如何使用eclipse看代码

-C/C++目录，file->new->C Project->Project name，使用已有目录的名字

-C/C++目录下可以open和close一个工程

-

**内容提要:**

嵌入式linux内核制作

Liux系统调用, r7

地址映射ioremap

字符设备驱动模型,cdev

设备控制理论,ioctl

混杂设备驱动模型, 主设备号为10,miscdevice

Linux中断处理程序, irq\_handler

工作队列，workqueue\_struct

阻塞型驱动设计，wait\_event

总线设备驱动模型，match，probe

平台设备驱动，platform\_match，probe

串口驱动程序设计

块设备驱动程序MTD

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

上学期

专题1

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

专题2-U-boot新手入门

拷贝uboot压缩包到linux系统

配置uboot，make forlinx\_nand\_ram256\_config

编译uboot，make ARCH=arm CROSS\_COMPILE=arm-linux-

生成uboot.bin,就是要烧写到板子里面的uboot程序，

Uboot烧写，在第2季第19课已经学习过，裸机开发快速体验.wmv

\*从SD卡启动

\*选择format the nand flash->burn image from usb->flash u-boot

insmod dnw\_usb.ko, 安装usb驱动

./dnw u-boot.bin 50000000

uboot命令

#help

#printenv

#print

#print ipaddr

#setenv

-setenv filename test.txt, 新增或修改

-setenv filename, 删除

#saveenv

文件下载,usb,网络,串口都可以下载

-虚拟机上运行一个tftp服务器

-开发板运行一个tftp客户端，就可以通过网络传送文件了

setenv ipaddr 192.168.1.100

setenv serverip 192.168.1.107

setenv ethaddr 00:40:5c:26:0a:5b

板子ping不通PC的原因:

-windows 防火墙

PC ping不通虚拟机的原因:

-/etc/init.d/iptables stop, 关闭防火墙

-setenforce 0, 关闭selinux防火墙。

不要用板子和PC直连的方式，要多ping几次才能ping通，

我遇到的板子ping虚拟机网卡的问题:

\*板子->主机物理网卡->主机虚拟网卡->虚拟机网卡

\*板子->路由器->主机物理网卡->主机虚拟网卡->虚拟机网卡

-网线连接到主机和板子，从板子ping主机的物理网卡可以ping通，但是ping主机的虚拟网卡ping不通

如果换成下面的方式，则通:

\*板子->路由器->主机物理网卡->虚拟机网卡,这里是禁用主机的虚拟网卡，当时主机不能上网，所以建立一个虚拟网卡用于主机和虚拟机能互通，

现在添加了路由器，主机物理网卡连接上，那么主机和虚拟机之间的通信就无需借助主机的虚拟网卡了，直接用真实的网卡

也就是说，主机的物理网卡和主机的虚拟网卡不能同时使用，会有冲突。

因为从主机ping板子，板子是不会回的。所以中间搭建一个交换机，大大增加ping通的几率

ping要分段:

-板子ping交换机

-板子ping主机物理网卡

-主机ping虚拟机网卡

注意：使用路由器是ping不通的

配置文件:

vi /etc/xinetd.d/tftp

/etc/init.d/xinetd restart

netstat -a | grep tftp

MMU映射之前使用50008000,映射之后用c0008000

通过tftp下载uImage

tftp 0xc0008000 uImage，下载内核

bootm 0xc0008000

md

mm

nand erase 0x400000 0x500000, 开始地址和长度

nand write 0x50008000 0x400000 0x500000

nand read 0x50008000 0x400000 0x500000

自动跳转到内核

bootcmd, 环境变量

bootcmd=nand read 0xc0008000 0x200000 0x500000;bootm 0xc0008000

自动从flash读取内核到ram，然后运行:

setenv bootcmd nand read 0xc0008000 0x200000 0x500000 \; bootm 0xc0008000

自动从tftp服务器读取内核到ram，然后运行内核:

setenv bootcmd tftp 0xc0008000 uImage \; bootm 0xc0008000

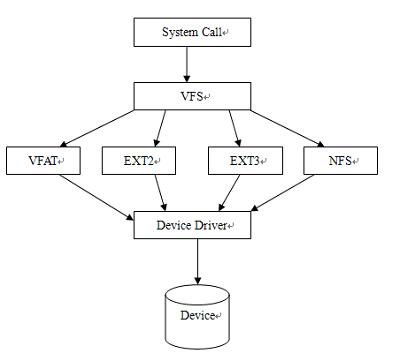
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

嵌入式linux内核制作

linux系统架构

用户空间，内核空间，那么用户程序就无法破坏操作系统的寄存器或资源

通过系统调用和硬件中断可以从用户空间转移到内核空间。



linux内核架构

-system call interface

-process management

-memory management

-arch，和cpu相关的代码

-device drivers

-virtual file system

-network stack

linux内核源代码

\*如何下载

-www.kernel.org

\*目录结构

-arch, archtecture，

-documentation

-drivers

-include

-fs, 各种文件系统的实现代码

-net，网络，实现网络协议

\*代码管理

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

第2课-Linux内核配置与编译

为什么要配置内核:选出需要的,去掉不要的.

-根据硬件需求

-根据软件需求

方法:一定是在内核代码顶层命令来运行的

make oldconfig, 就是把当前运行的linux内核的配置load进来

make config, 基于文本交互模式

make menuconfig,就是把当前运行的linux内核的配置load进来基于界面交互模式

<M>内核模块,E.o, D.o, 用于动态加载到内存，节省内存

<\*>静态连接到内核映像里面

内核配置的结果保存在什么地方:.config

配置原则:

-基于已有的配置文件进行配置

-不知道怎么配置的保持默认

pc的linux内核配置位置:

/boot/config

把配置文件拷贝到当前目录，可以基于这个配置进行配置

cp /boot/config-2.6.32-279.el6.i686 ./

ARM系列板子的配置文件

arch/arm/configs/

编译所有.o, .ko:

make

编译内核

make zImage, 这个内核要小于512k

make bzImage

make bzImage V=1, 会输出很多编译信息

编译好的内核放在:

arch/<cpu>/boot/目录下

编译内核模块, 结果会散落在各个目录下

make modules

归集，打包内核模块

-make modules\_install，把散落的.ko内核模块拷贝到/lib/modules下,为打包做准备，这条目录必须在linux最上层目录下执行

-make install, 会帮你把bzImage, system.map更名并拷贝到/boot下，并且更新各种软连接文件

-mkinitrd initrd-2.6.39 2.6.39, 把2.6.39目录打包，mkinitrd是一个工具，mkinitrd是一个工具, 用于建立要载入ramdisk的映像文件，任意目录下执行都可以，因为这个命令会自动在/lib/modules下找

-mkinitrd -o /boot/initrd.img-2.6.10 2.6.10, 有些系统使用这种格式

安装内核

-cp arch/x86/boot/bzImage /boot/vmlinuz-2.6.39

-cp initrd-2.6.39 /boot/

-修改/etc/grub.conf, 启动项修改，主要修改内核和内核模块的名字，有些系统是修改/boot/grub/menu.lst

uname -r， 查看当前内核

内核：运行在内存中

app和文件系统：存放到硬盘或flash，所以更新内核之后，看到的这些文件是一样的

清理内核代码：

-配置阶段，产生了.config

-编译过程中产生很多.o

make clean, 清理.o(.ko)文件，不清理.config

make distclean, 清理.o(.ko)文件，也清理.config

模块编译时的-C,-M:

KERNELRELEASE 是在内核源码顶层Makefile中定义的一个变量，在第一次读取本地Makefile的时候，

KERNELRELEASE为空，是没有定义的，所以走else分支。

-C $(KDIR) 指明跳转到内核源代码目录下读取那里的Makefile(这里定义了KERNELRELEASE);

-M $(PWD) 指返回到当前工作目录继续读入和执行当前的Makefile。

all: hello.o; hello.o 默认的规则是gcc -c -o hello.o hello.c, 所以会生成hello.o, 但是不会生成all,因为没有生成all的规则

all: hello.o; 既生成hello.o也生成all, 因为all也有源材料和规则

gcc -o all hello.o

all: hello.o; 即使all有源材料和规则, 但是因为声明all为伪目标,所以不生成all

.PHONY all

gcc -o all hello.o

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

嵌入式linux内核制作

\*make distclean,清除

\*cp config-file .config

\*make menuconfig ARCH=arm, 默认是x86，这里制定会运行在arm上

\*make uImage ARCH=arm CROSS\_COMPILE=arm-linux-, 编译内核

-有个错误

mkimage，这是一个工具，给image加一个头的，在uboot\_ok6410/tools里面有

cp mkimage /bin/

uboot, kernel, rootfs, 3个文件缺一不可，kernel在启动到最后会挂载根文件系统

uImage是zImage经过压缩得来的，压缩工具是mkimage. uboot目前只支持uImage启动,不支持zImage启动

mkimage在制作映象文件的时候，是在原来的可执行映象文件的前面加上一个0x40字节的头，记录参数所指定的信息，这样uboot才能识别这个映象是针对哪个CPU体系结构的，哪个OS的，哪种类型，加载内存中的哪个位置， 入口点在内存的那个位置以及映象名是什么

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

嵌入式文件系统

rootfs, 根文件系统，第一个文件系统，必须有的

-如何建立根文件系统目录与文件, 创建了目录之后, 要填充bin etc lib dev目录下的文件

--创建目录

--创建设备文件

--加入配置文件

--添加内核模块

--编译busybox

-挂载根文件系统到内核

文件系统: 目录 + 文件

文件系统: 目录 + 文件的加载方式

\*mkdir rootfs

cd rootfs

mkdir bin dev etc lib proc sbin sys usr mnt tmp var

mkdir usr/bin usr/lib usr/sbin lib/modules

\*创建设备文件，有2个是linux启动的时候必须要的

cd rootfs/dev

mknod -m 666 console c 5 1

mknod -m 666 null c 1 3

注意: mknod 创建一个特殊文件对应一个设备, 权限是666, 这里创建了名字为console, c为字符设备, 主设备号为5,次设备号为1

\*加入配置文件，和体系结构无关

tar etc.tar

cp etc/\* rootfs/etc/ -rf

\*添加内核模块

-cd ../linux-ok6410

-make modules ARCH=arm CROSS\_COMPILE=arm-linux-

-make modules\_install ARCH=arm INSTALL\_MOD\_PATH=../rootfs

\*编译和安装busybox

-busybox提供bin目录下的小工具

-tar xvzf busybox-1.13.3.tar.gz

\*make menuconfig,

-Busybox Settings->Build Options->Build BusyBox as a static binary.

是否静态链接, yes

用什么编译器

--cross compiler, arm-linux-

安装在哪里

--installation, don't user, 应该是/home/l4/rootfs

make,编译

make install, 把工具拷贝到我们的安装目录/rootfs下

第二个话题，挂载根文件系统

-文件系统类型

-initram

-nfs

文件系统类型

-RAM和flash



基于nandflash

-yaffs2,ubifs, 可读写

基于norflash

-jffs2, 可读写

基于内存的文件系统, 不能保存用户的数据

-ramdisk,指定大小, 早期的

-initramfs,不指定大小，可以扩展

基于网路的文件系统

-NFS，开发阶段使用

一个linux系统中一般使用多个文件系统

-根文件系统(一般使用initram快速启动), 再加一个flash的文件系统(可保存数据)

\*initramfs

-ln -s ./bin/busybox init, 建立软连接，这个如果没有，根文件系统挂载会失败

-到linux下, make menuconfig, General setup -> Initial RAM filesystem and RAM disk

-设置根文件系统的位置, /home/season4/1st/l4/rootfs

-make uImage ARCH=arm CROSS\_COMPILE=arm-linux-,把uImage拷贝到tftp目录

-通过串口设置环境变量,setenv bootargs noinitrd console=ttySAC0,115200 (第一个串口，波特率为115200)

-然后tftp下载，bootm启动，就挂载了文件系统了

init进程是由内核启动的第一个也是惟一的一个用户进程，它根据配置文件决定启动哪些程序，比如执行某些脚本，启动shell，运行用户指定的程序等。init进程是后续所有进程的发起者，比如init进程启动/bin/sh程序后，才能够在控制台上输入各种命令。

busybox是目标板系统上执行的第一个应用程序，当调用Busybox它会执行Busybox自身的init进程

\*\*\*\*\*\*\*\*\*\*\*\*\*

uImage: 包括kernel和压缩的文件系统, 内核启动的时候会把压缩的文件系统解压缩到ram中

频繁更改文件系统，通过nfs开发比较方便

-首先把linux配置成不需要initram系统的, general setting->取消initramfs

-然后在file syste->network file system-> rootfile system on NFS选项 下 网络配置中，添加支持NFS,rootfile system on NFS

-然后编译

设置启动参数，uboot告诉内核以什么参数来启动，文件系统就从nfs指定地方加载了

setenv bootargs noinitrd console=ttySAC0,115200 init=/init root=/dev/nfs rw nfsroot=192.168.31.105:/home/season4/1st/l4/rootfs,ip=192.168.31.106:192.168.31.107:192.168.1.1:255.255.255.0::eth0:off

bootargs, uboot告诉内核的启动方式

console, 控制台指定串口0，波特率115200

init,指定第一个初始化进程

root, 根文件系统取=/dev/nfs rw， 可读可写

nfsroot=192.168.31.105:/home/season4/1st/l4/rootfs, NFS网络文件系统路径

ip=, 第一个是板子IP，第二个是NFS服务器ip，第三个是网关，第四个是掩码，eth0是把板子ip赋给哪个网卡，off取消DHCP

检查主机是否配置好nfs服务器:

vi /etc/exports

home/season4/1st/l4/rootfs \*(rw,sync,no\_root\_squash)

/etc/init.d/nfs restart

使用nfs加载文件系统之后，pc机直接在上面放文件，在板子上查询很好查询

连网安装wireshare:

yum install wireshark,安装wireshark

yum install wireshark-gnome

yum erase wireshark,卸载 wireshark

rpm -qa | grep wire

whereis wireshark

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

内核模块基础

zImage, 所有组件都在这里，但是很费内存

动态加载的话，就能节省很多ram

什么是内核模块:

-不被编译进内核文件,zImage,bzImage

-动态的安装和卸载

安装与卸载

-insmod /home/dnw\_usb.ko

-rmmod dnw\_usb

-lsmod

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

内核模块设计

#include <linux/init.h>

#include <linux/module.h>

static int hello\_init()

{

printk(KERN\_WARNING"hellow world");

return 0;

}

static int hello\_exit()

{

printk(KERN\_WARNING"end world");

return 0;

}

module\_init(hello\_init);//加载函数

module\_exit(hello\_exit);//卸载函数

三要素:

-头文件

-加载函数

-卸载函数

insmod 执行的时候 会调用module\_init指明的函数hello\_init

rmmod 执行的时候 会调用module\_exit指明的函数hello\_exit, 模块要在/lib/modules/$(uname -r) 下才能卸载

mkdir -p /lib/modules/$(uname -r)

编写makefile和编译内核模块

C,大C代表进入到下面的路径执行命令

obj-m := hello.o

hello-objs := file1.o file2.o

KDIR := /Home/linux

all:

make -C $(KDIR) M=$(PWD) modules CROSS\_COMPILE=arm-linux- ARCH=arm

clean: rm -f \*.o \*.ko \*.order \*symvers

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

模块可选信息

-模块声明

-模块参数

-符号输出

\*模块声明

-MODULE\_LICENSE("GPL")

-MODULE\_AUTHOR("")

-MODULE\_DESCRIPTION("")

-MODULE\_VERSION("V1.0")

\*模块参数

int a = 3;

module\_param(a,int,S\_IRUGO | S\_IWUGO);//声明模块参数

insmod helloworld.ko a=10, 这样, 模块参数a=10了

char \*p;

module\_parm(p,charp,S\_IRUGO | S\_IWUGO);

insmod helloworld.ko p=hello

\*符号输出

-EXPORT\_SYMBOL(add), 声明这个符号可以给其他模块使用

-在使用add的模块中,extern add(int a, int b)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Linux内存管理

子系统由以下几个部分组成：

-系统调用

-进程管理

-内存管理

-文件系统

-网络协议栈

-Arch

-设备驱动

内存管理子系统

-内存管理模型

-地址映射管理

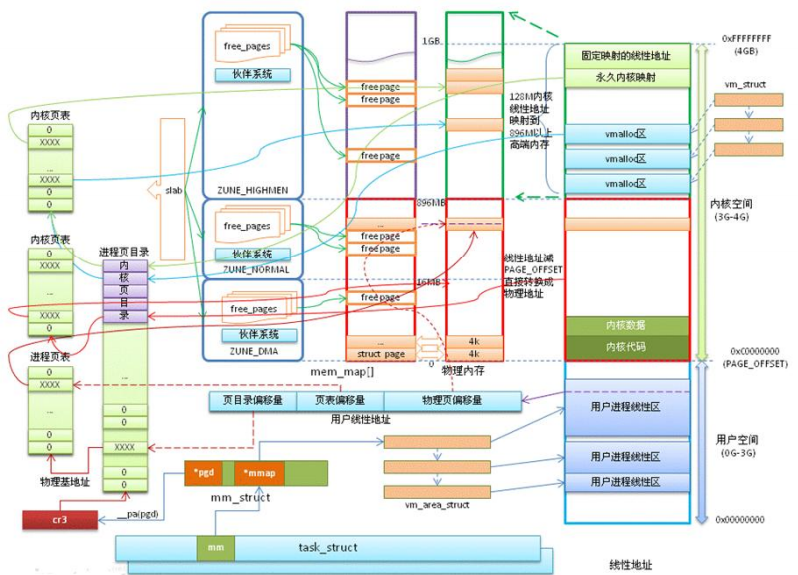
-物理地址分配管理

职能：

-物理内存的分配管理

-虚拟地址与物理地址的映射管理

管理模型:



看课件经典图, 对于物理内存有:

低端内存:0~896M

高端内存:896M以上

用户空间, 虚拟地址:

0~3G

内核空间, 虚拟地址：

直接映射区：3G~3G+896M，这部分是直接映射到物理地址0~896M

vmalloc区: 可以映射到高端或低端内存

永久内核映射区:固定用于访问高端内存的

固定映射的线性地址区:访问固定的寄存器地址

物理内存分配:

malloc、vmalloc分配一段虚拟内存的时候，并没有分配物理内存，而是访问虚拟地址的时候才会分配实在的物理内存

机制是通过缺页异常

kmalloc调用的时候会分配物理内存和虚拟地址

空闲页框: 物理空闲内存池

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

进程管理子系统

-linux进程要素

-进程调度

进程4要素

-有一段程序供其执行

-内核空间堆栈

-要有一个pcb，进程控制块 task\_struct

-有独立的用户空间

有独立的用户空间-Y->进程

|

N

|

有共享的用户空间-Y->用户线程

|

N

|

内核线程

linux进程状态：就绪，执行，阻塞

-TASK\_RUNNING,就绪或执行

-TASK\_INTERRUPTIBLE, 阻塞态, 可以通过发送信号唤醒

-TASK\_UNINTERRUPTIBLE，阻塞态,信号不可以唤醒

-TASK\_KILLABLE, 阻塞态,只原理类似有发送SIGKILL信号可以唤醒

-TASK\_TRACED,正在被调试

-TASK\_DEAD,进程退出时，调用do\_exit

PCB:进程控制块

-pid

-status

-priority

linux进程的调度

-从就绪的进程中选出最适合的一个来执行

要掌握哪些知识点:

-调度策略

-调度时机

-调度步骤

调度策略：

-SCHED\_NORMAL: 普通的分时进程

-SCHED\_FIFO: 先入先出的实时进程

-SCHED\_RR：时间片轮转的实时进程

-SCHED\_BATCH: 批处理进程

-SCHED\_IDLE: 只有在系统空闲时才能被调度执行的进程

调度时机: 也就是schedule()函数什么时候被调用，linux是通过调用schedule()来实现调度的

\*内核主动调用。当当前运行进程进入阻塞状态时，会主动调用

-current->state = TASK\_INTERRUPTIBLE;

