**1.内核模块 linux/module.h**

1.至少包含两个函数 1.初始化函数 2.卸载函数

通过module\_init()声明初始化函数

通过module\_exit()声明卸载函数

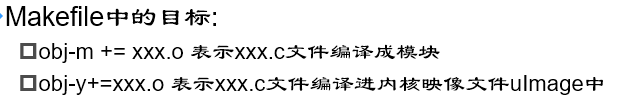
2.任一个内核模块需要包含linux/module.h

3.通过宏MODULE\_LICENSE(“GPL”)，设置模块遵守GPL证书，取消警告信息。

4.如何编译成模块



Make –C <内核源码path> M=<模块代码path> modules



**2.linux内核链表**

1.Linux内核的标准链表就是采用**环形双向链表**形式.

2.内核链表代码在头文件**<linux/list.h**>中声明

3. 使用宏container\_of(ptr, type, member)可找到父结构中包含的任何变量

* + - ptr: list\_head节点指针
    - type: 父结构的**类型**
    - member：list\_head结构在夫结构中的**成员名**

4.创建链表 2种方法

1.LIST\_HEAD(name);

2. struct list\_head name = LIST\_HEAD\_INIT(name);

5.遍历（宏函数）

1.**list\_for\_each\_entry(pos, head, member)**

pos： 自己声明一个父结构体的指针传递

head：链表的名字

member：在父结构体中list\_head类型的名字

6.杂项、备忘

判断节点是否是尾节点：

int list\_is\_last(struct list\_head \*list,struct list\_head \*head)

判断链表是否为空：

int list\_empty(struct list\_head \*head)

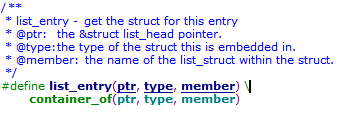
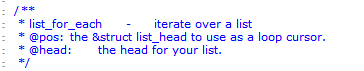
判断链表是否只有一个元素

int list\_is\_singular(struct list\_head \*head)

（1）**list\_for\_each\_entry(pos, head, member) 与 list\_for\_each(pos, head)的区别**

**1.由于list\_for\_each的两个参数都是list\_head类型，所以需要配合list\_entry使用**

**2.？？？**

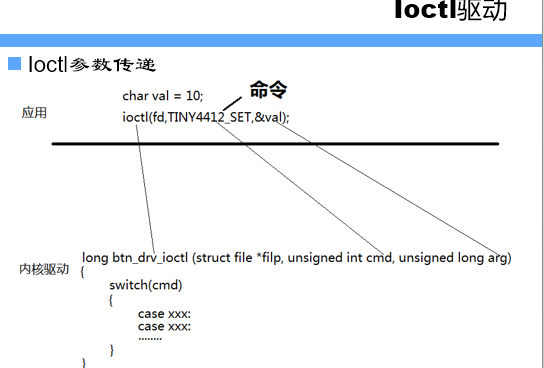
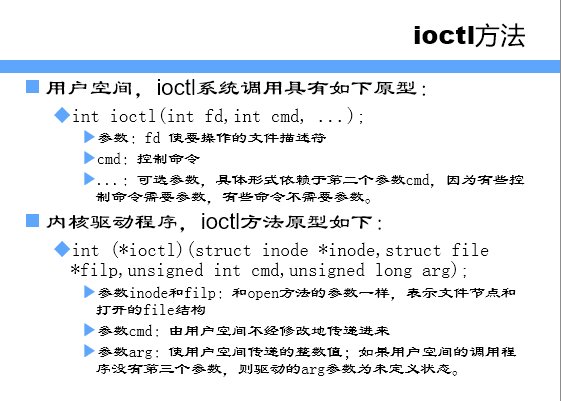


（2）内核链表与普通链表的区别

Linux内核中，链表的方式与众不同，它不是将数据结构塞入链表，而是**将链表节点（私人辅助理解：node概念）塞人数据结构**.

