一 Fasync........................................ 2

二 同步互斥阻塞.................................... 2

三 poll............................................ 3

四 input子系统..................................... 3

五 LCD驱动程序..................................... 7

六 触摸屏驱动程序.................................. 9

七 USB驱动程序..................................... 10

八 块设备驱动程序.................................. 14

九 NAND FLASH驱动程序.............................. 16

十 NOR FLASH驱动程序............................... 20

十一 网卡驱动程序.................................... 23

十二 I2C驱动程序..................................... 24

十三 RTC驱动程序..................................... 25

十四 声卡驱动程序.....................................27

十五 uevnt........................................... 30

十六 裸板调试 ....................................... 33

十七 驱动调试 ....................................... 33

十八 应用调试 ....................................... 38

十九 自己写bootloader .............................. 43

二十 移植最新的u-boot .............................. 43

二十一 移植最新版本3.4.2内核.......................... 49

二十二 tslib编译使用方法............................... 52

二十三 移植所有驱动到3.4.2内核去...................... 54

二十四 3.4.2内核下的I2C驱动........................... 54

二十五 怎么看原理图.................................... 57

二十六 初接触开发板之基本操作.......................... 58

1. Fasync

为了使设备支持异步通知机制，驱动程序中涉及以下3项工作：

1. 支持F\_SETOWN命令，能在这个控制命令处理中设置filp->f\_owner为对应进程ID。

不过此项工作已由内核完成，设备驱动无须处理。

2. 支持F\_SETFL命令的处理，每当FASYNC标志改变时，驱动程序中的fasync()函数将得以执行。

驱动中应该实现fasync()函数。

3. 在设备资源可获得时，调用kill\_fasync()函数激发相应的信号

应用程序：

fcntl(fd, F\_SETOWN, getpid()); // 告诉内核，发给谁

Oflags = fcntl(fd, F\_GETFL);

fcntl(fd, F\_SETFL, Oflags | FASYNC); // 改变fasync标记，最终会调用到驱动的faync > fasync\_helper：初始化/释放fasync\_struct

1. 同步互斥阻塞

1. 原子操作

原子操作指的是在执行过程中不会被别的代码路径所中断的操作。

常用原子操作函数举例：

atomic\_t v = ATOMIC\_INIT(0); //定义原子变量v并初始化为0

atomic\_read(atomic\_t \*v); //返回原子变量的值

void atomic\_inc(atomic\_t \*v); //原子变量增加1

void atomic\_dec(atomic\_t \*v); //原子变量减少1

int atomic\_dec\_and\_test(atomic\_t \*v); //自减操作后测试其是否为0，为0则返回true，否则返回false。

2. 信号量

信号量（semaphore）是用于保护临界区的一种常用方法，只有得到信号量的进程才能执行临界区代码。

当获取不到信号量时，进程进入休眠等待状态。

定义信号量

struct semaphore sem;

初始化信号量

void sema\_init (struct semaphore \*sem, int val);

void init\_MUTEX(struct semaphore \*sem);//初始化为0

static DECLARE\_MUTEX(button\_lock); //定义互斥锁

获得信号量

void down(struct semaphore \* sem);

int down\_interruptible(struct semaphore \* sem);

int down\_trylock(struct semaphore \* sem);

释放信号量

void up(struct semaphore \* sem);

3. 阻塞

阻塞操作

是指在执行设备操作时若不能获得资源则挂起进程，直到满足可操作的条件后再进行操作。

被挂起的进程进入休眠状态，被从调度器的运行队列移走，直到等待的条件被满足。

非阻塞操作

进程在不能进行设备操作时并不挂起，它或者放弃，或者不停地查询，直至可以进行操作为止。

fd = open("...", O\_RDWR | O\_NONBLOCK);

1. Poll

1. poll > sys\_poll > do\_sys\_poll > poll\_initwait，poll\_initwait函数注册一下回调函数\_\_pollwait，它就是我们的驱动程序执行poll\_wait时，真正被调用的函数。

2. 接下来执行file->f\_op->poll，即我们驱动程序里自己实现的poll函数

它会调用poll\_wait把自己挂入某个队列，这个队列也是我们的驱动自己定义的；

它还判断一下设备是否就绪。

3. 如果设备未就绪，do\_sys\_poll里会让进程休眠一定时间

4. 进程被唤醒的条件有2：一是上面说的“一定时间”到了，二是被驱动程序唤醒。驱动程序发现条件就绪时，就把“某个队列”上挂着的进程唤醒，这个队列，就是前面通过poll\_wait把本进程挂过去的队列。

1. 如果驱动程序没有去唤醒进程，那么chedule\_timeout(\_\_timeou)超时后，会重复2、3动作，直到应用程序的poll调用传入的时间到达。
2. input子系统

drivers/input/input.c：

input\_init > err = register\_chrdev(INPUT\_MAJOR, "input", &input\_fops);

static const struct file\_operations input\_fops = {

.owner = THIS\_MODULE,

.open = input\_open\_file,

};

问：怎么读按键？

input\_open\_file

struct input\_handler \*handler = input\_table[iminor(inode) >> 5];

new\_fops = fops\_get(handler->fops) // =>&evdev\_fops

file->f\_op = new\_fops;

err = new\_fops->open(inode, file);

app: read > ... > file->f\_op->read

input\_table数组由谁构造？

input\_register\_handler

注册input\_handler：

input\_register\_handler

// 放入数组

input\_table[handler->minor >> 5] = handler;

// 放入链表

list\_add\_tail(&handler->node, &input\_handler\_list);

// 对于每个input\_dev，调用input\_attach\_handler

list\_for\_each\_entry(dev, &input\_dev\_list, node)

input\_attach\_handler(dev, handler); // 根据input\_handler的id\_table判断能否支持这个input\_dev

注册输入设备:

input\_register\_device

// 放入链表

list\_add\_tail(&dev->node, &input\_dev\_list);

// 对于每一个input\_handler，都调用input\_attach\_handler

list\_for\_each\_entry(handler, &input\_handler\_list, node)

input\_attach\_handler(dev, handler); // 根据input\_handler的id\_table判断能否支持这个input\_dev

input\_attach\_handler

id = input\_match\_device(handler->id\_table, dev);

error = handler->connect(handler, dev, id);

注册input\_dev或input\_handler时，会两两比较左边的input\_dev和右边的input\_handler,

根据input\_handler的id\_table判断这个input\_handler能否支持这个input\_dev，

如果能支持，则调用input\_handler的connect函数建立"连接"

怎么建立连接？

1. 分配一个input\_handle结构体

2.

input\_handle.dev = input\_dev; // 指向左边的input\_dev

input\_handle.handler = input\_handler; // 指向右边的input\_handler

3. 注册：

input\_handler->h\_list = &input\_handle;

inpu\_dev->h\_list = &input\_handle;

evdev\_connect

evdev = kzalloc(sizeof(struct evdev), GFP\_KERNEL); // 分配一个input\_handle

// 设置

evdev->handle.dev = dev; // 指向左边的input\_dev

evdev->handle.name = evdev->name;

evdev->handle.handler = handler; // 指向右边的input\_handler

evdev->handle.private = evdev;

// 注册

error = input\_register\_handle(&evdev->handle);

怎么读按键？

app: read

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

.......

evdev\_read

// 无数据并且是非阻塞方式打开，则立刻返回

if (client->head == client->tail && evdev->exist && (file->f\_flags & O\_NONBLOCK))

return -EAGAIN;

// 否则休眠

retval = wait\_event\_interruptible(evdev->wait,

client->head != client->tail || !evdev->exist);

谁来唤醒？

evdev\_event

wake\_up\_interruptible(&evdev->wait);

evdev\_event被谁调用？

猜：应该是硬件相关的代码，input\_dev那层调用的

在设备的中断服务程序里，确定事件是什么，然后调用相应的input\_handler的event处理函数

gpio\_keys\_isr

// 上报事件

input\_event(input, type, button->code, !!state);

input\_sync(input);

input\_event(struct input\_dev \*dev, unsigned int type, unsigned int code, int value)

struct input\_handle \*handle;

list\_for\_each\_entry(handle, &dev->h\_list, d\_node)

if (handle->open)

handle->handler->event(handle, type, code, value);

怎么写符合输入子系统框架的驱动程序？

1. 分配一个input\_dev结构体

2. 设置

3. 注册

4. 硬件相关的代码，比如在中断服务程序里上报事件

struct input\_dev {

void \*private;

const char \*name;

const char \*phys;

const char \*uniq;

struct input\_id id;

unsigned long evbit[NBITS(EV\_MAX)]; // 表示能产生哪类事件

unsigned long keybit[NBITS(KEY\_MAX)]; // 表示能产生哪些按键

unsigned long relbit[NBITS(REL\_MAX)]; // 表示能产生哪些相对位移事件, x,y,滚轮

unsigned long absbit[NBITS(ABS\_MAX)]; // 表示能产生哪些绝对位移事件, x,y

unsigned long mscbit[NBITS(MSC\_MAX)];

unsigned long ledbit[NBITS(LED\_MAX)];

unsigned long sndbit[NBITS(SND\_MAX)];

unsigned long ffbit[NBITS(FF\_MAX)];

unsigned long swbit[NBITS(SW\_MAX)];

测试:

1.

hexdump /dev/event1 (open(/dev/event1), read(), )

秒 微秒 类 code value

0000000 0bb2 0000 0e48 000c 0001 0026 0001 0000

0000010 0bb2 0000 0e54 000c 0000 0000 0000 0000

0000020 0bb2 0000 5815 000e 0001 0026 0000 0000

0000030 0bb2 0000 581f 000e 0000 0000 0000 0000

2. 如果没有启动QT：

cat /dev/tty1

按:s2,s3,s4

就可以得到ls

或者：

exec 0</dev/tty1

然后可以使用按键来输入

3. 如果已经启动了QT：

可以点开记事本

然后按:s2,s3,s4

1. LCD驱动程序

假设

app: open("/dev/fb0", ...) 主设备号: 29, 次设备号: 0

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

kernel:

fb\_open

int fbidx = iminor(inode);

struct fb\_info \*info = = registered\_fb[0];

app: read()

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

kernel:

fb\_read

int fbidx = iminor(inode);

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

if (info->fbops->fb\_read)

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

src = (u32 \_\_iomem \*) (info->screen\_base + p);

dst = buffer;

\*dst++ = fb\_readl(src++);

copy\_to\_user(buf, buffer, c)

问1. registered\_fb在哪里被设置？

答1. register\_framebuffer

怎么写LCD驱动程序？

1. 分配一个fb\_info结构体: framebuffer\_alloc

2. 设置

3. 注册: register\_framebuffer

4. 硬件相关的操作

测试：

1. make menuconfig去掉原来的驱动程序

-> Device Drivers

-> Graphics support

<M> S3C2410 LCD framebuffer support

2. make uImage

make modules

3. 使用新的uImage启动开发板:

4.

insmod cfbcopyarea.ko

insmod cfbfillrect.ko

insmod cfbimgblt.ko

insmod lcd.ko

echo hello > /dev/tty1 // 可以在LCD上看见hello

cat lcd.ko > /dev/fb0 // 花屏

5. 修改 /etc/inittab

tty1::askfirst:-/bin/sh

用新内核重启开发板

insmod cfbcopyarea.ko

insmod cfbfillrect.ko

insmod cfbimgblt.ko

insmod lcd.ko

insmod buttons.ko

1. 触摸屏驱动程序

测试2th~7th：

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

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

使用:

