

第8章 设备驱动程序设计



目 录

- 8.1 设备驱动程序开发概述
 - 8.2 内核设备模型
 - 8.3 字符设备驱动设计框架
 - 8.4 GPIO驱动概述
 - 8.5 IIC总线驱动设计
 - 8.6 块设备驱动程序设计概述
 - 8.7 嵌入式网络设备驱动设计
- 

设备驱动程序是应用程序和硬件设备之间的一个软件层，它向下负责和硬件设备的交互，向上通过一个通用的接口挂接到文件系统上，从而使用户或应用程序可以无需考虑具体的硬件实现环节。

由于设备驱动程序为应用程序屏蔽了硬件细节，在用户或者应用程序看来，硬件设备只是一个透明的设备文件，应用程序对该硬件进行操作就像是对普通的文件进行访问和控制硬件设备（如打开、关闭、读和写等）。

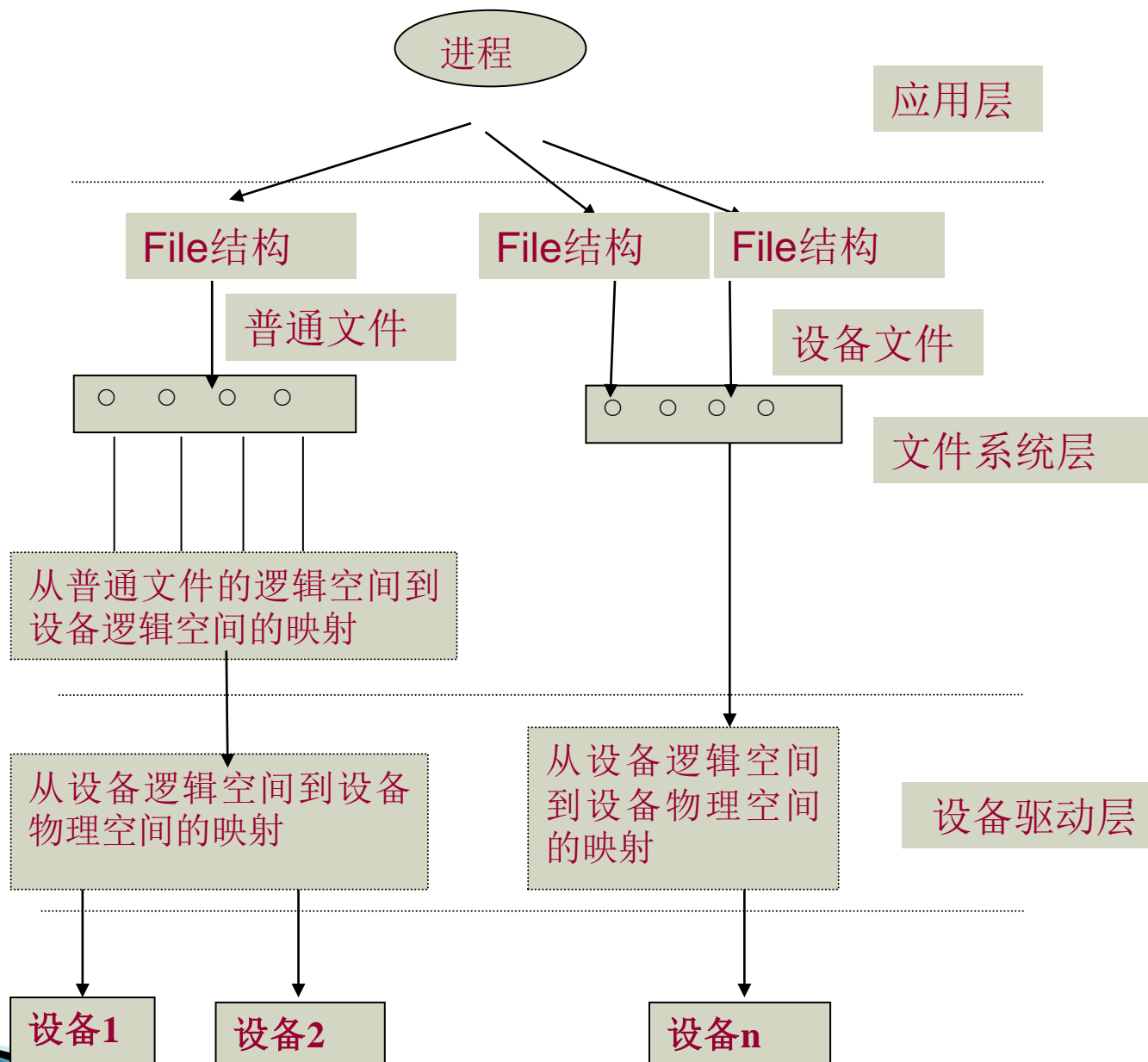


- ❖ Linux操作系统把设备纳入文件系统的范畴来管理
- ❖ 每个设备都对应一个文件名，在内核中也就对应一个索引节点
- ❖ 对文件操作的系统调用大都适用于设备文件
- ❖ 从应用程序的角度看，设备文件逻辑上的空间是一个线性空间（起始地址为0，每读取一个字节加1）。从这个逻辑空间到具体设备物理空间（如磁盘的磁道、扇区）的映射则是由内核提供，并被划分为文件操作和设备驱动两个层次

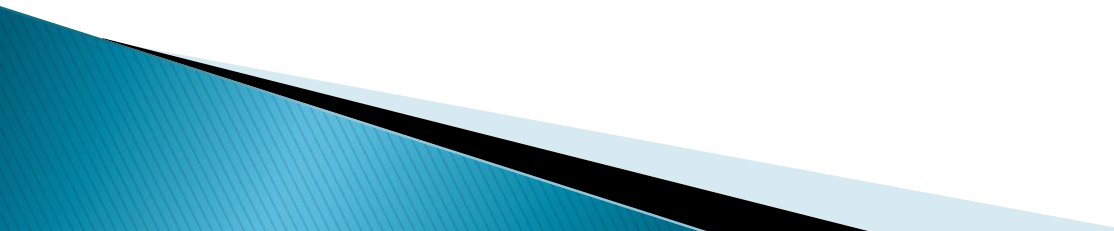
文件操作是对设备操作的组织和抽象，而设备操作则是对文件操作的最终实现



- ❖ 对于一个具体的设备而言，文件操作和设备驱动是一个事物的不同层次。从这种观点出发，从概念上可以把一个系统划分为应用、文件系统和设备驱动三个层次
- ❖ Linux将设备分成三大类。一类是像磁盘那样以块或扇区为单位，成块进行输入 / 输出的设备，称为**块设备**；另一类像键盘那样以字符（字节）为单位，逐个字符进行输入 / 输出的设备，称为**字符设备**，还有一类是**网络设备**。
- ❖ 文件系统通常都建立在块设备上



作为Linux内核的重要组成部分，设备驱动程序主要完成以下的功能：

- (1) 对设备初始化和释放。
 - (2) 把数据从内核传送到硬件和从硬件读取数据。
 - (3) 读取应用程序传送给设备文件的数据和回送应用程序请求的数据。
 - (4) 检测错误和处理中断。
- 

设备驱动程序有如下特点：

- ✓ 驱动程序是与设备相关的。
- ✓ 驱动程序的代码由内核统一管理。
- ✓ 驱动程序在具有特权级别的内核态下运行。
- ✓ 设备驱动程序是输入输出系统的一部分。
- ✓ 驱动程序是为某个进程服务的，其执行过程仍处在进程运行的过程中，即处于进程的上下文中。
- ✓ 若驱动程序需要等待设备的某种状态，它将阻塞当前进程，把进程加入到该设备的等待队列中。

8.1

Part One

设备驱动程序开发概述

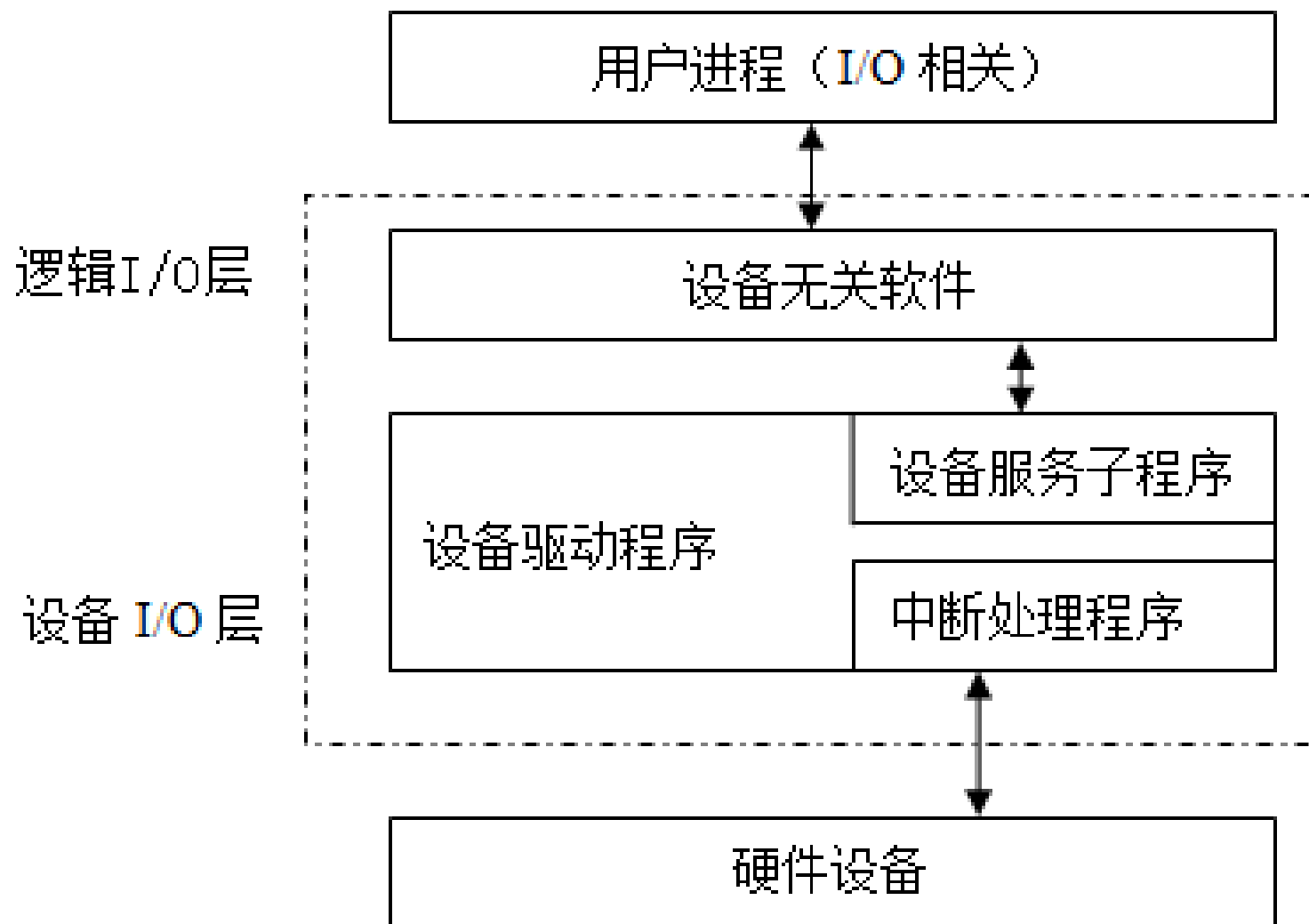


图8-1 驱动层次结构图

Linux 设备驱动程序可以分为两个主要组成部分：

(1) 对子程序进行自动配置和初始化，检测驱动的硬件设备是否正常，能否正常工作。

(2) 设备服务子程序和中断服务子程序，这两者分别是驱动程序的上下两部分。驱动上部分即设备服务子程序的执行是系统调用的结果，并且伴随着用户态向核心态的演变，在此过程中还可以调用与进程运行环境有关的函数，比如 `sleep()` 函数。驱动程序的下半部分即中断服务子程序。

8.1.1 Linux 设备驱动程序分类

1. 字符设备

字符设备是传输数据以字符为单位进行的设备，字符设备驱动程序通常实现**open**、**close**、**read**和**write**等系统调用函数，常见的字符设备有键盘、串口、控制台等。通过文件系统节点可以访问字符设备，例如**/dev/tty1**和**/dev/lp1**。字符设备和普通文件系统之间唯一的区别是普通文件允许往复读写，而大多数字符设备驱动仅是数据通道，只能顺序读写。此外，字符设备驱动程序不需要缓冲且不以固定大小进行操作，它与用户进程之间直接相互传输数据。

2. 块设备

所谓块设备是指对其信息的存取以“块”为单位。如常见的光盘、硬磁盘、软磁盘、磁带等，块长大小通常取512B、1024B或4096B等。块设备和字符设备一样可以通过文件系统节点来访问。在大多数linux系统中，只能将块设备看作多个块进行访问，一个块设备通常是1024B数据。

块设备的特点是对设备的读写是以块为单位的，并且对设备的访问是随机的。块设备和字符设备的区别主要在于内核内部的管理上，其中应用程序对于字符设备的每个I/O操作都会直接传递给系统内核对应的驱动程序；而应用程序对于块设备的操作要经过系统的缓冲区管理间接地传递给驱动程序处理。

每一个字符设备或块设备都在/dev目录下对应一个设备文件。linux用户程序通过设备文件（或称设备节点）来使用驱动程序操作字符设备和块设备。

3. 网络设备

网络设备驱动通常是通过套接字（**Socket**）等接口来实现操作。任何网络事务处理都可以通过接口来完成和其他宿主机数据的交换。内核和网络设备驱动程序之间的通信与字符设备驱动程序和块设备驱动程序与内核的通信是完全不同的。

8.1.2 驱动程序的处理过程

如果逻辑I/O层请求读取块设备的第j块，假设请求到来时驱动程序处于空闲状态，那么驱动程序立刻执行该请求，由于外设速度相比CPU要慢很多，因此进程会在该数据块缓存上阻塞，并调度新的进程运行。但是如果驱动程序同时正在处理另一个请求，那么就将请求挂在一个请求队列中，对应的请求进程也阻塞于所请求的数据块。

当完成一个请求的处理时，设备控制器向系统发出一个中断信号。结束中断的处理方法是将设备控制器和通道的控制块均置为空闲状态，然后查看请求队列是否为空。如果为空则驱动程序返回，反之则继续处理下一个请求。如果传输错误，则向系统报告错误或者进行相应进程重复执行处理。对于故障中断，则向系统报告故障，由系统进一步处理。

驱动程序涉及的重要概念:

1. 内存与 I/O 端口

编写驱动程序大多数情况下其本质都是对内存和 I/O 端口的操作。

(1) 内存

Linux通常有以下几种地址类型:

用户虚拟地址

物理地址

总线地址

内核逻辑地址

内核虚拟地址

(2) I/O 端口

有两个重要的内核调用可以保证驱动程序使用正确的端口，它们定义在 `include/linux/ioport.h` 中。

```
int __check_region(struct resource *, resource_size_t,  
resource_size_t);
```

该函数的作用是查看系统I/O表，看是否有别的驱动程序占用某一段I/O口。

根据 CPU 系统结构的不同，CPU 对 I/O 端口的编址方式通常有两种：

第一种是 I/O 映射方式，如 x86 处理器为外设专门实现了一个单独的地址空间，称为 I/O 地址空间，CPU 通过专门的 I/O 指令来访问这一空间的地址单元；

第二种是内存映射方式，RISC 指令系统的 CPU（如 ARM、Power PC 等）通常只实现一个物理地址空间，外设 I/O 端口成为了内存的一部分，此时 CPU 访问 I/O 端口就像访问一个内存单元，不需要单独的 I/O 指令。

这两种方式在硬件实现上的差异对软件来说是完全可见的。

2. 并发控制

在驱动程序中经常会出现多个进程同时访问相同的资源时可能会出现竞态（**race condition**），即竞争资源状态，因此必须对共享资料进行并发控制。Linux 内核中解决并发控制最常用的方法是自旋锁（**spinlocks**）和信号量（**semaphores**）。

（1）自旋锁

自旋锁是一个互斥现象的设备，它只能是两个值：**locked**（锁定）或**unlocked**（解锁）。它通常作为一个整型值的单位来实现。在任何时刻，自旋锁只能有一个保持者，也就是说在同一时刻只能有一个进程获得锁。

（2）信号量

信号量是一个结合一对函数的整型值，这对函数通常称为**P**操作和**V**操作。

自旋锁和信号量有很多相似之处但又有些本质的不同。其相同之处主要有：首先它们对互斥来说都是非常有用的工具；其次在任何时刻最多只能有一个线程获得自旋锁或信号量。

不同之处主要有：

首先自旋锁可在不能睡眠的代码中使用，如在中断服务程序（ISR）中使用，而信号量不可以；

其次自旋锁和信号量的实现机制不一样；最后通常自旋锁被用在多处理器系统。

总体而言，自旋锁通常适合保持时间非常短的情况，它可以在任何上下文中使用，而信号量用于保持时间较长的情况，只能在进程上下文中使用。

3. 阻塞与非阻塞

在驱动程序的处理过程中我们提到了阻塞的概念，这里进行以下说明。**阻塞（blocking）**和**非阻塞（nonblocking）**是设备访问的两种不同模式，前者在I/O操作暂时不可进行时会让进程睡眠，而后者在I/O操作暂时不可进行时并不挂起进程，它或者放弃，或者不停地查询，直到可以进行操作为止。