-schedule();释放CPU

\*被动式，抢占式：用户态枪战(2.4,2.6)和内核态枪战(2.6)

用户态抢占:

-从系统调用返回用户空间

-从中断处理程序返回用户空间

内核即将返回用户空间的时候，如果need\_resched标志被设置(有更高级的用户进程)，会导致schedule()被调用,即发生抢占

-当某个进程耗尽它的时间片时，会设置need\_resched标志

-当一个优先级更高的进程进入可执行状态的时候，也会设置need\_resched标志。

用户态抢占缺陷

如果内核线程一直运行，不返回到用户空间，那么就不会产生调度释放CPU给用户进程。

直到这个内核线程执行完才能释放CPU，则降低实时性。

2.6版本改进方式: 内核态抢占

内核抢占可能发生在

-中断处理程序完成，返回内核空间之前，谁的优先级高就执行谁，不管是在内核还是在进程

-释放信号量的时候也可以发生抢占

下列情况不可以发生内核抢占:

-内核正在运行中断处理

-中断上下文处理

-进程正持有spinlock自旋锁,writelock/readloc读写锁等，不能被抢占。否则由于抢占将导致其他进程长期得不到锁，而让系统处于死锁状态。

-内核正在执行scheduler程序时,不能抢占。

为了保证linux内核在上述情况不会被抢占，抢占式内核使用了一个变量preempt\_count,称为内核抢占数。

当以上任何一种情况进入，preempt\_count会加1，当内核调度程序运行时，判断preempt\_count>0则不运行抢占。

调度的步骤，实际上是schedule函数是怎么完成调度的：

-清理当前运行的进程，退出CPU的清理工作

-根据策略从就绪队列中选择出要运行的进程

-设置新进程的运行环境，sp堆栈指针之类的

-进程上下文的切换

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

linux内核链表深度分析

-内核链表的使用

-内核链表的实现

-内核链表的移植

linux内核链表属于:双向循环链表

传统链表:

-链表指针指向和元素相关的类型

内核链表

-链表指针指向一个统一的指针域(指针头)，指针指的位置是下一个节点指针的位置

struct list\_head

{

struct list\_head \*pre;

struct list\_head \*next;

};

-使用内核链表，插入，删除元素，只需要调用函数就可以

INIT\_LIST\_HEAD, 创建链表

list\_add, 添加节点

list\_add\_tail, 添加节点到链表尾巴

list\_del, 删除节点

list\_entry,获取节点信息

list\_for\_each,遍历链表

学习内核代码最好看内核代码。

#include <linux/list.h>

struct score

{

int num;

int english;

int math;

struct list\_head list;

};

struct list\_head score\_head;

struct score stu1;

struct list\_head \*pos;

int mylist\_init()

{

score \*temp;

INIT\_LIST\_HEAD(&score\_head);

stu1.num = 1;

stu1.english = 90;

stu1.math = 50;

list\_add\_tail(&(stu1.list),&score\_head);

list\_for\_each(pos,&score\_head)

{

temp = list\_entry(pos, struct score, list); //pos 减去差值就可以得到score的地址

}

}

void mylist\_exit()

{

list\_del(&(stu1.list));

}

module\_init(mylist\_init);

module\_exit(mylist\_exit);

可以把这些链表函数移植到应用程序中去。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Liux系统调用

用户空间的应用程序调用内核空间的函数，实现体在内核。

每个系统调用有一个编号，在swi执行的时候携带编号参数提交给内核

内核根据编号,在sys\_call\_table找到相应的入口

read

swi + r7(2)

内核就从sys\_call\_table(calls.S)选取第2个系统调用进行调用

内核代码在:

entry-common.s, vector\_swi作为 入口，sys\_call\_table在calls.S, unistd.h里存储了系统调用编号

实现新的系统调用：

-在kenel写系统调用函数体

void sys\_pk()

{

printk("this is sys call test.");

}

-在arch\arm\kernel\call.s的sys\_call\_table里面要添加CALL(sys\_pk)，要加在最后面

-arch\arm\include\asm\unistd.h, 添加宏，定义新的系统调用编号

-用户应用程序来进行系统调用

void pk()

{

\_\_asm\_\_ (

"ldr r7,=363 \n"

"swi \n"

:

:

:"memory");

}

void main()

{

pk();

}

-编译这个应用程序然后放到rootfs中, 板子起来之后运行这个执行程序

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Linux开发前奏

概述

驱动分类

\*常规分类法

-字符设备，按字节为最小访问单位的设备，这种设备通常支持open,close,read,write,例如串口，led，按键

-块设备，最小访问单位为块(通常512字节)，flash，SD

-网络设备，可以是硬件设备如网卡，也可以是纯粹的软件设备，比如环回2接口lo，一个网络接口负责发送和接收数据报文

\*总线分类法

-USB设备，PCI设备，平台总线设备

驱动学习方法

-驱动模型

-硬件操作

驱动学习步骤:

-驱动程序模型(写框架代码)

-硬件操作实现

-驱动程序测试

驱动学习切忌: 驱动学习初期，请不要过多的去阅读内核代码

必要的内核代码，老师会带着我们读，其他的内核代码就没必要读了。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

硬件访问技术

-访问流程

-地址映射

-寄存器读写

硬件访问实质: 访问寄存器

地址映射

-linux中必须使用虚拟地址，所以把要访问的物理地址转换成虚拟地址，然后访问虚拟地址

-动态映射，void \*ioremap(physaddr, size)

-静态映射，用户事先指定映射关系，内核启动时，自动将物理地址映射为虚拟地址

--映射关系的指定，填充map\_desc, virtual(虚拟地址), pfn(物理地址页帧号)， 通过\_\_phys\_to\_pfn(物理地址)可以获得pfn

--告诉内核，通过s3c\_iodesc[]数组告诉内核

--内核启动时通过iotable\_init(s3c\_iodesc, ARRAY\_SIZE(s3c\_iodesc))完成静态的建立

寄存器访问:

unsigned ioread8(void \*addr)

unsigned ioread16(void \*addr)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

使用字符设备驱动程序

\*编译安装驱动

\*创建设备文件

\*访问设备

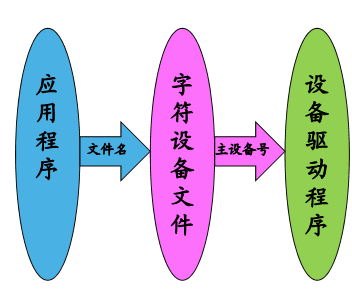
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

字符设备驱动模型

使用字符设备驱动程序

\*编译安装驱动模块

\*字符设备文件



-应用程序通过文件名找到字符设备文件->通过主设备号来找到对应设备驱动程序调用来控制设备, 一个字符设备文件对应一个主设备号，一个驱动程序也对应一个主设备号

-创建字符设备文件方法

--mknod /dev/文件名 c 主设备号 次设备号(0~255)，主设备号可以通过cat /proc/devices 查询到设备对应的主设备号

[Linux](http://lib.csdn.net/base/linux) 内核提供了一种通过 /proc 文件系统，在运行时访问内核内部[数据结构](http://lib.csdn.net/base/datastructure)、改变内核设置的机制。proc文件系统是一个伪文件系统，它只存在内存当中，而不占用外存空间。它以文件系统的方式为访问系统内核数据的操作提供接口。

系统中当前运行的每一个进程都有对应的一个目录在/proc下，以进程的 PID号为目录名，它们是读取进程信息的接口。

也就是说:

-Insmod的时候就可以在/proc/devices找到新增的主设备号

- mknod /dev/文件名 c 主设备号 次设备号(0~255), 就可以把这个文件通过主设备号和新增的设备进行关联

例子:

一个应用程序写设备(write\_mem)，另外一个应用程序读设备

查看一个函数需要什么头文件，可以通过man 2 open, man 2 write

./write\_mem 提示找不到的 错误 :

-/bin/sh: ./write\_mem: not found

arm-linux-readelf -d write\_mem, 可以查询到这个应用程序依赖的动态链接库

NEEDED Share library:[libc.sa.6]，这里是因为需要这个动态链接库，但是系里面没有

所以可以通过静态编译解决：

Arm-linux-gcc –static write\_mem.c –o write\_mem

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

字符设备驱动编程模型

1.设备描述结构cdev

2.字符设备驱动模型

3.范例驱动分析

要找到各种驱动模型的共性。

设备驱动模型,90%都使用这个模型去套

-驱动初始化

--分配设备描述结构

--初始化设备描述结构

--注册设备描述结构

--硬件初始化

-实现设备操作

-驱动注销

设备描述结构:每一种设备都有专门的结构来描述它,字符设备使用的描述结构是cdev

字符设备描述结构: cdev

struct cdev {

struct kobject kobj;

struct module \*owner;

const struct file\_operations \*ops;//设备操作

struct list\_head list;

dev\_t dev;//设备号，高12bit为主设备号，低20位为次设备号

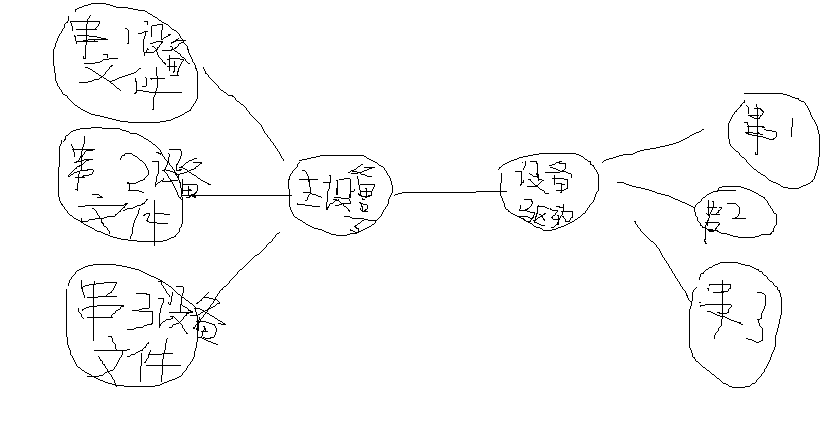
unsigned int count;//设备数

};

ls -l /dev/ 这里放了所有的设备文件,可以查询设备文件的主次设备号

一个设备驱动程序对应多个物理串口，但是每一个物理串口都有自己的设备文件，这些设备文件的主设备号

是一样的，但是次设备号不一样。



字符设备文件与字符驱动程序如何建立对应关系？

答案: 主设备号。

驱动程序用什么来区分串口1和串口2设备文件:

答案: 次设备号

dev\_t dev = MADEV(主设备号，次设备号)

主设备号 = MAJOR(dev\_t dev)

次设备号 = MINOR(dev\_t dev)

主设备号的分配方法:

\*静态申请

-register\_chrdev\_region, 如果设备号已经被占用会申请失败

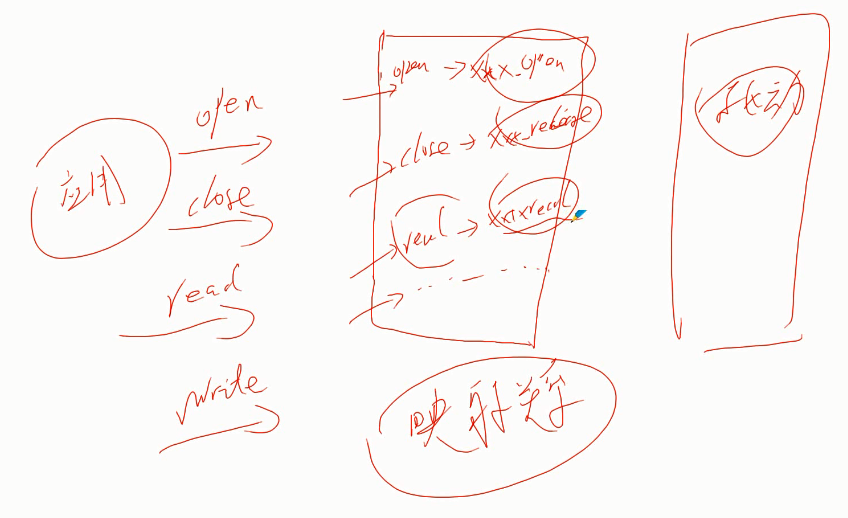
\*动态分配

-alloc\_chrdev\_region,不会申请失败

设备号注销:

-unregister\_chrdev\_region

file\_operations \*ops:并不是每一个操作函数都要实现的。



应用程序通过各种系统调用，调用到驱动程序的接口。这张映射关系表其实就是file\_operations 里面的各种指针。

下面真正开始设备驱动模型讲解。

第一步: 驱动初始化

描述结构-分配

\*静态分配

struct cdev mdev;

\*动态分配

struct cdev \*pdev = cdev\_alloc();

描述结构-初始化

cdev\_init(struct dev \*cdev, const struct file\_operations \*fops);

描述结构-注册,注册到内核

cdev\_add(struct cdev \*p, dev\_t dev, unsigned count);

-p,待添加到内核的字符设备结构

count, 该设备的设备个数

硬件初始化。

第二步，实现设备操作

-设备方法什么时候被调用

-如何实现设备方法

\*struct file

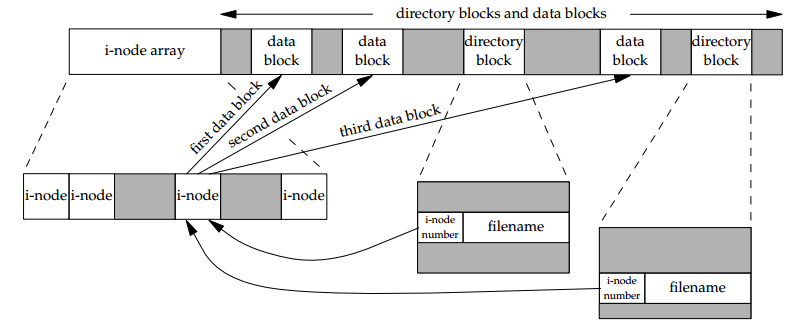
-linux系统中，每打开一个文件，都在内核中关联一个struct file,内核在打开文件时创建，在文件关闭后释放。

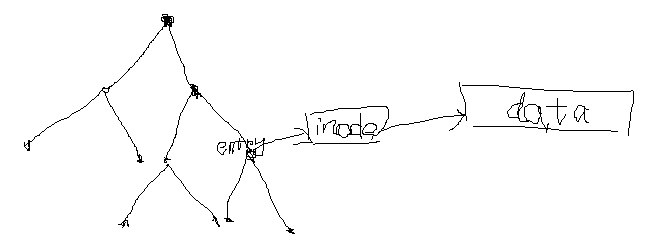
-重要成员:loff\_t f\_pos; struct file\_operations \*f\_op;

\*struct inode

每一个文件都关联一个inode,记录的是文件物理上的信息，次设备号只在inode里面有

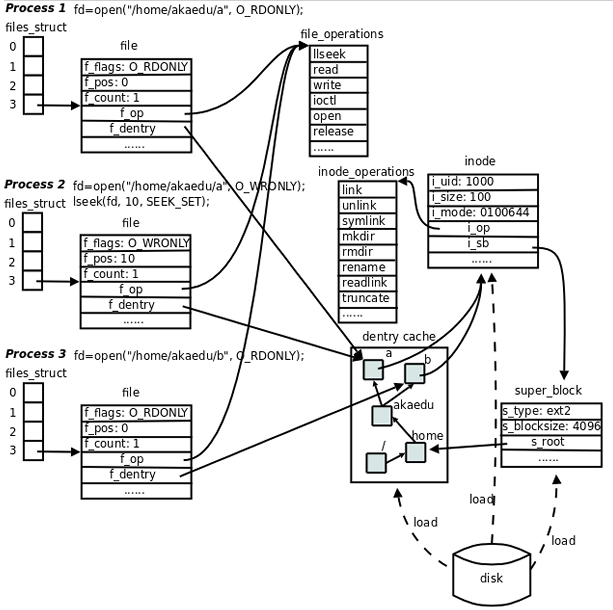
对于linux系统的文件系统来说，就是一个倒生长的树，每个树节点都是一个entry，entry指向一个inode，inode指向真正的文件内容数据。也就是说，文件内容数据不在这棵树上，而是在别的地方。





Linux内核的VFS子系统可以图示如下：非常好

http://www.cnblogs.com/Jezze/archive/2011/12/23/2299861.html



设备操作-read

-设备驱动从设备寄存器读取数据

-设备驱动将数据提交给应用程序

--buff参数是来源于用户空间的指针, 这类指针不能被内核代码直接饮用，必须通过专门的函数

--int copy\_from\_user(void \*to, const void \_\_user \*from, int n)

--int copy\_to\_user(void \_\_user \*to, const void \*from, int n)

驱动注销

-cdev\_del

范例代码分析: memdev.c

总结:

\*编写驱动模块代码，编译完成，拷贝到rootfs

\*板子启动，insmod memdev.ko， 安装模块成功，则在cat /proc/devices 可查询设备驱动的主设备号

\*mknod /dev/memdev0 c 253 0, 建立和设备驱动对应的一个设备访问文件

\*./write\_mem, ./read\_mem, 对文件发起访问

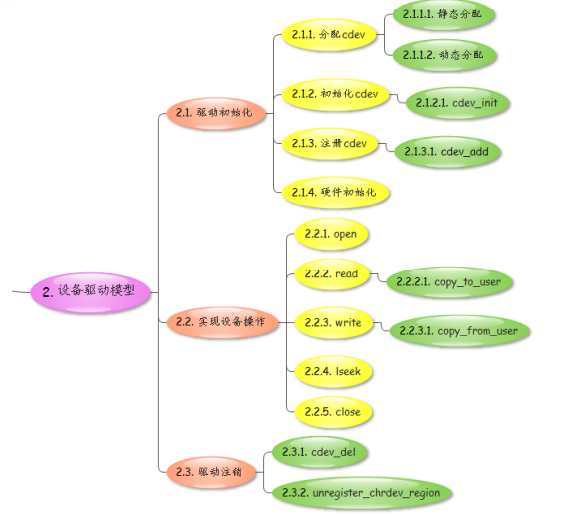
应用程序:





驱动程序：





\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

驱动访问模型

应用程序read/write ->内核?-> 驱动程序的read/write

内核是怎么做到的?

编译应用程序之后，通过arm-linux-objdump 来查看反汇编

r7 =3;

svc;系统调用指令，pc指针会从用户空间进入到内核空间,入口是固定的，取3出来,查找一个系统调用表, 查到什么调用什么

代码分析:

\*找到entry-common.s

\*找到vector\_swi,这就是入口

\*Get system call number

\*从sys\_call\_table(call.s)取出第3个函数sys\_read进行调用

\*sys\_read 可以查找read\_write.c文件

\*SYSCALL\_DEFINE3(read,unsigned int, fd, char \_\_user \*buf, size\_t, count)

\*file = fget\_light(fd, &fput\_needed);一个应用层的fd对应一个内核的file结构

-vfs\_read, file->f\_op->read()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

字符设备控制技术

-设备控制理论

-设备控制实现

-手动编码

设备控制理论

配置波特率，应用层可以采用控制系统调用

int ioctl(int fd, unsigned long cmd, ...)