1. USB设备驱动程序

现象：把USB设备接到PC

1. 右下角弹出"发现android phone"

2. 跳出一个对话框，提示你安装驱动程序

问1. 既然还没有"驱动程序"，为何能知道是"android phone"

答1. windows里已经有了USB的总线驱动程序，接入USB设备后，是"总线驱动程序"知道你是"android phone"

提示你安装的是"设备驱动程序"

USB总线驱动程序负责：识别USB设备, 给USB设备找到对应的驱动程序

问2. USB设备种类非常多，为什么一接入电脑，就能识别出来？

答2. PC和USB设备都得遵守一些规范。

比如：USB设备接入电脑后，PC机会发出"你是什么"？

USB设备就必须回答"我是xxx", 并且回答的语言必须是中文

USB总线驱动程序会发出某些命令想获取设备信息(描述符)，

USB设备必须返回"描述符"给PC

问3. PC机上接有非常多的USB设备，怎么分辨它们？

USB接口只有4条线: 5V,GND,D-,D+

答3. 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

接在USB总线上的每一个USB设备都有自己的编号(地址)

PC机想访问某个USB设备时，发出的命令都含有对应的编号(地址)

问4. USB设备刚接入PC时，还没有编号；那么PC怎么把"分配的编号"告诉它？

答4. 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编号和它通信。

问5. 为什么一接入USB设备，PC机就能发现它？

答5. PC的USB口内部，D-和D+接有15K的下拉电阻，未接USB设备时为低电平

USB设备的USB口内部，D-或D+接有1.5K的上拉电阻；它一接入PC，就会把PC USB口的D-或D+拉高，从硬件的角度通知PC有新设备接入

其他概念:

1. USB是主从结构的

所有的USB传输，都是从USB主机这方发起；USB设备没有"主动"通知USB主机的能力。

例子：USB鼠标滑动一下立刻产生数据，但是它没有能力通知PC机来读数据，只能被动地等得PC机来读。

2. USB的传输类型:

a. 控制传输：可靠，时间有保证，比如：USB设备的识别过程

b. 批量传输: 可靠, 时间没有保证, 比如：U盘

c. 中断传输：可靠，实时，比如：USB鼠标

d. 实时传输：不可靠，实时，比如：USB摄像头

3. USB传输的对象：端点(endpoint)

我们说"读U盘"、"写U盘"，可以细化为：把数据写到U盘的端点1，从U盘的端点2里读出数据

除了端点0外，每一个端点只支持一个方向的数据传输

端点0用于控制传输，既能输出也能输入

4. 每一个端点都有传输类型，传输方向

5. 术语里、程序里说的输入(IN)、输出(OUT) "都是" 基于USB主机的立场说的。

比如鼠标的数据是从鼠标传到PC机, 对应的端点称为"输入端点"

6. USB总线驱动程序的作用

a. 识别USB设备

b. 查找并安装对应的设备驱动程序

c. 提供USB读写函数

USB驱动程序框架:

app:

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

USB设备驱动程序 // 知道数据含义

内核 --------------------------------------

USB总线驱动程序 // 1. 识别, 2. 找到匹配的设备驱动, 3. 提供USB读写函数 (它不知道数据含义)

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

USB主机控制器

UHCI OHCI EHCI

硬件 -----------

USB设备

UHCI: intel, 低速(1.5Mbps)/全速(12Mbps)

OHCI: microsoft 低速/全速

EHCI: 高速(480Mbps)

USB总线驱动程序的作用

1. 识别USB设备

1.1 分配地址

1.2 并告诉USB设备(set address)

1.3 发出命令获取描述符

描述符的信息可以在include\linux\usb\Ch9.h看到

2. 查找并安装对应的设备驱动程序

3. 提供USB读写函数

把USB设备接到开发板上，看输出信息:

usb 1-1: new full speed USB device using s3c2410-ohci and address 2

usb 1-1: configuration #1 chosen from 1 choice

scsi0 : SCSI emulation for USB Mass Storage devices

scsi 0:0:0:0: Direct-Access HTC Android Phone 0100 PQ: 0 ANSI: 2

sd 0:0:0:0: [sda] Attached SCSI removable disk

拔掉

usb 1-1: USB disconnect, address 2

再接上:

usb 1-1: new full speed USB device using s3c2410-ohci and address 3

usb 1-1: configuration #1 chosen from 1 choice

scsi1 : SCSI emulation for USB Mass Storage devices

scsi 1:0:0:0: Direct-Access HTC Android Phone 0100 PQ: 0 ANSI: 2

sd 1:0:0:0: [sda] Attached SCSI removable disk

在内核目录下搜:

grep "USB device using" \* -nR

drivers/usb/core/hub.c:2186: "%s %s speed %sUSB device using %s and address %d\n",

hub\_irq

kick\_khubd

hub\_thread

hub\_events

hub\_port\_connect\_change

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

dev->dev.bus = &usb\_bus\_type;

choose\_address(udev); // 给新设备分配编号(地址)

hub\_port\_init // usb 1-1: new full speed USB device using s3c2410-ohci and address 3

hub\_set\_address // 把编号(地址)告诉USB设备

usb\_get\_device\_descriptor(udev, 8); // 获取设备描述符

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

usb\_new\_device(udev)

err = usb\_get\_configuration(udev); // 把所有的描述符都读出来，并解析

usb\_parse\_configuration

device\_add // 把device放入usb\_bus\_type的dev链表,

// 从usb\_bus\_type的driver链表里取出usb\_driver，

// 把usb\_interface和usb\_driver的id\_table比较

// 如果能匹配，调用usb\_driver的probe

<LINUX内核源代码情景分析>

怎么写USB设备驱动程序？

1. 分配/设置usb\_driver结构体

.id\_table

.probe

.disconnect

2. 注册

测试1th/2th:

1. make menuconfig去掉原来的USB鼠标驱动

-> Device Drivers

-> HID Devices

<> USB Human Interface Device (full HID) support

2. make uImage 并使用新的内核启动

3. insmod usbmouse\_as\_key.ko

4. 在开发板上接入、拔出USB鼠标

测试3th:

1. insmod usbmouse\_as\_key.ko

2. ls /dev/event\*

3. 接上USB鼠标

4. ls /dev/event\*

5. 操作鼠标观察数据

测试4th:

1. insmod usbmouse\_as\_key.ko

2. ls /dev/event\*

3. 接上USB鼠标

4. ls /dev/event\*

5. cat /dev/tty1 然后按鼠标键

6. hexdump /dev/event0

1. 块设备驱动程序

框架：

app: open,read,write "1.txt"

--------------------------------------------- 文件的读写

文件系统: vfat, ext2, ext3, yaffs2, jffs2 (把文件的读写转换为扇区的读写)

-----------------ll\_rw\_block----------------- 扇区的读写

1. 把"读写"放入队列

2. 调用队列的处理函数(优化/调顺序/合并)

块设备驱动程序

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

硬件: 硬盘,flash

<LINUX内核源代码情景分析>

分析ll\_rw\_block

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 (block input/output)

submit\_bio(rw, bio);

// 通用的构造请求: 使用bio来构造请求(request)

generic\_make\_request(bio);

\_\_generic\_make\_request(bio);

request\_queue\_t \*q = bdev\_get\_queue(bio->bi\_bdev); // 找到队列

// 调用队列的"构造请求函数"

ret = q->make\_request\_fn(q, bio);

// 默认的函数是\_\_make\_request

\_\_make\_request

// 先尝试合并

elv\_merge(q, &req, bio);

// 如果合并不成，使用bio构造请求

init\_request\_from\_bio(req, bio);

// 把请求放入队列

add\_request(q, req);

// 执行队列

\_\_generic\_unplug\_device(q);

// 调用队列的"处理函数"

q->request\_fn(q);

怎么写块设备驱动程序呢？

1. 分配gendisk: alloc\_disk

2. 设置

2.1 分配/设置队列: request\_queue\_t // 它提供读写能力

blk\_init\_queue

2.2 设置gendisk其他信息 // 它提供属性: 比如容量

3. 注册: add\_disk

参考：

drivers\block\xd.c

drivers\block\z2ram.c

测试3th,4th:

在开发板上:

1. insmod ramblock.ko

2. 格式化: mkdosfs /dev/ramblock

3. 挂接: mount /dev/ramblock /tmp/

4. 读写文件: cd /tmp, 在里面vi文件

5. cd /; umount /tmp/

6. cat /dev/ramblock > /mnt/ramblock.bin

7. 在PC上查看ramblock.bin

sudo mount -o loop ramblock.bin /mnt

测试5th:

1. insmod ramblock.ko

2. ls /dev/ramblock\*

3. fdisk /dev/ramblock

1. NAND FLASH驱动程序

NAND FLASH是一个存储芯片

那么: 这样的操作很合理"读地址A的数据，把数据B写到地址A"

问1. 原理图上NAND FLASH和S3C2440之间只有数据线，

怎么传输地址？

答1．在DATA0～DATA7上既传输数据，又传输地址

当ALE为高电平时传输的是地址，

问2. 从NAND FLASH芯片手册可知，要操作NAND FLASH需要先发出命令

怎么传入命令？

答2．在DATA0～DATA7上既传输数据，又传输地址，也传输命令

当ALE为高电平时传输的是地址，

当CLE为高电平时传输的是命令

当ALE和CLE都为低电平时传输的是数据

问3. 数据线既接到NAND FLASH，也接到NOR FLASH，还接到SDRAM、DM9000等等

那么怎么避免干扰？

答3. 这些设备，要访问之必须"选中"，

没有选中的芯片不会工作，相当于没接一样

问4. 假设烧写NAND FLASH，把命令、地址、数据发给它之后，

NAND FLASH肯定不可能瞬间完成烧写的，

怎么判断烧写完成？

答4. 通过状态引脚RnB来判断：它为高电平表示就绪，它为低电平表示正忙

问5. 怎么操作NAND FLASH呢？

答5. 根据NAND FLASH的芯片手册，一般的过程是：

发出命令

发出地址

发出数据/读数据

NAND FLASH S3C2440

发命令 选中芯片

CLE设为高电平 NFCMMD=命令值

在DATA0~DATA7上输出命令值

发出一个写脉冲

发地址 选中芯片 NFADDR=地址值

ALE设为高电平

在DATA0~DATA7上输出地址值

发出一个写脉冲

发数据 选中芯片 NFDATA=数据值

ALE,CLE设为低电平

在DATA0~DATA7上输出数据值

发出一个写脉冲

读数据 选中芯片 val=NFDATA

发出读脉冲

读DATA0~DATA7的数据

用UBOOT来体验NAND FLASH的操作：

1. 读ID

S3C2440 u-boot

选中 NFCONT的bit1设为0 md.l 0x4E000004 1; mw.l 0x4E000004 1

发出命令0x90 NFCMMD=0x90 mw.b 0x4E000008 0x90

发出地址0x00 NFADDR=0x00 mw.b 0x4E00000C 0x00

读数据得到0xEC val=NFDATA md.b 0x4E000010 1

读数据得到device code val=NFDATA md.b 0x4E000010 1

0xda

退出读ID的状态 NFCMMD=0xff mw.b 0x4E000008 0xff

2. 读内容: 读0地址的数据

使用UBOOT命令:

nand dump 0

Page 00000000 dump:

