《嵌入式系统》

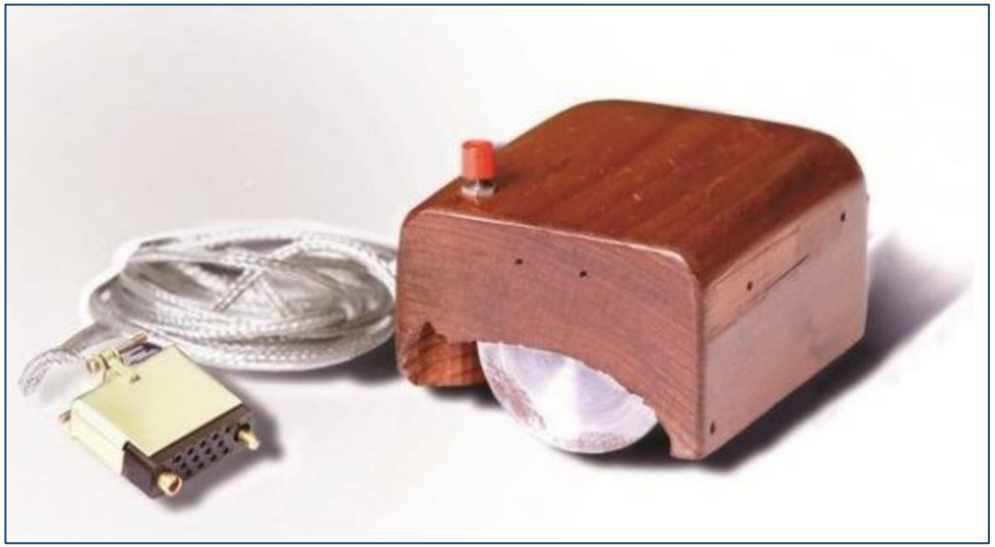
6-2 字符设备驱动程序

认识字符设备

提供连续的数据流，应用程序可以顺序读取的设备，通常没有缓冲、不支持随机存取，支持按字节/字 来读写数据。

字符设备-鼠标

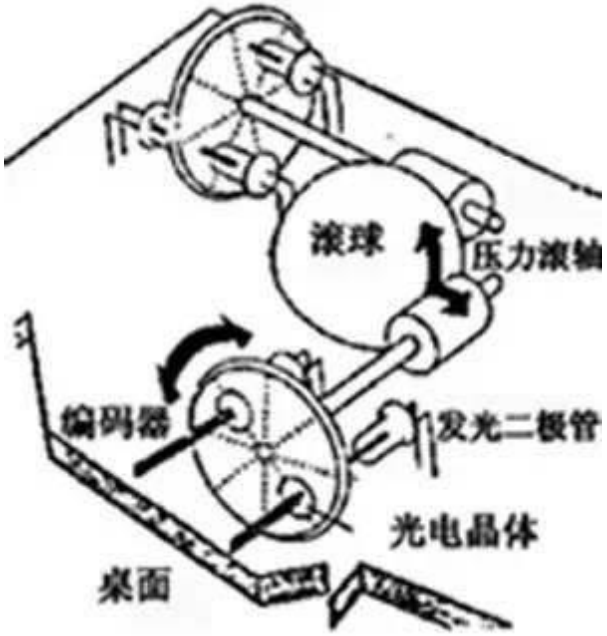
- 鼠标是最常见的字符输入设备，经过五十多年的发展与积累，鼠标从滚轮到光电，从有线到无线一步步发展，而原理依然大同小异。



1968年12月9日，美国斯坦福大学博士道格拉斯·恩格尔巴特展示了世界上第一个鼠标。其设计目的，是为了代替键盘繁琐的输入指令，从而使计算机的操作更加简便。其工作原理是由它底部的小球带动枢轴转动，继而带动变阻器改变阻值来产生位移信号，并将信号传至主机。

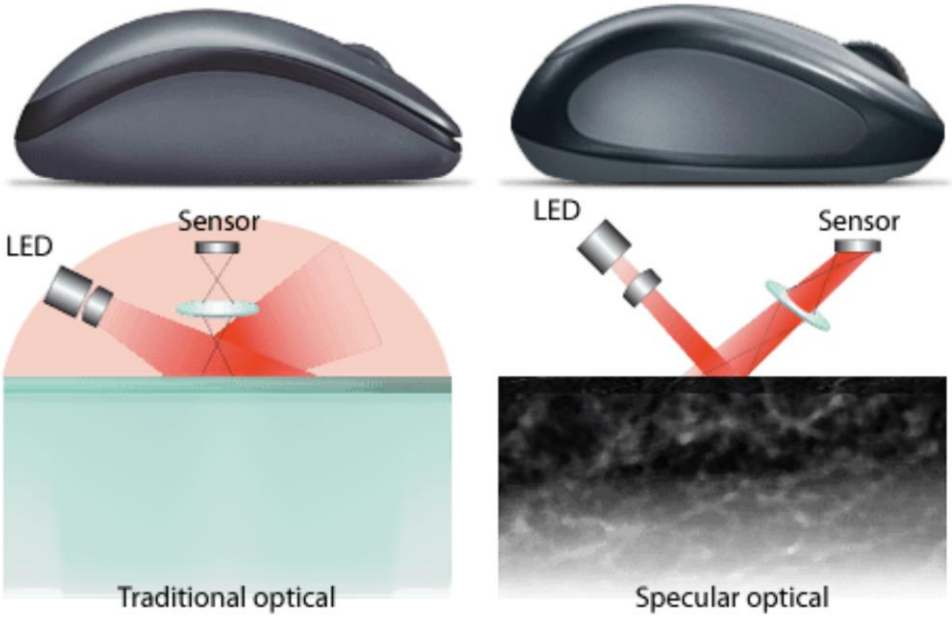
字符设备-鼠标

- 机械鼠标是通过移动鼠标，带动胶球，胶球滚动又摩擦鼠标内水平和垂直两个方向的栅轮滚轴，驱动栅轮转动。栅轮格栅两侧，一侧是一红外发光管，另一侧是红外接收组件。
- 正常工作时，鼠标的移动转换为水平和垂直栅轮不同方向和转速的转动。栅轮转动时，栅轮的轮齿周期性遮挡红外发光管发出的红外线照射到接收组件中的甲管和乙管,从而甲和乙输出端输出至鼠标内控制芯片的脉冲具有相位差，由此可计算出鼠标运动的方向和速度。



字符设备-鼠标

- 光电鼠标的工作原理是：在光电鼠标内部有一个发光二极管，通过该发光二极管发出的光线，照亮光电鼠标底部表面。然后将光电鼠标底部表面反射回的一部分光线，经过一组光学透镜，传输到一个光感应器件内成像，当光电鼠标移动时，其移动轨迹便会被记录为一组高速拍摄的连贯图像。



- 利用光电鼠标内部的DSP对移动轨迹上摄取的一系列图像进行分析处理，通过对这些图像上特征点位置的变化进行分析，来判断鼠标的移动方向和移动距离，从而完成光标的定位。
- 其实光电鼠标还细分为普通光电(optical)鼠标和激光光电(laser)鼠标，同学们可以课外延伸了解它们工作原理和性能上的区别。

字符设备-触摸屏

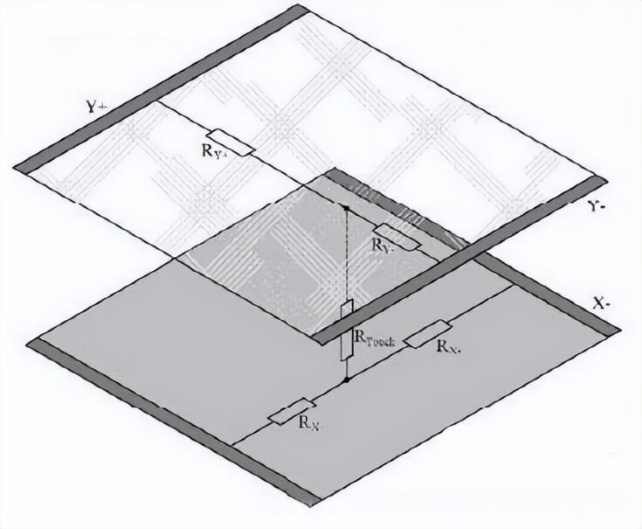
- 触摸屏系统一般包括两个部分：触摸检测装置和触摸屏控制器。触摸检测装置安装在显示器屏幕前面，用于检测用户触摸位置，接收后送触摸屏控制器；触摸屏控制器的主要作用是从触摸点检测装置上接收触摸信息，并将它转换成触点坐标，再送给CPU，它同时能接收CPU发来的命令并加以执行。
- 触摸屏技术也经历了从低档向高档逐步升级和发展的过程。根据其工作原理，其目前一般被分为四大类：电阻式触摸屏、电容式触摸屏、红外线式触摸屏和表面声波触摸屏。

