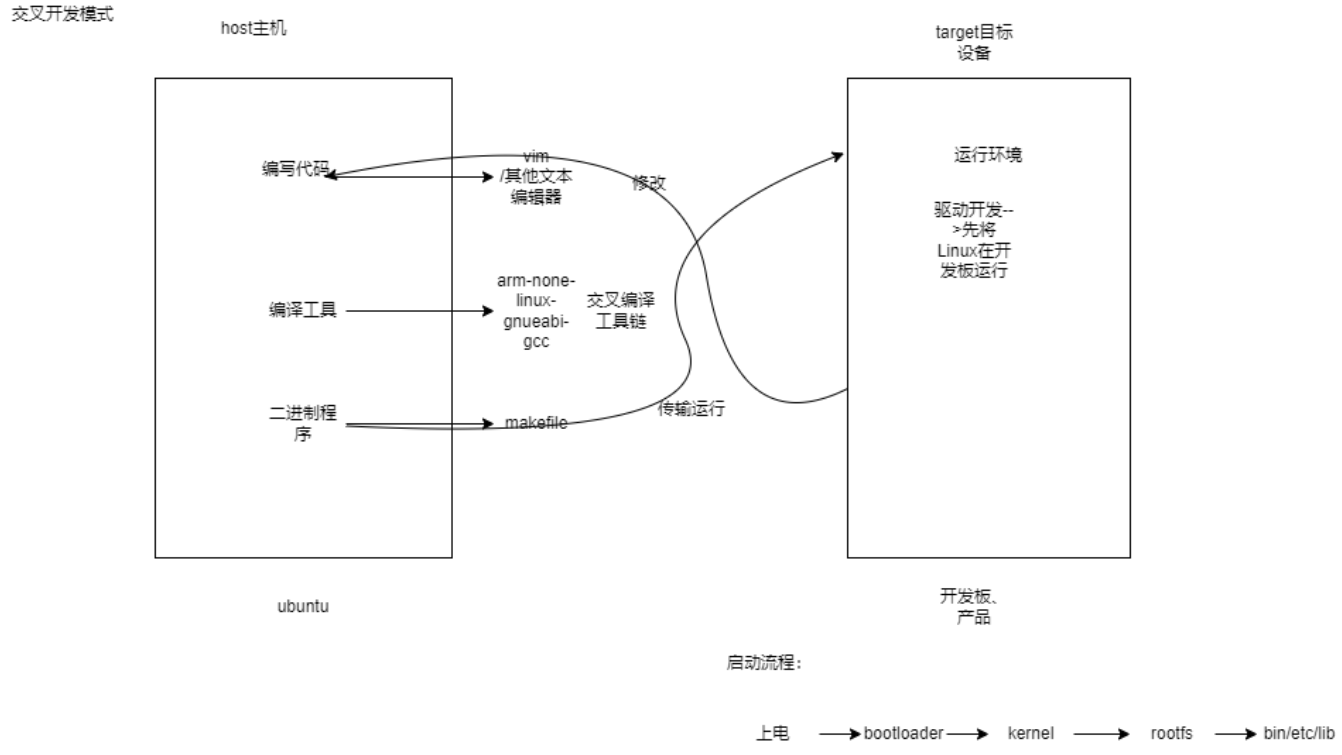
驱动

1.Linux驱动开发环境搭建

1.1 交叉编译环境



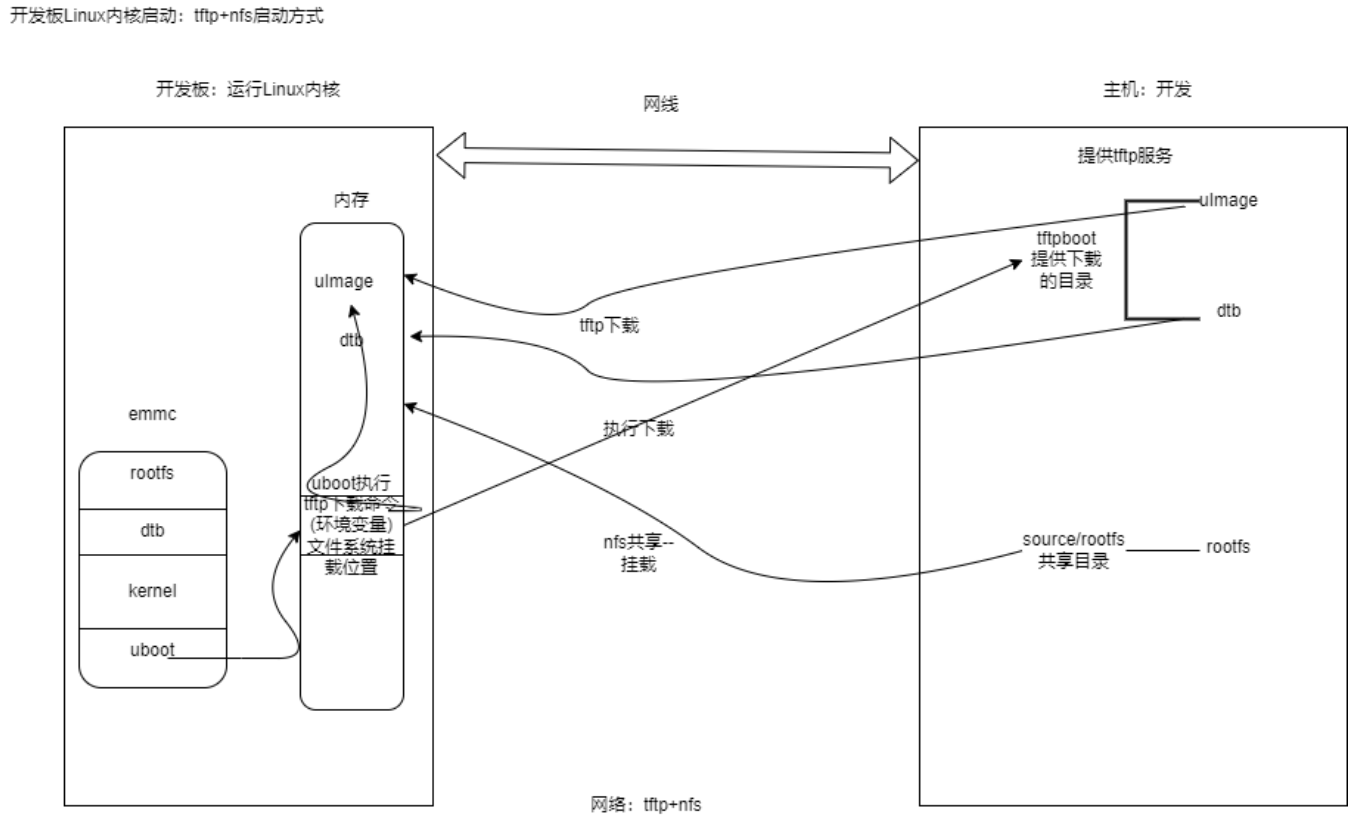
1. ubuntu中环境配置（开发环境）

   设置交叉编译工具链

   1. 拷贝交叉编译工具链到ubuntu系统中
      /home/yky/Tools
   2. 解压交叉编译工具链
      tar xvf gcc-4.6.4.tar.xz
      位置：/home/yky/Tools目录下--：/home/yky/Tools/gcc-4.6.4
   3. 设置环境变量
      vim ~/.bashrc
      最后添加：export PATH=$PATH:/home/yky/Tools/gcc-4.6.4/bin
   4. tftp服务
   5. nfs服务