17 00 00 ea 14 f0 9f e5 14 f0 9f e5 14 f0 9f e5

S3C2440 u-boot

选中 NFCONT的bit1设为0 md.l 0x4E000004 1; mw.l 0x4E000004 1

发出命令0x00 NFCMMD=0x00 mw.b 0x4E000008 0x00

发出地址0x00 NFADDR=0x00 mw.b 0x4E00000C 0x00

发出地址0x00 NFADDR=0x00 mw.b 0x4E00000C 0x00

发出地址0x00 NFADDR=0x00 mw.b 0x4E00000C 0x00

发出地址0x00 NFADDR=0x00 mw.b 0x4E00000C 0x00

发出地址0x00 NFADDR=0x00 mw.b 0x4E00000C 0x00

发出命令0x30 NFCMMD=0x30 mw.b 0x4E000008 0x30

读数据得到0x17 val=NFDATA md.b 0x4E000010 1

读数据得到0x00 val=NFDATA md.b 0x4E000010 1

读数据得到0x00 val=NFDATA md.b 0x4E000010 1

读数据得到0xea val=NFDATA md.b 0x4E000010 1

退出读状态 NFCMMD=0xff mw.b 0x4E000008 0xff

NAND FLASH驱动程序层次

看内核启动信息

S3C24XX NAND Driver, (c) 2004 Simtec Electronics

s3c2440-nand s3c2440-nand: Tacls=3, 30ns Twrph0=7 70ns, Twrph1=3 30ns

NAND device: Manufacturer ID: 0xec, Chip ID: 0xda (Samsung NAND 256MiB 3,3V 8-bit)

Scanning device for bad blocks

Bad eraseblock 256 at 0x02000000

Bad eraseblock 257 at 0x02020000

Bad eraseblock 319 at 0x027e0000

Bad eraseblock 606 at 0x04bc0000

Bad eraseblock 608 at 0x04c00000

Creating 4 MTD partitions on "NAND 256MiB 3,3V 8-bit":

0x00000000-0x00040000 : "bootloader"

0x00040000-0x00060000 : "params"

0x00060000-0x00260000 : "kernel"

0x00260000-0x10000000 : "root"

搜"S3C24XX NAND Driver"

S3c2410.c (drivers\mtd\nand)

s3c2410\_nand\_inithw

s3c2410\_nand\_init\_chip

nand\_scan // drivers/mtd/nand/nand\_base.c 根据nand\_chip的底层操作函数识别NAND FLASH，构造mtd\_info

nand\_scan\_ident

nand\_set\_defaults

if (!chip->select\_chip)

chip->select\_chip = nand\_select\_chip; // 默认值不适用

if (chip->cmdfunc == NULL)

chip->cmdfunc = nand\_command;

chip->cmd\_ctrl(mtd, command, ctrl);

if (!chip->read\_byte)

chip->read\_byte = nand\_read\_byte;

readb(chip->IO\_ADDR\_R);

if (chip->waitfunc == NULL)

chip->waitfunc = nand\_wait;

chip->dev\_ready

nand\_get\_flash\_type

chip->select\_chip(mtd, 0);

chip->cmdfunc(mtd, NAND\_CMD\_READID, 0x00, -1);

\*maf\_id = chip->read\_byte(mtd);

dev\_id = chip->read\_byte(mtd);

nand\_scan\_tail

mtd->erase = nand\_erase;

mtd->read = nand\_read;

mtd->write = nand\_write;

s3c2410\_nand\_add\_partition

add\_mtd\_partitions

add\_mtd\_device

list\_for\_each(this, &mtd\_notifiers) { // 问. mtd\_notifiers在哪设置

// 答. drivers/mtd/mtdchar.c,mtd\_blkdev.c调用register\_mtd\_user

struct mtd\_notifier \*not = list\_entry(this, struct mtd\_notifier, list);

not->add(mtd);

// mtd\_notify\_add 和 blktrans\_notify\_add

先看字符设备的mtd\_notify\_add

class\_device\_create

class\_device\_create

再看块设备的blktrans\_notify\_add

list\_for\_each(this, &blktrans\_majors) { // 问. blktrans\_majors在哪设置

// 答. drivers\mtd\mdblock.c或mtdblock\_ro.c register\_mtd\_blktrans

struct mtd\_blktrans\_ops \*tr = list\_entry(this, struct mtd\_blktrans\_ops, list);

tr->add\_mtd(tr, mtd);

mtdblock\_add\_mtd (drivers\mtd\mdblock.c)

add\_mtd\_blktrans\_dev

alloc\_disk

gd->queue = tr->blkcore\_priv->rq; // tr->blkcore\_priv->rq = blk\_init\_queue(mtd\_blktrans\_request, &tr->blkcore\_priv->queue\_lock);

add\_disk

测试4th:

1. make menuconfig去掉内核自带的NAND FLASH驱动

-> Device Drivers

-> Memory Technology Device (MTD) support

-> NAND Device Support

< > NAND Flash support for S3C2410/S3C2440 SoC

2. make uImage

使用新内核启动, 并且使用NFS作为根文件系统

3. insmod s3c\_nand.ko

4. 格式化 (参考下面编译工具)

flash\_eraseall /dev/mtd3 // yaffs

5. 挂接

mount -t yaffs /dev/mtdblock3 /mnt

6. 在/mnt目录下建文件

编译工具：

1. tar xjf mtd-utils-05.07.23.tar.bz2

2. cd mtd-utils-05.07.23/util

修改Makefile:

#CROSS=arm-linux-

改为

CROSS=arm-linux-

3. make

4. cp flash\_erase flash\_eraseall /work/nfs\_root/first\_fs/bin/

1. NOR FLASH驱动程序

使用UBOOT体验NOR FLASH的操作(开发板设为NOR启动，进入UBOOT)

先使用OpenJTAG烧写UBOOT到NOR FLASH

1. 读数据

md.b 0

2. 读ID

NOR手册上:

往地址555H写AAH

往地址2AAH写55H

往地址555H写90H

读0地址得到厂家ID: C2H

读1地址得到设备ID: 22DAH或225BH

退出读ID状态: 给任意地址写F0H

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554写55H mw.w 554 55

往地址AAAH写90H mw.w aaa 90

读0地址得到厂家ID: C2H md.w 0 1

读2地址得到设备ID: 22DAH或225BH md.w 2 1

退出读ID状态: mw.w 0 f0

3. NOR有两种规范, jedec, cfi(common flash interface)

读取CFI信息

NOR手册：

进入CFI模式 往55H写入98H

读数据: 读10H得到0051

读11H得到0052

读12H得到0059

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

退出CFI模式 mw.w 0 f0

4. 写数据: 在地址0x100000写入0x1234

md.w 100000 1 // 得到ffff

mw.w 100000 1234

md.w 100000 1 // 还是ffff

NOR手册：

往地址555H写AAH

往地址2AAH写55H

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

NOR FLASH驱动程序框架

测试1:通过配置内核支持NOR FLASH

1. make menuconfig

-> Device Drivers

-> Memory Technology Device (MTD) support

-> Mapping drivers for chip access

<M> CFI Flash device in physical memory map

(0x0) Physical start address of flash mapping // 物理基地址

(0x1000000) Physical length of flash mapping // 长度

(2) Bank width in octets (NEW) // 位宽

2. make modules

cp drivers/mtd/maps/physmap.ko /work/nfs\_root/first\_fs

3. 启动开发板

ls /dev/mtd\*

insmod physmap.ko

ls /dev/mtd\*

cat /proc/mtd

测试2: 使用自己写的驱动程序:

1. ls /dev/mtd\*

2. insmod s3c\_nor.ko

3. ls /dev/mtd\*

4. 格式化: flash\_eraseall -j /dev/mtd1

5. mount -t jffs2 /dev/mtdblock1 /mnt

在/mnt目录下操作文件

NOR FLASH识别过程:

do\_map\_probe("cfi\_probe", s3c\_nor\_map);

drv = get\_mtd\_chip\_driver(name)

ret = drv->probe(map); // cfi\_probe.c

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)

cfi\_probe\_chip

// 进入CFI模式

cfi\_send\_gen\_cmd(0x98, 0x55, base, map, cfi, cfi->device\_type, NULL);

// 看是否能读出"QRY"

qry\_present(map,base,cfi)

.....

do\_map\_probe("jedec\_probe", s3c\_nor\_map);

drv = get\_mtd\_chip\_driver(name)

ret = drv->probe(map); // jedec\_probe

jedec\_probe

mtd\_do\_chip\_probe(map, &jedec\_chip\_probe);

genprobe\_ident\_chips(map, cp);

genprobe\_new\_chip(map, cp, &cfi)

cp->probe\_chip(map, 0, NULL, cfi)

jedec\_probe\_chip

// 解锁

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);

// 读ID命令

cfi\_send\_gen\_cmd(0x90, cfi->addr\_unlock1, base, map, cfi, cfi->device\_type, NULL);

// 得到厂家ID，设备ID

cfi->mfr = jedec\_read\_mfr(map, base, cfi);

cfi->id = jedec\_read\_id(map, base, cfi);

// 和数组比较

jedec\_table

1. 网卡驱动程序

网卡驱动程序框架:

app: socket

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

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

--------------- 若干层网络协议--纯软件

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

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

hard\_start\_xmit|| /\

\/ || netif\_rx sk\_buff

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

硬件相关的驱动程序(要提供hard\_start\_xmit, 有数据时要用netif\_rx上报)

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

硬件

怎么写网卡驱动程序？

1. 分配一个net\_device结构体

2. 设置:

2.1 发包函数: hard\_start\_xmit

2.2 收到数据时(在中断处理函数里)用netif\_rx上报数据

2.3 其他设置

3. 注册: register\_netdevice

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 // 成功

ping 3.3.3.4 // 死机

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 192.168.1.17

ping 192.168.1.1

1. I2C驱动程序

i2c\_add\_driver

i2c\_register\_driver

driver->driver.bus = &i2c\_bus\_type;

driver\_register(&driver->driver);

list\_for\_each\_entry(adapter, &adapters, list) {

driver->attach\_adapter(adapter);

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

i2c\_probe\_address // 发出S信号,发出设备地址(来自addr\_data)

i2c\_smbus\_xfer

i2c\_smbus\_xfer\_emulated

i2c\_transfer

adap->algo->master\_xfer // s3c24xx\_i2c\_xfer

怎么写I2C设备驱动程序？

1. 分配一个i2c\_driver结构体

2. 设置

attach\_adapter // 它直接调用 i2c\_probe(adap, 设备地址, 发现这个设备后要调用的函数);

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

3. 注册：i2c\_add\_driver

测试1th:

1. insmod at24cxx.ko

观察输出信息

2. 修改normal\_addr里的0x50为0x60

编译加载，观察输出信息

1. RTC驱动程序

drivers\rtc\rtc-s3c.c

s3c\_rtc\_init

platform\_driver\_register

s3c\_rtc\_probe

rtc\_device\_register("s3c", &pdev->dev, &s3c\_rtcops, THIS\_MODULE)

rtc\_dev\_prepare

cdev\_init(&rtc->char\_dev, &rtc\_dev\_fops);

rtc\_dev\_add\_device

cdev\_add

app: open("/dev/rtc0");

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

kernel: sys\_open

rtc\_dev\_fops.open

rtc\_dev\_open

// 根据次设备号找到以前用"rtc\_device\_register"注册的rtc\_device

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;

s3c\_rtc\_open

app: ioctl(fd, RTC\_RD\_TIME,...)

