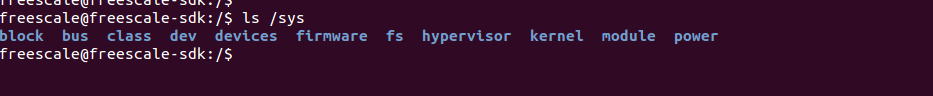
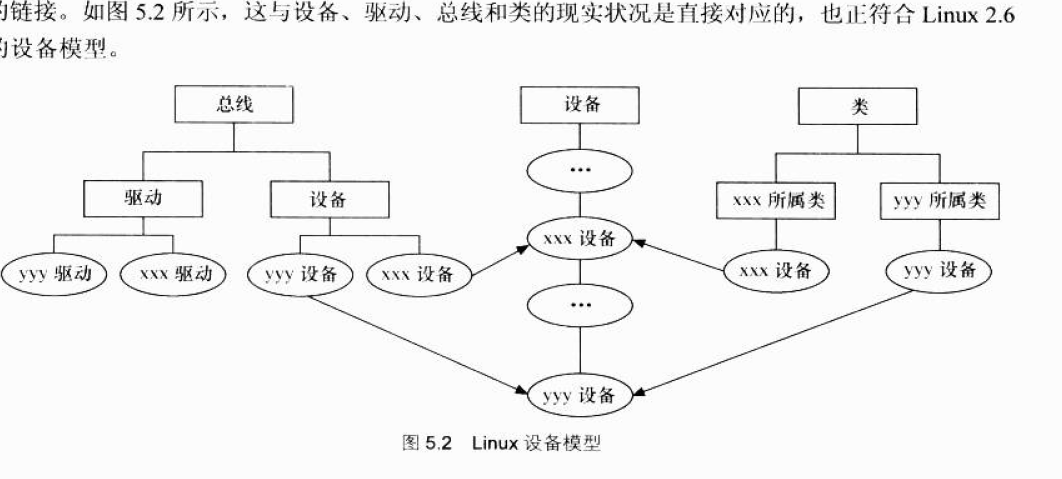
* 1. Linux系统移植架构
     1. Linux操作系统的熟悉
     2. 开发环境搭建
        1. 虚拟机安装
        2. 交叉编译工具链
        3. 超级中断和minicom(串口工具)
        4. 内核、文件系统加载工具
        5. 网络文件系统（NFS）
     3. Bootloader移植
     4. Linux内核裁剪与移植
     5. 嵌入式文件系统制作（一般为Flash文件系统）
     6. 驱动移植
        1. I2C
        2. PCI
        3. USB
        4. NAND Flash and nor Flash
        5. LCD
        6. 待定
     7. 系统移植高级篇
        1. MiniGUI
        2. QT
        3. KVM
        4. Web server
        5. 数据库
        6. 待定
  2. Linux设备驱动
     1. Sys伪文件系统

Sys文件系统挂载于/目录下，以文件目录的形式呈现。



各级目录的关系如下图：



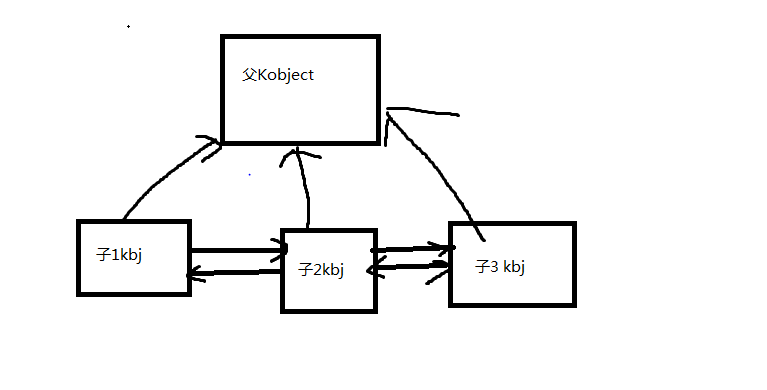
* + 1. Proc伪文件系统

Proc文件系统与传统保存在磁盘上的文件系统不同，它是存在于内存中的文件系统，里面包含内核的进程信息，CPU信息等等。

* + 1. udev的工作过程

当/sbin/hotplug程序被内核调用时，udev正式执行。具体执行步骤如下：

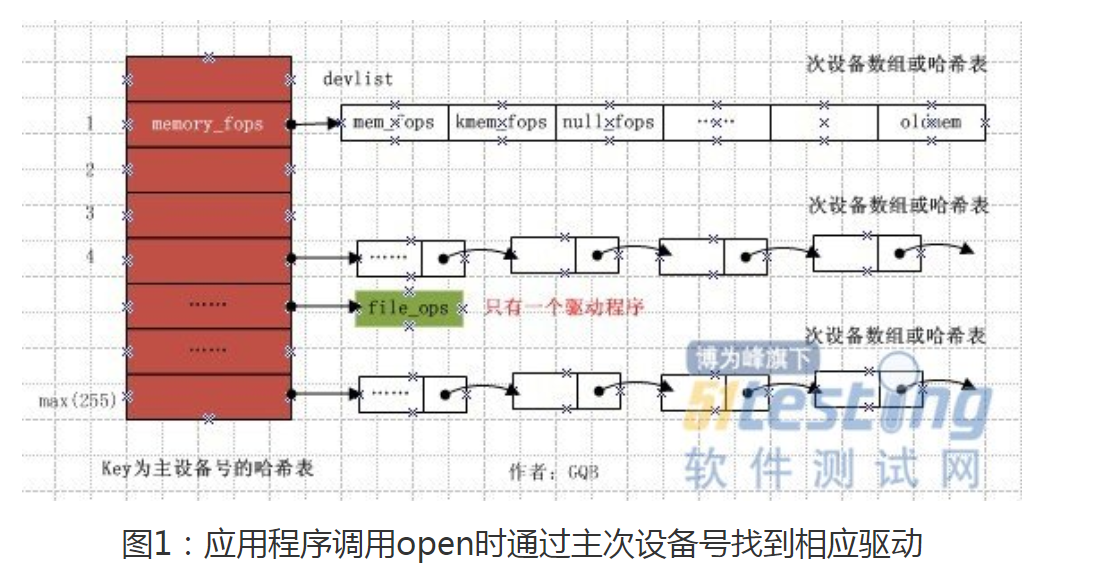
1. 内核检测到有新的设备注册后，将在sys文件系统中创建对应的节点，并写入该设备的一些信息；
2. 接着，udev获取内核导出的信息，进行对应的操作：
3. 首先，调用namedev为新创建的设备命名；
4. 如果该设备的动作是插入，那么udev将调用libsys库的接口从sys文件系统中读取设备的信息，用以在/dev下创建设备，如果设备的动作是删除，那么udev就直接调用移除函数，将设备从/dev下删除
   * 1. Kobject树形结构