-cmd,发送给设备的控制命令

-...，可选的参数，存在与否依赖于第2个参数

内核对应的方法:

2.6.36之前

long (\*ioctl)(struct inode \*node, struct file \*filp, unsigned int cmd, unsigned long arg)

2.6.36之后

long (\*unlocked\_ioctl)(struct file \*filp, unsigned int cmd, unsigned long arg)

设备控制实现：

\*定义命令,32bit划分为类型，序号，参数传送方向，参数长度4段

-type, 8bit, 表明属于哪个设备的命令

-number, 同一设备的不同命令

-direction, \_IOC\_READ, \_IOC\_WRITE(向设备写入参数), \_IOC\_NONE

-size,参数长度

linux提供了\_IOxx的宏来帮助定义命令。

例：

#define MEM\_MAGIC 'm'

#define MEM\_SET \_IOW(MEM\_MAGIC,0,int)

unlocked\_ioctl里面就switch 各种命令, 如果不能匹配任何支持的命令，返回-EINVAL.

\*实现设备方法

例子:

memdev.c

led.c





手把手编写

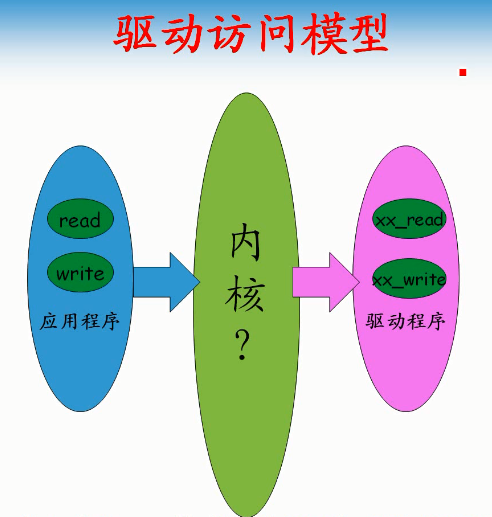
-重启设备

-设置参数

man ioctl可以查询ioctl需要包含什么头文件

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

驱动访问大揭秘



Read系统调用是怎么找到内核驱动xxx\_read的实现的呢？

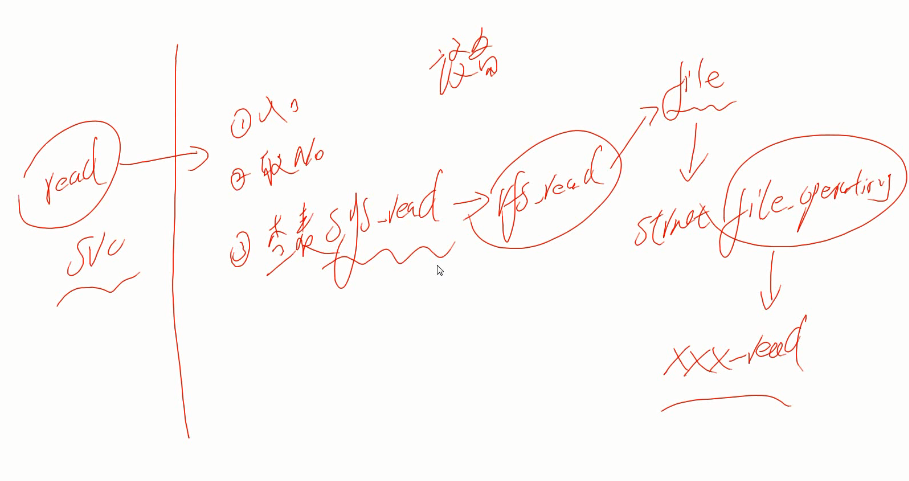
arm-linux-gcc -static -g read\_mem.c -o read mem

arm-linux-objdump -D -S read\_mem > dump.txt

查看汇编

mov r7, #3; 系统调用号存到r7寄存器

svc 0x00000000; 转入svc模式



内核中:

入口: vetor\_swi, entry-common.s

取系统调用编号: r7

查表:从sys\_call\_table内核系统调用数组中取第r7个调用来调

其实就是:

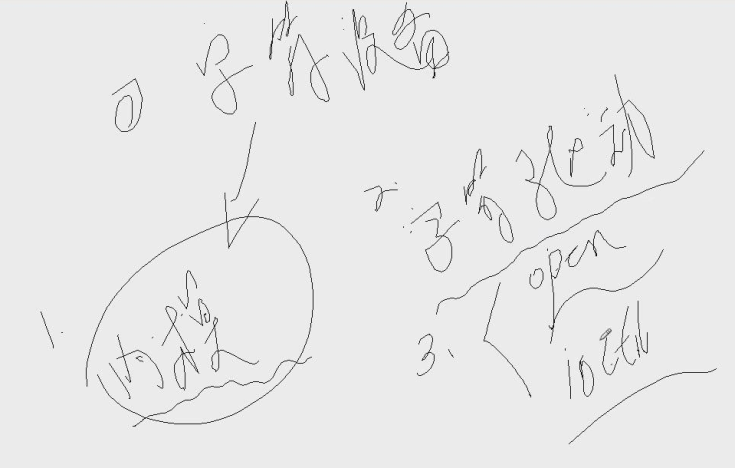
read(用户)->sys\_read(内核)->SYSCALL\_DEFINE3(read, ...)->vfs\_read->file->f\_op->read(驱动)

int fd 是应用层的文件描述符

structu file \*flip, 是内核的打开了的文件描述符

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

LED驱动程序设计



\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

专题一

混杂设备驱动模型

-混杂设备描述

-混杂设备驱动注册

-范例驱动分析

主设备号为10，但是次设备号不同，这些设备为混杂字符设备。

-结构==设备

-如何注册结构

struct miscdevice {

int minor;//次设备号

const char \*name;//设备名字

const struct file\_operations \*fops;//文件操作

struct list\_head list;

struct device \*parent;

struct device \*this\_device;

const char \*nodename;

mode\_t mode;

};

设备注册

int misc\_register(struct miscdevice \* misc)

LED混杂设备例子:

-初始化miscdevice,包括次设备号，名字和文件操作

-注册miscdevice, 使用misc\_register

-注销混杂设备，misc\_deregister(&key\_miscdev);



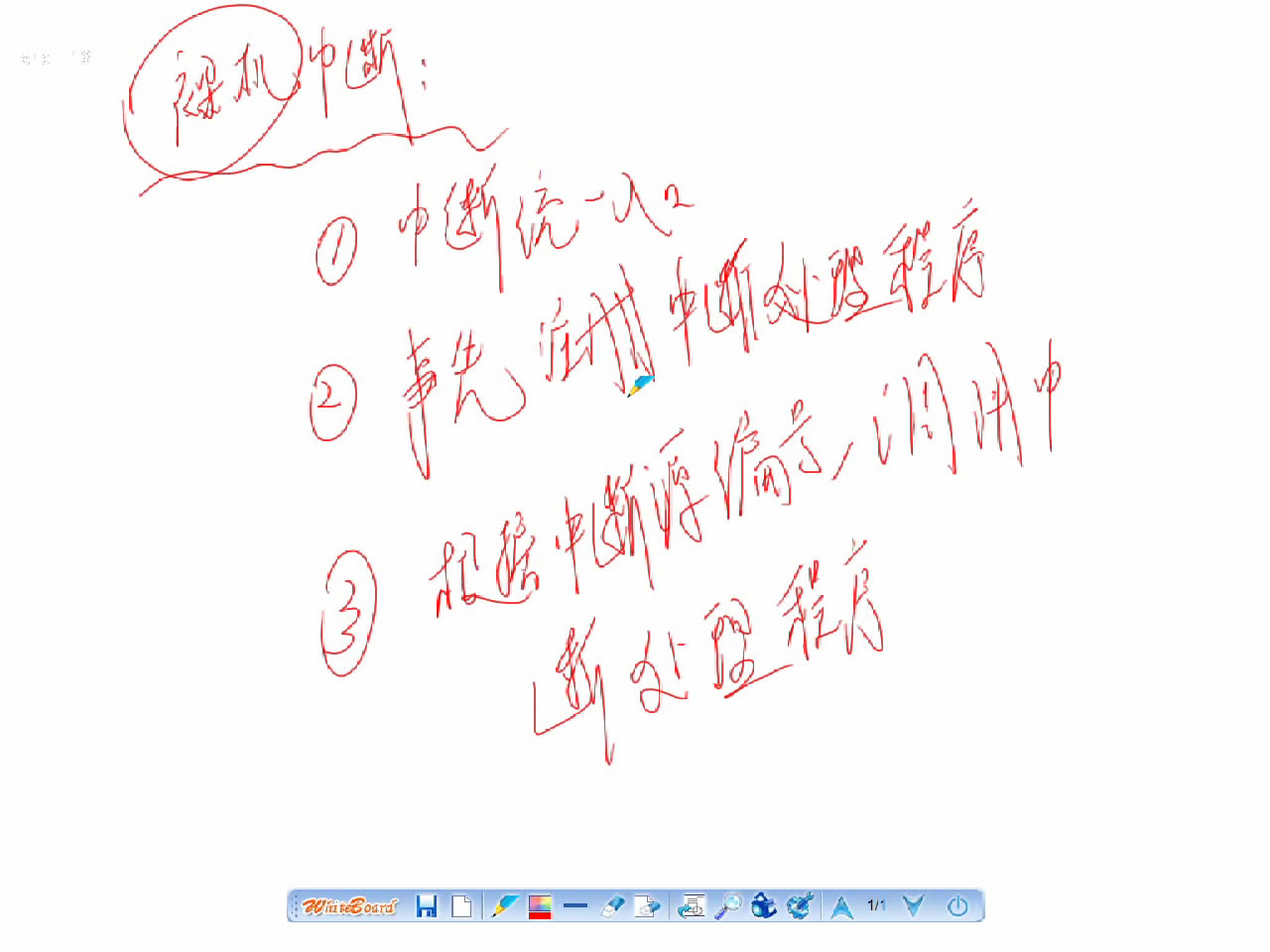
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Linux中断处理程序

-裸机中断处理流程

-linux中断处理流程

-添加到key混杂设备的例子



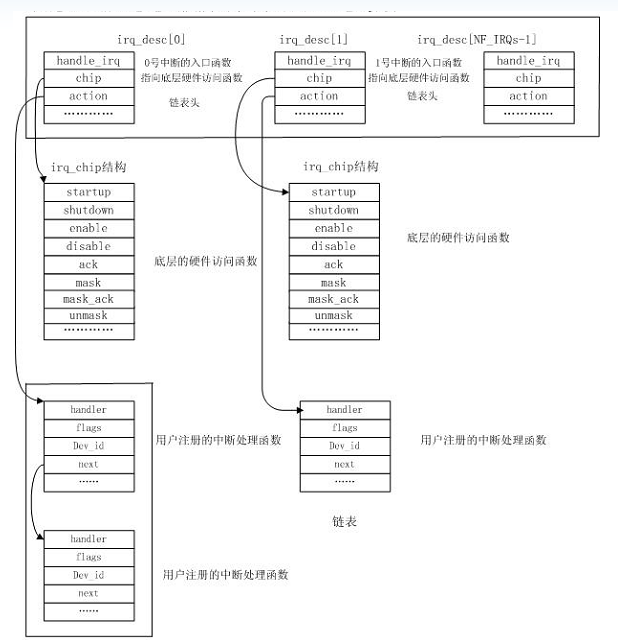
Linux中源文件分析

-entry-armv.s中中断统一入口为\_\_irq\_svc:

-entry-macro.s

\_\_irq\_svc->irq\_handler**->**arch\_irq\_handler\_default(获取产生中断的中断源)**->**asm\_do\_IRQ(irq)

generic\_handle\_irq (irq)**->**generic\_handle\_irq\_desc(irq)**->**desc->handle\_irq(irq, desc);



Linux中断流程总结

-\_\_irq\_svc, 总入口

-获取中断号

-根据中断号找到irq\_desc[n]结构

- desc->handle\_irq(irq, desc);调用事先注册的处理函数来运行， 一个中断号对应一个中断描述结构

驱动程序要干什么:

-事先中断处理程序

-处理程序要注册到对应的irq\_desc[irq]

Linux中断驱动流程

-注册

-处理函数

-注销

注册中断

int request\_irq(unsigned int irq,

irq\_handler\_t handler,

unsigned long flags, const char \*devname, void \*dev\_id)

flags: 与中断管理有关的各种选项

-IRQF\_DISABLED, 设置后表示是快速的不可以被打断的中断, 通过CPSR的IFbit来实现

-IRQF\_SHARE, 多个中断(网卡和串口)共享同一个中断号

devname: 设备名称

dev\_id：共享中断时使用，用于区分共享中断中不通的中断，相当于次中断号

中断处理程序是在中断上下文当中的

-不能调用可能阻塞的函数

-不能调用可能引起调度的函数

中断处理程序流程

-检查设备是否产生了中断

-清除中断产生标志,如果不清除，那么就无法响应下一次中断了， 这里注意CPU内部的中断寄存器默认CPU会清除，

但是外部IP核的，要在这里手动清除，例如DM9000内部有中断相关寄存器的，要清除

-相应硬件操作

注销中断

void free\_irq(unsigned int irq, void \*dev\_id)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

按键中断程序

硬件初始化一般在open函数或者模块初始化函数中。

void key\_hw\_init()

{

unsigned short data;

unsigned int \*gpio\_config;

gpio\_config = ioremap(GPFCON,4);

data = readw(gpio\_config);

data &= ~0b11;

data |= 0b10;

writew(data,gpio\_config);

}

request\_irq(IRQ\_EINT0,key\_int,IRQF\_TRIGGER\_FALLING,"tq2440key",0); 指定中断的下降沿触发

中断号怎么指定:实际硬件中断号+基数

在irqs.h可以查到中断号

#define S3C\_IRQ\_OFFSET (32)//预留32个中断号给swi中断

#define S3C\_IRQ(x) ((x) + S3C\_IRQ\_OFFSET)

#define IRQ\_VIC0\_BASE S3C\_IRQ(0)

#define S3C64XX\_IRQ\_VIC0(x) (IRQ\_VIC0\_BASE + (x))