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

kernel: sys\_ioctl

rtc\_dev\_fops.ioctl

rtc\_dev\_ioctl

struct rtc\_device \*rtc = file->private\_data;

rtc\_read\_time(rtc, &tm);

err = rtc->ops->read\_time(rtc->dev.parent, tm);

s3c\_rtc\_gettime

测试RTC:

1. 修改arch\arm\plat-s3c24xx\common-smdk.c

static struct platform\_device \_\_initdata \*smdk\_devs[] = {

&s3c\_device\_nand,

&smdk\_led4,

&smdk\_led5,

&smdk\_led6,

&smdk\_led7,

改为(在数组smdk\_devs里加上s3c\_device\_rtc):

static struct platform\_device \_\_initdata \*smdk\_devs[] = {

&s3c\_device\_nand,

&smdk\_led4,

&smdk\_led5,

&smdk\_led6,

&smdk\_led7,

&s3c\_device\_rtc,

2. make uImage, 使用新内核启动

3. ls /dev/rtc\* -l

date /\* 显示系统时间 \*/

date 123015402011.30 /\* 设置系统时间 date [MMDDhhmm[[CC]YY][.ss]] \*/

hwclock -w /\* 把系统时间写入RTC \*/

短电,重启,执行date

1. 声卡驱动程序

sound\soc\s3c24xx\s3c2410-uda1341.c

s3c2410\_uda1341\_init

driver\_register(&s3c2410iis\_driver);

.....

s3c2410iis\_probe

/\* 使能时钟 \*/

/\* 配置GPIO \*/

/\* 设置S3C2440的IIS控制器 \*/

init\_s3c2410\_iis\_bus

/\* 使用L3接口初始化uda1341芯片 \*/

init\_uda1341();

/\* 设置两个DMA通道:一个用于播放,另一个用于录音 \*/

.....

register\_sound\_dsp(&smdk2410\_audio\_fops, -1);

sound\_insert\_unit(&chains[3], fops, dev, 3, 131, "dsp", S\_IWUSR | S\_IRUSR, NULL); // /dev/dsp

register\_sound\_mixer(&smdk2410\_mixer\_fops, -1);

sound\_insert\_unit(&chains[0], fops, dev, 0, 128, "mixer", S\_IRUSR | S\_IWUSR, NULL); // /dev/mixer

/dev/dsp: 用于播放/录音

/dev/mixer: 调整音量

1. 主设备号

2. file\_operations

3. register\_chrdev

app: open () // 假设主设备号为14

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

soundcore\_open

int unit = iminor(inode);

s = \_\_look\_for\_unit(chain, unit);

// 从chains数组里得到, 谁来设置这个数组?

new\_fops = fops\_get(s->unit\_fops);

file->f\_op = new\_fops;

err = file->f\_op->open(inode,file);

录音:

app: read

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

file->f\_op->read

播放:

app: write

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

file->f\_op->write

测试:

1. 确定内核里已经配置了sound\soc\s3c24xx\s3c2410-uda1341.c

-> Device Drivers

-> Sound

-> Advanced Linux Sound Architecture

-> Advanced Linux Sound Architecture

-> System on Chip audio support

<\*> I2S of the Samsung S3C24XX chips

2. make uImage

使用新内核启动

3. ls -l /dev/dsp /dev/mixer

4. 播放:

在WINDOWS PC里找一个wav文件,放到开发板根文件系统里

cat Windows.wav > /dev/dsp

5. 录音:

cat /dev/dsp > sound.bin

然后对着麦克风说话

ctrl+c退出

cat sound.bin > /dev/dsp // 就可以听到录下的声音

怎么写WM8976驱动程序?

1. IIS部分一样，保持不变

2. 控制部分不同，重写

测试WM8976:

1. 确定内核里已经配置了sound\soc\s3c24xx\s3c2410-uda1341.c

-> Device Drivers

-> Sound

-> Advanced Linux Sound Architecture // 兼容OSS

-> Advanced Linux Sound Architecture

-> System on Chip audio support

<\*> I2S of the Samsung S3C24XX chips

2. 修改sound/soc/s3c24xx/Makefile

obj-y += s3c2410-uda1341.o

改为:

obj-y += s3c-wm8976.o

3. make uImage

使用新内核启动

4. ls -l /dev/dsp /dev/mixer

5. 播放:

在WINDOWS PC里找一个wav文件,放到开发板根文件系统里

cat Windows.wav > /dev/dsp

6. 录音:

cat /dev/dsp > sound.bin

然后对着麦克风说话

ctrl+c退出

cat sound.bin > /dev/dsp // 就可以听到录下的声音

使用madplay测试声卡：

1. 解压:

tar xzf libid3tag-0.15.1b.tar.gz // 库

tar xzf libmad-0.15.1b.tar.gz // 库

tar xzf madplay-0.15.2b.tar.gz // APP

2. 编译 libid3tag-0.15.1b

mkdir tmp

cd libid3tag-0.15.1b

./configure --host=arm-linux --prefix=/work/drivers\_and\_test/21th\_sound/app/tmp

make

make install

3. 编译 libmad-0.15.1b

cd libmad-0.15.1b

./configure --host=arm-linux --prefix=/work/drivers\_and\_test/21th\_sound/app/tmp

make

make install

4. 编译madplay

cd madplay-0.15.2b/

./configure --host=arm-linux --prefix=/work/drivers\_and\_test/21th\_sound/app/tmp LDFLAGS="-L/work/drivers\_and\_test/21th\_sound/app/tmp/lib" CFLAGS="-I /work/drivers\_and\_test/21th\_sound/app/tmp/include"

make

make install

5. 把tmp/bin/\* tmp/lib/\*so\* 复制到根文件系统：

6. 把一个mp3文件复制到根文件系统

7. madplay --tty-control /1.mp3

播放过程中不断按小键盘的减号("-")会降低音量

不断按小键盘的加号("+")会降低音量

1. Uevent

class\_device\_create

class\_device\_register

class\_device\_add

kobject\_uevent(&class\_dev->kobj, KOBJ\_ADD);

kobject\_uevent\_env(kobj, action, NULL);

// action\_string = "add";

action\_string = action\_to\_string(action);

/\* 分配保存环境变量的内存 \*/

/\* environment values \*/

buffer = kmalloc(BUFFER\_SIZE, GFP\_KERNEL);

/\* 设置环境变量 \*/

envp [i++] = scratch;

scratch += sprintf(scratch, "ACTION=%s", action\_string) + 1;

envp [i++] = scratch;

scratch += sprintf (scratch, "DEVPATH=%s", devpath) + 1;

envp [i++] = scratch;

scratch += sprintf(scratch, "SUBSYSTEM=%s", subsystem) + 1;

/\* 调用应用程序: 比如mdev \*/

/\* 启动脚本 echo /sbin/mdev > /proc/sys/kernel/hotplug

\* 设置了uevent\_helper为“/sbin/mdev“

\*/

argv [0] = uevent\_helper; // = "/sbin/mdev"

argv [1] = (char \*)subsystem;

argv [2] = NULL;

call\_usermodehelper (argv[0], argv, envp, 0);

分析: busybox mdev.c

100ask: uevent\_helper = /sbin/mdev

envp[0] = HOME=/

envp[1] = PATH=/sbin:/bin:/usr/sbin:/usr/bin

envp[2] = ACTION=add

envp[3] = DEVPATH=/class/sixth\_drv/buttons

envp[4] = SUBSYSTEM=sixth\_drv

envp[5] = SEQNUM=720

envp[6] = MAJOR=252

envp[7] = MINOR=0

mdev\_main

temp = /sys/class/sixth\_drv/buttons

make\_device(temp, 0);

/\* 确定设备文件名,类型,主次设备号 \*/

device\_name = bb\_basename(path); = "buttons"

'c' == > 字符设备节点

根据"/sys/class/sixth\_drv/buttons/dev"的内容确定主次设备号

mknod(device\_name, mode | type, makedev(major, minor)

我接上U盘，想自动挂载，怎么办？

mdev.conf的格式：

<device regex> <uid>:<gid> <octal permissions> [<@|$|\*> <command>]

device regex：正则表达式，表示哪一个设备

uid: owner

gid: 组ID

octal permissions：以八进制表示的属性

@：创建设备节点之后执行命令

$：删除设备节点之前执行命令

\*: 创建设备节点之后 和 删除设备节点之前 执行命令

command：要执行的命令

写mdev.conf

1.

leds 0:0 777

led1 0:0 777

led2 0:0 777

led3 0:0 777

2.

leds?[123]? 0:0 777

3.

leds?[123]? 0:0 777 @ echo create /dev/$MDEV > /dev/console

4.

leds?[123]? 0:0 777 \* if [ $ACTION = "add" ]; then echo create /dev/$MDEV > /dev/console; else echo remove /dev/$MDEV > /dev/console; fi

5.

leds?[123]? 0:0 777 \* /bin/add\_remove\_led.sh

把命令写入一个脚本：

add\_remove\_led.sh

#!/bin/sh

if [ $ACTION = "add" ];

then

echo create /dev/$MDEV > /dev/console;

else

echo remove /dev/$MDEV > /dev/console;

fi

6. U盘自动加载

sda[1-9]+ 0:0 777 \* if [ $ACTION = "add" ]; then mount /dev/$MDEV /mnt; else umount /mnt; fi

7.

sda[1-9]+ 0:0 777 \* /bin/add\_remove\_udisk.sh

add\_remove\_udisk.sh

#!/bin/sh

if [ $ACTION = "add" ];

then

mount /dev/$MDEV /mnt;

else

umount /mnt;

fi

1. 裸板调试

1. 点灯

2. 串口打印

3. JTAG调试器

3.1 命令行调试

3.2 源码级别的调试

前提

a. 程序必须已经重定位好，位于它的链接地址

a.1 如果程序的链接地址是SDRAM, 使用openocd初始化SDRAM

a.2 使用arm-linux-gdb/arm-elf-gdb下载程序

b. 链接脚本必须把text,rodata,data,bss等分开存放

c. 被调试的程序为ELF格式，内含调试信息(即编译时有-g选项)

1. 驱动调试

驱动程序的调试

一. 打印: prink, 自制proc文件

UBOOT传入console=ttySAC0 console=tty1

1. 内核处理UBOOT传入的参数

console\_setup

add\_preferred\_console // 我想用名为"ttySAC0"的控制台，先记录下来

2. 硬件驱动的入口函数里:

drivers/serial/s3c2410.c

register\_console(&s3c24xx\_serial\_console);

3. printk

vprintk

/\* Emit the output into the temporary buffer \*/

// 先把输出信息放入临时BUFFER

vscnprintf

// Copy the output into log\_buf.