**3.ioctl方法**

**Ioctl的作用：通过设备驱动程序执行各种类型的硬件控制。**



**构造命令编号的方法 <<asm-generic/ioctl.h>>**

**宏函数**

**1.\_IO(type,nr):用于构造无参数的命令号**

**2.\_IOR(type,nr,datetype):用于构造从驱动程序中读取数据的命令号**

**3.\_IOW(type,nr,datatype):用于构造向驱动程序写入数据的命令号**

**4.\_IORW(type,nr,datatype):用于构造双向传输的命令号**

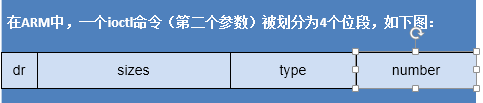
**解开位字段的宏：**

**\_IOC\_DIR(cmd):获得传输方向位段的值**

**\_IOC\_TYPE(cmd):获得类型的值**

**\_IOC\_NR(cmd);获得编号的值**

**\_IOC\_SIZE(cmd):获得大小的值**



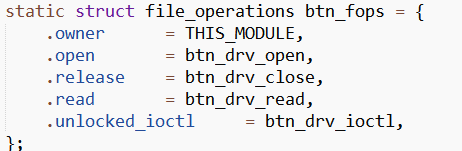
**私人备忘：**

**Ioctl驱动编写**

**1.根据ioctl方法原型，在驱动里实现一个xxx\_ioctl（）函数**

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

**2.将xxx\_ioctl添加到操作集合file\_operations**



**unlocked\_ioctl:不阻塞的ioctl**

**compat\_ioctl:兼容性的ioctl（用户空间为32位模式，而内核运行在64位模式时）。