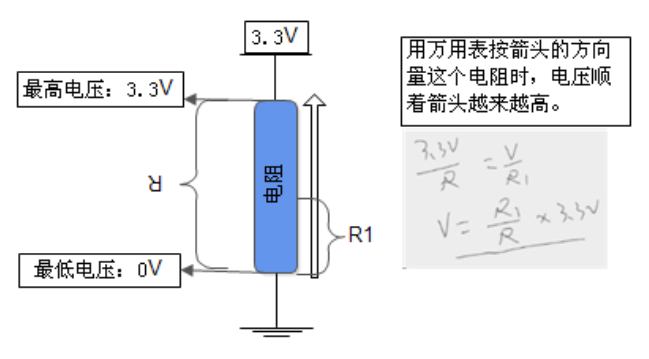
[10.1 Nor Flash硬件原理 110](#_Toc530571750)

[内存控制器 113](#_Toc530571751)

# 1 触摸屏驱动

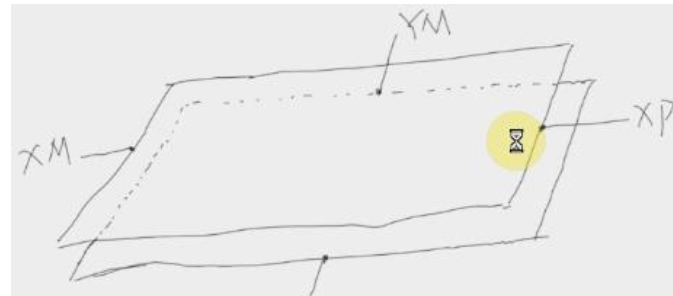
## 1.1触摸屏介绍

### 1.1.1触摸屏的基本原理是欧姆定律

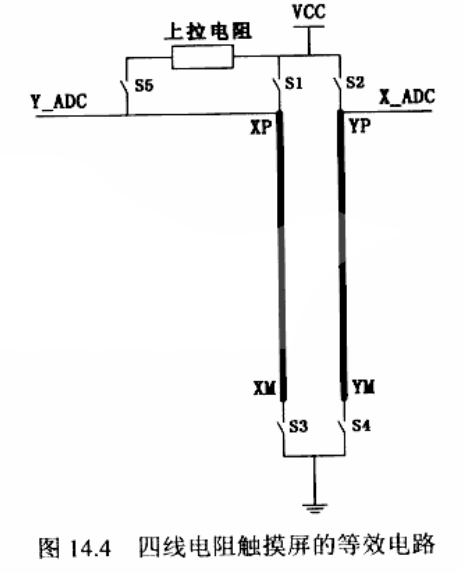


### 1.1.2 触摸屏结构

LCD 屏上有触摸屏，触摸屏就是 2 层很薄的膜。覆盖在 LCD 屏上面。 LCD 屏和触摸屏不同的器件，只是粘在一起了。触摸屏两层膜引出 4 条线。XM,YM,XP,YP。上面一层膜：XP 边（P 表示正极的意思）。XM 边（M 表示负极的意思）。下面一层膜：YP 边YM 边。平时这两层膜接触不到，当有触摸时，即按压时，上下两层膜就会粘在一起。



### 1.1.3 触摸屏等效电路



### 1.1.4 如何测量触摸屏坐标

测试 X 坐标：  
①， XP 接 3.3V  
②， XM 接地0V。  
③， YP 与 YM 都不接。这样上面一层相当于一个两端有3.3v电压的电阻了。  
④，按下触摸屏，上下两面接触。这个接触点设为x,YP就相当于从x引出的一根线。测试 YP 电压。就得到了x点的电压。

同理测试 Y 坐标：  
①， YP 接 3.3V  
②， YM 接地0V。  
③， XP 与 XM 都不接。  
④，测试 XP 电压

### 1.1.4 触摸屏驱动思路

按下触摸屏，产生一个外部中断。

在外部中断处理函数里面，启动ADC转换x,y坐标电压值。

转换完毕，产生ADC中断

在ADC中断处理函数里面，用input\_event()来上报。（这个过程有个缺点，就是按下后只会启动一次，按下不松开时“粘点”划动，这样就不会新“按下”中断产生，所以上报后启动定时器处理长按、滑动）。启动定时器。

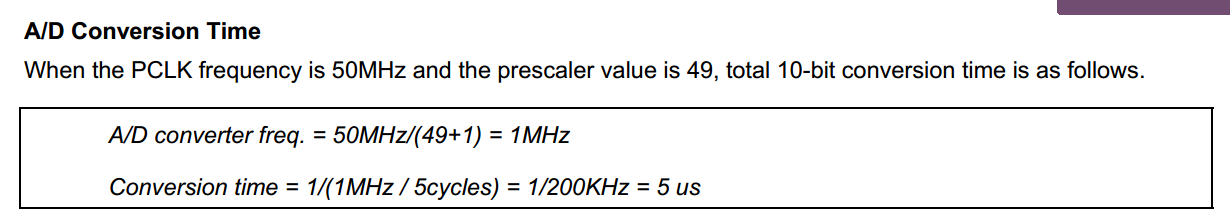
定时时间到，在定时器到期处理函数中再次启动ADC转换x,y坐标。转换完毕跳到执行，直到松开。

### 1.1.5 触摸屏ADC介绍

查看s3c2440触摸屏章节。发现接口是10bit ADC，最大频率2.5MHZ，最大电压3.3V。

ADC共8路输入，AIN4-AIN7对应YM,YP,XM,XP。

ADC转换时间



adc工作模式

a、Normal Conversion Mode //普通ADC转换

b、Separate X/Y position conversion Mode//x,y坐标分别转换，转换完x 就产生中断。

c、Auto(Sequential) X/Y Position Conversion Mode//x,y坐标转换完毕再产生中断。

d、Waiting for Interrupt Mode//等待触摸屏按下松开模式。触摸屏按下或者松开产生中断

## 1.2触摸屏驱动框架

### 1.2.1 设备注册

分配一个 input\_dev 结构体。

s3c\_ts\_dev = input\_allocate\_device();

设置,

能产生哪类事件;

set\_bit(EV\_KEY, s3c\_ts\_dev->evbit); //按键类事件

set\_bit(EV\_ABS, s3c\_ts\_dev->evbit);//绝对位移事件

b, 能产生这类事件里的哪些事件.,

set\_bit(BTN\_TOUCH, s3c\_ts\_dev->keybit);//按键类事件的触摸事件

c、 设置事件的参数

input\_set\_abs\_params(s3c\_ts\_dev, ABS\_X, 0, 0x3FF, 0, 0);

input\_set\_abs\_params(s3c\_ts\_dev, ABS\_Y, 0, 0x3FF, 0, 0);

input\_set\_abs\_params(s3c\_ts\_dev, ABS\_PRESSURE, 0, 1, 0, 0)

/\*中间两位代表最大值和最小值。3FF代表Adc最大值为1024，10bitAdc

压力中的0,1代表压力只有两个级别，按下，松开。最后两位不清楚是什么意思。\*/

注册输入设备

input\_register\_device(s3c\_ts\_dev);

### 1.2.2 硬件相关设置

触摸屏按下位置主要是通过ADC测量电压得到的。

打开ADC时钟。

clk = clk\_get(NULL, "adc");

clk\_enable(clk); // 设置CLKCON寄存器。

shift+ctrl+f \在linux-3.4.2\arch\arm 文件夹搜索"adc"，搜到s3c2410-clock.c (\plat-s3c24xx)

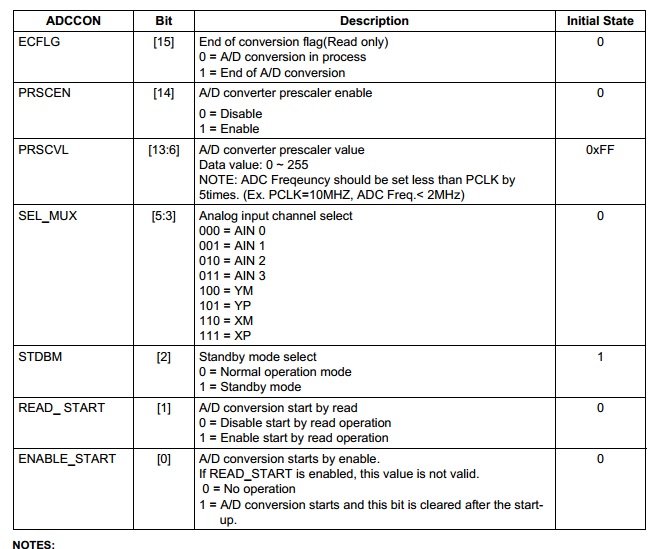
.name = "adc",

.parent = &clk\_p,

.enable = s3c2410\_clkcon\_enable, //使能时钟函数

.ctrlbit = S3C2410\_CLKCON\_ADC, //时钟控制器的哪一位

ADC 寄存器设置



**struct** s3c\_ts\_regs {

    unsigned **long** adccon;

    unsigned **long** adctsc;

    unsigned **long** adcdly;

    unsigned **long** adcdat0;

    unsigned **long** adcdat1;

    unsigned **long** adcupdn;

};

**static**  **volatile**  **struct** s3c\_ts\_regs \*s3c\_ts\_regs;

s3c\_ts\_regs = ioremap(0x58000000, **sizeof**(**struct** s3c\_ts\_regs));//

/\* bit[14]  : 1-A/D converter prescaler enable

     \* bit[13:6]: A/D converter prescaler value,

     \* 49, ADCCLK=PCLK/(49+1)=50MHz/(49+1)=1MHz

     \* bit[0]: A/D conversion starts by enable. 先设为0

    \*/

    s3c\_ts\_regs->adccon = (1<<14)|(49<<6);

注册触摸中断和ADC转换完成中断

request\_irq(IRQ\_TC, pen\_down\_up\_irq, IRQF\_SAMPLE\_RANDOM, "ts\_pen", NULL);

request\_irq(IRQ\_ADC, adc\_irq, IRQF\_SAMPLE\_RANDOM, "adc", NULL);

ADC进入等待中断模式(按下模式)

enter\_wait\_pen\_down\_mode();

//设置等待按下模式的函数

static void enter\_wait\_pen\_down\_mode(void)

{

s3c\_ts\_regs->adctsc = 0xd3;//1101

}

//设置等待松开模式的函数

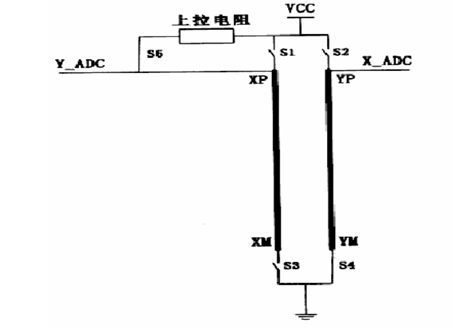
static void enter\_wait\_pen\_up\_mode(void)

{

s3c\_ts\_regs->adctsc = 0x1d3;

}

触摸屏等待按下中断模式的等效电路图



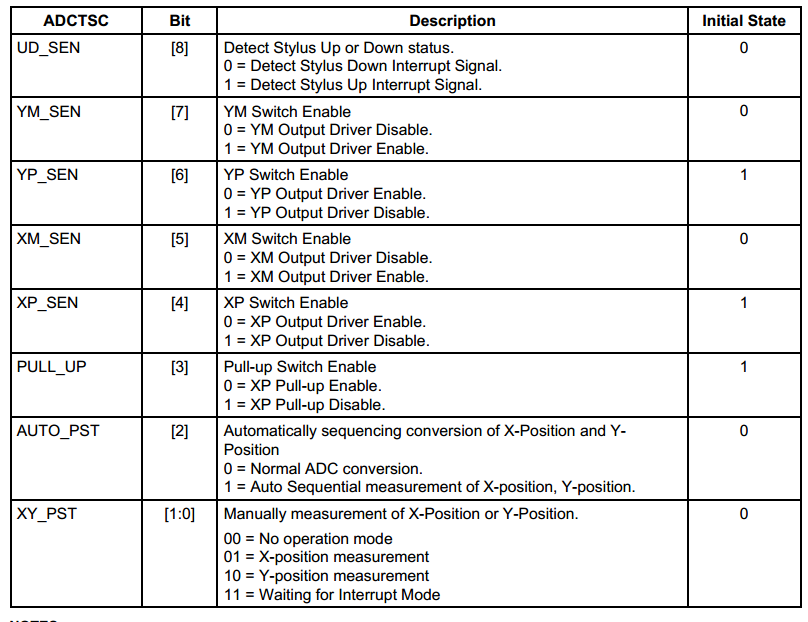
Y\_ADC初始电平为高，一旦触摸屏按下，那么Y\_ADC就会变为低电平。就会产生一个中断。

等待按下中断模式的开关是s1,s2,s3打开，s4关闭。

ADCTSC寄存器。

YM Output Driver Enable：意味着开关闭合。YM选Enable,其他都选择disable.

XP Pull-up Enable意味着连接到VCC



### 1.2.4 触摸屏按下松开中断处理函数

如果是触摸屏按下中断，触摸屏进入自动测量x,y坐标模式，开启ADC开始转换。

如果是触摸屏松开中断，上报触摸屏松开事件。

**static** irqreturn\_t pen\_down\_up\_irq(**int** irq, **void** \*dev\_id) //触摸屏松开

{

**if** (s3c\_ts\_regs->adcdat0 & (1<<15)) {

        input\_report\_abs(s3c\_ts\_dev, ABS\_PRESSURE, 0);

        input\_report\_key(s3c\_ts\_dev, BTN\_TOUCH, 0); // 上报触摸屏松开事件。

        input\_sync(s3c\_ts\_dev);

        enter\_wait\_pen\_down\_mode();   //等待触摸屏按下

    }

**else** //触摸屏按下

    {

        enter\_measure\_xy\_mode();//自动xy坐标测量模式

        start\_adc();

    }

**return** IRQ\_HANDLED;

}

}

**static** **void** enter\_measure\_xy\_mode(**void**) //自动测量xy模式,是否

{

    s3c\_ts\_regs->adctsc = (1<<3)|(1<<2); //是否需要手动禁止上拉电阻？

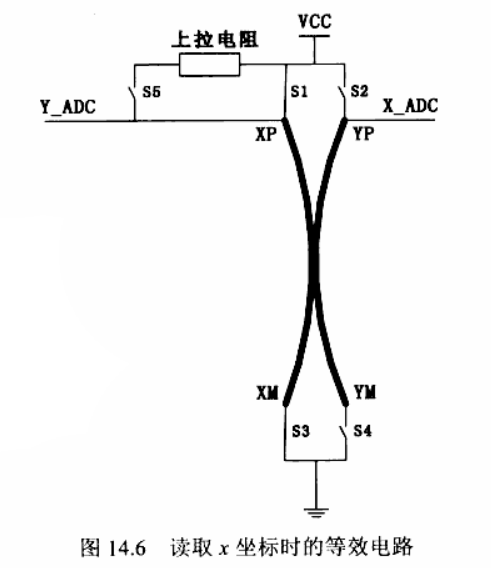
}

**static** **void** start\_adc(**void**)  //启动ADC

{

    s3c\_ts\_regs->adccon |= (1<<0);

}



### 1.2.4 ADC中断处理函数中得到测量电压值。

**static** irqreturn\_t adc\_irq(**int** irq, **void** \*dev\_id)

{

**static** **int** cnt = 0;

       printk("adc\_irq cnt = %d, x = %d, y = %d\n", ++cnt, s3c\_ts\_regs->adcdat0 & 0x3ff, s3c\_ts\_regs->adcdat1 & 0x3ff);

    enter\_wait\_pen\_up\_mode();//测量完毕，等待触摸屏松开

**return** IRQ\_HANDLED;

}

### 1.2.5 测量准确度改进

优化措施1: 设置ADCDLY为最大值, 这使得电压稳定后再发出IRQ\_TC中断，再开始测量电压值。

s3c\_ts\_regs->adcdly = 0xffff;

优化措施2: 如果ADC完成时, 发现触摸笔已经松开, 结果可能不准确，则丢弃此次结果

优化措施3，多次测量求平均值

**static** irqreturn\_t adc\_irq(**int** irq, **void** \*dev\_id)

{

**static** **int** cnt = 0;

**int** adcdat0, adcdat1;

    adcdat0 = s3c\_ts\_regs->adcdat0;

    adcdat1 = s3c\_ts\_regs->adcdat1;

**if** (s3c\_ts\_regs->adcdat0 & (1<<15)) /\* 已经松开，上报松开事件，不上报测量的值 \*/

    {

        input\_report\_abs(s3c\_ts\_dev, ABS\_PRESSURE, 0);

        input\_report\_key(s3c\_ts\_dev, BTN\_TOUCH, 0);

        input\_sync(s3c\_ts\_dev);

        enter\_wait\_pen\_down\_mode(); //等待按下

    }

**else** //触摸屏按下

{

printk("adc\_irq cnt = %d, x = %d, y = %d\n", ++cnt, s3c\_ts\_regs->adcdat0 & 0x3ff, s3c\_ts\_regs->adcdat1 & 0x3ff);

enter\_wait\_pen\_up\_mode();

    }

**return** IRQ\_HANDLED;

}

优化措施4，软件过滤

**static** **int** s3c\_filter\_ts(**int** x[], **int** y[])

{

#define ERR\_LIMIT 10

**int** avr\_x, avr\_y;

**int** det\_x, det\_y;

    avr\_x = (x[0] + x[1])/2;

    avr\_y = (y[0] + y[1])/2;

    det\_x = (x[2] > avr\_x) ? (x[2] - avr\_x) : (avr\_x - x[2]);

    det\_y = (y[2] > avr\_y) ? (y[2] - avr\_y) : (avr\_y - y[2]);

**if** ((det\_x > ERR\_LIMIT) || (det\_y > ERR\_LIMIT))

**return** 0;

    avr\_x = (x[1] + x[2])/2;

    avr\_y = (y[1] + y[2])/2;

    det\_x = (x[3] > avr\_x) ? (x[3] - avr\_x) : (avr\_x - x[3]);

    det\_y = (y[3] > avr\_y) ? (y[3] - avr\_y) : (avr\_y - y[3]);

**if** ((det\_x > ERR\_LIMIT) || (det\_y > ERR\_LIMIT))

**return** 0;

**return** 1; //测量正确

}

**static** irqreturn\_t adc\_irq(**int** irq, **void** \*dev\_id)

{

**static** **int** cnt = 0;

**static** **int** x[4], y[4];

**int** adcdat0, adcdat1;

    /\* 优化措施2: 如果ADC完成时, 发现触摸笔已经松开, 则丢弃此次结果 \*/

    adcdat0 = s3c\_ts\_regs->adcdat0;

    adcdat1 = s3c\_ts\_regs->adcdat1;

**if** (s3c\_ts\_regs->adcdat0 & (1<<15))

    {

        /\* 已经松开 \*/

        cnt = 0;

        input\_report\_abs(s3c\_ts\_dev, ABS\_PRESSURE, 0);

        input\_report\_key(s3c\_ts\_dev, BTN\_TOUCH, 0);

        input\_sync(s3c\_ts\_dev);

        enter\_wait\_pen\_down\_mode();

    }

**else**

    {

        // printk("adc\_irq cnt = %d, x = %d, y = %d\n", ++cnt, adcdat0 & 0x3ff, adcdat1 & 0x3ff);

        /\* 优化措施3: 多次测量求平均值 \*/

        x[cnt] = adcdat0 & 0x3ff;

        y[cnt] = adcdat1 & 0x3ff;

        ++cnt;

**if** (cnt == 4)

        {

            /\* 优化措施4: 软件过滤 \*/

**if** (s3c\_filter\_ts(x, y))

            {

                input\_report\_abs(s3c\_ts\_dev, ABS\_X, (x[0]+x[1]+x[2]+x[3])/4);

                input\_report\_abs(s3c\_ts\_dev, ABS\_Y, (y[0]+y[1]+y[2]+y[3])/4);

                input\_report\_abs(s3c\_ts\_dev, ABS\_PRESSURE, 1);

                input\_report\_key(s3c\_ts\_dev, BTN\_TOUCH, 1);

                input\_sync(s3c\_ts\_dev);

            }

            cnt = 0;

            enter\_wait\_pen\_up\_mode();

            /\* 启动定时器处理长按/滑动的情况 \*/

            mod\_timer(&ts\_timer, jiffies + HZ/100);

        }

**else**

        {

            enter\_measure\_xy\_mode();

            start\_adc();

        }

    }

**return** IRQ\_HANDLED;

}

优化措施5，使用定时器处理长按，滑动。

**static** **void** s3c\_ts\_timer\_function(unsigned **long** data)

{

**if** (s3c\_ts\_regs->adcdat0 & (1<<15))

    {

        /\* 已经松开 \*/

        input\_report\_abs(s3c\_ts\_dev, ABS\_PRESSURE, 0);

        input\_report\_key(s3c\_ts\_dev, BTN\_TOUCH, 0);

        input\_sync(s3c\_ts\_dev);

        enter\_wait\_pen\_down\_mode();

    }

**else**

    {

        /\* 测量X/Y坐标 \*/

        enter\_measure\_xy\_mode();

        start\_adc();

    }

}

## 1.3 触摸屏驱动完整程序

#include <linux/errno.h>

#include <linux/kernel.h>

#include <linux/module.h>

#include <linux/slab.h>

#include <linux/input.h>

#include <linux/init.h>

#include <linux/serio.h>

#include <linux/delay.h>

#include <linux/platform\_device.h>

#include <linux/clk.h>

#include <asm/io.h>

#include <asm/irq.h>

**struct** s3c\_ts\_regs {

    unsigned **long** adccon;

    unsigned **long** adctsc;

    unsigned **long** adcdly;

    unsigned **long** adcdat0;

    unsigned **long** adcdat1;

    unsigned **long** adcupdn;

};

**static** **struct** input\_dev \*s3c\_ts\_dev;

**static** **volatile** **struct** s3c\_ts\_regs \*s3c\_ts\_regs;

**static** **struct** timer\_list ts\_timer;

**static** **void** enter\_wait\_pen\_down\_mode(**void**)

{

    s3c\_ts\_regs->adctsc = 0xd3;

}

**static** **void** enter\_wait\_pen\_up\_mode(**void**)

{

    s3c\_ts\_regs->adctsc = 0x1d3;

}

**static** **void** enter\_measure\_xy\_mode(**void**)

{

    s3c\_ts\_regs->adctsc = (1<<3)|(1<<2);

}

**static** **void** start\_adc(**void**)

{

    s3c\_ts\_regs->adccon |= (1<<0);

}

#define ERR\_LIMIT 10

**int** avr\_x, avr\_y;

**int** det\_x, det\_y;

    avr\_x = (x[0] + x[1])/2;

    avr\_y = (y[0] + y[1])/2;

    det\_x = (x[2] > avr\_x) ? (x[2] - avr\_x) : (avr\_x - x[2]);

    det\_y = (y[2] > avr\_y) ? (y[2] - avr\_y) : (avr\_y - y[2]);

**if** ((det\_x > ERR\_LIMIT) || (det\_y > ERR\_LIMIT))

**return** 0;

    avr\_x = (x[1] + x[2])/2;

    avr\_y = (y[1] + y[2])/2;

    det\_x = (x[3] > avr\_x) ? (x[3] - avr\_x) : (avr\_x - x[3]);

    det\_y = (y[3] > avr\_y) ? (y[3] - avr\_y) : (avr\_y - y[3]);

**if** ((det\_x > ERR\_LIMIT) || (det\_y > ERR\_LIMIT))

**return** 0;

**return** 1;

**static** **void** s3c\_ts\_timer\_function(unsigned **long** data)

{

**if** (s3c\_ts\_regs->adcdat0 & (1<<15))

    {

        /\* 已经松开 \*/

        input\_report\_abs(s3c\_ts\_dev, ABS\_PRESSURE, 0);

        input\_report\_key(s3c\_ts\_dev, BTN\_TOUCH, 0);

input\_sync(s3c\_ts\_dev);

        enter\_wait\_pen\_down\_mode();

    }

**else**

    {

        /\* 测量X/Y坐标 \*/

        enter\_measure\_xy\_mode();

        start\_adc();

    }

}

**static** irqreturn\_t pen\_down\_up\_irq(**int** irq, **void** \*dev\_id)

{

**if** (s3c\_ts\_regs->adcdat0 & (1<<15))

    {

        //printk("pen up\n");

        input\_report\_abs(s3c\_ts\_dev, ABS\_PRESSURE, 0);

        input\_report\_key(s3c\_ts\_dev, BTN\_TOUCH, 0);

        input\_sync(s3c\_ts\_dev);

        enter\_wait\_pen\_down\_mode();

    }

**else**

    {

        //printk("pen down\n");

        //enter\_wait\_pen\_up\_mode();

        enter\_measure\_xy\_mode();

        start\_adc();

    }

**return** IRQ\_HANDLED;

}

**static** irqreturn\_t adc\_irq(**int** irq, **void** \*dev\_id)

{

**static** **int** cnt = 0;

**static** **int** x[4], y[4];

**int** adcdat0, adcdat1;

    /\* 优化措施2: 如果ADC完成时, 发现触摸笔已经松开, 则丢弃此次结果 \*/

    adcdat0 = s3c\_ts\_regs->adcdat0;

    adcdat1 = s3c\_ts\_regs->adcdat1;

**if** (s3c\_ts\_regs->adcdat0 & (1<<15))

 {

        /\* 已经松开 \*/

        cnt = 0;

        input\_report\_abs(s3c\_ts\_dev, ABS\_PRESSURE, 0);

        input\_report\_key(s3c\_ts\_dev, BTN\_TOUCH, 0);

        input\_sync(s3c\_ts\_dev);

        enter\_wait\_pen\_down\_mode();

    }

**else**

    {

        // printk("adc\_irq cnt = %d, x = %d, y = %d\n", ++cnt, adcdat0 & 0x3ff, adcdat1 & 0x3ff);

        /\* 优化措施3: 多次测量求平均值 \*/

        x[cnt] = adcdat0 & 0x3ff;

        y[cnt] = adcdat1 & 0x3ff;

        ++cnt;

**if** (cnt == 4)

        {

            /\* 优化措施4: 软件过滤 \*/

**if** (s3c\_filter\_ts(x, y))

            {

                //printk("x = %d, y = %d\n", (x[0]+x[1]+x[2]+x[3])/4, (y[0]+y[1]+y[2]+y[3])/4);

                input\_report\_abs(s3c\_ts\_dev, ABS\_X, (x[0]+x[1]+x[2]+x[3])/4);

                input\_report\_abs(s3c\_ts\_dev, ABS\_Y, (y[0]+y[1]+y[2]+y[3])/4);

                input\_report\_abs(s3c\_ts\_dev, ABS\_PRESSURE, 1);

                input\_report\_key(s3c\_ts\_dev, BTN\_TOUCH, 1);

                input\_sync(s3c\_ts\_dev);

            }

            cnt = 0;

            enter\_wait\_pen\_up\_mode();

            /\* 启动定时器处理长按/滑动的情况 \*/

            mod\_timer(&ts\_timer, jiffies + HZ/100);

        }

**else**

        {

            enter\_measure\_xy\_mode();

            start\_adc();

        }

    }

**return** IRQ\_HANDLED;

}

**static** **int** s3c\_ts\_init(**void**)

{

**struct** clk\* clk;

/\* 1. 分配一个input\_dev结构体 \*/

    s3c\_ts\_dev = input\_allocate\_device();

    /\* 2. 设置 \*/

    s3c\_ts\_dev->name="s3c\_ts";

    /\* 2.1 能产生哪类事件 \*/

    set\_bit(EV\_KEY, s3c\_ts\_dev->evbit);

    set\_bit(EV\_ABS, s3c\_ts\_dev->evbit);

    /\* 2.2 能产生这类事件里的哪些事件 \*/

    set\_bit(BTN\_TOUCH, s3c\_ts\_dev->keybit);

/\*中间两位代表最大值和最小值。

        3FF代表Adc最大值为1024，10bitAdc

        压力中的0,1代表压力只有两个级别，按下，松开。

        最后两位不清楚是什么意思    \*/

    input\_set\_abs\_params(s3c\_ts\_dev, ABS\_X, 0, 0x3FF, 0, 0);

    input\_set\_abs\_params(s3c\_ts\_dev, ABS\_Y, 0, 0x3FF, 0, 0);

    input\_set\_abs\_params(s3c\_ts\_dev, ABS\_PRESSURE, 0, 1, 0, 0);

    /\* 3. 注册 \*/

    input\_register\_device(s3c\_ts\_dev);

    /\* 4. 硬件相关的操作 \*/

    /\* 4.1 使能时钟(CLKCON[15]) \*/

    clk = clk\_get(NULL, "adc");

    clk\_enable(clk);

    /\* 4.2 设置S3C2440的ADC/TS寄存器 \*/

    s3c\_ts\_regs = ioremap(0x58000000, **sizeof**(**struct** s3c\_ts\_regs));

    /\* bit[14]  : 1-A/D converter prescaler enable

     \* bit[13:6]: A/D converter prescaler value,

     \*            49, ADCCLK=PCLK/(49+1)=50MHz/(49+1)=1MHz

     \* bit[0]: A/D conversion starts by enable. 先设为0

     \*/

    s3c\_ts\_regs->adccon = (1<<14)|(49<<6);

    request\_irq(IRQ\_TC, pen\_down\_up\_irq, IRQF\_SAMPLE\_RANDOM, "ts\_pen", NULL);

    request\_irq(IRQ\_ADC, adc\_irq, IRQF\_SAMPLE\_RANDOM, "adc", NULL);

 /\* 优化措施1:

     \* 设置ADCDLY为最大值, 这使得电压稳定后再发出IRQ\_TC中断

     \*/

    s3c\_ts\_regs->adcdly = 0xffff;

    /\* 优化措施5: 使用定时器处理长按,滑动的情况

     \*

     \*/

    init\_timer(&ts\_timer);

    ts\_timer.function = s3c\_ts\_timer\_function;

    add\_timer(&ts\_timer);

    enter\_wait\_pen\_down\_mode();

**return** 0;

}

**static** **void** s3c\_ts\_exit(**void**)

{

    free\_irq(IRQ\_TC, NULL);

    free\_irq(IRQ\_ADC, NULL);

    iounmap(s3c\_ts\_regs);

    input\_unregister\_device(s3c\_ts\_dev);

    input\_free\_device(s3c\_ts\_dev);

    del\_timer(&ts\_timer);

}

module\_init(s3c\_ts\_init);

module\_exit(s3c\_ts\_exit);

MODULE\_LICENSE("GPL");

## 1.4 触摸屏测试

1. make menuconfig 去掉原来的触摸屏驱动程序

-> Device Drivers

-> Input device support

-> Generic input layer

-> Touchscreens

<> S3C2410/S3C2440 touchscreens

make uImage

使用新内核启动

2. insmod s3c\_ts.ko

按下/松开触摸笔

测试2th~7th：

1. ls /dev/event\*

2. insmod s3c\_ts.ko

3. ls /dev/event\*

4. hexdump /dev/event0

0003类型代表ABS事件，0001按键事件，0000同步事件，

0000 code 代表x轴，0001代表y轴，0018代表压力，014a ,BTN\_TOUCH

value是电压值数字化后的值

秒 微秒 type code value

0000000 29a4 0000 8625 0008 0003 0000 0172 0000

0000010 29a4 0000 8631 0008 0003 0001 027c 0000

0000020 29a4 0000 8634 0008 0003 0018 0001 0000

0000030 29a4 0000 8638 0008 0001 014a 0001 0000

0000040 29a4 0000 863c 0008 0000 0000 0000 0000

0000050 29a4 0000 c85e 0008 0003 0000 0171 0000

0000060 29a4 0000 c874 0008 0003 0001 027d 0000

0000070 29a4 0000 c87b 0008 0000 0000 0000 0000

0000080 29a4 0000 ed37 0008 0003 0018 0000 0000

0000090 29a4 0000 ed48 0008 0001 014a 0000 0000

00000a0 29a4 0000 ed4a 0008 0000 0000 0000 0000

最终测试：

使用tslib测试。tslib再3.4.2驱动移植和QT移植里面想想的记录了如何使用。此处略过。通过tslib测试可以得到点阵坐标值。

## 1.5 内核自带驱动

三星公司的触摸屏驱动程序是：s3c2410\_ts.c。设备名字为"samsung-ts"。

MODULE\_DEVICE\_TABLE(platform, s3cts\_driver\_ids);

### 1.5.1 设备注册

**static** **struct** resource s3c\_ts\_resource[] = {

    [0] = DEFINE\_RES\_MEM(S3C24XX\_PA\_ADC, S3C24XX\_SZ\_ADC),

    [1] = DEFINE\_RES\_IRQ(IRQ\_TC),

};