每个kobject对应一个sys目录，kobject中存在struct list\_head成员，通过list\_head，各个父、子kobject形成一个双向列表。

* + 1. 用户空间的设备与内核空间的设备、驱动关系

主设备号找到对应的驱动，次设备号找到对应的设备。内核维护了一张关于主次设备号的哈希表，其中，主设备号作为哈希表的key，内核根据这个key找到对应的驱动程序指针（驱动程序指针数组或者列表），如果同一类设备存在多个驱动，则根据次设备号遍历驱动程序指针数组（或者列表）。

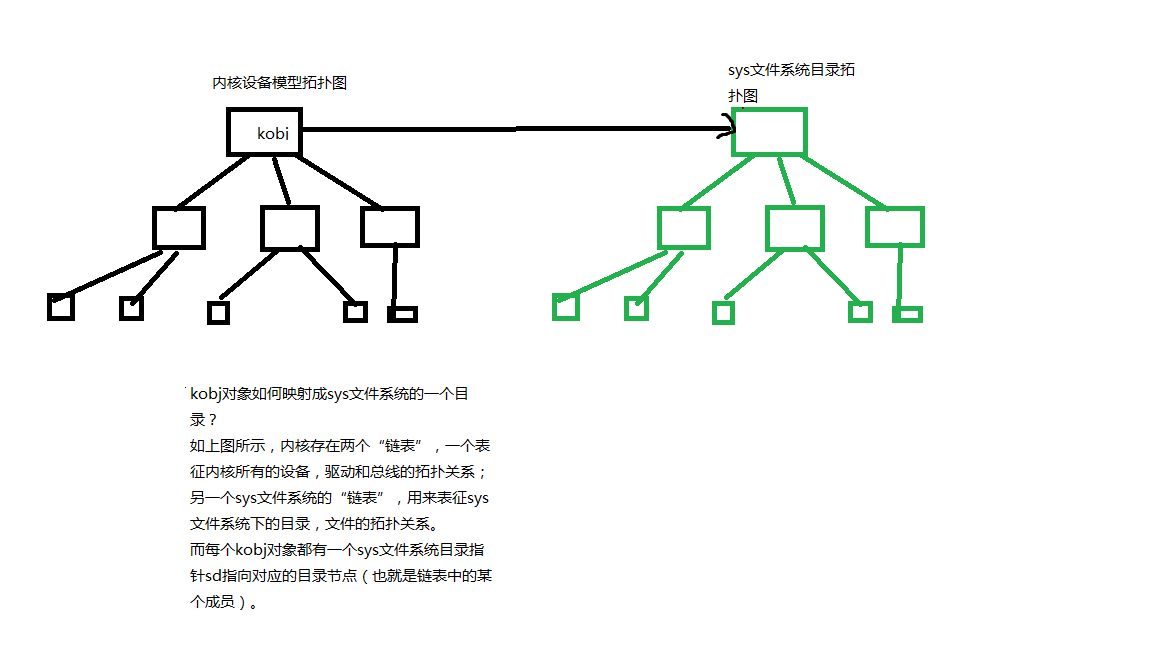


2017-10-25 11:14:17 关系修正：

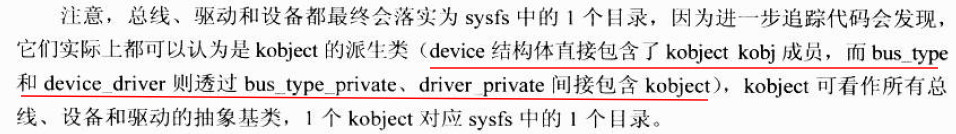
以字符设备为例，应用层open->创建file结构体->关联inode->根据主次设备号在kobj拓扑图中找到驱动的fops->调用驱动的open函数->返回应用层

所以，以key为主设备号的哈希表，实际上就是1.6中的kobj内核表。

* + 1. Kobj与sys文件系统映射关系



无论是总线，设备，还是驱动都是kobj的派生类，所以，它们均在sys文件系统中体现。（具体查看结构体：struct bus\_type, struct devices,struct devices\_driver）



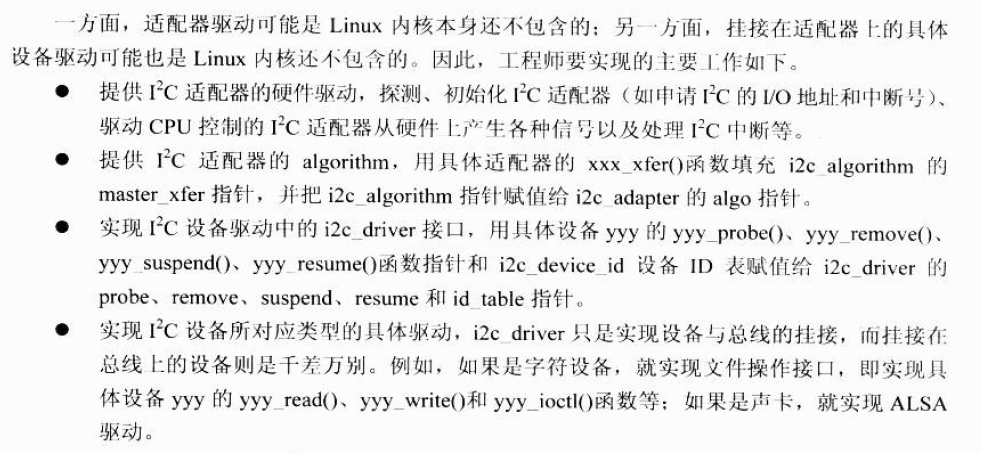
* + 1. 驱动注册，设备注册

驱动注册实际上就是将device\_driver插入内核的kobj列表上，更具体点，就是挂在xxx\_bus kobj列表下，设备的注册类似。

驱动和设备注册不分先后，但最终都会调用匹配函数进行相互绑定，而所谓绑定就是将驱动的结构体指针赋值给对应的设备，形如：

struct device\_driver \*driver = &xxx\_driver;