**

**4.poll() 阻塞与非阻塞**

**1.Struct pollfd结构体**

**struct pollfd {**

**int fd; /\*文件描述符\*/**

**short events; /\* 等待的需要测试事件 \*/**

**short revents; /\* 实际发生了的事件，也就是返回结果 \*/**

**};**

**2.等待队列**

**DECLARE\_WAIT\_QUEUE\_HEAD（）宏可以作为定义并初始化等待队列头部的“快捷方式” DECLARE\_WAIT\_QUEUE\_HEAD (name)**

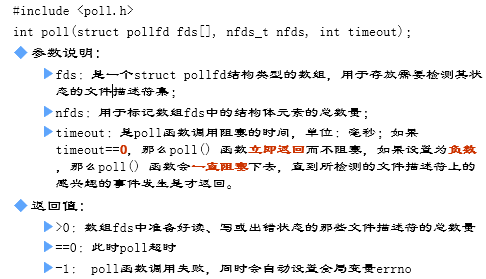
**等待事件**

**wait\_event\_interruptible(queue, condition)**

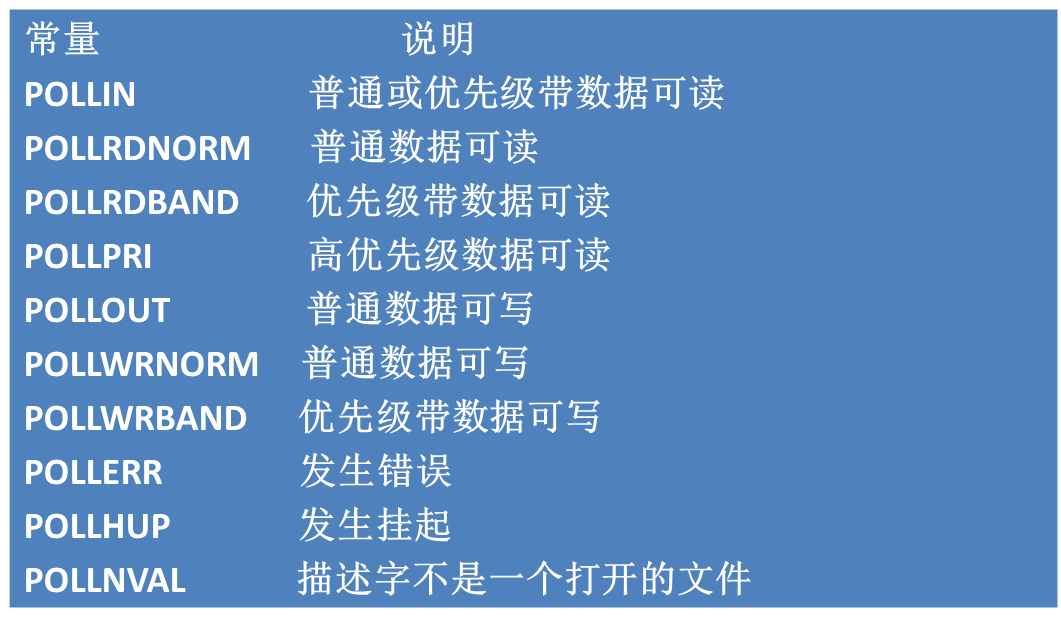
**唤醒队列**

**void wake\_up\_interruptible(wait\_queue\_head\_t \*queue);**

**3.poll函数**



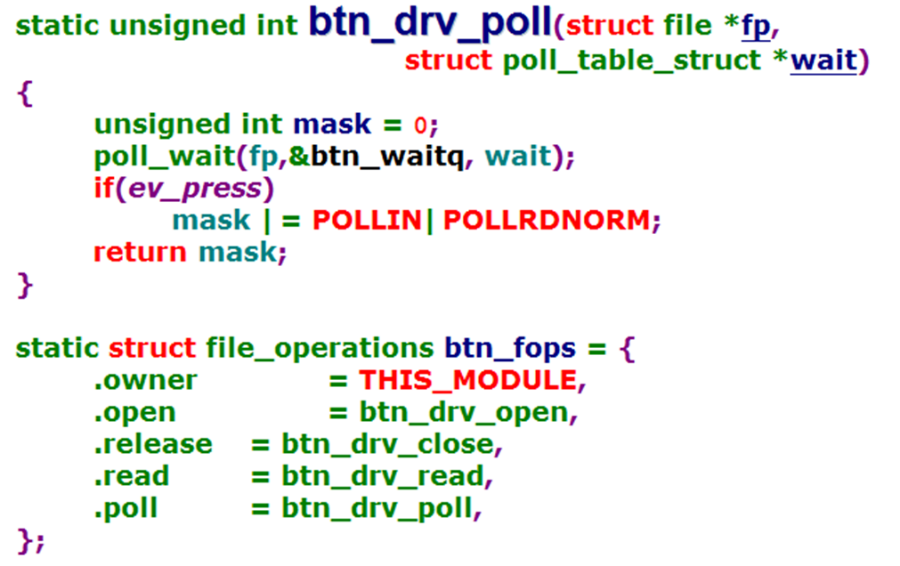
**events 与 revents的值**

****

**驱动程序中poll方法的实现**

**1.实现一个xxx\_poll()函数**

**2.添加到file\_operations集合中**

****

**内核中poll机制的实现**

****

1.应用程序调用poll()时，内核中会调用每个设备驱动中的poll函数，这些底层函数都会调用poll\_wait()将本设备驱动中的等待队列添加到一个等待队列表中（table）。