**struct** platform\_device s3c\_device\_ts = {

    .name       = "s3c2410-ts",

    .id     = -1,

    .dev.parent = &s3c\_device\_adc.dev,

    .num\_resources  = ARRAY\_SIZE(s3c\_ts\_resource),

    .resource   = s3c\_ts\_resource,

};

**static** **struct** s3c2410\_ts\_mach\_info s3c2410\_ts\_cfg = {

        .delay = 10000,

        .presc = 49,

        .oversampling\_shift = 2,

};

// s3c24xx\_ts\_set\_platdata(&s3c2410\_ts\_cfg) 必须有这些信息才能使用内核自带的触摸屏驱//动程序

**void** \_\_init s3c24xx\_ts\_set\_platdata(**struct** s3c2410\_ts\_mach\_info \*hard\_s3c2410ts\_info)

{

    s3c\_set\_platdata(hard\_s3c2410ts\_info,

**sizeof**(**struct** s3c2410\_ts\_mach\_info), &s3c\_device\_ts);

}

**void** \_\_init \*s3c\_set\_platdata(**void** \*pd, **size\_t** pdsize,

**struct** platform\_device \*pdev)

{

**if** (!pd) {

**return** NULL;

    }

npd = kmemdup(pd, pdsize, GFP\_KERNEL);

    pdev->dev.platform\_data = npd;

}

### 1.5.2 设备驱动匹配方式采用id\_table[]

**static** **struct** platform\_device\_id s3cts\_driver\_ids[] = {

    { "s3c2410-ts", 0 },

    { "s3c2440-ts", 0 },

    { "s3c64xx-ts", FEAT\_PEN\_IRQ },

    { }

};

 //内核中有同名的设备注册时，会调用驱动的probe函数。

**static** **struct** platform\_driver s3c\_ts\_driver = {

    .driver         = {

        .name   = "samsung-ts",

        .owner  = THIS\_MODULE,

#ifdef CONFIG\_PM

        .pm = &s3c\_ts\_pmops,

#endif

    },

    .id\_table   = s3cts\_driver\_ids,

    .probe      = s3c2410ts\_probe,

    .**remove**     = \_\_devexit\_p(s3c2410ts\_remove),

};

1.5.3 内核自带的probe函数分析

**static** **int** \_\_devinit s3c2410ts\_probe(**struct** platform\_device \*pdev)// pdev=s3c\_device\_ts

{

**struct** s3c2410\_ts\_mach\_info \*info;//触摸屏平台数据

**struct** device \*dev = &pdev->dev; //驱动和设备结构体建立联系

**struct** input\_dev \*input\_dev;

**struct** resource \*res;

**int** ret = -EINVAL;

    /\* Initialise input stuff \*/

**memset**(&ts, 0, **sizeof**(**struct** s3c2410ts));//ts就是触摸屏设备

    ts.dev = dev; //触摸屏驱动设备结构体指针指向设备

/\*

    s3c2410\_ts\_mach\_info就是指向platform\_data;

    \*/

    info = pdev->dev.platform\_data; //指向为 platform\_data分配的空间

    ts.**clock** = clk\_get(dev, "adc");

    clk\_enable(ts.**clock**);

    ts.irq\_tc = ret = platform\_get\_irq(pdev, 0);

    res = platform\_get\_resource(pdev, IORESOURCE\_MEM, 0);

    ts.io = ioremap(res->start, resource\_size(res));

    /\* inititalise the gpio \*/

**if** (info->cfg\_gpio)

        info->cfg\_gpio(to\_platform\_device(ts.dev));

    ts.client = s3c\_adc\_register(pdev, s3c24xx\_ts\_select,

                     s3c24xx\_ts\_conversion, 1);

    /\* Initialise registers \*/

**if** ((info->delay & 0xffff) > 0)

        writel(info->delay & 0xffff, ts.io + S3C2410\_ADCDLY);

        writel(WAIT4INT | INT\_DOWN, ts.io + S3C2410\_ADCTSC);//等待中断模式

    input\_dev = input\_allocate\_device();

    ts.input = input\_dev;

    ts.input->evbit[0] = BIT\_MASK(EV\_KEY) | BIT\_MASK(EV\_ABS);

    ts.input->keybit[BIT\_WORD(BTN\_TOUCH)] = BIT\_MASK(BTN\_TOUCH);

    input\_set\_abs\_params(ts.input, ABS\_X, 0, 0x3FF, 0, 0);

    input\_set\_abs\_params(ts.input, ABS\_Y, 0, 0x3FF, 0, 0);

    ts.input->name = "S3C24XX TouchScreen";

    ts.input->id.bustype = BUS\_HOST;

    ts.input->id.vendor = 0xDEAD;

    ts.input->id.product = 0xBEEF;

    ts.input->id.version = 0x0102;

    ts.shift = info->oversampling\_shift;

    ts.features = platform\_get\_device\_id(pdev)->driver\_data;

    ret = request\_irq(ts.irq\_tc, stylus\_irq, 0,

              "s3c2410\_ts\_pen", ts.input);

        ret = input\_register\_device(ts.input);

}

# 2 LCD驱动程序

## 2.1 lcd驱动程序框架

APP open read write /dev/fb0 主设备号29 次设备号0

--------------------------------------------------------

driver drv\_open drv\_read drv\_write

------------------------------------------------

硬件

fbmem.c 内核中LCD核心层函数

fbmem\_init(**void**)

{

/\*注册主设备号，但并未添加设备\*/

     register\_chrdev(FB\_MAJOR,"fb",&fb\_fops); //主设备号29

    fb\_class = class\_create(THIS\_MODULE, "graphics");    /\*创建一个设备类结构\*/

}

这里 fbmem.c 没有在设备类下创建设备，只有真正有硬件设备时才有必要在这个类下去创建设备。

//这个结构体是所有LCD和上层应用程序的接口，

**static** **const** **struct** file\_operations fb\_fops = {

.read =        fb\_read,

    .write =       fb\_write,

    .ioctl =       fb\_ioctl,

    .mmap =        fb\_mmap,

    .open =        fb\_open,

    .release =    fb\_release,

}

**static** **int** fb\_open(**struct** inode \*inode, **struct** file \*file)

{

**int** fbidx = iminor(inode);//根据次设备号计算设备操作函数索引

**struct** fb\_info \*info;

**int** res = 0;

info = registered\_fb[fbidx])) //register\_framebuffer中得到了registered\_fb[fbidx]

**if** (info->fbops->fb\_open) {

        res = info->fbops->fb\_open(info,1);

//调用设备真正的open函数，并没有设置open函数，不需要。

    }

}

**static** ssize\_t fb\_read(**struct** file \*file, **char** \_\_user \*buf, **size\_t** count, loff\_t \*ppos)

{

    unsigned **long** p = \*ppos;

**struct** inode \*inode = file->f\_path.dentry->d\_inode;

**int** fbidx = iminor(inode);//设次设备号为0

**struct** fb\_info \*info = registered\_fb[fbidx];

    u32 \*buffer, \*dst;

    u32 \_\_iomem \*src;

**int** c, i, cnt = 0, err = 0;

    unsigned **long** total\_size;

**int** fbidx = iminor(inode);//设次设备号为0

**struct** fb\_info \*info = registered\_fb[fbidx];//找到fops指针

**if** (info->fbops->fb\_read)//如果有read函数。

**return** info->fbops->fb\_read(info, buf, count, ppos)；

        //没有就使用使用下面的读函数

buffer = kmalloc((count > PAGE\_SIZE) ? PAGE\_SIZE : count, GFP\_KERNEL);

src = (u32 \_\_iomem \*) (info->screen\_base + p);//info->screen\_base显存基址，P是偏移

{ //读内存复制到用户空间

**if** (copy\_to\_user(buf, buffer, c)) //复制到用户空间

     }

}

//和用户空间的信息交互是由fbmem.c中的fb\_ioctl()作为接口的

**static** **int**

fb\_ioctl(**struct** inode \*inode, **struct** file \*file, unsigned **int** cmd,

     unsigned **long** arg)

{

**switch** (cmd) {

**case** FBIOGET\_VSCREENINFO: //获取LCD可变参数信息

**return** copy\_to\_user(argp, &info->var,

**sizeof**(var)) ? -EFAULT : 0;

**case** FBIOPUT\_VSCREENINFO: //设置LCD可变参数信息

**if** (copy\_from\_user(&var, argp, **sizeof**(var)))

**return** -EFAULT;

}

在哪里设置了registered\_fb[i]？

/\*注册一个帧缓冲区,在注册LCD设备的时候使用

在s3c2410fb.c中的prob函数中调用。也就是注册LCD设备时调用

 \*/

**int**  register\_framebuffer(**struct** fb\_info \*fb\_info)

{

**int** i;

**struct** fb\_event event;

**struct** fb\_videomode mode;

**if** (num\_registered\_fb == FB\_MAX)//支持的lcd最大个数。

**return** -ENXIO;

    num\_registered\_fb++;

**for** (i = 0 ; i < FB\_MAX; i++) // registered\_fb[i]找到一个空闲的位置存储新LCD的fopss

**if** (!registered\_fb[i])

**break**;

    fb\_info->node = i;

    fb\_info->dev = device\_create(fb\_class, fb\_info->device,

                     MKDEV(FB\_MAJOR, i), "fb%d", i);//创建Lcd设备，设备名为fb%d;

    registered\_fb[i] = fb\_info;//此处得到了registered\_fb[i]

}

s3c2410fb.c

**static** **struct** platform\_driver s3c2410fb\_driver = {

    .probe      = s3c2410fb\_probe,

    .driver     = {

        .name   = "s3c2410-lcd",

            },

};

//当有同名的"s3c2410-lcd"设备注册时，调用probe函数。

**static** **int** \_\_init s3c2410fb\_probe(**struct** platform\_device \*pdev)

{

    分配

    fbinfo = framebuffer\_alloc(**sizeof**(**struct** s3c2410fb\_info), &pdev->dev);

    设置fbinfo

    fbinfo->fbops            = &s3c2410fb\_ops;

         注册

ret = register\_framebuffer(fbinfo);

硬件相关

}

**static** **struct** fb\_ops  s3c2410fb\_ops = {

    .owner      = THIS\_MODULE,

    .fb\_check\_var   = s3c2410fb\_check\_var,

    .fb\_set\_par = s3c2410fb\_set\_par,

    .fb\_blank   = s3c2410fb\_blank,

    .fb\_setcolreg   = s3c2410fb\_setcolreg,

    .fb\_fillrect    = cfb\_fillrect,

    .fb\_copyarea    = cfb\_copyarea,

    .fb\_imageblit   = cfb\_imageblit,

};

## 2.2 LCD 硬件原理

将LCD抽象为像素矩阵。每个vclk描绘一个点。从左到右，从上到下显示。

板子上的lcd是16位，实际有24条线，按RGB565模式，有的线没有使用。

一般使用高位而不使用低位，高位权重大。

lcd接口使用RGB接口：

VD0-VD23 ：数据线，板子使用了16位

HSYNC ： 又称为VLINE行同步信号，lcd收到该信号后显示下一行

VSYNC ：又称为VFRAME。帧同步信号。LCD收到该信号，从头开始显示。

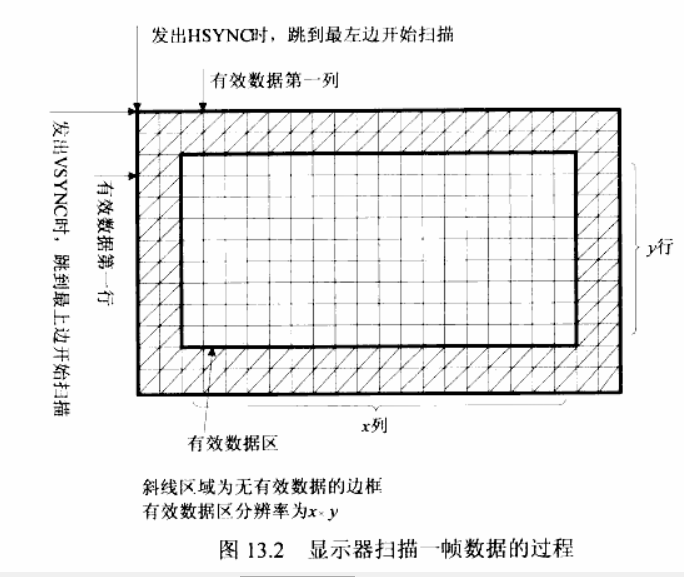
VDEN: 数据有效

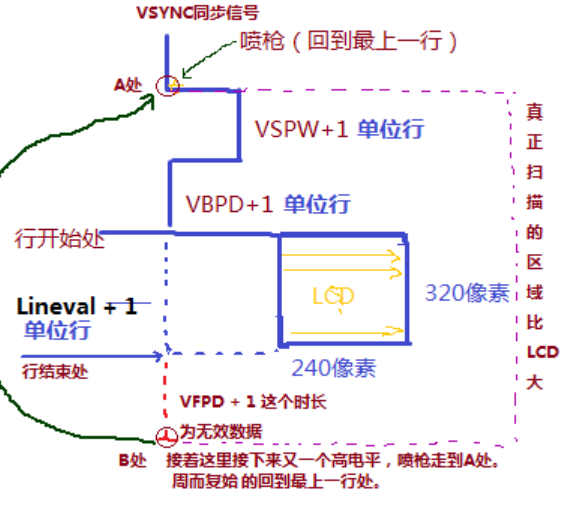
LEND: 行结束信号，不是必需的。

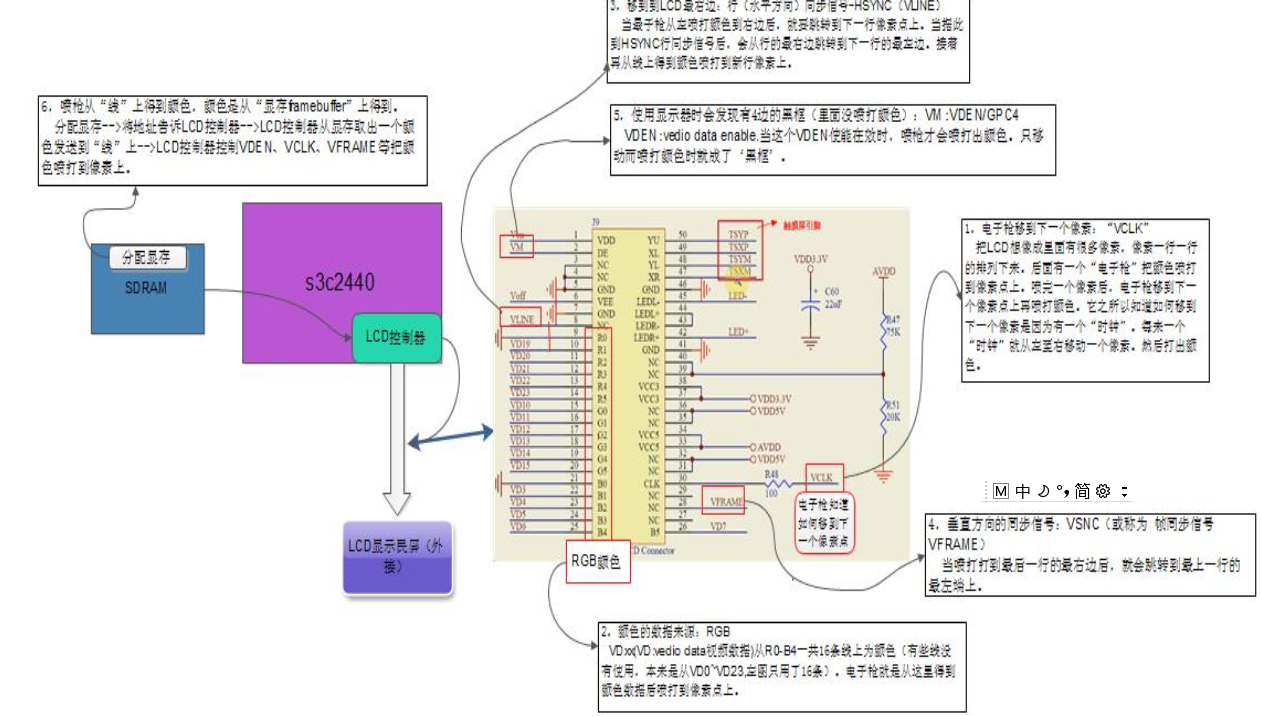
PWEN: 电源开关

VCLK: 像素时钟，切换显示下一个点。

设置 LCD 控制器时，不同的 LCD 的"VCLK"(频率）可能不会一样，所以要  
调整 LCD 控制器来设置频率的快和慢，使其发出合适的时钟（若 1 秒一次，则  
喷打颜色到像素点时都看的清晰太慢了。若 VCLK 为 1GHz，可能又反应不过  
来。所以要调 LCD 控制器使其发出合适的时钟）。

  
发出合适的“时钟”要看 LCD 屏的硬件手册





LCD硬件相关操作：

1. 根据 LCD 手册设置 LCD 控制器。（时钟参数等）  
   2、 分配显存，并将显存的地址告诉 LCD 控制器。  
   还要说明颜色格式。一个像素用多少个字节来表示等。  
   3、之后就是 2440 相关的操作:配置引脚用于 LCD。芯片的管脚可以配置成输入  
   输出，还可以配置为 LCD 引脚。

## 2.3 LCD 驱动程序编写

### 2.3.1 lcd 初始化

**static** **struct** fb\_info \*s3c\_lcd;

**static** **int** lcd\_init(**void**)

{

    1. 分配一个fb\_info

    s3c\_lcd = framebuffer\_alloc(0, NULL);

     2. 设置

     2.1 设置固定的参数

    2.2 设置可变的参数

     2.3 设置操作函数

     2.4 其他的设置

     3. 硬件相关的操作

    3.1 配置GPIO用于LCD

     3.2 根据LCD手册设置LCD控制器, 比如VCLK的频率等

     3.3 分配显存(framebuffer), 并把地址告诉LCD控制器

    4. 注册

    register\_framebuffer(s3c\_lcd);

**return** 0;

}

**struct** fb\_info \*framebuffer\_alloc(**size\_t** size, **struct** device \*dev);

参 1 为大小。在定义 fb\_info 结构时，结构体的大小是固定了的。内核中经常有这个取巧的方法，本来一个结构体原本分配的大小，而内核紧接着再分配了一段大小给这个结构体。定义时的结构体里面有一个指针，指向这个内核又分配的一段空间，这个空间里放“私有数据”。传入了一个“size”大小，本来 fb\_info 结构体的大小如下：int fb\_info\_size = sizeof(struct fb\_info);然后传入的 size 大小与原本定义的“fb\_info”结构体大小相加：分配这一  
段 p 空间。p = kzalloc(fb\_info\_size + size, GFP\_KERNEL);然后 par 指向额外分配的那段 size 空间。info->par = p + fb\_info\_size;我们不需要这段额外空间就不设置。就把size=0.

### 2.3.2 lcd参数设置

/\* 1. 分配一个fb\_info \*/

    s3c\_lcd = framebuffer\_alloc(0, NULL);

    /\* 2. 设置 \*/

    /\* 2.1 设置固定的参数 \*/

**strcpy**(s3c\_lcd->fix.id, "mylcd");                          //名字

    s3c\_lcd->fix.smem\_len = 240\*320\*16/8;           //显存的长度，单位字节

    s3c\_lcd->fix.type     = FB\_TYPE\_PACKED\_PIXELS;//  压缩像素数据存储方式

    s3c\_lcd->fix.visual   = FB\_VISUAL\_TRUECOLOR; /\* TFT 真彩色，还有单色模式\*/

    s3c\_lcd->fix.line\_length = 240\*2;           //单行数据占用字节数

    /\* 2.2 设置可变的参数 \*/

    s3c\_lcd->var.xres           = 240;        //分辨率x

    s3c\_lcd->var.yres           = 320;        //分辨率y

    s3c\_lcd->var.xres\_virtual   = 240;          //虚拟分辨率

    s3c\_lcd->var.yres\_virtual   = 320;

    s3c\_lcd->var.bits\_per\_pixel = 16;          //一个像素占用位数

    /\* RGB:565 \*/

    s3c\_lcd->var.red.offset     = 11;          //16位数据从第几位开始

    s3c\_lcd->var.red.length     = 5;              //总共占用位数

    s3c\_lcd->var.red.msb\_right=0;               //最重要的一位在左边（高位）

    s3c\_lcd->var.green.offset   = 5;

    s3c\_lcd->var.green.length   = 6;

    s3c\_lcd->var.blue.offset    = 0;

    s3c\_lcd->var.blue.length    = 5;

    s3c\_lcd->var.activate       = FB\_ACTIVATE\_NOW;   //重新设置参数后立即生效

/\* 2.3 设置操作函数 \*/

    s3c\_lcd->fbops              = &s3c\_lcdfb\_ops;

/\* 2.4 其他的设置 \*/

/\*假的调色板，16个元素的数组，static u32 pseudo\_palette[16];

在函数s3c\_lcdfb\_setcolreg中设置该数组各个位的值\*/

    s3c\_lcd->pseudo\_palette = pseudo\_palette;

    //s3c\_lcd->screen\_base  = ;               /\* 显存的虚拟地址 \*/

    s3c\_lcd->screen\_size   = 240\*320\*16/8;     // 占用字节空间数

    /\* 3. 硬件相关的操作 \*/

    /\* 3.1 配置GPIO用于LCD \*/

    gpbcon = ioremap(0x56000010, 8);

    gpbdat = gpbcon+1;

    gpccon = ioremap(0x56000020, 4);

    gpdcon = ioremap(0x56000030, 4);

    gpgcon = ioremap(0x56000060, 4);

    \*gpccon  = 0xaaaaaaaa;   /\* GPIO管脚用于VD[7:0],LCDVF[2:0],VM,VFRAME,VLINE,VCLK,LEND \*/

    \*gpdcon  = 0xaaaaaaaa;   /\* GPIO管脚用于VD[23:8] \*/

    \*gpbcon &= ~(3);  /\* GPB0设置为输出引脚 \*/

    \*gpbcon |= 1;

    \*gpbdat &= ~1;     /\* 输出低电平 \*/

    \*gpgcon |= (3<<8); /\* GPG4用作LCD\_PWREN ，背光灯开关\*/

    2.3.4 lcd时序参数设置

    /\* 3.2 根据LCD手册设置LCD控制器, 比如VCLK的频率等 \*/

    lcd\_regs = ioremap(0x4D000000, **sizeof**(**struct** lcd\_regs));

    /\* bit[17:8]: VCLK = HCLK / [(CLKVAL+1) x 2],  三星LCD手册P14

        VCLK:lcd像素时钟，需要看lcd手册。HCLK,是高速外设时钟，等于内存时钟频率

     \*            10MHz(100ns) = 100MHz / [(CLKVAL+1) x 2]

     \*            CLKVAL = 4

     \* bit[6:5]: 0b11, TFT LCD

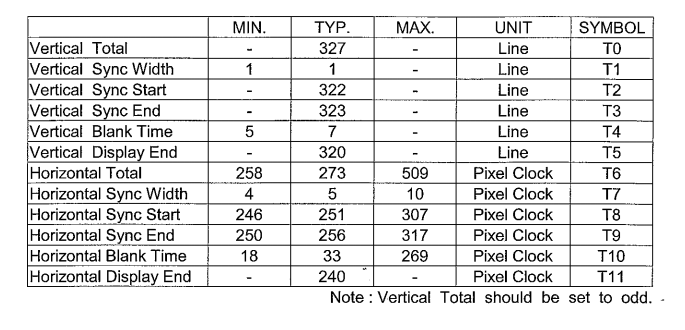
     \* bit[4:1]: 0b1100, 16 bpp for TFT

     \* bit[0]  : 0 = Disable the video output and the LCD control signal.

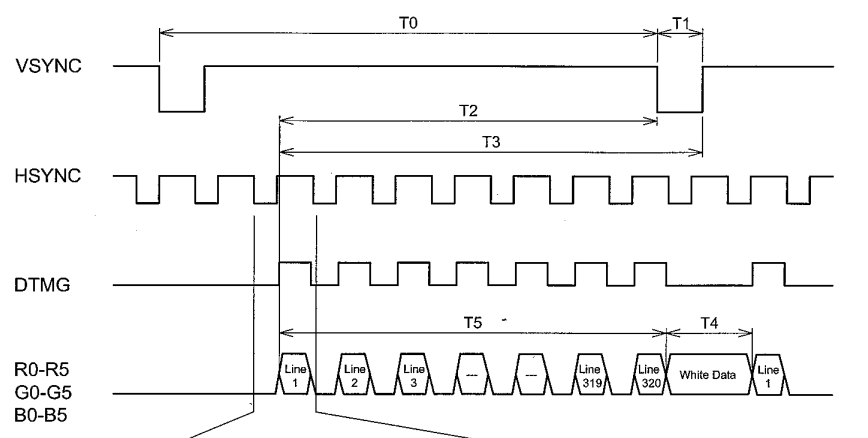
     \*/

    lcd\_regs->lcdcon1  = (4<<8) | (3<<5) | (0x0c<<1);

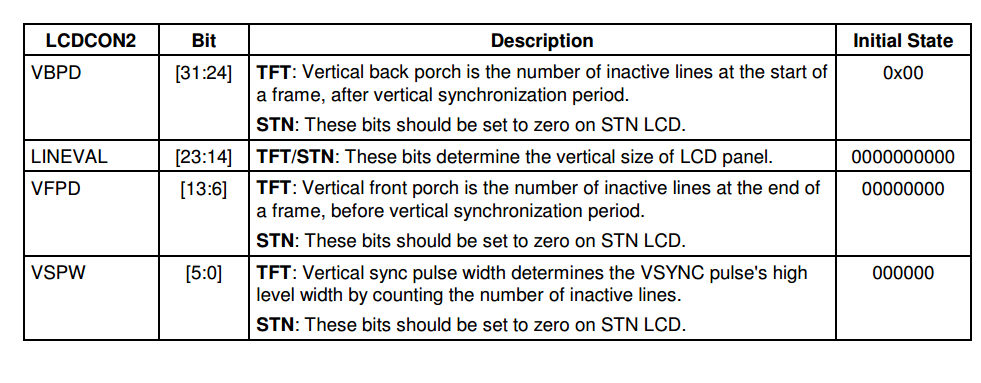
  LCD屏时间参数



LCD屏垂直方向时序图



LCD控制器寄存器



#if 1

/\* 垂直方向的时间参数

寄存器中设置数据为实际值减去1

VSYNC:帧同步脉冲

     \* bit[31:24]: VBPD, VSYNC之后再过多长时间才能发出第1行数据

     \*             LCD手册 T0-T2-T1=4

     \*             VBPD=3

     \* bit[23:14]: LINVAL多少行, 320, 所以LINEVAL=320-1=319

     \* bit[13:6] : VFPD, 发出最后一行数据之后，再过多长时间才发出VSYNC

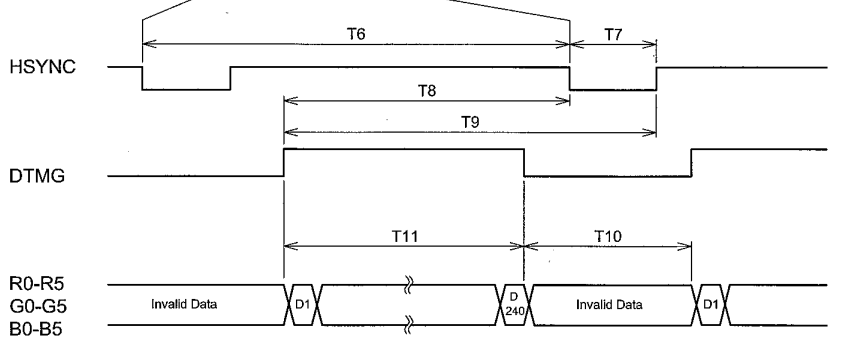
     \*             LCD手册T2-T5=322-320=2, 所以VFPD=2-1=1

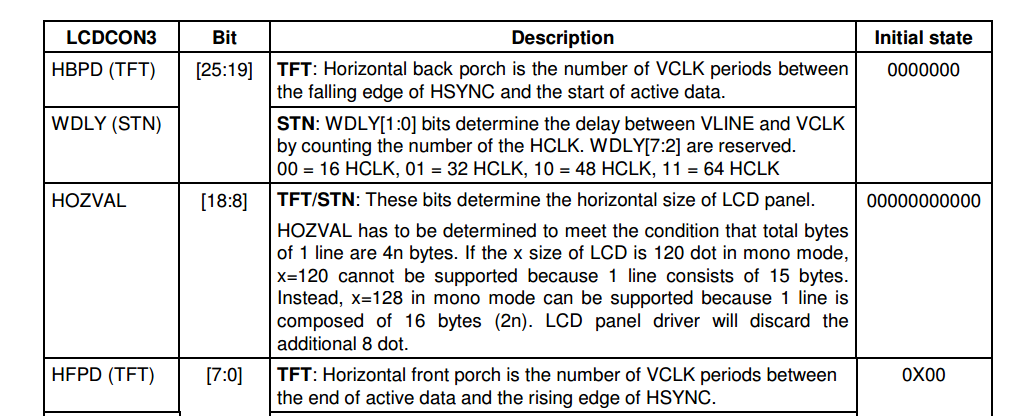
     \* bit[5:0]  : VSPW, VSYNC信号的脉冲宽度, LCD手册T1=1, 所以VSPW=1-1=0

     \*/

    lcd\_regs->lcdcon2  = (3<<24) | (319<<14) | (1<<6) | (0<<0);

LCD屏水平方向时序





/\* 水平方向的时间参数

     \* bit[25:19]: HBPD,HSYNC之后再过多长时间才能发出第1个像素数据

     \*             LCD手册 T6-T7-T8=17

     \*             HBPD=16

     \* bit[18:8]: HOZVAL多少列, 240, 所以HOZVAL=240-1=239

     \* bit[7:0] : HFPD, 发出一行里最后一个象素数据之后，再过多长时间才发出HSYNC

     \*             LCD手册T8-T11=251-240=11, 所以HFPD=11-1=10

     \*/

    lcd\_regs->lcdcon3 = (16<<19) | (239<<8) | (10<<0);

    /\* 水平方向的同步信号

     \* bit[7:0]   : HSPW, HSYNC信号的脉冲宽度, LCD手册T7=5, 所以HSPW=5-1=4

     \*/

    lcd\_regs->lcdcon4 = 4;

#else

lcd\_regs->lcdcon2 = S3C2410\_LCDCON2\_VBPD(5) | \

        S3C2410\_LCDCON2\_LINEVAL(319) | \

        S3C2410\_LCDCON2\_VFPD(3) | \

        S3C2410\_LCDCON2\_VSPW(1);

lcd\_regs->lcdcon3 = S3C2410\_LCDCON3\_HBPD(10) | \

        S3C2410\_LCDCON3\_HOZVAL(239) | \

        S3C2410\_LCDCON3\_HFPD(1);

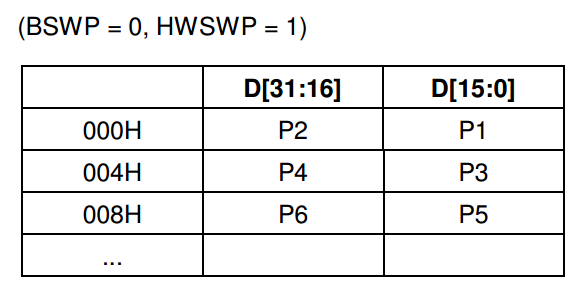
lcd\_regs->lcdcon4 = S3C2410\_LCDCON4\_MVAL(13) | \

        S3C2410\_LCDCON4\_HSPW(0);