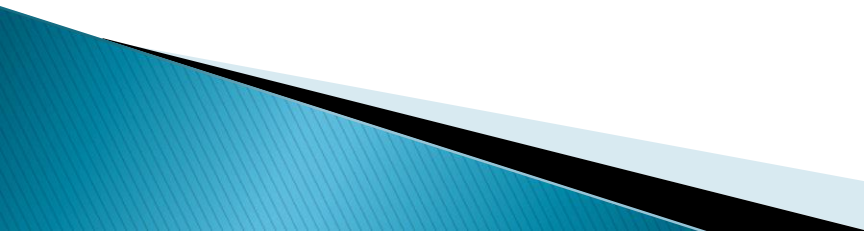
4. 中断处理

与Linux设备驱动程序中断处理相关的函数首先是申请和释放IRQ(中断请求)函数，即`request_irq` 和 `free_irq`，这两个重要的中断函数原型如下，在头文件`include/linux/interrupt.h` 中声明。

申请函数原型如下：

```
int request_irq (unsigned int irq, void (*handler)(int, void *, struct  
pt_regs *), unsigned long flags, const char *device, void  
*dev_id); (2.4 内核中)
```

```
request_irq(unsigned int irq, irq_handler_t handler, unsigned  
long flags, const char *name, void *dev); (2.6 内核及以后)
```



释放函数原型如下：

`void free_irq(unsigned int irq,void *dev_id);` （2.4版本）

`void free_irq(unsigned int irq,void *dev) ;` （2.6版本及以后）

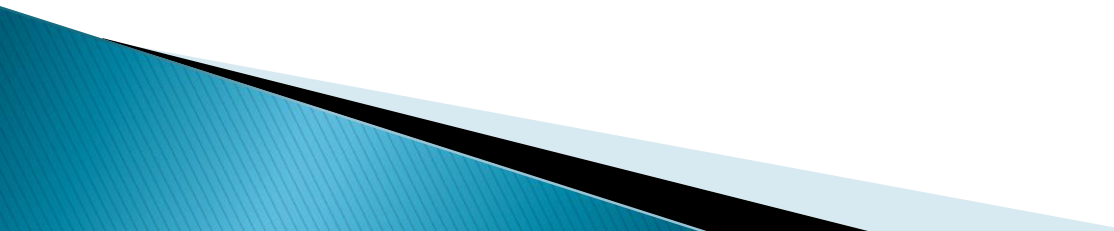
该函数的作用是释放一个IRQ，一般是在退出设备或关闭设备时调用

Linux将中断分为两个部分：上半部分（top half）和下半部分（bottom half）

上半部分的功能是注册中断

上半部和下半部最大的不同是下半部是可中断的，而上半部是不可中断的，会被内核立即执行，

下半部完成了中断处理程序的大部分工作，所以通常比较耗时，因此下半部由系统自行安排运行，不在中断服务上下文执行。



5. 设备号

用户进程与硬件的交流是通过设备文件进行的，硬件在系统中会被抽象成为一个设备文件，访问设备文件就相当于访问其所对应的硬件。每个设备文件都有其文件属性(c/b)，表示是字符设备还是块设备。

每个设备文件的设备号有两个：

第一个是主设备号，标识驱动程序对应一类设备的标识；


第二个是从设备号，用来区分使用共用的设备驱动程序的不同硬件设备。

在linux2.6内核中，主从设备被定义为一个dev_t类型的32位数，其中前12位表示主设备号，后20位表示从设备号。另外，在include/linux/kdev.h中定义了如下的几个宏来操作主从设备号。

```
#define MAJOR(dev)      ((unsigned int) ((dev) >> MINORBITS))  
#define MINOR(dev)     ((unsigned int) ((dev) & MINORMASK))  
#define MKDEV(ma,mi)   (((ma) << MINORBITS) | (mi))
```

上述宏分别实现从32位dev_t类型数据中获得主设备号、从设备号及将主设备号和从设备号转换为dev_t类型数据的功能。

主设备号和次设备号



brw-rw----	1	root	disk	8,	0	Jul	28	12:38	sda
brw-rw----	1	root	disk	8,	1	Jul	28	12:38	sda1
brw-rw----	1	root	disk	8,	2	Jul	28	12:38	sda2
brw-rw----	1	root	disk	8,	5	Jul	28	12:38	sda5
crw-rw----	1	root	disk	21,	0	Jul	28	12:38	sg0
crw-rw----+	1	root	cdrom	21,	1	Jul	28	12:38	sg1
drwxrwxrwt	3	root	root		240	Jul	29	15:35	shm
crw-----	1	root	root	10,	231	Jul	28	12:38	snapshot
drwxr-xr-x	3	root	root		200	Jul	28	12:38	snd
brw-rw----+	1	root	cdrom	11,	0	Jul	28	12:38	sr0
lrwxrwxrwx	1	root	root		15	Jul	28	12:38	stderr -> /proc/self/fd/2
lrwxrwxrwx	1	root	root		15	Jul	28	12:38	stdin -> /proc/self/fd/0
lrwxrwxrwx	1	root	root		15	Jul	28	12:38	stdout -> /proc/self/fd/1
crw-rw-rw-	1	root	tty	5,	0	Jul	29	15:33	tty
crw--w----	1	root	tty	4,	0	Jul	28	12:38	tty0
crw--w----	1	root	tty	4,	1	Jul	28	12:56	tty1
crw--w----	1	root	tty	4,	10	Jul	28	12:38	tty10
crw--w----	1	root	tty	4,	11	Jul	28	12:38	tty11
crw--w----	1	root	tty	4,	12	Jul	28	12:38	tty12
crw--w----	1	root	tty	4,	13	Jul	28	12:38	tty13
crw--w----	1	root	tty	4,	14	Jul	28	12:38	tty14

8.1.3 设备驱动程序框架

Linux的设备驱动程序可以分为以下部分：

（1）驱动程序与内核的接口，这是通过关键数据结构`file_operations`来完成的。

（2）驱动程序与系统引导的接口，这部分利用驱动程序对设备进行初始化。

（3）驱动程序与设备的接口，这部分描述了驱动程序如何与设备进行交互,这与具体设备密切相关。

根据功能划分，设备驱动程序代码通常可分为以下几个部分：

（1）驱动程序的注册与注销

对于字符设备或者是块设备，关键的一步还要向内核注册该设备，linux操作系统也专门提供了相应的功能函数，如字符设备注册函数 `register_chrdev()`、块设备注册函数 `register_blkdev()`。

在设备关闭时，要在内核中注销该设备，操作系统也相应的提供了注销设备的函数 `unregister_chrdev()`、`unregister_blkdev()`，并释放设备号。

在内核中使用一个数组chrdevs[]保存所有字符设备驱动程序的信息，在fs/char_dev.c中该数组的数据结构如下所示：

```
static struct char_device_struct {  
    struct char_device_struct *next;  
    unsigned int major;    //主设备号  
    unsigned int baseminor; //从设备号  
    int minorct;    //从设备个数  
    char name[64];  
    struct cdev *cdev;  
} *chrdevs[CHRDEV_MAJOR_HASH_SIZE]; //最大是255个主设备
```



```
<include/linux/cdev.h>
struct cdev {
struct kobject kobj; //内嵌的内核对象.
struct module *owner;
//该字符设备所在的内核模块（所有者）的对象指针，一般为
THIS_MODULE主要用于模块计数
const struct file_operations *ops;
//该结构描述了字符设备所能实现的操作集（打开、关闭、读/写、...），是
极为关键的一个结构体
struct list_head list;
//用来将已经向内核注册的所有字符设备形成链表
dev_t dev;
//字符设备的设备号，由主设备号和次设备号构成（如果是一次申请多个设
备号，此设备号为第一个）
unsigned int count;
//隶属于同一主设备号的次设备号的个数
... };
```

字符设备驱动程序的注册其实就是将字符设备驱动程序插入到chrdevs[]数组中。Linux通过字符设备注册函数register_chrdev()来完成注册功能。其函数原型如下：

```
int __register_chrdev(unsigned int major, unsigned int  
baseminor, unsigned int count, const char *name, const  
struct file_operations *fops)
```

Linux通过字符设备注销函数 `unregister_chrdev`来完成注销功能。其函数原型如下：

```
void __unregister_chrdev(unsigned int major, unsigned int baseminor,  
                        unsigned int count, const char *name)
```

块设备比字符设备要复杂，但是块设备驱动程序也需要一个主设备号来标识。块设备驱动程序的注册时通过 `register_blkdev()`函数实现的。其函数原型如下：

```
int __register_blkdev(unsigned int major, const char *name)
```



/dev/hello

fd=open("/dev/hello",O_RDWR)

VFS

```
struct inode {  
    dev_t      i_rdev;  
};
```

```
struct file {  
    const struct file_operations *f_op;  
    ...  
};
```

chrdevs

```
static struct char_device_struct {  
    struct char_device_struct *next;  
    unsigned int major;  
    unsigned int baseminor;  
    int minorct;  
    char name[64];  
    struct cdev *cdev;  
} *chrdevs[CHRDEV_MAJOR_HASH_SIZE];
```

```
struct cdev {  
    struct kobject kobj;  
    struct module *owner;  
    const struct file_operations *ops;  
    struct list_head list;  
    dev_t dev;  
    unsigned int count;  
};
```

```
struct file_operations {  
    struct module *owner;  
    loff_t (*llseek) (struct file *, loff_t, int);  
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);  
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);  
};
```

(2) 设备的打开与释放

打开设备是由调用定义在include/linux/fs.h中的file_operations结构体中的 open()函数完成的。open()函数主要完成的主要工作：

增加设备的使用计数。

检测设备是否异常，及时发现设备相关错误，防止设备有未知硬件问题。

若是首次打开，首先完成设备初始化。

读取设备次设备号。

其函数原型如下：

```
int (*open) (struct inode *, struct file *);
```

释放设备由 **release()** 完成，包括以下几件事情：

释放 **open** 时系统为之分配的内存；

释放所占用的资源，并进行检测，关闭设备，并递减设备使用计数。

其函数原型如下：

```
int (*release) (struct inode *, struct file *);
```

(3) 设备的读写操作

字符设备对数据的读写操作是由各自的 `read()` 函数和 `write()` 函数来完成的。

对块设备的读写操作,由文件 `block_devices.C` 中定义的函数 `blk_read()` 和 `blk_write()` 完成。真正需要读写的时候由每个设备的 `request()` 函数根据其参数 `cmd` 与块设备进行数据交换。

(4) 设备的控制操作

ioctl()函数就是驱动程序提供的控制函数。该函数的使用和具体设备密切相关。在linux内核版本**2.6.35**以前，**ioctl()**函数原型如下：

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


在2.6.35以后，系统已经完全删除了struct file_operations 中的ioctl 函数指针，剩下unlocked_ioctl和compat_ioctl，取而代之的是unlocked_ioctl,主要改进就是不再需要上大内核锁（调用之前不再先调用lock_kernel()然后再unlock_kernel()）。

如下是unlocked_ioctl和compat_ioctl的原型：

```
long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);  
long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
```

(5) 设备的轮询和中断处理

对于支持中断的设备，可以按照正常的中断方式进行。但是对于不支持中断的设备，过程就相对繁琐，在确定是否继续进行数据传输时都需要轮询设备的状态。

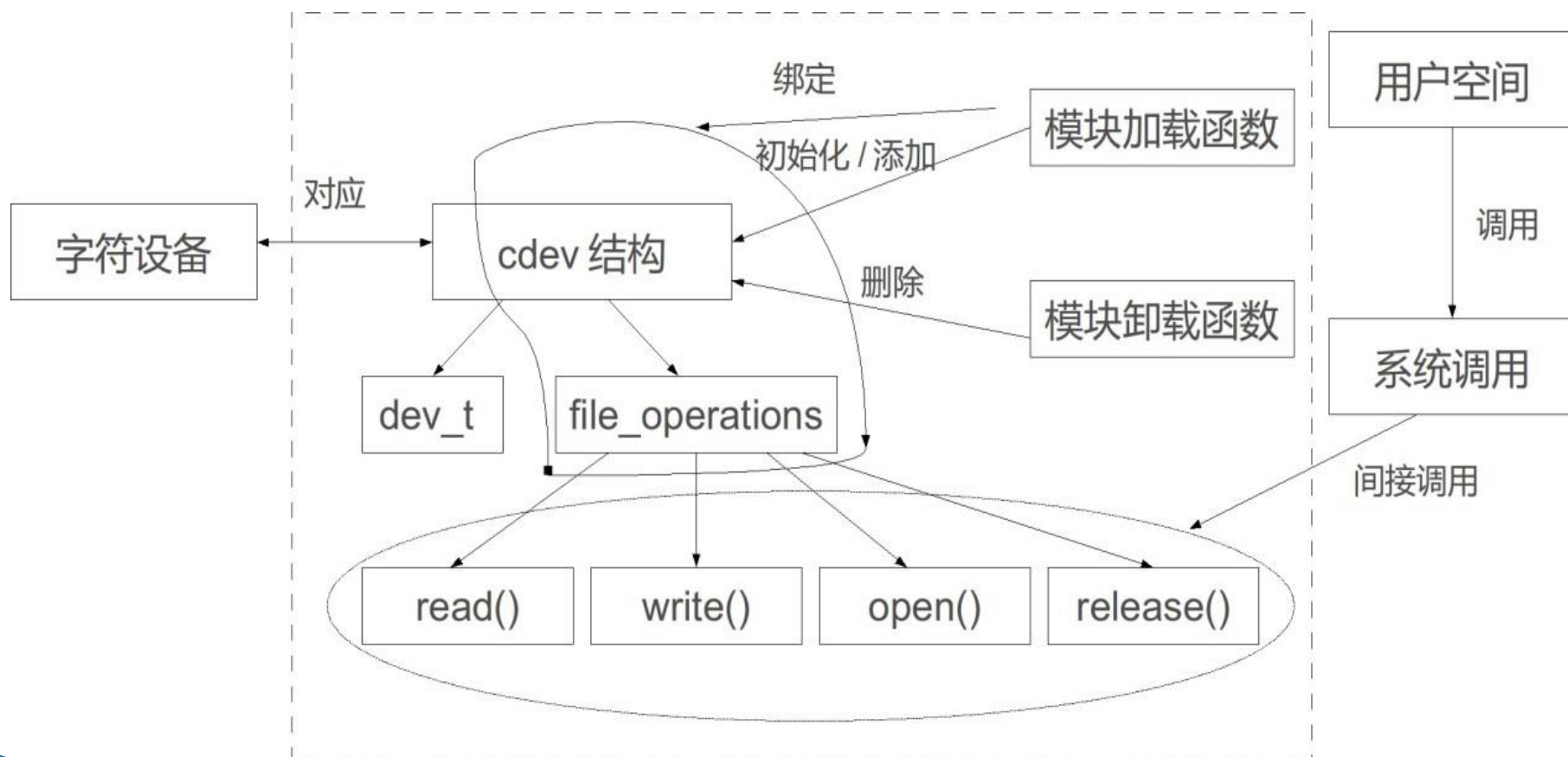
8.1.4 驱动程序的加载

通常 linux 驱动程序可通过两种方式进行加载：

一种是将驱动程序编译成模块形式进行动态加载，常用命令有insmod（加载）、rmmod（卸载）等；

另一种是静态编译，即将驱动程序直接编辑放进内核。动态加载模块设计使 Linux 内核功能更容易扩展。而静态编译方法对于在要求硬件只是完成比较特定、专一的功能的一些嵌入式系统中，具有更高的效率。