2.如果监控设备没有事件发生，在调用完所有要监控设备驱动的poll函数后，进程会休眠（table）

3.如果任何一个监控的设备有事件发生，内核中的poll又会重新带调用每个设备驱动中的poll方法。看看有多少个设备有事件发生，然后才返回到应用层。

**私人备忘：**

**I/O复用指的是：**

**解决能够同时操作多个设备的方法，及时处理多个设备的数据。**

**方法大概有：**

**1.应用程序使用poll,select系统调用**

**2.应用程序使用多线程技术**

**例如：处理按键一个线程，处理鼠标一个线程，由于线程是并行运行的，所有可以同时处理按键和鼠标。**

**5.sigio 异步通知**

**1.SIGIO信号**

**1.一般是驱动程式异步通知应用程序有事件发生的信号。**

**2.** **应用程序一般是忽略这个信号的；**

**3.应用程序要处理这个信号，要做以下设置**

**（1）设置设备驱动的拥有者是本进程**

**fcntl(fd, F\_SETOWN, getpid());**

**（2）设置文件标志位 FASYNC，才能使用异步通知**

**flags = fcntl(fd, F\_GETFL);**

**fcntl(fd, F\_SETFL, flags | FASYNC);**

**2.驱动程序实现异步通知实现的一个结构体、两个函数**

**（1）struct fasync\_struct;**

**（2）函数：**

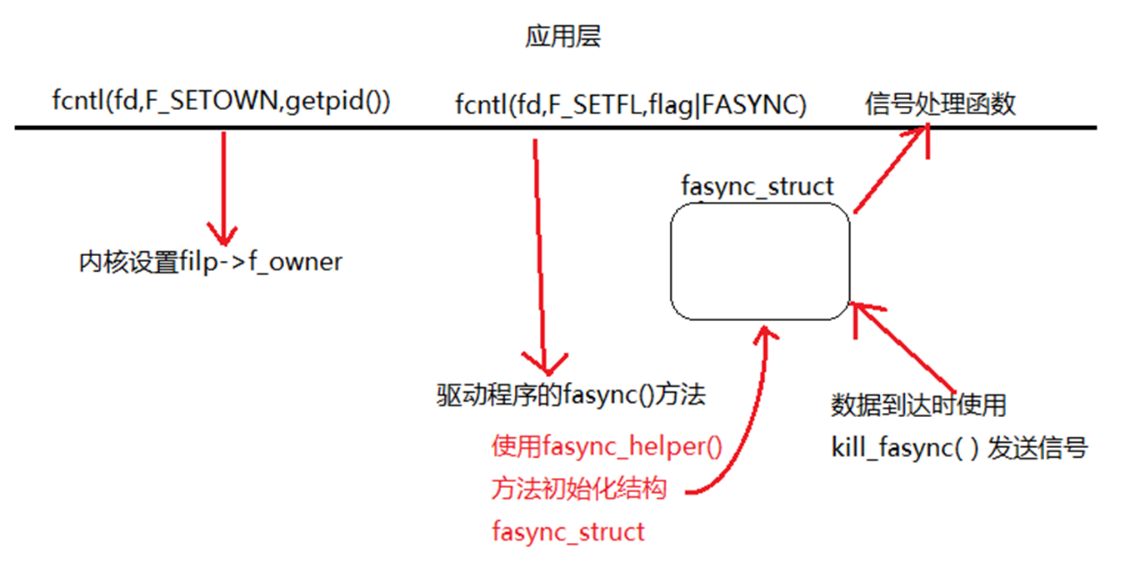
**fasync\_helper()**

**用于处理FASYNC标志变更**

**初始化fasync\_struct结构体变量，并设置异步通知队列的**

**kill\_fasync()**

**发送信号**

****

**删除异步通知**



**附录：**

**驱动编写：**

**1.全局结构体fasync\_struct \*xxx\_fasync**

**2.初始化函数xxx\_fasync() >> fasync\_helper()**

**3.中断处理函数xxx\_irq\_handler() >> kill\_fasync();**

**4.操作集合file\_operations**

**.release = xxx\_close,**

**.fasync = xxx\_fasync函数**

**1.** **int fasync\_helper(**

**int fd, //文件描述符**

**struct file \* filp, //文件指针**

**int on, //on为真则初始化**

**struct fasync\_struct \*\*fapp); //要设置的结构**

**2. void kill\_fasync(**

**struct fasync\_struct \*\*fp, //已初始化的fasync\_struct**

**int sig, //要发送的信号**

**int band)； //可读为POLL\_IN,可写为POLL\_OUT**

**6.misc (杂项设备：没有规律的设备)驱动 主设备号位10 <linux/miscdecice.h>**

**1.代表misc设备的结构体**

**struct miscdevice {**

**int minor; //次设备号**

**const char \*name; //名字，用这个名字来生产设备节点（设备文件）；**

**const struct file\_operations \*fops; //该设备的操作函数集合；**

**struct list\_head list;**

**struct device \*parent;**

**struct device \*this\_device;**

**const char \*nodename;**

**umode\_t mode;**

**};**

**2.杂项设备的注册、删除**

**（1）int misc\_register(struct miscdevice \* misc)；**

**（2）int misc\_deregister(struct miscdevice \*misc)；**

**3.驱动的实现**

* + **创建一个miscdevice结构变量，初始化其minor、 name、fops成员；**
  + **然后使用misc\_register注册该结构体即可；**



**4.如何找到操作集合**

**在misc\_opne()中以次设备号在链表misc\_list中找到对应的驱动。**

**7.input子系统 重点 主设备号 13**

**（1）在Linux中，输入子系统是由：**

* + **输入子系统设备驱动层**
  + **输入子系统核心层(Input Core)**
  + **输入子系统事件处理层(Event Handler)组成。**

**（2）各层次完成的工作**

**一、设备驱动层**

提供对硬件各寄存器的读写访问和将底层硬件对用户输入访问的响应转换为标准的输入事件，再通过核心层提交给事件处理层；

**二、核心层**

对下提供了设备驱动层的编程接口，对上又提供了事件处理层的编程接口；

**三、事件处理层**

为用户空间的应用程序提供了统一访问设备的接口和驱动层提交来的事件处理。

**（3）input driver编写要点**

* + **1 分配、释放、注册、注销input设备**

struct input\_dev \*input\_allocate\_device(void)

void input\_free\_device(struct input\_dev \*dev);

int input\_register\_device(struct input\_dev \*dev)

void input\_unregister\_device(struct input\_dev \*dev)