#endif

lcd数据存储方式：

16位数据，在32位整型存储空间中，前一个像素数据存储在低16位空间



/\* 信号的极性

     \* bit[11]: 1=565 format

     \* bit[10]: 0 = The video data is fetched at VCLK falling edge

     \* bit[9] : 1 = HSYNC信号要反转,即低电平有效

     \* bit[8] : 1 = VSYNC信号要反转,即低电平有效

     \* bit[6] : 0 = VDEN不用反转

     \* bit[3] : 0 = PWREN输出0

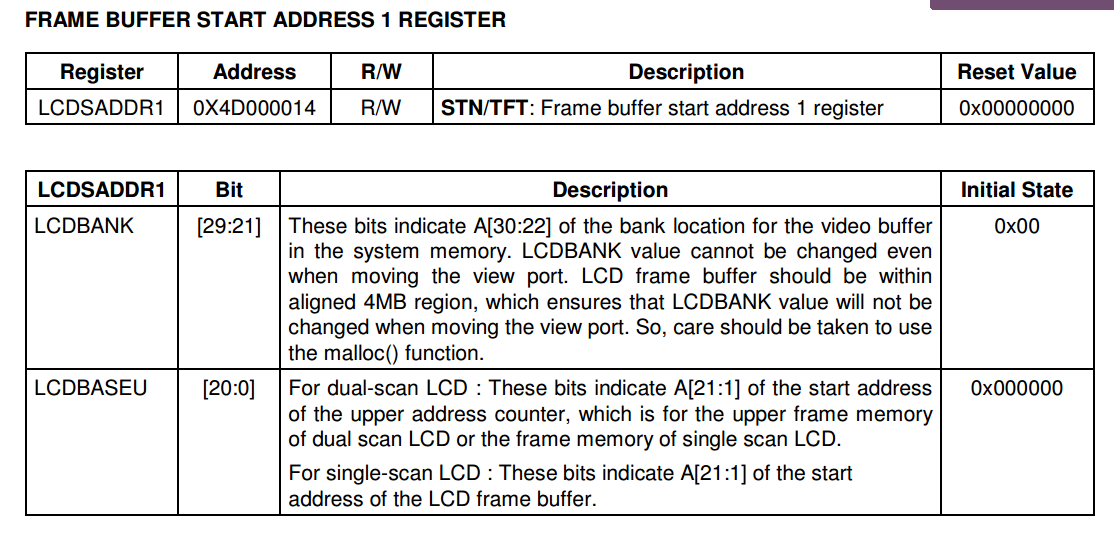
     \* bit[1] : 0 = BSWP

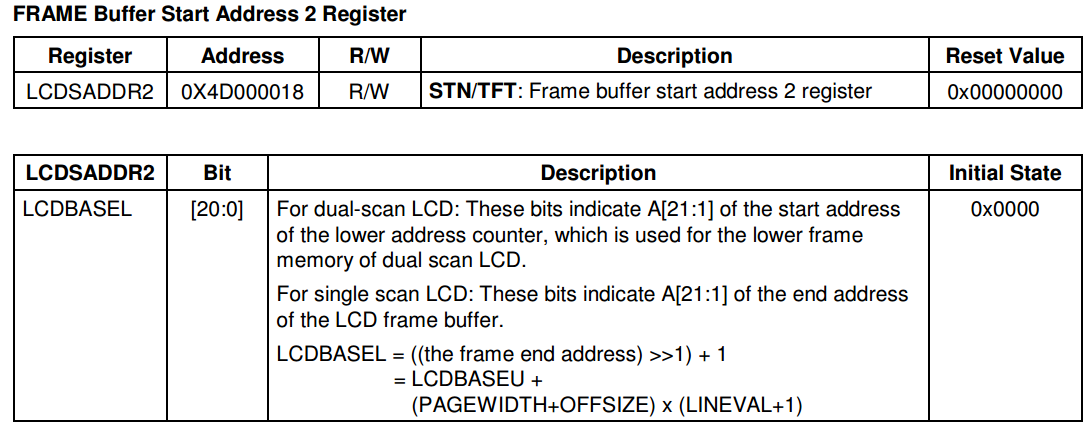
     \* bit[0] : 1 = HWSWP 2440手册P413

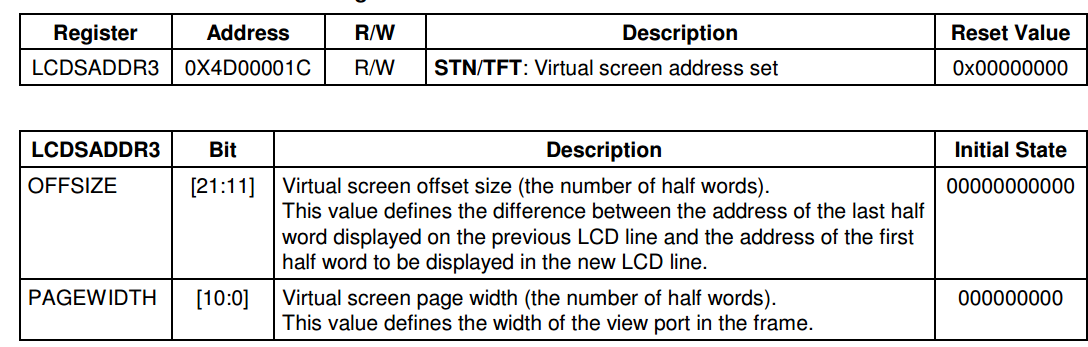
     \*/

    lcd\_regs->lcdcon5 = (1<<11) | (0<<10) | (1<<9) | (1<<8) | (1<<0);

lcd 起始地址寄存器1：







如何计算设置寄存器



/\* 3.3 分配显存(framebuffer), 并把地址告诉LCD控制器

s3c\_lcd->fix.smem\_start是物理地址，DMA使用物理地址。

 s3c\_lcd->screen\_base 是虚拟地址，应用读写数据使用虚拟地址

lcd为单扫描模式。

LCDBANK:存放起始地址的22-30位

LCDBASEU: 存放起始地址的1-21位。2^22=4M

lcdsaddr1：起始地址

lcdsaddr2 ：结束地址1：21位

\*/

    s3c\_lcd->screen\_base = dma\_alloc\_writecombine(NULL, s3c\_lcd->fix.smem\_len, &s3c\_lcd->fix.smem\_start, GFP\_KERNEL);

    lcd\_regs->lcdsaddr1  = (s3c\_lcd->fix.smem\_start >> 1) & ~(3<<30);//最高两位清零

    lcd\_regs->lcdsaddr2  = ((s3c\_lcd->fix.smem\_start + s3c\_lcd->fix.smem\_len) >> 1) & 0x1fffff;

    lcd\_regs->lcdsaddr3  = (240\*16/16);  /\* 一行的长度(单位: 2字节) \*/

    //s3c\_lcd->fix.smem\_start = xxx;  /\* 显存的物理地址 \*/

    /\* 启动LCD \*/

    lcd\_regs->lcdcon1 |= (1<<0); /\* 使能LCD控制器LCD video output and the logic enable/disable \*/

    lcd\_regs->lcdcon5 |= (1<<3); /\* 使能LCD本身 LCD\_PWREN output signal enable/disable \*/

\*gpbdat |= 1;     /\* 输出高电平, 使能背光 \*/

2.3.2 设置调色板

/\*

如果一张图片的颜色不超过16种，那么用调色板存储可以节省空间。

static u32 pseudo\_palette[16]。

regno: pseudo\_palette数组索引，

\*/

static inline unsigned int chan\_to\_field(unsigned int chan,

                     struct fb\_bitfield \*bf)

{

    chan &= 0xffff;   //保留低16位

    chan >>= 16 - bf->length;//保留颜色高bf->length位

**return** chan << bf->offset;//移动到该颜色的偏移位

}

static int s3c\_lcdfb\_setcolreg(unsigned int regno, unsigned int red,

                 unsigned int green, unsigned int blue,

                 unsigned int transp, struct fb\_info \*info)

{

    unsigned int val;

**if** (regno > 16)

**return** 1;

    /\* 用red,green,blue三原色构造出val \*/

    val  = chan\_to\_field(red,    &info->**var**.red);

    val |= chan\_to\_field(green, &info->**var**.green);

    val |= chan\_to\_field(blue,    &info->**var**.blue);

    //((u32 \*)(info->pseudo\_palette))[regno] = val;

    pseudo\_palette[regno] = val;

**return** 0;

}

## 2.4 lcd 驱动完整程序

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/errno.h>

#include <linux/string.h>

#include <linux/mm.h>

#include <linux/slab.h>

#include <linux/delay.h>

#include <linux/fb.h>

#include <linux/init.h>

#include <linux/dma-mapping.h>

#include <linux/interrupt.h>

#include <linux/workqueue.h>

#include <linux/wait.h>

#include <linux/platform\_device.h>

#include <linux/clk.h>

#include <asm/io.h>

#include <asm/uaccess.h>

#include <asm/div64.h>

#include <asm/mach/map.h>

#include <asm/arch/regs-lcd.h>

#include <asm/arch/regs-gpio.h>

#include <asm/arch/fb.h>

**static** **int** s3c\_lcdfb\_setcolreg(unsigned **int** regno, unsigned **int** red,

                 unsigned **int** green, unsigned **int** blue,

                 unsigned **int** transp, **struct** fb\_info \*info);

**struct** lcd\_regs {

    unsigned **long**  lcdcon1;

    unsigned **long**  lcdcon2;

    unsigned **long**  lcdcon3;

    unsigned **long**  lcdcon4;

    unsigned **long**  lcdcon5;

    unsigned **long**  lcdsaddr1;

    unsigned **long**  lcdsaddr2;

    unsigned **long**  lcdsaddr3;

    unsigned **long**  redlut;

    unsigned **long**  greenlut;

    unsigned **long**  bluelut;

    unsigned **long**  reserved[9];

    unsigned **long**  dithmode;

    unsigned **long**  tpal;

    unsigned **long**  lcdintpnd;

    unsigned **long**  lcdsrcpnd;

    unsigned **long**  lcdintmsk;

    unsigned **long**  lpcsel;

};

**static** **struct** fb\_ops s3c\_lcdfb\_ops = {

    .owner      = THIS\_MODULE,

    .fb\_setcolreg   = s3c\_lcdfb\_setcolreg,

    .fb\_fillrect    = cfb\_fillrect,

    .fb\_copyarea    = cfb\_copyarea,

    .fb\_imageblit   = cfb\_imageblit,

};

**static** **struct** fb\_info \*s3c\_lcd;

**static** **volatile** unsigned **long** \*gpbcon;

**static** **volatile** unsigned **long** \*gpbdat;

**static** **volatile** unsigned **long** \*gpccon;

**static** **volatile** unsigned **long** \*gpdcon;

**static** **volatile** unsigned **long** \*gpgcon;

**static** **volatile** **struct** lcd\_regs\* lcd\_regs;

**static** u32 pseudo\_palette[16];

 /\* from pxafb.c \*/

**static** **inline** unsigned **int** chan\_to\_field(unsigned **int** chan, **struct** fb\_bitfield \*bf)

{

    chan &= 0xffff;

    chan >>= 16 - bf->length;

**return** chan << bf->offset;

}

**static** **int** s3c\_lcdfb\_setcolreg(unsigned **int** regno, unsigned **int** red,

                 unsigned **int** green, unsigned **int** blue,

                 unsigned **int** transp, **struct** fb\_info \*info)

{

    unsigned **int** val;

**if** (regno > 16)

**return** 1;

    /\* 用red,green,blue三原色构造出val \*/

    val  = chan\_to\_field(red,    &info->var.red);

    val |= chan\_to\_field(green, &info->var.green);

    val |= chan\_to\_field(blue,    &info->var.blue);

    //((u32 \*)(info->pseudo\_palette))[regno] = val;

    pseudo\_palette[regno] = val;

**return** 0;

}

**static** **int** lcd\_init(**void**)

{

    /\* 1. 分配一个fb\_info \*/

    s3c\_lcd = framebuffer\_alloc(0, NULL);

    /\* 2. 设置 \*/

    /\* 2.1 设置固定的参数 \*/

**strcpy**(s3c\_lcd->fix.id, "mylcd");

    s3c\_lcd->fix.smem\_len = 480\*272\*16/8;

    s3c\_lcd->fix.type     = FB\_TYPE\_PACKED\_PIXELS;

    s3c\_lcd->fix.visual   = FB\_VISUAL\_TRUECOLOR; /\* TFT \*/

    s3c\_lcd->fix.line\_length = 480\*2;

    /\* 2.2 设置可变的参数 \*/

    s3c\_lcd->var.xres           = 480;

    s3c\_lcd->var.yres           = 272;

    s3c\_lcd->var.xres\_virtual   = 480;

    s3c\_lcd->var.yres\_virtual   = 272;

    s3c\_lcd->var.bits\_per\_pixel = 16;

    /\* RGB:565 \*/

    s3c\_lcd->var.red.offset     = 11;

    s3c\_lcd->var.red.length     = 5;

    s3c\_lcd->var.green.offset   = 5;

    s3c\_lcd->var.green.length   = 6;

    s3c\_lcd->var.blue.offset    = 0;

    s3c\_lcd->var.blue.length    = 5;

    s3c\_lcd->var.activate       = FB\_ACTIVATE\_NOW;

    /\* 2.3 设置操作函数 \*/

    s3c\_lcd->fbops              = &s3c\_lcdfb\_ops;

    /\* 2.4 其他的设置 \*/

    s3c\_lcd->pseudo\_palette = pseudo\_palette;

    //s3c\_lcd->screen\_base  = ;  /\* 显存的虚拟地址 \*/

    s3c\_lcd->screen\_size   = 480\*272\*16/8;

    /\* 3. 硬件相关的操作 \*/

    /\* 3.1 配置GPIO用于LCD \*/

    gpbcon = ioremap(0x56000010, 8);

    gpbdat = gpbcon+1;

    gpccon = ioremap(0x56000020, 4);

    gpdcon = ioremap(0x56000030, 4);

    gpgcon = ioremap(0x56000060, 4);

    \*gpccon  = 0xaaaaaaaa;   /\* GPIO管脚用于VD[7:0],LCDVF[2:0],VM,VFRAME,VLINE,VCLK,LEND \*/

    \*gpdcon  = 0xaaaaaaaa;   /\* GPIO管脚用于VD[23:8] \*/

    \*gpbcon &= ~(3);  /\* GPB0设置为输出引脚 \*/

    \*gpbcon |= 1;

    \*gpbdat &= ~1;     /\* 输出低电平 \*/

    \*gpgcon |= (3<<8); /\* GPG4用作LCD\_PWREN \*/

    /\* 3.2 根据LCD手册设置LCD控制器, 比如VCLK的频率等 \*/

    lcd\_regs = ioremap(0x4D000000, **sizeof**(**struct** lcd\_regs));

    /\* bit[17:8]: VCLK = HCLK / [(CLKVAL+1) x 2], LCD手册P14

     \*            10MHz(100ns) = 100MHz / [(CLKVAL+1) x 2]

     \*            CLKVAL = 4

     \* bit[6:5]: 0b11, TFT LCD

     \* bit[4:1]: 0b1100, 16 bpp for TFT

     \* bit[0]  : 0 = Disable the video output and the LCD control signal.

     \*/

    lcd\_regs->lcdcon1  = (4<<8) | (3<<5) | (0x0c<<1);

#if 1

    /\* 垂直方向的时间参数

     \* bit[31:24]: VBPD, VSYNC之后再过多长时间才能发出第1行数据

     \*             LCD手册 T0-T2-T1=4

     \*             VBPD=3

     \* bit[23:14]: 多少行, 320, 所以LINEVAL=320-1=319

     \* bit[13:6] : VFPD, 发出最后一行数据之后，再过多长时间才发出VSYNC

     \*             LCD手册T2-T5=322-320=2, 所以VFPD=2-1=1

     \* bit[5:0]  : VSPW, VSYNC信号的脉冲宽度, LCD手册T1=1, 所以VSPW=1-1=0

     \*/

    lcd\_regs->lcdcon2  = (1<<24) | (271<<14) | (1<<6) | (9);

    /\* 水平方向的时间参数

     \* bit[25:19]: HBPD, VSYNC之后再过多长时间才能发出第1行数据

     \*             LCD手册 T6-T7-T8=17

     \*             HBPD=16

     \* bit[18:8]: 多少列, 240, 所以HOZVAL=240-1=239

     \* bit[7:0] : HFPD, 发出最后一行里最后一个象素数据之后，再过多长时间才发出HSYNC

     \*             LCD手册T8-T11=251-240=11, 所以HFPD=11-1=10

     \*/

    lcd\_regs->lcdcon3 = (1<<19) | (479<<8) | (1);

    /\* 水平方向的同步信号

     \* bit[7:0]   : HSPW, HSYNC信号的脉冲宽度, LCD手册T7=5, 所以HSPW=5-1=4

     \*/

    lcd\_regs->lcdcon4 = 40;

#else

lcd\_regs->lcdcon2 = S3C2410\_LCDCON2\_VBPD(5) | \

        S3C2410\_LCDCON2\_LINEVAL(319) | \

        S3C2410\_LCDCON2\_VFPD(3) | \

        S3C2410\_LCDCON2\_VSPW(1);

lcd\_regs->lcdcon3 = S3C2410\_LCDCON3\_HBPD(10) | \

        S3C2410\_LCDCON3\_HOZVAL(239) | \

        S3C2410\_LCDCON3\_HFPD(1);

lcd\_regs->lcdcon4 = S3C2410\_LCDCON4\_MVAL(13) | \

        S3C2410\_LCDCON4\_HSPW(0);

#endif

    /\* 信号的极性

     \* bit[11]: 1=565 format

     \* bit[10]: 0 = The video data is fetched at VCLK falling edge

     \* bit[9] : 1 = HSYNC信号要反转,即低电平有效

     \* bit[8] : 1 = VSYNC信号要反转,即低电平有效

     \* bit[6] : 0 = VDEN不用反转

     \* bit[3] : 0 = PWREN输出0

     \* bit[1] : 0 = BSWP

     \* bit[0] : 1 = HWSWP 2440手册P413

     \*/

    lcd\_regs->lcdcon5 = (1<<11) | (0<<10) | (1<<9) | (1<<8) | (1<<0);

    /\* 3.3 分配显存(framebuffer), 并把地址告诉LCD控制器 \*/

    s3c\_lcd->screen\_base = dma\_alloc\_writecombine(NULL, s3c\_lcd->fix.smem\_len, &s3c\_lcd->fix.smem\_start, GFP\_KERNEL);

    lcd\_regs->lcdsaddr1  = (s3c\_lcd->fix.smem\_start >> 1) & ~(3<<30);

    lcd\_regs->lcdsaddr2  = ((s3c\_lcd->fix.smem\_start + s3c\_lcd->fix.smem\_len) >> 1) & 0x1fffff;

    lcd\_regs->lcdsaddr3  = (480\*16/16);  /\* 一行的长度(单位: 2字节) \*/

    //s3c\_lcd->fix.smem\_start = xxx;  /\* 显存的物理地址 \*/

    /\* 启动LCD \*/

    lcd\_regs->lcdcon1 |= (1<<0); /\* 使能LCD控制器 \*/

    lcd\_regs->lcdcon5 |= (1<<3); /\* 使能LCD本身 \*/

    \*gpbdat |= 1;     /\* 输出高电平, 使能背光 \*/

    /\* 4. 注册 \*/

    register\_framebuffer(s3c\_lcd);

**return** 0;

}

**static** **void** lcd\_exit(**void**)

{

    unregister\_framebuffer(s3c\_lcd);

    lcd\_regs->lcdcon1 &= ~(1<<0); /\* 关闭LCD本身 \*/

    \*gpbdat &= ~1;     /\* 关闭背光 \*/

    dma\_free\_writecombine(NULL, s3c\_lcd->fix.smem\_len, s3c\_lcd->screen\_base, s3c\_lcd->fix.smem\_start);

    iounmap(lcd\_regs);

    iounmap(gpbcon);

    iounmap(gpccon);

    iounmap(gpdcon);

    iounmap(gpgcon);

    framebuffer\_release(s3c\_lcd);

}

module\_init(lcd\_init);

module\_exit(lcd\_exit);

MODULE\_LICENSE("GPL");

## 2.5 编译测试

配置内核去掉内核带的驱动

make uImage

make modules

//2.6内核需要三个模块，3.4不需要。

insmod cfb\_fillrect.ko

insmod cfb\_copyarea.ko

insmod cfb\_imageblit.ko

insmod lcd.ko

echo hello >/dev/tty1 //lcd应该有hello

cat lcd.ko > /dev/fb0 //lcd应该有显示，花屏

## 2.6 修改源码使用内核自带的lcd驱动

static struct platform\_device \*smdk2440\_devices[] \_\_initdata = {

    &s3c\_device\_usb,

    &s3c\_device\_lcd, //添加lcd设备

    &s3c\_device\_wdt,

    &s3c\_device\_i2c,

    &s3c\_device\_iis,

};

static void \_\_init smdk2440\_machine\_init(void)

{

    s3c24xx\_fb\_set\_platdata(&smdk2440\_lcd\_cfg); //添加lcd的配置参数

    platform\_add\_devices(smdk2440\_devices, ARRAY\_SIZE(smdk2440\_devices));

    /\*添加i2c设备\*/

    i2c\_register\_board\_info(0,smdk2440\_i2c\_devs,ARRAY\_SIZE(smdk2440\_i2c\_devs));

    smdk\_machine\_init();

}

//修改下面结构体设置参数

static struct s3c2410fb\_mach\_info smdk2440\_lcd\_cfg \_\_initdata = {

    .regs   = {

        .lcdcon1    = S3C2410\_LCDCON1\_TFT16BPP |

                  S3C2410\_LCDCON1\_TFT |

                  S3C2410\_LCDCON1\_CLKVAL(0x04),

        .lcdcon2    = S3C2410\_LCDCON2\_VBPD(7) |

                  S3C2410\_LCDCON2\_LINEVAL(319) |

                  S3C2410\_LCDCON2\_VFPD(6) |

                  S3C2410\_LCDCON2\_VSPW(3),

        .lcdcon3    = S3C2410\_LCDCON3\_HBPD(19) |

                  S3C2410\_LCDCON3\_HOZVAL(239) |

                  S3C2410\_LCDCON3\_HFPD(7),

        .lcdcon4    = S3C2410\_LCDCON4\_MVAL(0) |

                  S3C2410\_LCDCON4\_HSPW(3),

        .lcdcon5    = S3C2410\_LCDCON5\_FRM565 |

                  S3C2410\_LCDCON5\_INVVLINE |

                  S3C2410\_LCDCON5\_INVVFRAME |

                  S3C2410\_LCDCON5\_PWREN |

                  S3C2410\_LCDCON5\_HWSWP,

    },

#if 0

    /\* currently setup by downloader \*/

    .gpccon     = 0xaa940659,

    .gpccon\_mask    = 0xffffffff,

    .gpcup      = 0x0000ffff,

    .gpcup\_mask = 0xffffffff,

    .gpdcon     = 0xaa84aaa0,

    .gpdcon\_mask    = 0xffffffff,

    .gpdup      = 0x0000faff,

    .gpdup\_mask = 0xffffffff,

#endif

.lpcsel     = ((0xCE6) & ~7) | 1<<4,

    .type       = S3C2410\_LCDCON1\_TFT16BPP,

    .width      = 240,

    .height     = 320,

    .xres       = {

        .min    = 240,

        .max    = 240,

        .defval = 240,

    },

    .yres       = {

        .min    = 320,

        .max    = 320,

        .defval = 320,

    },

    .bpp        = {

        .min    = 16,

        .max    = 16,

        .defval = 16,

    },

};

# 3 I2C驱动程序

## 3.1 i2c 总线协议原理

### 3.1.1 i2c总线接口

i2c 总线接口总共需要四条线，VCC,GND,SCL,SDA。是主从结构，所有输出是由主机发起。SCL,SDA必需外接上拉电阻。

### 3.1.2 i2c总线信号传输时序

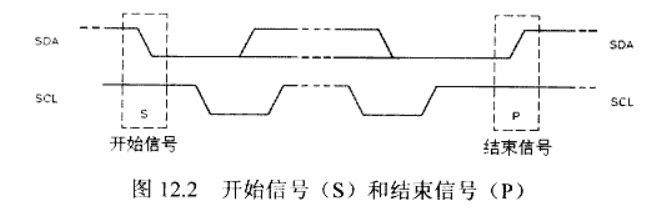
初始状态：SDA和SCL初始电平为高电平。

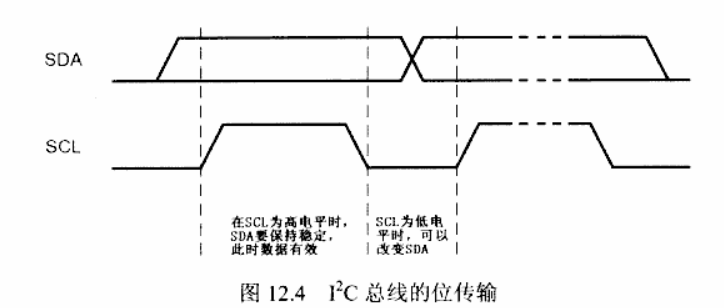
开始信号S：SCL为高电平时，SDA由高电平向低电平跳变

结束信号P：SCL为低电平时，SDA由低电平向高电平跳变

响应信号ACK: 主机在发送完8位数据后会释放总线控制权，接收器在收到8位数据后，在第9个时钟周期，拉低SDA电平。

数据传输：数据传输过程中SDA数据只能在SCL低电平时变化，SCL高电平时，SDA上传输的数据必须保持稳定





### 3.1.3 i2c总线的数据传输格式

启动一个传输时，主机先发出S信号，然后发出8位数据。8位数据前7位为从机的设备地址。第8位标识传输方向，0表示是写操作，1表示是读操作。

#### AT24CXX 时序

容量位 xxkb ，例如24c08 就是8kb。

可以一个字节一个字节的读写数据，内部256B分页。

对于128B和256B的芯片可以8字节页写，即8字节对齐的连续地址只需要发送一次地址即可读写8字节数据。设备地址固定地址部分1010.可变部分3bit, 对于256B-1024B, 内部分页地址也由该三位给出:

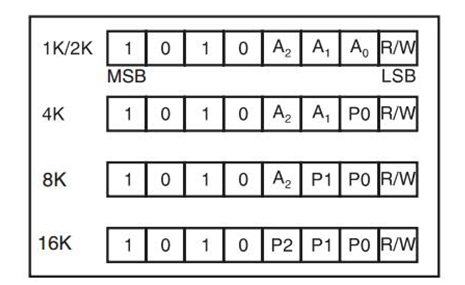
对于AT24C01A和AT24C02，A2、A1和A0引脚是配置器件的硬件地址输入。一根总线上可以连接多达八个1K / 2K的设备（器件寻址部分详细讨论了器件寻址）。

AT24C04使用A2和A1引脚作为硬件地址输入，在一根总线上有4个4K 的设备可用来寻址。A0引脚没有连接。

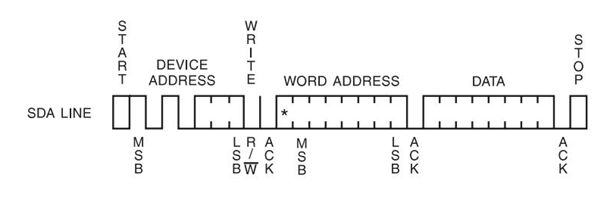
AT24C08A只使用A2引脚作为硬件地址输入，在一根总线上有2个8K 的设备可用来寻址。A0和A1引脚没有连接。

AT24C16A不使用设备地址引脚，这限制了一根总线上只能挂一个设备。A0、A1和A2引脚没有连接。

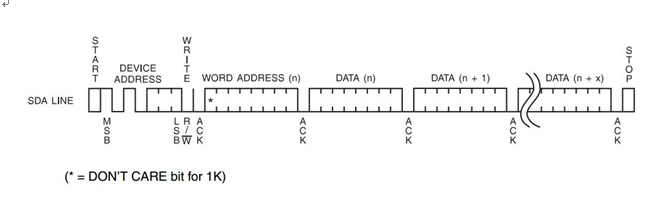
设备地址：



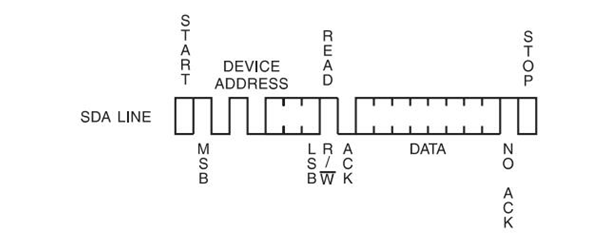
字节写入：



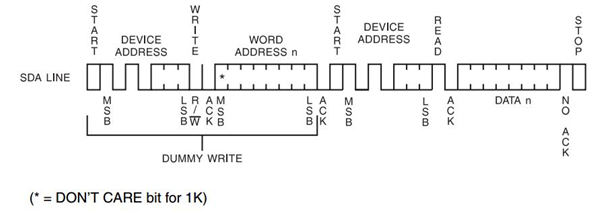
页写入：



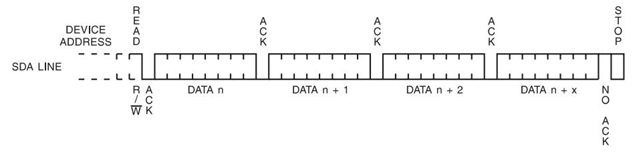
当前地址读：



随机读：

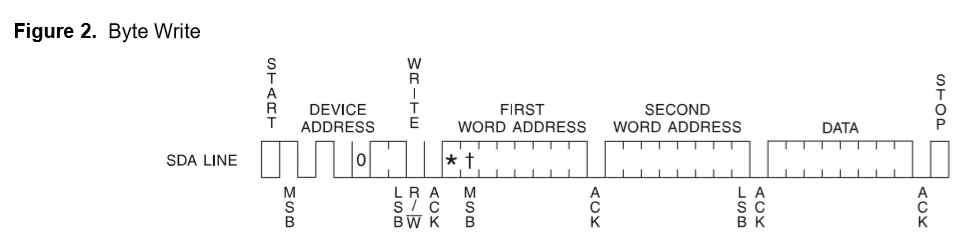


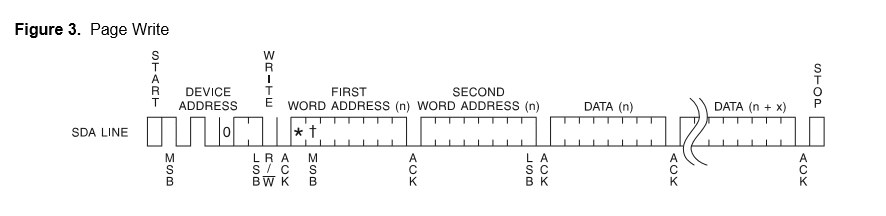
顺序读：

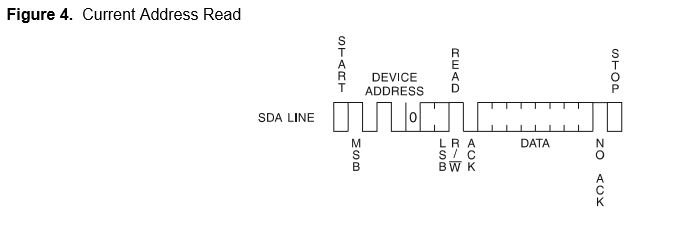


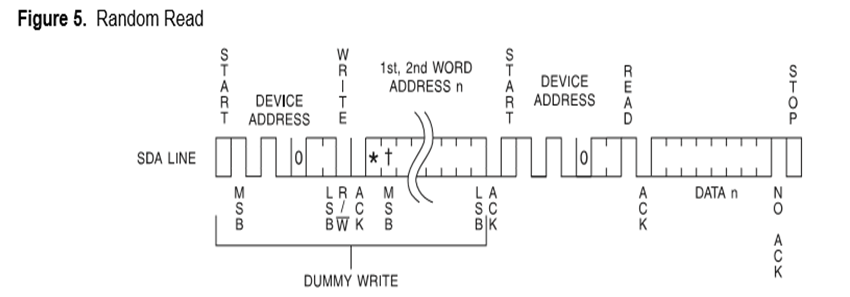
#### AT24C128/AT24C256时序

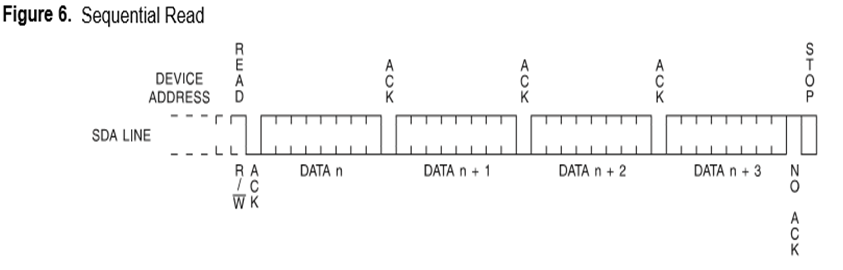
容量是32KB和64KB.64-Byte Page Write Mode。支持发送一次地址，连续写64B数据。读没有限制。The 128K/256K uses the two device address bits A1, A0 to allow as many as four devices on the same bus。意思是A1A0用于设备选择，最多可以接四个设备。BYTE WRITE: A write operation requires two 8-bit data word addresses following the device address word and acknowledgment。说明了需要两个字节的内部数据存储地址。(为什么以前的程序读不出来数据原因)