字符设备、字符设备驱动与用户空间访问该设备的程序三者之间的关系

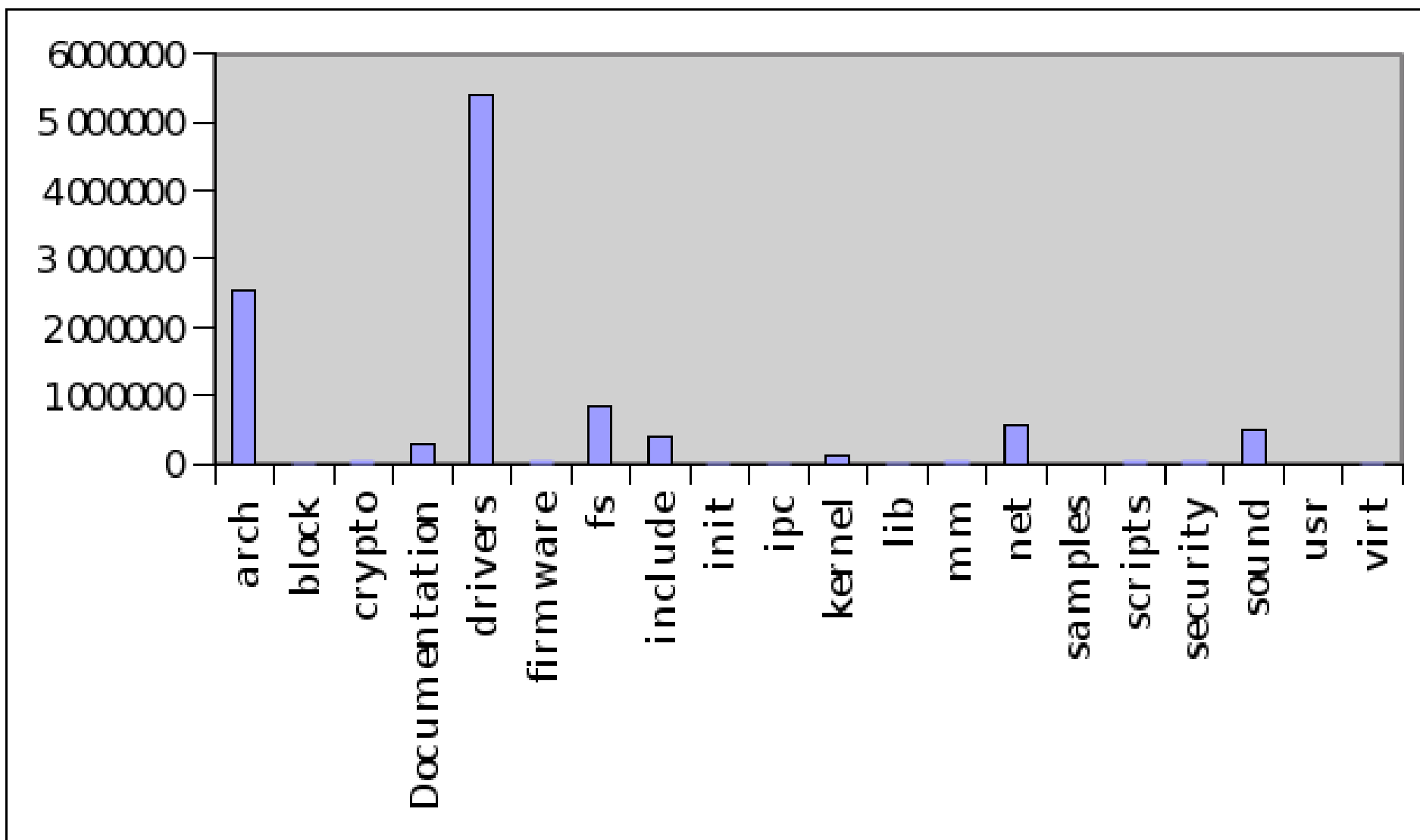


- 如图，在Linux内核中：
 - a -- 使用cdev结构体来描述字符设备；
 - b -- 通过其成员dev_t来定义设备号（分为主、次设备号）以确定字符设备的唯一性；
 - c -- 通过其成员file_operations来定义字符设备驱动提供给VFS的接口函数，如常见的open()、read()、write()等；
- 在Linux字符设备驱动中：
 - a -- 模块加载函数通过 register_chrdev_region() 或 alloc_chrdev_region() 来静态或者动态获取设备号；
 - b -- 通过 cdev_init() 建立cdev与 file_operations之间的连接，通过 cdev_add() 向系统添加一个cdev以完成注册；
 - c -- 模块卸载函数通过cdev_del()来注销cdev，通过 unregister_chrdev_region()来释放设备号；
- 用户空间访问该设备的程序：
 - a -- 通过Linux系统调用，如open()、read()、write()，来“调用” file_operations来定义字符设备驱动提供给VFS的接口函数；

8.2

Part Two

内核设备模型



linux内核各目录代码量对比

设备模型概述

- 设备模型提供独立的机制表示设备及其在系统中的拓扑结构

- ✓首先linux设备模型是一个具有清晰结构的组织所有设备和驱动的树状结构，用户就可以通过这棵树去遍历所有的设备，建立设备和驱动程序之间的联系；

- ✓其次，Linux驱动模型把很多设备共有的一些操作抽象出来，大大减少重复开发的可能；

- ✓再次，Linux设备模型提供了一些辅助的机制，比如引用计数，让开发者可以安全高效的开发驱动程序。同时，Linux设备模型还提供了一个非常有用的虚拟的基于内存的文件系统sysfs。Sysfs解释了内核数据结构的输出、属性以及它们之间及用户空间的连接关系。

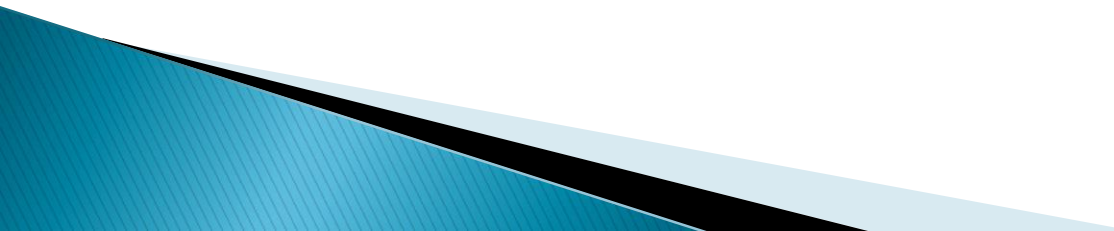
▶ 设备驱动模型基本概念：

设备驱动模型主要包含：类（class）、总线（bus）、设备（device）、驱动（driver），它们的本质都是内核中的几种数据结构的“实例”

- 类的本质是class结构体类型，各种不同的类其实就是class的各种实例
- 总线的本质是bus_type结构体类型，各种不同的总线其实就是bus_type的各种实例
- 设备的本质是device结构体类型，各种不同的设备其实就是device的各种实例
- 驱动的本质是device_driver结构体类型，各种不同的驱动其实就是device_driver的各种实例

8.2.1 设备模型功能

在Linux2.6 内核及后续版本中，设备模型为设备驱动程序管理、描述设备抽象数据结构之间关系等提供了一个有效的手段，其主要功能包括：

- (1) 电源管理和系统关机
 - (2) 与用户空间通信
 - (3) 热插拔(hotplug)设备管理
 - (4) 设备类型管理
 - (5) 对象生命周期处理
- 

8.2.2 sysfs

sysfs给用户提供了一个从用户空间去访问内核设备的方法，它在Linux里的路径是**/sys**。这个目录并不是存储在硬盘上的真实的文件系统，只有在系统启动之后才会建起来。**sysfs**是设备拓扑结构的文件系统表现。

可以使用**tree /sys**这个命令显示**sysfs**的结构。由于信息量较大，这里只列出第一层目录结构：

```
/sys
|-- block
|-- bus
|-- class
|-- dev
|-- devices
|-- firmware
|-- fs
|-- kernel
|-- module
`-- power
```

8.2.3 sysfs的实现机制kobject

“内核对象” (kernel object)的设备管理机制，该机制是基于一种底层数据结构，通过这个数据结构，可以使所有设备在底层都具有一个公共接口，便于设备或驱动程序的管理和组织。

kobject是Linux 2.6内核中由struct kobject表示。通过这个数据结构使所有设备在底层都具有统一的接口，kobject提供基本的对象管理，是构成Linux2.6设备模型的核心结构。它与sysfs文件系统紧密关联，每个在内核中注册的kobject对象都对应于sysfs文件系统中的目录。从面向对象的角度来说，kobject可以看作是所有设备对象的基类。

由于C语言并没有面向对象的语法，所以一般是把kobject内嵌到其他结构体里来实现类似的作用，这里的其他结构体可以看作是kobject的派生类。Kobject为Linux设备模型提供了很多有用的功能，比如引用计数，接口抽象，父子关系等等。

内核里的设备之间是以树状形式组织的，在这种组织架构里比较靠上层的节点可以看作是下层节点的父节点，反映到sysfs里就是上级目录和下级目录之间的关系。在内核里，kobject实现了这种父子关系。

8.2.4 设备模型的组织-platform总线

Platform 总线就是从2.6 内核开始引入的一种**虚拟总线**，主要用来管理 CPU 的片上资源，具有更好的**移植性**。相对于USB、PCI、I2C、SPI等物理总线来说，platform总线是一种虚拟、抽象出来的总线，实际中并不存在这样的总线。

那为什么需要platform总线呢？

其实是Linux设备驱动模型为了保持设备驱动的**统一性**而虚拟出来的总线。因为对于usb设备、i2c设备、pci设备、spi设备等等，他们与cpu的通信都是直接挂在相应的总线下面与我们的cpu进行数据交互的，但是在我们的嵌入式系统当中，并不是所有的设备都能够归属于这些常见的总线，在嵌入式系统里面，SoC系统中集成的独立的外设控制器、挂接在SoC内存空间的外设却不依附与此类总线。所以Linux驱动模型为了保持完整性，将这些设备挂在一条虚拟的总线上（platform总线），而不至于使得有些设备挂在总线上，另一些设备没有挂在总线上。

目前，大部分的驱动都是用Platform总线编写的,除了极少数情况之外如构建内核最小系统之内的而且能够采用CPU存储器总线直接寻址的设备。

Platform总线模型主要包括platform_device、platform_bus、platform_driver三个部分。

8.3

Part Three

字符设备驱动设计框架

8.3.1 字符设备的重要数据结构

字符设备驱动程序编写通常都要涉及到三个重要的内核数据结构，分别是**file_operations**结构体、**file**结构体和 **inode**结构体。

File_operations为用户态应用程序提供接口，是系统调用和驱动程序关联的重要数据结构。

File 结构体在内核代码 `include/linux/fs.h` 中定义，表示一个抽象的打开的文件，**file_operations** 结构体就是 **file** 结构的一个成员。

Inode 结构表示一个文件，而 **file** 结构表示一个打开的文件。这正是二者间最重要的关系。

每个进程为每个打开的文件分配一个文件描述符，每个文件描述符对应一个**file**结构，同一个文件被不同的进程打开后，在不同的进程中会有不同的**file**文件结构，其中包括了文件的操作方式(只读\只写\读写)，偏移量，以及指向**inode**的指针等等。这样，不同的**file**结构指向了同一个**inode**节点。

字符设备的分配和初始化有两种不同的方式。**cdev_alloc()**函数用于动态分配一个新的**cdev**结构体并初始化。一般如果建立新的**cdev**结构体可以使用该方式，这里给出一个参考代码：

```
struct cdev * my_cdev=cdev_alloc();  
my_cdev->owner=THIS_ MODULE;  
my_cdev->ops=&fops;
```

如果需要把cdev结构体嵌入到指定设备结构中，可以采用静态分配方式。cdev_init()函数可以初始化一个静态分配的cdev结构体，并建立 cdev 和file_operation 之间的连接。与cdev_alloc()唯一不同的是，cdev_init()函数用于初始化已经存在的cdev结构体。这里给出一段参考代码：

```
struct cdev my_cdev ;  
cdev_init(&my_cdev, &fops);  
my_cdev.owner=THIS_ MODULE;
```

8.3.2 字符设备驱动框架

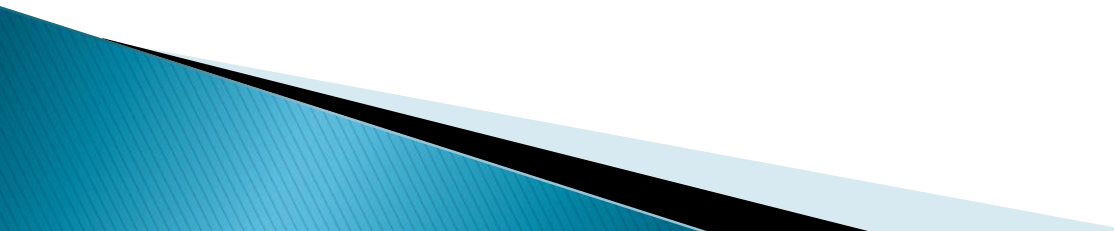
字符设备驱动程序的初始化流程一般可以用如下的过程来表示：

- （1）定义相关的设备文件结构体（如`file_operation()`中的相关成员函数的定义）。
- （2）向内核申请主设备号（建议采用动态方式）。
- （3）申请成功后，通过调用`MAJOR()`函数获取主设备号。
- （4）初始化`cdev`的结构体，可以通过调用`cdev_init()`函数实现。
- （5）通过调用`cdev_add()`函数注册`cdev`到内核。
- （6）注册设备模块，主要使用`module_init()`函数和`module_exit()`函数。

编写一个字符设备的驱动程序，首先要注册一个设备号。内核提供了三个函数来注册一组字符设备编号，这三个函数分别是：

- `alloc_chrdev_region()`
- `register_chrdev_region()`
- `register_chrdev()`。

其中`register_chrdev()`在上节已经介绍过。这里首先介绍的是 `alloc_chrdev_region()`函数，该函数用于动态申请设备号范围，通过指针参数返回实际分配的起始设备号。



Register_chrdev_region()函数用于向内核申请分配已知可用的设备号（次设备号通常为0）范围。下面是该函数原型：

```
int register_chrdev_region(dev_t from, unsigned count, const char *name)。
```

参数**from** 是要分配的设备号的 **dev_t** 类型数据，表示了要分配的设备编号的起始值，参数 **count** 表示了允许分配设备编号的范围。

register_chrdev()是一个老版本内核的设备号分配函数，不过新内核对其还是兼容的。**Register_chrdev()**兼容了动态和静态两种分配方式。**Register_chrdev()**不仅分配了设备号，同时也注册了设备。这是 **register_chrdev()**与前两个函数的最大区别。也就是说，如果使用 **alloc_chrdev_region()** 或 **register_chrdev_region()** 分配设备号，还需要对 **cdev** 结构体初始化。而**register_chrdev()**则把对 **cdev** 结构体的操作封装在了函数的内部。所以在一般的字符设备驱动程序中，不会看到对 **cdev** 的操作。

与注册分配字符设备编号的方法类似，内核提供了两个注销字符设备编号范围的函数

`unregister_chrdev_region()`和`unregister_chrdev()`。

这两个函数实际上都调用了

`__unregister_chrdev_region()`函数，原理是一样的。

`Register_chrdev()`函数封装了`cdev`结构的操作，而`alloc_chrdev_region()`或`register_chrdev_region()`只提供了设备号的注册，并未真正的初始化一个设备，只有 `cdev` 这个表示设备的结构体初始化了，才可以说设备初始化了。

这里举出字符设备驱动程序的常见的两种编程架构。

架构一：

```
static int __init xxx_init(void)
{
    ... register_chrdev(xxx_dev_no, DEV_NAME,&fops);
}
static void __exit xxx_exit(void)
{
    unregister_chrdev(xxx_dev_no, DEV_NAME);
    ...
}
module_init(xxx_init);
module_exit(xxx_exit);
```


架构二：

```
struct xxx_dev_t
{
    struct cdev cdev;
    ...
} xxx_dev;
static int __init xxx_init(void)
{
    ...
    cdev_init(&xxx_dev.cdev, &xxx_fops);
    xxx_dev.cdev.owner = THIS_MODULE;
    alloc_chrdev_region(&xxx_dev_no, 0, 1, DEV_NAME);
}
ret = cdev_add(&xxx_dev.cdev, xxx_dev_no, 1);
...
}
static void __exit xxx_exit(void)
{
    unregister_chrdev_region(xxx_dev_no, 1);
    cdev_del(&xxx_dev.cdev);
    ...
}
module_init(xxx_init),
module_exit(xxx_exit);
```

这两个结构中，前一个应用 `register_chrdev` 函数封装了 `cdev`，后面可以直接定义 `file_operations` 结构体提供系统调用接口。后一种架构用 `alloc_chrdev_region` 注册设备号，然后用 `cdev_init` 初始化了一个设备，接着用 `cdev_add` 添加了该设备。两种架构在模块卸载函数中，分别用相应的卸载函数实现。

当file_operations 结构与设备关联在一起后，就可以在驱动的架构中补全file_operations 的内容，实现一个完整的驱动架构，比如：

```
static unsigned int xxx_open()
{ ... }
static unsigned int xxx_ioctl()
{ ... }
struct file_operations fops = {
.owner = THIS_MODULE,
.open = xxx_open,
.ioctl = xxx_ioctl, //注意新式写法这里应是 .unlocked_ioctl= xxx_ioctl
};
static int __init xxx_init(void)
{
...
register_chrdev(xxx_dev_no, DEV_NAME,&fops);
}
static void __exit xxx_exit(void)
{
unregister_chrdev(xxx_dev_no, DEV_NAME);
...
}
module_init(xxx_init);
module_exit(xxx_exit);
```

8.4 Part Four

GPIO驱动概述

GPIO概述

GPIO（**General-Purpose Input/Output Ports**）全称是通用编程I/O端口。它们是CPU的引脚，可以通过它们向外输出高低电平，或者读入引脚的状态，这里的状态也是通过高电平或低电平来反应的，所以**GPIO**接口技术可以说是CPU众多接口技术中最为简单、常用的一种。