* + 1. 驱动开发所需要做的事情（举例：I2C）



* + 1. Platform驱动注册过程（platform\_probe调用xxx\_probe的过程）

platform\_driver\_register()->driver\_register()->bus\_add\_driver()->driver\_attach()->bus\_for\_each\_dev() 对每个挂在虚拟的platform bus的设备作 \_\_driver\_attach()->driver\_probe\_device()->drv->bus->match()==platform\_match()-& gt;比较strncmp(pdev->name, drv->name, BUS\_ID\_SIZE)，如果相符就调用platform\_drv\_probe()->driver->probe()，如果probe成 功则绑定该设备到该驱动

* + 1. 总线注册函数

int bus\_register(struct bus\_type \*bus)

例如：retval = bus\_register(&i2c\_bus\_type);

* + 1. BSP板级信息，设备，驱动

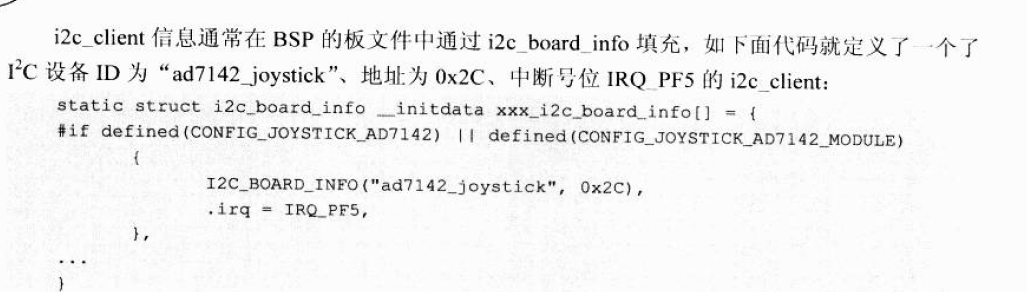
以mini2440的nandFlash板级信息为例。

mini2440\_init->s3c\_nand\_set\_platdata（&mini2440\_nand\_info）->platform\_add\_devices（mini2440\_devices）

mini2440\_nand\_info 、mini2440\_devices 定义于Match-mini2440.c中。

在板级文件Mach-mini2440.c中定义了关于2440 nandFlash的硬件信息，该硬件信息两部分组成，一部分以成员的形式存在（s3c\_device\_nand）与mini2440\_devices设备数组中，另外一部分以s3c2410\_platform\_nand结构体的信息存在s3c\_device\_nand的dev.platdata字段，该设备数组（mini2440\_devices）最后会被调用platform\_add\_devices完成设备注册。（PS:在nandflash驱动则通过platform\_data获取注册的硬件信息）

* + 1. I2C 子系统
       1. Client设备注册



再通过i2c\_scan\_static\_board\_info->i2c\_new\_devices进行设备注册,且注册后需要依附在一个adpter上。

Client需要的i2c\_board\_info：flag,irq,addr,type（名称），platform\_data

* + - 1. Adpater注册

Driver/bus/i2c/busses目录下提供的各个平台的I2C控制器驱动，都是以platform驱动的形式注册到内核列表中，然后注册完成后，再调用probe函数，在probe函数中完成adpater的驱动注册。

* + - 1. I2c-dev.c

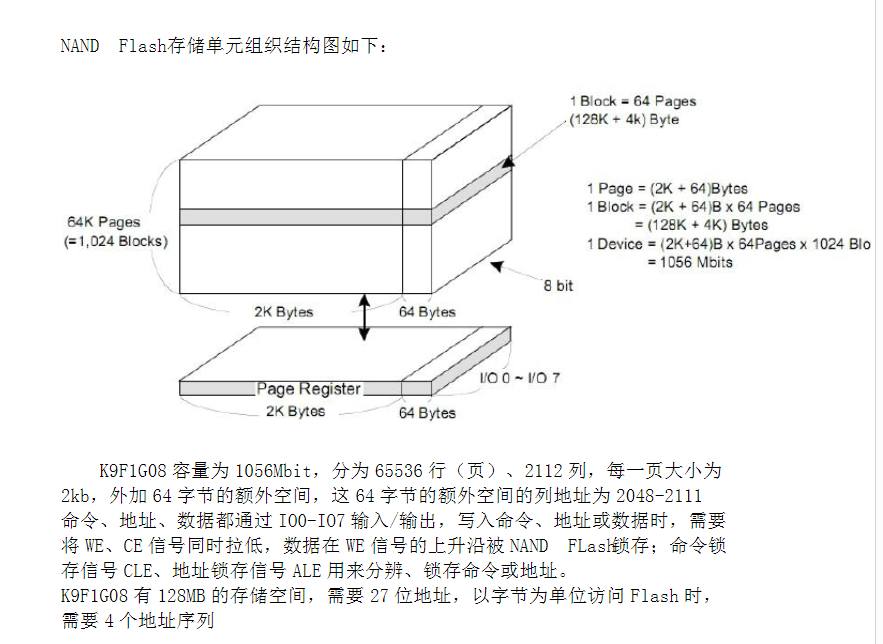
I2c-dev.c针对每个I2C控制器创建一个主设备号为89的设备。它的作用是为用户提供设备文件，以便用户以标准的方式操作I2C设备：open,close,read,write,ioctl。它和busess目录下注册的各种平台I2C控制器的关系建立，是通过i2c\_bus\_type完成的。