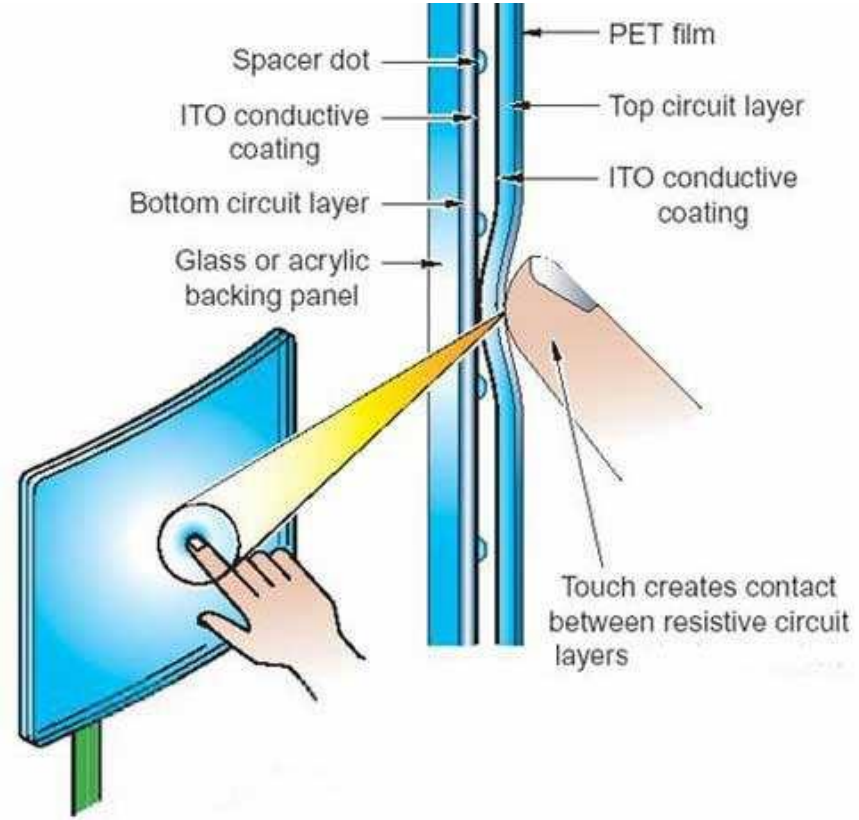
字符设备-触摸屏

- 电阻式触摸屏
  - 电阻触摸屏的屏体部分是一块多层复合薄膜，由一层玻璃或有机玻璃作为基层，表面涂有一层透明的导电层(ITO膜)，上面再盖有一层外表面经过硬化处理、光滑防刮的塑料层。它的内表面也涂有一层ITO，在两层导电层之间有许多细小(小于千分之一英寸)的透明隔离点把它们隔开。
  - 当接触屏幕时，两层ITO发生接触，电阻发生变化，控制器根据检测到的电压(/电阻）变化来计算接触点的坐标，再依照这个坐标来进行相应的操作，因此这种技术必须是要施力到屏幕上，才能获得触摸效果。

字符设备-触摸屏

- 电阻式触摸屏

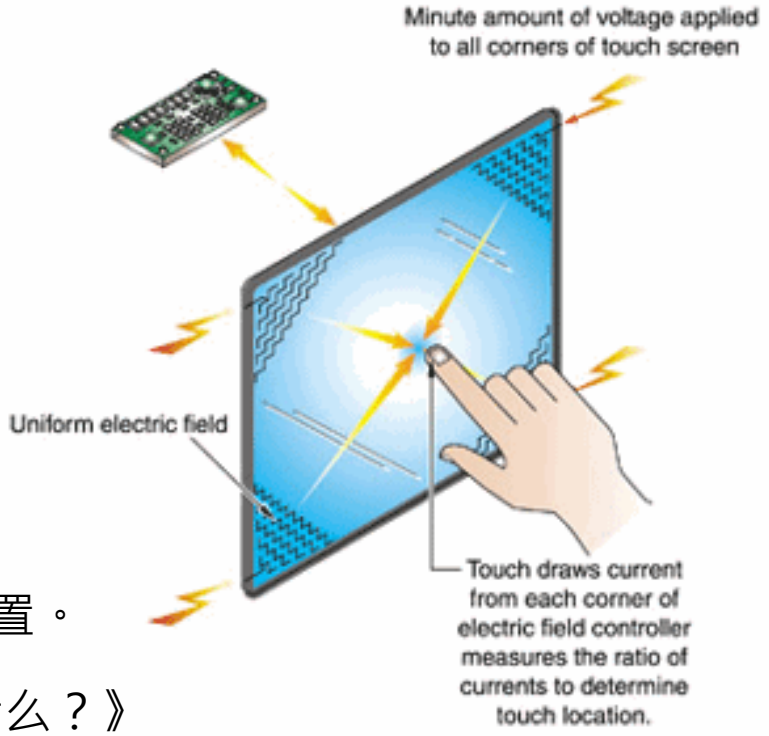




- 如今日常生活中还能见到电阻式触摸屏吗，可否举例一二？

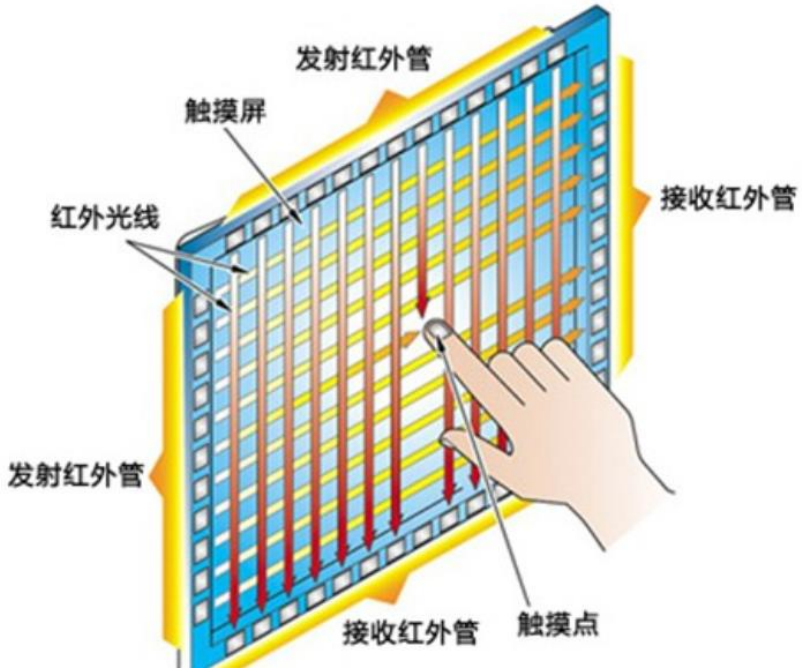
字符设备-触摸屏

- 电容式触摸屏（单点触控式）
  - 电容式触摸屏的四边均镀上了狭长的电极，其内部形成一个低电压交流电场。触摸屏上贴有一层透明的薄膜层，它是一种特殊的金属导电物质。当用户触摸电容屏时，用户手指和工作面形成一个耦合电容，因为工作面上接有高频信号，于是手指会吸走一个很小的电流，这个电流分别从屏的四个角上的电极中流出；且理论上流经四个电极的电流与手指到四角的距离成比例，控制器通过对四个电流比例的精密计算，即可得出接触点位置。



- 推荐阅读：李永乐《手机触摸屏的原理是什么？》
  https://baijiahao.baidu.com/s?id=1603579964610413218

字符设备-触摸屏

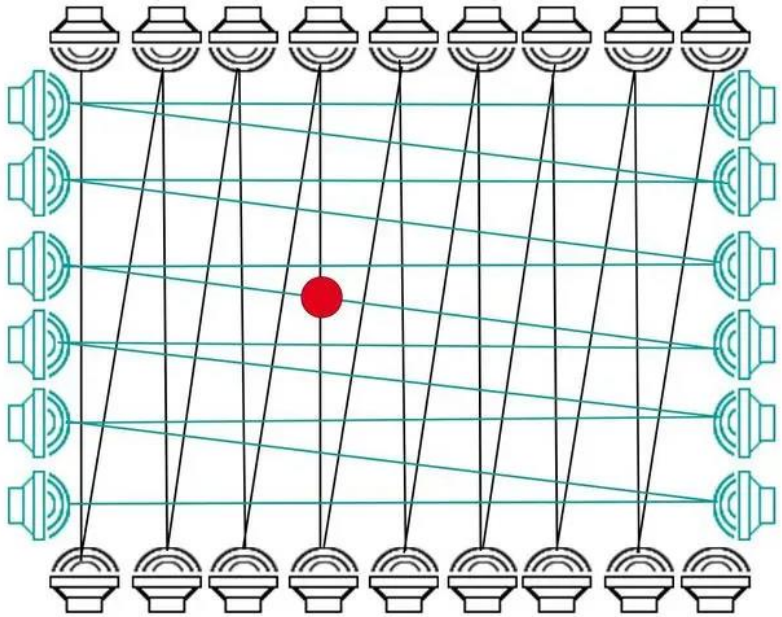


- 红外线式触摸屏
  - 红外触摸屏的四边排布了红外发射管和红外接收管，它们一一对应形成横竖交叉的红外线矩阵。用户在触摸屏幕时，手指会挡住经过该位置的横竖两条红外线，控制器通过计算即可判断出触摸点的位置。

字符设备-触摸屏

❑表面声波（Surface Acoustic Wave）触摸屏

- ❑表面声波触摸屏分别贴有X、Y方向发射和接收声波的换能器。换能器在传感器接收到的表面上创建一个不可见的超声波网格，因此称为表面声波。当手指或柔软物体触摸屏幕时，部分声能被吸收，从而改变接收到的信号，控制器处理得到触摸的X、Y坐标。