// 把临时BUFFER里的数据稍作处理，再写入log\_buf

// 比如printk("abc")会得到"<4>abc", 再写入log\_buf

// 可以用dmesg命令把log\_buf里的数据打印出来重现内核的输出信息

// 调用硬件的write函数输出

release\_console\_sem();

call\_console\_drivers(\_con\_start, \_log\_end);

// 从log\_buf得到数据，算出打印级别

\_call\_console\_drivers(start\_print, cur\_index, msg\_level);

// 如果可以级别够格打印

if ((msg\_log\_level < console\_loglevel

\_\_call\_console\_drivers

con->write(con, &LOG\_BUF(start), end - start);

二. 根据内核打印的段错误信息分析

a. 作为模块：

1. 根据pc值确定该指令属于内核还是外加的模块

pc=0xbf000018 它属于什么的地址？是内核还是通过insmod加载的驱动程序？

先判断是否属于内核的地址: 看System.map确定内核的函数的地址范围:c0004000~c03265a4

如果不属于System.map里的范围，则它属于insmod加载的驱动程序

2. 假设它是加载的驱动程序引入的错误，怎么确定是哪一个驱动程序？

先看看加载的驱动程序的函数的地址范围

cat /proc/kallsyms (内核函数、加载的函数的地址)

从这些信息里找到一个相近的地址, 这个地址<=0xbf000018

比如找到了：

bf000000 t first\_drv\_open [first\_drv]

3. 找到了first\_drv.ko

在PC上反汇编它: arm-linux-objdump -D first\_drv.ko > frist\_drv.dis

在dis文件里找到first\_drv\_open

first\_drv.dis文件里 insmod后

00000000 <first\_drv\_open>: bf000000 t first\_drv\_open [first\_drv]

00000018 pc = bf000018

./firstdrvtest on

Unable to handle kernel paging request at virtual address 56000050

内核使用56000050来访问时发生了错误

pgd = c3eb0000

[56000050] \*pgd=00000000

Internal error: Oops: 5 [#1]

Modules linked in: first\_drv

CPU: 0 Not tainted (2.6.22.6 #1)

PC is at first\_drv\_open+0x18(该指令的偏移)/0x3c(该函数的总大小) [first\_drv]

PC就是发生错误的指令的地址

大多时候，PC值只会给出一个地址，不到指示说是在哪个函数里

LR is at chrdev\_open+0x14c/0x164

LR寄存器的值

pc = 0xbf000018

pc : [<bf000018>] lr : [<c008d888>] psr: a0000013

sp : c3c7be88 ip : c3c7be98 fp : c3c7be94

r10: 00000000 r9 : c3c7a000 r8 : c049abc0

r7 : 00000000 r6 : 00000000 r5 : c3e740c0 r4 : c06d41e0

r3 : bf000000 r2 : 56000050 r1 : bf000964 r0 : 00000000

执行这条导致错误的指令时各个寄存器的值

Flags: NzCv IRQs on FIQs on Mode SVC\_32 Segment user

Control: c000717f Table: 33eb0000 DAC: 00000015

Process firstdrvtest (pid: 777, stack limit = 0xc3c7a258)

发生错误时当前进程的名称是firstdrvtest

栈

Stack: (0xc3c7be88 to 0xc3c7c000)

be80: c3c7bebc c3c7be98 c008d888 bf000010 00000000 c049abc0

bea0: c3e740c0 c008d73c c0474e20 c3e766a8 c3c7bee4 c3c7bec0 c0089e48 c008d74c

bec0: c049abc0 c3c7bf04 00000003 ffffff9c c002c044 c3d10000 c3c7befc c3c7bee8

bee0: c0089f64 c0089d58 00000000 00000002 c3c7bf68 c3c7bf00 c0089fb8 c0089f40

bf00: c3c7bf04 c3e766a8 c0474e20 00000000 00000000 c3eb1000 00000101 00000001

bf20: 00000000 c3c7a000 c04a7468 c04a7460 ffffffe8 c3d10000 c3c7bf68 c3c7bf48

bf40: c008a16c c009fc70 00000003 00000000 c049abc0 00000002 bec1fee0 c3c7bf94

bf60: c3c7bf6c c008a2f4 c0089f88 00008520 bec1fed4 0000860c 00008670 00000005

bf80: c002c044 4013365c c3c7bfa4 c3c7bf98 c008a3a8 c008a2b0 00000000 c3c7bfa8

bfa0: c002bea0 c008a394 bec1fed4 0000860c 00008720 00000002 bec1fee0 00000001

bfc0: bec1fed4 0000860c 00008670 00000002 00008520 00000000 4013365c bec1fea8

bfe0: 00000000 bec1fe84 0000266c 400c98e0 60000010 00008720 00000000 00000000

Backtrace: (回溯)

[<bf000000>] (first\_drv\_open+0x0/0x3c [first\_drv]) from [<c008d888>] (chrdev\_open+0x14c/0x164)

[<c008d73c>] (chrdev\_open+0x0/0x164) from [<c0089e48>] (\_\_dentry\_open+0x100/0x1e8)

r8:c3e766a8 r7:c0474e20 r6:c008d73c r5:c3e740c0 r4:c049abc0

[<c0089d48>] (\_\_dentry\_open+0x0/0x1e8) from [<c0089f64>] (nameidata\_to\_filp+0x34/0x48)

[<c0089f30>] (nameidata\_to\_filp+0x0/0x48) from [<c0089fb8>] (do\_filp\_open+0x40/0x48)

r4:00000002

[<c0089f78>] (do\_filp\_open+0x0/0x48) from [<c008a2f4>] (do\_sys\_open+0x54/0xe4)

r5:bec1fee0 r4:00000002

[<c008a2a0>] (do\_sys\_open+0x0/0xe4) from [<c008a3a8>] (sys\_open+0x24/0x28)

[<c008a384>] (sys\_open+0x0/0x28) from [<c002bea0>] (ret\_fast\_syscall+0x0/0x2c)

Code: e24cb004 e59f1024 e3a00000 e5912000 (e5923000)

Segmentation fault

#

b. 编入内核

Modules linked in:

CPU: 0 Not tainted (2.6.22.6 #2)

PC is at first\_drv\_open+0x18/0x3c

LR is at chrdev\_open+0x14c/0x164

pc : [<c014e6c0>] lr : [<c008638c>] psr: a0000013

sp : c3a03e88 ip : c3a03e98 fp : c3a03e94

r10: 00000000 r9 : c3a02000 r8 : c03f3c60

r7 : 00000000 r6 : 00000000 r5 : c38a0c50 r4 : c3c1e780

r3 : c014e6a8 r2 : 56000050 r1 : c031a47c r0 : 00000000

Flags: NzCv IRQs on FIQs on Mode SVC\_32 Segment user

Control: c000717f Table: 339f0000 DAC: 00000015

Process firstdrvtest (pid: 750, stack limit = 0xc3a02258)

1. 根据pc值确定该指令属于内核还是外加的模块

pc=c014e6c0 属于内核(看System.map)

2. 反汇编内核: arm-linux-objdump -D vmlinux > vmlinux.dis

在dis文件里搜c014e6c0

c014e6a8 <first\_drv\_open>:

c014e6a8: e1a0c00d mov ip, sp

c014e6ac: e92dd800 stmdb sp!, {fp, ip, lr, pc}

c014e6b0: e24cb004 sub fp, ip, #4 ; 0x4

c014e6b4: e59f1024 ldr r1, [pc, #36] ; c014e6e0 <.text+0x1276e0>

c014e6b8: e3a00000 mov r0, #0 ; 0x0

c014e6bc: e5912000 ldr r2, [r1]

c014e6c0: e5923000 ldr r3, [r2] // 在此出错 r2=56000050

3. 根据栈信息分析函数调用过程

# ./firstdrvtest on

Unable to handle kernel paging request at virtual address 56000050

pgd = c3e78000

[56000050] \*pgd=00000000

Internal error: Oops: 5 [#1]

Modules linked in: first\_drv

CPU: 0 Not tainted (2.6.22.6 #48)

PC is at first\_drv\_open+0x18/0x3c [first\_drv]

LR is at chrdev\_open+0x14c/0x164

pc : [<bf000018>] lr : [<c008c888>] psr: a0000013

3.1 根据PC确定出错位置

bf000018 属于 insmod的模块

bf000000 t first\_drv\_open [first\_drv]

3.2 确定它属于哪个函数

反汇编first\_drv.ko

sp : c3e69e88 ip : c3e69e98 fp : c3e69e94

r10: 00000000 r9 : c3e68000 r8 : c0490620

r7 : 00000000 r6 : 00000000 r5 : c3e320a0 r4 : c06a8300

r3 : bf000000 r2 : 56000050 r1 : bf000964 r0 : 00000000

Flags: NzCv IRQs on FIQs on Mode SVC\_32 Segment user

Control: c000717f Table: 33e78000 DAC: 00000015

Process firstdrvtest (pid: 752, stack limit = 0xc3e68258)

Stack: (0xc3e69e88 to 0xc3e6a000)

9e80: c3e69ebc c3e69e98 c008c888 bf000010 00000000 c0490620

first\_drv\_open'sp lr chrdev\_open'sp

9ea0: c3e320a0 c008c73c c0465e20 c3e36cb4 c3e69ee4 c3e69ec0 c0088e48 c008c74c

lr

9ec0: c0490620 c3e69f04 00000003 ffffff9c c002b044 c06e0000 c3e69efc c3e69ee8

\_\_dentry\_open'sp

9ee0: c0088f64 c0088d58 00000000 00000002 c3e69f68 c3e69f00 c0088fb8 c0088f40

lr nameidata\_to\_filp'sp lr

9f00: c3e69f04 c3e36cb4 c0465e20 00000000 00000000 c3e79000 00000101 00000001

do\_filp\_open'sp

9f20: 00000000 c3e68000 c04c1468 c04c1460 ffffffe8 c06e0000 c3e69f68 c3e69f48

9f40: c008916c c009ec70 00000003 00000000 c0490620 00000002 be94eee0 c3e69f94

9f60: c3e69f6c c00892f4 c0088f88 00008520 be94eed4 0000860c 00008670 00000005

lr do\_sys\_open'sp

9f80: c002b044 4013365c c3e69fa4 c3e69f98 c00893a8 c00892b0 00000000 c3e69fa8

lr sys\_open'sp

9fa0: c002aea0 c0089394 be94eed4 0000860c 00008720 00000002 be94eee0 00000001

lr ret\_fast\_syscall'sp

9fc0: be94eed4 0000860c 00008670 00000002 00008520 00000000 4013365c be94eea8

9fe0: 00000000 be94ee84 0000266c 400c98e0 60000010 00008720 00000000 00000000

三. 自制工具

寄存器编辑器

四. 修改内核来定位系统僵死问题

./firstdrvtest on

asm\_do\_IRQ => s3c2410\_timer\_interrupt : pid = 752, task name = firstdrvtest

pc = bf000084

asm\_do\_IRQ => s3c2410\_timer\_interrupt : pid = 752, task name = firstdrvtest

pc = bf000084 // 对于中断, pc-4才是发生中断瞬间的地址

看/proc/kallsyms

first\_drv.dis

00000000 <first\_drv\_open>: bf000000 t first\_drv\_open [first\_drv]

0000003c <first\_drv\_write>:

3c: e1a0c00d mov ip, sp

40: e92dd800 stmdb sp!, {fp, ip, lr, pc}

44: e24cb004 sub fp, ip, #4 ; 0x4

48: e24dd004 sub sp, sp, #4 ; 0x4

4c: e3cd3d7f bic r3, sp, #8128 ; 0x1fc0

50: e3c3303f bic r3, r3, #63 ; 0x3f

54: e5933008 ldr r3, [r3, #8]