具体函数调用：

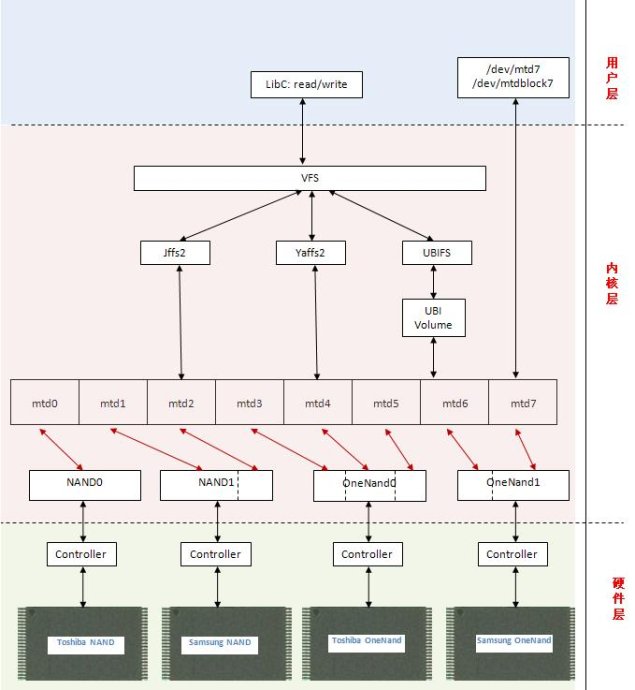
i2c\_dev\_init->i2c\_for\_each\_dev->bus\_for\_each\_dev

此处该函数传递的Bus\_type为i2c\_bus\_type，所以bus\_for\_each\_dev内核设备树时，就会从i2c\_bus\_type子树中找到对应的adpater。

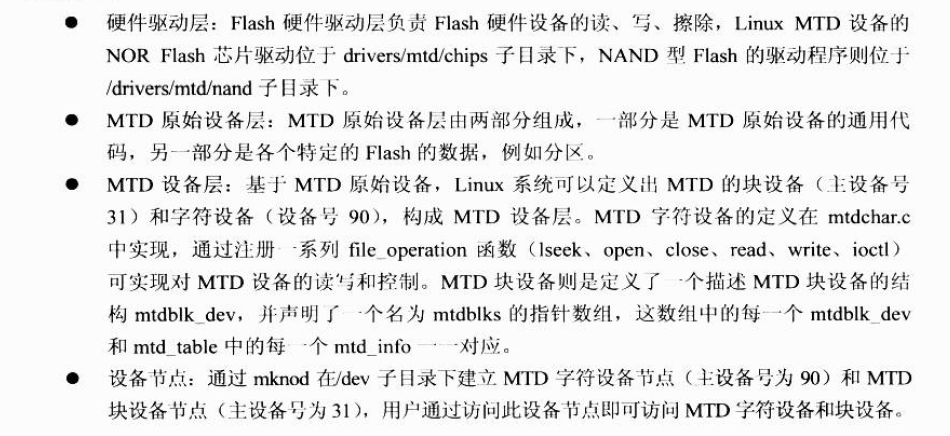
* + 1. MTD子系统
       1. NAND FLASH原理



* + - 1. 闪存，MTD及文件系统之间的关系



存储介质（硬盘，磁盘，Flash，U盘）以某种特定的格式（文件系统）保存着数据，驱动层提供不同的存储介质的读/写，控制等操作方法，而针对存储介质上的不同格式的文件系统，在操作系统中都一一提供了对应的软件抽象层（即操作系统中的文件系统），这些软件抽象层实现了不同格式的文件系统的数据组织和管理。



* + - 1. mtd\_info,mtd\_part,mtd\_partion关系

• mtd\_info描述原始设备层的一个分区的结构, 描述一个设备或一个多分区设备中的一个分区

• ~~mtd\_table 管理原始设备层的mtd\_info的数组~~

• mtd\_part表示一个分区, 其中的struct mtd\_info mtd描述该分区的信息， 一个物理Flash设备可以有多于1个mtd\_part，每个mtd\_part都对应一个mtd\_info。

• mtd\_partition描述一个分区表, 通过管理mtd\_part以及每一个mtd\_part中的mtd\_info来描述所有的分区，一个物理Flash设备只有一个mtd\_partition

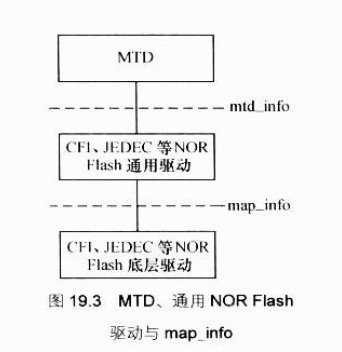
• mtd\_partitions是一个list\_head对象，用于管理mtd\_partition们