2. 目标设备配置（运行环境）

需要运行Linux内核
1. 通过tftp去启动内核（通过uboot设置环境变量）
修改bootcmd环境变量 set bootcmd tftp 0×41000000 uImage ; tftp 0×42000000 exynos4412-fs4412.dtb ; bootm 0×41000000 - 0×42000000 修改服务端ip地址 set serverip 192.168.124.85 2. 通过nfs挂载rootfs文件系统 修改bootargs环境变量 set bootargs console=ttySAC2,115200 init=/linuxrc root=/dev/nfs rw nfsroot=192.168.124.85:/source/rootfs ip=192.168.124.250



1.2 代码编写环境

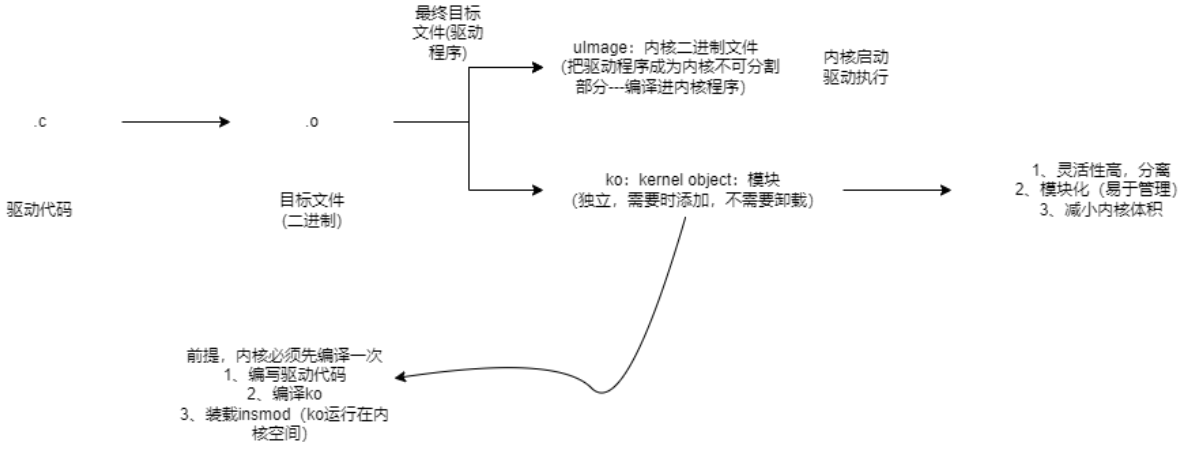
1. 使用什么工具来写驱动代码
   编写驱动时，所使用的API函数都是内核中提供的函数(只有内核代码才有)，只能查看内核源码才能找到函数的使用(函数的原型定义)，在编写驱动代码时要查看内核函数功能代码
   source insight(查看代码的工具)

2. 安装source insight工具

   1. 找到软件提示把工具安装激活
   2. 把Linux内核代码解压到windows目录中

3. 打开工具添加查看的项目
   project——→new project:
   第一个对话框：
   第一个文本框：输入工程名字 第二个文本框：路径(默认) 第二个对话框： 在项目源码位置，选择 Linux内核代码位置 点击OK 第三个对话框： 选择需要查看的Linux内核代码的包含的源码文件（要查看的内核目录）
   需要添加的目录：
   arch/arm/kernel
   arch/arm/include/asm
   driver/base
   driver/char
   driver/iic
   deriver/spi
   include
   kernel
   init
   fs/char_dev.c(文件)
   点击close 退出,
   然后 重新打开刚才创建的工程 project→open project，如果提示 同步,点击确定,可能需要很长时间

3. 编写代码
   使用source insight工具编写代码，然后把这个程序拷贝到ubuntu系统中，使用交叉编译工具进行编译，生成适配与开发板的程序
   编写驱动准备：