58: e0910002 adds r0, r1, r2

5c: 30d00003 sbcccs r0, r0, r3

60: 33a03000 movcc r3, #0 ; 0x0

64: e3530000 cmp r3, #0 ; 0x0

68: e24b0010 sub r0, fp, #16 ; 0x10

6c: 1a00001c bne e4 <init\_module+0x5c>

70: ebfffffe bl 70 <first\_drv\_write+0x34>

74: ea00001f b f8 <init\_module+0x70>

78: e3520000 cmp r2, #0 ; 0x0

7c: 11a01002 movne r1, r2

80: 1bfffffe blne 80 <first\_drv\_write+0x44> // 卡死的地方

84: ea00001f b 108 <init\_module+0x80>

1. 应用调试

一、应用调试1：使用strace命令来跟踪系统调用

二、应用调试2：使用GDB来调试应用程序

编译gdb,gdbserver

tar xjf gdb-7.4.tar.bz2

cd gdb-7.4/

./configure --target=arm-linux

make

把arm-linux-gdb复制到/bin目录

cd gdb/gdbserver/

./configure --host=arm-linux

cp gdbserver /work/nfs\_root/first\_fs/bin

编译要调试的应用，编译时加上-g选项

调试：

1. 在ARM板上

gdbserver 192.168.1.17:2345 ./test\_debug

2. 在PC上

/bin/arm-linux-gdb ./test\_debug

输入：target remote 192.168.1.17:2345

然后: 使用gdb命令来控制程序

另一种方法：

让程序在开发板上直接运行，当它发生错误时，令它产生core dump文件

然后使用gdb根据core dump文件找到发生错误的地方

在ARM板上：

1. ulimit -c unlimited

2. 执行应用程序 : 程序出错时会在当前目录下生成名为core的文件

在PC上：

3. /bin/arm-linux-gdb ./test\_debug ./core

三、应用调试3：配置内核输出应用程序的段错误信息

arch/arm/mm/fault.c

\_\_do\_user\_fault(struct task\_struct \*tsk, unsigned long addr,

unsigned int fsr, unsigned int sig, int code,

struct pt\_regs \*regs)

{

struct siginfo si;

#ifdef CONFIG\_DEBUG\_USER // 1. 配置内核

if (user\_debug & UDBG\_SEGV) {

printk(KERN\_DEBUG "%s: unhandled page fault (%d) at 0x%08lx, code 0x%03x\n",

tsk->comm, sig, addr, fsr);

show\_pte(tsk->mm, addr);

show\_regs(regs);

}

#endif

2. uboot: set bootargs user\_debug=0xff

set bootargs console=ttySAC0 root=/dev/nfs nfsroot=192.168.1.123:/work/nfs\_root/first\_fs ip=192.168.1.17 ipaddr=192.168.1.17 user\_debug=0xff

3. 执行APP

./test\_debug

a = 0x12

pgd = c04c8000

[00000000] \*pgd=33d08031, \*pte=00000000, \*ppte=00000000

Pid: 772, comm: test\_debug

CPU: 0 Not tainted (2.6.22.6 #1)

PC is at 0x84ac

LR is at 0x84d0

pc : [<000084ac>] lr : [<000084d0>] psr: 60000010

sp : bed9fe40 ip : bed9fe54 fp : bed9fe50

r10: 4013365c r9 : 00000000 r8 : 00008514

r7 : 00000001 r6 : 000085cc r5 : 00008568 r4 : bed9fec4

r3 : 00000012 r2 : 00000000 r1 : 00001000 r0 : 00000000

Flags: nZCv IRQs on FIQs on Mode USER\_32 Segment user

Control: c000717f Table: 304c8000 DAC: 00000015

[<c002cd1c>] (show\_regs+0x0/0x4c) from [<c0031a98>] (\_\_do\_user\_fault+0x5c/0xa4)

r4:c04a6840

[<c0031a3c>] (\_\_do\_user\_fault+0x0/0xa4) from [<c0031d38>] (do\_page\_fault+0x1dc/0x20c)

r7:c00261e0 r6:c0024cf8 r5:c04a6840 r4:ffffffec

[<c0031b5c>] (do\_page\_fault+0x0/0x20c) from [<c002b224>] (do\_DataAbort+0x3c/0xa0)

[<c002b1e8>] (do\_DataAbort+0x0/0xa0) from [<c002be48>] (ret\_from\_exception+0x0/0x10)

Exception stack(0xc3e7bfb0 to 0xc3e7bff8)

bfa0: 00000000 00001000 00000000 00000012

bfc0: bed9fec4 00008568 000085cc 00000001 00008514 00000000 4013365c bed9fe50

bfe0: bed9fe54 bed9fe40 000084d0 000084ac 60000010 ffffffff

r8:00008514 r7:00000001 r6:000085cc r5:00008568 r4:c039bfc8

Segmentation fault

Stack:

00000000 becd3e64 becd3e54 000084d0 000084a0 00000000 becd3e78 becd3e68

C's sp return addr B's sp

000084f0 000084c4 00000000 becd3e98 becd3e7c 00008554 000084e4 00000000

ret addr A's sp ret addr main's sp

00000012 becd3ec4 00000001 00000000 becd3e9c 40034f14 00008524 00000000

ret addr caller's sp

对于动态链接，已经退出的程序不好确定动态库的地址

00000000 0000839c 00000000 00000000 4001d594 000083c4 000085cc 4000c02c

becd3ec4 becd3f7b 00000000 becd3f88 becd3f92 becd3f99 becd3fad becd3fb8

becd3fdb becd3fe9 00000000 00000010 00000003 00000006 00001000 00000011

00000064 00000003 00008034 00000004 00000020 00000005 00000006 00000007

40000000 00000008 00000000 00000009 0000839c 0000000b 00000000 0000000c

00000000 0000000d 00000000 0000000e 00000000 00000017 00000000 0000000f

becd3f77 00000000 00000000 00000000 00000000 76000000 2e006c34 7365742f

65645f74 00677562 52455355 6f6f723d 4f480074 2f3d454d 61706900 3d726464

2e323931 2e383631 37312e31 52455400 74763d4d 00323031 48544150 62732f3d

2f3a6e69 2f727375 6e696273 69622f3a 752f3a6e 622f7273 53006e69 4c4c4548

69622f3d 68732f6e 44575000 2e002f3d 7365742f 65645f74 00677562 00000000

4. 反汇编app:

arm-linux-objdump -D test\_debug > test\_debug.dis

对于静态链接的test\_debug

PC is at 0x81e0

LR is at 0x8204

pc : [<000081e0>] lr : [<00008204>] psr: 60000010

sp : be93dc60 ip : be93dc74 fp : be93dc70

r10: 000085f4 r9 : 00008248 r8 : be93deb4

r7 : 00000001 r6 : 00000000 r5 : be93dd3e r4 : 00000000

r3 : 00000012 r2 : 00000000 r1 : 00001000 r0 : 00000000

Flags: nZCv IRQs on FIQs on Mode USER\_32 Segment user

Stack:

00000000 be93dc84 be93dc74 00008204 000081d4 00000000 be93dc98 be93dc88

C'sp ret addr B'sp

00008224 000081f8 00000000 be93dcb8 be93dc9c 00008288 00008218 00000000

ret addr A'sp ret addr main'sp

00000012 be93deb4 00000001 00000000 be93dcbc 000084ac 00008258 756e694c

ret addr \_\_libc\_start\_main'sp

00000078 00000000 00000000 00000000 00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000 00000000 00000000 00000000 32393100

3836312e 312e312e 00000037 00000000 00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000 00000000 00000000 00000000 2e320000

32322e36 0000362e 00000000 00000000 00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000 00000000 00000000 00000000 23000000

54203335 4d206575 31207961 31322035 3a34303a 43203834 32205453 00323130

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

766d7261 006c7434 00000000 00000000 00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

6f6e2800 0029656e 00000000 00000000 00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

00000000 000080f8 00008680 00000000 be93deb4 be93df6a 00000000 be93df77

be93df81 be93df92 be93df99 be93dfad be93dfb8 be93dfdb be93dfe9 00000000

00000010 00000003 00000006 00001000 00000011 00000064 00000003 00008034

00000004 00000020 00000005 00000003 00000007 00000000 00000008 00000000

00000009 000080d0 0000000b 00000000 0000000c 00000000 0000000d 00000000

0000000e 00000000 00000017 00000000 0000000f be93df66 00000000 00000000

00000000 34760000 2f2e006c 74736574 6265645f 55006775 3d524553 746f6f72

444c4f00 3d445750 6f72702f 34372f63 4f480033 2f3d454d 61706900 3d726464

2e323931 2e383631 37312e31 52455400 74763d4d 00323031 48544150 62732f3d

2f3a6e69 2f727375 6e696273 69622f3a 752f3a6e 622f7273 53006e69 4c4c4548

69622f3d 68732f6e 44575000 2e002f3d 7365742f 65645f74 00677562 00000000

四、应用调试4：自制系统调用、编写进程查看器

把29th\_app\_system\_call\kernel里的文件复制到内核目录

syscalls.h ==> include/linux

read\_write.c ==> fs/

calls.S ==> arch/arm/kernel

五、应用调试5：编写输入模拟器

1. 产品要经过测试才能发布，一般都是人工操作，比如手机触摸屏、遥控器

2. 操作过程中发现错误，要再次复现，找到规律，修改程序

3. 能否在驱动程序里把所有的操作记录下来，存为文件

当出错时，可以通过文件里的数据来"复现"输入

input\_event

nfs 32000000 192.168.1.123:/work/nfs\_root/uImage\_nots; bootm 32000000

1. 自己写bootloader

最简单的bootloader的编写步骤：

1. 初始化硬件：关看门狗、设置时钟、设置SDRAM、初始化NAND FLASH

2. 如果bootloader比较大，要把它重定位到SDRAM

3. 把内核从NAND FLASH读到SDRAM

4. 设置"要传给内核的参数"

5. 跳转执行内核

改进：

1. 提高CPU频率, 200MHZ ==> 400MHZ

2. 启动ICACHE

1. 移植最新的uboot

1、下载、建立source insight工程、编译、烧写、如果无运行分析原因

tar xjf u-boot-2012.04.01.tar.bz2

cd u-boot-2012.04.01

make smdk2410\_config

make

2. 分析u-boot: 通过链接命令分析组成文件、阅读代码分析启动过程

a. 初始化硬件：关看门狗、设置时钟、设置SDRAM、初始化NAND FLASH

b. 如果bootloader比较大，要把它重定位到SDRAM

c. 把内核从NAND FLASH读到SDRAM

d. 设置"要传给内核的参数"

e. 跳转执行内核

2.1 set the cpu to SVC32 mode

2.2 turn off the watchdog

2.3 mask all IRQs by setting all bits in the INTMR

2.4 设置时钟比例

2.5 设置内存控制器

2.6 设置栈，调用C函数board\_init\_f

2.7 调用函数数组init\_sequence里的各个函数

2.7.1 board\_early\_init\_f : 设置系统时钟、设置GPIO

......