## 3.2 i2c裸板程序

### 3.2.1 i2c控制器初始化

/\*

 \* I2C初始化

 \*/

**void** i2c\_init(**void**)

{

    GPEUP  |= 0xc000;       // 禁止内部上拉

    GPECON |= 0xa0000000;   // 选择引脚功能：GPE15:IICSDA, GPE14:IICSCL

    INTMSK &= ~(BIT\_IIC);  //允许i2c中断

    /\* bit[7] = 1, 使能ACK

     \* bit[6] = 0, IICCLK = PCLK/16

     \* bit[5] = 1, 使能中断

     \* bit[3:0] = 0xf, Tx clock = IICCLK/16

     \* PCLK = 50MHz, IICCLK = 3.125MHz, Tx Clock = 0.195MHz

     \*/

    IICCON = (1<<7) | (0<<6) | (1<<5) | (0xf);  // 0xaf

    IICADD  = 0x10;     // S3C24xx作为从机时自身的地址

    IICSTAT = 0x10;     // I2C串行输出使能(Rx/Tx)

}

### 3.2.2 at24cxx读写

**void** at24cxx\_write(unsigned **char** address, unsigned **char** data)//随机写数据程序

{

    unsigned **char** val[2];

    val[0] = address;   //数据地址

    val[1] = data;       //数据

    i2c\_write(0xA0, val, 2); //0xA0是从机设备地址，该程序只适合256B以下地址访问。

}

unsigned **char** at24cxx\_read(unsigned **char** address)//随机读数据程序

{

    unsigned **char** val;

**printf**("at24cxx\_read address = %d\r\n", address);

    i2c\_write(0xA0, &address, 1);  //先写入数据地址

**printf**("at24cxx\_read send address ok\r\n");

i2c\_read(0xA0, (unsigned **char** \*)&val, 1);

//然后从总线读一字节数据。此时地址应该0xA1,标识是读方向。

**printf**("at24cxx\_read get data ok\r\n");

**return** val;

}

### 3.2.3 i2c读写

/\*

 \* I2C中断服务程序

 \* 根据剩余的数据长度选择继续传输或者结束

 \*/

**void** I2CIntHandle(**void**)

{

    unsigned **int** iicSt,i;

    // 清中断

    SRCPND = BIT\_IIC;

    INTPND = BIT\_IIC;

    iicSt  = IICSTAT;

**if**(iicSt & 0x8){ **printf**("Bus arbitration failed\n\r"); }

**switch** (g\_tS3C24xx\_I2C.Mode)

    {

**case** WRDATA:

        {

**if**((g\_tS3C24xx\_I2C.DataCount--) == 0)

            {

                // 下面两行用来恢复I2C操作，发出P信号

                IICSTAT = 0xd0;

                IICCON  = 0xaf;

                Delay(10000);  // 等待一段时间以便P信号已经发出

**break**;

            }

            IICDS = g\_tS3C24xx\_I2C.pData[g\_tS3C24xx\_I2C.Pt++];

                        // 将数据写入IICDS后，需要一段时间才能出现在SDA线上

**for** (i = 0; i < 10; i++);

            IICCON = 0xaf;      // 恢复I2C传输

**break**;

        }

**case** RDDATA:

        {

**if** (g\_tS3C24xx\_I2C.Pt == -1)

            {

                // 这次中断是发送I2C设备地址后发生的，没有数据

                // 只接收一个数据时，不要发出ACK信号

                g\_tS3C24xx\_I2C.Pt = 0;

**if**(g\_tS3C24xx\_I2C.DataCount == 1)

                   IICCON = 0x2f;   // 恢复I2C传输，开始接收数据，接收到数据时不发出ACK

**else**

                   IICCON = 0xaf;   // 恢复I2C传输，开始接收数据

**break**;

            }

            g\_tS3C24xx\_I2C.pData[g\_tS3C24xx\_I2C.Pt++] = IICDS;

            g\_tS3C24xx\_I2C.DataCount--;

**if** (g\_tS3C24xx\_I2C.DataCount == 0)

            {

                // 下面两行恢复I2C操作，发出P信号

                IICSTAT = 0x90;

                IICCON  = 0xaf;

                Delay(10000);  // 等待一段时间以便P信号已经发出

**break**;

            }

**else**

            {

               // 接收最后一个数据时，不要发出ACK信号

**if**(g\_tS3C24xx\_I2C.DataCount == 1)

                   IICCON = 0x2f;   // 恢复I2C传输，接收到下一数据时无ACK

**else**

                   IICCON = 0xaf;   // 恢复I2C传输，接收到下一数据时发出ACK

            }

**break**;

        }

**default**:

**break**;

    }

}

/\*

 \* 主机发送

 \* slvAddr : 从机地址，buf : 数据存放的缓冲区，len : 数据长度

 \*/

**void** i2c\_write(unsigned **int** slvAddr, unsigned **char** \*buf, **int** len)

{

    g\_tS3C24xx\_I2C.Mode = WRDATA;   // 写操作

    g\_tS3C24xx\_I2C.Pt   = 0;        // 索引值初始为0

    g\_tS3C24xx\_I2C.pData = buf;     // 保存缓冲区地址

    g\_tS3C24xx\_I2C.DataCount = len; // 传输长度

    IICDS   = slvAddr;

    IICSTAT = 0xf0;         // 主机发送启动，发送完一字节数据会产生中断

    /\* 等待直至数据传输完毕 \*/

**while** (g\_tS3C24xx\_I2C.DataCount != -1);

}

/\*

 \* 主机接收

 \* slvAddr : 从机地址，buf : 数据存放的缓冲区，len : 数据长度

 \*/

**void** i2c\_read(unsigned **int** slvAddr, unsigned **char** \*buf, **int** len)

{

    g\_tS3C24xx\_I2C.Mode = RDDATA;   // 读操作

    g\_tS3C24xx\_I2C.Pt   = -1;       // 索引值初始化为-1，表示第1个中断时不接收数据(地址中断)

    g\_tS3C24xx\_I2C.pData = buf;     // 保存缓冲区地址

    g\_tS3C24xx\_I2C.DataCount = len; // 传输长度

    IICDS        = slvAddr;

    IICSTAT      = 0xb0;    // 主机接收，启动

    /\* 等待直至数据传输完毕 \*/

**while** (g\_tS3C24xx\_I2C.DataCount != 0);

}

## 3.3 2.6内核下的i2c驱动程序

### 3.3.1 i2c驱动框架

I2C 子系统要处理的问题主要有两个：

控制总线的 I2C 控制器和总线上的从机器件。

I2C 子系统一方面要驱动 I2C 控制器，以实现 I2C 总线上的通信；另一方面要使 I2C 总线上的从机器件能很好地工作起来。

注册设备就是把设备的某个结构体挂接到内核设备链表中。驱动挂接在驱动链表中。

注册驱动或者设备时内核都会进行设备和驱动的匹配,匹配成功调用驱动的probe函数。：

怎么匹配：名字

驱动分为三层：设备层，核心层和适配器层。

设备层知道数据含义，发什么。

核心层是协议相关，决定如何发送，时序。

适配器负责最终把数据发送出去。

i2c\_client :设备 🡪i2c\_driver

i2c\_adapter：🡪适配器，即i2c控制器

#### 设备驱动：

知道数据的含义

#### 总线驱动（厂家做好）

1、识别设备

2、提供实际的读写函数，知道怎么收发数据

共有四个对象：

设备client对象：设置描述对设备的结构体

设备驱动对象：

一个具体的I2C设备驱动有两部分组成，一部分是i2c\_driver,用于将设备挂接于I2C总线，一部分是设备本身的驱动

适配器adapter对象：s3c24xx\_i2c

I2C总线驱动：提供数据传输函数

#### 如何操作：

设备会对上层提供读写函数，读写函数会调用设备驱动，设备驱动调用adapter驱动进行通信。adapter和设备是绑定的。

#### i2c-s3c2410.c 总线驱动模块

#### 对应I2C adapter驱动结构

**static** **struct** s3c24xx\_i2c s3c24xx\_i2c = {

    .lock       = \_\_SPIN\_LOCK\_UNLOCKED(s3c24xx\_i2c.lock),

    .wait       = \_\_WAIT\_QUEUE\_HEAD\_INITIALIZER(s3c24xx\_i2c.wait),

    .tx\_setup   = 50,

    .adap       = { //adapter描述

        .name           = "s3c2410-i2c",

        .owner          = THIS\_MODULE,

        .algo           = &s3c24xx\_i2c\_algorithm,

        .retries        = 2,

        .**class**          = I2C\_CLASS\_HWMON,

    },

};

#### adapter 注册

platform\_driver\_register(&s3c2410\_i2c\_driver);             //注册了i2c适配器s3c2410\_i2c\_driver

    s3c24xx\_i2c\_probe(**struct** platform\_device \*pdev)      /\*注册驱动时如果发现同名的adapter就会调用该函数。 自动传入描述设备的结构体\*/

        i2c\_add\_adapter(&i2c->adap);

           i2c\_register\_adapter

#### eeprom.c I2C设备注册

 //注册驱动时会检测是否有匹配的设备，有则将相应的设备挂接到正确的adapter

i2c\_add\_driver(&eeprom\_driver); //i2c\_driver \*driver=&eeprom\_driver

    driver\_register(&driver->driver)//把设备驱动eeprom\_driver挂入链表

    list\_for\_each\_entry(adapter, &adapters, list) //遍历adapter链表中的所有已经注册的adapter

    driver->attach\_adapter(adapter);  //相当于调用eeprom\_attach\_adapter匹配eeprom和adapter

         eeprom\_attach\_adapter（eeprom\_driver）

            i2c\_probe(adapter, &addr\_data, eeprom\_detect)

                  i2c\_probe\_address //使用adapter里面的master\_xfer发送设备地址，如果能够收到ACK，标识发现一个支持的设备

                      i2c\_smbus\_xfer

                          i2c\_smbus\_xfer\_emulated

                              i2c\_transfer

                                  adap->algo->master\_xfer /\*master\_xfer 该函数即s3c24xx\_i2c\_xfer发信号确定是否有该I2C设备，如果有调用 eeprom\_detect。 found\_proc=eeprom\_detect\*/

                                      err = found\_proc(adapter, addr, kind);

                                               err = i2c\_attach\_client(new\_client)//将新设备挂接在dapter

### 3.3.2 i2c驱动编写

#### 添加一个i2c设备

mach-smdk2440.c添加AT24C02设备的板级信息如下：

**static** **struct** at24\_platform\_data at24c02= {

6.      .byte\_len   = SZ\_2K / 8,

7.      .page\_size  = 8,

8.      .flags      = AT24\_FLAG\_ADDR8,

9. };

10.

11.  **static** **struct** i2c\_board\_info \_\_initdata smdk2440\_i2c\_devs[] = {

12.      {

13.          I2C\_BOARD\_INFO("24c02", 0x50),

14.          .platform\_data = &at24c02,

15.      },

16.      /\*  more devices can be added using expansion connectors \*/

17. };

#### 在smdk2440\_machine\_init 函数增加如下：

**static** **void** \_\_init smdk2440\_machine\_init(**void**)

{

    s3c24xx\_fb\_set\_platdata(&smdk2440\_lcd\_cfg);

    platform\_add\_devices(smdk2440\_devices, ARRAY\_SIZE(smdk2440\_devices));

    /\*添加i2c设备\*/

    i2c\_register\_board\_info(0,smdk2440\_i2c\_devs,ARRAY\_SIZE(smdk2440\_i2c\_devs));

    smdk\_machine\_init();

}

#### 怎么写一个I2c驱动程序：

参照时钟芯片 Ds1374.c eeprom.c

1、分配一个i2c\_driver结构体

2、设置

attach\_adapter //直接调用i2c\_probe(adap,设备地址，发现设备后要调用的函数)

detach\_client //卸载这个驱动后，如果之前发现能够处理的设备，则调用它来清理

1. 注册

#### 2.6 版本下at24cxx 驱动程序

#include <linux/kernel.h>

#include <linux/init.h>

#include <linux/module.h>

#include <linux/slab.h>

#include <linux/jiffies.h>

#include <linux/i2c.h>

#include <linux/mutex.h>

#include <linux/fs.h>

#include <asm/uaccess.h>

static unsigned short ignore[]      = { I2C\_CLIENT\_END };

static unsigned short normal\_addr[] = { 0x50, I2C\_CLIENT\_END }; /\* 地址值是7位 0:7 \*/

       /\* 改为0x60的话, 由于不存在设备地址为0x60的设备, 所以at24cxx\_detect不被调用 \*/

 static unsigned short force\_addr[] = {ANY\_I2C\_BUS, 0x60, I2C\_CLIENT\_END};

static unsigned short \* forces[] = {force\_addr, NULL};

       static struct i2c\_client\_address\_data addr\_data = {

    .normal\_i2c = normal\_addr,  /\* 要发出S信号和设备地址并得到ACK信号,才能确定存在这个设备 \*/

    .probe      = ignore,

    .ignore     = ignore,

    //.forces     = forces, /\* 强制认为存在这个设备 \*/

    /\*

         forces[kind][i] 双字节标识一个设备，第一个字节标识连接的adapter编号，

         如果设置为ANY\_I2C\_BUS标识该可能设备挂接在所有的adapter上，第二个字节标识设备地址\*/

};

static struct i2c\_driver at24cxx\_driver;

 static int major;

static struct class \*cls;

struct i2c\_client \*at24cxx\_client;

static ssize\_t at24cxx\_read(struct file \*file, char \_\_user \*buf, size\_t size, loff\_t \* offset)

{

    unsigned char address;

    unsigned char data;

    struct i2c\_msg msg[2];

    int ret;

    /\* address = buf[0]

     \* data    = buf[1]

     \*/

**if** (size != 1)

**return** -EINVAL;

    copy\_from\_user(&address, buf, 1);

    /\* 数据传输三要素: 源,目的,长度 \*/

    /\* 读AT24CXX时,要先把要读的存储空间的地址发给它 \*/

    msg[0].addr  = at24cxx\_client->addr;  /\* 目的 \*/

    msg[0].buf   = &address;              /\* 源 \*/

    msg[0].len   = 1;                     /\* 地址=1 byte \*/

    msg[0].flags = 0;                     /\* 表示写 \*/

    /\* 然后启动读操作 \*/

    msg[1].addr  = at24cxx\_client->addr;  /\* 源 \*/

    msg[1].buf   = &data;                 /\* 目的 \*/

    msg[1].len   = 1;                     /\* 数据=1 byte \*/

    msg[1].flags = I2C\_M\_RD;                     /\* 表示读 \*/

    ret = i2c\_transfer(at24cxx\_client->adapter, msg, 2);

**if** (ret == 2)//成功传输两个字节

    {

        copy\_to\_user(buf, &data, 1);//返回读到的1字节数据

**return** 1;

    }

**else**

**return** -EIO;

}

static ssize\_t at24cxx\_write(struct file \*file, const char \_\_user \*buf, size\_t size, loff\_t \*offset)

{

    unsigned char val[2];

    struct i2c\_msg msg[1];

int ret;

 /\* address = buf[0]

 \* data    = buf[1]

     \*/

**if** (size != 2)

**return** -EINVAL;

    copy\_from\_user(val, buf, 2);

    /\* 数据传输三要素: 源,目的,长度 \*/

    msg[0].addr  = at24cxx\_client->addr;  /\* 目的 \*/

    msg[0].buf   = val;                   /\* 源 \*/

    msg[0].len   = 2;                     /\* 地址+数据=2 byte \*/

    msg[0].flags = 0;                     /\* 表示写 \*/

    ret = i2c\_transfer(at24cxx\_client->adapter, msg, 1);

**if** (ret == 1)

**return** 2;

**else**

**return** -EIO;

}

static struct file\_operations at24cxx\_fops = {

    .owner = THIS\_MODULE,

    .read  = at24cxx\_read,

    .write = at24cxx\_write,

};

static int at24cxx\_detect(struct i2c\_adapter \*adapter, int address, int kind)

{

    printk("at24cxx\_detect\n");

    /\* 构构一个i2c\_client结构体: 以后收改数据时会用到它 \*/

    at24cxx\_client = kzalloc(sizeof(struct i2c\_client), GFP\_KERNEL);

    at24cxx\_client->addr    = address;

    at24cxx\_client->adapter = adapter;

    at24cxx\_client->driver  = &at24cxx\_driver;

    strcpy(at24cxx\_client->name, "at24cxx");

    i2c\_attach\_client(at24cxx\_client);

    major = register\_chrdev(0, "at24cxx", &at24cxx\_fops);

    cls = class\_create(THIS\_MODULE, "at24cxx");

    class\_device\_create(cls, NULL, MKDEV(major, 0), NULL, "at24cxx"); /\* /dev/at24cxx \*/

**return** 0;

}

static int at24cxx\_attach(struct i2c\_adapter \*adapter)

{

**return** i2c\_probe(adapter, &addr\_data, at24cxx\_detect);

}

static int at24cxx\_detach(struct i2c\_client \*client)

{

    printk("at24cxx\_detach\n");

    class\_device\_destroy(cls, MKDEV(major, 0));

    class\_destroy(cls);

    unregister\_chrdev(major, "at24cxx");

    i2c\_detach\_client(client);

    kfree(i2c\_get\_clientdata(client));

**return** 0;

}

/\* 1. 分配一个i2c\_driver结构体 \*/

/\* 2. 设置i2c\_driver结构体 \*/

static struct i2c\_driver at24cxx\_driver = {

    .driver = {

        .name   = "at24cxx",

    },

    .attach\_adapter = at24cxx\_attach,

    .detach\_client  = at24cxx\_detach,

};

static int at24cxx\_init(void)

{

    i2c\_add\_driver(&at24cxx\_driver);

**return** 0;

}

static void at24cxx\_exit(void)

{

    i2c\_del\_driver(&at24cxx\_driver);

}

module\_init(at24cxx\_init);

module\_exit(at24cxx\_exit);

MODULE\_LICENSE("GPL");

## 3.4 Linux 3.4.2内核下的i2c驱动程序设计

### 3.4.1 I2C驱动架构

I2C驱动组成：I2C 核心、I2C总线驱动和I2C设备驱动。

(1)I2c核心：提供了I2c总线驱动和设备驱动的注册注销方法，I2c协议相关的通信方法及探

测设备、检测设备地址的方法。

(2)I2c总线驱动控制适配器以主控方式产生开始位、停止位、读写周期、产生ACK等

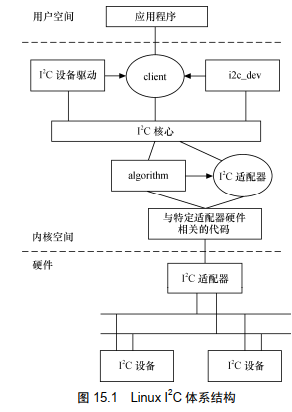
(3)I2c设备驱动提供设备读写函数。

I2C设备和驱动关系：

I2C设备被称为client,设备驱动是i2c\_driver。

I2C控制器被称为adapter，即适配器，适配器驱动被称为总线驱动，属于platform\_driver。

i2c0和i2c1即是适配器的名字。数字表示是那条i2c总线



内核I2c驱动目录

(1) i2c-core.c

实现了I2c核心功能

(2) i2c-dev.c

通用的I2c设备驱动，主设备号89。

(3)busses

适配器驱动程序，i2c-s3c2410.c

(4)algos

协议相关的通信方法。设备驱动调用该文件内部i2c传输函数，这些函数内部最终调用adapter的收发函数。

驱动和设备如何匹配：

设备和驱动都挂接在i2c\_bus\_type总线上。使用总线匹配函数匹配，匹配后调用i2c\_device\_probe函数

struct bus\_type i2c\_bus\_type = {

    .name       = "i2c",

    .match      = i2c\_device\_match,

    .probe      = i2c\_device\_probe,

    .remove     = i2c\_device\_remove,

    .shutdown   = i2c\_device\_shutdown,

    .pm     = &i2c\_device\_pm\_ops,

};

static int i2c\_device\_match(struct device \*dev, struct device\_driver \*drv)

{

    struct i2c\_client  \*client = i2c\_verify\_client(dev);

    struct i2c\_driver  \*driver;

**if** (!client)

**return** 0;

    /\* Attempt an OF style match \*/

**if** (of\_driver\_match\_device(dev, drv))

**return** 1;

    driver = to\_i2c\_driver(drv);

    /\* match on an id table if there is one主要匹配table表中的名字 \*/

**if** (driver->id\_table)

**return** i2c\_match\_id(driver->id\_table, client) != NULL;

**return** 0;

}

### 3.4.2 I2C clent的四种注册方法

#### 第一种方法：构造i2c\_board\_info

static struct at24\_platform\_data at24c08 = {

.byte\_len = SZ\_8K / 8,

.page\_size = 16,

};

static struct i2c\_board\_info mini2440\_i2c\_devs[] \_\_initdata = {

{

I2C\_BOARD\_INFO("24c08", 0x50),

.platform\_data = &at24c08,

},

};

然后i2c\_register\_board\_info(0, mini2440\_i2c\_devs,ARRAY\_SIZE(mini2440\_i2c\_devs));

(把它们放入\_\_i2c\_board\_list链表)

list\_add\_tail(&devinfo->list, &\_\_i2c\_board\_list);

链表何时使用：

i2c\_register\_adapter

i2c\_scan\_static\_board\_info

i2c\_new\_device

使用限制：必须在 i2c\_register\_adapter 之前 i2c\_register\_board\_info所以：不适合我们动态加载insmod

#### 第二种方法：直接 i2c\_new\_device()

会认为设备一定存在，不管是否真的存在

at24cxx\_drv.c

static int \_\_devinit  at24cxx\_probe(struct i2c\_client \*client,

const struct i2c\_device\_id \*id)

{

("%s %s %d\n", \_\_FILE\_\_, \_\_FUNCTION\_\_, \_\_LINE\_\_);

**return** 0;

}

static int \_\_devexit at24cxx\_remove(struct i2c\_client \*client)

{

printk("%s %s %d\n", \_\_FILE\_\_, \_\_FUNCTION\_\_, \_\_LINE\_\_);

**return** 0;

}

//adapter设备链表里面如果有设备名字和id\_table相同，调用probe

static const struct i2c\_device\_id at24cxx\_id\_table[] = {

{ "at24c08", 0 },

{}

};

static struct i2c\_driver at24cxx\_driver = {

.driver = {

.name = "100ask",

.owner = THIS\_MODULE,

},

.probe = at24cxx\_probe,

.remove = \_\_devexit\_p(at24cxx\_remove),

.id\_table = at24cxx\_id\_table,

};

static int at24cxx\_drv\_init(void)

{

/\* 2. 注册i2c\_driver \*/

i2c\_add\_driver(&at24cxx\_driver);//有匹配设备调用probe函数

**return** 0;

}

at24cxx\_dev.c

static struct i2c\_board\_info at24cxx\_info = {

I2C\_BOARD\_INFO("at24c08", 0x50),

};

static struct i2c\_client \*at24cxx\_client;

static int at24cxx\_dev\_init(void)

{

struct i2c\_adapter \*i2c\_adap;

i2c\_adap = i2c\_get\_adapter(0);//直接将设备挂接在adapter0上认为设备肯定存在

at24cxx\_client = i2c\_new\_device(i2c\_adap, &at24cxx\_info);

i2c\_put\_adapter(i2c\_adap);

**return** 0;

}

#### 第三种方法：直接 i2c\_new\_probed\_device()

对于"已经识别出来的设备"(probed\_device)，接收到ack才会创建设备

at24cxx\_drv.c

static int \_\_devinit at24cxx\_probe(struct i2c\_client \*client,

                  const struct i2c\_device\_id \*id)

{

    printk("%s %s %d\n", \_\_FILE\_\_, \_\_FUNCTION\_\_, \_\_LINE\_\_);

**return** 0;

}

static int \_\_devexit at24cxx\_remove(struct i2c\_client \*client)

{

    printk("%s %s %d\n", \_\_FILE\_\_, \_\_FUNCTION\_\_, \_\_LINE\_\_);

**return** 0;

}

static const struct i2c\_device\_id at24cxx\_id\_table[] = {

    { "at24c08", 0 },

    {}

};

/\* 1. 分配/设置i2c\_driver \*/

static struct i2c\_driver at24cxx\_driver = {

    .driver = {

        .name   = "100ask",

        .owner  = THIS\_MODULE,

    },

    .probe      = at24cxx\_probe,

    .remove     = \_\_devexit\_p(at24cxx\_remove),

    .id\_table   = at24cxx\_id\_table,

};

static int at24cxx\_drv\_init(void)

{

    /\* 2. 注册i2c\_driver \*/

    i2c\_add\_driver(&at24cxx\_driver);

**return** 0;

}

static void at24cxx\_drv\_exit(void)

{

    i2c\_del\_driver(&at24cxx\_driver);

}

  at24cxx\_dev.c

static struct i2c\_client \*at24cxx\_client;

static const unsigned short addr\_list[] = { 0x60, 0x50, I2C\_CLIENT\_END };

static int at24cxx\_dev\_init(void)

{

    struct i2c\_adapter \*i2c\_adap;

    struct i2c\_board\_info at24cxx\_info;

    memset(&at24cxx\_info, 0, sizeof(struct i2c\_board\_info));

    strlcpy(at24cxx\_info.type, "at24c08", I2C\_NAME\_SIZE);

    i2c\_adap = i2c\_get\_adapter(0);

    at24cxx\_client = i2c\_new\_probed\_device(i2c\_adap, &at24cxx\_info, addr\_list, NULL);

    i2c\_put\_adapter(i2c\_adap);

**if** (at24cxx\_client)

**return** 0;

**else**

**return** -ENODEV;

}

static void at24cxx\_dev\_exit(void)

{

    i2c\_unregister\_device(at24cxx\_client);

}

  //如何检测设备是否存在

i2c\_new\_probed\_device(\*adapter,\*i2c\_board\_info,\*addr\_list,probe(\*adapter,addr))

probe = i2c\_default\_probe;//设置默认的probe函数

         i2c\_smbus\_xfer() //检测地址是否真实存在

probe(adap, addr\_list[i]) /\* 调用probe确定设备是否真实存在,有默认的probe函数 \*/

info->addr = addr\_list[i];

i2c\_new\_device(adap, info);

#### 第四种方法：从用户空间创建设备

/ # cd /sys/class/i2c-adapter/

/sys/class/i2c-adapter # ls

i2c-0 //查看适配器

/sys/class/i2c-adapter # ls i2c-0/

delete\_device name power uevent

device new\_device subsystem

创建设备：

echo at24c08 0x50 > /sys/class/i2c-adapter/i2c-0/new\_device

导致i2c\_new\_device被调用

删除设备

echo 0x50 > /sys/class/i2c-adapter/i2c-0/delete\_device

导致i2c\_unregister\_device

加载驱动程序检测到设备。

#### 第五种方法：不确定设备会连接那个adapter

前面3中方法需要事先确定连接的适配器。如果我事先并不知道这个I2C设备在哪个适配器上，怎么办？

答：去class表示的所有的适配器上查找

有一些I2C设备的地址是一样，如果同一地址挂接不同设备，怎么继续分配它是哪一款？

答：用detect函数

去"class表示的这一类"I2C适配器，用"detect函数"来确定能否找到"address\_list里的设备",

如果能找到就调用i2c\_new\_device来注册i2c\_client, 这会和i2c\_driver的id\_table比较，

如果匹配，调用probe

at24cxx\_drv.c

static int \_\_devinit at24cxx\_probe(struct i2c\_client \*client,

const struct i2c\_device\_id \*id)

{

printk("%s %s %d\n", \_\_FILE\_\_, \_\_FUNCTION\_\_, \_\_LINE\_\_);

**return** 0;

}

static int \_\_devexit at24cxx\_remove(struct i2c\_client \*client)

{

printk("%s %s %d\n", \_\_FILE\_\_, \_\_FUNCTION\_\_, \_\_LINE\_\_);

**return** 0;

}

static const struct i2c\_device\_id at24cxx\_id\_table[] = {

{ "at24c08", 0 },

{}

};

static int at24cxx\_detect(struct i2c\_client \*client,struct i2c\_board\_info \*info)

{

/\* 能运行到这里, 表示该addr的设备是存在的

\* 但是有些设备单凭地址无法分辨(A芯片的地址是0x50, B芯片的地址也是0x50)

\* 还需要进一步读写I2C设备来分辨是哪款芯片，比如at24cxx，容量不同设备地址可以一样。

\* detect就是用来进一步分辨这个芯片是哪一款，并且设置info->type

\*/

printk("at24cxx\_detect : addr = 0x%x\n", client->addr);

/\* 实际程序种需要进一步判断是哪一款 \*/

strlcpy(info->type, "at24c08", I2C\_NAME\_SIZE);

**return** 0;

}

static const unsigned short addr\_list[] = { 0x60, 0x50, I2C\_CLIENT\_END };

/\* 1. 分配/设置i2c\_driver \*/

static struct i2c\_driver at24cxx\_driver = {

.class = I2C\_CLASS\_HWMON, /\* 表示去哪些适配器上找设备 \*/

.driver = {

.name = "100ask",

.owner = THIS\_MODULE,

},

.probe = at24cxx\_probe,

.remove = \_\_devexit\_p(at24cxx\_remove),

.id\_table = at24cxx\_id\_table,

.detect = at24cxx\_detect, /\* 用这个函数来检测设备确实存在 \*/

.address\_list = addr\_list, /\* 这些设备的地址 \*/

};

static int at24cxx\_drv\_init(void)

{

/\* 2. 注册i2c\_driver \*/

i2c\_add\_driver(&at24cxx\_driver);

**return** 0;

}

#### 注册设备的流程：

i2c\_add\_driver

i2c\_register\_driver

driver\_register()

/\* a 、driver\_register()里面会把 at24cxx\_driver放入i2c\_bus\_type的drv链表，并且从dev链表里取出i2c\_client，如果能匹配就调用probe\*/

/\*b. 对于每一个适配器，调用\_\_process\_new\_driver

对于每一个适配器，调用它的函数确定address\_list里的设备是否存在

如果存在，再调用detect进一步确定、设置，然后i2c\_new\_device\*/

/\* 遍历所有的adapter调用 \_\_process\_new\_driver \*/

i2c\_add\_driver

i2c\_register\_driver

driver\_register()

i2c\_for\_each\_dev(driver, \_\_process\_new\_driver);

\_\_process\_new\_driver

i2c\_do\_add\_adapter

i2c\_detect(adap, driver);//检测支持的设备

**for** (i = 0; address\_list[i] != I2C\_CLIENT\_END; i += 1)

err = i2c\_detect\_address(temp\_client, driver);

/\* 判断这个设备是否存在：简单的发出S信号确定有ACK \*/

**if** (!i2c\_default\_probe(adapter, addr))

**return** 0;

memset(&info, 0, sizeof(struct i2c\_board\_info));

info.addr = addr;

err = driver->detect(temp\_client, &info);

i2c\_new\_device

该方法应该最后使用

### 3.4.3 采用第二种方法编写完整的驱动

at24cxx\_dev.c

#include <linux/kernel.h>

#include <linux/module.h>

#include <linux/platform\_device.h>

#include <linux/i2c.h>

#include <linux/err.h>

#include <linux/regmap.h>

#include <linux/slab.h>

static struct i2c\_board\_info at24cxx\_info = {

    I2C\_BOARD\_INFO("at24c08", 0x50),

};

static struct i2c\_client \*at24cxx\_client;

static int at24cxx\_dev\_init(void)

{

    struct i2c\_adapter \*i2c\_adap;

    i2c\_adap = i2c\_get\_adapter(0);

    at24cxx\_client = i2c\_new\_device(i2c\_adap, &at24cxx\_info);

    i2c\_put\_adapter(i2c\_adap);

**return** 0;

}

static void at24cxx\_dev\_exit(void)

{

    i2c\_unregister\_device(at24cxx\_client);

}

module\_init(at24cxx\_dev\_init);

module\_exit(at24cxx\_dev\_exit);

MODULE\_LICENSE("GPL");

at24cxx\_drv.c

#include <linux/kernel.h>

#include <linux/module.h>

#include <linux/platform\_device.h>

#include <linux/i2c.h>

#include <linux/err.h>

#include <linux/regmap.h>

#include <linux/slab.h>

#include <linux/fs.h>

#include <asm/uaccess.h>

static int major;

static struct class \*class;

static struct i2c\_client \*at24cxx\_client;

/\* 传入: buf[0] : addr

 \* 输出: buf[0] : data

 \*/

static ssize\_t at24cxx\_read(struct file \* file, char \_\_user \*buf, size\_t count, loff\_t \*off)

{

    unsigned char addr, data;

        copy\_from\_user(&addr, buf, 1);

/\*E:\kernel\code\linux3.4.2\linux-3.4.2\Documentation\i2c\smbus-protocol中推荐的读写方法

smbus是i2c子集，尽量使用该协议

i2c\_smbus\_read\_byte\_data(const struct i2c\_client \*client,u8 command);