清華大學
Tsinghua University

提纲

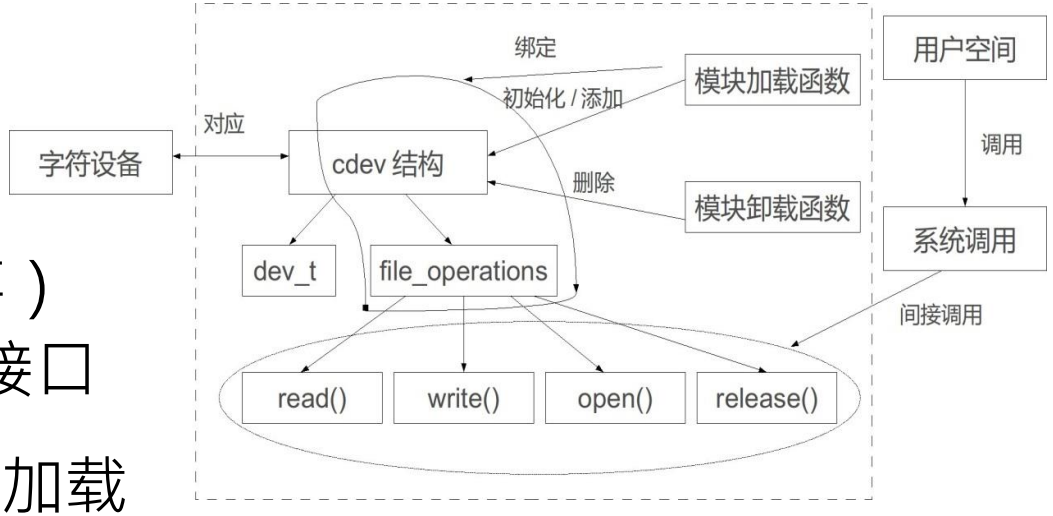
- 字符设备驱动框架
- 字符设备驱动开发
- GPIO驱动概述
- 串行总线概述
- I^2C总线驱动开发

字符设备驱动框架（1）

- 字符设备驱动程序是嵌入式Linux最基本、也是最常用的驱动程序
- 字符设备在Linux内核中使用struct cdev结构来表示
- 在struct cdev结构中包含着字符设备需要的全部信息，其中最主要的两个：
  - 设备号dev_t
  - 文件操作file_operations

字符设备驱动框架（2）

- 设备号(dev_t)将驱动程序与设备文件关联
- 文件操作函数（read等）是实现上层系统调用的接口
- 驱动程序加载时，模块加载函数
  1. 负责初始化cdev结构
  2. 将cdev与文件操作函数绑定
  3. 向内核中添加该cdev结构
- 模块卸载函数从内核中删除cdev结构



清華大學
Tsinghua University

提纲

- 字符设备驱动框架
- 字符设备驱动开发
- GPIO驱动概述
- 串行总线概述
- I^2C总线驱动

设备号

- 设备号
  - 设备结点的主、次设备号用于表示一个硬件设备；
  - 对于每个类型的设备，其主、次设备号都有相关的规定说明；
  - 在内核源码里面的Documentation\devices.txt说明文件中，就规定了各种设备的主、次设备号。

设备号

- 设备号
  - 在内核驱动程序中，使用dev_t类型用于保存设备号。
    - dev_t是无符号32位整型；
    - 高12位表示主设备号；
    - 低20位是次设备号；
  - 在驱动程序中，常常要通过设备号获取主、次设备号：
    - MAJOR(dev_t dev);　　//获取主设备号
    - MINOR(dev_t dev);　　//获取次设备号
  - 通过主、次设备号合成设备号：
    - MKDEV(int major, int minor);

设备号

- 分配和释放字符设备号
  - 编写驱动程序要做的第一件事，为字符设备获取一个设备号，分**静态分配**和**动态分配**两种方式。
  - **静态分配**：如果事先知道所需要的设备号（主设备号）的情况，可以使用以下函数，向内核申请已知的设备号：
    - int register_chrdev_region(dev_t first,
      unsigned count,
      const char *name)；

      first是要分配的起始设备号值。

      count 所请求的连续设备编号的个数。

      name设备名称，指和该编号范围建立关系的设备。
    - 分配成功返回0，失败返回负数。

设备号

- 分配和释放字符设备号
  - **动态分配**:可以让内核动态分配设备号（主要是主设备号）；使用下面的函数
    - int alloc_chrdev_region(dev_t *dev,
      unsigned baseminor,
      unsigned count,
      const char *name)；
    - dev 是一个仅用于输出的参数, 它在函数成功完成时保存已分配范围的第一个编号。
    - baseminor 应当是请求的第一个要用的次设备号，它常常是 0.
    - count 和 name 参数跟register_chrdev_region 的一样.

设备号

- 分配和释放字符设备号
  - 不再使用时，释放这些设备编号。使用以下函数：
    - void unregister_chrdev_region(dev_t from, unsigned count)；
  - 在模块的卸载函数中调用该函数。

设备的分配与初始化

- 如果想在运行时获取一个独立的cdev结构体，可以使用下面函数**动态分配**设备，随后显式地初始化cdev结构体的owner和ops成员。

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

设备的分配与初始化

- 如果要把cdev结构体嵌入到自己的设备特定结构中，可以采用**静态分配**方式。cdev_init()函数和cdev_alloc()函数功能基本相同，唯一的区别是cdev_init用于初始化已经存在的cdev结构体。

```
struct cdev my_cdev;

cdev_init(&my_cdev,&fops);

//用于初始化一个静态分配的cdev结构体，并建立cdev和file_operations之间的连接

my_cdev->owner = THIS_MODULE;

//只需初始化owner成员
```

字符设备的注册

- 注册一个独立的cdev设备的基本过程如下：
  - 为struct cdev 分配空间

    struct cdev *my_cdev = cdev_alloc();
  - 初始化struct cdev

    void cdev_init(struct cdev *dev, const struct file_operations *fops)

前面介绍了初始化的过程

- cdev设置完成，通知内核struct cdev的信息（在执行这步之前必须确定你对struct cdev的以上设置已经完成！）

  int cdev_add(struct cdev *dev, dev_t num, unsigned int count);

字符设备的注册

- 通知内核struct cdev的信息：

  int cdev_add(struct cdev *dev, dev_t num, unsigned int count);

  - dev 是要添加的设备的 cdev 结构,
  - num 是这个设备对应的第一个设备号,
  - count 是应当关联到设备的设备号的数目.

- 卸载字符设备时，调用相反的动作函数：

  void cdev_del(struct cdev *dev);

字符设备的注册

- 早期方法：
  - 内核中仍有许多字符驱动不使用刚刚描述过的cdev接口。没有更新到 2.6 内核接口的老代码。
  - 注册一个字符设备的早期方法:

    int register_chrdev(unsigned int major, const char *name, struct file_operations *fops);

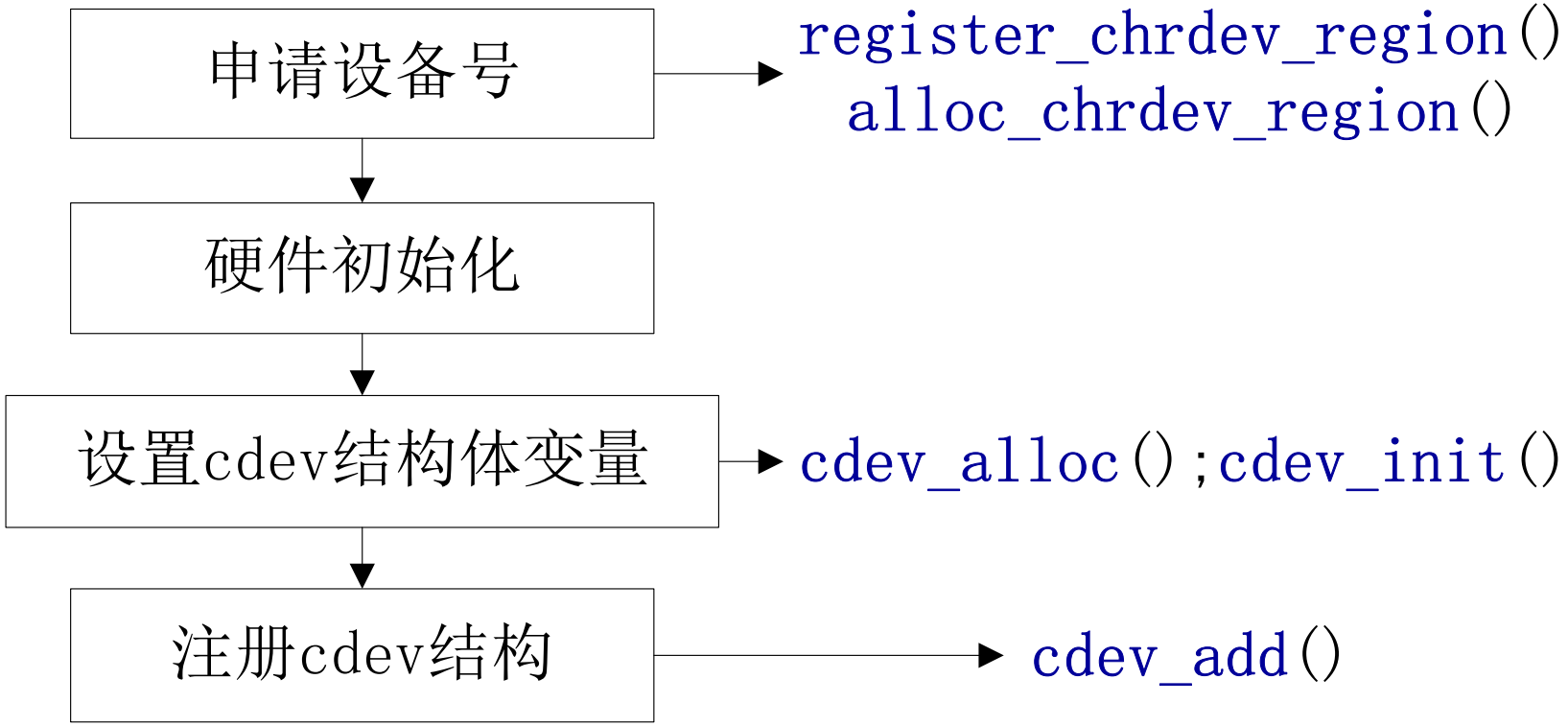
    - major 是给定的主设备号。为0代表什么？
    - name 是驱动的名字(将出现在 /proc/devices),
    - fops 是设备驱动的file_operations 结构。
    - register_chrdev 将给设备分配 0 - 255 的次设备号, 并且为每一个建立一个缺省的 cdev 结构。
  - 从系统中卸载字符设备的函数:

    int unregister_chrdev(unsigned int major, const char *name);

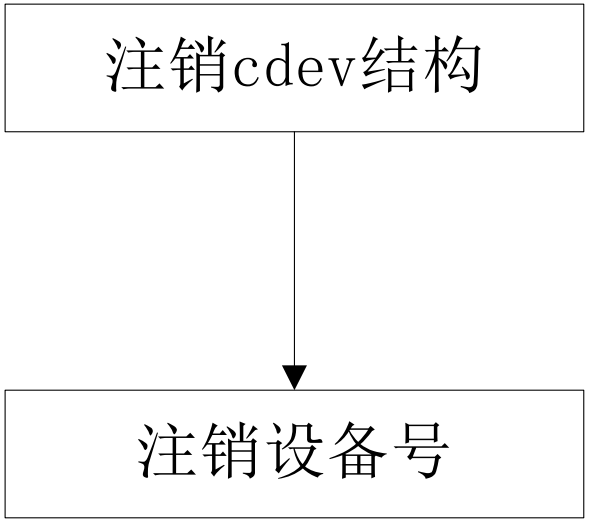
字符设备驱动程序编写流程

- 实现底层操作函数
  - 将这些底层操作函数结合到file_operations结构中；
- 在模块的入口函数
  - 申请、注册设备号；
  - 初始化cdev(要关联一个file_operations结构)
  - 注册cdev；
- 在模块的出口函数
  - 注销cdev；
  - 注销设备号；

字符设备模块入口函数的执行流程



字符设备模块出口函数的执行流程



cdev_del()

unregister_chrdev_region()

提纲

- 字符设备驱动框架
- 字符设备驱动开发
- GPIO驱动概述
- 串行总线概述
- I^2C总线驱动

GPIO驱动概述

- 在ARM里，所有I/O都是通用的，称为GPIO（General Purpose Input/Output，通用输入输出）
- 每个GPIO端口一般包含8个引脚，例如PA端口为PA0～PA7，可以控制I/O接口作为输入或者输出
- 许多设备或电路通常只需要一位，即表示开/关两状态就够了，例如LED灯的亮和灭
- GPIO接口一般至少会有两个寄存器，即控制寄存器和数据寄存器

GPIO特性

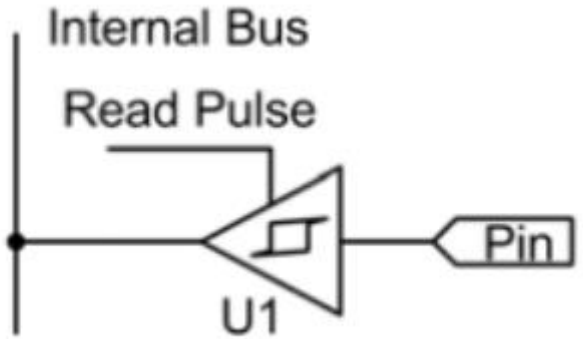
- 可编程控制GPIO中断
  - 屏蔽中断发生
  - 边沿触发（上升沿、下降沿、双边沿）
  - 电平触发（高电平、低电平）
- 输入/输出可承受5V
- 在读和写操作中通过地址线进行位屏蔽
- 可编程控制GPIO管脚配置：
  - 弱上拉或弱下拉电阻
  - 2mA、4mA、8mA驱动，以及带驱动转换速率（Slew Rate）控制的8mA驱动
  - 开漏使能
  - 数字输入使能

GPIO的模式

- GPIO管脚可以被配置为多种工作模式，其中有3种比较常用：高阻输入、推挽输出、开漏输出
- 高阻输入
  - 大多数计算机中的信息传输线采用总线形式，为防止信息相互干扰，要求凡挂到总线上的寄存器或存储器等，它的输入输出端不仅能呈现0、1两个信息状态，而且还应能呈现第三个状态----高阻抗状态，即此时好像它们的输出被开关断开，对总线状态不起作用，此时总线可由其他器件占用。三态缓冲器即可实现上述功能，它除具有输入输出端之外，还有一控制端。

GPIO的模式

- 如图为GPIO管脚在高阻输入模式下的等效结构示意图。输入模式的结构比较简单，就是一个带有施密特触发输入的三态缓冲器（U1），并具有很高的输入等效阻抗。执行GPIO管脚读操作时，在读脉冲（Read Pulse）的作用下会把管脚（Pin）的当前电平状态读到内部总线上（Internal Bus）。在不执行读操作时，外部管脚与内部总线之间是隔离的。

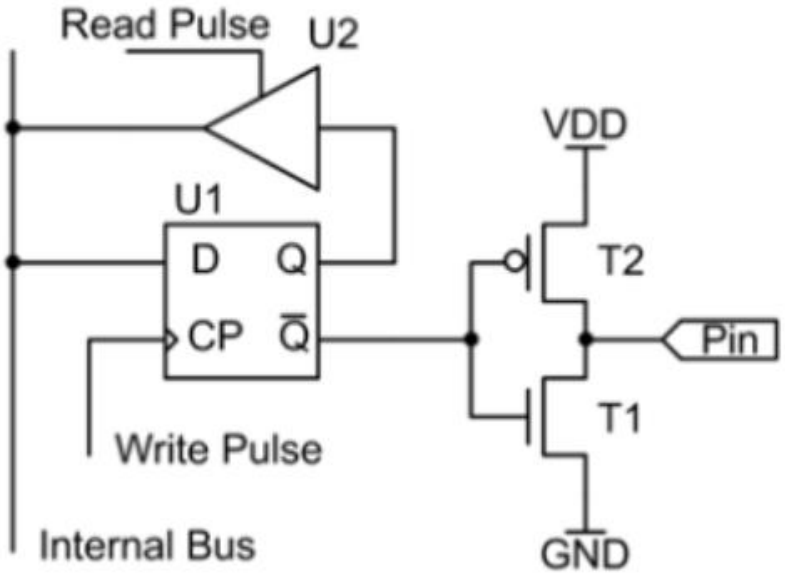


GPIO的模式

- 推挽输出
  - 推挽输出原理：在功率放大器电路中大量采用推挽放大器电路，这种电路中用两只三极管构成一级放大器电路，两只三极管分别放大输入信号的正半周和负半周，两只三极管输出的半周信号在放大器负载上合并后得到一个完整周期的输出信号。
  - 推挽放大器电路中，一只三极管工作在导通、放大状态时，另一只三极管处于截止状态，当输入信号变化到另一个半周后，原先导通、放大的三极管进入截止，而原先截止的三极管进入导通、放大状态，两只三极管在不断地交替导通放大和截止变化，所以称为推挽放大器。

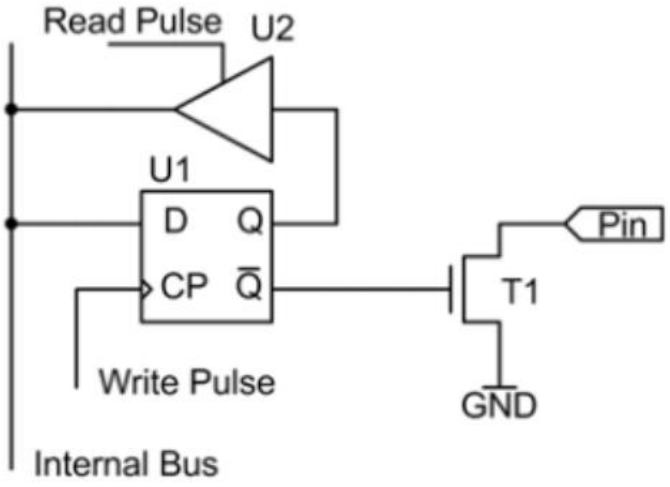
GPIO的模式

- 如图是GPIO管脚在推挽输出模式下的等效结构示意图。U1是输出锁存器，执行GPIO管脚写操作时，在写脉冲（Write Pulse）的作用下，数据被锁存到Q和/Q。T1和T2构成CMOS反相器，T1导通或T2导通时都表现出较低的阻抗，但T1和T2不会同时导通或同时关闭，最后形成的是推挽输出。