• map\_info描述一个NOR Flash设备

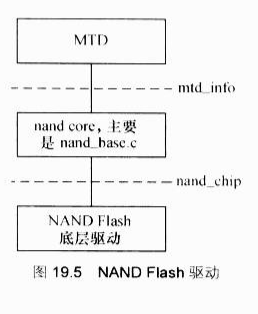
• nand\_chip描述一个NAND Flash设备

换言之，每个Flash存在一个分区表，分区表管理着所有的分区，而每个分区又对应着一个mtd设备。

* + - 1. Norflash驱动
         1. 层次结构

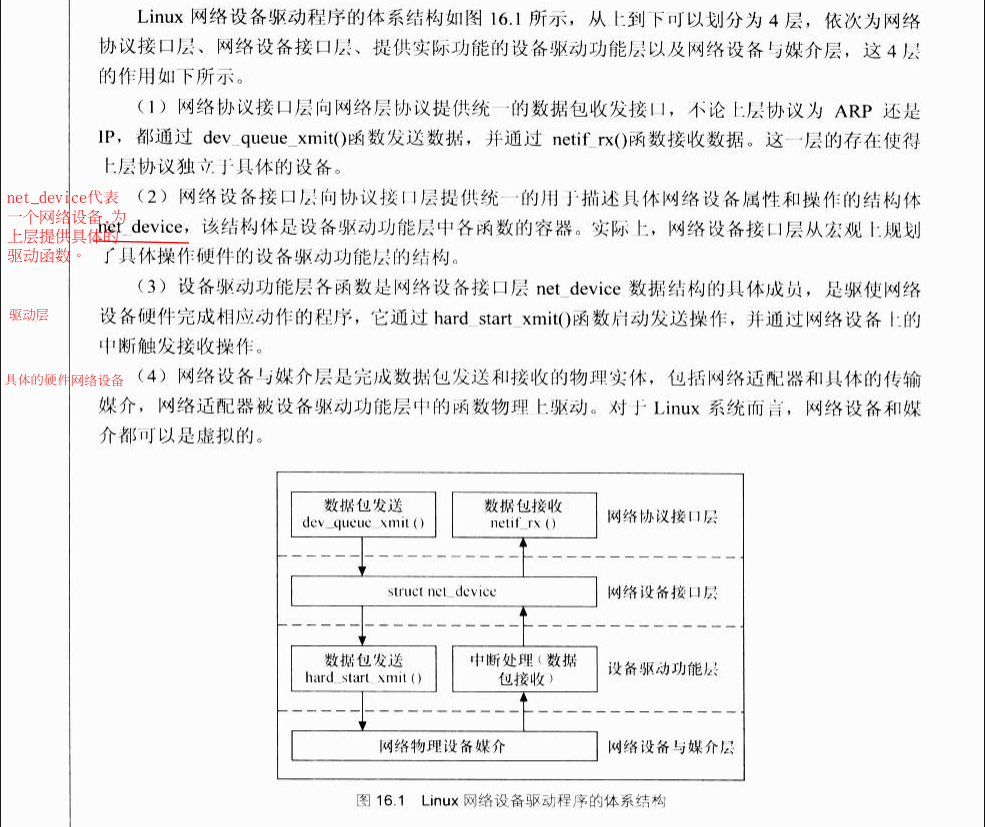


* + - 1. Nandflash驱动
         1. 层次结构



* + 1. 文件系统
       1. 概念

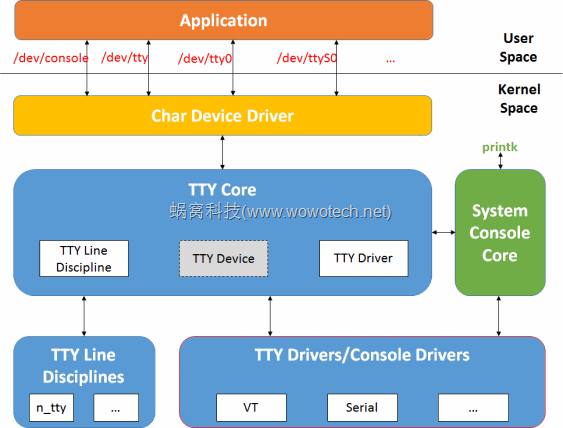
1. 一种特定的格式，控制着存储介质上数据的管理和组织形式，如果把U盘，硬盘或者NAND Flash格式化成某种文件系统；
2. 操作系统中一套管理“文件”的软件集合；
   * 1. 网络子系统
        1. 网络设备驱动



* + 1. 终端设备驱动(TTY)
       1. 概述

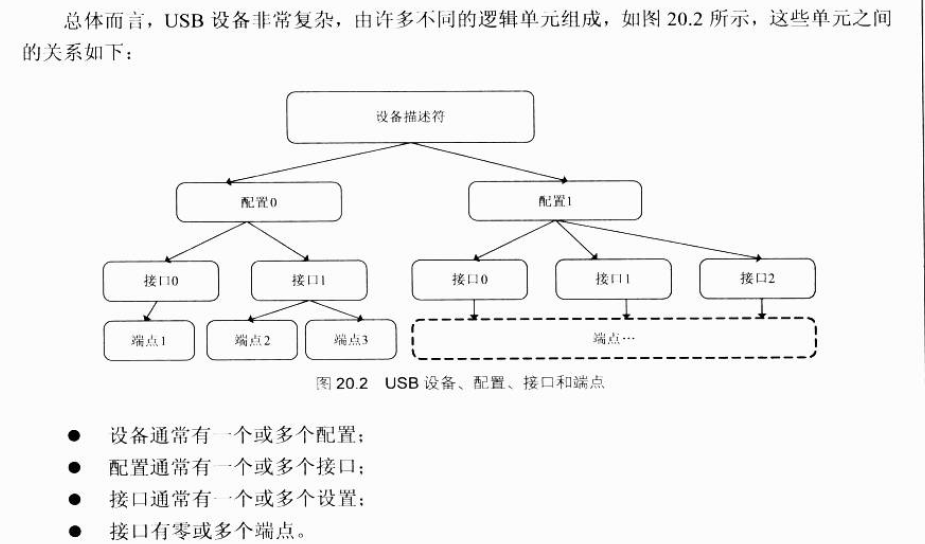
对于TTY系统的理解（图解）：

Linux tty framework框架：



Tty为底层驱动提供注册（注册为字符设备）的接口，自身不会注册成设备。

* + - 1. TTY子系统
      2. UART驱动
    1. USB子系统
       1. USB设备逻辑组织



* + 1. PCI总线驱动
    2. LCD驱动
       1. LCD显示原理

人眼识别颜色的原理：光，不同颜色的光被反射到眼睛中，如果没有光（没有光源或者所有光线被吸收），则看到的是黑色，如果所有颜色的光一起进入眼睛，看到的则是白色。

LCD原理利用人眼的感光原理，采用发光源+彩色滤光片（三原色）的形式构成，其基本原理是，由发光源发光，然后通过三原色的滤光片透过3种颜色的光。3种颜色的光组成一个像素点，通根据空间混叠原理（空间混叠：当三原色距离较近时，人眼会将其自动混合）和三原色叠加原理组合出不同的颜色。