每个**GPIO**端口至少需要两个寄存器，一个是用于控制的“通用I/O端口控制寄存器”，一个是存放数据的“通用I/O端口数据寄存器”。控制和数据寄存器的每一位和**GPIO**的硬件引脚相对应，由控制寄存器设置每一个引脚的数据流向，数据寄存器设置引脚输出的高低电平或读取引脚上的电平。除了这两个寄存器以外，还有其它相关寄存器，比如上拉/下拉寄存器设置**GPIO**输出模式是高阻、带上拉电平输出还是不带上拉电平输出等。

S5PV210共有237个GPIO端口，分成15组：

- GPA0： 8输入/输出引脚。
- GPA1： 4输入/输出引脚。
- GPB： 8输入/输出引脚。
- GPC0： 5输入/输出引脚。
- GPC1： 5输入/输出引脚。
- GPD0： 4输入/输出引脚。
- GPD1： 6输入/输出引脚。
- GPE0、GPE1： 13输入/输出引脚。
- GPF0、GPF1、GPF2、GPF3： 30输入/输出引脚。
- GPG0、GPG1、GPG2、GPG3： 28输入/输出引脚。
- GPH0、GPH1、GPH2、GPH3： 32输入/输出引脚。
- GPP1： 低功率I2S、PCM。
- GPJ0、GPJ1、GPJ2、GPJ3、GPJ4： 35输入/输出引脚。
- MP0_1、MP_2、MP_3： 20输入/输出引脚。
- MP0_4、MP_5、MP_6、MP_7： 32输入/输出存储器引脚。

GPIO主要的相关寄存器：

➤**GPIO控制寄存器GPxnCON**。用于控制GPIO的引脚功能，向该寄存器写入数据来设置相应引脚是输入/输出，还是其它功能。该寄存器中每4位控制一个引脚，写入0000设置为输入IO口，从引脚上读入外部输入的数据；写入0001设置为输出IO口，向该位写入的数据被发送到对应的引脚上；写入其它值可设置引脚的第二功能，具体功能可查阅S5PV210处理器的芯片手册。

➤**GPIO数据寄存器GPxnDAT**。用于读写引脚的状态，即该端口的数据。当引脚被设置为输出引脚，写该寄存器的对应位为1，设置该引脚输出高电平，写入0设置该引脚输出低电平；当引脚被设置为输入引脚，读取该寄存器对应位中的数据可得到端口电平状态。

➤**GPIO上拉/下拉寄存器GPxnPUD**，用于控制每个端口上拉/下拉电阻的使能/禁止。对应位为0时，该引脚使用上拉/下拉电阻；对应位为1时，该引脚不使用上拉/下拉电阻。

➤**GPIO掉电模式上拉/下拉寄存器GPxnPUDPDN**，用于掉电模式下使用。每两位对应一个引脚，为00时输出0，01时输出1，10时为输入功能，11时保持原有状态。

GPIO的驱动主要作用就是读取**GPIO**口的内容，或者设置**GPIO**口的状态。

GPIO是与硬件体系密切相关，在linux内核目录下的相关文件中我们可以发现针对不同硬件芯片的**GPIO**定义和使用方法，如**S5PV210**芯片linux内核中也有相应的驱动程序支持（如在/**drivers/gpio/**）。当然，linux内核也提供了一个模型框架，能够使用统一的接口来操作**GPIO**，这个架构被称作"**gpiolib**"，系统通过**gpiolib.c**文件来描述该架构。说明文档可见**Documentation/gpio.txt**。

8.4.1 gpiolib关键数据结构

Gpiolib架构下最重要的数据结构是gpio_chip结构体和gpio_desc结构体。

gpio_chip结构体的部分定义如下：

```
Struct gpio_chip {
```

```
... ..
```

```
int (*request)(struct gpio_chip *chip, unsigned offset); //申请gpio资源;
```

```
void (*free)(struct gpio_chip *chip, unsigned offset); //释放gpio资源;
```

```
int (*direction_input)(struct gpio_chip *chip, unsigned offset); // 设置GPIO口方向的操作
```

```
int (*get)(struct gpio_chip *chip, unsigned offset);
```

```
int (*direction_output)(struct gpio_chip *chip, unsigned offset, int value); //设置GPIO口方向的操作
```

```
int (*set_debounce)(struct gpio_chip *chip, unsigned offset, unsigned debounce);
```

```
void (*set)(struct gpio_chip *chip, unsigned offset, int value); // 设置GPIO口高低电平值操作
```

```
int (*to_irq)(struct gpio_chip *chip, unsigned offset);
```

```
void (*dbg_show)(struct seq_file *s, struct gpio_chip *chip);
```

```
int base;
```

```
u16 ngpio;
```

```
... ..
```

```
};
```

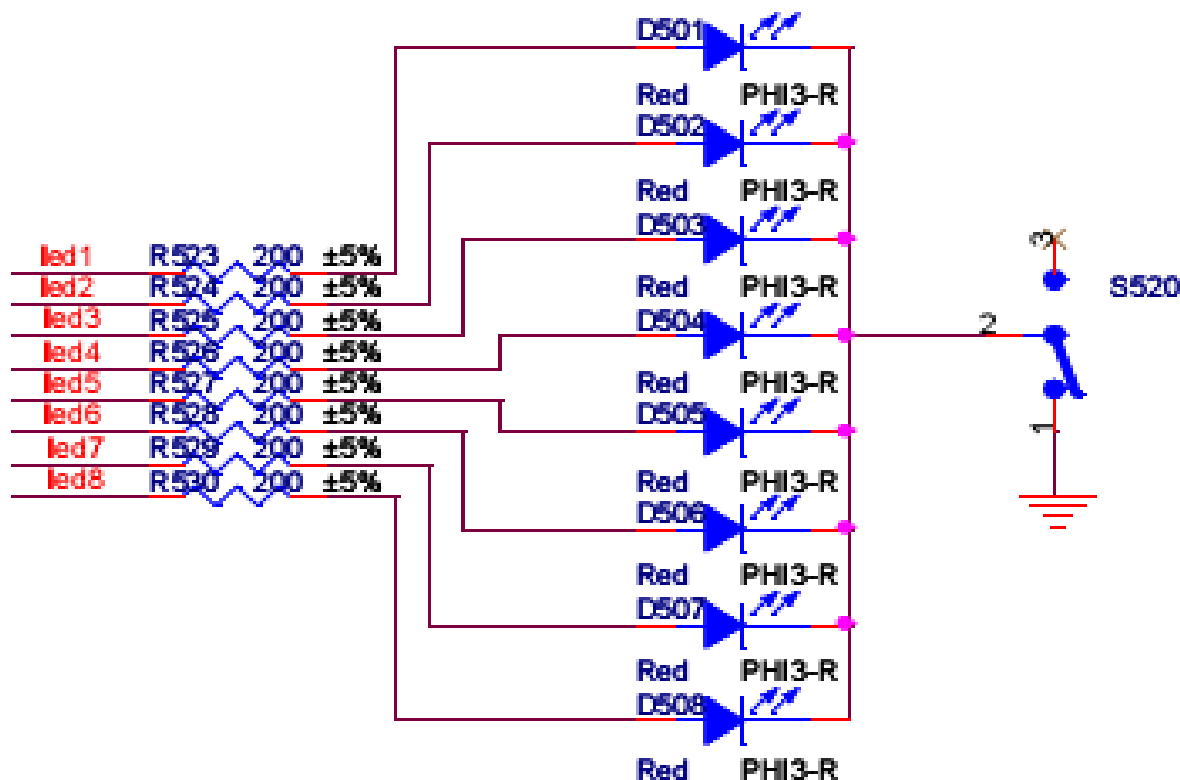
8.4.2 GPIO的申请和注册

GPIO的申请在gpiolib架构下是通过gpio_request_array函数实现的。这里的申请的主要标识就是检测GPIO描述符desc->flags的FLAG_REQUESTED标识，如果已申请该标识是1，否则是0，往往多个gpio会作为一个数组来进行申请。

当然，如前文所述，用户也可以使用linux系统已经支持的芯片GPIO接口驱动来完成目标系统设计。

一个LED灯的例子

IO口的操作是硬件控制的基础，本例子是一个最简单的IO口操作。8个LED灯分别用8个GPIO口来进行单独控制，通过I/O控制发光二极管的亮和灭。



GPIO与LED灯的连接

霍耳效应按键利用压电效应。压键时，晶体便产生电压。晶体在磁场中移动会产生电压，该电压经过放大电路放大。晶体切割磁力线会产生电压。

内含两个金属片和一个复位弹簧，按下时，两个金属片便被压在一起。结构简单成本低，缺点是容易产生抖动。

动。
小电

去是

多种多样

是几种常用的按钮：

- 机械式按钮
- 电容式按钮
- 薄膜式按钮
- 霍耳效应按钮

这是一种特殊的机械式按键开关，由三层塑料或橡胶夹层结构构成。

上面一层在每一行键下面有一条印制银导线，中间层在每一行键下面有一个小圆孔，下面一层在每一行键下面有一个小圆孔。

压键时，可活动的金属片向两块固定的金属片靠近，从而改变了两块固定的金属片之间的电容。此时，检测电容变化的电路就会产生一个逻辑电平信号以表示该键已被按下。

层
银

的
导线

键盘接口

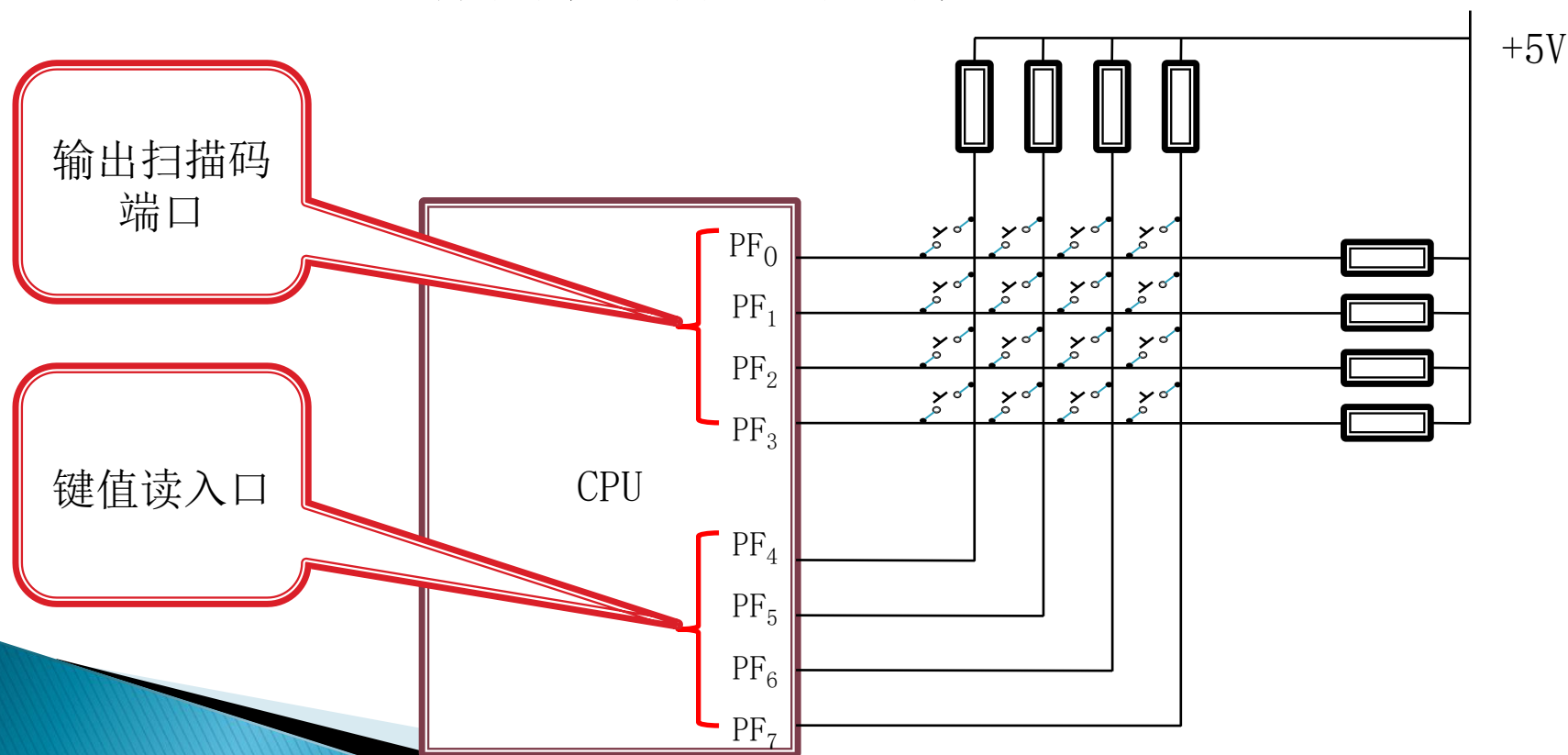
- ▶ 键盘有两种方案：

- 1、采用芯片实现键盘扫描；zlg7289
- 2、用软件实现键盘扫描。嵌入式控制器的功能很强，可充分利用这一资源。

计算机的键实际上就是开关，制造这种键的方法是多种多样的，以下是几种常用的按键：

用ARM芯片实现键盘接口

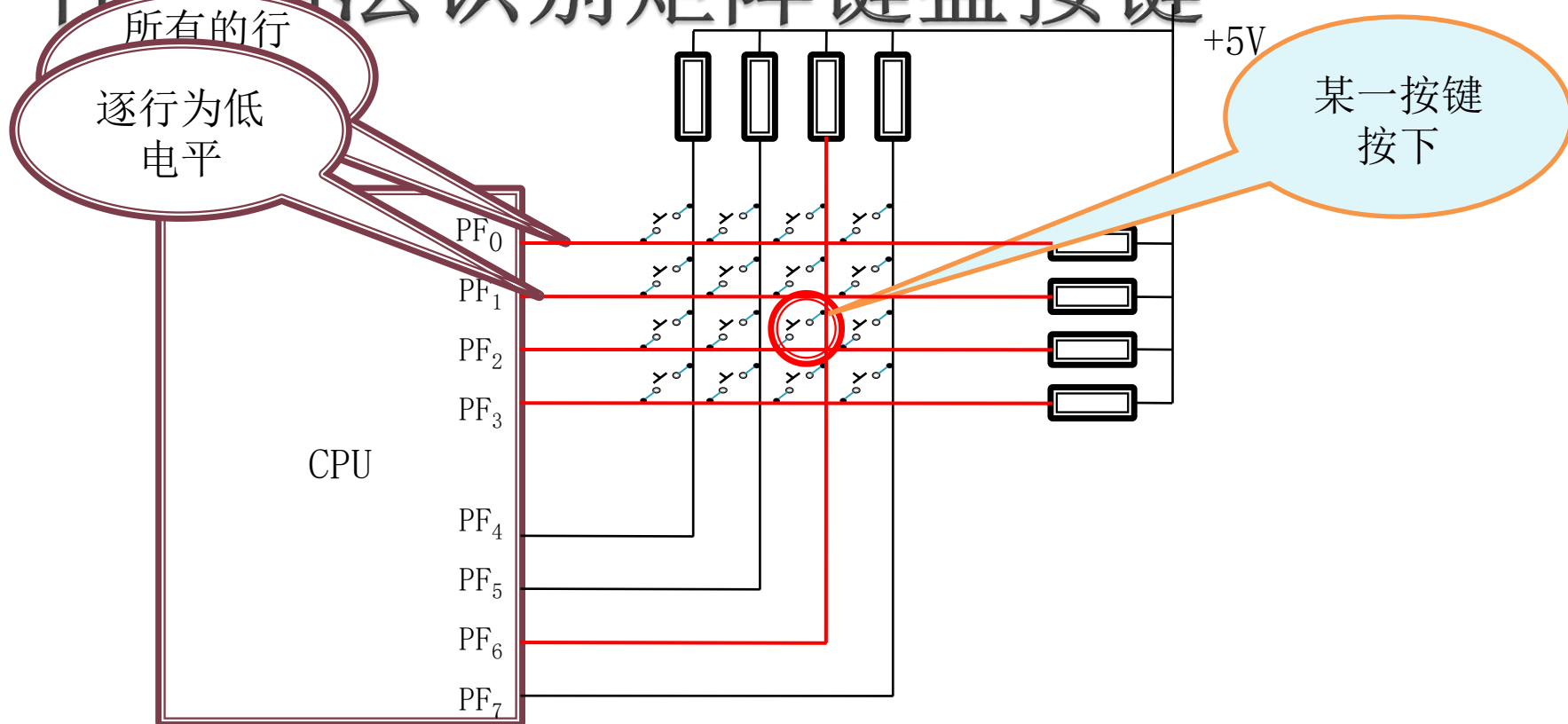
- ▶ 与4X4的矩阵键盘接口，采用“行扫描法”方法来检测键盘，只需要8根口线，
- ▶ 选取PF口作为检测键盘用端口，