2.8 重定位代码:

2.8.1 从NOR FLASH把代码复制到SDRAM

2.8.2 程序的链接地址是0，访问全局变量、静态变量、调用函数时是使"基于0地址编译得到的地址"

现在把程序复制到了SDRAM

需要修改代码，把"基于0地址编译得到的地址"改为新地址

2.8.3 程序里有些地址在链接时不能确定，要到运行前才能确定：fixabs

2.9 clear\_bss

2.10 调用C函数board\_init\_r：第2阶段的代码

可以修改配置定义CONFIG\_S3C2440

3. 修改U-BOOT代码

3.1 建一个单板

cd board/samsung/

cp smdk2410 smdk2440 -rf

cd ../../include/configs/

cp smdk2410.h smdk2440.h

修改boards.cfg:

仿照

smdk2410 arm arm920t - samsung s3c24x0

添加：

smdk2440 arm arm920t - samsung s3c24x0

3.2 烧写看结果

3.3 调试：

a. 阅读代码发现不足：UBOOT里先以60MHZ的时钟计算参数来设置内存控制器，但是MPLL还未设置

处理措施: 把MPLL的设置放到start.S里，取消board\_early\_init\_f里对MPLL的设置

编译出来的uboot非常大，可以先烧写主光盘里的u-boot.bin到nor，然后用这个uboot来烧写新的uboot

3.4 乱码，查看串口波特率的设置，发现在get\_HCLK里没有定义CONFIG\_S3C2440

处理措施：include/configs/smdk2440.h: 去掉CONFIG\_S3C2410

#define CONFIG\_S3C2440

//#define CONFIG\_CMD\_NAND

3.5 修改UBOOT支持NAND启动

原来的代码在链接时加了"-pie"选项, 使得u-boot.bin里多了"\*(.rel\*)", "\*(.dynsym)"

使得程序非常大，不利于从NAND启动(重定位之前的启动代码应该少于4K)

3.5.1 去掉 "-pie"选项

arch/arm/config.mk:75:LDFLAGS\_u-boot += -pie 去掉这行

3.5.2 参考"毕业班第1课"的start.S, init.c来修改代码

把init.c放入board/samsung/smdk2440目录, 修改Makefile

修改CONFIG\_SYS\_TEXT\_BASE为0x33f80000

修改start.S

3.5.3 修改board\_init\_f, 把relocate\_code去掉

3.5.4 修改链接脚本: 把start.S, init.c, lowlevel.S等文件放在最前面

./arch/arm/cpu/u-boot.lds：

board/samsung/smdk2440/libsmdk2440.o

3.6 修改UBOOT支持NOR FLASH

drivers\mtd\jedec\_flash.c 加上新的型号

#define CONFIG\_SYS\_MAX\_FLASH\_SECT (128)

修复了重定时留下来的BUG：SP要重新设置

3.7 修改UBOOT支持NAND FLASH

修改：include/configs/smdk2440.h: #define CONFIG\_CMD\_NAND

把drivers\mtd\nand\s3c2410\_nand.c复制为s3c2440\_nand.c

分析过程：

nand\_init

nand\_init\_chip

board\_nand\_init

设置nand\_chip结构体, 提供底层的操作函数

nand\_scan

nand\_scan\_ident

nand\_set\_defaults

chip->select\_chip = nand\_select\_chip;

chip->cmdfunc = nand\_command;

chip->read\_byte = busw ? nand\_read\_byte16 : nand\_read\_byte;

nand\_get\_flash\_type

chip->select\_chip

chip->cmdfunc(mtd, NAND\_CMD\_RESET, -1, -1);

nand\_command() // 即可以用来发命令，也可以用来发列地址(页内地址)、行地址(哪一页)

chip->cmd\_ctrl

s3c2440\_hwcontrol

chip->cmdfunc(mtd, NAND\_CMD\_READID, 0x00, -1);

\*maf\_id = chip->read\_byte(mtd);

\*dev\_id = chip->read\_byte(mtd);

3.8 修改UBOOT支持DM9000网卡

eth\_initialize

board\_eth\_init

cs8900\_initialize

\*\*\* ERROR: `ethaddr' not set

set ipaddr 192.168.1.17

set ethaddr 00:0c:29:4d:e4:f4

set serverip 192.168.1.3

4. 易用性修裁剪及制作补丁

内核打印出来的分区信息

0x00000000-0x00040000 : "bootloader"

0x00040000-0x00060000 : "params"

0x00060000-0x00260000 : "kernel"

0x00260000-0x10000000 : "root"

nand erase 60000 200000

nand write 30000000 60000 200000

tftp 30000000 uImage

nand erase.part kernel

nand write 30000000 kernel

烧写JFFS2

tftp 30000000 fs\_mini\_mdev.jffs2

nand erase.part rootfs

nand write.jffs2 30000000 0x00260000 5b89a8

set bootargs console=ttySAC0,115200 root=/dev/mtdblock3 rootfstype=jffs2

烧写YAFFS

tftp 30000000 fs\_mini\_mdev.yaffs2

nand erase.part rootfs

nand write.yaffs 30000000 260000 889bc0

更新UBOOT:

tftp 30000000 u-boot\_new.bin; protect off all; erase 0 3ffff; cp.b 30000000 0 40000

制作补丁：

diff -urN u-boot-2012.04.01 u-boot-2012.04.01\_100ask > u-boot-2012.04.01\_100ask.patch

分析"重定位之修改代码为新地址":

#ifndef CONFIG\_SPL\_BUILD

/\*

\* fix .rel.dyn relocations

\*/

ldr r0, \_TEXT\_BASE /\* r0 <- Text base \*/

// r0=0, 代码基地址

sub r9, r6, r0 /\* r9 <- relocation offset \*/

// r9 = r6-r0 = 0x33f41000 - 0 = 0x33f41000

ldr r10, \_dynsym\_start\_ofs /\* r10 <- sym table ofs \*/

// r10 = 00073608

add r10, r10, r0 /\* r10 <- sym table in FLASH \*/

// r10 = 00073608 + 0 = 00073608

ldr r2, \_rel\_dyn\_start\_ofs /\* r2 <- rel dyn start ofs \*/

// r2=0006b568

add r2, r2, r0 /\* r2 <- rel dyn start in FLASH \*/

// r2=r2+r0=0006b568

ldr r3, \_rel\_dyn\_end\_ofs /\* r3 <- rel dyn end ofs \*/

// r3=00073608

add r3, r3, r0 /\* r3 <- rel dyn end in FLASH \*/

// r3=r3+r0=00073608

fixloop:

ldr r0, [r2] /\* r0 <- location to fix up, IN FLASH! \*/

1. r0=[0006b568]=00000020

add r0, r0, r9 /\* r0 <- location to fix up in RAM \*/

1. r0=r0+r9=00000020 + 0x33f41000 = 0x33f41020

ldr r1, [r2, #4]

1. r1=[0006b568+4]=00000017

and r7, r1, #0xff

1. r7=r1&0xff=00000017

cmp r7, #23 /\* relative fixup? \*/

1. r7 == 23(0x17)

beq fixrel

cmp r7, #2 /\* absolute fixup? \*/

beq fixabs

/\* ignore unknown type of fixup \*/

b fixnext

fixabs:

/\* absolute fix: set location to (offset) symbol value \*/

mov r1, r1, LSR #4 /\* r1 <- symbol index in .dynsym \*/

add r1, r10, r1 /\* r1 <- address of symbol in table \*/

ldr r1, [r1, #4] /\* r1 <- symbol value \*/

add r1, r1, r9 /\* r1 <- relocated sym addr \*/

b fixnext

fixrel:

/\* relative fix: increase location by offset \*/

ldr r1, [r0]

1. r1=[00000020]=000001e0

add r1, r1, r9

1. r1=r1+r9=000001e0 + 0x33f41000 = 33F411E0

fixnext:

str r1, [r0]

1. [0x33f41020] = 33F411E0

add r2, r2, #8 /\* each rel.dyn entry is 8 bytes \*/

1. r2=r2+8=0006b568+8=6B570

cmp r2, r3

1.

blo fixloop

#endif

1. 移植最新版本3.4.2内核

一. 内核启动流程，据此配置内核(机器ID)

1.1 修改Makefile

1.2 选择默认配置 : make s3c2410\_defconfig

1.3 make uImage

步骤1：

在UBOOT里：

set machid 16a // smdk2440 mach-smdk2440.c

或

set machid 7CF // mini2440 mach-mini2440.c

步骤2：

arch\arm\mach-s3c24xx\mach-smdk2440.c

s3c24xx\_init\_clocks(16934400);

改为

s3c24xx\_init\_clocks(12000000);

步骤3：

配置/编译: make s3c2410\_defconfig 或 make mini2440\_defconfig

make uImage

步骤4：

在uboot里：set bootargs console=ttySAC0,115200 .....

uboot的默认MACH ID：

gd->bd->bi\_arch\_number = MACH\_TYPE\_SMDK2410; // 193

如果s = getenv("machid");成功，则使用它

否则使用默认的

set machid 16a // smdk2440 mach-smdk2440.c

set machid 7CF // mini2440 mach-mini2440.c

nfs 32000000 192.168.1.123:/work/nfs\_root/uImage\_new

bootm 32000000

二. 修改分区, 制作新的文件系统

nfs 30000000 192.168.1.123:/work/nfs\_root/fs\_mini\_mdev.yaffs2

nand erase.part rootfs

nand write.yaffs 30000000 260000 889bc0

nfs 30000000 192.168.1.123:/work/nfs\_root/fs\_mini\_mdev.jffs2

nand erase.part rootfs

nand write.jffs2 30000000 260000 $filesize

set bootargs console=ttySAC0,115200 root=/dev/mtdblock3 rootfstype=jffs2

制作文件系统

1. 交叉编译busybox

安装：make install CONFIG\_PREFIX=/work/nfs\_root/fs\_mini\_mdev\_new

2. 安装库

mkdir /work/nfs\_root/fs\_mini\_mdev\_new/lib

mkdir /work/nfs\_root/fs\_mini\_mdev\_new/usr/lib -p

cp /usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc/armv4t/lib/\*so\* /work/nfs\_root/fs\_mini\_mdev\_new/lib -d

cp /usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc/armv4t/usr/lib/\*so\* /work/nfs\_root/fs\_mini\_mdev\_new/usr/lib -d

3. 构建etc目录

4. 构建dev目录

5. 其他空目录

mkfs.jffs2 -n -s 2048 -e 128KiB -d fs\_mini\_mdev\_new -o fs\_mini\_mdev\_new.jffs2

烧写：

nfs 30000000 192.168.1.123:/work/nfs\_root/fs\_mini\_mdev\_new.jffs2

nand erase.part rootfs

nand write.jffs2 30000000 260000 $filesize

set bootargs console=ttySAC0,115200 root=/dev/mtdblock3 rootfstype=jffs2

nfs 32000000 192.168.1.123:/work/nfs\_root/uImage\_new

bootm 32000000

SIGILL illegal instruction

6. 重新配置内核支持EABI

set bootargs console=ttySAC0,115200 root=/dev/mtdblock3 rootfstype=jffs2

nfs 32000000 192.168.1.123:/work/nfs\_root/uImage\_eabi

bootm 32000000

三. 移植YAFFS文件系统

1. 获得源码

git clone git://www.aleph1.co.uk/yaffs2

2. 打补丁

cd yaffs-dir

./patch-ker.sh c m linux-tree 比如 ./patch-ker.sh c m /work/system/linux-3.4.2

3. 配置内核支持YAFFS

4. 编译、使用uImage

5. 制作、烧写yaffs映象

mkyaffs2image fs\_mini\_mdev\_new fs\_mini\_mdev\_new.yaffs2

uboot:

nfs 30000000 192.168.1.123:/work/nfs\_root/fs\_mini\_mdev\_new.yaffs2

nand erase.part rootfs

nand write.yaffs 30000000 260000 $filesize

6. 启动

set bootargs console=ttySAC0,115200 root=/dev/mtdblock3

nfs 32000000 192.168.1.123:/work/nfs\_root/uImage\_yaffs

bootm 32000000

7. 没成功，用替代法查找问题

7.1 UBOOT可能有问题：换上1.1.6的UBOOT

tftp 30000000 u-boot.bin

nand erase.part u-boot

nand write 30000000 u-boot

reset

nfs 30000000 192.168.1.123:/work/nfs\_root/fs\_mini\_mdev\_new.yaffs2

nand erase rootfs

nand write.yaffs 30000000 260000 $(filesize)

set bootargs console=ttySAC0,115200 root=/dev/mtdblock3

nfs 32000000 192.168.1.123:/work/nfs\_root/uImage\_yaffs

bootm 32000000

发现1.1.6的UBOOT没问题，所以就是我们移植的新UBOOT有BUG：

drivers\mtd\nand\Nand\_util.c

if (!need\_skip && !(flags & WITH\_DROP\_FFS)) {

改为

if (!need\_skip && !(flags & WITH\_DROP\_FFS) && !(flags & WITH\_YAFFS\_OOB)) {

7.2 YAFFS映象可能有问题

制作了u-boot\_new.bin, uImage\_new, fs\_mini\_mdev\_new.yaffs

重烧整个系统：

使用jtag工具烧u-boot\_new.bin

或使用uboot来更新自己: tftp 30000000 u-boot\_new.bin; nand erase.part u-boot; nand write 30000000 u-boot