SMBus Read Byte: i2c\_smbus\_read\_byte\_data()

This reads a single byte from a device,

S Addr Wr [A] Comm [A] S Addr Rd [A] [Data] NA P

\*/

    data = i2c\_smbus\_read\_byte\_data(at24cxx\_client, addr);

    copy\_to\_user(buf, &data, 1);

**return** 1;

}

 /\* buf[0] : addr

 \* buf[1] : data

 \*/

static ssize\_t at24cxx\_write(struct file \*file, const char \_\_user \*buf, size\_t count, loff\_t \*off)

{

    unsigned char ker\_buf[2];

    unsigned char addr, data;

    copy\_from\_user(ker\_buf, buf, 2);

    addr = ker\_buf[0];

    data = ker\_buf[1];

printk("addr = 0x%02x, data = 0x%02x\n", addr, data);

/\*

SMBus Write Byte: i2c\_smbus\_write\_byte\_data()

i2c\_smbus\_write\_byte\_data(const struct i2c\_client \*client,

u8 command, u8 value);

This writes a single byte to a device,Comm作为数据地址

S Addr Wr [A] Comm [A] Data [A] P

\*/

**if** (!i2c\_smbus\_write\_byte\_data(at24cxx\_client, addr, data))

**return** 2;

**else**

**return** -EIO;

}

static struct file\_operations at24cxx\_fops = {

    .owner = THIS\_MODULE,

    .read  = at24cxx\_read,

    .write = at24cxx\_write,

};

static int \_\_devinit at24cxx\_probe(struct i2c\_client \*client,

                  const struct i2c\_device\_id \*id)

{

    at24cxx\_client = client;

    //printk("%s %s %d\n", \_\_FILE\_\_, \_\_FUNCTION\_\_, \_\_LINE\_\_);

    major = register\_chrdev(0, "at24cxx", &at24cxx\_fops);

    class = class\_create(THIS\_MODULE, "at24cxx");

    device\_create(class, NULL, MKDEV(major, 0), NULL, "at24cxx"); /\* /dev/at24cxx \*/

**return** 0;

}

static int \_\_devexit at24cxx\_remove(struct i2c\_client \*client)

{

    //printk("%s %s %d\n", \_\_FILE\_\_, \_\_FUNCTION\_\_, \_\_LINE\_\_);

    device\_destroy(class, MKDEV(major, 0));

    class\_destroy(class);

    unregister\_chrdev(major, "at24cxx");

**return** 0;

}

static const struct i2c\_device\_id at24cxx\_id\_table[] = {

    { "at24c08", 0 },

    {}

};

/\* 1. 分配/设置i2c\_driver \*/

static struct i2c\_driver at24cxx\_driver = {

    .driver = {

        .name   = "100ask",

        .owner  = THIS\_MODULE,

    },

    .probe      = at24cxx\_probe,

    .remove     = \_\_devexit\_p(at24cxx\_remove),

    .id\_table   = at24cxx\_id\_table,

};

static int at24cxx\_drv\_init(void)

{

    /\* 2. 注册i2c\_driver \*/

    i2c\_add\_driver(&at24cxx\_driver);

**return** 0;

}

static void at24cxx\_drv\_exit(void)

{

    i2c\_del\_driver(&at24cxx\_driver);

}

module\_init(at24cxx\_drv\_init);

module\_exit(at24cxx\_drv\_exit);

MODULE\_LICENSE("GPL");

#### 大容量的at24c256驱动读写函数：

该芯片需要两字节数据地址。

static ssize\_t at24cxx\_read(struct file \* file, char \_\_user \*buf, size\_t count, loff\_t \*off)

{

   unsigned char address[2];

    unsigned char data;

    struct i2c\_msg msg[2];

    int ret;

    /\* addres[0] = buf[0]

       addres[1] = buf[1]

     \* data    = buf[2]

     \*/

**if** (count != 1)

**return** -EINVAL;

    copy\_from\_user(&address, buf, 2);

    /\* 数据传输三要素: 源,目的,长度 \*/

    /\* 读AT24CXX时,要先把要读的存储空间的地址发给它 \*/

    msg[0].addr  = at24cxx\_client->addr;  /\* 目的 \*/

    msg[0].buf   = &address;              /\* 源 \*/

    msg[0].len   = 2;                     /\* 地址=2 byte \*/

    msg[0].flags = 0;                     /\* 表示写 \*/

    /\* 然后启动读操作 \*/

    msg[1].addr  = at24cxx\_client->addr;  /\* 源 \*/

    msg[1].buf   = &data;                 /\* 目的 \*/

    msg[1].len   = 1;                     /\* 数据=1 byte \*/

    msg[1].flags = I2C\_M\_RD;                     /\* 表示读 \*/

    ret = i2c\_transfer(at24cxx\_client->adapter, msg, 2);

**if** (ret == 2)//成功传输两条msg

    {

        copy\_to\_user(buf, &data, 1);//返回读到的1字节数据

**return** 1;

    }

**else**

**return** -EIO;

}

/\* buf[0] : addr

 \* buf[1] : data

 \*/

static ssize\_t at24cxx\_write(struct file \*file, const char \_\_user \*buf, size\_t count, loff\_t \*off)

{

    unsigned char val[3];

      struct i2c\_msg msg[1];

    int ret;

   /\* address[0] = buf[0]

       address[1] = buf[1]

     \* data    = buf[2]

     \*/

**if** (count!= 3)

**return** -EINVAL;

    copy\_from\_user(val, buf, 3);

    /\* 数据传输三要素: 源,目的,长度 \*/

    msg[0].addr  = at24cxx\_client->addr;  /\* 目的 \*/

    msg[0].buf   = val;                   /\* 源 \*/

    msg[0].len   = 3;                     /\* 地址+数据=3 byte \*/

    msg[0].flags = 0;                     /\* 表示写 \*/

    ret = i2c\_transfer(at24cxx\_client->adapter, msg, 1);

**if** (ret == 1)

**return** 3;

**else**

**return** -EIO;

}

#### 完整的测试程序

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

/\* i2c\_test r addr

 \* i2c\_test w addr val\*/

void print\_usage(char \*file,int argc)

{

    printf("%s r addrl addrh \n", file);

    printf("%s w addrl addrh val\n", file);

    printf("%d \n",argc);

}

int main(int argc, char \*\*argv)

{

    int fd;

    unsigned char buf[3];

**if** ((argc != 4) && (argc != 5))

     {

        print\_usage(argv[0],argc);

**return** -1;

    }

    fd = open("/dev/at24cxx", O\_RDWR);

**if** (fd < 0)

    {

        printf("can't open /dev/at24cxx\n");

**return** -1;

    }

**if** (strcmp(argv[1], "r") == 0)

     {

        buf[0] = strtoul(argv[2], NULL, 0);

        buf[1] = strtoul(argv[3], NULL, 0);

        read(fd, buf, 1);

        printf("data:%d\n", buf[0]);

    }

**else** **if** ((strcmp(argv[1], "w") == 0) && (argc == 5))

    {

        buf[0] = strtoul(argv[2], NULL, 0);

        buf[1] = strtoul(argv[3], NULL, 0);

        buf[2] = strtoul(argv[4], NULL, 0);

**if** (write(fd, buf, 3) != 3)

            printf("write err, addrl = 0x%02x, addrh = 0x%02x,data = 0x%02x\n", buf[0], buf[1],buf[2]);

    }

**else**

    {

        print\_usage(argv[0],argc);

**return** -1;

    }

**return** 0;

}

### 3.4.4 使用内核自带驱动程序

采用内核中的i2c-dev.c设备驱动程序。

int main(int argc, char \*\*argv)

{

    int fd;

    unsigned char addr, data;

    int dev\_addr;

**if** ((argc != 5) && (argc != 6))

    {

        print\_usage(argv[0]);

**return** -1;

    }

    fd = open(argv[1], O\_RDWR);//打开设备

**if** (fd < 0)

    {

        printf("can't open %s\n", argv[1]);

**return** -1;

    }

    dev\_addr = strtoul(argv[2], NULL, 0);//设置从机地址

**if** (ioctl(fd, I2C\_SLAVE, dev\_addr) < 0)

     {

         /\* ERROR HANDLING; you can check errno to see what went wrong \*/

        printf("set addr error!\n");

**return** -1;

    }

**if** (strcmp(argv[3], "r") == 0)

    {

        addr = strtoul(argv[4], NULL, 0);

        data = i2c\_smbus\_read\_word\_data(fd, addr);

        printf("data: %c, %d, 0x%2x\n", data, data, data);

    }

**else** **if** ((strcmp(argv[3], "w") == 0) && (argc == 6))

     {

        addr = strtoul(argv[4], NULL, 0);

        data = strtoul(argv[5], NULL, 0);

         i2c\_smbus\_write\_byte\_data(fd, addr, data);

    }

**else**

    {

        print\_usage(argv[0]);

**return** -1;

    }

**return** 0;

}

## 3.5 i2c总线驱动程序

i2c-s3c2410.c

static struct platform\_driver s3c24xx\_i2c\_driver = {

.probe = s3c24xx\_i2c\_probe,

.remove = s3c24xx\_i2c\_remove,

.id\_table = s3c24xx\_driver\_ids,

.driver = {

.owner = THIS\_MODULE,

.name = "s3c-i2c",

.pm = S3C24XX\_DEV\_PM\_OPS,

.of\_match\_table = s3c24xx\_i2c\_match,//支持设备树

},

};

static const struct i2c\_algorithm s3c24xx\_i2c\_algorithm = {

.master\_xfer = s3c24xx\_i2c\_xfer,

.functionality = s3c24xx\_i2c\_func,

};

platform\_driver\_register(&s3c24xx\_i2c\_driver);

s3c24xx\_i2c\_probe //有匹配设备时调用

i2c->adap.algo = &s3c24xx\_i2c\_algorithm;

i2c\_add\_numbered\_adapter

#### 读写调用过程

i2c\_smbus\_read\_byte\_data

i2c\_smbus\_xfer

i2c\_smbus\_xfer\_emulated

i2c\_transfer

adap->algo->master\_xfer

i2c\_add\_numbered\_adapter//直接挂接在确定的总线号上

#### 使用控制器驱动程序添加如下

s3c\_i2c0\_set\_platdata(NULL);//注册设备要先调用该函数

pd->bus\_num = 0; //设置adapter的bus number 是0

//mach-smdk2440.c

static struct platform\_device \*smdk2440\_devices[] \_\_initdata = {

&s3c\_device\_ohci,

&s3c\_device\_lcd,

&s3c\_device\_wdt,

&s3c\_device\_i2c0,

&s3c\_device\_iis,

&smdk2440\_device\_eth,

&s3c\_device\_spi0,

&s3c\_device\_spi1,

};

static void \_\_init smdk2440\_machine\_init(void)

{

s3c24xx\_fb\_set\_platdata(&smdk2440\_fb\_info);

s3c\_i2c0\_set\_platdata(NULL);

platform\_add\_devices(smdk2440\_devices, ARRAY\_SIZE(smdk2440\_devices));

smdk\_machine\_init();

}

# 4 网卡驱动程序

#### 四层网络模型：

**1.链路层（数据链路层/网络接口层）**：包括操作系统中的设备驱动程序、计算机中对应的网络接口卡

**2.网络层（互联网层）**：处理分组在网络中的活动，比如分组的选路。

**3.运输层**：主要为两台主机上的应用提供端到端的通信。

**4.应用层**：负责处理特定的应用程序细节。

## 4.1 网卡驱动框架

**APP**

**TCP层**

**IP层**

**发送(hard\_start\_xmit) 接收 netif\_rx(sk\_buff)**

**网卡驱动程序**

**网卡硬件**

1.分配 net\_device结构体 ndev=alloc\_etherdev(sizeof (struct board\_info));

2.设置

提供发包函数和收包程序

3.注册 register\_netdev(ndev)

4.硬件相关操作

init\_module

dmfe\_probe

struct net\_device \*dev;

dev=alloc\_etherdev(sizeof(struct board\_info)); //分配net\_device结构体

dmfe\_probe1(dev);//设置

register\_netdev(dev)//注册

#### 如何发送数据：

网络层调用hard\_start\_xmit(发包)，

#### 如何接收数据：

1、收到数据包硬件产生中断dm9000\_interrupt

2、调用dm9000\_rx接收函数

3、分配一个dev\_alloc\_skb 缓冲区接收数据sk\_buf

4、上报收到的数据给网络层 netif\_rx(skb);

测试1th/2th:

1. insmod virt\_net.ko

2. ifconfig vnet0 3.3.3.3

ifconfig // 查看

3. ping 3.3.3.3 // ping自己不发送数据，不调用硬件收发函数。

ping 3.3.3.4

无发送

ping 3.3.3.3 无发送函数也可以通过。

ping 3.3.3.4 有发送函数才能通过，

#### 数据包格式：

目的Mac,源Mac,type,目的IP，源IP,数据

## 4.2 虚拟网卡驱动

/\*

 \* 参考 drivers\net\cs89x0.c

 \*/

static struct net\_device \*vnet\_dev;

static void emulator\_rx\_packet(struct sk\_buff \*skb, struct net\_device \*dev)

{

    /\* 参考LDD3 \*/

    unsigned char \*type;

    struct iphdr \*ih;

    \_\_be32 \*saddr, \*daddr, tmp;

    unsigned char  tmp\_dev\_addr[ETH\_ALEN];

    struct ethhdr \*ethhdr;

     struct sk\_buff \*rx\_skb;

         // 从硬件读出/保存数据

    /\* 对调"源/目的"的mac地址 \*/

    ethhdr = (struct ethhdr \*)skb->data;

    memcpy(tmp\_dev\_addr, ethhdr->h\_dest, ETH\_ALEN);

    memcpy(ethhdr->h\_dest, ethhdr->h\_source, ETH\_ALEN);

    memcpy(ethhdr->h\_source, tmp\_dev\_addr, ETH\_ALEN);

    /\* 对调"源/目的"的ip地址 \*/

    ih = (struct iphdr \*)(skb->data + sizeof(struct ethhdr));

    saddr = &ih->saddr;

    daddr = &ih->daddr;

    tmp = \*saddr;

    \*saddr = \*daddr;

    \*daddr = tmp;

      //((u8 \*)saddr)[2] ^= 1; /\* change the third octet (class C) \*/

    //((u8 \*)daddr)[2] ^= 1;

    type = skb->data + sizeof(struct ethhdr) + sizeof(struct iphdr);

    //printk("tx package type = %02x\n", \*type);

    // 修改类型, 原来0x8表示ping

    \*type = 0; /\* 0表示reply \*/

      ih->check = 0;        /\* and rebuild the checksum (ip needs it) \*/

    ih->check = ip\_fast\_csum((unsigned char \*)ih,ih->ihl);//计算校验码

       // 构造一个sk\_buff

    rx\_skb = dev\_alloc\_skb(skb->len + 2);

    skb\_reserve(rx\_skb, 2);  /\* align IP on 16B boundary \*/

    memcpy(skb\_put(rx\_skb, skb->len), skb->data, skb->len);

    /\* Write metadata, and then pass to the receive level \*/

    rx\_skb->dev = dev;

    rx\_skb->protocol = eth\_type\_trans(rx\_skb, dev);

    rx\_skb->ip\_summed = CHECKSUM\_UNNECESSARY; /\* don't check it \*/

    dev->stats.rx\_packets++;

    dev->stats.rx\_bytes += skb->len;

    // 提交sk\_buff

    netif\_rx(rx\_skb);

}

static int virt\_net\_send\_packet(struct sk\_buff \*skb, struct net\_device \*dev)

{

    static int cnt = 0;

    printk("virt\_net\_send\_packet cnt = %d\n", ++cnt);

    /\* 对于真实的网卡, 把skb里的数据通过网卡发送出去 \*/

    netif\_stop\_queue(dev); /\* 停止该网卡的队列 \*/

    /\* ...... 数据写入网卡函数\*/           /\* 把skb的数据写入网卡 \*/

    /\* 构造一个假的sk\_buff,上报 发送成功\*/

    emulator\_rx\_packet(skb, dev);

    dev\_kfree\_skb (skb);   /\* 释放skb \*/

    netif\_wake\_queue(dev); /\* 数据全部发送出去后,唤醒网卡的队列，实际网卡发送完毕产生中断 \*/

    /\* 更新统计信息 \*/

    dev->stats.tx\_packets++;

    dev->stats.tx\_bytes += skb->len;

**return** 0;

}

static int virt\_net\_init(void)

{

    /\* 1. 分配一个net\_device结构体 \*/

    vnet\_dev = alloc\_netdev(0, "vnet%d", ether\_setup);;  /\* alloc\_etherdev \*/

    /\* 2. 设置 \*/

    vnet\_dev->hard\_start\_xmit = virt\_net\_send\_packet;

    /\* 设置MAC地址 \*/

    vnet\_dev->dev\_addr[0] = 0x08;

    vnet\_dev->dev\_addr[1] = 0x89;

    vnet\_dev->dev\_addr[2] = 0x89;

    vnet\_dev->dev\_addr[3] = 0x89;

    vnet\_dev->dev\_addr[4] = 0x89;

    vnet\_dev->dev\_addr[5] = 0x11;

    /\* 设置下面两项才能ping通 \*/

    vnet\_dev->flags           |= IFF\_NOARP;

    vnet\_dev->features        |= NETIF\_F\_NO\_CSUM;

    /\* 3. 注册 \*/

    //register\_netdevice(vnet\_dev);

    register\_netdev(vnet\_dev);

**return** 0;

}

## 4.3 DM9K驱动移植

内存接口芯片。

片选:NGCS4 为低电平时选中芯片，cpu发送的地址在片选地址空间内，该引脚自动变低。

#### 需要修改地方：

1、内存基地址

2、位宽

3、中断号

4、时序设置

int \_\_init dm9000c\_init(void)

{

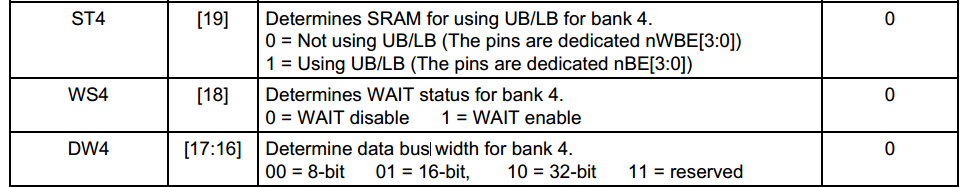
    volatile unsigned long \*bwscon; // 0x48000000

    volatile unsigned long \*bankcon4; // 0x48000014

    unsigned long val;

    iobase = (int)ioremap(0x20000000, 1024);  /\* 修改Iobase，基地址 \*/

    irq    = IRQ\_EINT7;                  /\* 中断号 \*/

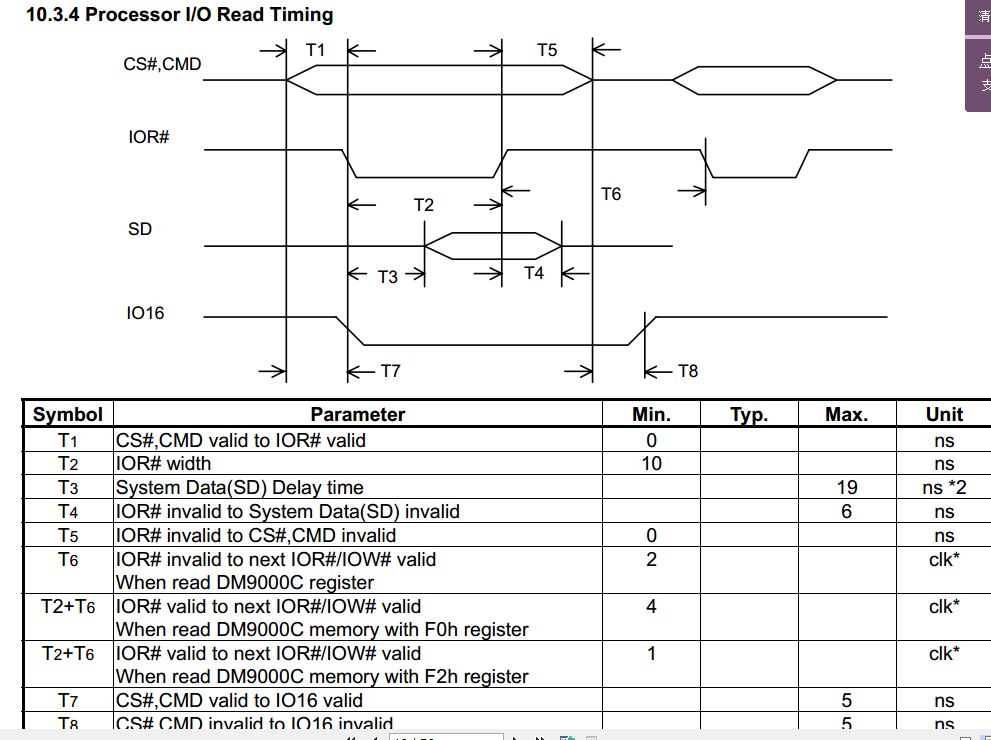


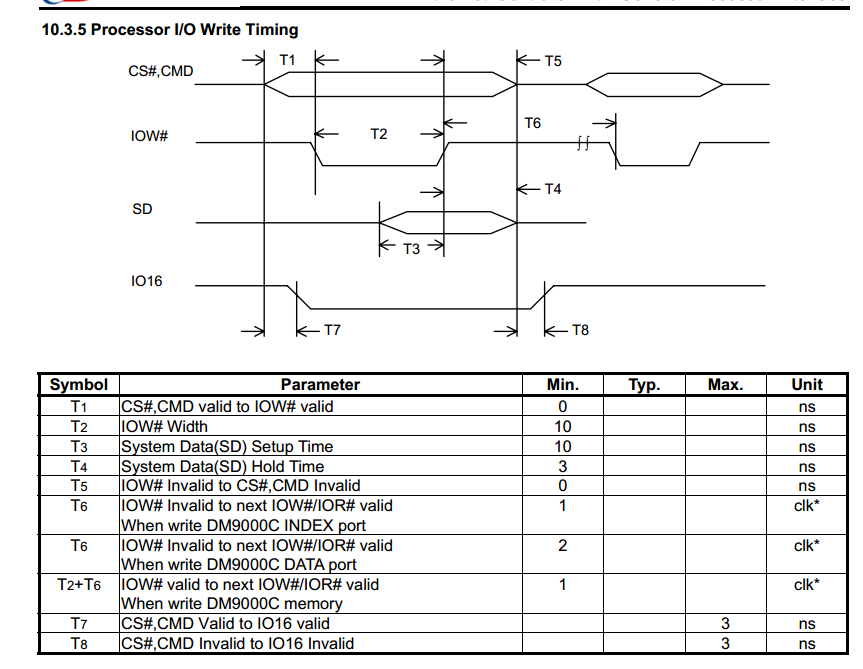
#### dm9k引脚：

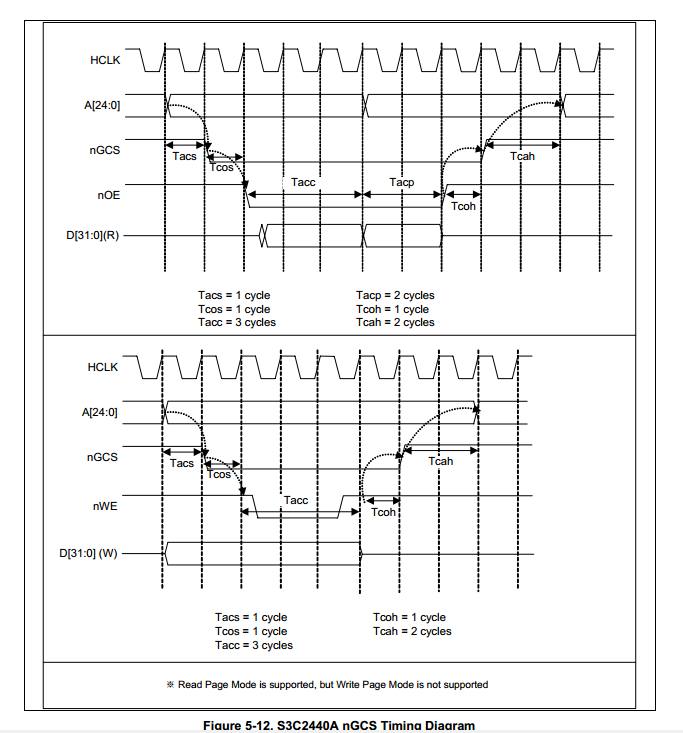
sd0-sd15: 数据接口

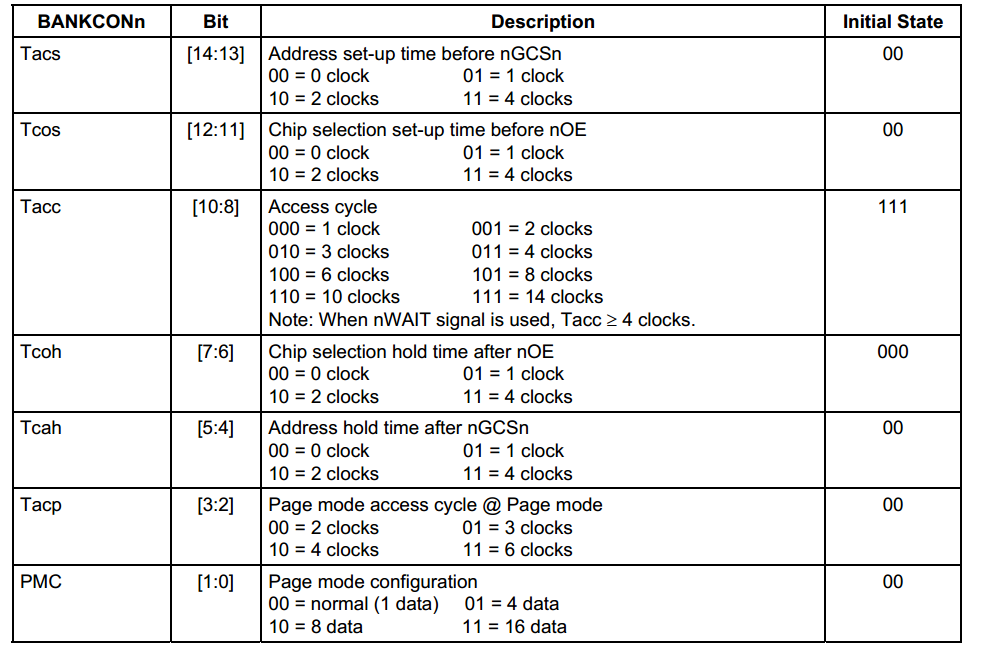
cmd: 高电平sd为数据接口，低电平时为索引寄存器接口

IOR: 读命令，指示读操作。由cpu发出

IOW: 写命令，指示写操作。由cpu发出







/\* 设置S3C2440的memory controller \*/

    bwscon   = ioremap(0x48000000, 4);

    bankcon4 = ioremap(0x48000014, 4);

    /\* DW4[17:16]: 01-16bit      设置位宽

     \* WS4[18]   : 0-WAIT disable

     \* ST4[19]   : 0 = Not using UB/LB (The pins are dedicated nWBE[3:0])字节写

     \*/

    val = \*bwscon;

    val &= ~(0xf<<16);

    val |= (1<<16);

    \*bwscon = val;

    /\*

outb(DM9KS\_VID\_L, iobase);  DM9000C的索引寄存器(cmd引脚为0)

        id\_val = inb(iobase + 4);  读DM9000C的数据寄存器(cmd引脚为1)

     \* Tacs[14:13]:发出地址信号后多长时间才能发出片选信号之前。  DM9000C的片选信号和CMD信号可以同时发出, 所以它设为0。cmd信号就是发送地址时变低的。

     \* Tcos[12:11]: 发出片选信号之后,多长时间才能发出读信号nOE

     \*              DM9000C的T1>=0ns,

     \*              所以它设为0

     \* Tacc[10:8] : 读写信号的脉冲长度,

     \*              DM9000C的T2>=10ns,

        所以它设为0, 表示1个hclk周期,hclk=100MHz,就是10ns

     \*            稳定期间也 也可以设为1, 表示2个hclk周期,hclk=100MHz,就是20ns

     \* Tcoh[7:6]  : 当读信号nOE变为高电平后,片选信号还要维持多长时间。T4

     \*              DM9000C进行写操作时, nWE变为高电平之后, 数据线上的数据还要维持最少3ns

     \*              DM9000C进行读操作时, nOE变为高电平之后, 数据线上的数据在6ns之内会消失

     \*              让片选信号在nOE放为高电平后,再维持10ns,

     \*              所以设为01

     \* Tcah[5:4]  : 当片选信号变为高电平后, 地址信号还要维持多长时间

     \*              DM9000C的片选信号和CMD信号可以同时出现,同时消失

     \*              所以设为0

     \* PMC[1:0]   : 00-正常模式

     \*

     \*/

    \*bankcon4 = (1<<8)|(1<<6);    /\* 对于DM9000C可以设Tacc为1, 对于DM9000E,Tacc要设大一点,比如最大值7  \*/

    //\*bankcon4 = (7<<8)|(1<<6);  /\* MINI2440使用DM9000E,Tacc要设大一点 \*/

    iounmap(bwscon);

    iounmap(bankcon4);

**switch**(mode) {

**case** DM9KS\_10MHD:

**case** DM9KS\_100MHD:

**case** DM9KS\_10MFD:

**case** DM9KS\_100MFD:

            media\_mode = mode;

**break**;

**default**:

            media\_mode = DM9KS\_AUTO;

    }

    dmfe\_dev = dmfe\_probe();

**if**(IS\_ERR(dmfe\_dev))

**return** PTR\_ERR(dmfe\_dev);

**return** 0;

}

//open函数里面修改中断触发方式

static int dmfe\_open

request\_irq(dev->irq,&dmfe\_interrupt, IRQF\_TRIGGER\_RISING,dev->name,dev)

## 4.4 网卡驱动测试

测试 DM9000C 驱动程序:  
1. 把 dm9dev9000c.c 放到内核的 drivers/net 目录下  
2. 修改 drivers/net/Makefile  
把  
obj-$(CONFIG\_DM9000) += dm9000.o  
改为  
obj-$(CONFIG\_DM9000) += dm9dev9000c.o  
3. make uImage  
使用新内核启动  
4. 使用 NFS 启动  
或  
ifconfig eth0 192.168.1.112

ping 192.168.1.111

# 5 usb驱动开发

## 5.1 usb 基础知识

#### 没有安装设备驱动程序，为何知道设备名字“android phone”

因为操作系统里面已经有了总线驱动程序，接入设备后是总线驱动程序知道你是"andriod phone"

#### 层次结构：

app

--------------------

设备驱动程序

USB总线驱动程序

-----------------------

USB主机控制器

USB设备

#### 手动挂载U盘：

mount /dev/sda1 /mnt //sda是u盘设备，sda1设备第一个分区，挂载的是分区

#### USB总线驱动程序负责：

识别设备，给USB设备找到对应的设备驱动程序，提供USB读写函数。

USB设备驱动程序：知道数据含义

总线驱动程序和USB设备的控制器，遵循USB规范。USB总线驱动程序发出命令获取设备的描述符信息：

USB设备返回这些描述符信息

#### PC 机上有非常多的USB设备，怎么分辨它们？

每一个USB设备接入PC时，USB总线驱动程序都会给它分配一个编号。每一个USB设备都有自己唯一 的地址编号。PC访问某个设备时，发出的命令都有对应的编号地址。

#### USB设备刚接入PC时还没有编号，怎么通信？

新接入的USB设备的默认编号是0，在未分配新编号前，PC使用0地址和它通信

#### 为什么一接入USB设备，PC机就发现它：

PC 的USB口内部，D-和D+内部都接有15K的电阻到地，未接USB设备时是低电平

USB设备内部,D-或者D+接1.5k电阻到高电平，一旦设备接入，就会把PC的USB的D-或者D+拉高，从硬件的角度通知USB有新设备接入。USB是主从结构，所有的传输都是有主机方发起的

#### USB的传输类型

控制传输 枚举设备时，使用端点0，端点0是双向的。其他的端点都是单向的

中断传输 可靠，实时， 实际上是内部查询方式，100ms interval 查询一次

批量传输 可靠，非实时

实时传输 实时，可靠

#### USB/传输的对象：端点 endpoint

每一个端点都有唯一传输类型和方向，把数据写入U盘的端点1，从U盘的端点2里读数据.