矩阵键盘按键的识别方法

- ①识别键盘哪一系列的键被按下：让所有行线均为低电平，检查各列线电平是否为低，如果有列线为低，则说明该列有键被按下，否则说明无键被按下。
- ②如果某列有键被按下，识别键盘哪一行的键被按下：逐行置低电平，并置其余各行为高电平，检查各列线电平的变化，如果列电平变为低电平，则可确定此行此列交叉点处按键被按下。

行扫描法识别矩阵键盘按键

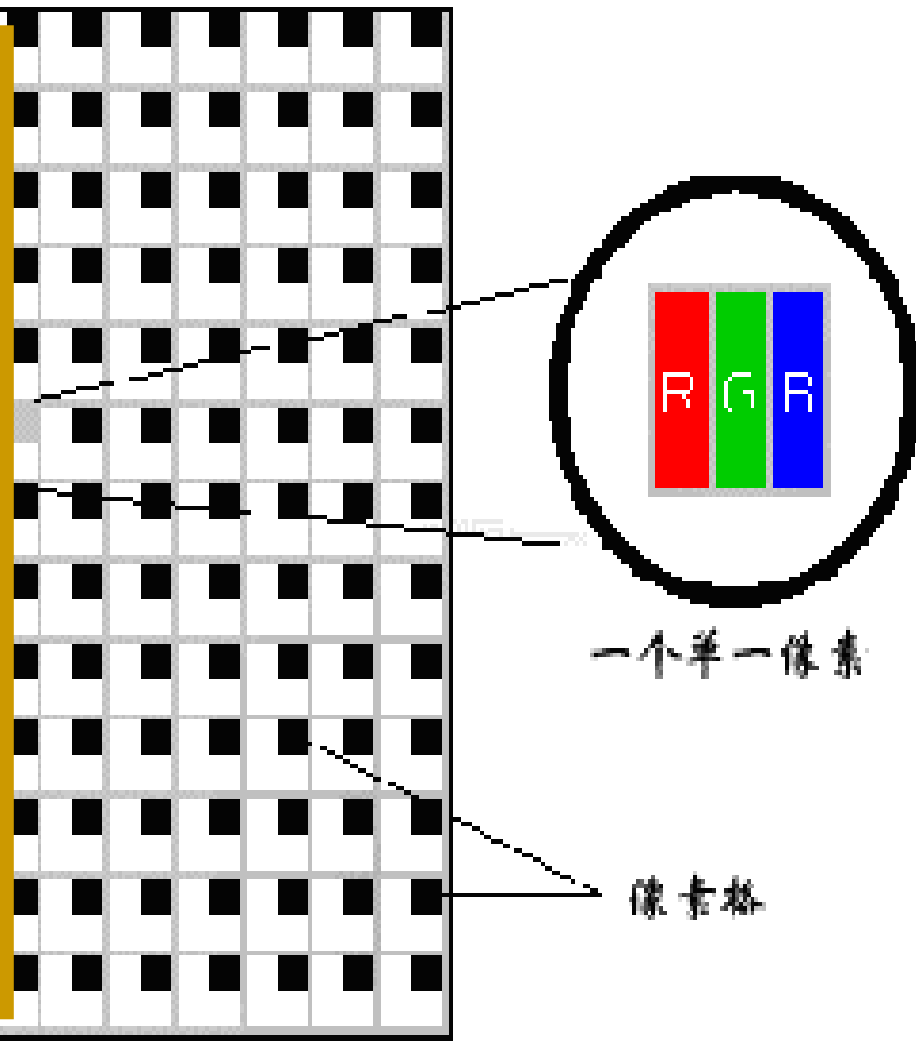


LCD接口概述

- ▶ 液晶显示是一种被动的显示，它不能发光，只能使用周围环境的光。
- ▶ 基本原理是通过给不同的液晶单元供电，控制其光线的通过与否，从而达到显示的目的。
- ▶ LCD有三种显示方式：反射型，透射型和透反射型。
- ▶ 市面上出售的LCD有两种类型：
 - 一种是带有驱动电路的LCD显示模块，这种LCD可以方便地与各种低档单片机进行接口；
 - 另一种是LCD显示屏，没有驱动电路，需要与驱动电路配合使用

● LCD工作原理

LCD 的横截面很像是很多层三明治叠在一起。每面最外一层是透明的玻璃基体，玻璃基体中间就是薄膜电晶体。颜色过滤器和液晶层可以显示出红、蓝和绿三种最基本的颜色。通常，LCD后面都有照明灯以显示画面。



LCD帧缓冲

- ▶ 帧缓冲（framebuffer）是Linux 系统为显示设备提供的一个接口，它将显示缓冲区抽象，屏蔽图像硬件的底层差异，允许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。用户不必关心物理显示缓冲区的具体位置及存放方式，这些都由帧缓冲设备驱动本身来完成。
- ▶ framebuffer机制模仿显卡的功能，将显卡硬件结构抽象为一系列的数据结构，可以通过framebuffer的读写直接对显存进行操作。用户可以将framebuffer看成是显存的一个映像，将其映射到进程空间后，就可以直接进行读写操作，写操作会直接反映在屏幕上。

- ▶ framebuffer是个字符设备，主设备号为29，对应于/dev/fb%d 设备文件。
- ▶ 通常，使用如下方式（前面的数字表示次设备号）
 - 0 = /dev/fb0 第一个fb 设备
 - 1 = /dev/fb1 第二个fb 设备
- ▶ fb 也是一种普通的内存设备，可以读写其内容。例如，屏幕抓屏：`cp /dev/fb0 myfilefb` 虽然可以像内存设备（/dev/mem）一样，对其read, write, seek 以及mmap。但区别在于fb 使用的不是整个内存区，而是显存部分。

fb与应用程序的交互

- ▶ 对于用户程序而言，它和其他的设备并没有什么区别，用户可以把fb看成是一块内存，既可以向内存中写数据，也可以读数据。fb的显示缓冲区位于内核空间，应用程序可以把此空间映射到自己的用户空间，在进行操作。
- ▶ 在应用程序中，操作/dev/fbn的一般步骤如下：
 - (1) 打开/dev/fbn设备文件。
 - (2) 用ioctl()操作取得当前显示屏幕的参数，如屏幕分辨率、每个像素点的比特数。根据屏幕参数可计算屏幕缓冲区的大小。
 - (3) 用mmap()函数，将屏幕缓冲区映射到用户空间。
 - (4) 映射后就可以直接读/写屏幕缓冲区，进行绘图和图片显示了。

触摸屏

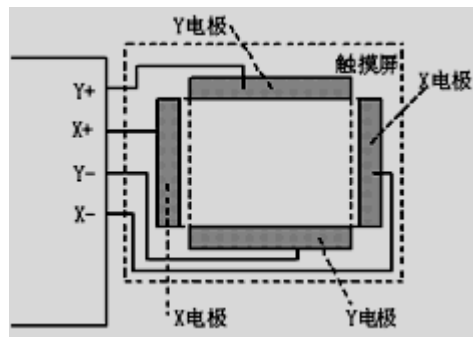
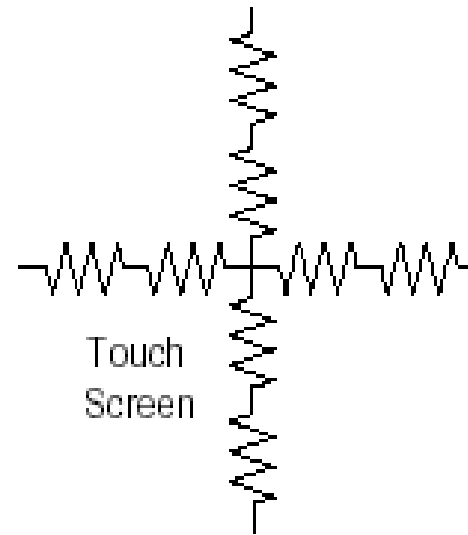
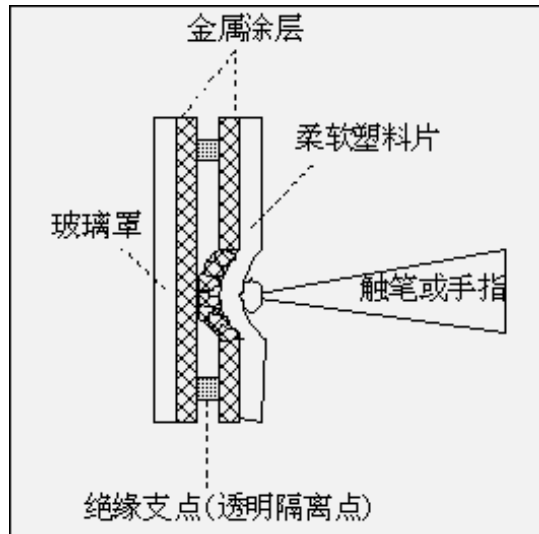
- ▶ 触摸屏是一种历史悠久的外部设备，早在上世纪60年代触摸屏就开始在一些公共场合得到使用。目前已经成为重要的嵌入式输入设备，广泛应用于个人自助存款取款机、PDA、媒体播放器、汽车导航器、智能手机、医疗电子设备等，使用频度仅次于键盘和鼠标。触摸屏不是独立的输入设备，它必须安装在液晶屏上，在液晶屏显示数据时才能使用。



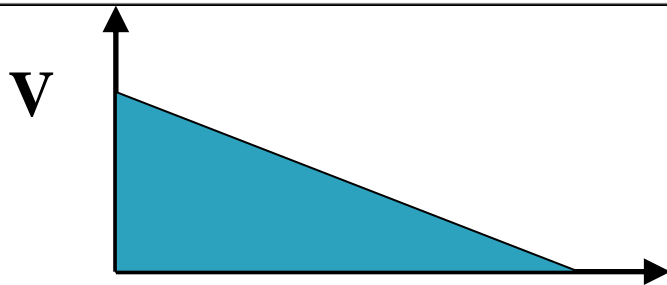
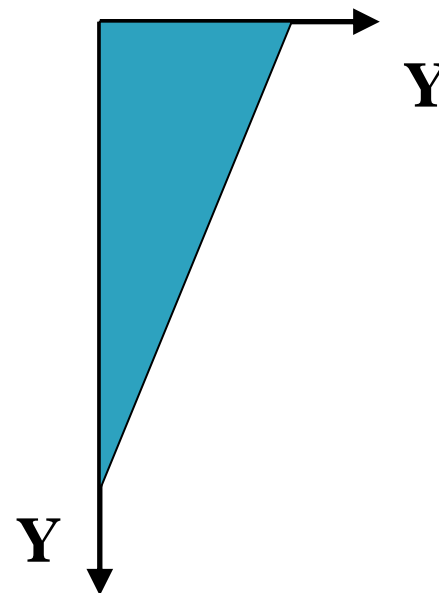
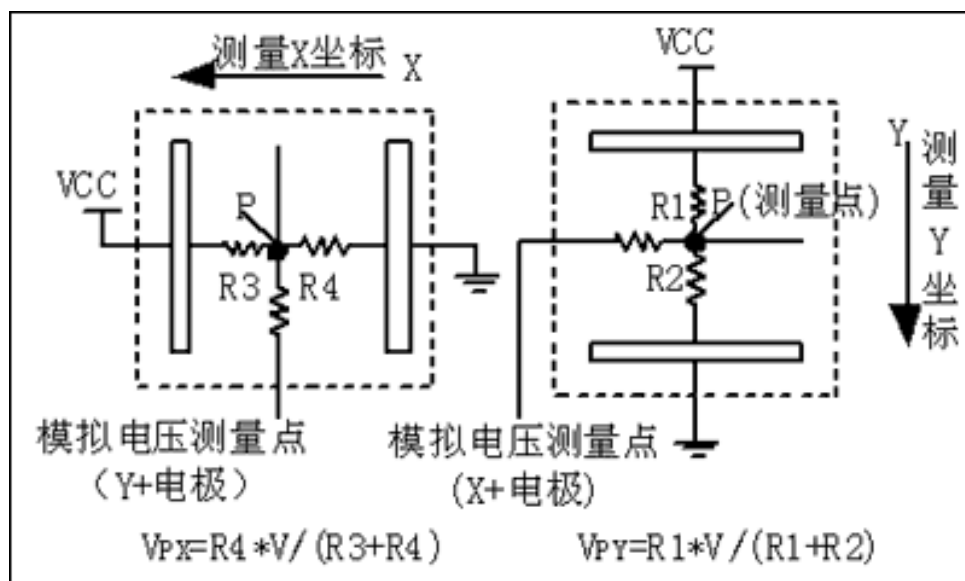
触摸屏的工作原理

- ▶ 主流触摸屏按其工作原理的不同主要可分为电阻式触摸屏和电容式触摸屏，其它还有表面声波触摸屏、红外式触摸屏等。
- ▶ 最常见的是电阻式触摸屏，其屏体部分是一块与显示器表面非常配合的多层复合薄膜。触摸屏工作时，上下导体层相当于电阻网络。当某一层电极加上电压时，会在该网络上形成电压梯度。如有外力使得上下两层在某一点接触，则在另一层未加电压的电极上可测得接触点处的电压，从而知道接触点处的坐标。

四线电阻触摸屏原理

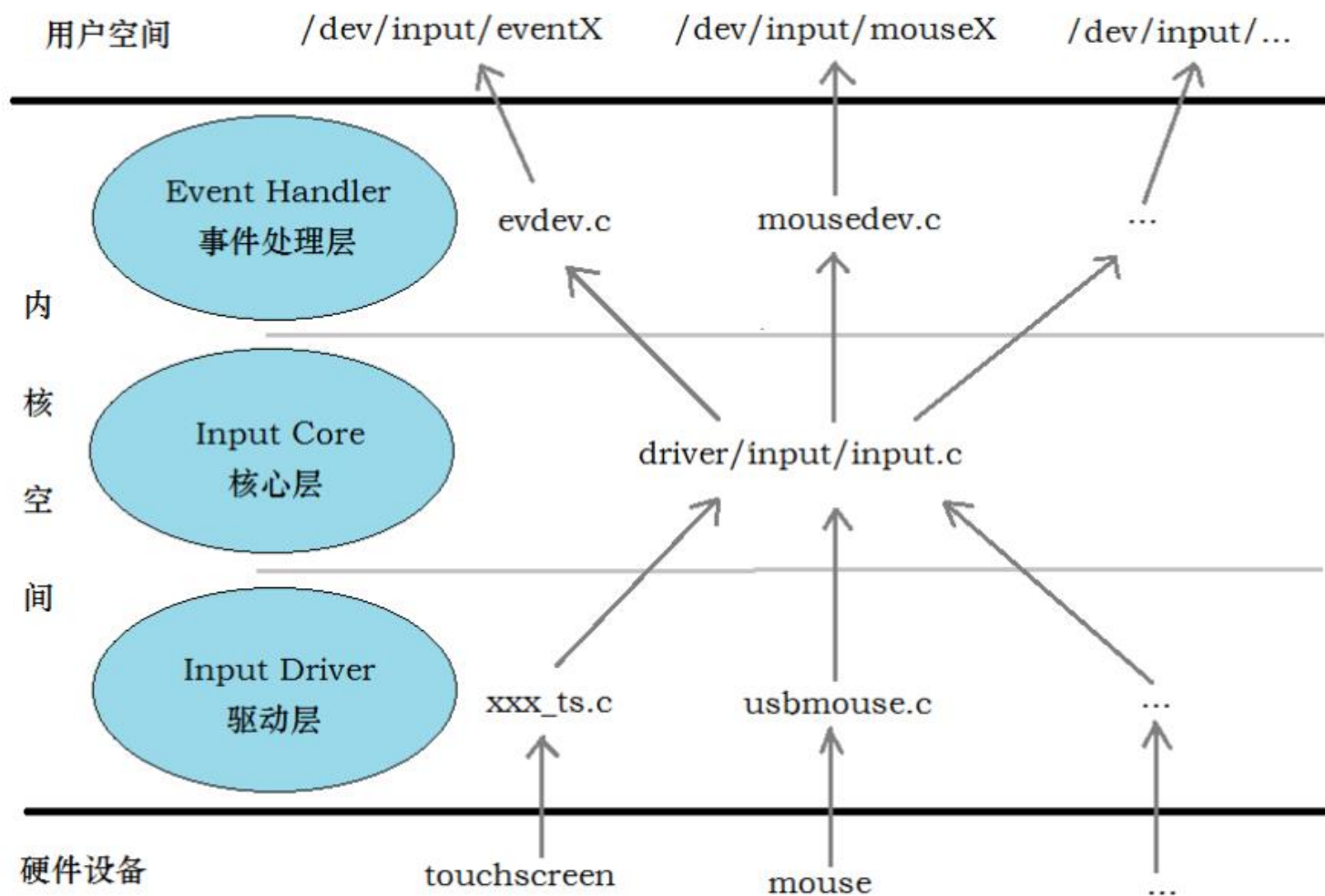


测量原理



- 在触摸点X、Y坐标的测量过程中，测量电压与测量点的等效电路图所示，图中P为测量点

Linux输入子系统(Input Subsystem)