4. 驱动模块编写(.ko) 驱动模块代码需要有四个部分

```
//模块
```

```c
/*
1、头文件
2、驱动模块装载入口和卸载入口声明
3、模块装载函数和卸载函数
4、GPL声明

装载入口:当内核加载这个驱动模块时，从那个函数执行(声明,实现)

*/

//头文件
#include <linux/init.h>
#include <linux/module.h>

//模块加载入口函数实现
static int __init 函数名1(void)
{
        //资源的创建，申请，创建驱动
        return 0;
}

//模块卸载入口函数实现
static void __exit 函数名2(void)
{
        //资源的释放，删除驱动
}


//模块入口声明
//装载声明(内核加载的入口指定)
module_init(函数名1);//只要加载就执行其中声明的函数

//卸载声明(内核卸载的入口指定)
module_exit(函数名2);

//GPL开源声明
MODULE_LICENSE("GPL");
```

编译驱动模块代码

```
KERNEL_PATH=/home/yky/Code/linux-3.14-fs4412 #内核中的Makefile需要配置交叉编译工具链

obj-m += 模块文件名.o #要编译为模块的文件

all:
```

```
	make modules -C $(KERNEL_PATH) M=$(shell pwd) #借助已经编译好的内核，编译模块


# -C 指定内核路径
# M:当前模块的位置
```

5. 设备加载ko
   insmod：加载模块

   ```
   insmod  模块路径
   ```

   rmmod：卸载模块

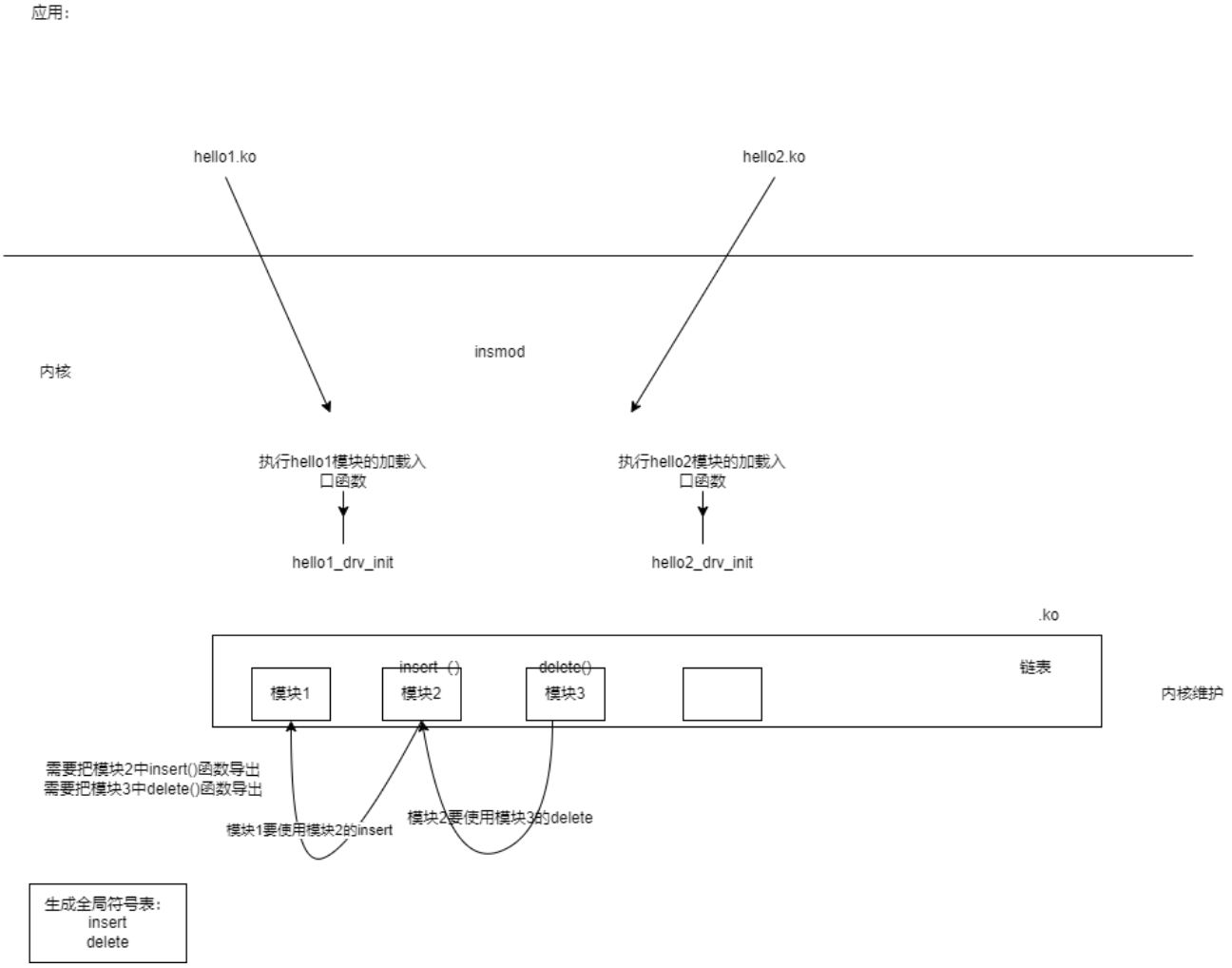
   ```
   mkdir  /lib/modules
   mkdir /lib/modules/3.14.0
   rmmod  模块名
   ```