* + **2 设置input设备支持的事件类型、事件码、事件值的范围、input\_id等信息**

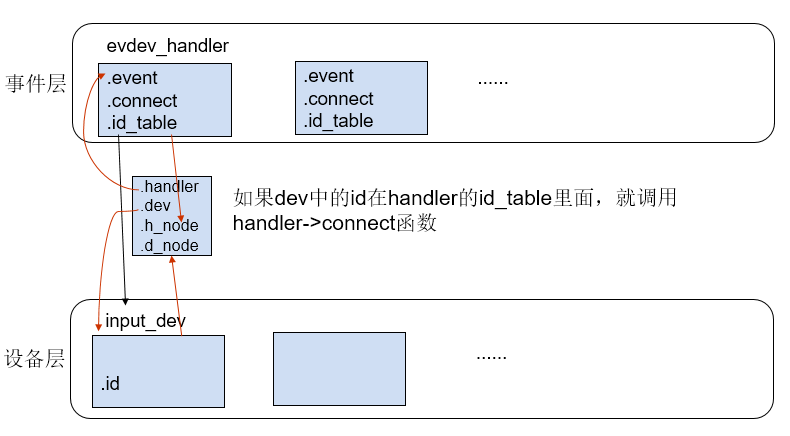
一个设备可以支持一个或多个事件类型。每个事件类型下面还需要设置具体的触发事件码。

* + **3 如果需要，设置input设备的打开、关闭、写入数据时的处理方法**
  + **4 在发生输入事件时，向子系统报告事件**

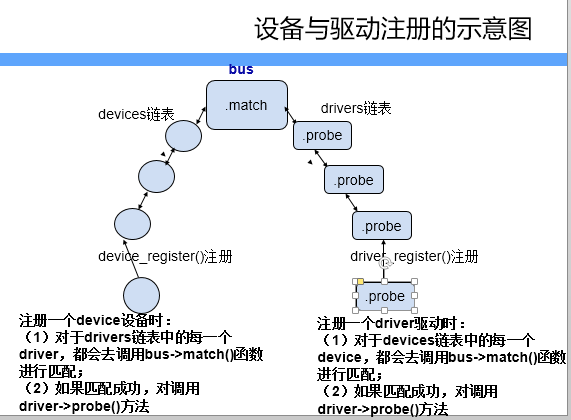
Input\_report\_xxx();用于上报发生的事件

input\_sync(input\_dev); 用于事件同步，它告知事件的接收者：驱动已经发出了一个完整的报告

（4）框架



**8.platform 平台设备驱动 重点：代码**



**1.platform\_driver的注册函数**

**int platform\_driver\_register(struct platform\_driver \*);**

**2.** **platform\_driver的注销函数**

**int platform\_driver\_unregister(struct platform\_driver \*);**

**3.** **platform\_device的注册函数**

**int platform\_device\_register(struct platform\_device \*);**

**4.** **platform\_device的注销函数**

**int platform\_device\_unregister(struct platform\_device \*);**

**个人备忘：**

**（1）bus\_device\_driver驱动机制**

**1.将驱动用到的数据和代码分离开**

**设备是数据，驱动是代码**

**2.好处：若设备参数变了，只需要修改数据，不用修改代码，就能驱动设备，省下程序员很多功夫。**

**（2）bus、device、dirver三者的定义在 include/linux/device.h中**

* + **struct bus\_type 表示总线**
  + **struct device\_driver 表示驱动**
  + **struct device 表示设备**