输入子系统设备驱动层实现原理：

- ▶ 在Linux中，Input设备用input_dev结构体描述，定义在input.h中。设备的驱动只需按照如下步骤就可实现了。
 - ①在驱动模块加载函数中设置Input设备支持input子系统的哪些事件；
 - ②将Input设备注册到input子系统中；
 - ③在Input设备发生输入操作时(如：键盘被按下/抬起、触摸屏被触摸/抬起/移动、鼠标被移动/单击/抬起时等)，提交所发生的事件及对应的键值/坐标等状态。

- ▶ Linux中输入设备的事件类型有(这里只列出了常用的一些, 更多请看linux/input.h中):

EV_SYN	0x00	同步事件
EV_KEY	0x01	按键事件
EV_REL	0x02	相对坐标(如: 鼠标移动, 报告的是相对最后一次位置的偏移)
EV_ABS	0x03	绝对坐标(如: 触摸屏和操作杆, 报告的是绝对的坐标位置)
EV_MSC	0x04	其它

代码示例：

```
while(1)
{
    rb = read(fd, &ev, sizeof(struct input_event)); // 读取设备
    if (rb < (int)sizeof(struct input_event)) // 读取错误
    {
        perror("read error");
        exit(1);
    }
    if (EV_ABS==ev.type) // 读取按键内容
    {
        printf("event=%s,value=%d\n",ev.code==ABS_X?"ABS_X":ev.code==ABS_Y
            ?"ABS_Y":ev.code==ABS_PRESSURE?"ABS_PRESSURE":"UNKNOWN",ev.value);
    }else{
        printf("not ev_abs\n");
    }
}
```

测试结果：（触笔按下触摸屏）

event=ABS_X, value=505

event=ABS_Y, value=334

event=ABS_PRESSURE, value=1

8.5

Part Five

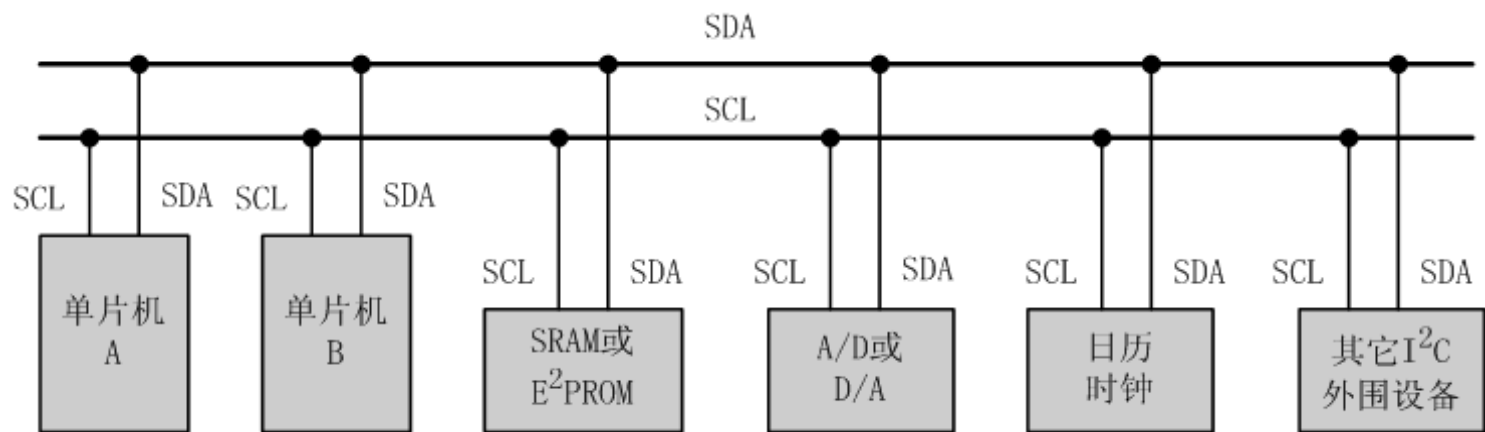
IIC总线驱动设计

串行总线概述

- ▶ 串行相比于并行的主要优点是要求的线数较少，通常只需要使用2条、3条或4条数据/时钟总线连续传输数据
 - RS232
 - RS485
 - I²C
 - SPI
 - SMBUS

8.5.1 IIC 总线概述

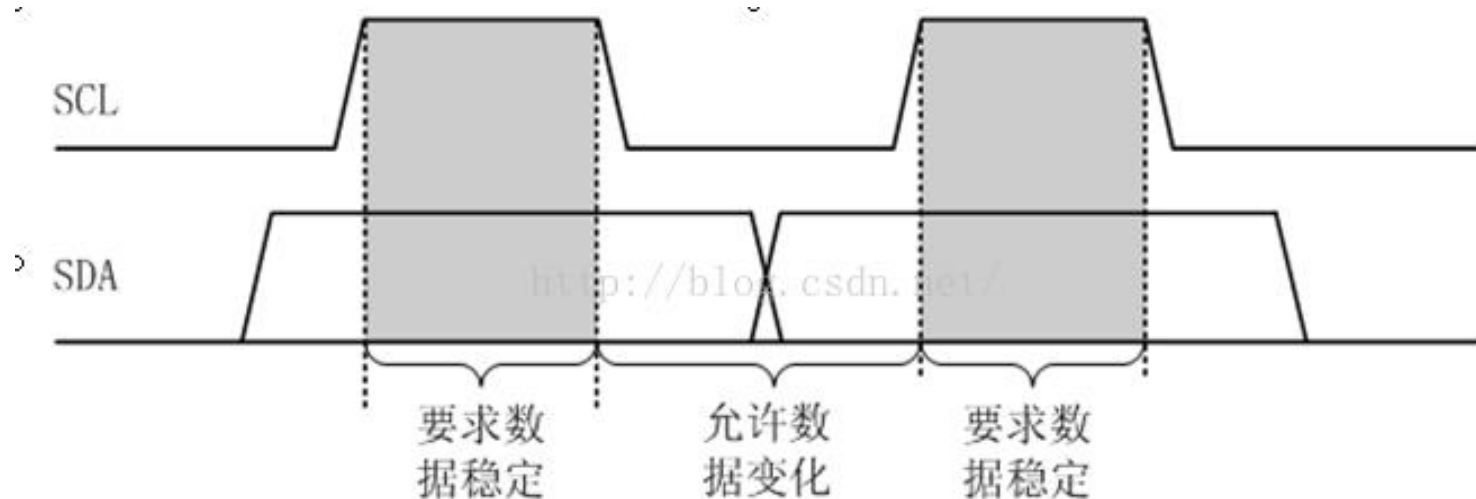
- ▶ Philips公司开发的二线式串行总线标准，内部集成电路（Internal Integrated Circuit），主要用于连接微控制器和外围设备。
 - 在标准模式下，位速率可以达到100Kbit/s
 - 在快速模式下则是400Kbit/s
 - 在高速模式下可以达到3.4Mbit/s
- ▶ I²C总线是由串行数据信号线SDA (Serial Data) 和串行时钟信号线SCL (Serial Clock) 构成的串行总线，可发送和接收数据。
- ▶ 采用该总线连接的设备工作在主/从模式下，主器件既可以作为发送器，也可以作为接收器
- ▶ I²C总线最主要的特点是它的简单性和高效性。
- ▶ 通信特征：串行、同步、半双工、低速率



IIC的协议层才是掌握IIC的关键。现在简单概括如下：

a.数据的有效性

在时钟的高电平周期内，SDA线上的数据必须保持稳定，数据线仅可以在时钟SCL为低电平时改变。

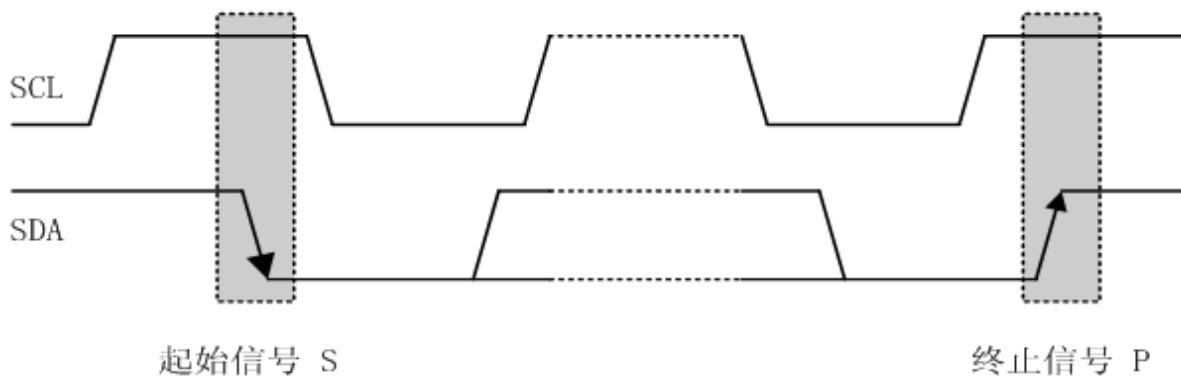


b.起始和结束信号

起始信号（START）：当SCL为高电平的时候，SDA线上由高到低的跳变被定义为起始信号

结束信号（STOP）：当SCL为高电平的时候，SDA线上由低到高的跳变被定义为停止信号

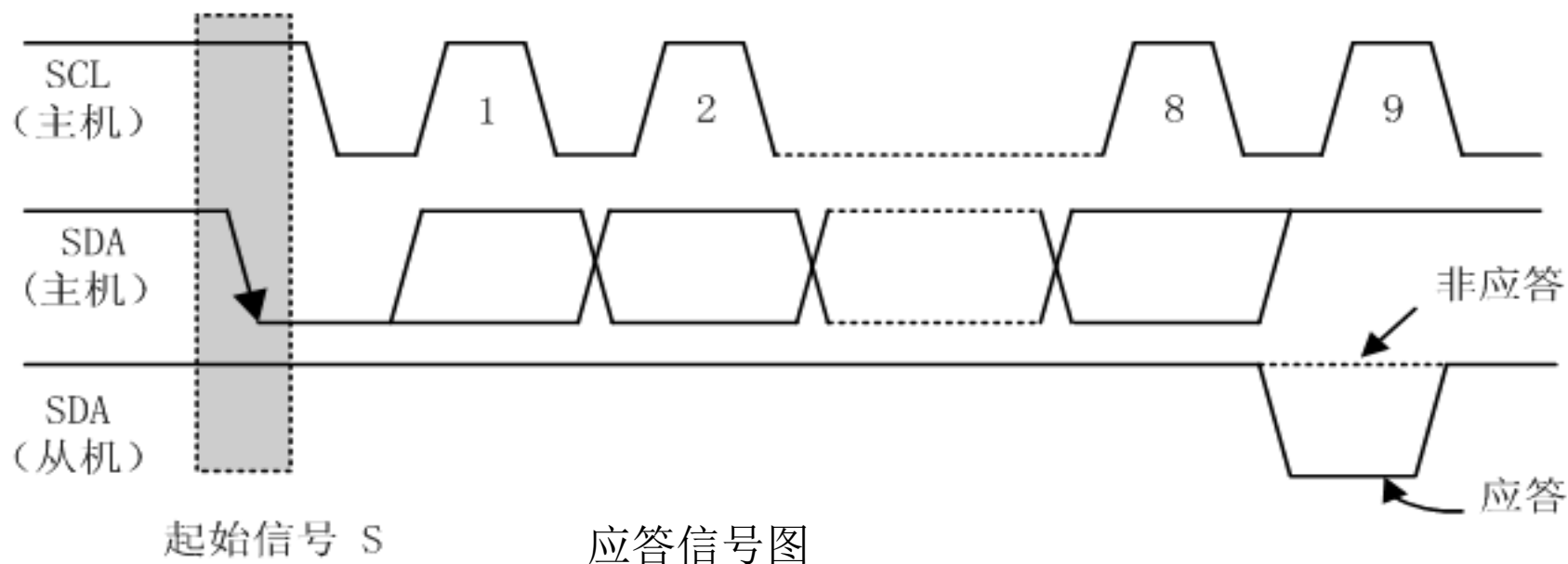
要注意起始和终止信号都是由主机发出的，连接到I2C总线上的器件，若具有I2C总线的硬件接口，则很容易检测到起始和终止信号。总线在起始信号之后，视为忙状态，在停止信号之后被视为空闲状态



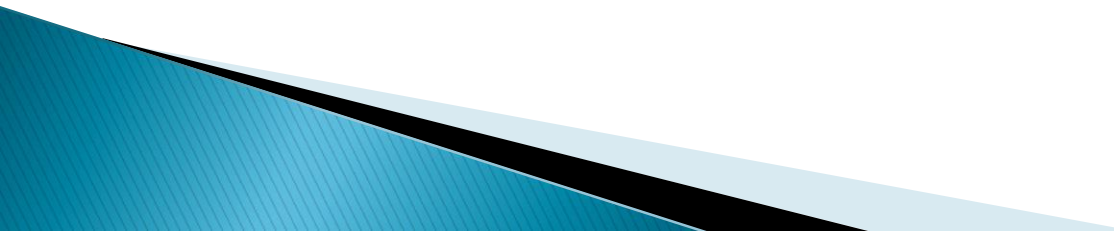
起始信号与终止信号

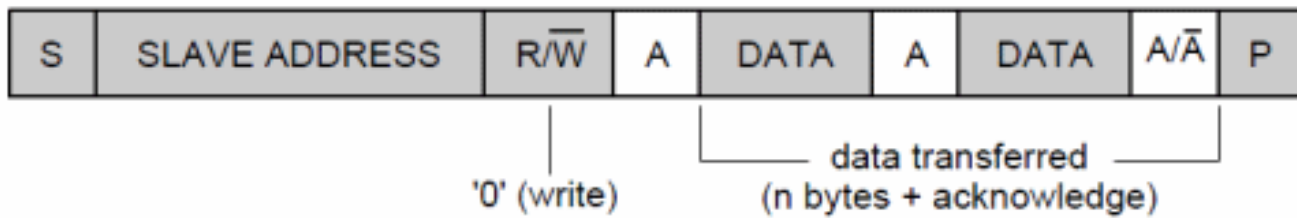
c. 应答和非应答信号


每当主机向从机发送完一个字节的数据，主机总是需要等待从机给出一个应答信号，以确认从机是否成功接收到了数据，从机应答主机所需要的时钟仍是主机提供的，应答出现在每一次主机完成8个数据位传输后紧跟着的时钟周期，低电平0表示应答(ACK)，1表示非应答(NACK)




非应答信号NACK:

- 1.从机没有准备好接收数据（在从机地址后）。
 - 2.从机接收的数据不能识别。
 - 3.在传输过程中，从机不能再接收数据（在第一个字节后）。
 - 4.主机读数据时，主机向从机表明已读完最后一个数据字节。
- 



 from master to slave

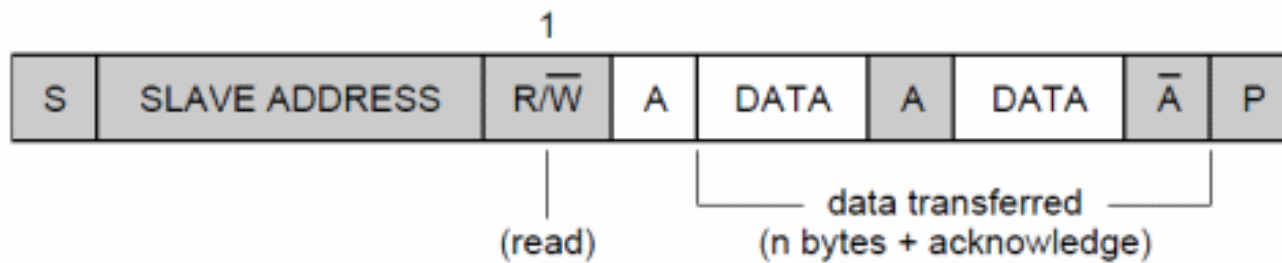
 from slave to master

A = acknowledge (SDA LOW)

\bar{A} = not acknowledge (SDA HIGH)

S = START condition

P = STOP condition



典型的I2C通信数据帧格式

所有的 **IIC** 总线上的数据帧格式均有如下这些特点：

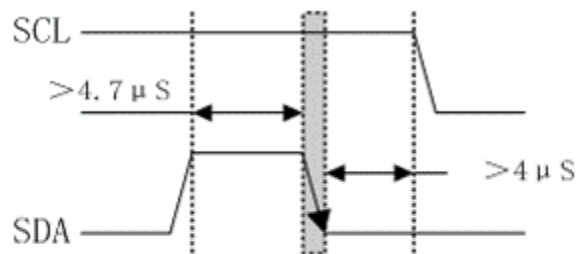
（1）无论何种方式，起始停止，寻址字节都由主控制器发送，数据字节的传送方向则遵循寻址字节中的方向位的规定。

（2）寻址字节只表明器件地址及传送方向，器件内部的 **N** 个数据地址由器件设计者在该器件的 **IIC** 总线数据操作格式化中指定第一个数据字节作为器件内的单元地址(**SIBADR**)数据，并且设置地址自动加减功能，以减少单元地址寻址操作。