   lsmod：查看已经加载的模块

2. 字符设备驱动编写

2.1 驱动模块

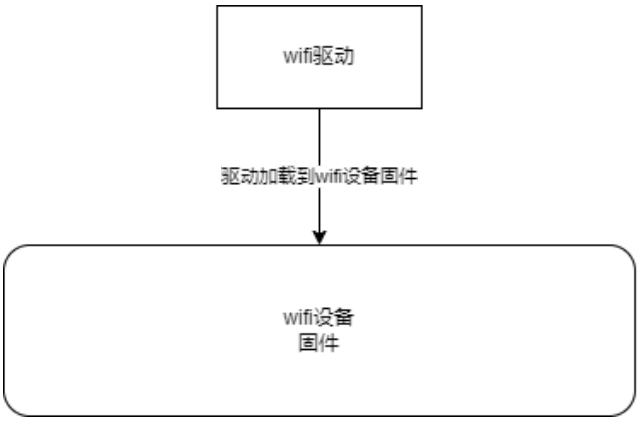
1. 符号导出



如果模块中内容需要在其他模块中使用，可以把内容进行导出：添加导出声明
EXPORT_SYMBOL(内容名字);
在要使用的模块中进行声明

如果模块只用于进行导出，可以不写入口声明以及定义

2. 参数传递
   在编写驱动时，有些变量是不确定的，是根据驱动具体加载到哪个设备才确定，在进行insmod装载驱动模块时再传递这些参数值
   在驱动代码中如何处理参数传递： module_param(name，type，perm);
   参数1：参数的名字，变量名
   参数2：参数的类型，int，char
   参数3：/sys/modules 文件的权限，0666



wifi硬件中也运行了内部的硬件代码，wifi厂商开发-----固件
加载wifi驱动时，需要告诉驱动，固件在哪里？

加载模块：
insmod perm.ko a=10 b=5 p="okokok"