#define IRQ\_EINT0\_3 S3C64XX\_IRQ\_VIC0(0)//等于32

Linux内核是怎么获取中断号的呢？

在entry-macro.s中arch\_irq\_handler\_default(获取产生中断的中断源)**->**

**ldr \irqnr, [\base, #INTOFFSET]; //从中断源寄存器中取出中断源**

**adds \irqnr, \irqnr, #IRQ\_EINT0; //中断号加上预留的基数**



http://blog.csdn.net/yimu13/article/details/6803957

这个逻辑的中断号是如何和芯片的中断号关联起来的呢？

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Linux中断分层技术

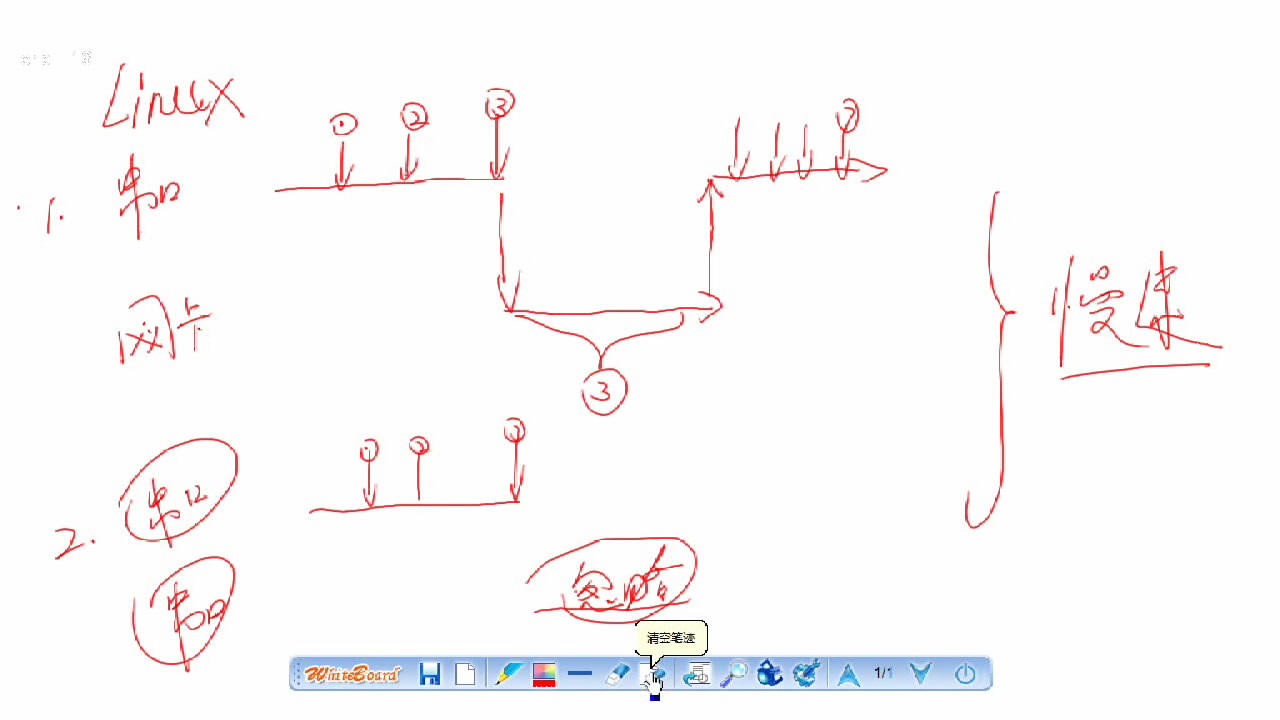
-中断嵌套

-中断分层方式

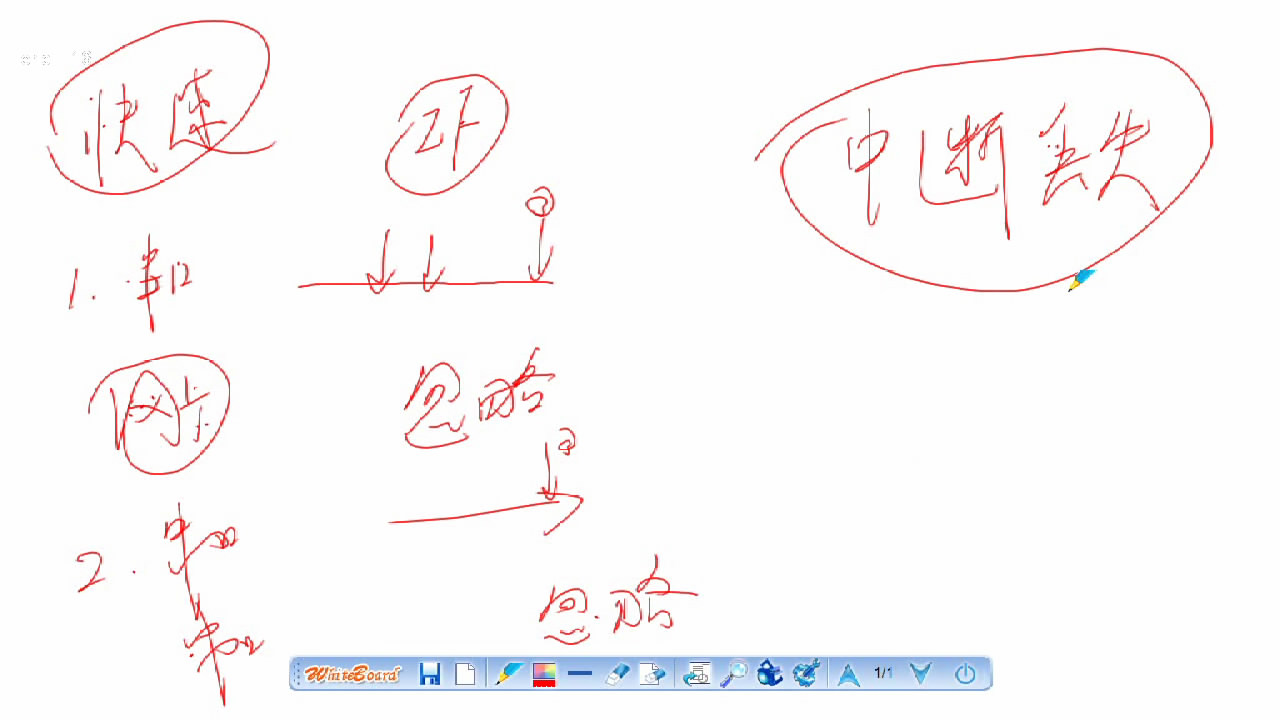
-使用工作队列来实现中断分层

中断嵌套

-慢速中断，进入中断后，CPSR的IF标志不禁止的



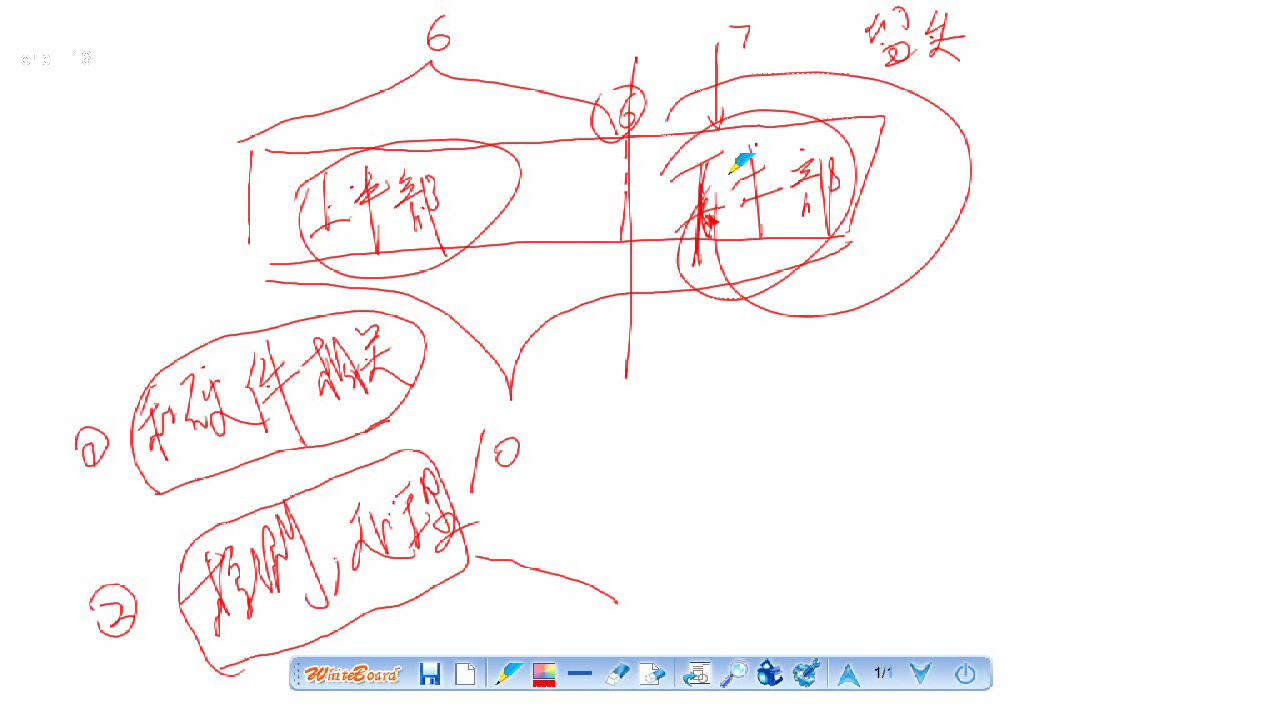
-快速中断，进入中断后，CPSR的IF标志禁止的, 根本产生不了其他中断，这个阶段其他中断是会丢掉的, 这就是中断丢失，是我们不希望的



如何解决中断丢失？ 需要用到中断分层技术

-中断处理时间要尽量短， 硬件相关的工作要在这里做，和硬件不相关的工作要移出中断处理程序中， 这就是所谓的中断分层技术

下面是中断处理函数，上半部应该为硬件处理部分，下半部分是和硬件无关的，则要抛到中断处理外部



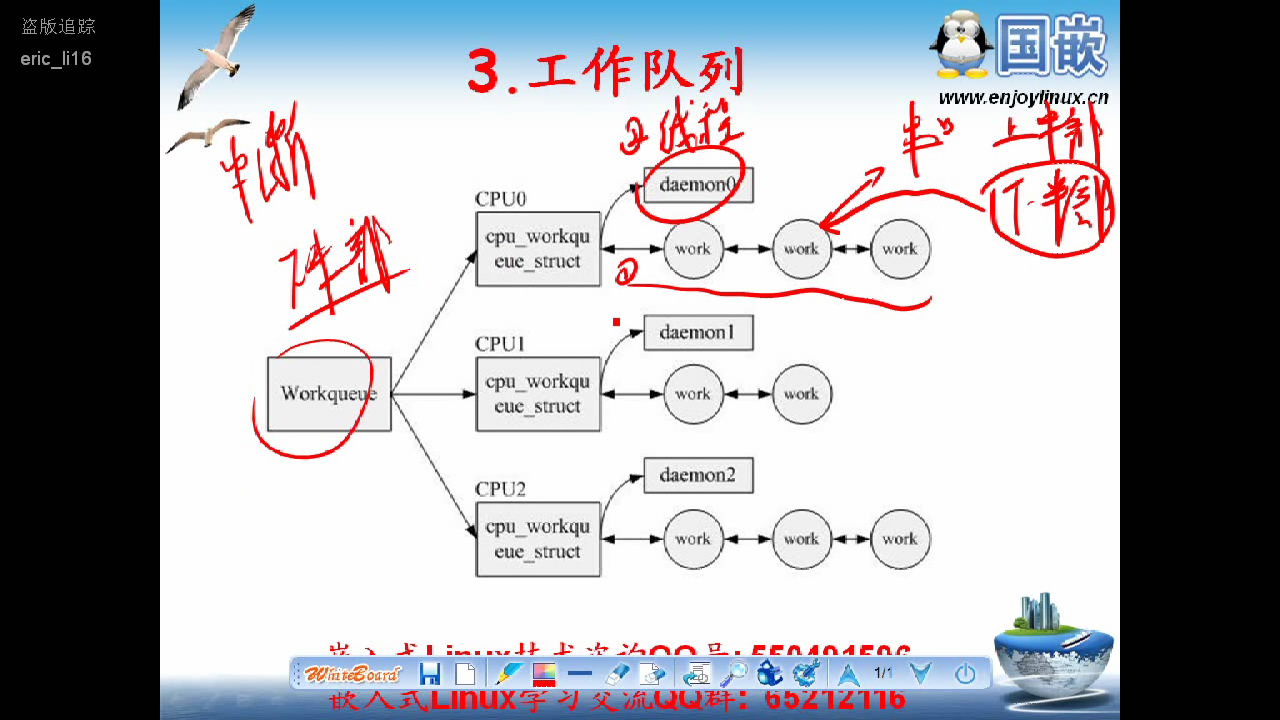
和硬件无关的部分如何抛到中断处理之外？

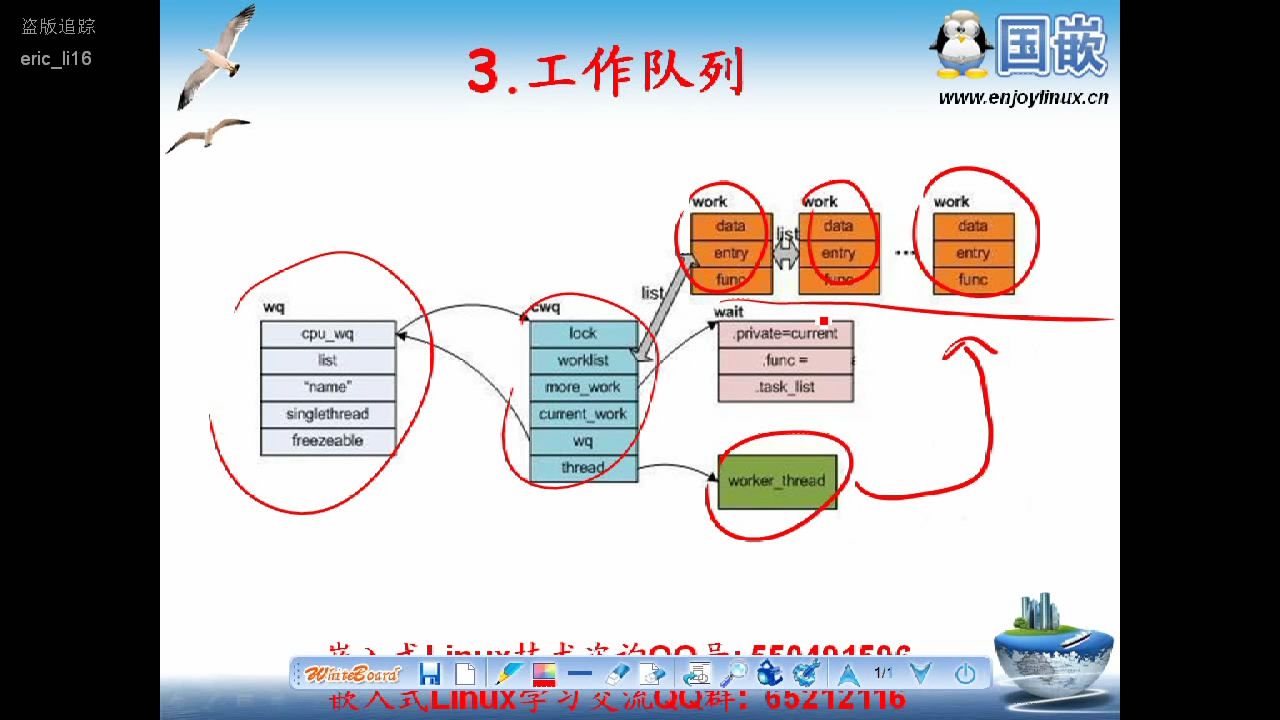
-软中断

-tasklet

-工作队列

工作队列

-每个被推后的任务叫“工作”（包括数据和处理函数），由这些工作组成的队列称为工作队列， 在内核比较空闲的时候，由Daemon线程来处理这些工作。



每个work都包含了数据和处理函数，线程只是在空闲的时候，从队列里面取出一项工作，并使用处理函数来处理这些数据而已。

workqueue\_struct来描述一个工作队列

struct workqueue\_struct {

unsigned int flags; /\* I: WQ\_\* flags \*/

union {

struct cpu\_workqueue\_struct \_\_percpu \*pcpu;

struct cpu\_workqueue\_struct \*single;

unsigned long v;

} cpu\_wq; /\* I: cwq's \*/

struct list\_head list; /\* W: list of all workqueues \*/

struct mutex flush\_mutex; /\* protects wq flushing \*/

int work\_color; /\* F: current work color \*/

int flush\_color; /\* F: current flush color \*/

atomic\_t nr\_cwqs\_to\_flush; /\* flush in progress \*/

struct wq\_flusher \*first\_flusher; /\* F: first flusher \*/

struct list\_head flusher\_queue; /\* F: flush waiters \*/

struct list\_head flusher\_overflow; /\* F: flush overflow list \*/

mayday\_mask\_t mayday\_mask; /\* cpus requesting rescue \*/

struct worker \*rescuer; /\* I: rescue worker \*/

int saved\_max\_active; /\* W: saved cwq max\_active \*/

const char \*name; /\* I: workqueue name \*/

#ifdef CONFIG\_LOCKDEP

struct lockdep\_map lockdep\_map;

#endif

};

work\_struct用于描述一项工作

struct work\_struct {

atomic\_long\_t data;

struct list\_head entry;

work\_func\_t func;

#ifdef CONFIG\_LOCKDEP

struct lockdep\_map lockdep\_map;

#endif

};

如何使用工作队列

-create\_workqueue, 创建工作队列

-INIT\_WORK， 创建工作

-queue\_work, 提交工作到工作队列

例子代码:

struct workqueue\_struct \*my\_wq;

struct work\_struct \*work1;

void work1\_func(struct work\_struct \*work)

{

printk("this is work1->\n");

}

int init\_que(void)

{ //1. 创建工作队列

my\_wq = create\_workqueue("my\_que");

//2. 创建工作

work1 = kmalloc(sizeof(struct work\_struct),GFP\_KERNEL);

INIT\_WORK(work1, work1\_func);

//3. 提交工作到工作队列

queue\_work(my\_wq,work1);

}

module\_init(init\_que);

一旦insmod que.ko, 就会打印"this is work1”, 内核线程会从工作队列中取出work来处理。

内核其实不需要创建工作队列的， 一般已经创建好了默认的工作队列keventd\_wq.

-创建工作

-提交工作，不用queue\_work, 而是使用schedule\_work(work1); 提交工作可以放到中断处理中

这个和在中断中post一个信号量给任务去处理数据道理是一样的。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

第6课-驱动支持多按键优化

多按键驱动优化, 可以响应中断，也可以响应app

-多按键支持, 注册多个中断

-按键访问应用程序设计

/\* 初始化一个定时器\*/

init\_timer(&buttons\_timer);

buttons\_timer.function = buttons\_timer\_function;

/\* 向内核注册一个定时器 \*/

add\_timer(&buttons\_timer);

//修改定时器超时时间并启动定时器

mod\_timer(&buttons\_timer, jiffies + (HZ /10));



\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

第7课，阻塞型驱动设计

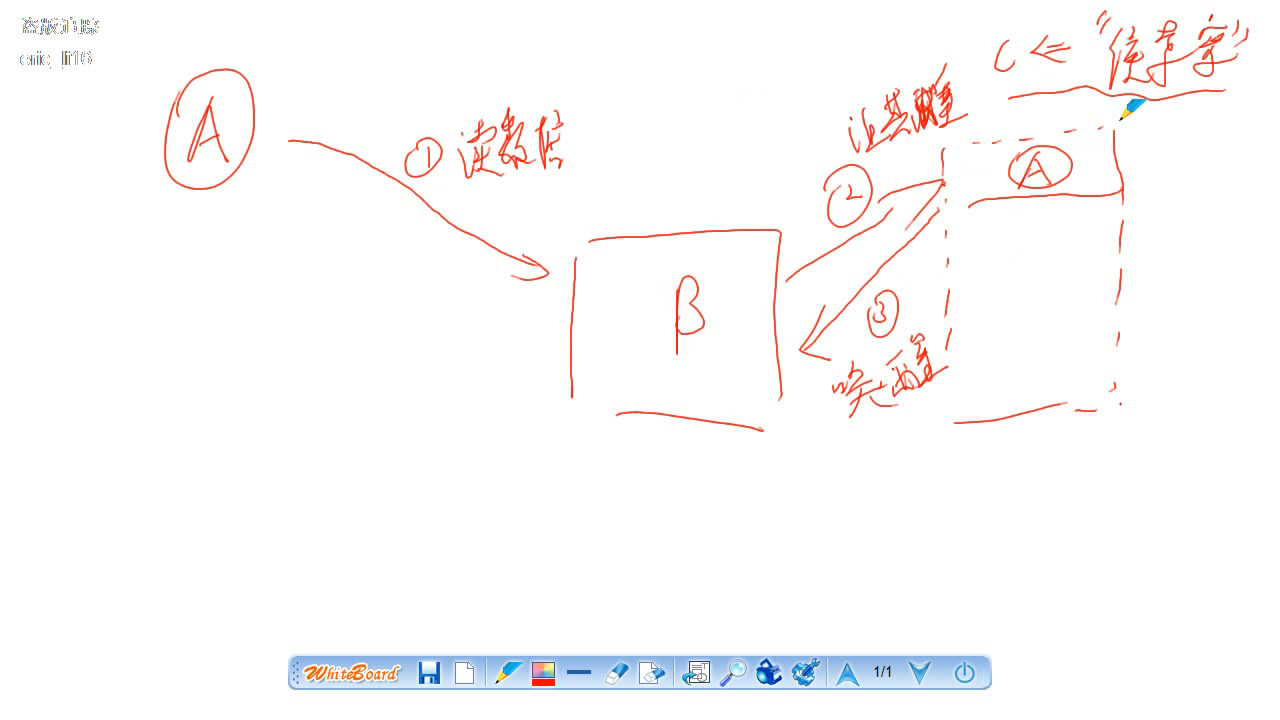
-阻塞必要性

-内核等待队列

-阻塞驱动优化

阻塞必要性

驱动程序应当缺省的阻塞进程，使进程进入阻塞状态，直到请求可以得到满足



A：进程

B： 设备资源

-A尝试读取设备B

-B没有数据，把A放入自己的候车室等待队列

-B有数据了，从自己的等待队列中唤醒A进行读取

候车室就是内核等待队列

-定义等待队列, wait\_queue\_head\_t my\_queue

-初始化等待队列, init\_waitqueue\_head(&my\_queue)

-定义+初始化, DECLARE\_WAIT\_QUEUE\_HEAD(my\_queue)

-进入等待队列，睡眠,

--wait\_event(queue, condition), 当condition为真, 则立刻返回，否则让进程进入

TASK\_UNINTERRUPTIBLE模式的不可中断睡眠状态，并挂在queue等待队列上

--wait\_event\_interruptible(queue, condition), 当condition为真, 则立刻返回，否则让进程进入

TASK\_INTERRUPTIBLE模式的可中断睡眠状态，并挂在queue等待队列上

--wait\_event\_killable(queue, condition),当condition为真, 则立刻返回，否则让进程进入

TASK\_KILLABLE模式的睡眠状态，并挂在queue等待队列上

-从等待队列中唤醒进程

--wake\_up(wait\_queue\_t \*q), 从等待队列q中唤醒状态为TASK\_UNINTERRUPTIBLE,TASK\_INTERRUPTIBLE,

ASK\_KILLABLE的所有进程, 然后后面会调度优先级高的进程

--wake\_up\_interruptible(wait\_queue\_t \*q),从等待队列q中唤醒状态为TASK\_INTERRUPTIBLE的所有进程

例子程序： 把之前的代码改成阻塞型的驱动

那这种驱动就是一种资源，类似以前的信号量等，都有一个等待队列



\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

专题2 第1课 总线设备驱动模型

-总线模型概述

-总线

-驱动

-设备

总线模型概述

图解总线设备模型

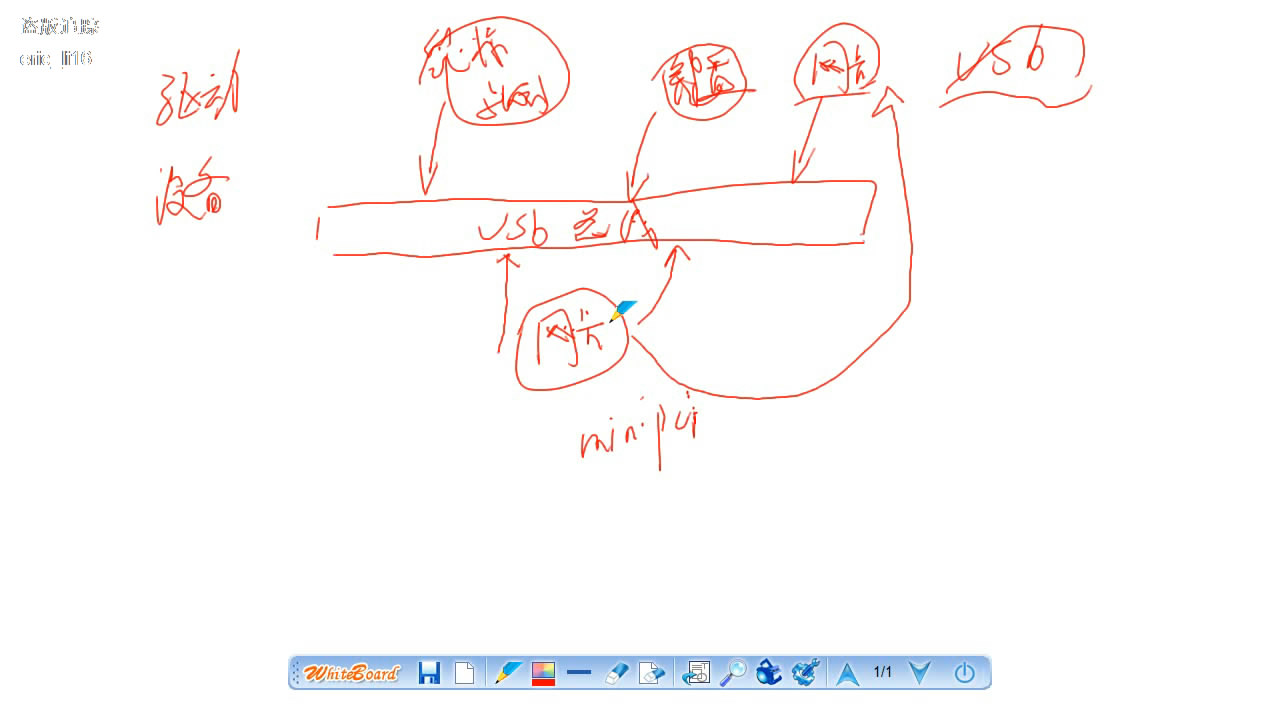
总线首先感知有新的网卡设备插进来，然后总线拿总线上的驱动和

新设备匹配，然后把控制权交给匹配上的网卡驱动了。当拔掉的时候，也给

网卡驱动来处理这种拔掉的处理。

-可以更好的支持热插拔

-可以更好的实现驱动移植性



编码步骤:

-创建一条总线

-各种驱动挂在到总线

-插入设备，总线调用match来匹配设备和驱动，找到相应驱动来调用驱动的probe函数

要点: 不同的总线，匹配规则会根据具体硬件而不同

总线

-描述结构

-注册

-注销

描述结构,bus\_type

struct bus\_type{

const char \*name;//总线名称

int (\*match)(struct device \*dev, struct device\_driver \*drv); //驱动与设备的匹配函数

总线的注册与注销

-注册, bus\_register (struct bus\_type \*bus), 若成功，新的总线被添加到系统，并可以在

/sys/bus下看到相应的目录

-注销, bus\_unregister(struct bus\_type \*bus),

例子：

//这里只是简单的以名字作为匹配规则，实际的设备例如USB，是使用硬件ID来进行匹配的。

int my\_match(struct device \*dev, struct device\_driver \*drv)

{

return !strncmp(dev->kobj.name,drv->name,strlen(drv->name));

}

如果使用dev->init\_nam，匹配的时候会出问题，init\_name 为空

int my\_match(struct device \*dev, struct device\_driver \*drv)

{

return !strncmp(dev->init\_name,drv->name,strlen(drv->name));

}

查看内核代码

int device\_register(struct device \*dev)

{

device\_initialize(dev);

return device\_add(dev);

}

int device\_add(struct device \*dev)

{

if (dev->init\_name) {

dev\_set\_name(dev, "%s", dev->init\_name);

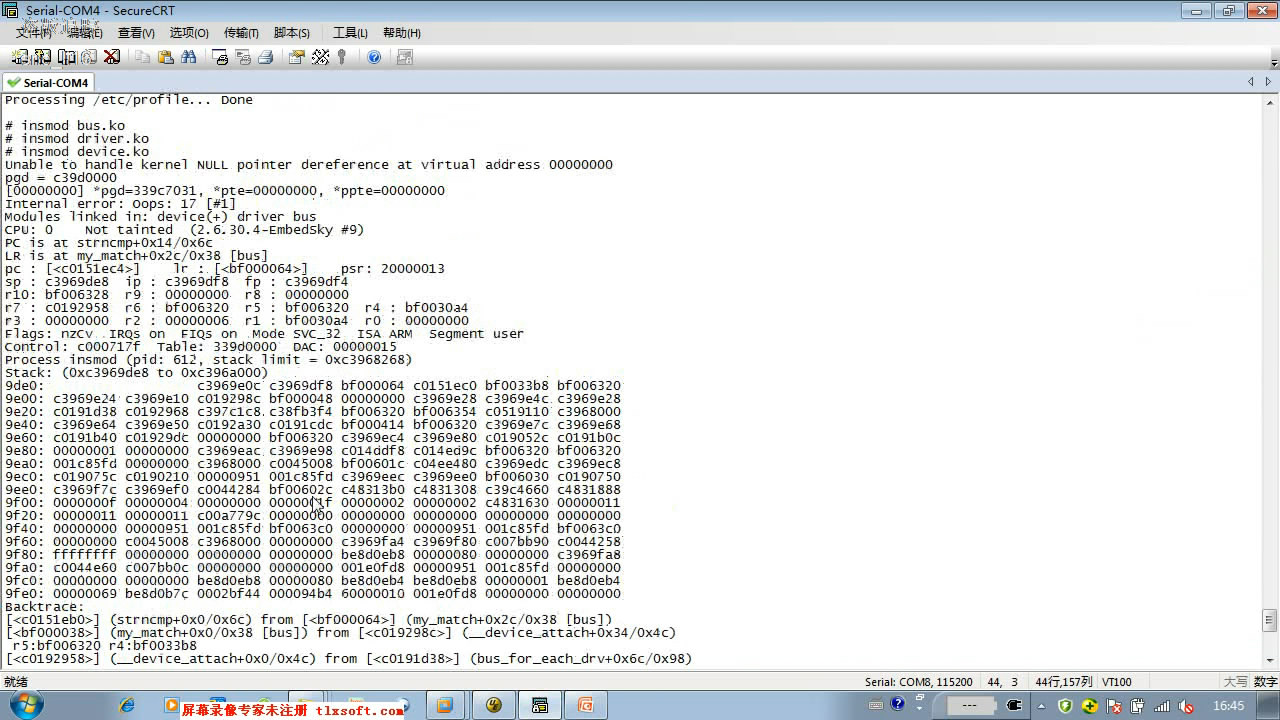
dev->init\_name = NULL;

}

...

}

内核异常截图与分析



-地址为空异常

-cpu当前寄存器信息

-stack信息

-backtrace, 调用栈信息

Insmod bus.ko出现没定义的字符，原因是本模块没有声明遵循GPL协议MODULE\_LICENSE("GPL");

内核对于不遵循GPL的模块，拒绝该模块访问内核符号表。

驱动

-描述结构

-注册

-注销

描述结构, device\_driver

struct device\_driver

{

const char \*name ;//驱动名称

struct bus\_type \*bus; //驱动所在总线

int (\*probe)(struct device \*dev);//当设备和本驱动匹配上后，总线会调用这个函数来初始化设备

...

}

注册与注销

-int driver\_register (struct device\_driver \*drv)

-int driver\_unregister (struct device\_driver \*drv)

例子：

EXPORT\_SYMBOL(my\_bus\_type);//在bus模块要输出这个符号给driver模块用

extern struct bys\_type my\_bus\_type;

struct device\_driver my\_driver = {

.name = "my\_dev",

.bus = &my\_bus\_type,

.probe = my\_probe,

};

insmod bus.ko

insmod my\_dev.ko

则在/sys/bus/driver 下面就有my\_dev这个驱动

设备

-描述结构

-注册

-注销

描述结构, struct device

struct device

{

const char \*init\_name;

Struct bus\_type \*bus;

}

-int device\_register(struct device \*dev)

-void device\_unregister(struct device \*dev)

obj-m := bus.o driver.o device.o 这里会生成3个模块

例子

struct device my\_dev = {

.init\_name = "my\_dev",//这里设备名字和驱动名字设置成一样，是为了作为匹配规则

.bus = &my\_bus\_type,

};

nsmod bus.ko

insmod driver.ko

insmod device.ko; //总线会把设备和名下驱动匹配，匹配成功后，会调用driver的my\_probe函数。

nsmod bus.ko

insmod device.ko;

insmod driver.ko//总线会把没找到驱动的设备和新驱动匹配，匹配成功后，会调用driver的my\_probe函数。

总结

-每添加新的 设备或新的驱动，总线都会去做匹配动作，很好的支持热插拔

-为什么说提高了驱动模块的可移植性？

例如将网卡驱动从USB总线移植到minipci总线上，只要把网卡驱动部分，挂在的总线从USB总线换成minipci总线就可以了

struct device\_driver my\_driver = {

.name = "my\_dev",

.bus = &my\_bus\_type,//这里可以换成别的总线

.probe = my\_probe,

};



插入设备的时候，是如何触发总线match的？

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

第2课， 平台设备驱动设计

-平台总线概述

-平台设备

-平台驱动

平台总线概述

Platform bus, 2.6之后加入的一种虚拟的总线，linux系统基本上不用自己去创建虚拟的总线，

内核已经提供了，我们基于这些平台总线进行开发就可以了

通过平台总线机制开发设备驱动的流程

-定义,platform\_device

-注册platform\_device

-定义platform\_driver

-注册platform\_driver

平台总线驱动与设备匹配机制， \driver\base\platform.c

struct bus\_type platform\_bus\_type = {

.name = "platform",

.dev\_attrs = platform\_dev\_attrs,

.match = platform\_match,

.uevent = platform\_uevent,

.pm = &platform\_dev\_pm\_ops,

};

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

{

struct platform\_device \*pdev = to\_platform\_device(dev);

struct platform\_driver \*pdrv = to\_platform\_driver(drv);

/\* Attempt an OF style match first \*/

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

return 1;

/\* Then try to match against the id table \*/如果驱动有ID表，则通过ID来匹配

if (pdrv->id\_table)

return platform\_match\_id(pdrv->id\_table, pdev) != NULL;

/\* fall-back to driver name match \*/通过名字来匹配

return (strcmp(pdev->name, drv->name) == 0);

}

平台设备

-platform\_device来描述

struct platform\_device

{

const char \*name; //设备名称

int id; //设备id

struct device dev;

u32 num\_resources;

struct resource \*resource; //设备资源，例如寄存器的基地址，中断号，这些都是资源

};

struct resource {

resource\_size\_t start;

resource\_size\_t end;

const char \*name;

unsigned long flags;//资源类型

struct resource \*parent, \*sibling, \*child;

};

static struct resource key\_resource[] = {

[0] = {

.start = GPFCON,

.end = GPFCON + 8,

.flags = IORESOURCE\_MEM,

},

[1] = {

.start = IRQ\_EINT0,

.end = IRQ\_EINT2,

.flags = IORESOURCE\_IRQ,

},

};

-注册, int platform\_device\_register(struct platform\_device \*pdev)

-注销, int platform\_device\_unregister(struct platform\_device \*pdev)

平台驱动

-描述符

struct platform\_driver

{

int (\*probe)(struct platform\_device \*);//插入匹配成功后调用

int (\*remove)(struct platform\_device \*);//拔掉设备时候调用

...

}

例子， 将按键驱动修改为平台驱动模型

对于 这种平台驱动模型，由于设备不一定已经插入，所以硬件初始化不能放到module\_init函数中，

要把初始化工作放到probe中，这和默认认为设备存在的驱动程序不一样的地方。

总线驱动主要起到找到这个设备的作用。总线驱动把设备资源和驱动方法分开，不能像之前那样，

把设备资源耦合到驱动方法中，驱动中要使用某些寄存器地址和中断号等资源，必须通过设备资源来获得。

这个例子很好，一定要看

key\_dev.c

key\_drv.c

insmod key\_dev.ko

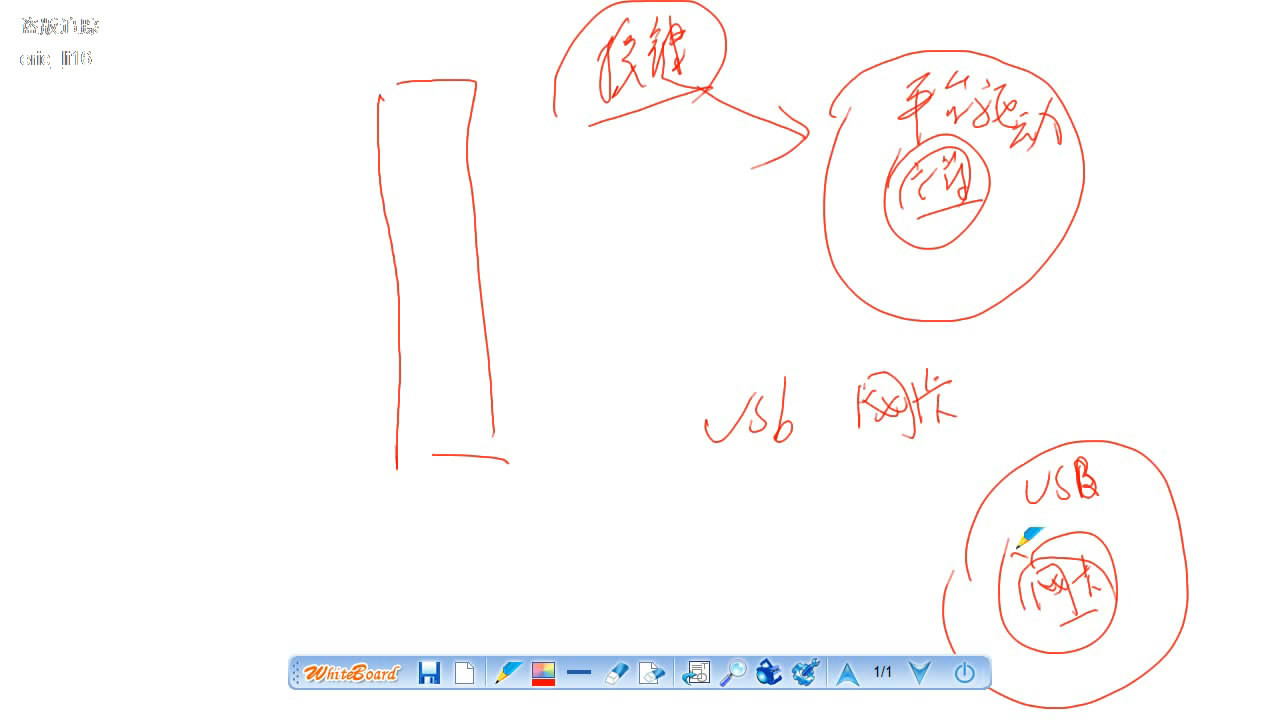
insmod key\_drv.ko

my-key: probe of my-key.o failed with error -1090503240//这个出错主要是probe函数返回错误了

平台的设备注册说白了就是一堆资源变量，可以在内核初始化的时候就调用

总结：

平台驱动只负责找到设备并初始化设备，后面才是设备真正的驱动



代码中各种描述符的关系

platform\_device->device

platform\_driver->driver

key\_miscdev



当向linux系统总线添加设备或驱动（driver\_register/device\_register）时，总是会调用各总线对应的match匹配函数来判断驱动和设备是否匹配. 插入硬件就会向总线添加设备吗？

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

专题3-串口驱动程序设计

第1课 tty驱动架构

TTY 概念解析

-通常使用tty来简称各种类型的终端设备。

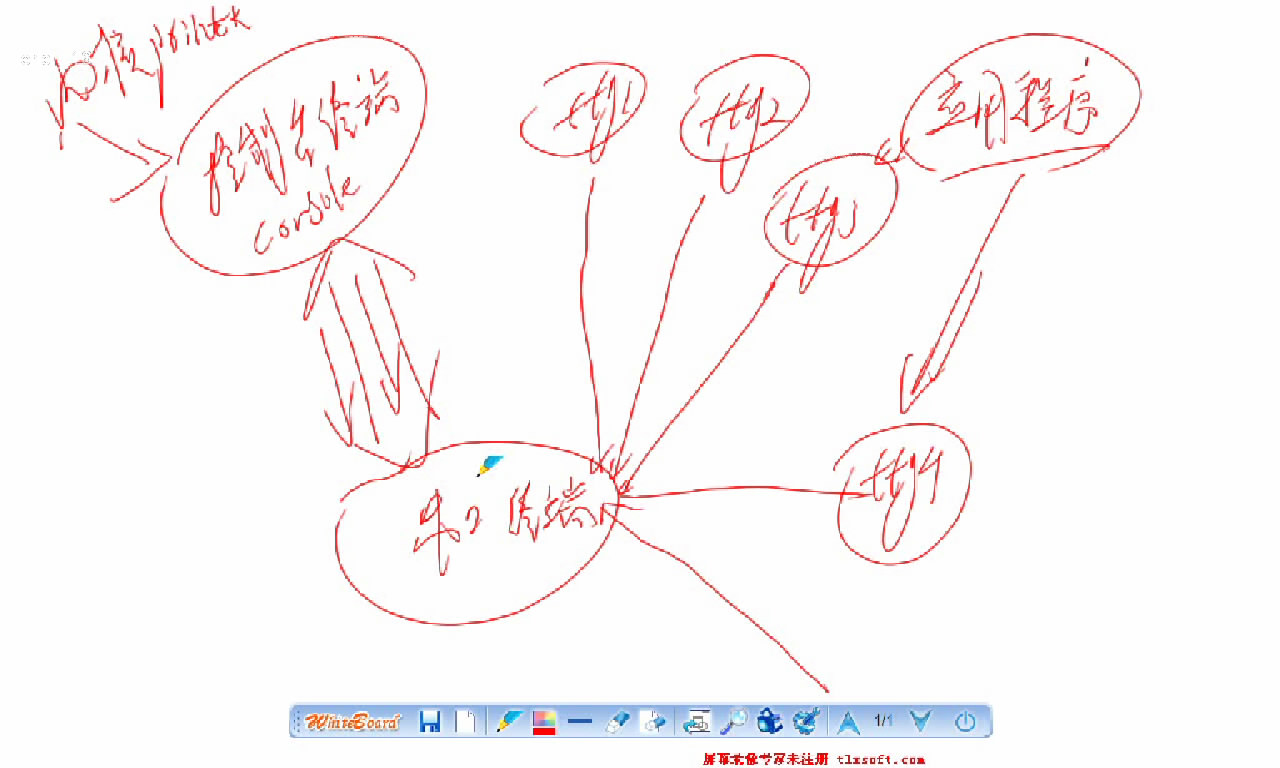
分类

-串口终端(/dev/ttyS\*), ttySAC0, ttySAC1

-控制台终端(/dev/console), 就是printk函数输出到的设备。/dev/console是一个虚拟的设备，必须要和一个实际的硬件关联起来。比如内核启动参数”console=ttySAC0”

所以, printk ->虚拟console ->ttySAC0或显示器

-虚拟终端(dev/tty\*), ctrl + alt + [F1-F6]可以切换虚拟终端，通过tty命令可以查询当前使用的是哪个虚拟终端。tty0实际上就是当前使用的虚拟终端，如果当前使用的是tty3，那么tty0=tty3