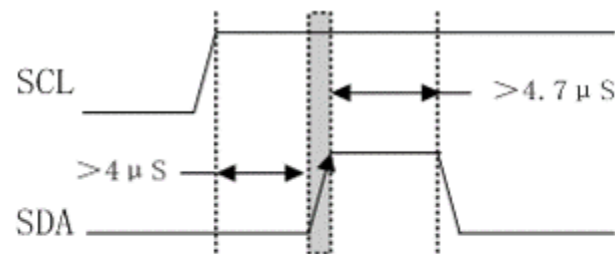
（3）每个字节传送都必须有应答信号相随。

（4）**IIC** 总线被控器在接收到起始信号后都必须复位它们的总线逻辑，以便对将要开始的被控器地址的传送进行预处理。

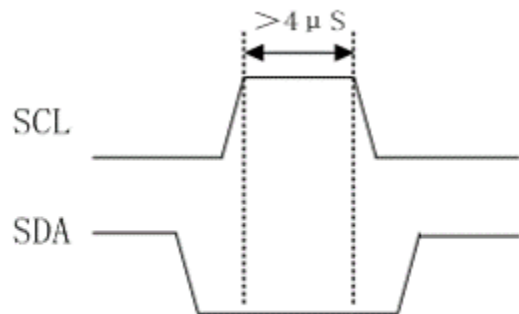
I2C总线各种信号的持续时间要求:



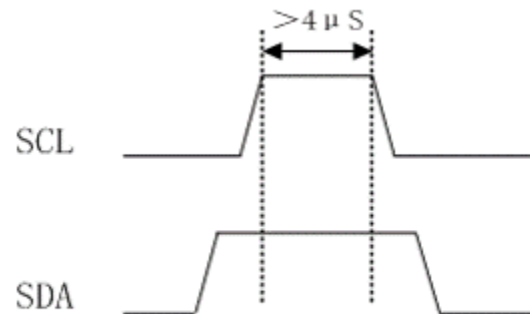
起始信号 S



终止信号 P



应答/“0”



非应答/“1”

<http://blog.pjromaniac.com/>

//产生起始信号

void I2C_Start(void)

{

 I2C_SDA_OUT();//配置一下引脚,引脚设置为输出

 I2C_SDA_H;//把数据线拉高

 I2C_SCL_H;//把时钟线拉高

 delay_us(5);//延时5微秒,要求大于4.7微秒

 I2C_SDA_L; //拉低, 产生下降沿

 delay_us(6);//这个过程大于4微秒

 I2C_SCL_L;//最后一定要把这个时钟线拉低, 因为只有时钟线拉低的时候才允许数据变化。

}

Linux中的时间

(1) HZ: Linux核心每隔固定周期会发出timer interrupt (IRQ 0), HZ是用来定义每一秒有几次timer interrupts。举例来说, HZ为1000, 代表每秒有1000次timer interrupts。 HZ可在编译核心时设定, 在内核的CONFIG_HZ中定义。

(2) Tick: Tick是HZ的倒数, 意即timer interrupt每发生一次中断的时间。如HZ为250时, tick为4毫秒(millisecond)。

(3) Jiffies: 每发生一次timer interrupt, Jiffies变数会被加一。jiffies记录了系统启动以来, 经过了多少tick。

(4) 全局变量xtime:xtime是从cmos电路中取得的时间, 一般是从某一历史时刻开始到现在的时间, 也就是为了取得我们操作系统上显示的日期。这个就是所谓的“实时时钟”, 它的精确度是微秒。

(5) RTC:这是一个硬件时钟, 用来持久存放系统时间, 系统关闭后靠主板上的微型电池保持计时。系统启动时, 内核通过读取RTC来初始化Wall Time, 并存放在xtime变量中, 这是RTC最主要的作用。

Linux中的时钟源

系统中时钟硬件有精度高的有精度低的。Linux操作系统抽象出了clocksource（时钟源）来管理这些硬件。Linux会在所有的硬件时钟中选择出精度最高作为当前在用的时钟源。

查看当前所有的可用的时钟源已经当前在用的时钟源：

```
$ cat /sys/devices/system/clocksource/clocksource0/available_clocksource  
tsc hpet acpi_pm
```

TSC

TSC是Time Stamp Counter。CPU 执行指令需要一个外部振荡器经过倍频后的时钟信号。x86 提供了一个 TSC 寄存器，该寄存器的值在每次收到一个时钟信号时加1，ARM中采用**PMCCNTR_ELO寄存器**。比如 CPU 的主频为1GHZ，则每一秒时间内，TSC 寄存器的值将增加 1G 次，或者说每一个纳秒加一次。可用dmesg|grep clock命令查看相关信息。

在Linux Driver开发中，经常要用到延迟函数：

sleep; usleep;

mdelay; udelay;ndelay。

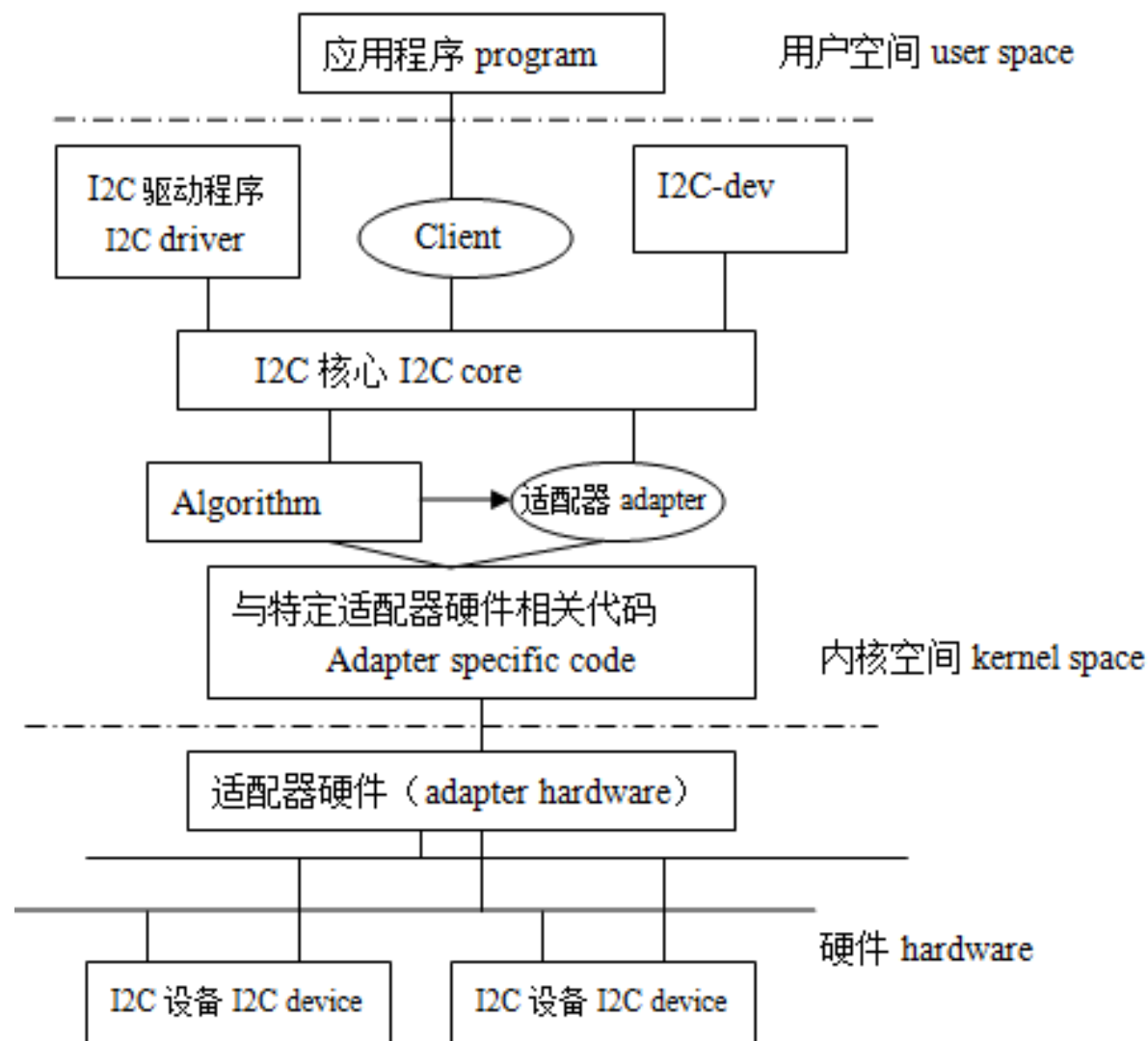
虽然sleep和mdelay都有延迟的作用，但他们是有区别的。

（1）mdelay是忙等待函数，在延迟过程中无法运行其他任务。这个延迟的时间是准确的。是需要等待多少时间就会真正等待多少时间。sleep是休眠函数，它不涉及忙等待。你如果是sleep(10)，那实际上延迟的时间，大部分时候是要多于10s的，是个不定的时间值。

（2）对于系统：mdelay() 会占用cpu资源，导致其他功能此时也无法使用cpu资源。sleep() 则不会占住cpu资源，其他模块此时也可以使用cpu资源。delay函数是忙则等待，占用CPU时间；而sleep函数使调用的进程进行休眠。

（3）mdelay, udelay, ndelay是内核延时函数。

8.5.2 IIC 驱动程序框架



I2C 总线驱动分成 3 个层次：I2C-core，I2总线驱动-Adapter，I2C 设备驱动-Client。

其中的 **I2C-core**（I2C核心）是 I2C 总线驱动程序体系结构的核心，它将 I2C 总线驱动体系分成包括它自身在内的 3 个层次。**I2C-core**为总线设备驱动提供统一的接口，通过这些接口来访问在特定 I2C 设备驱动程序中实现的功能，并实现从 IIC 总线驱动体系结构中添加和删除总线驱动的方法等。

Adapter 层代表 **I2C** 总线驱动即 **I2C** 适配器驱动，**Adapter**层是各个适配器驱动所构成的层，主要实现各相应适配器数据结构 **I2C_adapter**的具体的通信传输算法(**I2C_algorithm**)，此算法管理 **I2C** 控制器及实现总线数据的发送接收等操作。

Client 层则代表挂载在 **I2C** 总线上的设备驱动，**Client** 层是各个**I2C**设备构成的层，主要实现各描述**I2C** 设备的数据结构 **I2C_client** 及其私有数据结构，并通过 **I2C-core**提供的接口实现设备的注册，提供设备可使用的地址范围及地址检测成功后的回调函数。

在实际设计中，**I2C**核心提供的接口不需要修改，只需针对目标总线适配器驱动和设备驱动进行必要修改即可。

8.6

Part Six

块设备驱动程序设计概述

块设备

- ▶ 块设备是Linux系统中的一大类设备，包括IDE硬盘、SCSI硬盘、CD-ROM等设备
- ▶ 块设备数据存取的单位是块，块的大小通常为512字节到32K字节不等；块设备每次能传输一个或多个块，支持随机访问，并采用了缓存技术
- ▶ 块设备驱动主要针对磁盘等慢速设备，由于其支持随机访问，所以文件系统一般采用块设备作为载体

块设备（**block device**）是一类具有一定结构的随机存取设备，对这种设备的读写是按块进行的，为了使高速的**CPU**同低速块设备能够协调工作，提高读写效率，操作系统设计了缓冲机制。当进行读写的时候首先对缓冲区读写，只有缓冲区中没有需要读的数据或是需要读写的数据没有空间写时，才真正启动设备控制器去控制设备本身进行数据交换，而对于设备本身的数据交换同样也是通过缓冲区进行。

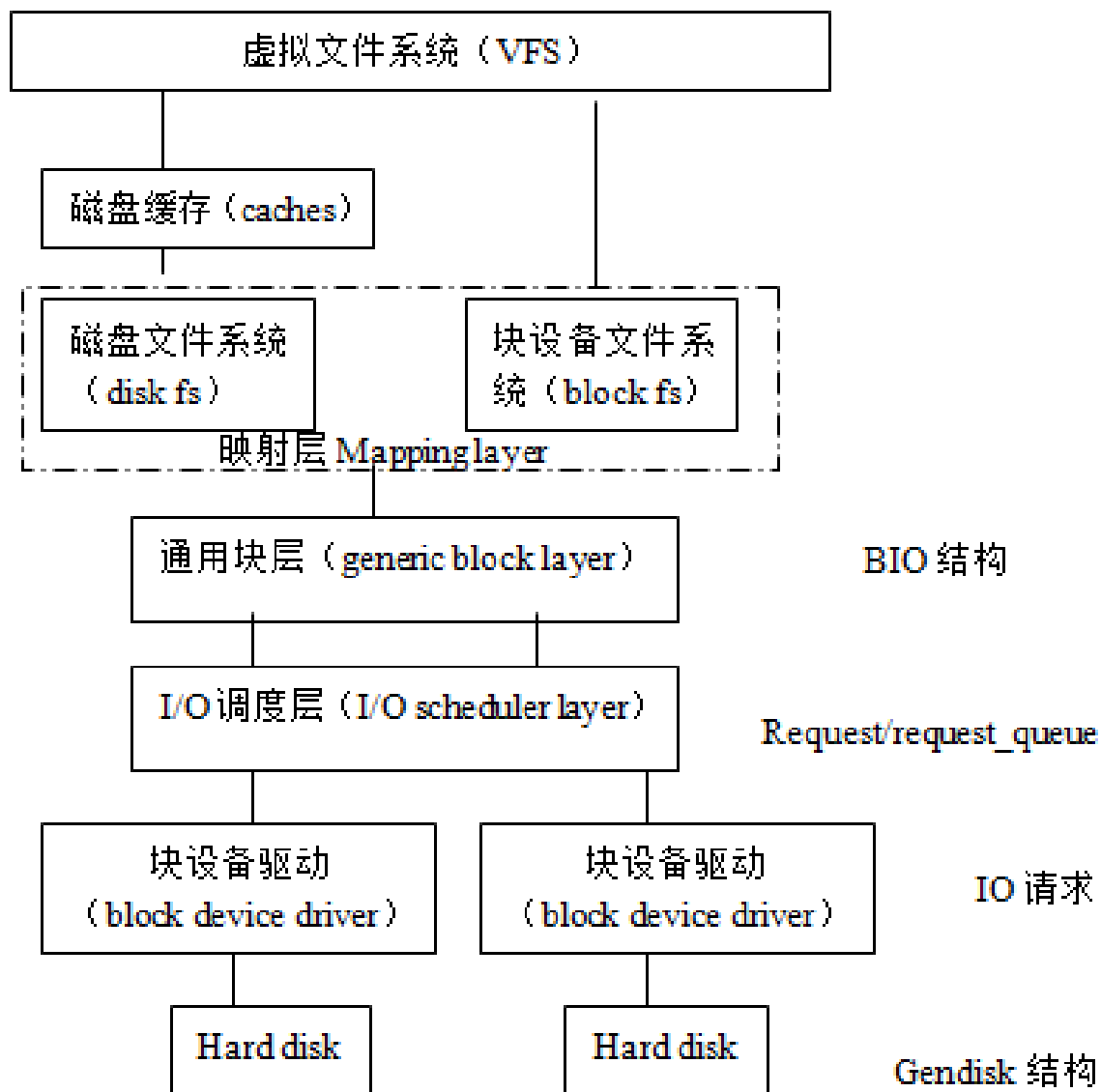
块设备主要涉及的三个存储概念：扇区、块和段。

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。（对设备而言）

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。

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

8.6.1块设备驱动整体框架



- 在Linux中，驱动对块设备的输入或输出(I/O)操作，都会向块设备发出一个请求，在驱动中用request结构体描述。
- 但对于一些磁盘设备而言请求的速度很慢，这时候内核就提供一种队列的机制把这些I/O请求添加到队列中（即请求队列），在驱动中用request_queue结构体描述。
- 在向块设备提交这些请求前内核会先执行请求的合并和排序预操作，以提高访问的效率，然后再由内核中的I/O调度程序子系统来负责提交 I/O请求，调度程序将磁盘资源分配给系统中所有挂起的块 I/O 请求，其工作是管理块设备的请求队列，决定队列中的请求的排列顺序以及什么时候派发请求到设备。

由通用块层(Generic Block Layer)负责维持一个I/O请求在上层文件系统与底层物理磁盘之间的关系。在通用块层中，通常用一个bio结构体来对应一个I/O请求。Linux提供了一个gendisk数据结构体，用来表示一个独立的磁盘设备或分区，用于对底层物理磁盘进行访问。在gendisk中有一个类似字符设备中file_operations的硬件操作结构指针，是block_device_operations结构体。

当多个请求提交给块设备时，执行效率依赖于请求的顺序。如果所有的请求是同一个方向（如写数据），执行效率是最大的。内核在调用块设备驱动程序例程处理请求之前，先收集I/O请求并将请求排序，然后，将连续扇区操作的多个请求进行合并以提高执行效率，对I/O请求排序的算法称为**电梯算法**（elevator algorithm）。

对I/O请求排序的算法称为电梯算法（**elevator algorithm**）。电梯算法在I/O调度层完成。IO调度层（包括请求合并排序算法）由内核完成，用户不需考虑。内核提供了不同类型的电梯算法，常见的电梯算法有：
noop（实现简单的FIFO，基本的直接合并与排序），
anticipatory（延迟I/O请求，进行临界区的优化排序，Linux系统默认），
deadline（针对anticipatory缺点进行改善，降低延迟时间），
cfq（均匀分配I/O带宽，公平机制）

IO调度器的总体目标是希望让磁头能够总是往一个方向移动,移动到底了再往反方向走,这恰恰就是现实生活中的电梯模型,所以IO调度器也被叫做电梯.(elevator)而相应的算法也就被叫做电梯算法. 具体使用哪种算法我们可以在启动的时候通过内核参数elevator来指定.