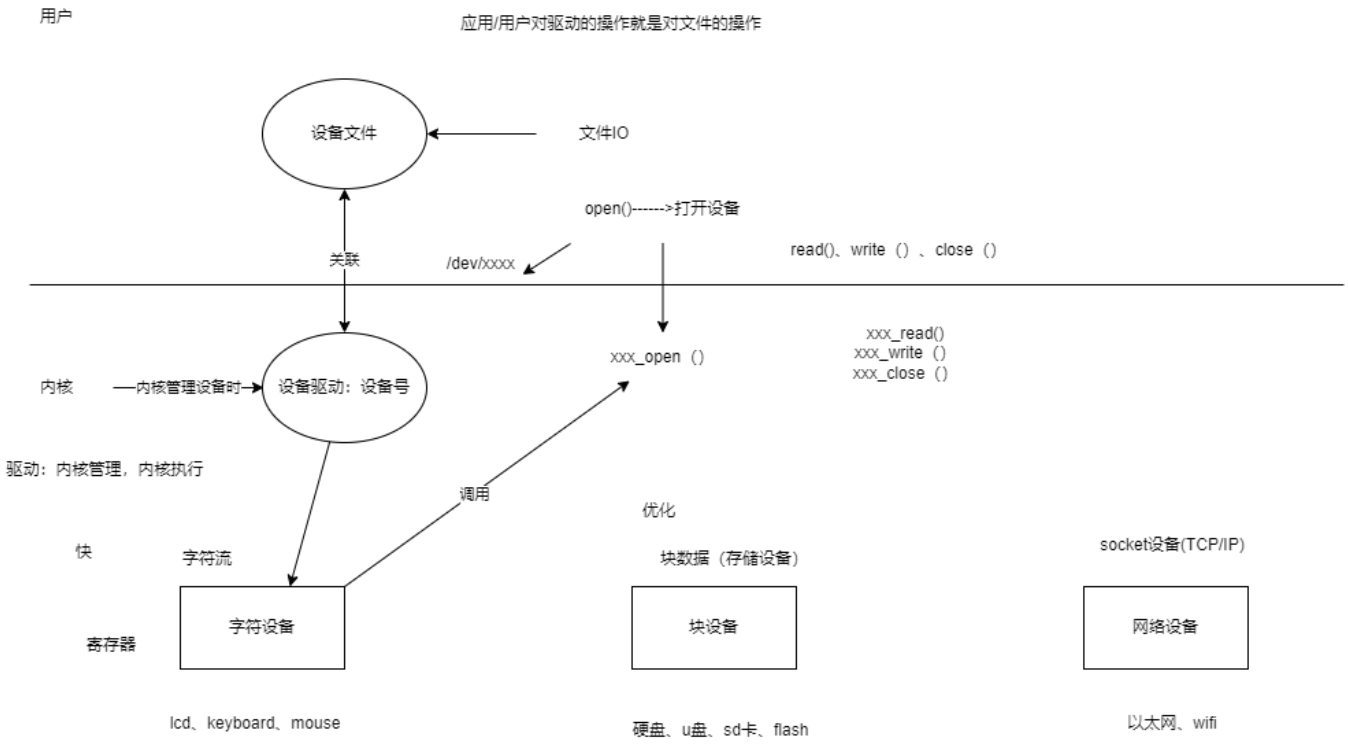
2.2 字符设备驱动

2.2.1 作为字符设备驱动要素：

1. 必须有一个设备号，用于在内核中，众多的设备驱动进行区分

2. 用户（应用）必须知道设备驱动对应到哪个设备文件(设备节点)
   Linux一切皆文件（把所有的设备都看作是文件）

3. 对设备进行操作(驱动)，其实就是对文件进行操作，应用空间操作open、read、write等文件IO时，实际上驱动代码中对应执行的open、read、write函数



2.2.2 作为驱动必须有设备号----向内核（系统）申请

```
//申请设备号
int register_chrdev(unsigned int major, const char *name,const struct
file_operations *fops)
参数：
    参数1：
        unsigned int major：主设备号
            设备号：32bit == 主设备号(12bit) + 次设备号(20bit)
            主设备号：表示同一类型的设备
            次设备号：表示同一类型中的不同设备

            参数有两种设置：
            静态：指定一个整数：250----主设备号为250
            动态：让内核随机指定,参数为0
    参数2：
        const char *name:一个字符串，描述设备信息，自定义
    参数3：
        const struct file_operations *fops：结构体指针，结构体变量的地址（结构体中就是应
用程序和驱动程序函数关联，open、read、write）---文件操作对象，提供open、read、write等驱动
中的函数


返回值：
    如果是静态指定主设备号，返回0表示申请成功，返回负数表示申请失败
    如果是动态申请，返回值就是申请成功的主设备号
    /proc/devices:文件中包含了所有注册到内核的设备
```

```
//销毁注销设备号
void unregister_chrdev(unsigned int major,const char * name)
参数:
    参数1:
        unsigned int major: 主设备号
    参数2:
        const char * name: 设备信息，自定义
```