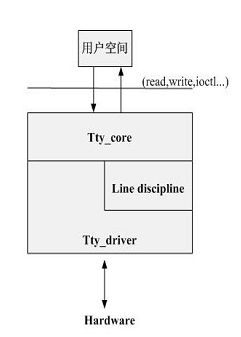
TTY架构分析

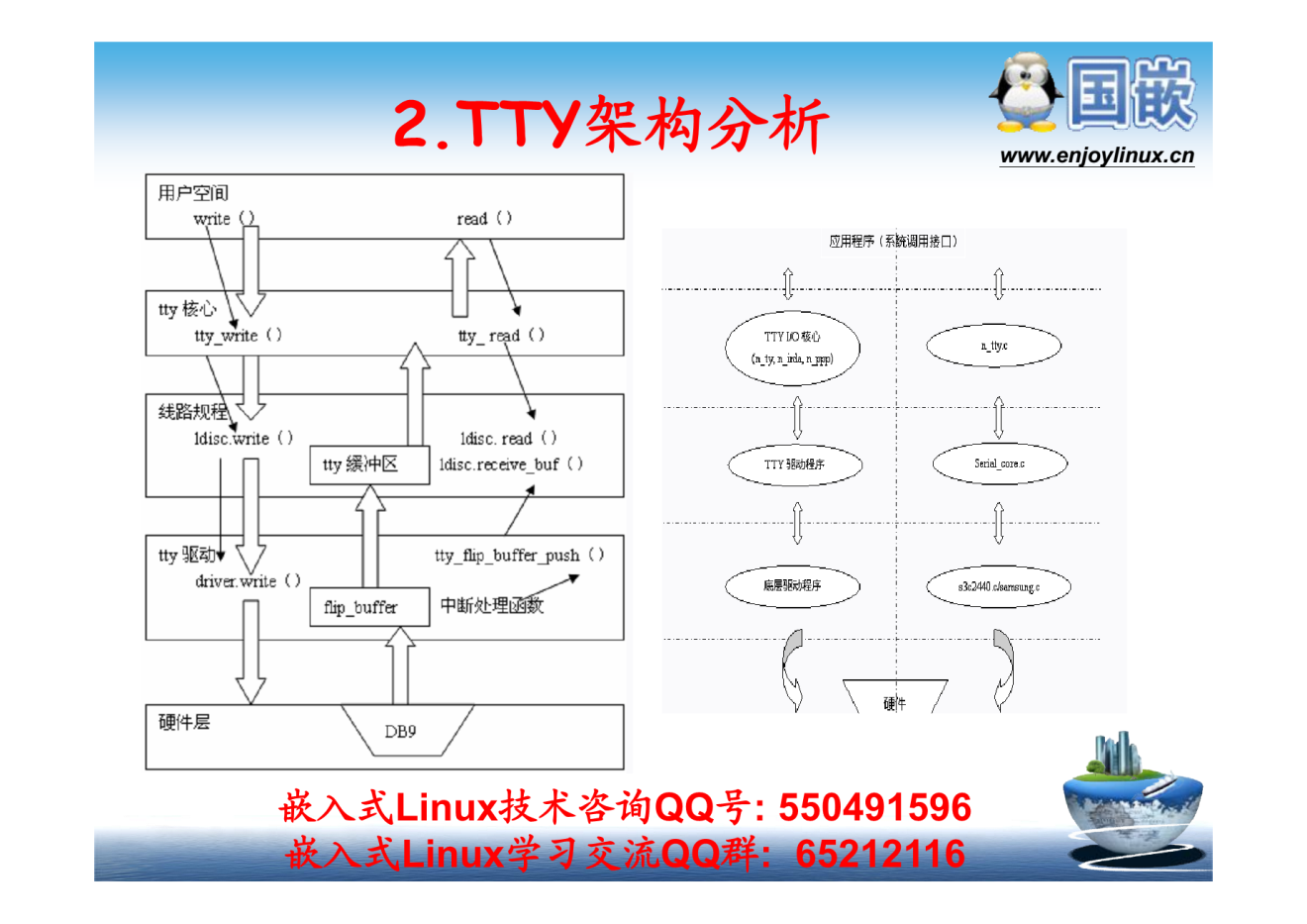
TTY子系统包含

-tty核心, 起到对用户程序统一接口的作用

-tty线路规程, 协议相关

-tty驱动, 控制实际硬件





串口发送回溯-函数调用流程

串口驱动中的发送函数s3c24xx\_serial\_start\_tx， /drivers/tty/serial/Samsung.c

在s3c24xx\_serial\_start\_tx中调用dump\_stack(); 就可以把调用关系dump出来。

-s3c24xx\_serial\_start\_tx

//-redirected\_tty\_write->do\_tty\_write

-tty(file->private\_data)->tty)->uart\_ops(**tty\_operations**, s3c24xx\_serial\_ops)->uart\_write, tty驱动，这里的tty是什么东西?

-do\_tty\_write(ld->ops->write, tty, file, buf, count);线路规程的ops(**tty\_ldisc\_ops** tty\_ldisc\_N\_TTY)->n\_tty\_write

-file(**file\_operations**)->f\_op->write, 实际上是tty\_cdev->tty\_fops->tty\_write, tty\_cdev是一个cdev，对应"/dev/tty", 这层属于tty core driver

-vfs\_write, 这里其实已经是内核态了

-应用层的write

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

第2课 串口驱动程序初始化

-串口驱动程序结构

-串口驱动中的重要数据结构

-初始化分析

串口驱动程序结构

-初始化

-打开

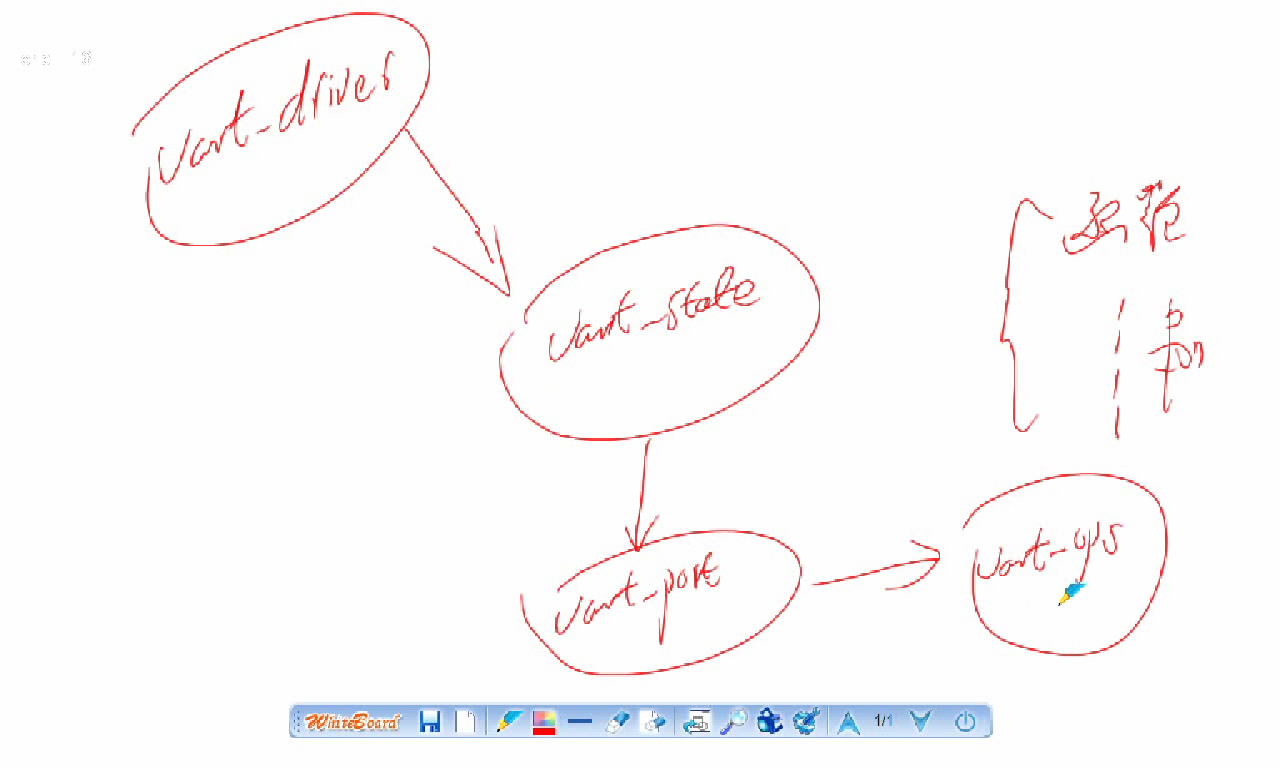
-读

-写

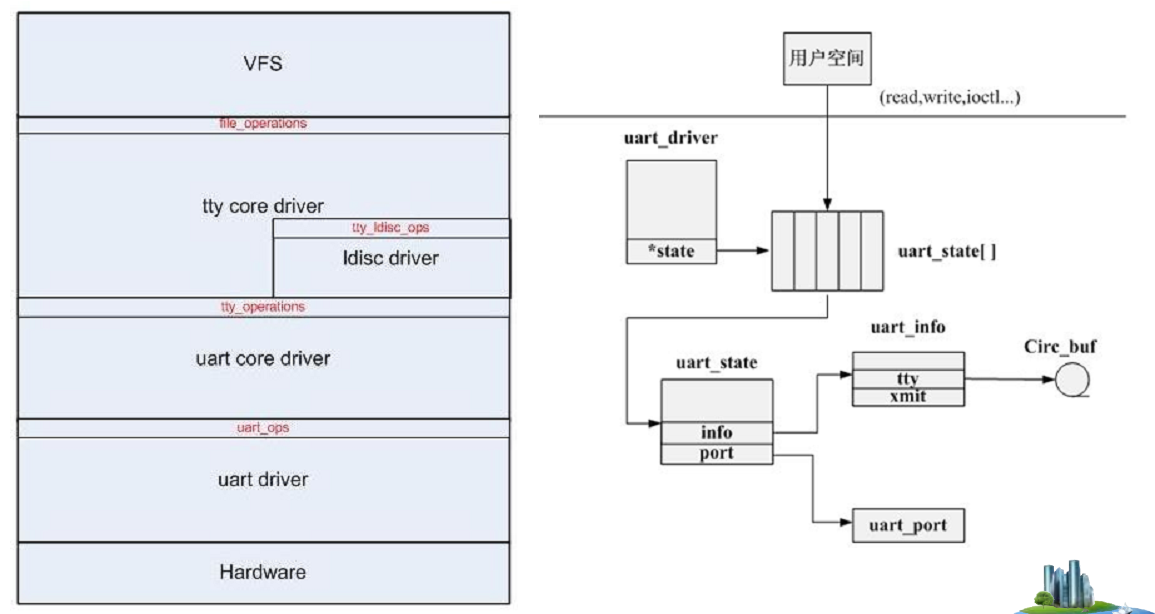
/drivers/tty/serial/samsung.c->通用

/drivers/tty/serial/s3c6400.c->和设备相关。

Write的调用关系



串口驱动中的数据结构



-

从uart\_write开始分析

-uart\_state \*state = tty->driver\_data;

-port = state->uart\_port;

-port->ops, 操作硬件

uart\_open:state从何而来

state = uart\_get(drv, line);, 从uart\_driver拿到的, struct uart\_driver s3c24xx\_uart\_drv

tty->driver\_data = state;

UART驱动程序结构: struct uart\_driver, 对应一个串口驱动， 像平台驱动

UART 端口结构 ： struct uart\_port, 一个uart\_port代表一个串口

UART相关操作函数结构: struct uart\_ops

UART 状态结构: struct uart\_state

UART 信息结构: struct uart\_info

Uart\_driver ======tty\_driver---tty\_fops

|

cdev

串口初始化

在samsung.c中，

-uart\_register\_driver注册串口驱动s3c24xx\_uart\_drv

cdev\_init(&driver->cdev, &tty\_fops);

tty\_register\_driver(struct tty\_driver \*driver)

uart\_register\_driver(struct uart\_driver \*drv), uart\_driver->tty\_driver = normal; tty\_driver.driver\_state = uart\_driver, tty\_set\_operations(normal, &uart\_ops);

uart\_register\_driver(&s3c24xx\_uart\_drv);//"s3c2410\_serial"

s3c24xx\_serial\_modinit()

在s3c24xx.c中

-由于是平台驱动，分析probe函数， 取出匹配的port数据，并调用s3c24xx\_serial\_init\_port初始化某个port

--获取设备虚拟基地址

--获取设备中断号

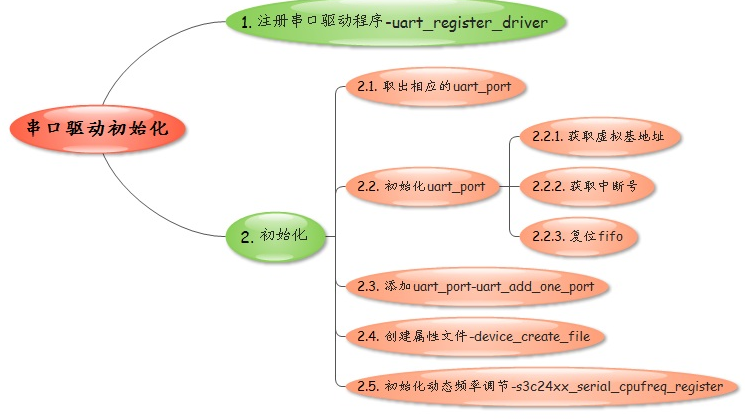
--reset both fifos

-uart\_add\_one\_port添加端口到uart

-platform\_set\_drvdata(dev, &ourport->port);//把port放到平台设备下

-device\_create\_file, 在/sys/下创建属性文件

-s3c24xx\_serial\_cpufreq\_register， 设置串口频率



\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

打开设备

-应用如何调用到驱动的xxx\_open

-xxx\_open的实现

应用如何调用到驱动的xxx\_open

-cdev\_init(&driver->cdev, &tty\_fops);//这体现了串口是个字符设备，重点是tty\_fops

-tty\_register\_driver

-uart\_register\_driver

-s3c24xx\_serial\_modinit

Open函数 ：

-s3c24xx\_serial\_startup, 属于串口驱动

-uart\_startup，这里调用uport->ops->startup(uport); 属于tty子系统

-uart\_open, 属于tty子系统

-tty->ops->open(tty, filp);//这个ops实际上是uart\_ops, 类型是tty\_operations，属于tty子系统

-tty\_open, 属于tty子系统

s3c24xx\_serial\_startup实现串口打开:

-rx\_enabled(port)， 使能串口接收

-request\_irq(ourport->rx\_irq, s3c24xx\_serial\_rx\_char…)， 注册接收数据中断处理程序

-tx\_enabled(port)， 使能串口发送

-request\_irq(ourport->tx\_irq, s3c24xx\_serial\_tx\_char…)， 注册发送数据中断处理程序

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

串口驱动分析-发送

TTY数据发送调用关系

-s3c24xx\_serial\_start\_tx

-tty(file->private\_data)->tty)->uart\_ops(**tty\_operations**, s3c24xx\_serial\_ops)->uart\_write, 数据写到xmit, 这里的tty是什么东西?

-do\_tty\_write(ld->ops->write, tty, file, buf, count);线路规程的ops(**tty\_ldisc\_ops** tty\_ldisc\_N\_TTY)->n\_tty\_write

-file(**file\_operations**)->f\_op->write, 实际上是tty\_cdev->tty\_fops->tty\_write, tty\_cdev是一个cdev，对应"/dev/tty", 这层属于tty core driver

-vfs\_write

-应用层的write

其实就是：

-file\_operations.write, file文件的tty\_fops.tty\_write

-tty\_ldisc\_ops.write, 线路规程的tty\_ldisc\_N\_TTY.n\_tty\_write

-tty\_operations.write, uart\_ops.uart\_write， 这个uart\_ops是一个变量，下面的uart\_ops是结构

-port->uart\_ops.start\_tx, s3c24xx\_serial\_ops.s3c24xx\_serial\_start\_tx, 这里才是真正驱动

串口发送函数s3c24xx\_serial\_start\_tx

-enable\_irq(ourport->tx\_irq), 打开中断后，因为txfifo里面数据为空，则产生发送中断

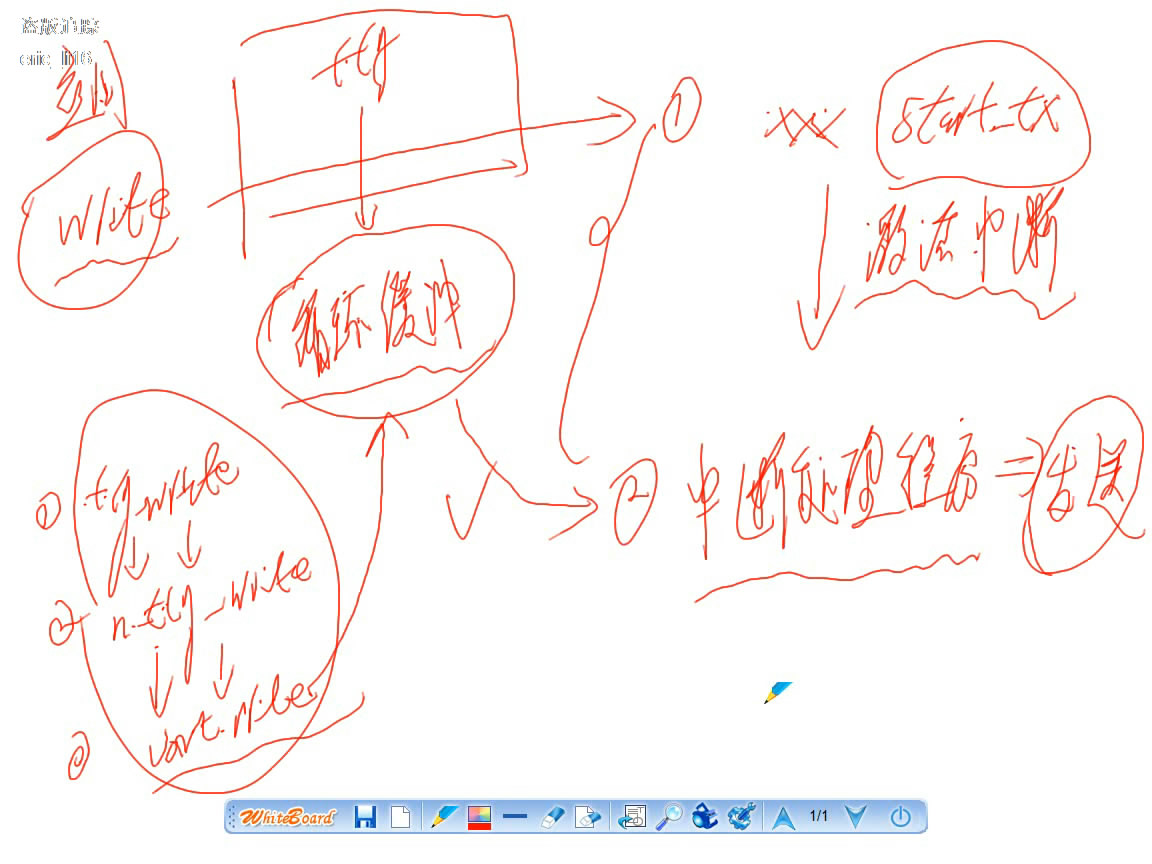
s3c24xx\_serial\_tx\_chars中断中

-数据从port的xmit循环缓冲取出来,这个xmit是由uart\_write里面写进去的。

-填充数据到数据寄存器进行发送

学习这些标准性的东西，例如tty子系统，各种外设信号协议，才有价值，虽然你觉得死板。熟悉随便定义的东西，灵活性大，但是没什么价值，就像我们camera和生产FCT的接口，可随便灵活定义，但懂这些东西，