端点1为输出端点，端点2为输入端点

术语里，输入IN和输出Out都是基于USB主机的立场说的。比如鼠标数据从鼠标传入主机，

那么这个端点为输入端点。即是在开发设备时这个端点也要设定为输入。

## 5.2 usb 总线驱动程序

#### USB规范

UHCI Intel 低速/全速

OHCI microsoft 低速/全速 OHCI\_S3C2410 使用该规范

EHCI 高速

#### 描述符：

设备描述符

配置描述符 一次只能有一个配置生效，决定和哪个接口通信

接口描述符

端点描述符

一般只有一个配置描述符

一个物理设备可以包含多个逻辑设备

一个接口对应一个逻辑设备，对应一个驱动程序。比如音频设备，录音和播放两个接口，需要两个驱动。

#### USB 枚举 :

由hub\_thread处理，hub\_thread平时休眠，USB设备接入产生中断，中断函数kick\_khubd()里面唤醒hub\_thread。

创建hub\_thread并休眠

static int \_\_init usb\_init(void) // usb.c usb总线驱动模块

usb\_hub\_init() //创建hub\_thread

khubd\_task = kthread\_run(hub\_thread, NULL, "khubd");//create and wake a thread

hub\_thread

wait\_event\_interruptible(khubd\_wait,!list\_empty(

&hub\_event\_list) ||kthread\_should\_stop()); // hub\_thread 开始休眠

USB设备接入产生中断

hub\_irq

kick\_khubd

wake\_up(&khubd\_wait); hub\_thread 被唤醒

//之前创建hub\_thread ;khubd\_task = kthread\_run(hub\_thread, NULL, "khubd");

hub\_thread

hub\_events // hub\_thread被唤醒后执行

hub\_port\_connect\_change

usb\_alloc\_dev(hdev, hdev->bus, port1);

dev->dev.bus = &usb\_bus\_type; //设置usb设备结构体

choose\_address(udev); 分配地址编号

hub\_port\_init()

dev\_info //打印出发现新的USB设备

hub\_set\_address(udev);

usb\_get\_device\_descriptor(udev, 8);

usb\_get\_device\_descriptor(udev, USB\_DT\_DEVICE\_SIZE);

usb\_new\_device

usb\_get\_configuration(udev)

usb\_parse\_configuration

device\_add(&udev->dev)

bus\_add\_device

//把设备放入usb\_bus\_type的dev链表，从总线usb\_bus\_typed的driver列表

//取出驱动一一比较,匹配则调用设备driver的probe函数

usb\_register

usb\_register\_driver

new\_driver->drvwrap.driver.probe = usb\_probe\_interface;

device\_add(&udev->dev) //注册设备时的调用过程

usb\_probe\_interface

usb\_match\_id(intf, driver->id\_table); //通过id\_table匹配

driver->probe(intf, id); //设备驱动程序的probe函数

usb\_device\_match //注册驱动时调用

usb\_bus\_type

/ \

usb\_new\_device usb\_register

/ \

usb\_interface usb\_driver

通过id\_table匹配，然后调用Probe

## 5.3 编写一个USB设备驱动 参考usb\_mouse.c

usb 设备已经由总线驱动构造并注册，我们需要写设备驱动程序。

usb鼠标用作按键， 左--L 右--S 中键Enter

是一个Usb设备，也是一个输入设备

设备操作

1、分配Input\_dev

2、设置

3、注册

4、硬件相关

驱动编写

#### 如何匹配设备？

#define USB\_DEVICE\_ID\_MATCH\_INT\_INFO \

(USB\_DEVICE\_ID\_MATCH\_INT\_CLASS | \

USB\_DEVICE\_ID\_MATCH\_INT\_SUBCLASS | \

USB\_DEVICE\_ID\_MATCH\_INT\_PROTOCOL)

#define USB\_INTERFACE\_INFO(cl,sc,pr) \

.match\_flags = USB\_DEVICE\_ID\_MATCH\_INT\_INFO, .bInterfaceClass = (cl), \

.bInterfaceSubClass = (sc), .bInterfaceProtocol = (pr)

//根据接口类，子类，协议匹配

#### static struct usb\_device\_id usbmouse\_as\_key\_id\_table [] =

#### {

#### { USB\_INTERFACE\_INFO(USB\_INTERFACE\_CLASS\_HID, USB\_INTERFACE\_SUBCLASS\_BOOT,

#### USB\_INTERFACE\_PROTOCOL\_MOUSE) },

#### };

#### USB设备驱动:

#include <linux/kernel.h>

#include <linux/slab.h>

#include <linux/module.h>

#include <linux/init.h>

#include <linux/usb/input.h>

#include <linux/hid.h>

static struct input\_dev  \*uk\_dev;

static char  \*usb\_buf;

static dma\_addr\_t usb\_buf\_phys;

static int len;

static struct urb \*uk\_urb;

static struct usb\_device\_id usbmouse\_as\_key\_id\_table [] = {

    { USB\_INTERFACE\_INFO(USB\_INTERFACE\_CLASS\_HID, USB\_INTERFACE\_SUBCLASS\_BOOT,

        USB\_INTERFACE\_PROTOCOL\_MOUSE) },

    //{USB\_DEVICE(0x1234,0x5678)},

    { }    /\* Terminating entry \*/

};

static void usbmouse\_as\_key\_irq(struct urb \*urb)

{

    static unsigned char pre\_val;

#if  0

/\* 用来获取鼠标按键数据，因为我们不知道鼠标数据什么意思用来做第三个实验\*/

    int i;

    static int cnt = 0;

    printk("data cnt %d: ", ++cnt);

**for** (i = 0; i < len; i++)

    {

        printk("%02x ", usb\_buf[i]);

    }

    printk("\n");

#endif

    /\* USB鼠标数据含义

     \* data[0]: bit0-左键, 1-按下, 0-松开

     \*          bit1-右键, 1-按下, 0-松开

     \*          bit2-中键, 1-按下, 0-松开

     \*

     \*/

**if** ((pre\_val & (1<<0)) != (usb\_buf[0] & (1<<0))) //按键之前的状态和现在的状态是否变化

    {

        /\* 左键发生了变化 \*/

        input\_event(uk\_dev, EV\_KEY, KEY\_L, (usb\_buf[0] & (1<<0)) ? 1 : 0);

        input\_sync(uk\_dev);

    }

**if** ((pre\_val & (1<<1)) != (usb\_buf[0] & (1<<1)))

    {

        /\* 右键发生了变化 \*/

        input\_event(uk\_dev, EV\_KEY, KEY\_S, (usb\_buf[0] & (1<<1)) ? 1 : 0);

        input\_sync(uk\_dev);

    }

**if** ((pre\_val & (1<<2)) != (usb\_buf[0] & (1<<2)))

    {

        /\* 中键发生了变化 \*/

        input\_event(uk\_dev, EV\_KEY, KEY\_ENTER, (usb\_buf[0] & (1<<2)) ? 1 : 0);

        input\_sync(uk\_dev);

    }

    pre\_val = usb\_buf[0];

    /\* 重新提交urb \*/

    usb\_submit\_urb(uk\_urb, GFP\_KERNEL);

}

static int usbmouse\_as\_key\_probe(struct usb\_interface \*intf, const struct usb\_device\_id \*id)

{

    struct usb\_device \*dev = interface\_to\_usbdev(intf);

    struct usb\_host\_interface \*interface;

    struct usb\_endpoint\_descriptor \*endpoint;

    int pipe;

    interface = intf->cur\_altsetting;

    endpoint = &interface->endpoint[0].desc;

/\* a. 分配一个input\_dev \*/

uk\_dev = input\_allocate\_device();

    /\* b. 设置 \*/

    /\* b.1 能产生哪类事件 \*/

    set\_bit(EV\_KEY, uk\_dev->evbit);

    set\_bit(EV\_REP, uk\_dev->evbit);

    /\* b.2 能产生哪些事件 \*/

    set\_bit(KEY\_L, uk\_dev->keybit);

    set\_bit(KEY\_S, uk\_dev->keybit);

    set\_bit(KEY\_ENTER, uk\_dev->keybit);

    /\* c. 注册 \*/

    input\_register\_device(uk\_dev);

    /\* d. 硬件相关操作 \*/

    /\* 数据传输3要素: 源,目的,长度 \*/

    /\* 源: USB设备的某个端点

构造一个输入通道\*/

    pipe = usb\_rcvintpipe(dev, endpoint->bEndpointAddress);

    /\* 长度: \*/

    len = endpoint->wMaxPacketSize;

    /\* 目的: \*/

    usb\_buf = usb\_buffer\_alloc(dev, len, GFP\_ATOMIC, &usb\_buf\_phys);

    /\* 使用"3要素" \*/

    /\* 分配usb request block \*/

    uk\_urb = usb\_alloc\_urb(0, GFP\_KERNEL);

    /\* 使用"3要素设置urb" \*/

    usb\_fill\_int\_urb(uk\_urb, dev, pipe, usb\_buf, len, usbmouse\_as\_key\_irq, NULL, endpoint->bInterval);

    uk\_urb->transfer\_dma = usb\_buf\_phys;

    uk\_urb->transfer\_flags |= URB\_NO\_TRANSFER\_DMA\_MAP;

    /\* 使用URB \*/

    usb\_submit\_urb(uk\_urb, GFP\_KERNEL);

**return** 0;

}

static void usbmouse\_as\_key\_disconnect(struct usb\_interface \*intf)

{

struct usb\_device \*dev = interface\_to\_usbdev(intf);

    //printk("disconnect usbmouse!\n");

    usb\_kill\_urb(uk\_urb);

    usb\_free\_urb(uk\_urb);

    usb\_buffer\_free(dev, len, usb\_buf, usb\_buf\_phys);

    input\_unregister\_device(uk\_dev);

    input\_free\_device(uk\_dev);

}

/\* 1. 分配/设置usb\_driver \*/

static struct usb\_driver usbmouse\_as\_key\_driver = {

    .name       = "usbmouse\_as\_key\_",

    .probe      = usbmouse\_as\_key\_probe,

    .disconnect = usbmouse\_as\_key\_disconnect,

    .id\_table   = usbmouse\_as\_key\_id\_table,

};

static int usbmouse\_as\_key\_init(void)

{

    /\* 2. 注册 \*/

    usb\_register(&usbmouse\_as\_key\_driver);

**return** 0;

}

static void usbmouse\_as\_key\_exit(void)

{

    usb\_deregister(&usbmouse\_as\_key\_driver);

}

module\_init(usbmouse\_as\_key\_init);

module\_exit(usbmouse\_as\_key\_exit);

测试驱动：

make menuconfig 去掉原来的鼠标驱动

Device Drivers │

│ -> HID Devices │

│ -> USB Human Interface Device (full HID) support (USB\_HID [=y])

│ -> Force feedback support (EXPERIMENTAL) (HID\_FF [=n])

make uImage

使用新内核：

#### 测试3th：

1. insmod usbmouse\_as\_key.k

2. ls /dev/event\*

3. 接上 USB 鼠标。

4. ls /dev/event\*

5. 操作鼠标观察数据

显示的数据是 16 进制。有 8 个字节。  
00 00 00 00 00 00 00 00  
按键 X方向位移 Y方向位移 滚轮  
不同的鼠标可能不一样，有的是 4 字节。  
00 00 00 00  
按键 X方向位移 Y方向位移 滚轮

X方向：往左移是负数，往右移是正数。  
Y 方向：往上移是负数，往下移是正数。  
滚 轮：往前滑是正数，往后滑是负数。

分析数据：

第一个字节的数据“01”里的 "1"是 bit0，表示左键按下

“02”中的“2”就是 bit1.表示右键按下

04”中的“4”就是 bit2，表示中建按下

#### 测试 4th:

1. insmod usbmouse\_as\_key.ko
2. ls /dev/event\*
3. cat /dev/tty1 然后按鼠标键

左键+右键后回车

1. hexdump /dev/event0 要最后那个 event （并不一定是 event0）

秒 微秒 type code value

秒 微秒 按键类 哪个按键 按下  
099a 0000 8eea 000e 0001 0026 0001 0000

# 6 字符设备驱动程序

struct device \*parent;//父设备指针，类似于包含关系，parentt包含子设备。

## 6.1 如何找到设备的file\_operations结构体

6.1.1 2.4内核中如何找到设备的file\_operations结构体  
①，确定主设备号。  
②， file\_operations 结构  
③， register\_chrdev(主设备号，名字， file\_operations 结构）；  
④，入口函数。  
⑤，出口函数。

内核中定义了chrdevs 数组，以主设备号作为索引，保存设备的file\_operations结构体地址。当open函数打开设备时，就以“主设备号”为下标在chrdevs数组中找到当前设备的file\_operations。

缺点：只能有255个不同类型的设备限制

6.1.2 2.6内核中如何找到设备的file\_operations结构体

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/fs.h>

#include <linux/init.h>

#include <linux/delay.h>

#include <linux/irq.h>

#include <asm/uaccess.h>

#include <asm/irq.h>

#include <asm/io.h>

#include <asm/arch/regs-gpio.h>

#include <asm/hardware.h>

#include <linux/poll.h>

#include <linux/cdev.h>

/\*由主设备号和次设备号共同决定设备驱动使用哪一个file\_operations结构体\*/

/\* 1. 确定主设备号 \*/

static int major;

static int hello\_open(struct inode \*inode, struct file \*file)

{

    printk("hello\_open\n");

**return** 0;

}

static int hello2\_open(struct inode \*inode, struct file \*file)

{

    printk("hello2\_open\n");

**return** 0;

}

/\* 2. 构造file\_operations \*/

static struct file\_operations hello\_fops = {

    .owner = THIS\_MODULE,

    .open  = hello\_open,

};

 static struct file\_operations hello2\_fops = {

    .owner = THIS\_MODULE,

    .open  = hello2\_open,

};

#define HELLO\_CNT   2

static struct cdev hello\_cdev;

static struct cdev hello2\_cdev;

static struct class \*cls;

 static int hello\_init(void)

{

    dev\_t devid;

         /\* 3. 告诉内核 \*/

#if 0

    major = register\_chrdev(0, "hello", &hello\_fops); /\* (major,  0), (major, 1), ..., (major, 255)都对应hello\_fops ，这是原来注册字符设备的写法\*/

#else //2.6内核以后采用下面方法注册设备

**if** (major) { //如果静态分配主设备号

        devid = MKDEV(major, 0); //构造设备号，设备号由主设备号和次设备号结合的int类型变量，//次设备占最低20位

        register\_chrdev\_region(devid, HELLO\_CNT, "hello");  /\* (major,0~1) ，设备号devid开始连续的应  HELLO\_CNT 个设备，设备名字就是hello0和hello1。hello\_fops, (major, 2~255)都不对应hello\_fops \*/

    } **else** {//动态分配设备号

        alloc\_chrdev\_region(&devid, 0, HELLO\_CNT, "hello"); /\* (major,0~1) 对应 hello\_fops, (major, 2~255)都不对应hello\_fops \*/

        major = MAJOR(devid);

}

// hello\_cdev字符设备对应的结构体cdev->ops = fops;

     cdev\_init(&hello\_cdev, &hello\_fops);

cdev\_add(&hello\_cdev, devid, HELLO\_CNT);

//单独为次设备号为2的设备注册驱动

     devid = MKDEV(major, 2);

    register\_chrdev\_region(devid, 1, "hello2");

    cdev\_init(&hello2\_cdev, &hello2\_fops);//一样的主设备号，不一样的次设备号，驱动可以不一样

    cdev\_add(&hello2\_cdev, devid, 1);

     #endif

    cls = class\_create(THIS\_MODULE, "hello");

    class\_device\_create(cls, NULL, MKDEV(major, 0), NULL, "hello0"); /\* /dev/hello0 \*/

    class\_device\_create(cls, NULL, MKDEV(major, 1), NULL, "hello1"); /\* /dev/hello1 \*/

    class\_device\_create(cls, NULL, MKDEV(major, 2), NULL, "hello2"); /\* /dev/hello2 \*/

    class\_device\_create(cls, NULL, MKDEV(major, 3), NULL, "hello3"); /\* /dev/hello3 \*/

**return** 0;

}

static void hello\_exit(void)

{

    class\_device\_destroy(cls, MKDEV(major, 0));

    class\_device\_destroy(cls, MKDEV(major, 1));

    class\_device\_destroy(cls, MKDEV(major, 2));

    class\_device\_destroy(cls, MKDEV(major, 3));

    class\_destroy(cls);

    cdev\_del(&hello\_cdev);

    unregister\_chrdev\_region(MKDEV(major, 0), HELLO\_CNT);

    cdev\_del(&hello2\_cdev);

    unregister\_chrdev\_region(MKDEV(major, 2), 1);

}

module\_init(hello\_init);

module\_exit(hello\_exit);

MODULE\_LICENSE("GPL");

# 7 RTC 设备驱动

## 7.1 RTC驱动程序框架

rtc-s3c.c

static struct platform\_driver s3c2410\_rtcdrv = {

.probe = s3c\_rtc\_probe,//探测函数

.remove = s3c\_rtc\_remove,//移除设备

.suspend = s3c\_rtc\_suspend,

.resume = s3c\_rtc\_resume,

.driver = {

.name = "s3c2410-rtc",

.owner = THIS\_MODULE,

},

};

s3c\_rtc\_init

platform\_driver\_register(&s3c2410\_rtcdrv);

s3c\_rtc\_probe

rtc\_device\_register("s3c", &pdev->dev, &s3c\_rtcops,THIS\_MODULE);//rtc/class.c中的函数

rtc->dev.parent = dev;//父设备

rtc\_dev\_prepare

cdev\_init(&rtc->char\_dev, &rtc\_dev\_fops);//注册rtc通用的接口函数

rtc\_dev\_add\_device(rtc);//注册字符设备

cdev\_add(&rtc->char\_dev, rtc->dev.devt, 1)//注册rtc设备

rtc/class.c //向linux设备模型核心注册了一个rtc类，然后向驱动程序提供了注册/注销接口设备的函数

rtc\_init(void)

rtc\_class = class\_create(THIS\_MODULE, "rtc");

rtc\_dev\_init(); //rtc-dev.c文件

alloc\_chrdev\_region(&rtc\_devt, 0, RTC\_DEV\_MAX, "rtc");//rtc-dev.c文件

rtc-dev.c //定义了基本的设备操作函数，用于用户程序接口

static const struct file\_operations rtc\_dev\_fops = {

.owner = THIS\_MODULE,

.llseek = no\_llseek,

.read = rtc\_dev\_read,

.poll = rtc\_dev\_poll,

.ioctl = rtc\_dev\_ioctl,

.open = rtc\_dev\_open,

.release = rtc\_dev\_release,

.fasync = rtc\_dev\_fasync,//异步通知

};

rtc\_dev\_open

struct rtc\_device \*rtc = container\_of(inode->i\_cdev,struct rtc\_device, char\_dev);

const struct rtc\_class\_ops \*ops = rtc->ops;

err = ops->open ? ops->open(rtc->dev.parent) : 0;//调用具体设备的open函数，为什么是dev.parent

用户程序一般通过Ioctl与rtc驱动交互

rtc\_dev\_ioctl

app: ioctl(fd, RTC\_RD\_TIME,...)  
-------------------------------------------  
kernel: sys\_ioctl  
 rtc\_dev\_fops.ioctl

rtc\_dev\_ioctl  
struct rtc\_device \*rtc = file->private\_data; 得到 rtc\_device 结构体  
 rtc\_read\_time(rtc, &tm);  
 err = rtc->ops->read\_time(rtc->dev.parent, tm);  
 s3c\_rtc\_gettime

rtc-s3c.c是真正的设备驱动函数

static const struct rtc\_class\_ops s3c\_rtcops = {

.open = s3c\_rtc\_open,

.release = s3c\_rtc\_release,

.ioctl = s3c\_rtc\_ioctl,

.read\_time = s3c\_rtc\_gettime,

.set\_time = s3c\_rtc\_settime,

.read\_alarm = s3c\_rtc\_getalarm,

.set\_alarm = s3c\_rtc\_setalarm,

.proc = s3c\_rtc\_proc,

};

//读寄存器

static int s3c\_rtc\_gettime(struct device \*dev, struct rtc\_time \*rtc\_tm)

{

rtc\_tm->tm\_min = readb(base + S3C2410\_RTCMIN);

rtc\_tm->tm\_hour = readb(base + S3C2410\_RTCHOUR);

rtc\_tm->tm\_mday = readb(base + S3C2410\_RTCDATE);

rtc\_tm->tm\_mon = readb(base + S3C2410\_RTCMON);

rtc\_tm->tm\_year = readb(base + S3C2410\_RTCYEAR);

rtc\_tm->tm\_sec = readb(base + S3C2410\_RTCSEC)

}

添加rtc设备

在开发板启动后，并没有加载 RTC 驱动，但内核中其实已经有了驱动，只是没有加载平台设备。

common-smdk.c

static struct platform\_device \_\_initdata \*smdk\_devs[] = {

&s3c\_device\_nand,

&smdk\_led4,

&smdk\_led5,

&smdk\_led6,

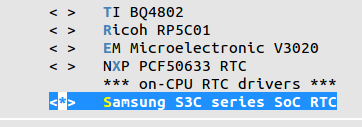
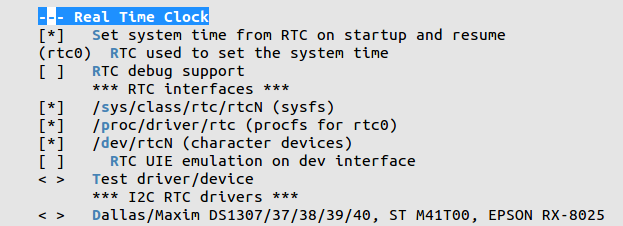
&smdk\_led7,

&s3c\_device\_rtc,

};

make menuconfig

drivers/



make uimage

使用新内核启动

ls /dev/rtc\* -l

date /\* 显示系统时间 \*/

系统时间是通过 date 来设置的， RTC 时间是通过 hwclock 来设置的。开机时系统时间首先通过 RTC 来获得，RTC没有设置时，系统时间也会使用Thu Jan 1 00:10:37 UTC 1970

date --help 获取帮助信息,设置软件系统时间  
date /\* 显示系统时间 \*/  
date 123015402011.30 /\* 设置系统时间 date [MMDDhhmm[[CC]YY][.ss]] \*/

date 111311582018.16 //2018 11 13 11:58:16

月天时分年秒  
hwclock -w /\* 把系统时间写入RTC 芯片内部\*/

# 8 块设备驱动

## 8.1 为何块设备需要优化读写过程

### 8.1.1 磁盘设备为何要优化读写过程？

磁盘的读写其实非常快，慢在机械结构读写装置的定位上面，从一个“磁头”的某  
“柱面”某“扇区”读到数据后（步骤 R0），跳到另一个“磁头”的某“柱面”的某“扇  
区”去写（步骤 W），接着再跳回原“磁头”相同柱面的下一个“扇区”去读（步骤  
R1）。慢就慢在读写扇区的跳转过程中。若按“字符设备”中的“open” ,"read","write"  
方式，则总体效率在硬盘的读写上会非常低。  
上面过程是“R0” ->"W"->"R1"，这个步骤跳转 2 次。  
若优化这个步骤为： R0->R1->W。这个步骤跳转 1 次。这样效率会高些。  
总结：先不执行而是放入队列，优化后再执行（对硬盘有这种要求）。用“字符设备驱  
动”程序那样读写时就会在硬盘上跳来跳去，整体效率会非常低。所以有必要引入“优化过程”。就是读写先不执行，先放到某个“队列”中去。（调整顺序）

### 8.1.2 flash设备为何要优化读写过程？

flash设备需要先擦出才能写入，而擦除是按块执行的。  
是“块”里有一个一个的扇区。  
假若现在要先写“扇区 0”和“扇区 1”。 FLASH 要先擦除再写，现在用字符设备驱动的读写方式来读写：对 FLASH 上的擦除是整块整块的进行的。故：  
写扇区 0 的过程：  
① 要写时，先把这整块读到一个 buf 中。  
② 然后修改 buf 中扇区 0 的数据。  
③ 这时再擦除整块。  
④ 再把修改过扇区 0 的数据的 buf 烧写到整块。  
写扇区 1 的过程：  
① 要写时，先把这整块读到一个 buf 中。  
② 然后修改 buf 中扇区 1 的数据。  
③ 这时再擦除整块。  
④ 再把修改过扇区 1 的数据的 buf 烧写到整块。  
则那么要修改多个扇区时，会擦除烧写多次。总体效率也会低。  
优化：  
①，先不执行。  
②，优化 - 合并后执行。  
合并：合并后只需要一次。  
a,读出整块到 buf 中。  
b,在 buf 中修改扇区 0 和扇区 1。  
c,擦除。  
d,烧写。  
故，块设备不能像字符设备那样提供读写函数。  
①，先把读写放入队列，先不执行。  
②，优化后再执行

### 8.1.3 字符设备和块设备区别

块设备对于I/O请求有对应的缓冲区，因此可以选择以什么样的时序进行响应，字符设备无需缓冲且被直接读写。对于存储设备而言，调整读写顺序作用巨大，因为在读写连续的扇区的存储速度比分离的扇区更快。

## 8.2 块设备驱动程序框架

APP: open,read,write "1.txt"

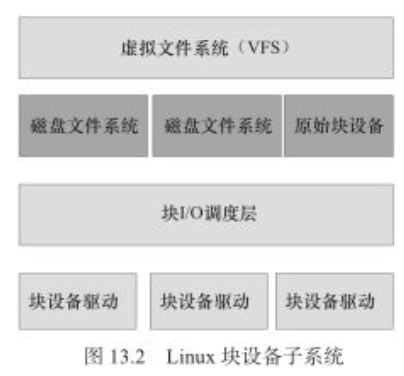
------------------------------------------ 文件读写

文件系统： vfat,ext2,ext3,yaffs （把文件的读写转换成对扇区的读写）

------------------ll\_rw\_block-------------- 扇区读写

块设备驱动程序

flash,磁盘



对普通文件 1.txt 的读写会转成对块设备的读写，要读写哪个扇区。从文件的读写转成对扇区的读写，中间会涉及到“文件系统”。应用程序读写一个普通的文件，最终会转换成操作硬件，由“块设备驱动程序”来操作硬件。普通的文件转换成对扇区的读写，是由“文件系统”转换。ll\_rw block 是通用的入口，读写请求会放入队列，优化后再来执行。 ll\_rw\_block 会把“读写”放入队列，调用队列的处理函数去优化（调顺序、合并）执行。如何知道是“ll\_rw\_block”可以看《LINUX 内核源代码情景分析》。文件系统不是我们关心的重点，分析块设备驱动程序，就是“ll\_rw\_block”这个函数开始分析

ll\_rw\_block ：通用的文件buffer.c，内核2.6 新版不一样

/\*low-level access to block devices

@rw: 读还是写

\* @nr: bhs[]的个数

\* @bhs: 读写数据三要素，源，目的，长度 \*/

void ll\_rw\_block(int rw, int nr, struct buffer\_head \*bhs[])

for (i = 0; i < nr; i++) {

struct buffer\_head \*bh = bhs[i];}

submit\_bh(rw, bh);

struct bio \*bio;//使用bh来构造bio，input/output block

submit\_bio(rw, bio);//提交bio

generic\_make\_request(bio); //使用bio来构造请求

\_\_generic\_make\_request

bdev\_get\_queue(bio->bi\_bdev);//找到队列

ret = q->make\_request\_fn(q, bio); //调用队列里面的构造请求的函数

q->make\_request\_fn = mfn;//搜索make\_request\_fn

//向内核注册一个请求队列

blk\_init\_queue(request\_fn\_proc \*rfn, spinlock\_t \*lock)

lk\_init\_queue\_node(request\_fn\_proc \*rfn, spinlock\_t \*lock, int node\_id)

q->request\_fn = rfn; //队列处理函数

blk\_queue\_make\_request(q, \_\_make\_request);

q->make\_request\_fn = mfn;

\_\_make\_request //分析\_\_make\_request 向下方向调用

el\_ret = elv\_merge(q, &req, bio);//电梯算法合并读或者写传输，合并到队列q中

//如果合并不成，使用bio构造请求，把请求放入队列

init\_request\_from\_bio(req, bio);

add\_request(q, req);

if (sync)

\_\_generic\_unplug\_device(q);//调用队列的处理函数

q->request\_fn(q);//执行请求队列

## 8.3 编写一个ramdisk驱动

采用内存模拟一个块设备。

#### 怎么写块设备驱动程序：

1 分配gendisk 结构体,alloc\_disk

2 设置

分配/设置队列：request\_queue\_t//提供读写能力

设置gendisk其他信息 //提供属性，如容量

3 注册add\_disk

/\* 参考:

 \* drivers\block\xd.c

 \* drivers\block\z2ram.c

 \*/

#include <linux/module.h>

#include <linux/errno.h>

#include <linux/interrupt.h>

#include <linux/mm.h>

#include <linux/fs.h>

#include <linux/kernel.h>

#include <linux/timer.h>

#include <linux/genhd.h>

#include <linux/hdreg.h>

#include <linux/ioport.h>

#include <linux/init.h>

#include <linux/wait.h>

#include <linux/blkdev.h>

#include <linux/blkpg.h>

#include <linux/delay.h>

#include <linux/io.h>

#include <asm/system.h>

#include <asm/uaccess.h>

#include <asm/dma.h>

#include <asm/system.h>

#include <asm/uaccess.h>

#include <asm/dma.h>

**static** **struct** gendisk \*ramblock\_disk;

**static** **struct** request\_queue \*ramblock\_queue;

**static** **int** major;

**static** DEFINE\_SPINLOCK(ramblock\_lock);

#define RAMBLOCK\_SIZE (1024\*1024)

**static** unsigned **char** \*ramblock\_buf;

**static** **int** ramblock\_getgeo(**struct** block\_device \*bdev, **struct** hd\_geometry \*geo)

{

/\* 容量=heads\*cylinders\*sectors\*512 \*/

    geo->heads     = 2;

    geo->cylinders = 32;

    geo->sectors   = RAMBLOCK\_SIZE/2/32/512;

**return** 0;

}

**static** **struct** block\_device\_operations ramblock\_fops = {

    .owner  = THIS\_MODULE,

    .getgeo = ramblock\_getgeo,

};

**static** **void** do\_ramblock\_request(**struct** request\_queue \* q)

{

**struct** request \*req;

    //printk("do\_ramblock\_request %d\n", ++cnt);

    req = blk\_fetch\_request(q);

**while** (req) {

        /\* 数据传输三要素: 源,目的,长度 \*/

        /\* 源/目的: \*/

        unsigned **long** offset = blk\_rq\_pos(req) << 9;

        /\* 目的/源: \*/

        // req->buffer

         /\* 长度: \*/

      unsigned **long** len = blk\_rq\_cur\_bytes(req);

**int** err=0;

**if** (rq\_data\_dir(req) == READ)

        {

             //printk("do\_ramblock\_request read %d\n", ++r\_cnt);

**memcpy**(req->buffer, ramblock\_buf+offset, len);

        }

**else**

         {

            //printk("do\_ramblock\_request write %d\n", ++w\_cnt);

**memcpy**(ramblock\_buf+offset, req->buffer, len);

        }

        \_\_blk\_end\_request\_all(req, 0);

        req = blk\_fetch\_request(q);

    }

}

**static** **int** ramblock\_init(**void**)

 {

     /\* 1. 分配一个gendisk结构体 \*/

    ramblock\_disk = alloc\_disk(16); /\* 次设备号个数: 分区个数+1 \*/

    /\* 2. 设置 \*/

    /\* 2.1 分配/设置队列: 提供读写能力 \*/

    ramblock\_queue = blk\_init\_queue(do\_ramblock\_request, &ramblock\_lock);

    ramblock\_disk->queue = ramblock\_queue;

    /\* 2.2 设置其他属性: 比如容量 \*/

     major = register\_blkdev(0, "ramblock");  /\* cat /proc/devices \*/

    ramblock\_disk->major       = major;

    ramblock\_disk->first\_minor = 0;

**sprintf**(ramblock\_disk->disk\_name, "ramblock");

    ramblock\_disk->fops        = &ramblock\_fops;

    set\_capacity(ramblock\_disk, RAMBLOCK\_SIZE / 512);

    /\* 3. 硬件相关操作 \*/

    ramblock\_buf = kzalloc(RAMBLOCK\_SIZE, GFP\_KERNEL);

    /\* 4. 注册 \*/

     add\_disk(ramblock\_disk);

**return** 0;

 }

**static** **void** ramblock\_exit(**void**)

{

    unregister\_blkdev(major, "ramblock");

    del\_gendisk(ramblock\_disk);

    put\_disk(ramblock\_disk);

    blk\_cleanup\_queue(ramblock\_queue);

    kfree(ramblock\_buf);

 }

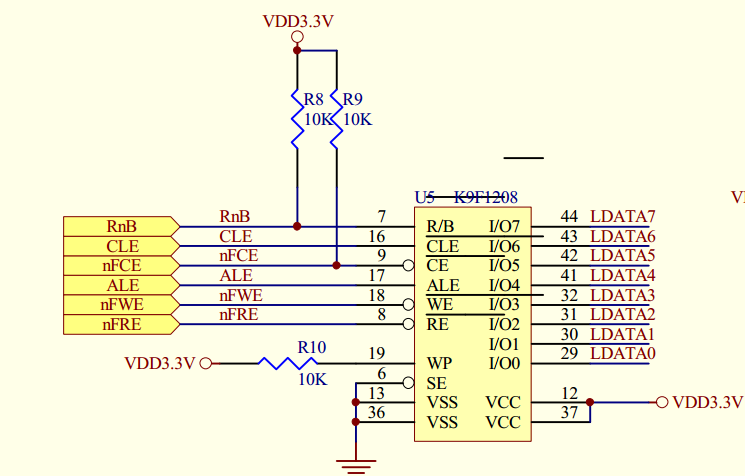
module\_init(ramblock\_init);

module\_exit(ramblock\_exit);

MODULE\_LICENSE("GPL");

# 9 Nand Flash 驱动

## 9.1 Nand 硬件原理



Nandflash引脚：

R/B 准备好和忙标志

CLE 命令锁存允许 读Id就是需要发送命令

CE 片选

ALE 地址锁存允许

WE 写脉冲

RE 读脉冲

WP

IO0-IO7 地址数据分时复用引脚

ALE 为高电平时传输的是地址，CLE为高电平传输的是命令，两者都为0是发送的是数据

R/B 高电平表示nand内部无读写数据。可以接受新的读写操作。

操作flash顺序：

1、发出命令

2、发出地址

3、发出数据/读数据

根据时序图

发命令：

1、选中芯片

2、CLE设为高电平

3、数据线上输出命令信息

4、发出一个写脉冲

其中2、4由nand 控制器自动发出，只需要把命令写到命令寄存器NFCMMD

发地址：

1、选中芯片

2、ALE设为高电平

3、数据线上输出地址信息

4、发出一个写脉冲

其中2、4由nand 控制器自动发出，只需要把地址写到地址寄存器NFADDR

发数据：

1、选中芯片

2、ALE,CLE设为低电平

3、数据线上输出数据信息

4、发出一个写脉冲

其中2、4由nand 控制器自动发出，只需要数据写到数据寄存器NFDATA

读数据：

1、选中芯片

2、发出读脉冲

3、读取数据

直接读取数据寄存器NFDATA

#### 手册时序图

读ID S3C2440 寄存器addr

UBOOT

1、选中 NECONT bit1设为0 0x4E000004

mw.l 0x4E000004 1

2、发出命令0x90 NFCMMD =0x90 0x4E000008

mw.b 0x4E000008 0x90

3、发出地址0x00 NFADDR =0x00 0x4E00000C

mw.b 0x4E00000C 0x00

4、读数据得到0xec val=NFDATA 0x4E000010

md.b 0x4E000010 1

5、读数据得到device code val=NFDATA 0x4E000010

md.b 0x4E000010 1

6、退出读id状态 NFCMMD =0xff 0x4E000008

mw.b 0x4E000008 0xff

UBOOT中的命令 help md/ help mw

md [.b, .w, .l] address [# of objects] 读

- memory display

mw [.b, .w, .l] address value [count] 写

- write memory

md.l 0x4E000004 1

读数据 S3C2440 addr UBOOT

1、选中 NECONT bit1设为0 0x4E000004 mw.l 0x4E000004 1

2、发出命令0x00 NFCMMD =0x00 0x4E000008 mw.b 0x4E000008 0x00

3、发出地址0x00 NFADDR =0x00 0x4E00000C mw.b 0x4E00000C 0x00

4、发出地址0x00 NFADDR =0x00 0x4E00000C mw.b 0x4E00000C 0x00

5、发出地址0x00 NFADDR =0x00 0x4E00000C mw.b 0x4E00000C 0x00

6、发出地址0x00 NFADDR =0x00 0x4E00000C mw.b 0x4E00000C 0x00

7、发出地址0x00 NFADDR =0x00 0x4E00000C mw.b 0x4E00000C 0x00

8、发出命令0x30 NFCMMD =0x30 0x4E000008 mw.b 0x4E000008 0x30

9、读数据 val=NFDATA 0x4E000010 md.b 0x4E000010 1

10、退出读id状态 NFCMMD =0xff mw.b 0x4E000008 0xff

nand dump 0 读出nandflash 页0数据，比较上面读出数据是否正确

## 9.2 nandfalsh 驱动框架

从内核启动信息找到：搜索这些信息可以找到驱动drivers/mtd/nand/s3c2410.c

S3C24XX NAND Driver, (c) 2004 Simtec Electronics

s3c2440-nand s3c2440-nand: Tacls=3, 30ns Twrph0=7 70ns, Twrph1=3 30ns

.......