启动uboot，用它来烧写内核、FS

tftp 30000000 uImage\_new; nand erase.part kernel; nand write 30000000 kernel

tftp 30000000 fs\_mini\_mdev\_new.yaffs2; nand erase.part rootfs; nand write.yaffs 30000000 260000 $filesize

设置参数

set 'nand read 30000000 kernel;bootm 30000000'

set bootargs console=ttySAC0,115200 root=/dev/mtdblock3

set machid 16a

save

使用内核补丁：

patch -p1 < ../linux-3.4.2\_100ask.patch

cp config\_ok .config

make uImage

1. Tslib编译使用方法

sudo apt-get install autoconf

sudo apt-get install automake

sudo apt-get install libtool

编译：

tar xzf tslib-1.4.tar.gz

cd tslib

./autogen.sh

mkdir tmp

echo "ac\_cv\_func\_malloc\_0\_nonnull=yes" >arm-linux.cache

./configure --host=arm-linux --cache-file=arm-linux.cache --prefix=$(pwd)/tmp

make

make install

安装：

cd tmp

cp \* -rf /nfsroot

使用：

先安装s3c\_ts.ko, lcd.ko

1．

修改 /etc/ts.conf第1行(去掉#号和第一个空格)：

# module\_raw input

改为：

module\_raw input

2．

export TSLIB\_TSDEVICE=/dev/event1

export TSLIB\_CALIBFILE=/etc/pointercal

export TSLIB\_CONFFILE=/etc/ts.conf

export TSLIB\_PLUGINDIR=/lib/ts

export TSLIB\_CONSOLEDEVICE=none

export TSLIB\_FBDEVICE=/dev/fb0

ts\_calibrate

ts\_test

cat /proc/mymsg

replay\_r, replay\_w

0x00075cf7 0x00000003 0x00000018 0

0x00076617 0x00000003 0x00000018 0

0x00076617 0x00000003 0x00000018 0

0x00076617 0x00000003 0x00000018 0

0x00076617 0x00000003 0x00000018 0

0x00076617 0x00000003 0x00000018 0

0x0007661b 0x00000003 0x00000018 0

0x0007661b 0x00000003 0x00000018 0

0x0007661b 0x00000003 0x00000018 0

0x0007661b 0x00000003 0x00000018 0

0x0007661b 0x00000003 0x00000018 0

1. 移植所有驱动到3.4.2内核去

移植：

1. 编译

2. 解决错误

2.1 头文件不对：去掉或改名

2.2 宏不对：改名使用新宏

2.3 有些函数没有了：改名使用新函数

mount -t nfs -o nolock,vers=2 192.168.1.123:/work/nfs\_root/fs\_mini\_mdev\_new /mnt

nfs 30000000 192.168.1.123:/work/nfs\_root/uImage\_net\_new; bootm 30000000

set bootargs console=ttySAC0,115200 root=/dev/nfs nfsroot=192.168.1.123:/work/nfs\_root/fs\_mini\_mdev\_new ip=192.168.1.17:192.168.1.123:192.168.1.1:255.255.255.0::eth0:off

nfsroot=[<server-ip>:]<root-dir>[,<nfs-options>]

ip=<client-ip>:<server-ip>:<gw-ip>:<netmask>:<hostname>:<device>:<autoconf>

nfs 30000000 192.168.1.123:/work/nfs\_root/uImage\_nolcd; bootm 30000000

nfs 30000000 192.168.1.123:/work/nfs\_root/uImage\_i2c; bootm 30000000

nfs 30000000 192.168.1.123:/work/nfs\_root/uImage\_nonand; bootm 30000000

set machid 16a

tslib:

/usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc/usr/include/linux/input.h

#define EV\_VERSION 0x010000

改为

#define EV\_VERSION 0x010001

1. 3.4.2内核下的I2C驱动

1. 框架

1.1 硬件协议简介

1.2 驱动框架

1.3 bus-drv-dev模型及写程序

a. 设备的4种构建方法

a.1 定义一个i2c\_board\_info, 里面有:名字, 设备地址

然后i2c\_register\_board\_info(busnum, ...) (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

a.2 直接i2c\_new\_device, i2c\_new\_probed\_device

a.2.1 i2c\_new\_device : 认为设备肯定存在

a.2.2 i2c\_new\_probed\_device ：对于"已经识别出来的设备"(probed\_device)，才会创建("new")

i2c\_new\_probed\_device

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

info->addr = addr\_list[i];

i2c\_new\_device(adap, info);

a.3 从用户空间创建设备

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

a.4 前面的3种方法都要事先确定适配器(I2C总线,I2C控制器)

如果我事先并不知道这个I2C设备在哪个适配器上，怎么办？去class表示的所有的适配器上查找

有上一些I2C设备的地址是一样，怎么继续分配它是哪一款？用detect函数

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 \*/

};

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

i2c\_add\_driver

i2c\_register\_driver

a. at24cxx\_driver放入i2c\_bus\_type的drv链表

并且从dev链表里取出能匹配的i2c\_client并调用probe

driver\_register

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

/\* Walk the adapters that are already present \*/

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

\_\_process\_new\_driver

i2c\_do\_add\_adapter

/\* Detect supported devices on that bus, and instantiate them \*/

i2c\_detect(adap, driver);

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;

// 设置info.type

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

i2c\_new\_device

b. 驱动的写法

2. 完善设备驱动程序

3. 不自己写驱动直接访问

Device Drivers

I2C support

<\*> I2C device interface

4. 编写"总线(适配器adapter)"驱动

Device Drivers

I2C support

I2C Hardware Bus support

< > S3C2410 I2C Driver

nfs 30000000 192.168.1.123:/work/nfs\_root/uImage\_noi2cbus; bootm 30000000

1. 怎么看原理图

微机原理 ：侧重于计算机结构

数字电子技术基础 ：侧重于门电路

1．GPIO和门电路 ：general peripheral input/ouput

1.1 输入、输出引脚、上拉电阻、三极管

1.2 与门、或门、非门

1.3 中断

2．协议类

2.1 UART

2.2 I2C

2.3 SPI

2.4 NAND ： K9F2G08U0M

2.5 LCD

a. 看原理图：2440和外接的芯片引脚对接即可

b. 弄清楚接口的协议: 数据怎么传输？各引脚怎么配合？

c. 设置时序 : 2440发出的各个信号，要让外接的芯片能反应得过来

c.1 看2440手册弄清楚能设置哪些参数，这些参数的含义 (以CLK为单位)

c.2 看外设芯片手册，弄清楚这些参数的取值范围 (以秒为单位)

c.3 根据c.1,c.2计算2440的寄存器取值

3．类似内存的接口 : ram-like

3.1 NOR, SDRAM, 网卡

3.2 不同位宽外设的接线、访问过程

3.3 怎么确定访问地址、设置内存控制器

问1：地址线、数据线上接有众多设备，怎样做到只访问其中一个而避免其余的影响？

答1：每一个芯片都有单后的片选引脚，访问它之前先设置片选引脚为低电平；其他芯片的片选引脚为高电平

问2：是否需要我们手工来设置片选引脚？

答2：不需要，CPU访问某个地址时，CPU发出地址信号给"内存控制器", 内存控制器根据该地址决定让哪一个片选引脚输出低电平

4. 从头到尾看几个开发板的原理图

2410，2440，6410

1. 初接触开发板之基本操作

1. 接口、接线、装驱动、装软件

1.1 如果PC有并口，装并口驱动

1.2 安装USB串口卡的驱动, 以openjtag为例: 先接上去，然后按提示安装

1.3 JZ2440v2集成的USB串口：运行PL2303\_Prolific\_DriverInstaller\_v1.5.0.exe, 然后接USB线(板子上的USB-COM1口<===>PC的USB口)

1.4 安装烧写软件oflash: 把oflash.exe, FTD2XX.dll复制到c:\windows\system32

1.5 安装secureCRT

1.6 使用串口操作开发板

2. 烧写: 裸板,

4种方法: 并口, openjtag, jlink, nor上的uboot

2.1 使用并口工具烧写: 接线, 使用oflash烧写, 重新上电观察效果

2.2 使用openjtag烧写: 接线，使用oflash烧写(oflash烧写完后会复位开发板)

2.3 JLINK只能烧写NOR FLASH，烧好u-boot.bin，使用NOR启动

2.4 使用nor flash上的uboot来烧写

2.4.1 使用菜单通过USB下载烧写

2.4.2 使用TFTP下载烧写

a. 设uboot里的

set ipaddr 192.168.1.17

set serverip 192.168.1.4

b. 启动tftp服务

c. u-boot:

tftp 30000000 lcd.bin

nand erase bootloader

nand write 30000000 bootloader

3. 重烧系统: uboot, 内核, 文件系统

3.1 u-boot的烧写和烧写裸板是一样的

3.2 烧写内核: dnw, tftp

dnw: 在菜单里输入k, 然后使用dnw.exe发送文件

tftp:

tftp 30000000 uImage

nand erase kernel

nand write.jffs2 30000000 kernel

3.3 文件系统:

dnw: 在菜单里输入y, 然后使用dnw.exe发送yaffs2文件

或

dnw: 在菜单里输入j, 然后使用dnw.exe发送jffs2文件, 再参考使用手册P44设置bootargs

tftp:

tftp 30000000 fs\_qtopia.yaffs2

nand erase root

nand write.yaffs 30000000 0x00260000 $(filesize)

or:

tftp 30000000 fs\_qtopia.jffs2

nand erase root

nand write.jffs2 30000000 0x00260000 $(filesize)

烧写完后，输入reset

4. 解压使用我制作好的ubuntu

4.1 自己下载安装vmware 6.0.5以上版本

4.2 按开发板使用手册解压ubuntu

4.3 设置vmware的网络环境

4.4 使用vmware打开ubuntu

4.5 安装FTP工具CuteFTP Professional : 传文件

secure CRT ：远程登录工具

5. uboot打补丁、建source insight工程、编译、烧写

6. 内核打补丁、建source insight工程、编译、烧写

7. 制作、烧写根文件系统，使用NFS，编译使用驱动程序

7.1 仅用flash上的根文件系统启动后，手工MOUNT NFS

mount -t nfs -o nolock,vers=2 192.168.1.19:/work/nfs\_root /mnt

7.2 使用NFS作为根文件系统来启动

set bootargs noinitrd root=/dev/nfs nfsroot=192.168.1.19:/work/nfs\_root/tmp/fs\_mini\_mdev ip=192.168.1.17:192.168.1.19:192.168.1.1:255.255.255.0::eth0:off init=/linuxrc console=ttySAC0

nfsroot=[<server-ip>:]<root-dir>[,<nfs-options>]

ip=<client-ip>:<server-ip>:<gw-ip>:<netmask>:<hostname>:<device>:<autoconf>