GPIO的模式

- 开漏输出
  - 下图为GPIO管脚在开漏输出模式下的等效结构示意图。开漏输出和推挽输出相比结构基本相同，但只有下拉晶体管T1而没有上拉晶体管。同样，T1实际上也是多组可编程选择的晶体管。开漏输出的实际作用就是一个开关，输出“1”时断开、输出“0”时连接到GND。



GPIO的模式

- 开漏输出
  - 开漏输出能够方便地实现“线与”逻辑功能，即多个开漏的管脚可以直接并在一起（不需要缓冲隔离）使用，并统一外接一个合适的上拉电阻，就自然形成“逻辑与”关系。开漏输出的另一种用途是能够方便地实现不同逻辑电平之间的转换（如3.3V到5V之间），只需外接一个上拉电阻，而不需要额外的转换电路。典型的应用例子就是基于开漏电路连接的I2C总线。

嵌入式设备驱动程序结构

- 根据功能划分，Linux的设备驱动程序的代码结构可分为：
  - 驱动程序的注册与注销
  - 设备的打开与释放
  - 设备的读写操作
  - 设备的控制操作
  - 设备的轮询和中断处理

GPIO驱动设计-LED灯控制

- 搭好框架

```
#include <linux/module.h>//模块加载的头文件
#include <linux/init.h>//用户定义模块初始函数名需引用的头文件
static int led_init(void)
{
    return 0;
}
static void led_exit(void)
{
}
MODULE_LICENSE("GPL");
module_init(led_init);
module_exit(led_exit);
```

GPIO驱动设计-LED灯控制

- 定义设备结构体、设备号和文件操作结构体,这里我们只实现文件的打开和I/O控制

```
struct cdev cdev;
dev_t devnum;
static struct file_operations led_fops = {
    .open = led_open,
    .ioctl = led_ioctl,
};
```

GPIO驱动设计-LED灯控制

- 初始化设备，分配设备号，注册设备

```
static int led_init(void)
{
    cdev_init(&cdev, &led_fops);                    //初始化设备

    alloc_chrdev_region(&devnum, 0, 1, "myled");    //动态分配主设备号，这里的0是我们写的，设备在注册时候会检查，如果我们写的不行，会重新分配
    cdev_add(&cdev, devnum, 1);                     //注册设备
    return 0;
}
```

GPIO驱动设计-LED灯控制

- 在led_exit(void)函数里添加设备注销的代码

```
static void led_exit(void) {
    cdev_del(&cdev);
    //删除这个设备
    unregister_chrdev_region(devnum, 1);
    //注销这个设备的设备号
}
```

GPIO驱动设计-LED灯控制

- 实现文件操作接口体的两个接口函数

```
static struct file_operations led_fops =
{
    .open = led_open,
    .ioctl = led_ioctl,
};
```

GPIO驱动设计-LED灯控制

❑查看芯片手册，确定I/O端口的寄存器地址

```
int led_open(struct inode *node, struct file *filp) {
    led_config = ioremap(GPBCON,4);
    //在ARM linux系统下操作硬件，不能直接使用物理地址，须转化成虚拟地址。用ioremap函数，4表示要映射的空间大小，函数返回映射的虚拟地址。
    writel(0x400,led_config);
    //对一个寄存器赋值，用writel这个函数，l表示写进的是一个32位的值
    led_data = ioremap(GPBDAT,4);
    return 0;
    }
```

GPIO驱动设计-LED灯控制

- 实现对LED灯的控制

```
int led_ioctl(struct inode *node, struct     file *filp, unsigned int
cmd, unsigned long arg) {
switch(cmd) {
case LED_ON: writel(0x00,led_data);
//这里是点亮LED1
return 0;
 case LED_OFF: writel(0x7ff,led_data);
//这里是熄灭LED1
return 0;
default: return -EINVAL; } }
```

GPIO驱动设计-LED灯控制

- 编写Makefile文件

```
obj-m := led.o KDIR := /root/myhome/linux-2.6.32.2
//内核文件的目录

all : make -C $(KDIR) M=$(PWD) modules
CROSS_COMPILE=arm-linux-ARCH=arm
clean : rm -f *.o *.ko *.order *.symvers
```

- 运行之后会生成led.ko文件

GPIO驱动设计-LED灯控制

❑编写应用程序

```
int main(int argc, char *argv[])    //带参数的函数
{
    int fd;
    int cmd;
    cmd = atoi(argv[1]);//读取参数
    fd = open( “/dev/myled”,O_RDWR);//打开设备文件
    if(cmd == 1){
        ioctl(fd,LED_ON);//打开LED灯
    }
    else{
        ioctl(fd,LED_OFF);//关闭LED灯
    }
    return 0;
}
```

GPIO驱动设计-LED灯控制

- 把编译好的led.ko和led_app这两个文件传到开发板
- 安装驱动：insmod led.ko
- 创建驱动：mknod /dev/myled c 253 0

mknod	创建设备文件指令
/dev/myled	这是目录
c	代表字符设备文件
253	主设备号
0	次设备号

- 运行应用程序./led_app 0,./led_app 1控制LED灯

清華大學
Tsinghua University

提纲

- 字符设备驱动框架
- 字符设备驱动开发
- GPIO驱动概述
- 串行总线概述
- I^2C总线驱动

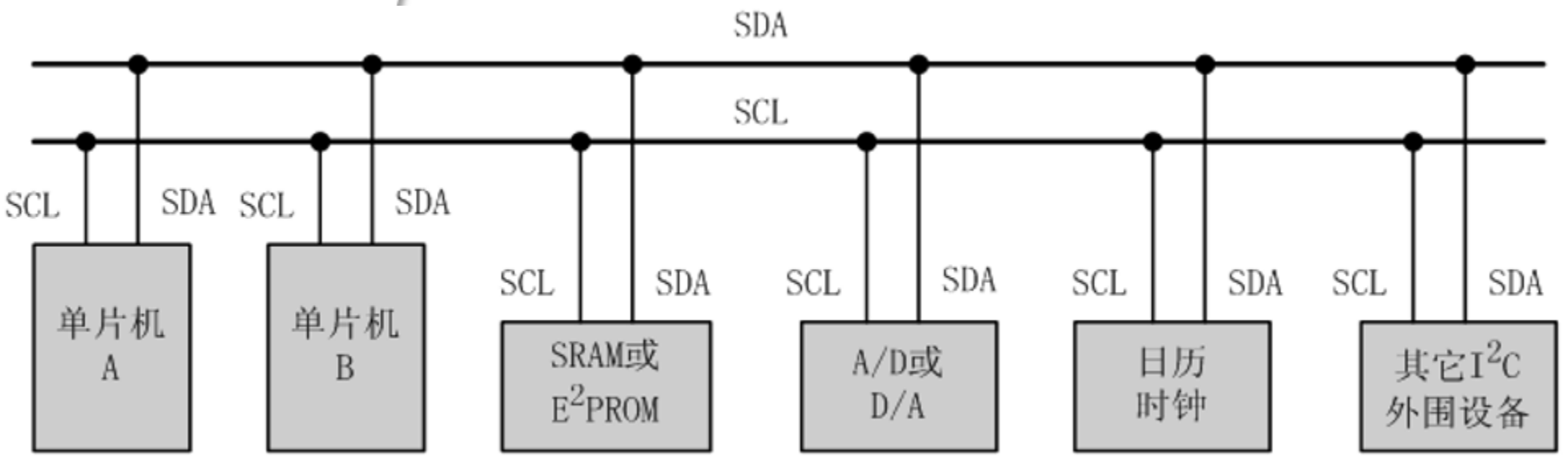
串行总线概述

- 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
  - I^2C
  - SPI
  - SMBUS

I²C总线

- 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和串行时钟信号线SCL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