Creating 4 MTD partitions on "NAND 256MiB 3,3V 8-bit":

0x00000000-0x00040000 : "bootloader"

0x00040000-0x00060000 : "params"

0x00060000-0x00260000 : "kernel"

0x00260000-0x10000000 : "root"

APP open read write

VFS sysopen sysread

-----------------------------

FS Vfat yaffs2,jffs2

--------------------------

字符设备 | 块设备层 mtd层 知道怎么优化

nand 协议层 知道发什么来读写，擦除

硬件相关层 知道怎样发命令地址读写数据

------------------------------

硬件

linux 下MTD系统：参看linux设备驱动开发详解

mtd\_info: 标识Mtd原始设备结构体，每个分区被认为是一个Mtd\_info.两个Mtd原始设备，每个上有3个分区，那么有6个mtd\_info结构体对应6个分区。而mtd\_info指针被存储在mtd\_table的数组里。

s3c24xx\_nand\_probe

s3c2410\_nand\_inithw(info, pdev);//时序配置，初始化硬件

s3c2410\_nand\_init\_chip(info, nmtd, sets)

nand\_scan  //扫描NANDflash，驱动中调用该函数，

ret = nand\_scan\_ident(mtd, maxchips);    //内核提供函数

          nand\_set\_defaults(chip, busw);//没有重新定义nand的操作函数使用默认

               chip->cmdfunc = nand\_command;//实际使用nand\_command\_lp

chip->cmd\_ctrl(mtd, column, ctrl);//发送地址函数

                          chip->select\_chip = nand\_select\_chip;

         nand\_get\_flash\_type

              chip->select\_chip(mtd, 0);//读id,然后查表得到芯片参数

             chip->cmdfunc(mtd, NAND\_CMD\_READID, 0x00, -1);

             \*maf\_id = chip->read\_byte(mtd);

             dev\_id = chip->read\_byte(mtd);

**if** (dev\_id == nand\_flash\_ids[i].id) {

                 type =  &nand\_flash\_ids[i];

chip->cmdfunc = nand\_command\_lp//根据参数知道是大页flash

         ret = nand\_scan\_tail(mtd);// 函数则会查找或者建立bbt (bad block table)

                mtd->erase = nand\_erase;

                mtd->read = nand\_read;

                mtd->write = nand\_write;

s3c2410\_nand\_add\_partition(info, nmtd, sets);//没有出错，注册分区

             add\_mtd\_device(&mtd->mtd);

                 list\_for\_each(**this**, &mtd\_notifiers) {//对mtd\_notifiers 里的每一项调用add函数

**struct** mtd\_notifier \*not = list\_entry(**this**, **struct** mtd\_notifier, list);

                    not->add(mtd);

//寻找add()函数

static struct mtd\_notifier notifier = {

.add = mtd\_notify\_add,

.remove = mtd\_notify\_remove,

};

static struct mtd\_notifier blktrans\_notifier = {

.add = blktrans\_notify\_add,

.remove = blktrans\_notify\_remove,

};

mtd\_notifiers在哪设置？

static LIST\_HEAD(mtd\_notifiers);

一个设备可以作为字符设备也可以作为块设备：

init\_mtdchar()

register\_mtd\_user (struct mtd\_notifier \*new)

list\_add(&new->list, &mtd\_notifiers)

init\_mtdblock ()

register\_mtd\_blktrans(&mtdblock\_tr)

register\_mtd\_user(&blktrans\_notifier);//register\_mtd\_user (struct mtd\_notifier \*new)

list\_add(&new->list, &mtd\_notifiers)

mtd\_notify\_add和blktrans\_notify\_add

#### 将Mtd注册为字符设备

mtd\_notify\_add（）

class\_device\_create(mtd\_class, NULL, MKDEV(MTD\_CHAR\_MAJOR, mtd->index\*2),

NULL, "mtd%d", mtd->index);//创建mtd设备类

class\_device\_create(mtd\_class, NULL,

MKDEV(MTD\_CHAR\_MAJOR, mtd->index\*2+1),

NULL, "mtd%dro", mtd->index); //创建只读mtd设备类

#### 将Mtd注册为块设备

blktrans\_notify\_add

list\_for\_each(this, &blktrans\_majors) {

struct mtd\_blktrans\_ops \*tr = list\_entry(this, struct mtd\_blktrans\_ops, list);

tr->add\_mtd(tr, mtd);

blktrans\_majors列表

register\_mtd\_blktrans(struct mtd\_blktrans\_ops \*tr)

tr->blkcore\_priv->rq = blk\_init\_queue(mtd\_blktrans\_request, &tr->blkcore\_priv->queue\_lock);

mtd\_blktrans\_request

wake\_up\_process(tr->blkcore\_priv->thread);

tr->blkcore\_priv->thread = kthread\_run(mtd\_blktrans\_thread, tr,

"%sd", tr->name);

mtd\_blktrans\_thread//主线程，平时休眠，有数据传输时唤醒

list\_add(&tr->list, &blktrans\_majors);

mtdblock\_add\_mtd

add\_mtd\_blktrans\_dev(dev);

alloc\_disk(1 << tr->part\_bits);

gd->fops = &mtd\_blktrans\_ops;

gd->queue = tr->blkcore\_priv->rq;//tr->blkcore\_priv->rq->queuedata = tr;tr=mtdblock\_tr

add\_disk(gd);

## 9.2 nandfalsh 驱动程序

#### 写代码时可以参考

driver/mtd/nand/s3c2410:

driver/mtd/nand/AT91\_nand.c:

arch/arm/plat-s3c24xx/common-smdk.c //分区信息

/\* 参考

 \* drivers\mtd\nand\s3c2410.c

 \* drivers\mtd\nand\at91\_nand.c

 \*/

 #include <linux/module.h>

#include <linux/types.h>

#include <linux/init.h>

#include <linux/kernel.h>

#include <linux/string.h>

#include <linux/ioport.h>

#include <linux/platform\_device.h>

#include <linux/delay.h>

#include <linux/err.h>

#include <linux/slab.h>

#include <linux/clk.h>

  #include <linux/mtd/mtd.h>

#include <linux/mtd/nand.h>

#include <linux/mtd/nand\_ecc.h>

#include <linux/mtd/partitions.h>

  #include <asm/io.h>

  #include <asm/arch/regs-nand.h>

#include <asm/arch/nand.h>

**struct** s3c\_nand\_regs {

    unsigned **long** nfconf  ;

    unsigned **long** nfcont  ;

    unsigned **long** nfcmd   ;

    unsigned **long** nfaddr  ;

    unsigned **long** nfdata  ;

    unsigned **long** nfeccd0 ;

    unsigned **long** nfeccd1 ;

    unsigned **long** nfeccd  ;

    unsigned **long** nfstat  ;

    unsigned **long** nfestat0;

    unsigned **long** nfestat1;

    unsigned **long** nfmecc0 ;

    unsigned **long** nfmecc1 ;

    unsigned **long** nfsecc  ;

    unsigned **long** nfsblk  ;

    unsigned **long** nfeblk  ;

};

**static** **struct** nand\_chip \*s3c\_nand;

**static** **struct** mtd\_info \*s3c\_mtd;

**static** **struct** s3c\_nand\_regs \*s3c\_nand\_regs;

**static** **struct** mtd\_partition s3c\_nand\_parts[] = {

    [0] = {

        .name   = "bootloader",

        .size   = 0x00040000,

        .offset = 0,

    },

    [1] = {

        .name   = "params",

        .offset = MTDPART\_OFS\_APPEND,

        .size   = 0x00020000,

    },

    [2] = {

        .name   = "kernel",

        .offset = MTDPART\_OFS\_APPEND,

        .size   = 0x00200000,

    },

    [3] = {

        .name   = "root",

        .offset = MTDPART\_OFS\_APPEND,

        .size   = MTDPART\_SIZ\_FULL,

    }

};

**static** **void** s3c2440\_select\_chip(**struct** mtd\_info \*mtd, **int** chipnr)

{

**if** (chipnr == -1)

    {

        /\* 取消选中: NFCONT[1]设为1 \*/

        s3c\_nand\_regs->nfcont |= (1<<1);

    }

**else**

    {

        /\* 选中: NFCONT[1]设为0 \*/

        s3c\_nand\_regs->nfcont &= ~(1<<1);

    }

}

**static** **void** s3c2440\_cmd\_ctrl(**struct** mtd\_info \*mtd, **int** dat, unsigned **int** ctrl)

{

**if** (ctrl & NAND\_CLE)

    {

        /\* 发命令: NFCMMD=dat \*/

        s3c\_nand\_regs->nfcmd = dat;

    }

**else**

    {

        /\* 发地址: NFADDR=dat \*/

        s3c\_nand\_regs->nfaddr = dat;

    }

}

**static** **int** s3c2440\_dev\_ready(**struct** mtd\_info \*mtd)

{

**return** (s3c\_nand\_regs->nfstat & (1<<0));

}

**static** **int** s3c\_nand\_init(**void**)

{

**struct** clk \*clk;

         /\* 1. 分配一个nand\_chip结构体 \*/

    s3c\_nand = kzalloc(**sizeof**(**struct** nand\_chip), GFP\_KERNEL);

     s3c\_nand\_regs = ioremap(0x4E000000, **sizeof**(**struct** s3c\_nand\_regs));

         /\* 2. 设置nand\_chip \*/

    /\* 设置nand\_chip是给nand\_scan函数使用的, 如果不知道怎么设置, 先看nand\_scan怎么使用

     \* 它应该提供:选中,发命令,发地址,发数据,读数据,判断状态的功能

     \*/

    s3c\_nand->select\_chip = s3c2440\_select\_chip;

    s3c\_nand->cmd\_ctrl    = s3c2440\_cmd\_ctrl;

    s3c\_nand->IO\_ADDR\_R   = &s3c\_nand\_regs->nfdata;

    s3c\_nand->IO\_ADDR\_W   = &s3c\_nand\_regs->nfdata;

    s3c\_nand->dev\_ready   = s3c2440\_dev\_ready;

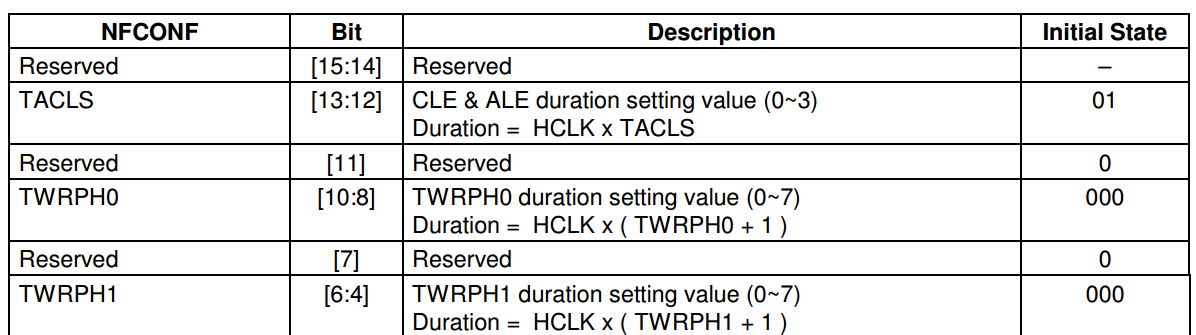
    s3c\_nand->ecc.mode    = NAND\_ECC\_SOFT;

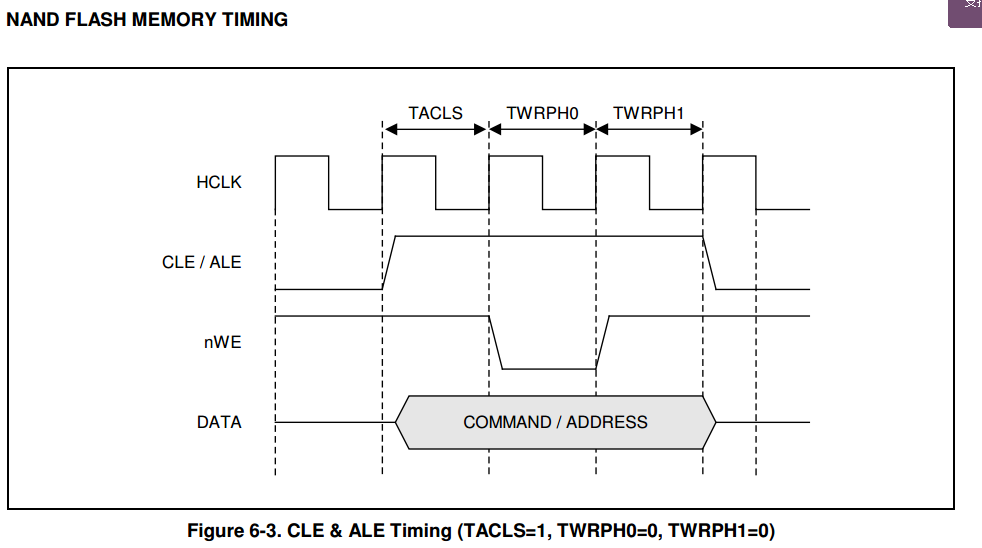
    /\* 3. 硬件相关的设置: 根据NAND FLASH的手册设置时间参数 \*/

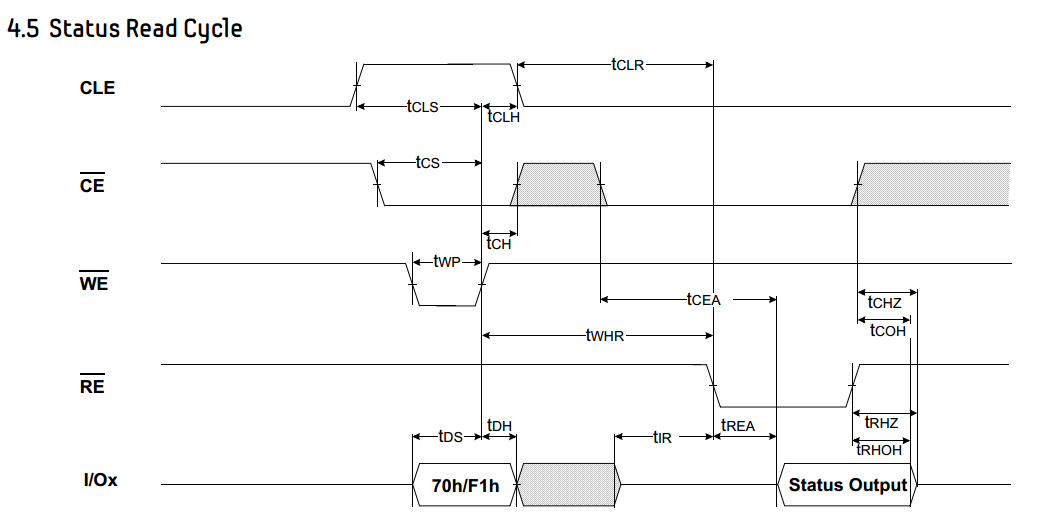
    /\* 使能NAND FLASH控制器的时钟 \*/

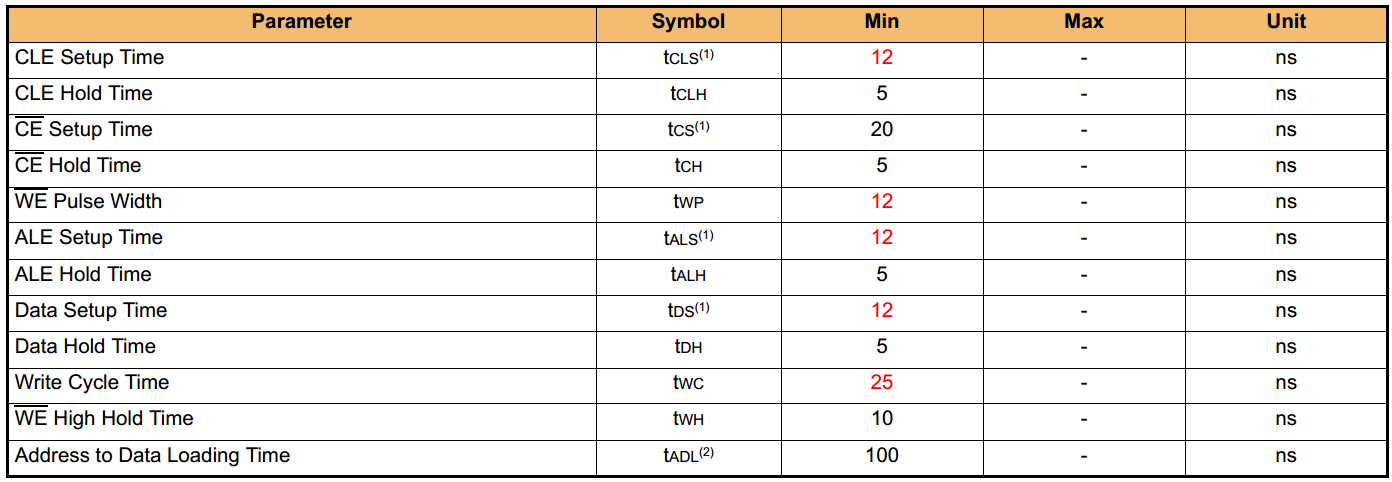
    clk = clk\_get(NULL, "nand");

clk\_enable(clk);              /\* CLKCON'bit[4] \*/









 /\* HCLK=100MHz

  \* TWRPH0: nWE的脉冲宽度, HCLK x ( TWRPH0 + 1 ), 从NAND手册可知它要>=12ns, 所以HCLK x ( TWRPH0 + 1 ) >=12ns ,TWRPH0>=1

   \* TWRPH1: nWE变为高电平后多长时间CLE/ALE才能变为低电平, 从NAND手册可知它要>=5ns, HCLK x ( TWRPH1 + 1 )>=5,所以TWRPH1>=0

     \* TACLS:  发出CLE/ALE之后多长时间才发出nWE信号,

从NAND手册可知tacls=tcls-twp=12-12=0 ,CLE/ALE与nWE可以同时发出,所以TACLS=0

          \*/

#define TACLS    0

#define TWRPH0   1

#define TWRPH1   0

    s3c\_nand\_regs->nfconf = (TACLS<<12) | (TWRPH0<<8) | (TWRPH1<<4);

    /\* NFCONT:

     \* BIT1-设为1, 取消片选

     \* BIT0-设为1, 使能NAND FLASH控制器

     \*/

    s3c\_nand\_regs->nfcont = (1<<1) | (1<<0);

    /\* 4. 使用: nand\_scan \*/

    s3c\_mtd = kzalloc(**sizeof**(**struct** mtd\_info), GFP\_KERNEL);

    s3c\_mtd->owner = THIS\_MODULE;

    s3c\_mtd->priv  = s3c\_nand;

    nand\_scan(s3c\_mtd, 1);  /\* 识别NAND FLASH, 构造mtd\_info \*/

    /\* 5. add\_mtd\_partitions \*/

    add\_mtd\_partitions(s3c\_mtd, s3c\_nand\_parts, 4);

    //add\_mtd\_device(s3c\_mtd); //如果不分区的话就用这个函数代替add\_mtd\_partitions

**return** 0;

}

**static** **void** s3c\_nand\_exit(**void**)

{

    del\_mtd\_partitions(s3c\_mtd);

    kfree(s3c\_mtd);

    iounmap(s3c\_nand\_regs);

    kfree(s3c\_nand);

}

module\_init(s3c\_nand\_init);

module\_exit(s3c\_nand\_exit);

MODULE\_LICENSE("GPL");

实验：

1、去掉内核自带的nand驱动

2、make uimage 生成uImage\_nonand

3、使用新内核启动，并且使用nfs作为根文件系统

//内核文件nfsroot.txt讲解了如何配置nfs，因为没有nand所以不能从文件系统启动

set bootargs console=ttySAC0 root=/dev/nfs nfsroot=192.168.0.111:/home/chao/work/nfs\_root

ip=192.168.0.112:192.168.0.111:192.168.0.1:255.255.255.0::eth0:off

init=/linuxrc

nfs 30000000 192.168.0.111:/home/chao/work/nfs\_root/uImage\_nonand

bootm

insmod nand-s3c.ko

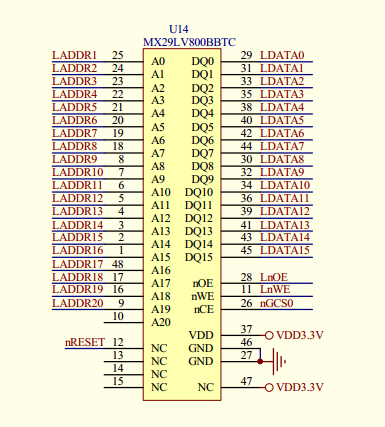
4、格式化 //mtd-utils 工具在linux环境搭建，不是必须的，第一次使用才需要格式化

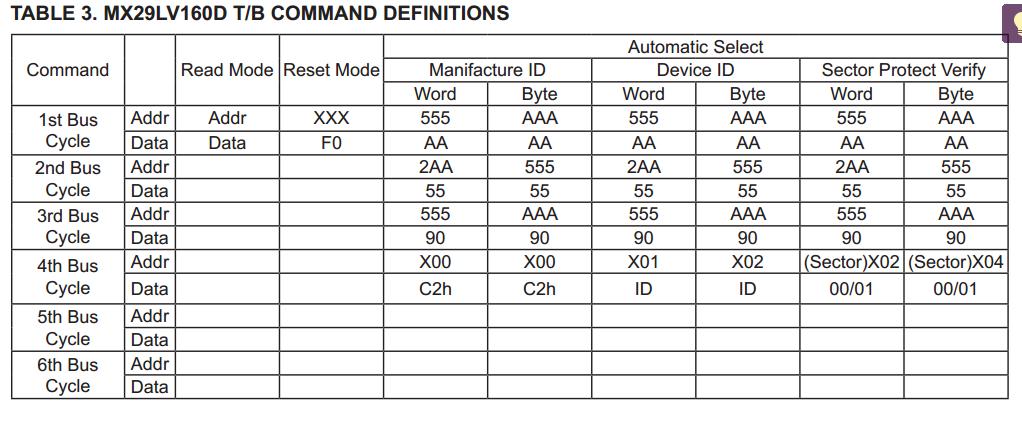
5、挂接 mount -t /mtdblock3 /mnt

# 10 NorFlash驱动

## 10.1 Nor Flash硬件原理

可以向RAM一样读，写需要特定时序发命令。





#### 如何发送地址

根据原理图知道，2440地址引脚A1和flash地址A0连接。这是因为flash是16bit.2440发一次地址能得到16bit数据。norflash看到的地址addr就相当于2440 发送的Addr>>1.

反过来就是Addr=addr<<1。那么为了使flash得到地址555那么2440发送的地址必须是555<<1。

根据上图，可知：

#### 读厂家ID命令顺序：

555地址写入AA,

2AA地址写入55，

555地址里面写入90，

读0地址得到厂家ID位C2h,

读1地址得到设备Id 22C4/ 2249

#### 使用uboot来体验nor flash的读操作。

md.b 读数据，一次读64B,nor flash可以直接读。

#### 读ID:

地址555<<1=AAA,2AA<<1=554。读ID命令：

AAA地址写入AA, mw.w aaa aa//写1word数据

554地址写入55， mw.w 554 55

AAA地址里面写入90， mw.w aaa 90

读0<<1地址得到厂家ID位C2h, md.w 0 1 //读1word数据

读1<<1地址得到设备Id 22C4/ 2249 md.w 2 1

退出读ID状态：任意地址写入f0 mw.w 0 f0

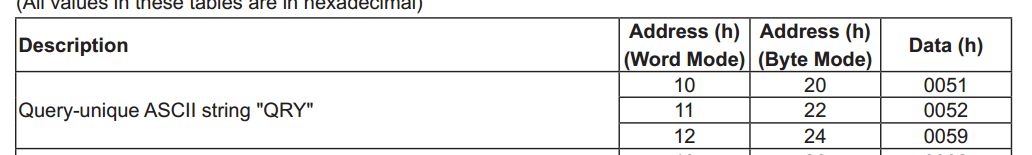
NOR Flash有两种规范：Jedec（老式flash接口）,cfi（通用flash接口，新规范）。

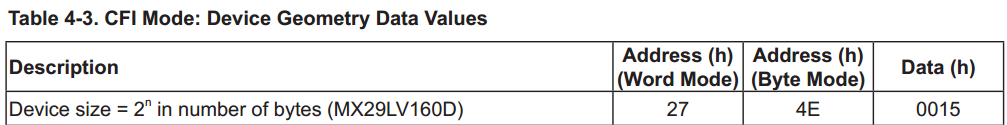
老式接口是通过读设备ID,然后和内核中jedec\_table[]查找匹配的设备，然后从表中获取设备信息，如果表中没有则需要自己添加信息。

#### 

对于支持CFI接口的芯片可以直接读，如上表：在55地址写入98进入cfi模式

进入 CFI 模式 往 55H 写入 98H （这之后便可以读数据





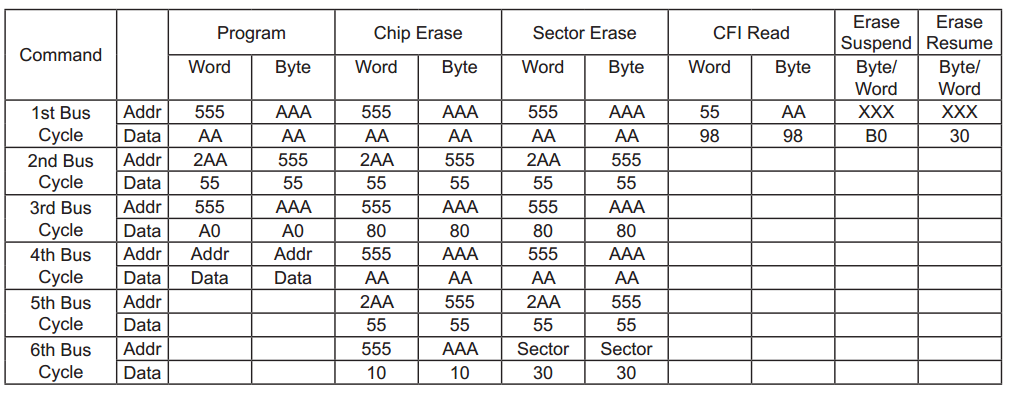
通过cfi读取芯片信息：  
读数据: 读 10H 得到 0051 ACSII 为“Q”  
读 11H 得到 0052 ACSII 为“R”  
读 12H 得到 0059 ACSII 为“Y”

读 27H 得到容量

2440 的 A1 接到 NOR 的 A0，所以 2440 发出(555h<<1), NOR 才能收到 555h 这个地址  
UBOOT 怎么操作？  
进入 CFI 模式 往 AAH 写入 98H mw.w aa 98  
读数据: 读 20H 得到 0051 md.w 20 1  
读 22H 得到 0052 md.w 22 1  
读 24H 得到 0059 md.w 24 1  
读 4EH 得到容量 md.w 4e 1 //读到15H，容量  
退出 CFI 模式 mw.w 0 f0 （往 0 地址写入“f0-复位”退出