* + - 1. LCD颜色变化与显示缓存值变化的关系

颜色变化的实质是灰度的变化，即亮度的变化，而在linux中，修改显存中对应的颜色值，实际上是改变了灰度的变化。

* + 1. Framebuffer驱动架构
       1. 应用层，fbmem(framebuffer核心层)，底层fbxx设备驱动关系

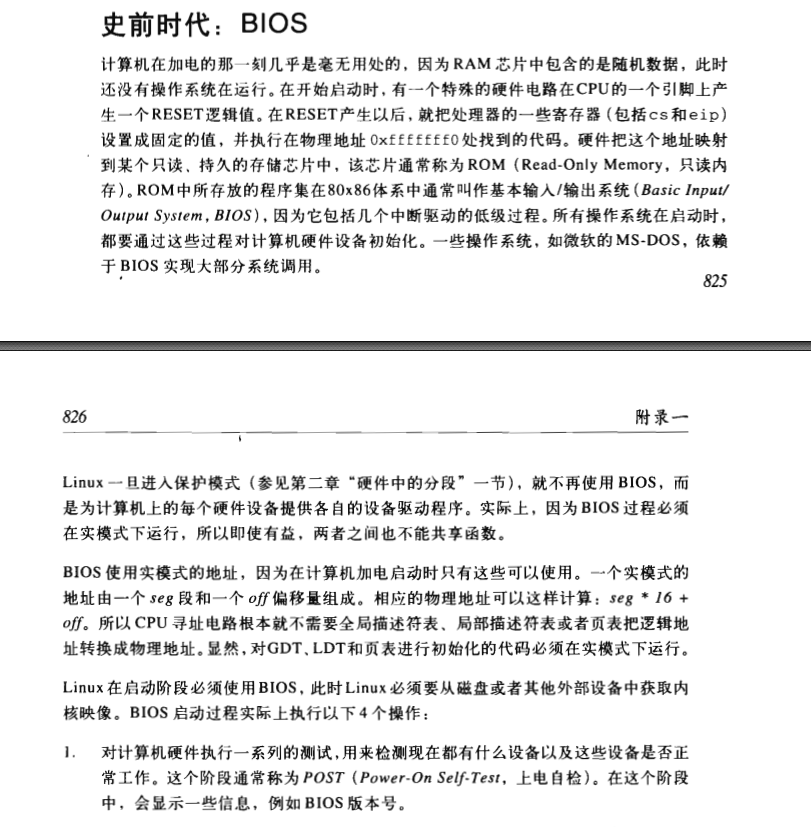
每个物理显示设备都用一个fb\_info表示，完成注册后这个fb\_info将会保存在全局数组fb中。而应用层访问具体的fb设备驱动流程（以open为例)：open->fb\_open->根据次设备号，从全局数组中获取设备的fb\_info->调用设备驱动的open，即fb\_xxxx\_open函数。

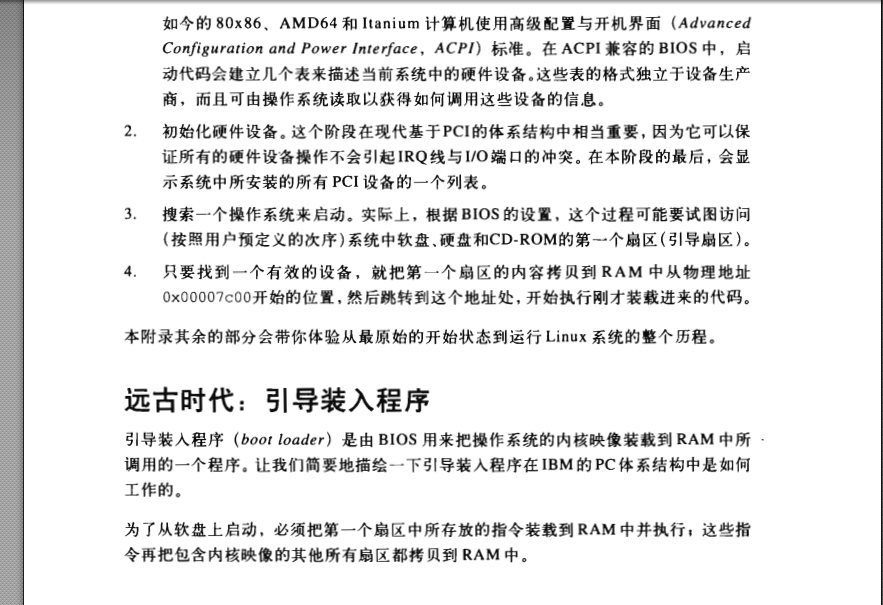
* + 1. Linux内核分析

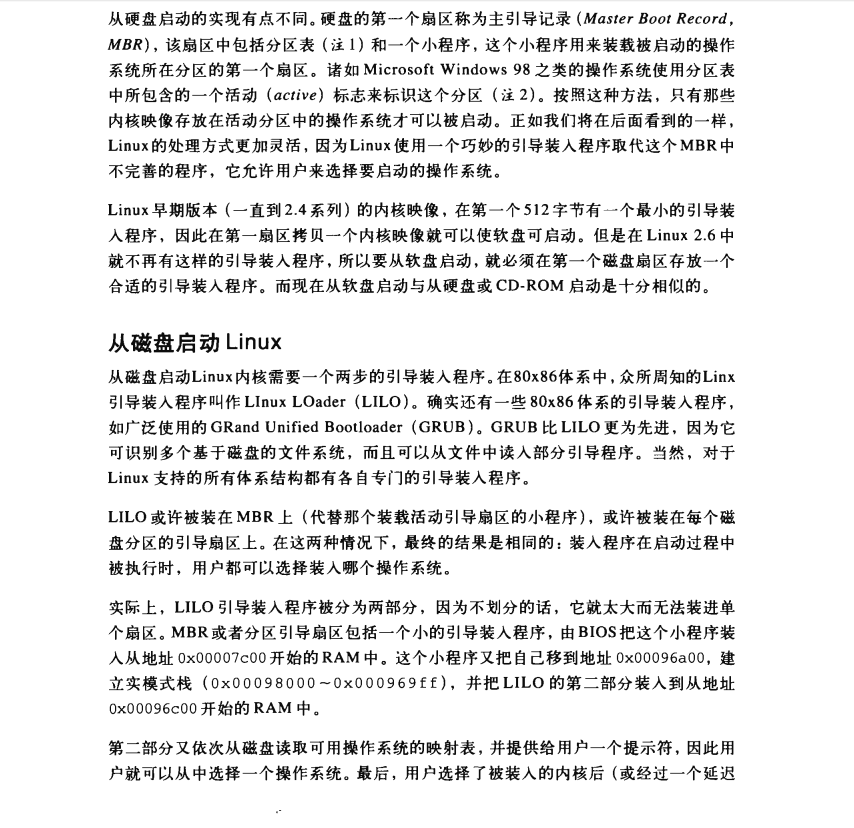
http://www.wowotech.net/tty\_framework/tty\_concept.html