- 在高速模式下可以达到3.4Mbit/s

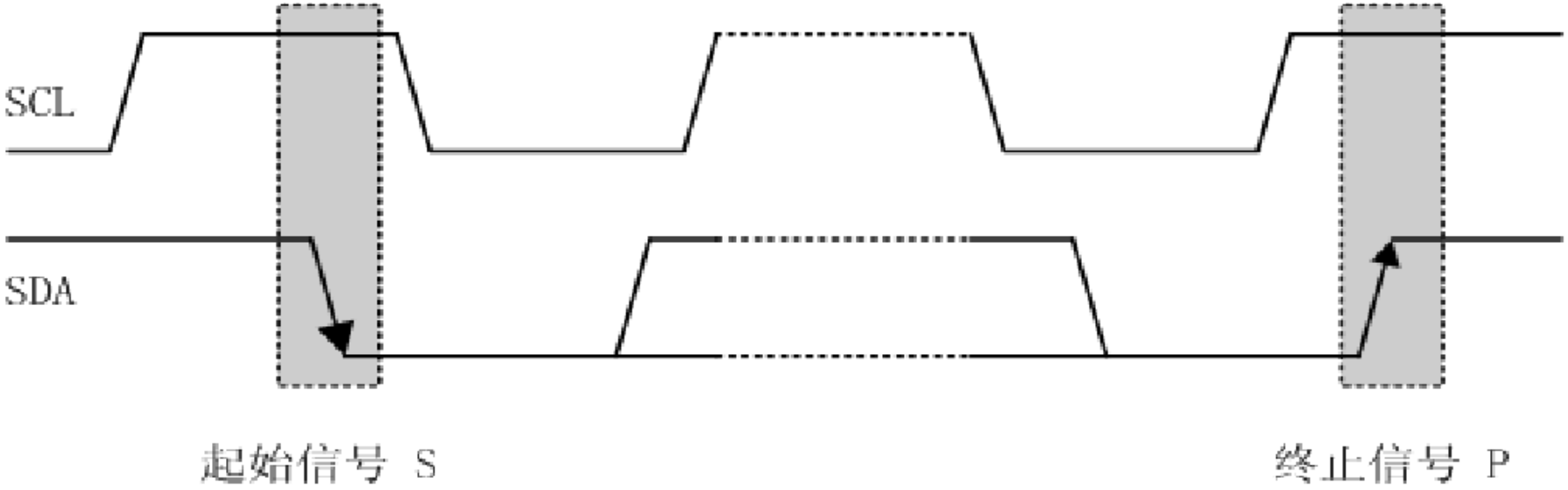
- I²C总线是由串行数据信号线SDA和串行时钟信号线SCL构成的串行总线，可发送和接收数据。
- 采用该总线连接的设备工作在主/从模式下，主器件既可以作为发送器，也可以作为接收器

I²C总线特点

- 每个连接到总线的器件都可以通过唯一的地址和一直存在的简单的主/从机关系软件设定地址，主机可以作为主机发送器或主机接收器。
- 它是一个真正的多主机总线，如果两个或更多主机同时初始化，数据传输可以通过冲突检测和仲裁防止数据被破坏。
- 连接到相同总线的IC数量只受到总线的最大电容400pF限制。

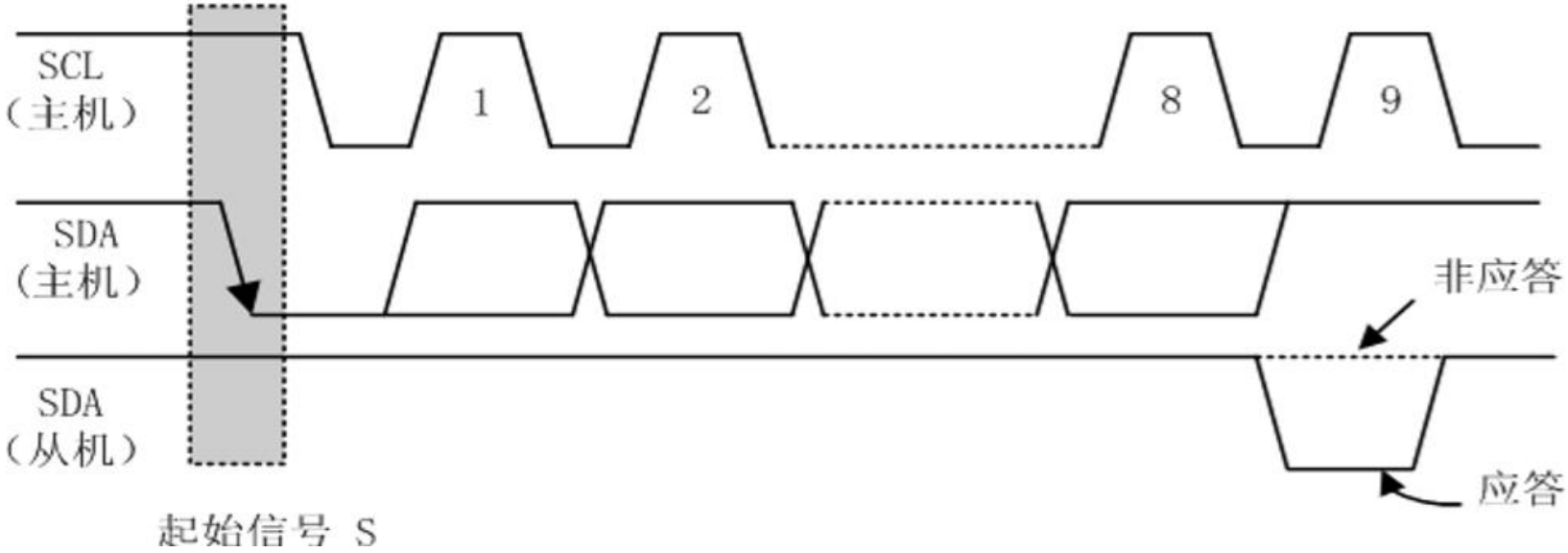
I²C总线传输

- I²C总线的起始和停止条件
  - SCL线为高电平期间，SDA线由高电平向低电平的变化表示起始信号；SCL线为高电平期间，SDA线由低电平向高电平的变化表示终止信号。



I²C总线传输

- I²C总线的数据传送格式
- 每一个字节必须保证是8位长度
- 数据传送时，先传送最高位（MSB），每一个被传送的字节后面都必须跟随一位从机应答位（即一帧共有9位）。



I²C总线传输

- I²C总线的收发同步
  - 接收器件收到一个完整的数据字节后，有可能需要完成一些其它工作，如处理内部中断服务等，可能无法立刻接收下一个字节，这时接收器件可以将SCL线拉成低电平，从而使主机处于等待状态。直到接收器件准备好接收下一个字节时，再释放SCL线使之为高电平，从而使数据传送可以继续进行。

SPI总线

- 同步外设接口（Serial Peripheral Interface，SPI）是由摩托罗拉公司推出的一种高速的、全双工、同步的串行总线
- 接口一般使用四条线：串行时钟线SCLK、主器件输入/从器件输出数据线MISO、主器件输出/从器件输入数据线MOSI和从器件选择线SS
- SPI接口在CPU和外围低速器件之间进行同步的串行数据传输，在主器件的移位脉冲下，数据按位传输，并且高位在前、低位在后，是一种全双工通信。数据传输速度总体上来说比I^2C总线要快，速度可以达到几Mbit/s
- SPI 的工作模式有两种：主模式和从模式，无论那种模式，都支持3Mbit/s的速率，并且还具有传输完成标志和写冲突保护标志

SPI总线特点

- SPI串行外设接口（Serial Peripheral Interface）的最大特点是由主机设备完全控制时钟信号，并决定主从设备通信的启停过程。
- 每次通信中必须有且只有一个设备确定为主设备
- 确定主从设备后，MISO和MOSI信号线传输方向可以根据实际配置改变。
- SPI可轻易扩展成总线配置，单主机驱动多个从机设备。
- 常见的使用SPI接口的器件：ADC、Flash存储芯片
- SPI通信方式灵活，占用口线少，可扩展总线方式，数据传输率高，最高可达36Mbps。
- 但和UART相比抗干扰性差，一般仅用于板上器件间通信

SMBus总线

- 系统管理总线（System Management Bus，SMBus）由Intel提出，应用于移动PC和桌面PC系统中的低速通讯
- SMBus总线同I^2C总线一样也是一种二线式串行总线，它使用一条数据线（SMBDATA）和一条时钟线（SMBCLK）进行通信
- SMBus的目标是通过一条廉价但功能强大的总线，来控制主板上的设备和收集设备的信息
- 虽然SMBus的数据传输率较慢，只有大约100kbit/s，却以其结构简单、造价低的特点，受到业界的普遍欢迎，例如：Windows系统中显示的各种设备制造商名称、型号，主板监控系统中的各种传感器读数，BIOS向监控芯片发送指令等，都是SMBUS实现的。
- SMBus总线大部分基于I^2C总线规范，许多I^2C设备也能够在SMBus上正常工作。 SMBus与I^2C总线之间在时序特性上存在一些差别