另一方面我们也可以单独的为某个设备指定它所采用的IO调度算法,这就通过修改在/sys/block/sda/queue/目录下面的scheduler文件.比如:

```
[root@localhost ~]# cat /sys/block/sda/queue/scheduler  
noop anticipatory deadline [cfq]  
这里采用的是cfq.
```

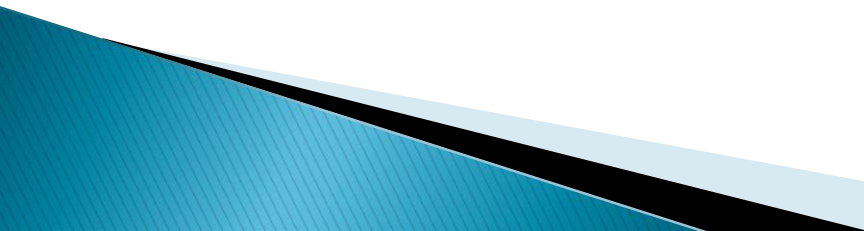
查看当前系统支持的IO调度算法

`dmesg | grep -i scheduler`

当前系统支持noop anticipatory deadline 和cfq

`echo deadline >/sys/block/sda/queue/scheduler` 来修改IO调度算法

块设备驱动程序设计方法

- ▶ 相关重要数据结构与函数
 - ▶ 块设备的注册与注销
 - ▶ 块设备初始化与卸载
 - ▶ 块设备操作
 - ▶ 请求处理
- 

关键数据结构

数据结构	说明
block_device	描述一个分区或整个磁盘对内核的一个块设备实例
gendisk	描述一个通用硬盘（generic hard disk）对象
hd_struct	描述分区应有的分区信息
bio	描述块数据传送时怎样完成填充或读取块给driver
request:	描述向内核请求一个列表准备做队列处理
request_queue:	描述内核申请request资源建立请求链表并填写BIO形成队列

块设备的注册与注销

▶ 注册

```
int register_blkdev(unsigned int major, const char  
*name);
```

▶ 注销

```
void unregister_blkdev(unsigned int major, const  
char *name);
```

块设备初始化与卸载

▶ 初始化

- 注册块设备及块设备驱动程序
- 分配、初始化、绑定请求队列（如果使用请求队列的话）
- 分配、初始化gendisk，为相应的成员赋值并添加gendisk。
- 其它初始化工作，如申请缓存区，设置硬件尺寸（不同设备，有不同处理）

▶ 卸载

- 删除请求队列
- 撤销对gendisk的引用并删除gendisk
- 释放缓冲区，撤销对块设备的应用，注销块设备驱动

块设备操作

```
struct block_device_operations {  
    /* 打开与释放*/  
    int (*open) (struct block_device *, fmode_t);  
    int (*release) (struct gendisk *, fmode_t);  
    /* I/O操作 */  
    int (*locked_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);  
    int (*ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);  
    int (*compat_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);  
    int (*direct_access) (struct block_device *, sector_t, void **, unsigned long *);  
    /*介质改变*/  
    int (*media_changed) (struct gendisk *);  
    unsigned long long (*set_capacity) (struct gendisk *, unsigned long long);  
    /* 使介质有效 */  
    int (*revalidate_disk) (struct gendisk *);  
    /*获取驱动器信息 */  
    int (*getgeo)(struct block_device *, struct hd_geometry *);  
    struct module *owner;  
};
```

请求处理

- ▶ 块设备不像字符设备操作，它并没有read和write。对块设备的读写是通过请求函数完成的，因此请求函数是块设备驱动的核心。
 - 使用请求队列：使用请求队列对于提高机械磁盘的读写性能具有重要意义，I/O调度程序按照一定算法（如电梯算法）通过优化组织请求顺序，帮助系统获得较好的性能。
 - 不使用请求队列：但是对于一些本身就支持随机寻址的设备，如SD卡、RAM盘、软件RAID组件、虚拟磁盘等设备，请求队列对其没有意义。针对这些设备的特点，块设备层提供了“无队列”的操作模式

MMC/SD芯片介绍

- ▶ MMC卡（Multimedia Card）是一种快闪记忆卡标准。在1997年由西门子及SanDisk共同开发，该技术基于东芝的NAND快闪记忆技术。它相对于早期基于Intel NOR Flash技术的记忆卡（例如CF卡），体积小得多
- ▶ SD卡（Secure Digital Memory Card）也是一种快闪记忆卡，同样被广泛地在便携式设备上使用，例如数码相机、PDA和多媒体播放器等。SD卡的数据传送协议和物理规范是在MMC卡的基础上发展而来。SD卡比MMC卡略厚，但SD卡有较高的数据传送速度，而且不断地更新标准

MMC/SD卡驱动结构

文件系统

块设备驱动(driver/mmc/card)

MMC/SD核心(driver/mmc/core)

MMC/SD接口(driver/mmc/host)

MMC卡块设备驱动分析

▶ 注册与注销

- `mmc_blk_init`
- `mmc_blk_exit`

▶ 设备加载与卸载

- `mmc_blk_probe`
- `mmc_blk_remove`

▶ 设备的打开与释放

- `mmc_blk_open`
- `mmc_blk_release`

▶ MMC驱动的请求处理函数

- `mmc_prep_request`
- `mmc_blk_issue_rq`
- `mmc_requset`

块设备的驱动小结：

- ▶块设备的I/O操作方式与字符设备存在较大的不同，因而引入了request_queue、request、bio等一系列数据结构。在整个块设备的I/O操作中，贯穿于始终的就是“请求”，字符设备的I/O操作则是直接进行，而块设备的I/O操作会排队和整合。
- ▶驱动的任务是处理请求，对请求的排队和整合由I/O调度算法解决，因此，块设备驱动的核心就是请求处理函数或“制造请求”函数。
- ▶尽管在块设备驱动中仍然存在block_device_operations结构体及其成员函数，但其不再包含读写一类的成员函数，而只是包含打开、释放及I/O控制等与具体读写无关的函数。块设备驱动的结构相当复杂的，但幸运的是，块设备不像字符设备那么包罗万象，它通常就是存储设备，而且驱动的主体已经由Linux内核提供，针对一个特定的硬件系统，驱动工程师所涉及到的工作往往只是编写少量的与硬件直接交互的代码。

8.7

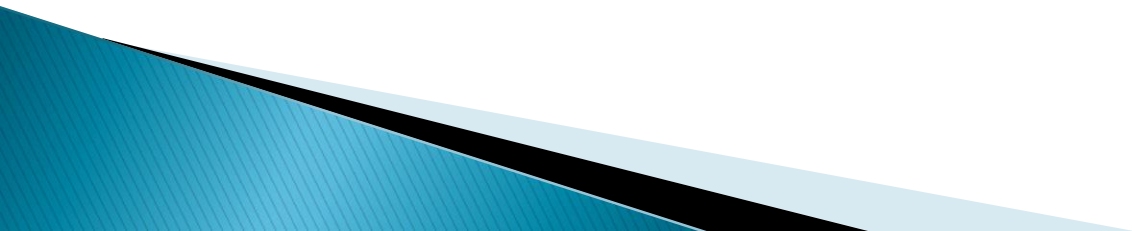
Part Seven

嵌入式网络设备驱动设计

以太网（**Ethernet**）技术凭借高速开放的特性在嵌入式系统中得到广泛应用。以太网对应**ISO**分层中的数据链路层和物理层。以太网接口包含**介质访问控制子层（MAC）**和**物理层（PHY）**。**MAC**通过读取和设置**PHY**的寄存器获得**PHY**的状态信息或者改变**PHY**的参数。

目前嵌入式系统使用以太网接口通常有两种方式：
一是片上系统携带**MAC**控制器配合外接**PHY**芯片，如**RTL8201**等；
二是片上系统外接同时具有**MAC**控制器和**PHY**接收器的网卡芯片，如**DM9000**等。

与字符设备和块设备的驱动程序处理方法有些类似，为了达到屏蔽网络环境中物理网络设备的多样性的效果，**Linux** 操作系统利用面向对象的思想对所有的网络物理设备进行抽象，并定义一个统一的接口。



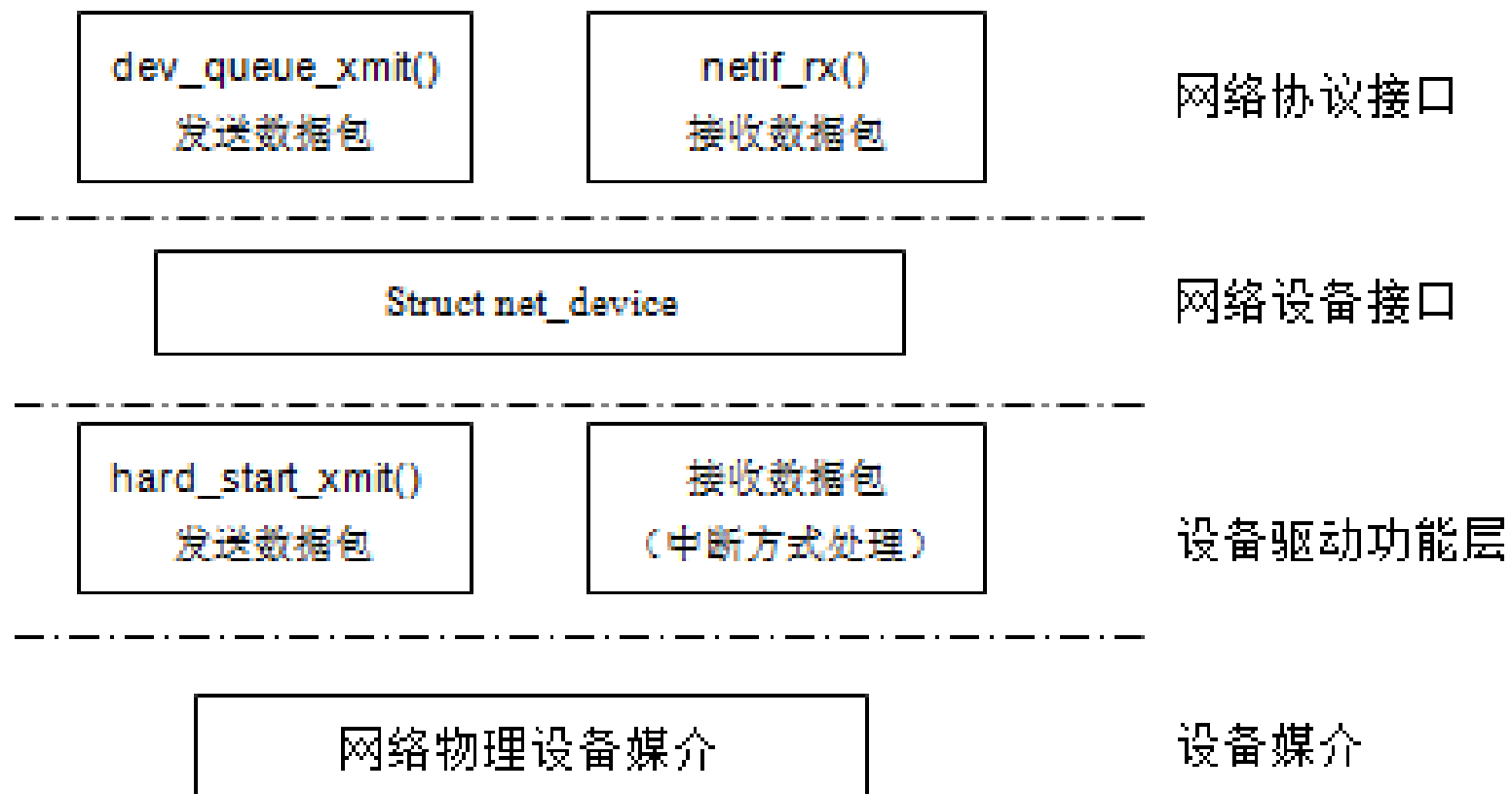
但是与其他两类设备驱动程序的框架不同的是，网络设备驱动程序有着自身的特点：

第一，网络接口是用一个 **net_device** 数据结构表示的。字符设备或块设备在文件系统中都存在一个相应的特殊设备的文件来表示其相对应的设备，如 **/dev/hda1**、**/dev/tty1** 等。对字符设备和块设备的访问都需通过文件操作界面。网络设备在对数据包进行发送或接收时，则直接通过网络接口（套接字 **socket**）访问，不需要进行文件上的操作。

第二，网络接口是在系统初始化时实时生成的，当物理网络设备不存在时，也不存在与之相对应的 **device** 结构。而即使字符设备和块设备的物理设备不存在，在 **/dev** 下也必定有与之相对应的文件。

- 嵌入式Linux 的网络系统主要采用 **socket** 机制，操作系统和驱动程序之间定义专门的数据结构 **sk_buff** 用来进行数据包的发送与接收。
- 对于 Linux 网络设备驱动程序可以分为网络协议接口层、网络设备接口层、提供实际功能的设备驱动功能层和网络设备与媒介层等四个层，

8.7.1 网络设备驱动程序框架



网络设备驱动模型层次结构

(1) 网络协议接口层负责向网络层协议提供统一的数据包发送和接收接口而不论上层协议是ARP或者IP，该层中的 **dev_queue_xmit()** 函数用来发送数据包，**netif_rx()** 函数用来接收数据包。

(2) 网络设备接口层能够给协议接口层提供统一并具有具体网络设备属性和操作的数据结构体 **device** (**struct net_device**)，**device** 结构体是网络设备驱动功能层中各个函数的容器。从宏观上出发，网络设备接口层规划了具体用来操作硬件的网络设备驱动功能层的结构。

(3) 设备驱动功能层中各个函数是网络设备接口层中 **device** 数据结构体的具体成员函数，其能够驱使网络设备硬件完成相应动作的程序，并通过函数 **hard_start_xmit()** 开启发送数据包的操作、通过网络设备的中断触发开启接收数据包操作。

(4) 网络设备和媒介层是完成数据包发送和接受的物理实体，包括网络适配器和具体的传输媒介，网络适配器被设备驱动功能层中的函数物理上驱动。对于 **Linux** 操作系统而言，网络设备和媒介都可以是虚拟的。

8.7.2 网络设备驱动程序关键数据结构

Linux 中网络驱动程序中最重要的是根据上层网络设备接口层定义的 **net_device** 数据结构和底层硬件特性，完成网络设备驱动程序的功能，主要包括数据的接收、发送等。

网络驱动程序部分最重要的就是这两个数据结构：一个是 **sk_buff** 数据结构

另一个重要的结构体是 **net_device** 数据结构

Sk_buff位于网络协议接口层，用于在linux网络子系统各层次之间传递数据。定义在include/linux/skbuff.h中。其主要使用思想是：当发送数据包时，将要发送的数据存入**sk_buff**中，传递给下层，通过添加相应的协议头交给网络设备发送；当接收数据包时，想将数据保存在**sk_buff**中，并传递到上层，上层通过剥去协议头直至交给用户。

结构 net_device 存储一个网络接口的重要信息，是网络驱动程序的核心。在逻辑上，它可以分割为两个部分：可见部分和隐藏部分。可见部分是由外部赋值；隐藏部分的域段仅面向系统内部，它们可以随时被改变。**net_device**结构体位于网络设备接口层，用于描述一个网络设备

8.7.3 网络设备驱动程序设计方法概述

概括的说，一个网络设备的驱动程序至少应该具有以下的内容：

（1）该网络设备的检测及初始化函数，供内核启动初始化时调用，若是要写成 **module** 兼容方式，还需要编写该网络设备的 **init_module**和 **cleanup_module** 函数。

（2）调用**open**、**close** 函数用来进行网络设备的打开和关闭操作。

（3）提供该网络设备的数据传输函数，负责向硬件发送数据包。当上层协议需要传输数据时，可作为函数 **dev_queue_xmit** 的调用；

（4）中断服务程序，用来处理数据的接收的发送。当物理网络设备有新数据到达或数据传输完毕时，将向系统发送硬件中断请求，该函数就是用来响应该中断请求的。

网络设备驱动程序的设计需要完成网络设备的注册、初始化与注销， 以及进行发送和接收数据处理， 并能针对传送超时、中断等情况进行及时处理。在 **linux** 内核中提供了设备驱动功能层主要的数据结构 and 函数的设计模板。普通开发者只需要根据实际硬件情况完成“**填空**”步骤即可完成相关工作。

网络设备驱动程序示例-网卡DM9000驱动程序分析