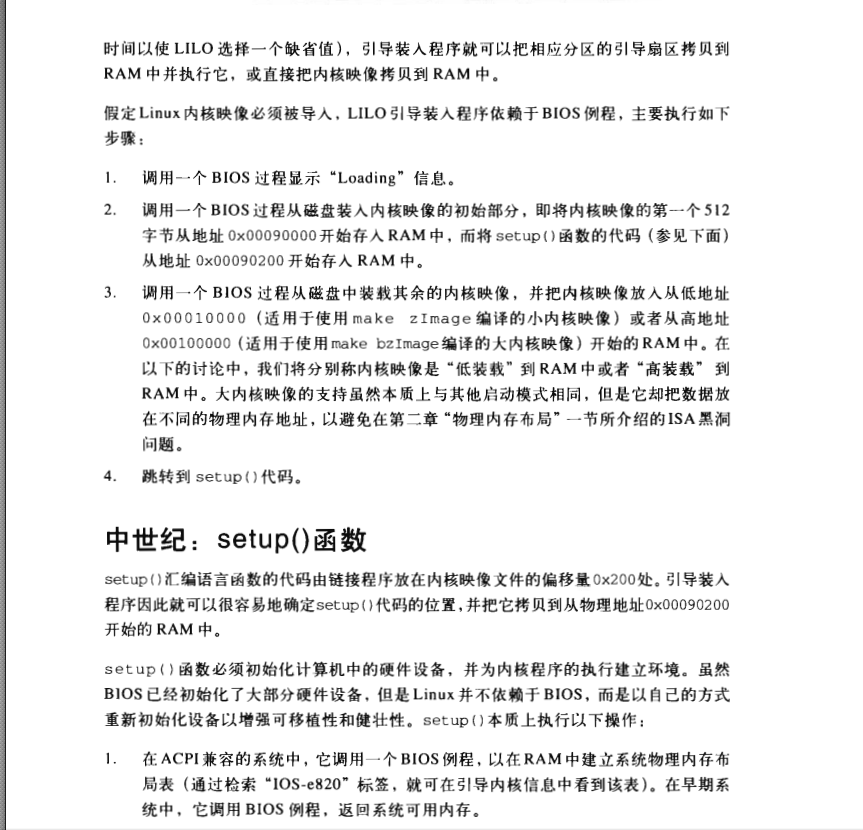
* + 1. Linux内核启动
       1. 历史

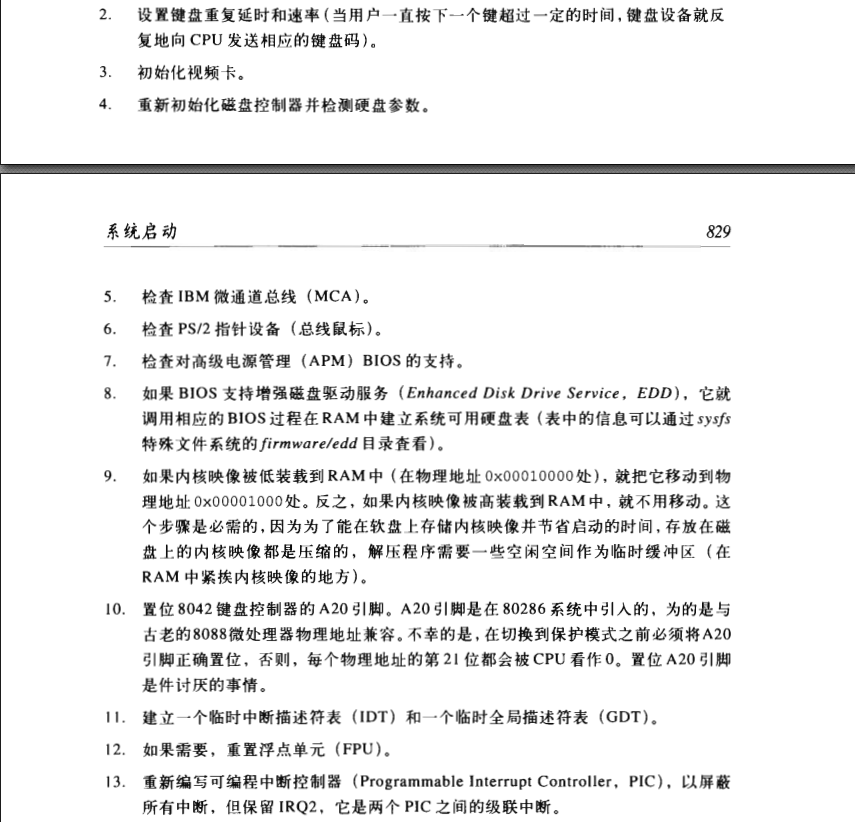
《深入Linux内核第三版》