没什么价值。



\*\*\*\*\*\*\*\*\*

串口发送函数分析

-s3c24xx\_serial\_start\_tx, 使能发送，打开tx中断

-s3c24xx\_serial\_tx\_chars

--判断是否有需要发送x\_char(起始、结束)字符, 如果有则将其写入UTXH寄存器进行发送

--如果当前循环bufer为空，就调用s3c24xx\_serial\_stop\_tx关掉发送中断，不允许发送

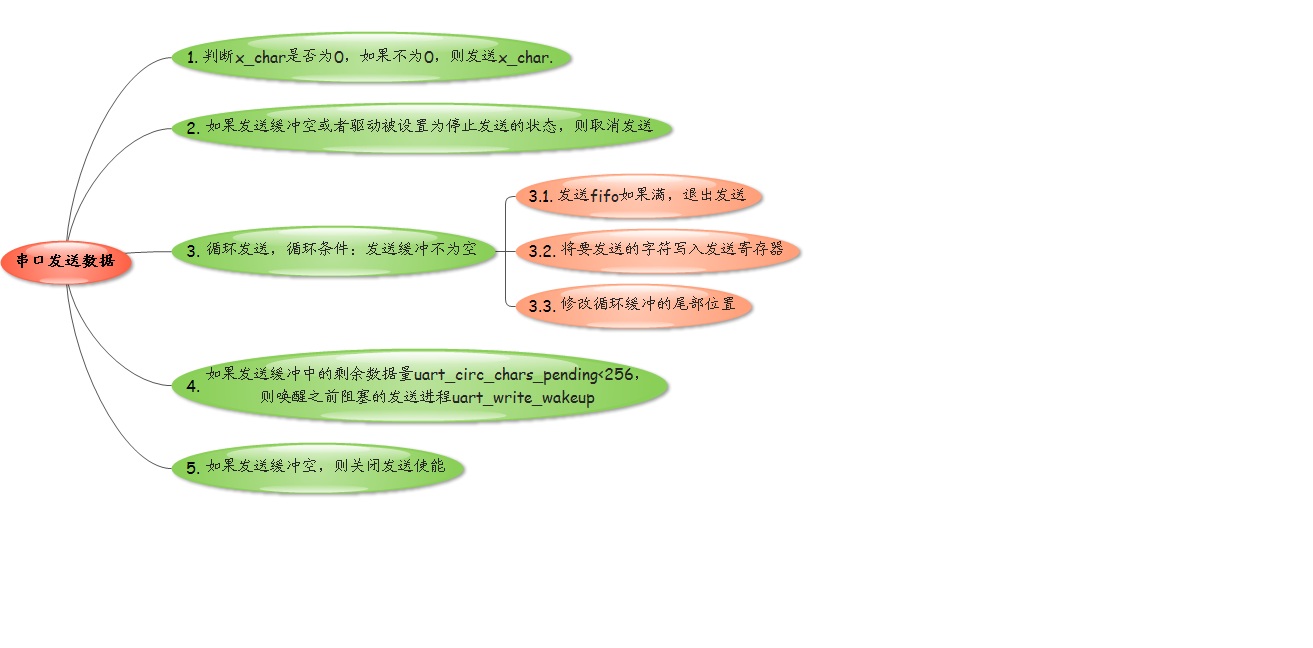
--利用while循环从循环buffer中取出字符，一次发送的数据量限制在256字节内

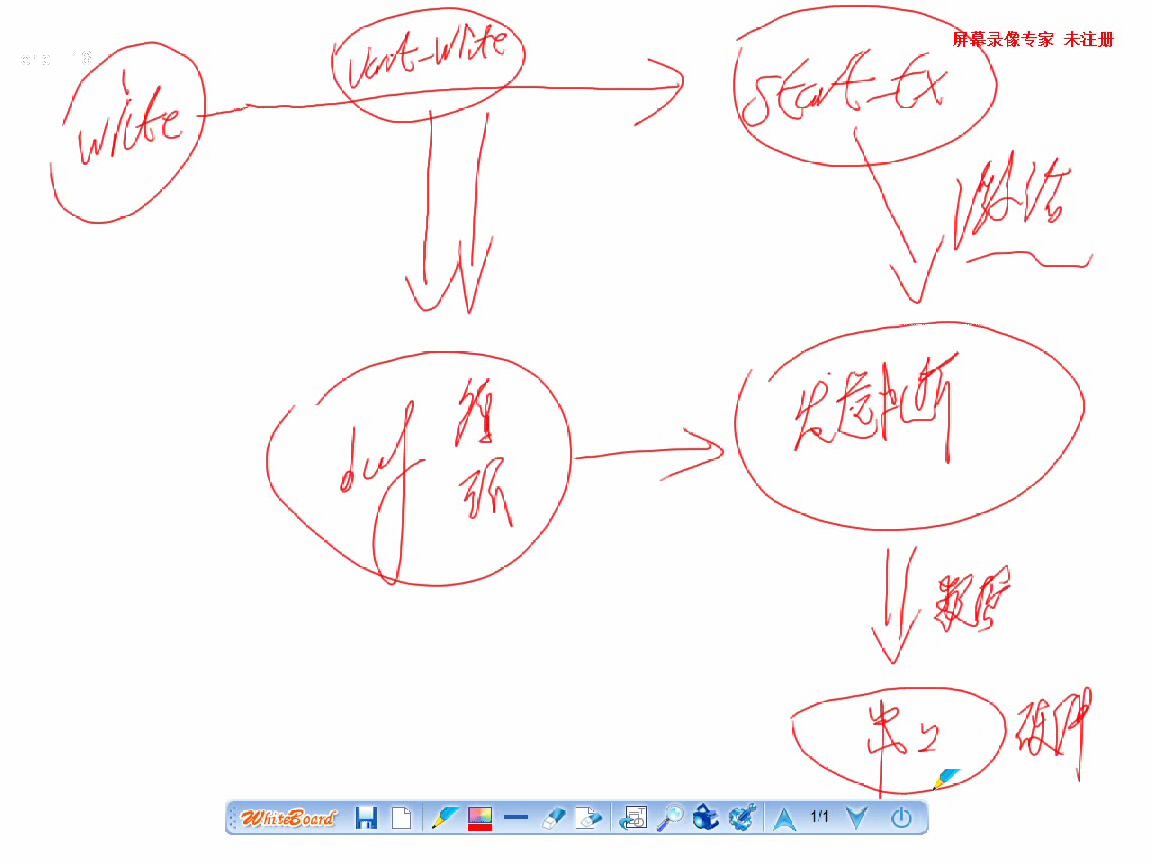
---读取txfifo状态寄存器看是否满，如果txfifo已满，就退出

---把循环缓冲的一个字符写入UTXH寄存器发送，并调整循环缓冲的头尾位置

--如果循环缓冲字符< 256字节,就是有空间让应用进程写数据进来，所以调用uart\_write\_wakeup唤醒进程

--发送循环缓冲为空，就调用s3c24xx\_serial\_stop\_tx关掉发送中断，停止发送





\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

串口驱动数据接收分析

-TTY数据接收流程分析

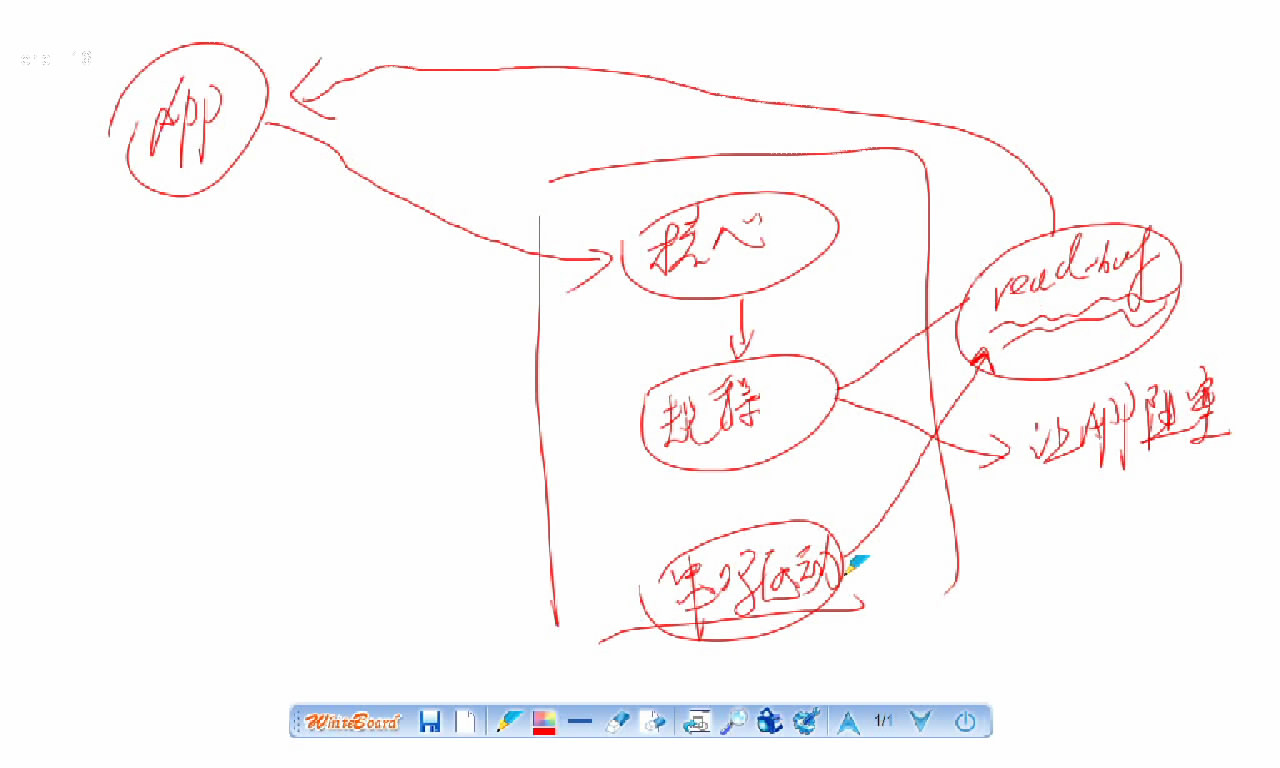
-串口驱动接收分析

-串口流控

TTY子系统数据接收流程分析

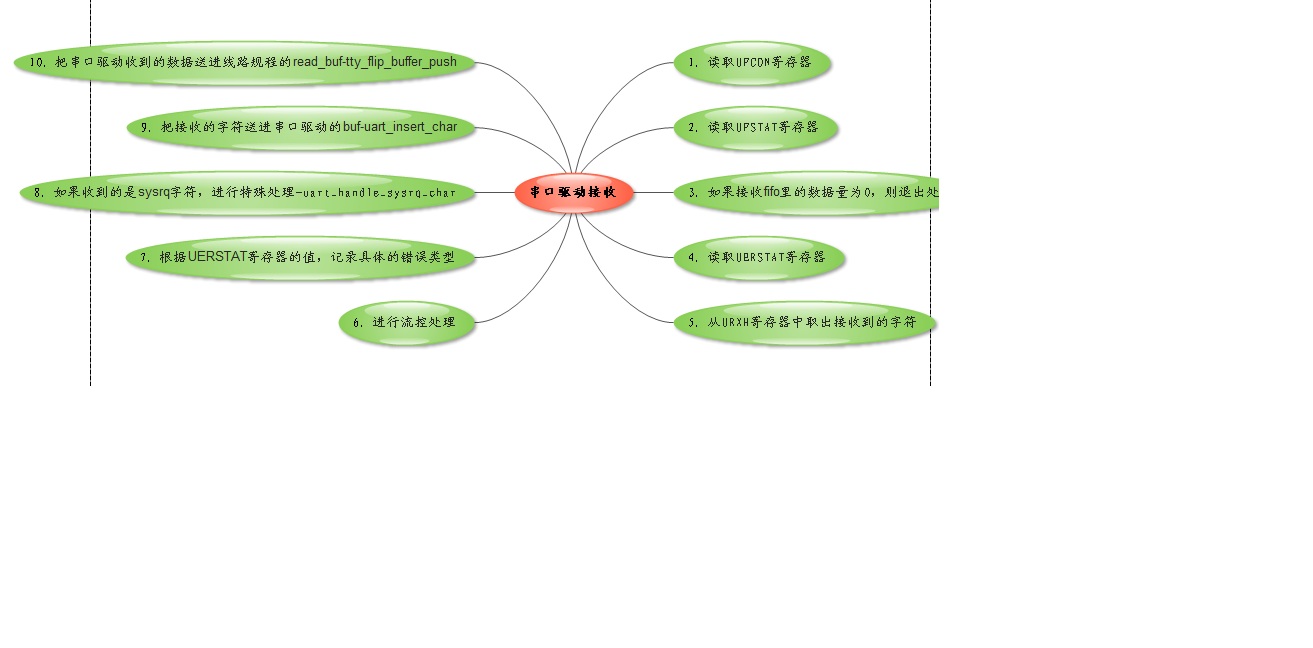
-n\_tty-read, 线路规程， 这里会调用copy\_from\_read\_buf, rx中断会把收到的数据填充给readbuf