清華大學
Tsinghua University

提纲

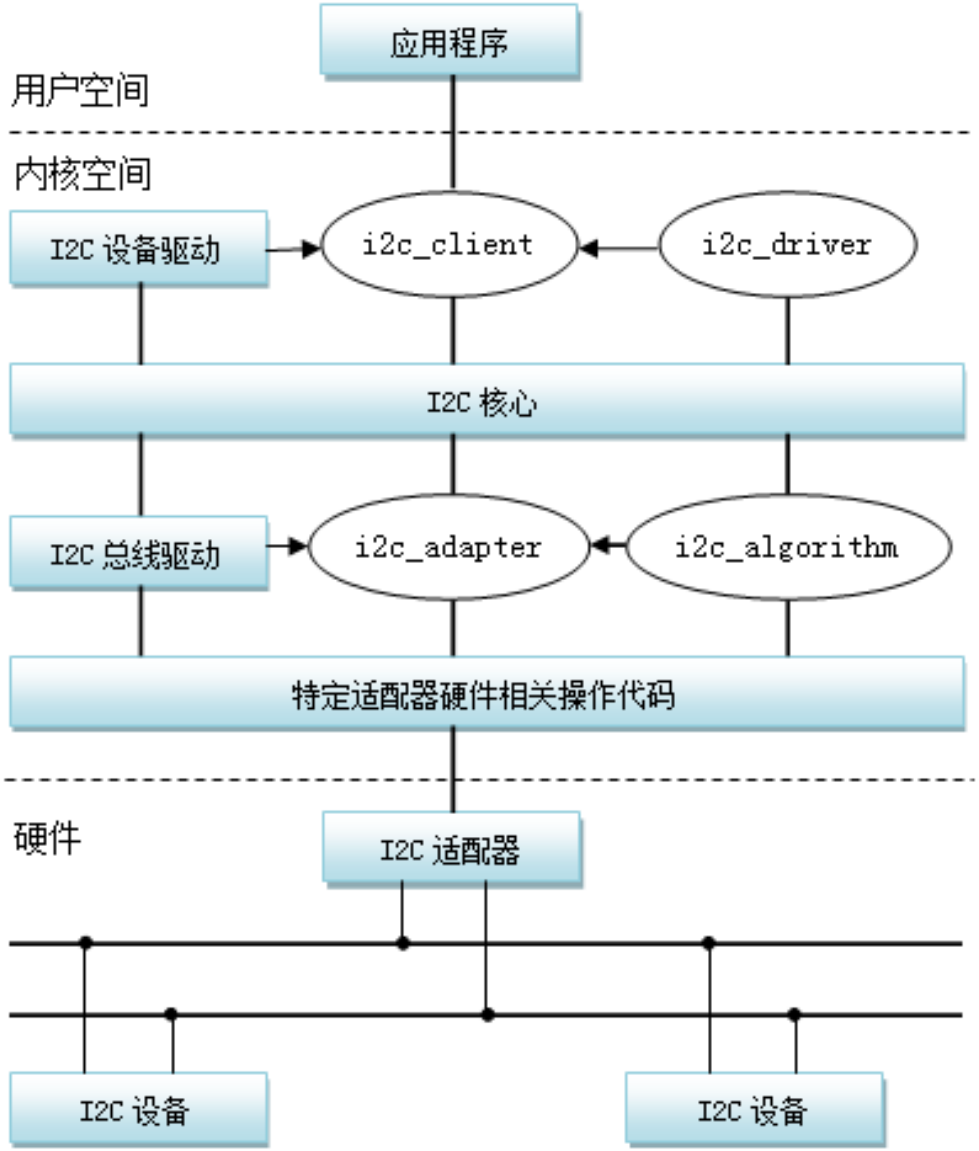
- 字符设备驱动框架
- 字符设备驱动开发
- GPIO驱动概述
- 串行总线概述
- I^2C总线驱动

I²C总线驱动概况

- 为了方便I²C设备驱动的开发，避免因为I²C控制器的硬件差异而导致设备驱动的差异性，linux对I²C总线进行了封装。为 I²C 设备、控制器及驱动提供了统一的注册平台，同时为数据传输提供了统一的接口。
- I²C驱动体系结构分为3个组成部分：
  - I²C核心
  - I²C总线驱动
  - I²C设备驱动

Linux的I²C体系结构

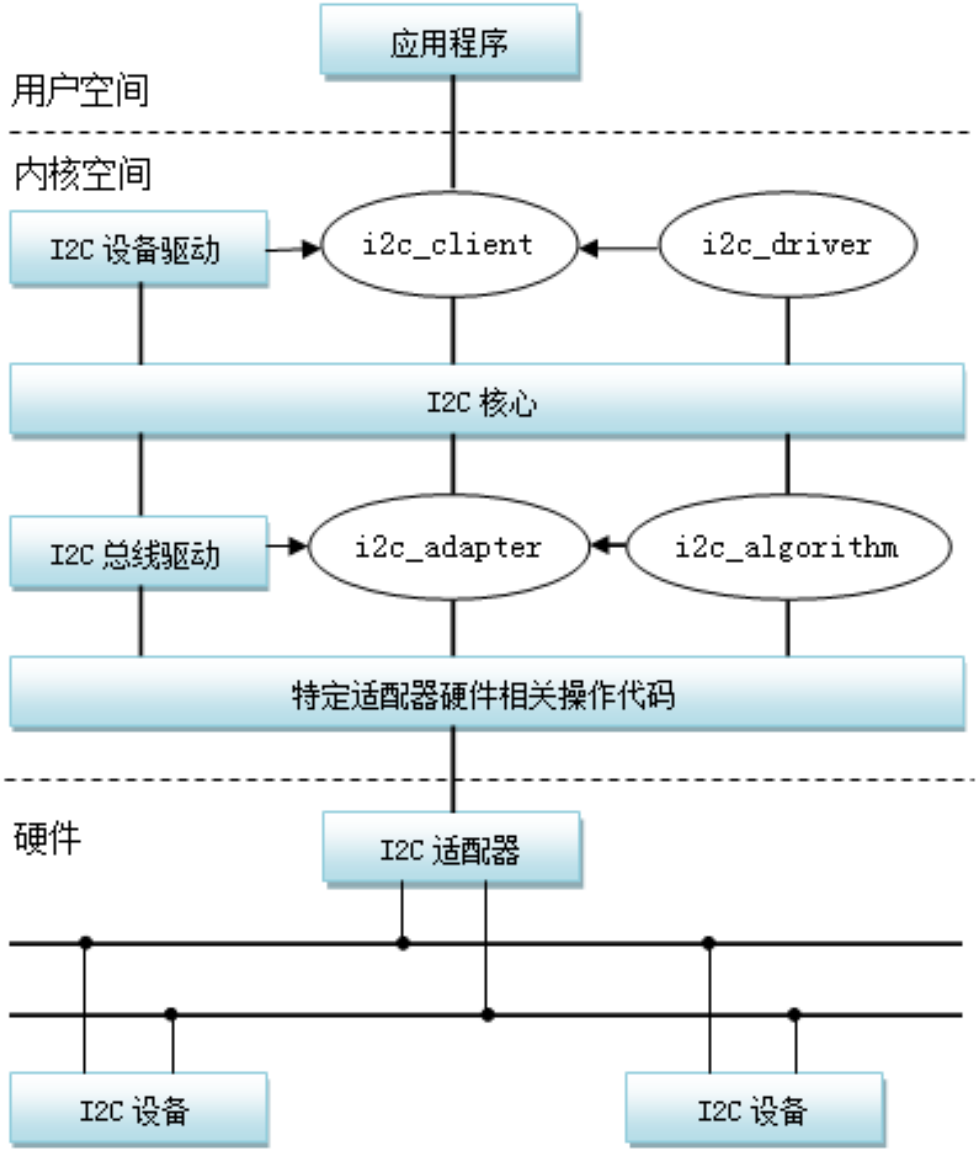
- I²C核心在整个架构中起关键作用，是I²C总线驱动和I²C设备驱动的中间枢纽
- I²C核心提供了I²C总线驱动和设备驱动的注册、注销方法，I²C通信方法（即“algorithm”）与具体适配器无关的上层代码实现，以及探测设备、检测设备地址的上层代码等。



Linux的I²C体系结构

❑I²C总线驱动

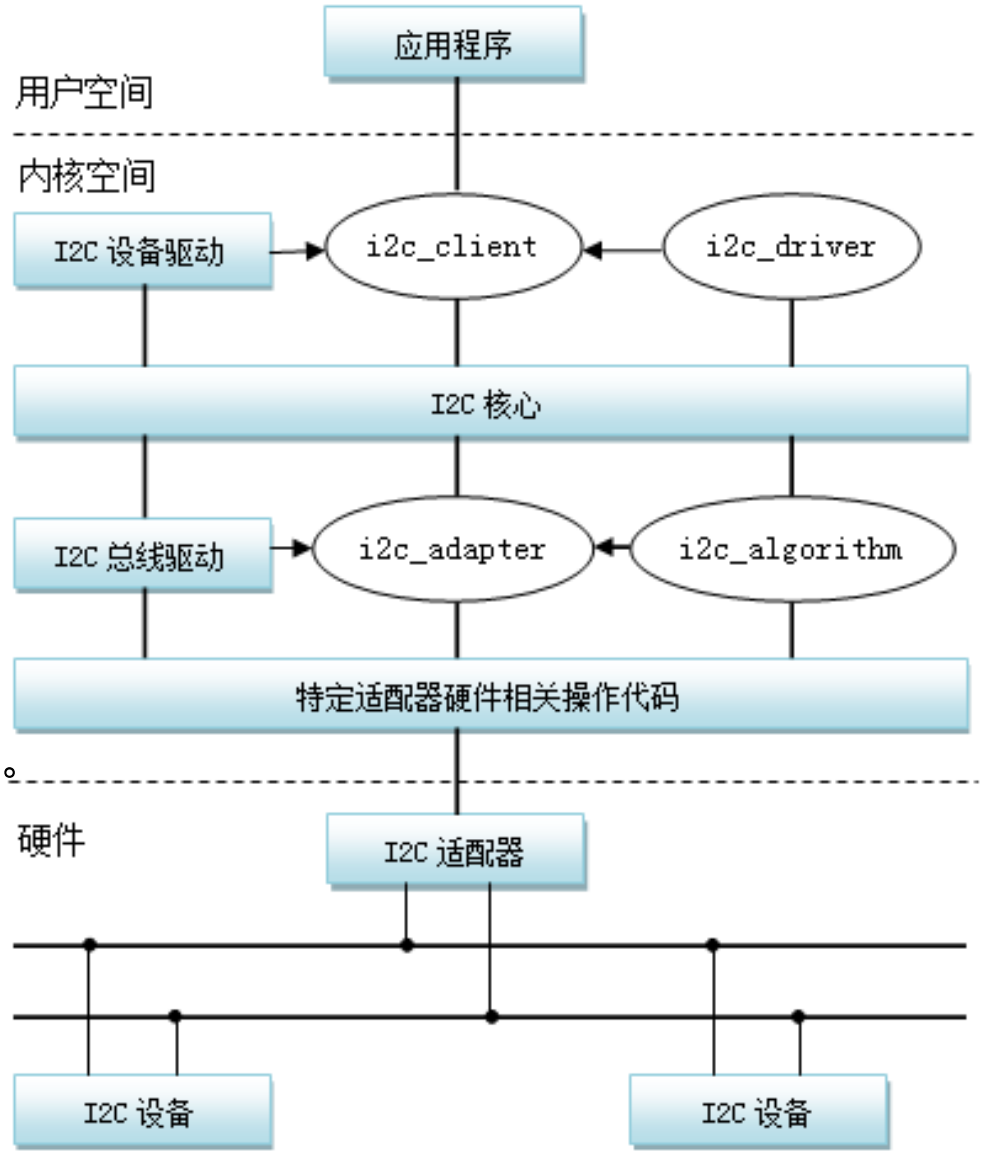
- I²C总线驱动是对I²C硬件体系结构中适配器端的实现，适配器可由CPU控制，甚至直接集成在CPU内部。I²C总线驱动主要包含了I²C适配器数据结构i2c_adapter、I²C适配器的algorithm数据结构i2c_algorithm和控制I²C适配器产生通信信号的函数。
- I²C总线作为一类抽象的总线模型，具体的通信由总线控制器i2c_adapter所提供的总线驱动算法i2c_algorithm 来完成。