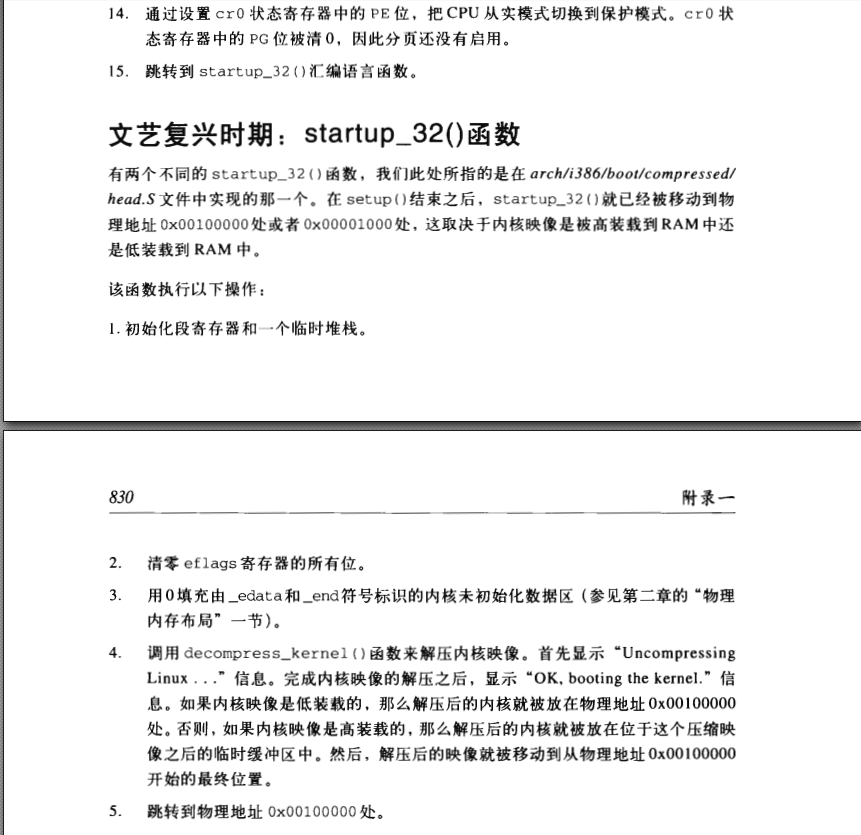


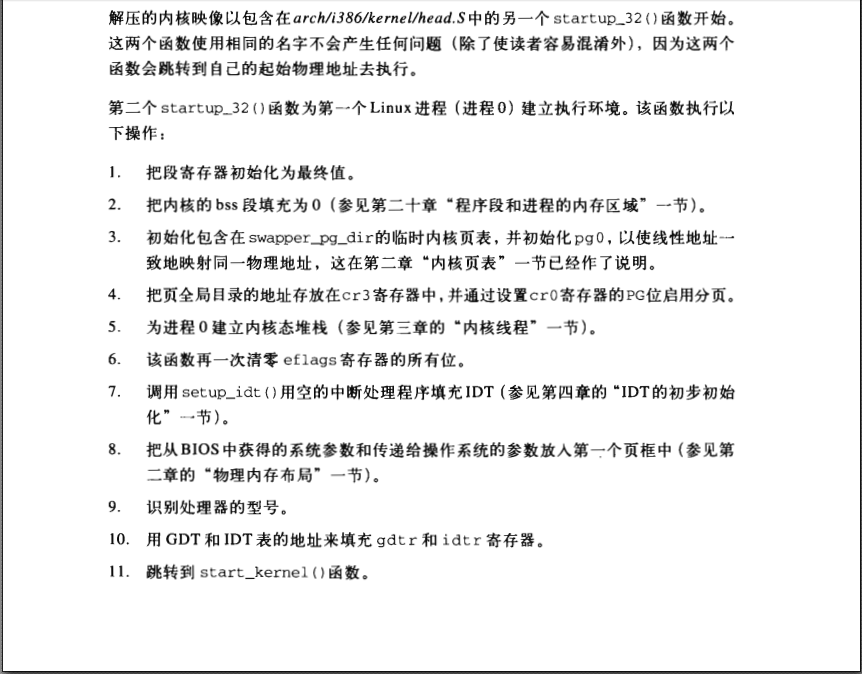


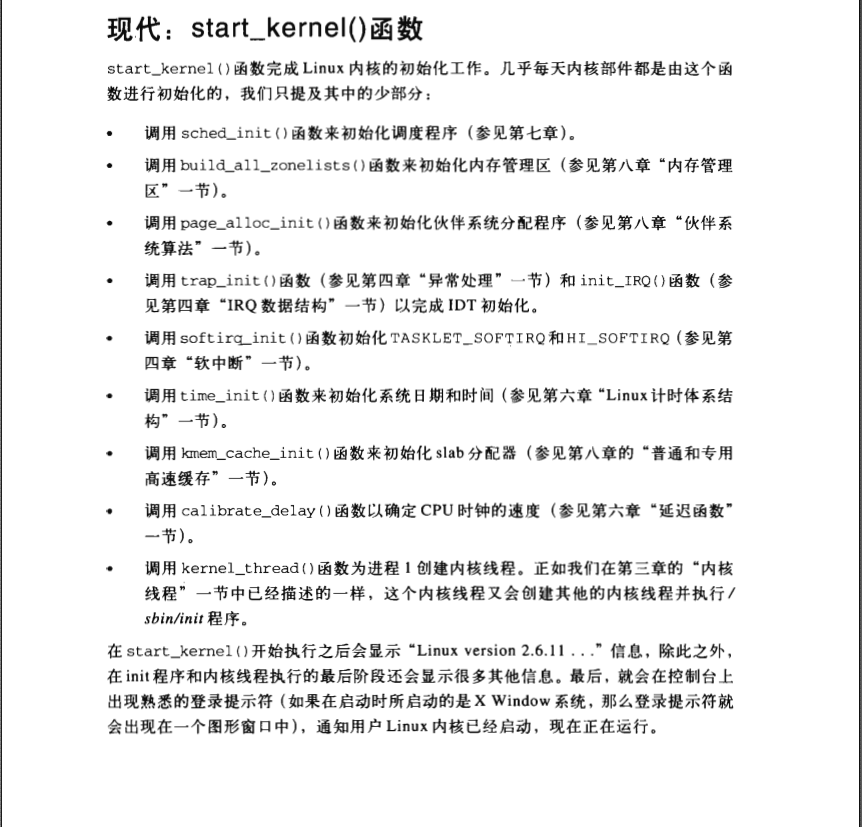












* + - 1. 嵌入式Linux内核启动