-tty\_read，文件file操作函数, 属于tty核心



s3c24xx\_serial\_rx\_chars 中断接收函数

-int rx ch -> tty\_bufhead 串口驱动buf->线路规程的buffer

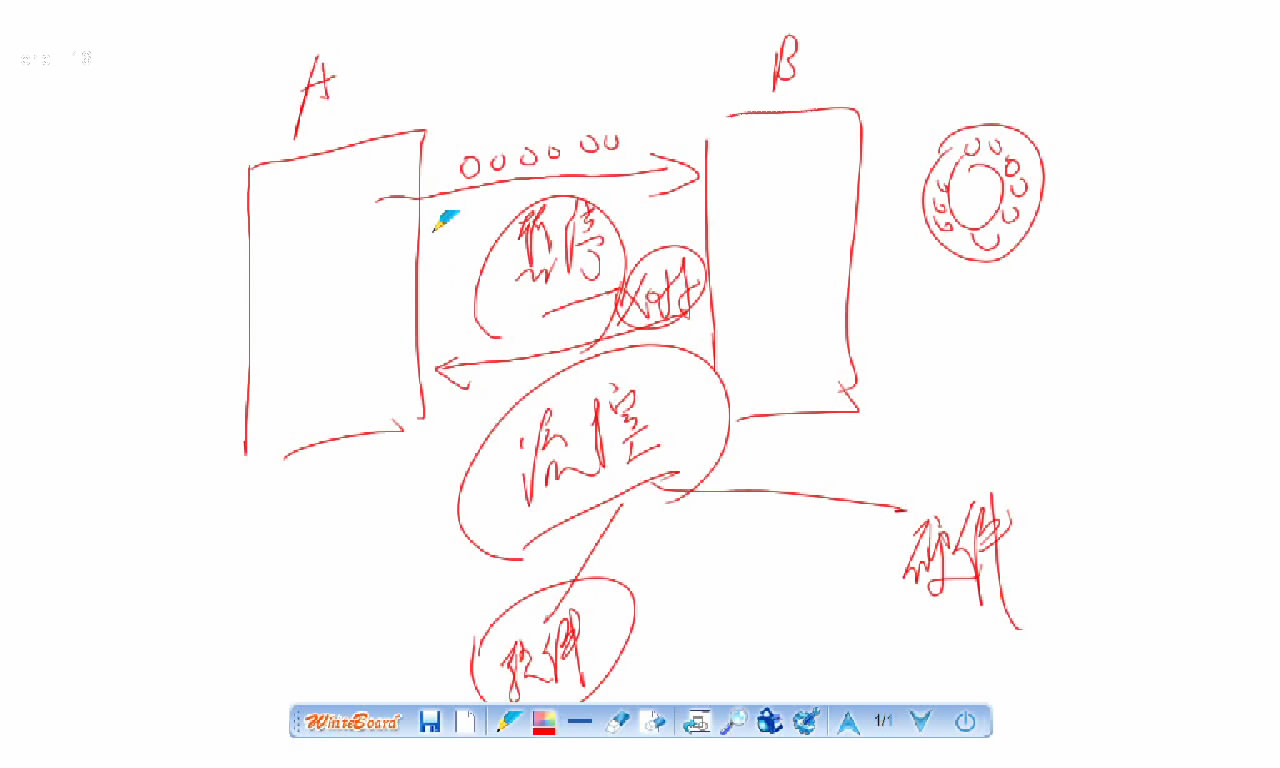


串口流控-接收方可以做

-软件流控, 通过xon，xoff

-硬件流控, 用得更多一些

软件流控



硬件流控

任何一方可以设置RTS(request to send)为高电平来告知对方自己是否可以接收数据，

任何一方可以检测CTS(clear to send)是否为高电平来判断对方是否可以接收数据。

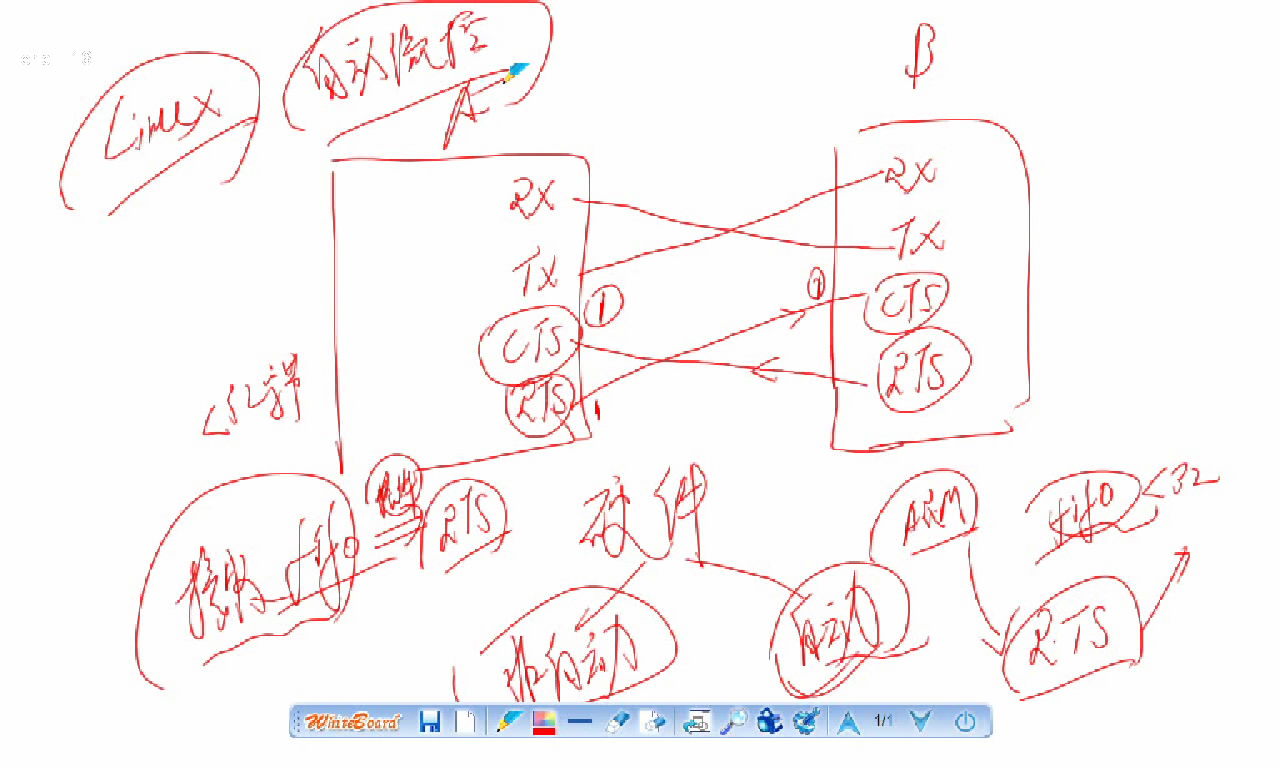
如果A想给B发送数据，发送前检测CTS是否为1，如果为1，则给B发送数据。

任何一方只能设置RTS,检测CTS。

硬件非自动流控: 如果RTS由软件来设置，则软件要根据接收fifo是否满来设置RTS。

硬件自动流控：如果rxfifo< 32， 则ARM自动的设置RTS为1，允许继续接收，不需要软件干预

Linux系统主要支持的是硬件自动流控



串口驱动中对流控的处理, UPF\_CONS\_FLOW

-停止发送的时候，立刻使能接收

- 启动发送的时候，立刻禁止接收

-在rx中断中，如果txfifo为空，则继续使能接收，使能接收失败，则重启接收部分

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

串口驱动程序实现

-读得懂

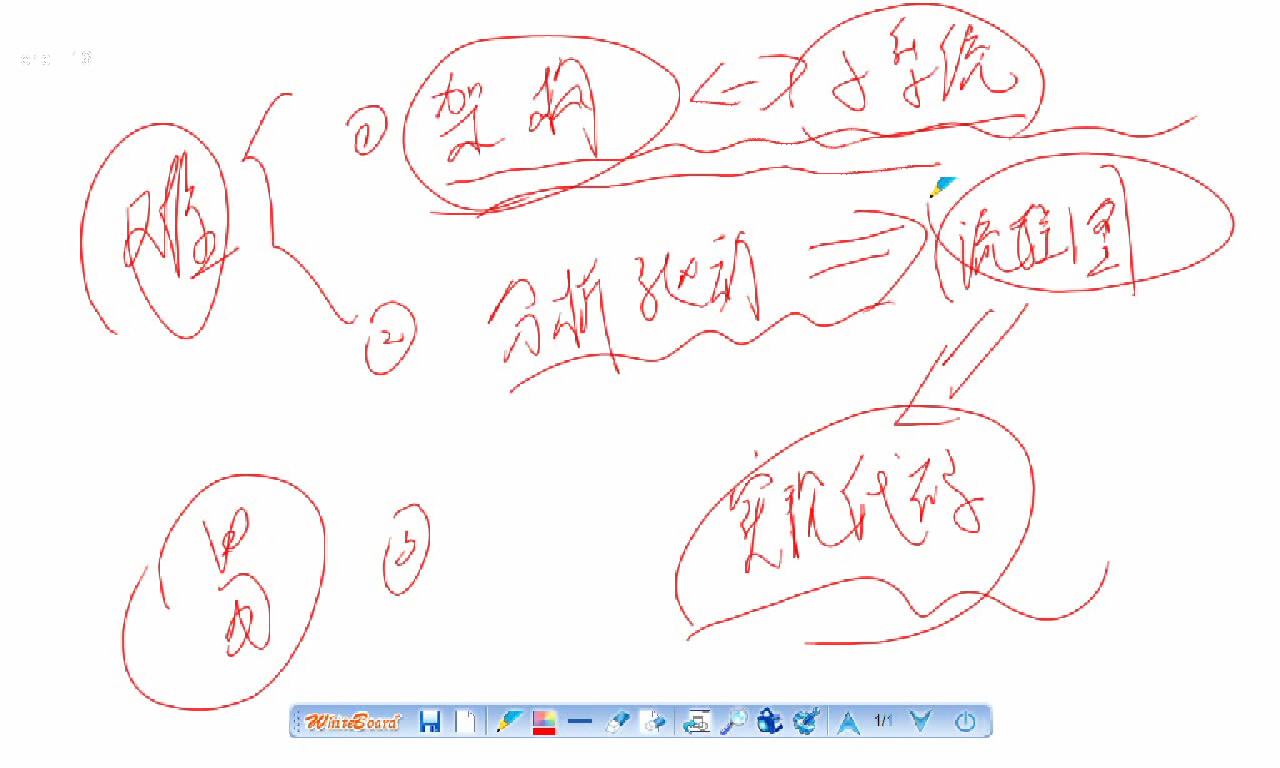
-懂得核心功能

-才能改

发送中断处理程序

接收中断处理程序

这个驱动程序是分水岭，驱动程序变复杂

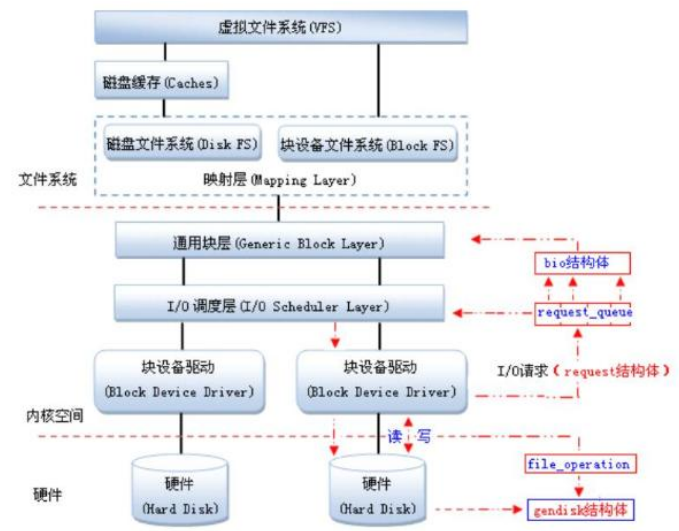


\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

块设备系统架构



块设备和光盘很类似，只是不需要格式化。



块设备驱动程序的分析

simple-blk\_read.c

简单块设备驱动程序的设计

把simple-blk\_read.c写一遍

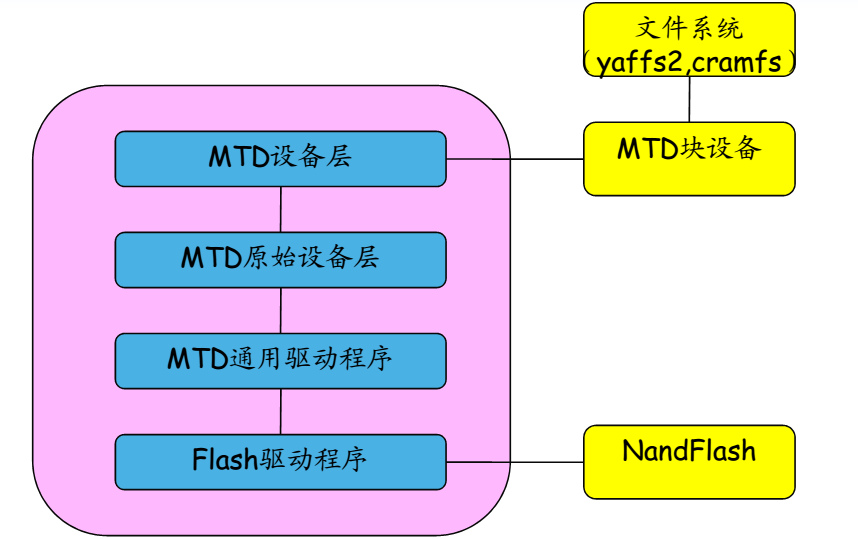
MTD系统架构

MTD- Memory Techknowledge Device, 为nor flash, nand flash提供统一接口，

从而使得flash驱动的设计大为简单

Cat /proc/mtd, 每个分区对应一个块设备

块设备驱动系统架构



MTD实际上是提取通用的东西到通用驱动程序，和硬件强相关的部分

才要写到flash驱动程序里。

Yaffs2文件系统应用

-MTD分区，由linux内核来实现

--Make menuconfig ARCH=arm

--Memory Technology Device (MTD) support

--MTD partitioning support

分成多少个区，怎么分区, 查找mach-real6410.c中的mtd\_partition

Nand flash分区:

--bootloader(uboot)

--linux内核,经过修改的内容，包括如何分区

--yaffs2, 在rootfs目录下，需要格式化成yaffs2文件系统，利用工具mkyaffs2image

Mkyaffs2image ./rootfs/ rootfs.img, yaffs2文件系统做好了

Uboot 参数调整，由环境变量来决定：

Uboot启动后没有任何操作，则会运行bootcmd命令来启动内核。

#define CONFIG\_BOOTCOMMAND “nand read xxxxx;bootm xxxxx”

Linux内核启动参数: 挂在根文件系统

#define CONFIG\_BOOTARGS “noinitrd root=/dev/mtdblock2” 用于指定文件系统，这里包含了文件系统的地址和长度

如何在uboot启动后写linux内核/文件系统到内核

-tftp

-usbslave, usbslave 1 300000, 等待，然后在pc 端用dnw来传送

-按照mtd的分区，使用nand erase把分区格式化, nand write把uboot/kernel/fs写入nand flash

Nand erase 0x50000, 不跟长度的话，后面的部分都会被格式化

Nand write .yaffs 30000000 500000 len, 这个len可以在pc上看长度，不能以传送的长度来填，因为不准

重新启动板子后

-uboot起来后会根据CONFIG\_BOOTCOMMAND 把linux读到ram，然后跳转到linux内核

-linux内核会根据CONFIG\_BOOTARGS 挂在文件系统

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Nand flash 驱动程序设计

-初始化

-读写操作

E:\code\_read\linux-ok6410\drivers\mtd\nand\s3c\_nand.c

s3c6410\_nand\_probe:初始化

-使能时钟，get the clock source and enable it

-进行物理地址到虚拟地址的转换

-硬件纠错初始化

-搜索nand flash

-add\_mtd\_partitions: Register the partitions

nandflash读操作:硬件操作部分如何融入linux驱动

通用驱动: nand\_base.c

-nand\_read->nand\_do\_read\_ops->cmd\_func(nand\_command\_lp)

->cmd\_ctrl (s3c\_nand\_hwcontrol)

学习linux有两大块

-linux的模型，子系统

-硬件操作

如何在linux驱动中把硬件操作部分找出来，是重点和难点。

-先熟悉硬件操作

-在熟悉linux驱动模型，把硬件操作和模型结合起来即可

发送开始命令，列地址，行地址，0x30这些都是操作nand通用操作，所以已经写在通用驱动里面

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

I2C总线介绍

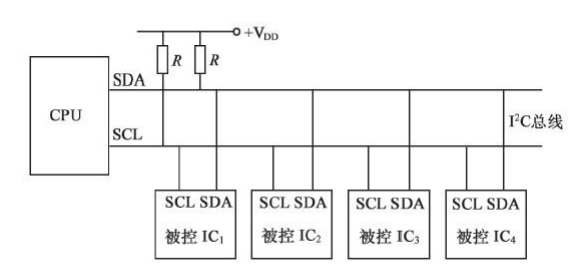
-I2C硬件结构

--电器特性

--总线寻址

-总线时序

飞利浦公司发明



主从设备



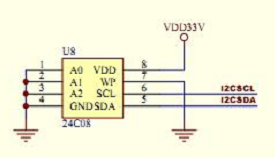
I2C总线协议规定：

主设备发送地址时，总线上每个从设备都将这7位地址和自己地址进行比较，

如果相同，则认为自己正被主设备寻址，根据R/W位将自己确定为发送器还是接收器。

设备地址由固定部分和用户自定义部分组成:

1. 固定部分：D7~D4， 由从设备的生产商生产时就已经确定。
2. 用户自定义部分: D3~D1, 这3位通常对应设备的3个引脚（A0~A2）.



\*总线时序

所有数据以8位来传输的。

空闲状态：SDA,SCL同时处于高电平

启始状态: SCL为高，SDA从高变低

结束状态: SCL为高， SDA从低变高

数据：地址和数据

应答：第9bit，低电平为ACK，高电平为NAK

应答位:

发送器每发送一个字节，就在时钟脉冲第9位释放数据线，

由接收器反馈一个应答信号。应答信号为低电平（ACK），表示接收器已经成功接收了该字节；

如果应答信号为高电平（NAK），表示接收器没有成功接收该字节。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

I2C裸机驱动设计

-EEPROM简介

-I2C初始化

-写数据设计

-读数据设计

-EEPROM驱动综合测试

\*EEPROM简介

EEPROM(Electrically Erasable Programmable Read-Only Memory), 电可擦可编程只读存储器。

和flash相比的区别:

1. EEPROM可以按位擦鞋，而flash按block擦除
2. EEPROM一般容量不大，一般都在64Kbit一下，一般是256字节，512字节。

EEPROM的自定义地址，可以从原理图中获得。前4bit可以通过EEPROM手册可以查到。

\*I2C初始化

The following steps must be executed before any IIC Tx/Rx operations.

1. Write own slave address on IICADD register, if needed.

2. Set IICCON register.

a) Enable interrupt

b) Define SCL period

3. Set IICSTAT to enable Serial Output

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

SPI总线介绍

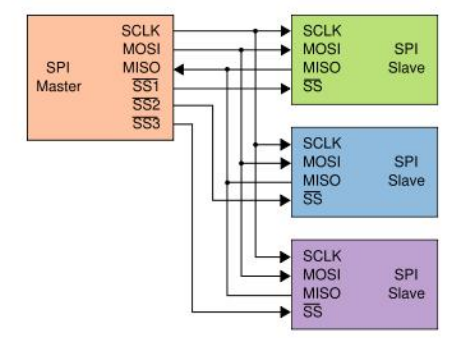
-总线结构

-数据传输过程

-总线时序

总线结构

SPI(Serial Peripheral Interface) 串行外设接口。采用主从模式。



SPI有4根信号线：

/SS, 片选信号线

SCK, 时钟信号线

MOSI, master output slave input

MISO, master input slave output

数据传输过程:

全双工，交换

总线时序: CPOL, CPHA

CPOL极性, 决定时钟空闲时为高电平还是低电平

CPOL=0: CLK空闲时是低电平，CLK有效时是高电平

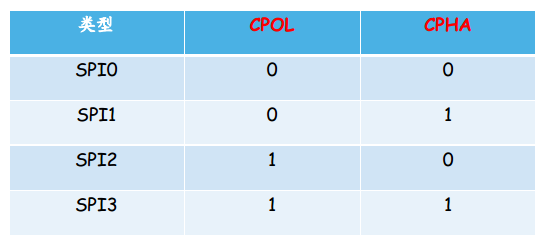
CPOL=1: CLK空闲时是高电平，CLK有效时是低电平

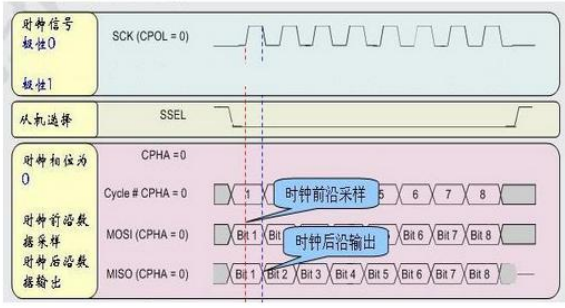
CPHA相位: 决定何时进行数据采样(读取)

CPHA=0: 第一边沿采样

CPHA=1: 第二边沿采样

CPOL和CPHA的组合就形成4种模式:





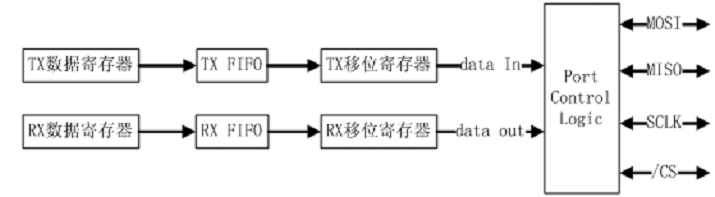
CPHA=0, 第一个沿采集，第二个沿数据转换

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

SPI裸机驱动设计

-SPI控制器工作流程

-6410驱动程序分析



读写操作的就是这两个寄存器，tx和rx寄存器。

6410驱动程序分析

-初始化，芯片手册有说setting sequence

--SPI软复位

--设置引脚，引脚一般都是复用的

--配置CFG寄存器，设置工作模式和主从

--配置MODE\_cfg0， 可以设置DMA模式，也就是DMA代替CPU直接从内存取值到寄存器，或从寄存器取值到内存

--设置时钟

可以设置和查询的东西包括:

-DMA方式

-中断方式，满或少于多少产生中断

-TX RX fifo的宽度

-移位寄存器的宽度

-可查询当前FIFO中有多少数据

-发送， 当tx fifo里面少于多少数据的时候，产生发送中断通知CPU可以继续发送

--查询发送fifo有空位，就可以写新的数据到tx寄存器

-接收， 当rx fifo里面多余多少数据的时候，产生接收中断通知CPU可以取数据了

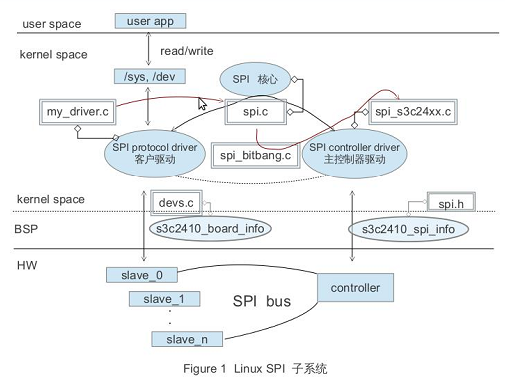
--查询接收fifo有数据，就可以从新rx寄存器取出来

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

SPI系统架构

-SPI子系统模型

-SPI控制器驱动分析



内核代码不可能每行都读懂，但是要了解关键点，以及和硬件操作相关的部分。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Linux SPI驱动设计

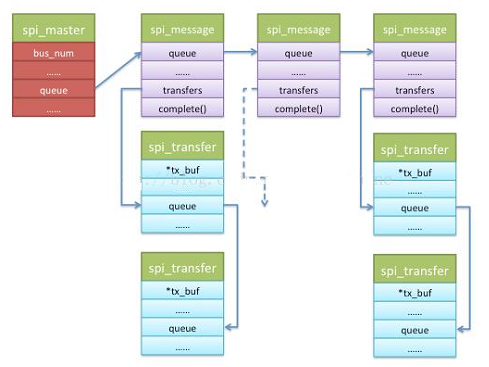
M25p80.c

M25p80\_write

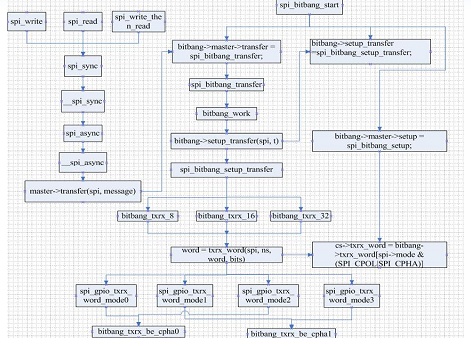
Spi\_message, 每个spi\_message 是spi控制器每次要完成的事物

Spi\_transfer, 有可能是数据，有可能是命令，可以挂多个在一个Spi\_message下

Spi\_sync, 把Spi\_message提交给控制器，控制器在合适的时候把东西发送出去



SPI控制器怎么处理呢？



\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

USB 驱动

full speed: USB 1.1 12Mb/s

high speed: USB 2.0 480Mb/s

super speed: USB 3.0 十倍于USB 2.0

标准接口:

标准A口:最常用的USB口

标准B口:仿真器的usb接口

mini-usb: 比micro-usb大一点

micro-usb: android手机充电口

信号线

4根信号线

-USB电源，5V，红

-地线，黑色

-数据线D+

-数据线D-

当两者电压差大于200mv时代表1，小于200mv代表0

USB拓扑结构

主机->根集线器->设备

->集线器hub

USB主控制器规格有:

-OHCI, 非PC系统，USB1.1, 嵌入式系统

-UHCI, pc usb1.1

-EHCI, USB2.0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

USB协议

\*USB设备逻辑结构

假如一个设备有音频，视频，按钮功能

配置，其实就是功能的组合

接口，对应一个功能

端点，寄存器，操作寄存器让接口工作起来

为什么电脑能认出是鼠标而不是U盘呢？

-是因为设备内部固定格式的数据

USB的描述符：

设备描述符，配置描述符，接口描述符，端点描述符，字符串描述符

设备描述符 USB specification 2.0

\*USB数据通信

传输->事物->包->域

传输(transfer)种类:

中断传输，批量传输，同步传输，控制传输

事务(transaction)种类:

-In事务

-Out事务

-Setup事务

包(packet)种类:

-令牌包(setup)

-数据包(data)

-握手包(ack)

-特殊包

一个包由多个域构成:

同步域(SYNC), 标识域(PID),地址域(ADDR)，端点域(ENDP)，帧号域(FRAM)

数据域(DATA)，校验域(CRC)

\*USB设备枚举过程, 可以用Lecroy来查看抓到的包

-获取设备描述符

-复位

-设置地址

-再次获取设备描述符

-获取配置描述符

-获取接口、端点描述符

-获取字符串描述符