“15H”十进制为“21”，那么 2 的 21 次方是“2097152”即 2M（除 1024\*1024）。

NORFLASH 支持不同的区域每次擦除的大小不同，如某区域是 32K 每次的擦除，而另一区域  
是 64K 每次的擦除。



#### 写数据:

根据上面表中program

在地址 0x100000 写入 0x1234（在前面写会破坏 UBOOT，故选择了 1M 处。 UBOOT 没有 1M.）

这里的 0x100000 1M 是 CPU 看到的地址。

#### UBOOT 中写：

md.w 100000 1 // 得到ffff 读出10000（1M）处的数据为“FFFF”

mw.w 100000 1234

md.w 100000 1 // 还是ffff 即写不进去

NOR 手册： program 烧写

往地址 555H 写 AAH

往地址 2AAH 写 55H

--上两个为解锁

往地址 555H 写 A0H

往地址 PA 写 PD

2440 的 A1 接到 NOR 的 A0，所以 2440 发出(555h<<1), NOR 才能收到 555h 这个地址。

#### UBOOT 怎么操作？

往地址AAAH写AAH mw.w aaa aa

往地址554H写55H mw.w 554 55

往地址AAAH写A0H mw.w aaa a0

往地址0x100000写1234h mw.w 100000 1234 （100000是CPU看到的地址）

## 10.2 Nor Flash 驱动程序

### 10.2.1 内核中驱动框架

physmap.c是内核中关于Nor Flash的驱动

#### static int \_\_init physmap\_init(void)

#### {

#### int err;

#### err = platform\_driver\_register(&physmap\_flash\_driver);

#### #ifdef PHYSMAP\_COMPAT

#### if (err == 0)

#### platform\_device\_register(&physmap\_flash);

#### #endif

#### return err;

#### }

//设备

#### static struct platform\_device physmap\_flash = {

#### .name = "physmap-flash",

#### .id = 0,

#### .dev = {

#### .platform\_data = &physmap\_flash\_data,

#### },

#### .num\_resources = 1,

#### .resource = &physmap\_flash\_resource,

#### };

//资源

#### static struct resource physmap\_flash\_resource = {

#### .start = CONFIG\_MTD\_PHYSMAP\_START,//基地址

#### .end = CONFIG\_MTD\_PHYSMAP\_START + CONFIG\_MTD\_PHYSMAP\_LEN - 1,//长度

#### .flags = IORESOURCE\_MEM,

#### };

//位宽设置

static struct physmap\_flash\_data physmap\_flash\_data = {

.width = CONFIG\_MTD\_PHYSMAP\_BANKWIDTH,//位宽

};

#### //驱动

#### static struct platform\_driver physmap\_flash\_driver = {

#### .probe = physmap\_flash\_probe,

#### .remove = physmap\_flash\_remove,

#### #ifdef CONFIG\_PM

#### .suspend = physmap\_flash\_suspend,

#### .resume = physmap\_flash\_resume,

#### .shutdown = physmap\_flash\_shutdown,

#### #endif

#### .driver = {

#### .name = "physmap-flash",

#### },

#### };

//probe函数

#### info->map.name = dev->dev.bus\_id;

#### info->map.phys = dev->resource->start;

#### info->map.size = dev->resource->end - dev->resource->start + 1;

#### info->map.bankwidth = physmap\_data->width;

#### info->map.set\_vpp = physmap\_data->set\_vpp;

#### simple\_map\_init(&info->map);

#### //static const char \*rom\_probe\_types[] = { "cfi\_probe", "jedec\_probe", "map\_rom", NULL };

#### probe\_type = rom\_probe\_types;

#### for (; info->mtd == NULL && \*probe\_type != NULL; probe\_type++)

#### info->mtd = do\_map\_probe(\*probe\_type, &info->map);

add\_mtd\_device(info->mtd);

### 10.2.2 使用内核中驱动测试

#### 1. make menuconfig -> Device Drivers -> Memory Technology Device (MTD) support //内存技术设备支持 -> Mapping drivers for chip access <M> CFI Flash device in physical memory map //M作为模块 (0x0) Physical start address of flash mapping // 物理基地址 写为0 (0x1000000) Physical length of flash mapping // 长度 这里写大些为16M,此长度要大于等于 NORFLASH的真实长度即可。因为它要ioremap。设置长度的意思是让它ioremap映射得到虚拟地址。 (2) Bank width in octets (NEW) // 位宽 16，单位是 octets（8 的意思） .故设为 2. 2. make modules（不需 make uImage）得到“physmap.ko”。 这是 NORFLASH 最底层硬件相关的驱动程序。 cp drivers/mtd/maps/physmap.ko /work/nfs\_root/first\_fs

#### 如果想分区的话，可以参考nand中添加分区方法。

## 10.3 自己写nor flashe驱动

/\*

 \* 参考 drivers\mtd\maps\physmap.c

 \*/

#include <linux/module.h>

#include <linux/types.h>

#include <linux/kernel.h>

#include <linux/init.h>

#include <linux/slab.h>

#include <linux/device.h>

#include <linux/platform\_device.h>

#include <linux/mtd/mtd.h>

#include <linux/mtd/map.h>

#include <linux/mtd/partitions.h>

#include <asm/io.h>

**static** **struct** map\_info \*s3c\_nor\_map;

**static** **struct** mtd\_info \*s3c\_nor\_mtd;

**static** **struct** mtd\_partition s3c\_nor\_parts[] = {

    [0] = {

        .name   = "bootloader\_nor",

        .size   = 0x00040000,

        .offset = 0,

    },

    [1] = {

        .name   = "root\_nor",

        .offset = MTDPART\_OFS\_APPEND,

        .size   = MTDPART\_SIZ\_FULL,

    }

};

**static** **int** s3c\_nor\_init(**void**)

{

    /\* 1. 分配map\_info结构体 \*/

    s3c\_nor\_map = kzalloc(**sizeof**(**struct** map\_info), GFP\_KERNEL);;

    /\* 2. 设置: 物理基地址(phys), 大小(size), 位宽(bankwidth), 虚拟基地址(virt) \*/

    s3c\_nor\_map->name = "s3c\_nor";

    s3c\_nor\_map->phys = 0;

    s3c\_nor\_map->size = 0x1000000; /\* >= NOR的真正大小 \*/

    s3c\_nor\_map->bankwidth = 2;

    s3c\_nor\_map->virt = ioremap(s3c\_nor\_map->phys, s3c\_nor\_map->size);

    simple\_map\_init(s3c\_nor\_map);

    /\* 3. 使用: 调用NOR FLASH协议层提供的函数来识别 \*/

    printk("use cfi\_probe\n");

    s3c\_nor\_mtd = do\_map\_probe("cfi\_probe", s3c\_nor\_map);

**if** (!s3c\_nor\_mtd)

    {

        printk("use jedec\_probe\n");

        s3c\_nor\_mtd = do\_map\_probe("jedec\_probe", s3c\_nor\_map);

    }

**if** (!s3c\_nor\_mtd)

    {

        iounmap(s3c\_nor\_map->virt);

        kfree(s3c\_nor\_map);

**return** -EIO;

    }

    /\* 4. add\_mtd\_partitions \*/

    add\_mtd\_partitions(s3c\_nor\_mtd, s3c\_nor\_parts, 2);

**return** 0;

}

**static** **void** s3c\_nor\_exit(**void**)

{

    del\_mtd\_partitions(s3c\_nor\_mtd);

    iounmap(s3c\_nor\_map->virt);

    kfree(s3c\_nor\_map);

}

module\_init(s3c\_nor\_init);

module\_exit(s3c\_nor\_exit);

MODULE\_LICENSE("GPL");

#### 测试

1. ls /dev/mtd\*
2. 2. insmod s3c\_nor.ko
3. ls /dev/mtd\*

/dev/mtd0是字符设备 /dev/mtdblock0是块设备 /dev/mtd0ro是只读

4. 格式化: flash\_eraseall -j /dev/mtd1  
格式化时用 “字符设备”  
Flash\_eraseall 格式 NAND 后，默认就是 yaffes 文件系统。  
对于 NORFLASH，格式化为jffs2文件系统比较好，所以使用-j选项

5. mount -t jffs2 /dev/mtdblock1 /mnt  
挂接时用 “块设备”。  
在/mnt 目录下操作文件

#### Nor Flash识别过程

do\_map\_probe(\*probe\_type, &info->map);

get\_mtd\_chip\_driver(name);

list\_for\_each(pos, &chip\_drvs\_list) {// 在chip\_drvs\_list查找驱动

this = list\_entry(pos, typeof(\*this), list);

if (!strcmp(this->name, name)) {

ret = this;

ret = drv->probe(map);

//chip\_drvs\_list链表在哪里设置

cfi\_probe\_init(void)//cfi接口芯片

register\_mtd\_chip\_driver

list\_add(&drv->list, &chip\_drvs\_list);

jedec\_probe\_init(void)//jedec芯片

register\_mtd\_chip\_driver(&jedec\_chipdrv);

//cfi接口识别过程

static struct mtd\_chip\_driver cfi\_chipdrv = {

.probe = cfi\_probe,

.name = "cfi\_probe",

.module = THIS\_MODULE

};

cfi\_probe

mtd\_do\_chip\_probe(map, &cfi\_chip\_probe);

cfi = genprobe\_ident\_chips(map, cp);

genprobe\_new\_chip(map, cp, &cfi)

cp->probe\_chip(map, 0, NULL, cfi)// 调用cfi\_chip\_probe

static struct chip\_probe cfi\_chip\_probe = {

.name = "CFI",

.probe\_chip = cfi\_probe\_chip

};

cfi\_probe\_chip //识别并且获取flash各种信息

cfi\_send\_gen\_cmd(0xF0, 0, base, map, cfi, cfi->device\_type, NULL);

cfi\_send\_gen\_cmd(0xFF, 0, base, map, cfi, cfi->device\_type, NULL);

cfi\_send\_gen\_cmd(0x98, 0x55, base, map, cfi, cfi->device\_type, NULL);

//jedec芯片识别过程

static struct mtd\_chip\_driver jedec\_chipdrv = {

.probe = jedec\_probe,

.name = "jedec\_probe",

.module = THIS\_MODULE

};

jedec\_probe

mtd\_do\_chip\_probe(map, &jedec\_chip\_probe);

cfi = genprobe\_ident\_chips(map, cp);

cp->probe\_chip(map, i << cfi.chipshift, chip\_map, &cfi);

static struct chip\_probe jedec\_chip\_probe = {

.name = "JEDEC",

.probe\_chip = jedec\_probe\_chip

};

jedec\_chip\_prob

cfi\_send\_gen\_cmd(0xaa, cfi->addr\_unlock1, base, map, cfi, cfi->device\_type, NULL);

cfi\_send\_gen\_cmd(0x55, cfi->addr\_unlock2, base, map, cfi, cfi->device\_type, NULL);

for (i = 0; i < ARRAY\_SIZE(jedec\_table); i++) {

if ( jedec\_match( base, map, cfi, &jedec\_table[i] ) ) {

在jedec\_table中查找需要的设备信息。

关于jedec\_table如何设置芯片信息和各项的意思查看uboot2012移植。

# 11 SPI设备驱动

## 11.1 spi协议

spi 接口：

MOSI:主机数据输出

MISO：主机数据输入

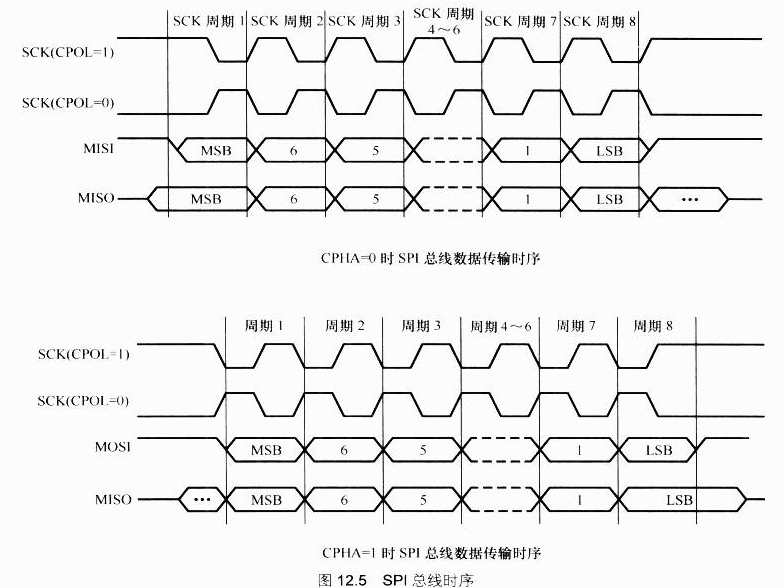
SCK：串行移位时钟

SS： 从机片选信号

数据在时钟的上升沿或着下降沿由MOSI输出，在紧接着的下降沿或者上升沿MISO读入。

这样8/16次时钟改变，完成8/16位数据传输。

spi模块可以根据外设要求，其输出的串行同步时钟极性CPOL和相位CPHA可以进行配置。如果CPOL=0,串行同步时钟的空闲状态为低电平；如果CPOL=1,串行同步时钟的空闲状态为高电平。如果CPHA=0,在串行同步时钟的第一个跳变沿（上升沿或者下降沿）数据被采集，如果CPHA=1,在串行同步时钟的第二个跳变沿（上升沿或下降沿）数据被采集。



MODE0：CPOL=0，CPHA=0; idle时候的是低电平，第一个边沿就是从低变到高，所以是上升沿；

MODE1：CPOL=0，CPHA=1;idle时候的是高电平，第一个边沿就是从高变到低，所以是下降沿；

MODE2：CPOL=1，CPHA=0; idle时候的是低电平，第二个边沿就是从高变到低，所以是下降沿；

MODE3: CPOL=1，CPHA=1; idle时候的是高电平，第一个边沿就是从低变到高，所以是上升沿；

w25q64 flash芯片支持MODE0和MODE3

## 11.1 spi设备驱动框架

spi驱动由spi设备驱动,spi核心层和主机控制器驱动三部分组成。

#### 由spi\_master描述一个主机控制器驱动

#### struct spi\_master {

#### struct class\_device cdev;

#### /\*总线（或控制器）编号，imx6q有5个spi控制器，0~4 \*/

#### s16 bus\_num;

#### /\* 片选数量，决定该控制器下面挂接多少个SPI设备，从设备的片选号不能大于这个数量 \*/

#### u16 num\_chipselect;

#### /\* 根据spi设备更新硬件配置。设置模式、时钟等，这个需要我们自己具体实现，

#### 主要设置SPI控制器和工作方式 \*/

#### int (\*setup)(struct spi\_device \*spi);

#### /\*加消息到队列的方法。这个函数不可睡眠。它的职责是安排发生的传送并且调用注册的回调函数complete()。这个不同的控制器要具体实现，传输数据最后都要调用这个函数 \*/

#### int (\*transfer)(struct spi\_device \*spi,struct spi\_message \*mesg);

#### /\* cleanup函数会在spidev\_release函数中被调用，spidev\_release

#### 被登记为spi dev的release函数\*/

#### void (\*cleanup)(struct spi\_device \*spi);

#### };

#### 分配、注册和注销SPI主机的API

#### struct spi\_master \*spi\_alloc\_master(struct device \*host, unsigned size);

int spi\_register\_master(struct spi\_master \*master);

void spi\_unregister\_master(struct spi\_master \*master);

#### Linux 中，用spi\_driver来描述一个spi外设驱动即client驱动

struct spi\_driver {

const struct spi\_device\_id \*id\_table; //新内核版本有该结构 支持的spi\_device设备表

int (\*probe)(struct spi\_device \*spi);//和spi匹配成功之后会调用这个方法。因此//这个方法需要对设备和私有数据进行初始化。

int (\*remove)(struct spi\_device \*spi);

void (\*shutdown)(struct spi\_device \*spi)//解除spi\_device和spi\_driver的绑定，//释放probe申请的资源。 ;

int (\*suspend)(struct spi\_device \*spi, pm\_message\_t mesg); //挂起

int (\*resume)(struct spi\_device \*spi);//恢复

struct device\_driver driver;

};

spi\_device代表一个外围spi设备，由master controller driver注册完成后扫描BSP中注册设备产生的设备链表并向spi\_bus注册产生。在内核中，每个spi\_device代表一个物理的spi设备

struct spi\_device {

struct device dev;

struct spi\_master \*master;//设备使用的master结构，挂在哪个主控制器下

u32 max\_speed\_hz;//通讯时钟最大频率

u8 chip\_select; //片选号，每个master支持多个spi\_device

u8 mode; //设备支持的模式，如片选是高or低？

#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 /\* 1时片选的有效信号是高电平chipselect active high? \*/

#define SPI\_LSB\_FIRST 0x08 /\* 发送时低比特在前per-word bits-on-wire \*/

u8 bits\_per\_word; //flash有8位，16位。一般spi flash为8位

int irq; //中断号

void \*controller\_state;

void \*controller\_data;//控制器相关的一些信息，指定片选的引脚，flash位数等

const char \*modalias; //设备驱动的名字

};

#### 采用spi\_transfer进行数据传输

spi\_transfer代表一个读写缓冲对，包含接收缓冲区及发送缓冲区，其实，spi\_transfer的发送是通过构建spi\_message实现，通过将spi\_transfer中的链表transfer\_list链接到spi\_message中的transfers，再以spi\_message形势向底层发送数据。每个spi\_transfer都可以对传输的一些参数进行设置，使得master controller按照它要求的参数进行数据发送。

#### struct spi\_transfer {

#### const void \*tx\_buf; //发送缓冲区，要写入设备的数据（必须是dma\_safe），或者为NULL

#### void \*rx\_buf;//接收缓冲区，要读取的数据缓冲（必须是dma\_safe），或者为NULL

#### unsigned len;//缓冲区长度，tx和rx的大小（字节数）。这里不是指它的和，而是各自的长度，它//们总是相等的。

#### dma\_addr\_t tx\_dma; //如果spi\_message.is\_dma\_mapped是真，这个是tx的dma地址

#### dma\_addr\_t rx\_dma; //如果spi\_message.is\_dma\_mapped是真，这个是rx的dma地址

#### unsigned cs\_change:1;//当前spi\_transfer发送完成之后重新片选。影响此次传输之后的片选。

#### //指示本次transfer结束之后是否要重新片选并调用setup改变设置。这个标志可以减少系统开销。

#### u8 bits\_per\_word; //每个字长的比特数，0代表使用spi\_device中的默认值8

#### u16 delay\_usecs;//发送完成一个spi\_transfer后延时时间，此次传输结束和片选改

#### //变之间的延时，之后就会启动另一个传输或者结束整个消息

#### u32 speed\_hz;//通信时钟。如果是0，使用默认值

#### struct list\_head transfer\_list;//用于链接到spi\_message，用来连接的双向链接节点

#### }

一次完整的SP传输流程可能不止包含一次spi\_transfer,它可能包含一个或者多个spi\_transfer,这些spi\_transfer最终通过spi\_message组织在一起。spi\_message用来原子的执行spi\_transfer表示的一串数组传输请求。这个传输队列是原子的，这意味着在这个消息完成之前不会有其它消息占用总线。消息的执行总是按照FIFO的顺序。向底层提交spi\_message的代码要负责管理它的内存空间。未显式初始化的内存需要使用0来初始化。先写入tx的数据，然后读取同样长度的数据。长度指示是len。如果tx\_buff是空指针，填充rx\_buff的时候会输出0（为了产生接收的时钟），如果rx\_buff是NULL，接收到的数据将被丢弃。

接收到的数据将被丢弃。

struct spi\_message {

struct list\_head transfers;//spi\_transfer链表队列，此次消息的传输段队列，一个消息可以//包含多个传输段。

struct spi\_device \*spi; //传输的目的设备

unsigned is\_dma\_mapped:1;//如果为真，此次调用提供dma和cpu虚拟地址。

void (\*complete)(void \*context); //异步调用完成后的回调函数

void \*context; //回调函数的参数

unsigned actual\_length;

int status; //该消息的发送结果，成功被置0，否则是一个负的错误码。

struct list\_head queue;//下面两个成员是给拥有本消息的驱动选用的。spi\_master会使用它们。//自己最好不要使用。

void \*state;

};

#### spi\_message\_init()可以初始化spi\_massage,而将spi\_transfer添加到spi\_message队列的方法：

#### void spi\_message\_add\_tail(struct spi\_transfer \*t, struct spi\_message \*m)

spi\_message的传输有同步和异步两种方式，使用同步API时，会阻塞等待这个消息处理完。

#### int spi\_sync(struct spi\_device \*spi, struct spi\_message \*message)//同步

int spi\_async(struct spi\_device \*spi, struct spi\_message \*message)//异步

#### /\*\*

#### \* spi\_write - SPI synchronous write同步写函数

#### \* @spi: device to which data will be written

#### \* @buf: data buffer

#### \* @len: data buffer size

#### \* Context: can sleep

#### \* This writes the buffer and returns zero or a negative error code.

#### \* Callable only from contexts that can sleep.

#### \*/

#### static inline int

#### spi\_write(struct spi\_device \*spi, const u8 \*buf, size\_t len)

#### {

#### struct spi\_transfer t = {

#### .tx\_buf = buf,

#### .len = len,

#### };

#### struct spi\_message m;

#### spi\_message\_init(&m);

#### spi\_message\_add\_tail(&t, &m);

#### return spi\_sync(spi, &m);

#### }

#### /\*\*

#### \* spi\_read - SPI synchronous read

#### \* @spi: device from which data will be read

#### \* @buf: data buffer

#### \* @len: data buffer size

#### \* Context: can sleep

#### \*

#### \* This reads the buffer and returns zero or a negative error code.

#### \* Callable only from contexts that can sleep.

#### \*/

#### static inline int

#### spi\_read(struct spi\_device \*spi, u8 \*buf, size\_t len)

#### {

#### struct spi\_transfer t = {

#### .rx\_buf = buf,

#### .len = len,

#### };

#### struct spi\_message m;

#### spi\_message\_init(&m);

#### spi\_message\_add\_tail(&t, &m);

#### return spi\_sync(spi, &m);

#### }

## spi设备信息

struct spi\_board\_info {

char modalias[KOBJ\_NAME\_LEN];//spi设备名字，设备驱动探测时会用到该项

const void \*platform\_data; //

void \*controller\_data;

int irq;

/\* slower signaling on noisy or low voltage boards \*/

u32 max\_speed\_hz;/\* max spi clock (SCK) speed in HZ \*/

u16 bus\_num;//记录了该spi设备是连接在哪个spi总线上的，所在总线的编号

u16 chip\_select;//该spi设备的片选编号

u8 mode;//这个spi设备支持spi总线的工作模式

};

#### spi\_bitbang

#### spi\_bitbang是具体的负责信息传输的数据结构，它维护一个workqueue\_struct，每收到

#### 一个消息，都会向其中添加一个work\_struct，由内核守护进程在将来的某个时间调用

#### 该work\_struct中的function进行消息发送。

#### struct spi\_bitbang {}

struct spi\_bitbang {

struct workqueue\_struct \*workqueue;

struct work\_struct work;

spinlock\_t lock;

struct list\_head queue;

u8 busy; //忙标志

u8 use\_dma;

struct spi\_master \*master;

//对数据传输进行设置

int (\*setup\_transfer)(struct spi\_device \*spi,struct spi\_transfer \*t);

void (\*chipselect)(struct spi\_device \*spi, int is\_on);//控制片选

#define BITBANG\_CS\_ACTIVE 1 /\* normally nCS, active low \*/

#define BITBANG\_CS\_INACTIVE 0

int (\*txrx\_bufs)(struct spi\_device \*spi, struct spi\_transfer \*t); //实际的数据传输函数

u32 (\*txrx\_word[4])(struct spi\_device \*spi,unsigned nsecs,u32 word, u8 bits);

};

#### SPI控制器驱动程序

#### SPI控制器不用关心设备的具体功能，它只负责把上层协议驱动准备好的数据按SPI总线的时序要求发送给SPI设备，同时把从设备收到的数据返回给上层的协议驱动，因此，内核把SPI控制器的驱动程序独立出来。SPI控制器驱动负责控制具体的控制器硬件，诸如DMA和中断操作等等，因为多个上层的协议驱动可能会通过控制器请求数据传输操作，所以，SPI控制器驱动同时也要负责对这些请求进行队列管理，保证先进先出的原则。

#### SPI通用接口封装层

#### 为了简化SPI驱动程序的编程工作，同时也为了降低协议驱动程序和控制器驱动程序的耦合程度，内核把控制器驱动和协议驱动的一些通用操作封装成标准的接口，加上一些通用的逻辑处理操作，组成了SPI通用接口封装层。这样的好处是，对于控制器驱动程序，只要实现标准的接口回调API，并把它注册到通用接口层即可，无需直接和协议层驱动程序进行交互。而对于协议层驱动来说，只需通过通用接口层提供的API即可完成设备和驱动的注册，并通过通用接口层的API完成数据的传输，无需关注SPI控制器驱动的实现细节。

#### SPI协议驱动程序

#### 上面我们提到，控制器驱动程序并不清楚和关注设备的具体功能，SPI设备的具体功

#### 能是由SPI协议驱动程序完成的，SPI协议驱动程序了解设备的功能和通信数据的协

#### 议格式。向下，协议驱动通过通用接口层和控制器交换数据，向上，协议驱动通常

#### 会根据设备具体的功能和内核的其它子系统进行交互，例如，和MTD层交互以便把S

#### PI接口的存储设备实现为某个文件系统，和TTY子系统交互把SPI设备实现为一个TTY设备，

#### 和网络子系统交互以便把一个SPI设备实现为一个网络设备，等等。当然，如果是一个

#### 专有的SPI设备，我们也可以按设备的协议要求，实现自己的专有协议驱动。

#### SPI通用设备驱动程序

#### 有时候，考虑到连接在SPI控制器上的设备的可变性，在内核没有配备相应的协议驱动

#### 程序，对于这种情况，内核为我们准备了通用的SPI设备驱动程序，该通用设备驱动程序

#### 向用户空间提供了控制SPI控制的控制接口，具体的协议控制和数据传输工作交由用户

#### 空间根据具体的设备来完成，在这种方式中，只能采用同步的方式和SPI设备进行通信，

#### 所以通常用于一些数据量较少的简单SPI设备。

#### 设备驱动编写：

目前有两种方法向内核注册一个spi设备。在稍微老点版本的内核（2.6.xx）中通过向内核注册struct spi\_board\_info对象，来申明一个spi设备。在比较新的内核中（3.xx）使用device tree的方式向内核申明并注册一个spi设备的。无论使用哪种方式，其实最终的目的就是为了建立一个struct spi\_deivce对象，并注册到spi子系统中

spi\_board\_info的方式（老内核方式）：

如何在板级代码中注册struct spi\_board\_info。

在代码kernel/arch/arm/mach-imx51-3ds.c

static struct spi\_board\_info mx51\_3ds\_spi\_nor\_device[] = {

{

.modalias = "m25p80", //spi设备名字，设备驱动探测时会用到该项

.max\_speed\_hz = 25000000, /\* max spi clock (SCK) speed in HZ \*/

.bus\_num = 1, //记录了该spi设备是连接在

//哪个spi总线上的，所在总线的编号

.chip\_select = 1, //该spi设备的片选编号

.mode = SPI\_MODE\_0, //这个spi设备支持spi总线的工作模式

.platform\_data = NULL,},

};

一个struct spi\_board\_info对象对应一个spi设备.注册设备到内核

spi\_register\_board\_info(mx51\_3ds\_spi\_nor\_device,

ARRAY\_SIZE(mx51\_3ds\_spi\_nor\_device));

spi\_register\_board\_info()这个函数调用做了两件事：

将struct spi\_board\_info对象转化为struct boardinfo对象，并将之连接到名为board\_list的全局链表中。 board\_list链表记录系统所有的spi设备对应的struct spi\_board\_info对象。 struct boardinfo就是对struct spi\_board\_info的一个简单封装，定义如下：

struct boardinfo {

struct list\_head list;

struct spi\_board\_info board\_info;

};

spi\_register\_board\_info

list\_add\_tail(&bi->list, &board\_list);//连接spi\_board\_info

list\_for\_each\_entry(master, &spi\_master\_list, list)

spi\_match\_master\_to\_boardinfo(master, &bi->board\_info)；

/\*遍历spi\_master\_list上的所有struct spi\_master对象，并将上面一步spi\_board\_info.bus\_num和spi\_master.bus\_num做比较，相等则创建struct spi\_device对象并注册。spi\_master\_list记录系统中所有的spi控制器。spi\_master.bus\_num和spi\_board\_info.bus\_num相等，则说明该spi\_board\_info所对应的spi设备连接在该spi控制器上。\*/

如何匹配master和dev

spi\_match\_master\_to\_boardinfo

if (master->bus\_num != bi->bus\_num)

return;

dev = spi\_new\_device(master, bi);

proxy=spi\_alloc\_device(master)

spi\_add\_device(proxy);

spi\_master.bus\_num和spi\_board\_info.bus\_num的简单比较。匹配的话就调用spi\_new\_device()创建struct spi\_device对象并注册.至此，我们就向系统中注册了一个spi设备。具体到我们这个例子就是向系统中注册了一个名为”m25p80”的spi设备

#### 2、spi设备驱动

#### \* 定义个一个struct spi\_driver\*/

#### static struct spi\_driver m25p80\_driver = {

#### .driver = {

#### .name = "m25p80",

#### .owner = THIS\_MODULE,

#### },

#### .id\_table = m25p\_ids,

#### .probe = m25p\_probe,

#### .remove = m25p\_remove,

#### }

#### static const struct spi\_device\_id m25p\_ids[] = {

#### { "at25fs010", INFO(0x1f6601, 0, 32 \* 1024, 4, SECT\_4K) },

#### { "at25fs040", INFO(0x1f6604, 0, 64 \* 1024, 8, SECT\_4K) },

}

其实这个数组就是列举了一些这个驱动所支持的spi flash设备。

#### 实现一个spi设备驱动所需要的步骤。

第一步：向内核注册一个struct spi\_device对象。可以通过spi\_board\_info的方式，如果支持device tree，也可以使用device tree的方式来注册。

第二步：实现一个struct spi\_driver对象，并实现probe和remove和id\_table接口。

第三步：在struct spi\_driver的probe接口中实现注册所需类型的设备的代码，并实现该类设备的访问接口。

## w25qflash

s3c24xx\_spi1\_set\_platdata(&s3c24xx\_spi1\_pdata, 18, 1);

struct s3c2410\_spi\_info {

int pin\_cs; /\* simple gpio cs \*/

unsigned int num\_cs; /\* total chipselects \*/

int bus\_num; /\* bus number to use. \*/

unsigned int use\_fiq:1; /\* use fiq \*/

void (\*gpio\_setup)(struct s3c2410\_spi\_info \*spi, int enable);

void (\*set\_cs)(struct s3c2410\_spi\_info \*spi, int cs, int pol);

};

#### void \_\_init s3c24xx\_spi1\_set\_platdata(struct s3c2410\_spi\_info \*pd,

#### int num\_cs,int bus\_num)

#### {

#### if (!pd) {

#### pr\_err("%s:Need to pass platform data\n", \_\_func\_\_);

#### return;

#### }

#### pd->bus\_num=bus\_num;

#### pd->num\_cs = num\_cs;

pd-> pin\_cs= S3C2410\_GPG(2);

#### if (!pd-> gpio\_setup)

#### pd-> gpio\_setup = s3c24xx\_spi1\_cfg\_gpio;

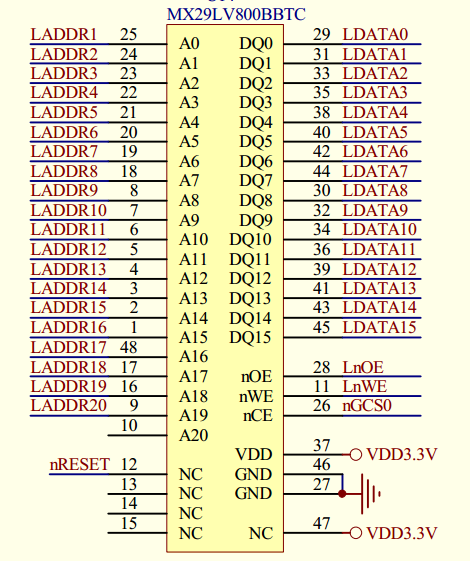
#### s3c\_set\_platdata(pd, sizeof(\*pd), &s3c24xx\_device\_spi1);

#### }

#### 挂载U盘：

1. insmod usb\_storage.ko
2. fdisk -l //查看U盘设备
3. mkdir /mnt/usb\_disk
4. **mount  /dev/sda   /mnt/usb\_disk**
5. umount /mnt/usb\_disk

# 内存控制器



#### 为何LADDR1接nor flash 的A0?

nor得到的地址就是addr>>1,即舍弃最后一位

例子1：假如cpu要从地址0读32位数据

mov r1,#0

ldr r0,[r1]

cpu要读4字节，但是nor一次只能提供2byte。

cpu不管如何得到数据，由内存控制器处理。

内存控制器需要发起两次传输

发出0地址，nor得到的地址为0，得到2byte

发出2地址，nor得到的地址为1，得到2byte

把1和2得到的4byte返回给cpu

因为有位宽设置，所以内存控制器知道每一次地址加2。

例子2：假如cpu要从地址3读8位数据

mov r1,#3

ldb r0,[r1]

内存控制器发出地址3，nor得到地址1，得到2byte

内存控制器从这两字节中取出高字节，返回给cpu