**（3）bus上比较重要的成员**

**1.** **match（） 匹配函数**

**2.** **probe（）探测函数**

**3.** **remove（）移除设备**

**4.shutdown（）系统关机**

**5.suspend（）设备休眠 resume（）设备恢复**

**6.pm（）电源管理**

**（4）结构体 struct platform\_device比较重要的成员**

**struct platform\_device {**

**const char \*name;//名字，用于匹配driver**

**int id;**

**struct device dev;**

PS：通过指定这两个成员来实现设备代表数据

**u32 num\_resources; //资源数目**

**struct resource \* resource; //资源**

**};**

**（5）Platform驱动的资源一般驱动根据硬件来调整的参数。**

**主要是中断号和寄存器地址**

**（6）常见的资源类型 struct resource <include/linux/ioport.h>**

**1. I/O 地址(IO一般特指x86,I/O端口)**

**2. 地址空间(寄存器空间) MEM,中断号,IRQ**

**#define IORESOURCE\_IO 0x00000100**

**#define IORESOURCE\_MEM 0x00000200**

**#define IORESOURCE\_IRQ 0x00000400**

**#define IORESOURCE\_DMA 0x00000800**

**（7）资源的组织与使用**

/\*\*

\* platform\_get\_resource - get a resource for a device

\* @dev: platform device

\* @type: resource type

\* @num: resource index

\*/

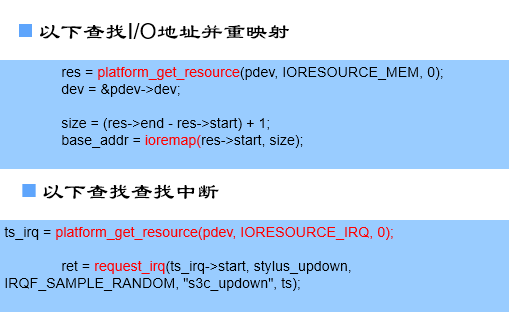
**参考 devs.c <arch/arm/plat-samsung>**

**（8）struct resource \*platform\_get\_resource(**

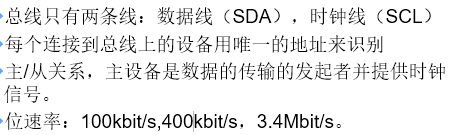
**struct platform\_device \*dev,**

**unsigned int type,**

**unsigned int num)**



**9.I2C串行总线**



**I2C总线信号（3种）**

* + **开始信号（S）：SCL为高电平时，SDA由高变低电平**
  + **结束信号（P）：SCL为高电平时，SDA由低变高电平**
  + **响应信号（ACK）：接收器在收到8位数据后，在第9个时钟周期拉低SDA。**

**1.Linux 的I2C体系结构**

**（1）I2C核心层**

提供了I2C**总线驱动**和**设备驱动**的**注册、注销方法**，**I2C通信方法**（即**Algorithm**）上层的与具体适配器无关的代码以及探测设备、检测设备地址的上层代码等