Linux的I²C体系结构

❑I²C设备驱动

- I²C设备驱动是对I²C硬件体系结构中设备端的实现，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。I²C设备驱动主要包含了数据结构i2c_driver和i2c_client，我们需要根据具体设备实现其中的成员函数。
- 与用户空间的交互由I²C设备驱动完成，由i2c_dev结构维护i2c_driver结构维护了一类设备的驱动方法，i2c_client结构维护i2c子系统中独立的I²C设备。



I²C总线驱动概况

- Linux系统里I²C实现的主要代码：
  - algos：包含了一些I²C总线适配器的algorithm实现
  - busses：包含了一些I²C总线的驱动，例如AT91的i2c-at91.c
  - chips：包含了一些I²C设备的驱动，例如Dallas公司的DS1682实时钟芯片
  - i2c-boardinfo.c：包含了一些板级信息
  - i2c-core.c/i2c.core.h：实现了I²C核心的功能以及/proc/bus/i2c*接口
  - i2c-dev.c：这是一个通用的驱动，基本上大多数I²C驱动都可以通过调用它操作

I²C总线驱动

- I²C总线驱动的任务，是为系统中各个I²C总线增加相应的读写方法
- I²C总线驱动模块的加载函数负责初始化I²C适配器所要使用的硬件资源，例如申请I/O地址、中断号等，然后通过i2c_add_adapter() 函数注册i2c_adapter结构体，此结构体的成员函数指针已经被相应的具体实现函数初始化
- 当I²C总线驱动模块被卸载时，卸载函数需要释放I²C适配器所占用的硬件资源，然后通过i2c_del_adapter() 函数注销i2c_adapter结构体
- 针对特定的I²C适配器，还需要实现适合其硬件特性的通信方法，即实现i2c_algorithm结构体。主要是实现其中的master_xfer() 函数和functionality() 函数

I²C设备驱动（1）

- 与I²C总线驱动对应的是I²C设备驱动， I²C只有总线驱动是不够的，必须有设备才能正常工作
- I²C设备驱动也分成两个模块，它们分别是i2c_driver和i2c_client结构体。 drivers/i2c/chips目录下已经包含了部分设备的设备驱动代码，负责相应从设备的注册
- i2c-dev.c文件中提供了一个通用的I²C设备驱动程序，实现了字符设备的文件操作接口，对设备的具体访问是通过I²C适配器来实现的。构造一个针对I²C核心层接口的数据结构，即i2c_driver结构体，通过接口函数向I²C核心注册一个I²C设备驱动。同时构造一个对用户层接口的数据结构，并通过接口函数向内核注册一个主设备号为89的字符设备

I²C设备驱动（2）

- 该文件提供了用户层对I²C设备的访问，包括 open、release、read、write、ioctl等常规文件操作，应用程序可以通过open函数打开 I²C的设备文件，通过ioctl函数设定要访问从设备的地址，然后就可以通过read和write函数完成对I²C设备的读写操作。
- i2c-dev.c中提供的i2cdev_read() 和 i2cdev_write() 函数分别实现了用户空间的read和write操作，这两个函数又分别会调用I²C核心的i2c_master_recv() 和 i2c_master_send() 函数来构造一条I²C消息，并且最终调用i2c_algorithm提供的函数接口来完成消息传输

关键数据结构

- i2c_adapter
- i2c_algorithm
- i2c_driver
- i2c_client

i2c_adapter

```
struct i2c_adapter {
        struct module *owner;   /*所属模块*/
        unsigned int id;
        unsigned int class;                   /*用来允许探测的类*/
        const struct i2c_algorithm *algo;  /*I2C algorithm结构体指针    */
        void *algo_data;          /*algorithm所需数据*/
                /*client注册和注销时调用*/
        int (*client_register)(struct i2c_client *) __deprecated;
        int (*client_unregister)(struct i2c_client *) __deprecated;
        int timeout;                          /*超时限制*/
        int retries;          /*重试次数*/
        struct device dev;                    /*适配器设备*/
        int nr;
        struct list_head clients; /* client链表头*/
        char name[48];       /*适配器名称*/
        struct completion dev_released;
};
```

i2c_algorithm

```
struct i2c_algorithm {
        //I2C传输函数指针
        int (*master_xfer)(struct i2c_adapter *adap, struct i2c_msg *msgs,
int num);
        //SMBus传输函数指针
        int (*smbus_xfer) (struct i2c_adapter *adap, u16 addr, unsigned
short flags, char read_write, u8 command, int size, union i2c_smbus_data
*data);
        //确定适配器所支持的功能
        u32 (*functionality) (struct i2c_adapter *);
};

struct i2c_msg {
        __u16 addr;         /*从设备地址*/
        __u16 flags;     /*标志位*/
        __u16 len;                  /*消息长度*/
        __u8 *buf;                  /*消息内容*/
};
```

i2c_client

```
struct i2c_client {
        unsigned short flags;                   /*标志*/
        unsigned short addr;                    /*芯片地址，注意：    7位地址存
储在低7位*/
        char name[I2C_NAME_SIZE];    /*设备名字*/
        struct i2c_adapter *adapter;        /*依附的        i2c_adapter指针
*/
        struct i2c_driver *driver;              /*依附的i2c_driver指针*/
        struct device dev;                      /*设备结构体*/
        int irq;
        struct list_head list;                  /*链表头*/
        struct list_head detected;
        struct completion released;     /*用于同步*/
};
```

i2c_driver

```
struct i2c_driver {
        int id;         /*唯一的驱动id*/
        unsigned int class;
        int (*attach_adapter)(struct i2c_adapter *);/*适配器添加函数（旧式）*/
        int (*detach_adapter)(struct i2c_adapter *);/*适配器删除函数（旧式）*/
        int (*detach_client)(struct i2c_client *) __deprecated;  /*设备删除函          数（旧式）*/
        int (*probe)(struct i2c_client *, const struct i2c_device_id *);
        /*设备添加函数（新式）*/
        int (*remove)(struct i2c_client *);         /*设备删除函数（新式）*/
        void (*shutdown)(struct i2c_client *);      /*设备关闭函数*/
        int (*suspend)(struct i2c_client *, pm_message_t mesg);  /*设备挂起函      数*/
        int (*resume)(struct i2c_client *);         /*设备恢复函数*/
        int (*command)(struct i2c_client *client, unsigned int cmd, void *arg);      /*类似ioctl*/
        struct device_driver driver;              /*设备驱动结构体*/
        const struct i2c_device_id *id_table;     /*此驱动支持的I2C设备列表*/
        int (*detect)(struct i2c_client *, int kind, struct i2c_board_info *);  /*检测函数*/
        const struct i2c_client_address_data *address_data;
        struct list_head clients;                 /*链表头*/
};
```

I²C核心

- 提供了一套接口函数，允许一个I²C adapter、 I²C driver和I²C client在初始化时在I²C Core中进行注册，以及在退出时进行注销。

```
int i2c_add_adapter(struct i2c_adapter *adapter);
int i2c_del_adapter(struct i2c_adapter *adapter);
int i2c_register_driver(struct module *owner, struct i2c_driver *driver);
void i2c_del_driver(struct i2c_driver *driver);
int  i2c_attach_client(struct i2c_client *);
int  i2c_detach_client(struct i2c_client *);
int i2c_transfer(struct i2c_adapter *adap, struct i2c_msg *msgs, int num);
int i2c_master_send(struct i2c_client *client, const char *buf, int count);
int i2c_master_recv(struct i2c_client *client, char *buf, int count);
```


Thank you!