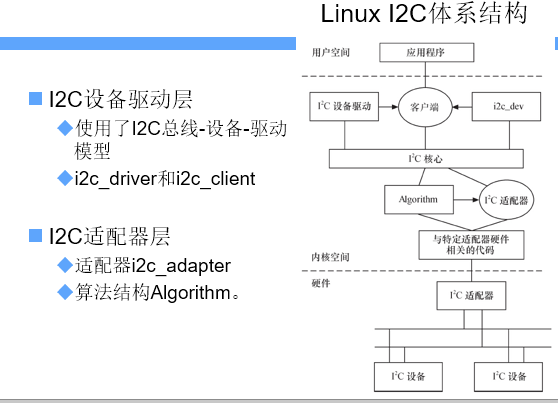
**（2）I2C适配层**

对I2C硬件体系结构中适配器端的实现，适配器可由CPU控制，甚至可以直接集成在CPU内部

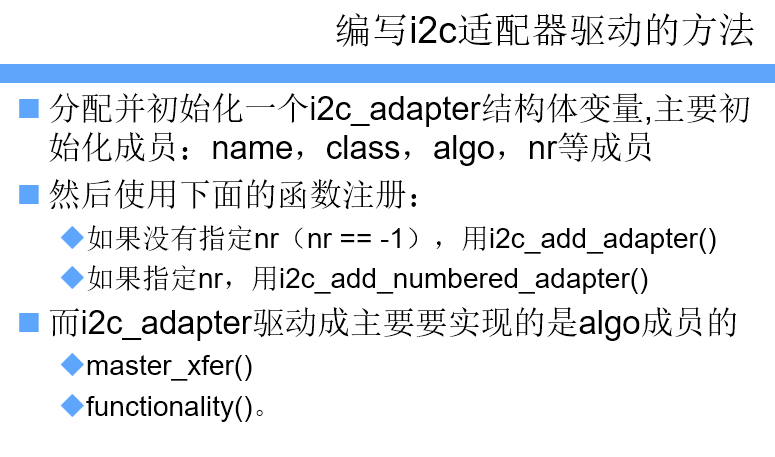
**（3）I2C设备驱动层**

I2C设备驱动（也称为客户驱动）是对设备端的实现，设备一般挂接在受CPU控制的I2C适配器上，通过I2C适配器与CPU交换数据。

**2.I2C体系结构图**

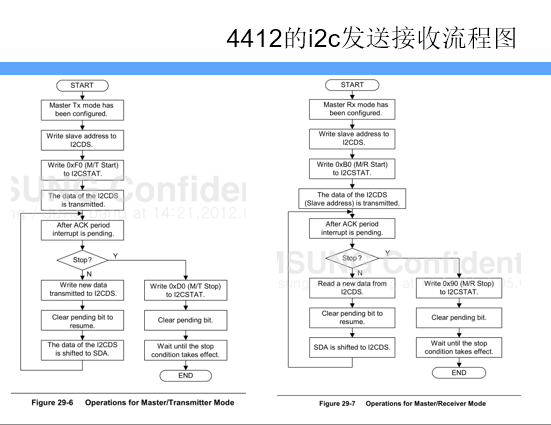


**3.I2C适配器驱动编写**



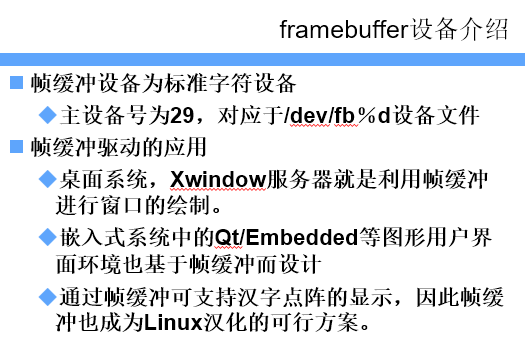
**备忘：**

**（1）适配器驱动最主要的数据传输函数是i2c\_xfer();**

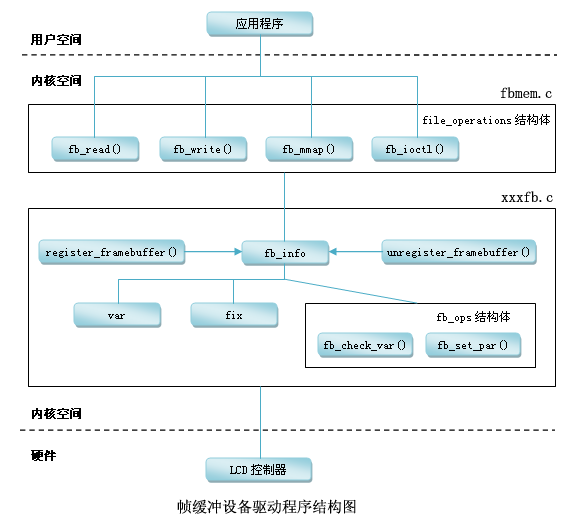
**该函数的实现与具体的的硬件相关** 

**11.framebuffer驱动 （帧缓冲） 主设备号 29**

**帧缓冲设备**



**（1）帧缓冲设备驱动程序结构图**

****

**备忘：**

**（0）帧缓冲的概念**

1.Linux系统为显示设备提供的一个接口;

2.显示缓冲区抽象，屏蔽图像硬件的底层差异;

3.上层应用程序在图形模式下直接对显示缓冲区进行读写操作;

4.用户不必关心物理显示缓冲区的具体位置及存放方式;

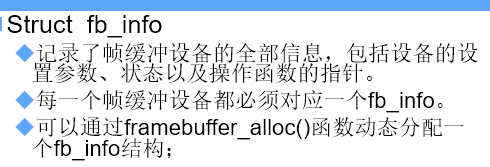
5.只要在显示缓冲区中与显示点对应的区域写入颜色值，对应的颜色会自动在屏幕上显示;

**（1）编写framebuffer驱动的主要步骤：**

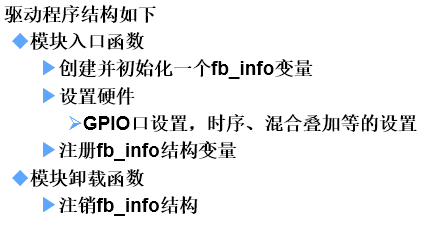
**1.** **初始化一个struct fb\_info结构；**

**2. 将这个结构注册到系统中（register\_framebuffer）；**

**（2）struct fb\_info**



**（3）Exynos 4412 的FB驱动程序设计**

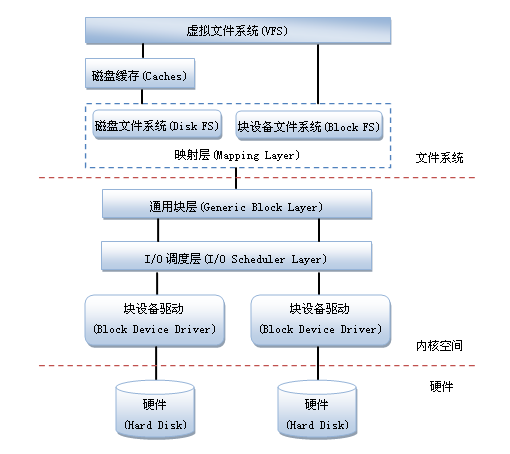


**（3）linux下的framebuffer驱动框架很简单，主要是围绕着一个struct fb\_info结构数组registered\_fb[]来操作**

**12.块设备驱动**

**（1）块设备驱动的实现步骤**

* **确定主设备号和次设备号**
* **确定设备名称**
* **创建设备文件**
* **实现块设备驱动程序** 
  + **实现block\_device\_operations结构体**
  + **实现request（请求）**
  + **实现初始化函数，注册块设备**
  + **实现销毁函数，取消块设备**

**（2）框架**

**备忘：**

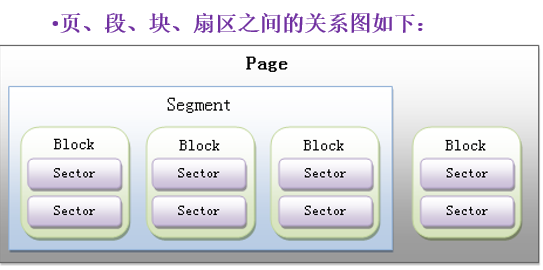
**（1）块设备的一些基本概念**

**1.扇区(Sectors)：任何块设备硬件对数据处理的基本单位。通常，1个扇区的大小为512byte。**

**2.块(Blocks)：由Linux制定对内核或文件系统等数据处理的基本单位。通常，1个块由1个或多个扇区组成。**

**3.段(Segments)：由若干个相邻的块组成。是Linux内存管理机制中一个内存页或者内存页的一部分。**

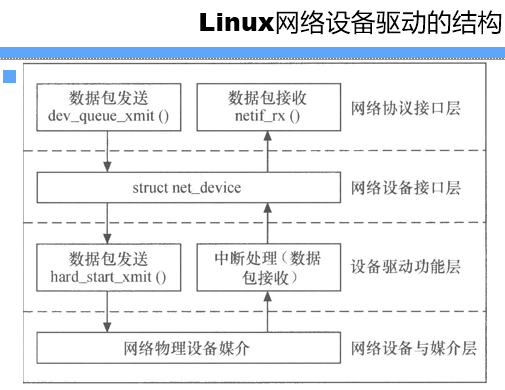
**（2）**



**（3）**

* **块设备驱动是基于扇区(sector)来访问底层物理磁盘，基于块(block)来访问上层文件系统。**
* **扇区一般是2的n次方大小，典型为512B，内核也要求块是2的n次方大小，且块大小通常为扇区大小的整数倍，并且块大小要小于页面大小，典型大小为512B、1K或4K。**

**13.网络设备驱动**



**具体了解**

****

**备忘：**

**网络设备的特点**

**1.** **网络设备，又叫网络接口是Linux第三类标准设备**

**2.** **网络设备异步的接收外来的数据包，有别于其他设备**

**3.** **网络子系统是完全与协议无关的**

**4.** **网络驱动程序与内核其余部分之间的每次交互处理的都是一个网络数据包**

**5.** **网络设备不会在/dev下存在一个设备入口，它使用保留的内部设备名**

**……**

**（1）TCP/IP与OSI参考模型的对应关系**

****

**（2）TCP/IP协议通信模型**

****