2.2.3 创建设备节点(设备文件)

1. 手动创建

```
# 创建设备节点---手动
mknod   设备节点名 设备类型   主设备号   次设备号
#如:
mknod   /dev/xxx        c     250       0
```

2. 自动创建（通过udev/mdev机制）

```
//创建一个类(信息)
struct class * class_create(owner,name)
参数:
    参数1:
        owner: 一般填写 THIS_MODULE
    参数2:
        name: 字符串首地址，名字，自定义

返回值:
    返回值就返回信息结构体的地址

//创建设备节点(设备文件)
struct device * device_create(struct class *class, struct device *parent,dev_t
devt, void *drvdata, const char *fmt, ...)
参数:
    参数1:
        struct class *class: class信息对象地址，通过 class_create()函数创建
    参数2:
        struct device *parent: 表示父亲设备，一般填 NULL
    参数3:
        dev_t devt: 设备号（主设备+次设备）
            #define MAJOR(dev)  ((unsigned int) ((dev) >> MINORBITS))
            #define MINOR(dev)  ((unsigned int) ((dev) & MINORMASK))
```

```
        #define MKDEV(ma,mi)        (((ma) << MINORBITS) | (mi))
    参数4：
        void *drvdata：私有数据，一般填NULL
    参数5、参数6：
        const char *fmt, ...：可变参数，表示字符设备文件名(设备节点名)


//销毁
//销毁设备节点
void device_destroy(struct class * class,dev_t devt)
```

3. 在驱动中实现文件io的接口，应用程序调用对应的文件io函数，驱动中如何调用

    1. 在驱动中实现文件io的接口操作
       const struct file_operations fops;结构体中就是 驱动 与 应用程序文件io的接口关联

```
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 *);
ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long,
loff_t);
ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long,
loff_t);
int (*iterate) (struct file *, struct dir_context *);
unsigned int (*poll) (struct file *, struct poll_table_struct *);
long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
int (*mmap) (struct file *, struct vm_area_struct *);
int (*open) (struct inode *, struct file *);
int (*flush) (struct file *, fl_owner_t id);
int (*release) (struct inode *, struct file *);
int (*fsync) (struct file *, loff_t, loff_t, int datasync);
int (*aio_fsync) (struct kiocb *, int datasync);
int (*fasync) (int, struct file *, int);
int (*lock) (struct file *, int, struct file_lock *);
ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long,
unsigned long, unsigned long);
int (*check_flags)(int);
int (*flock) (struct file *, int, struct file_lock *);
ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *,
size_t, unsigned int);
```

```
ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *,
size_t, unsigned int);
int (*setlease)(struct file *, long, struct file_lock **);
long (*fallocate)(struct file *file, int mode, loff_t offset,
                  loff_t len);
int (*show_fdinfo)(struct seq_file *m, struct file *f);
};
//函数指针集合，其实就是接口，表示文件IO函数用函数指针来表示，存储驱动中的关联函数
// 如：  read函数指针 =  xxxx_ok；表示 read函数，在驱动中的关联函数为xxxx_ok函数
```

2. 应用调用去调用文件io去控制驱动

```
open();
read();
write();
```

4. 应用程序需要传递数据给驱动

将驱动空间拷贝数据给应用空间

```
//这个功能一般用于驱动中的 read
long copy_to_user(void __user *to,const void *from, unsigned long n)
参数：
    参数1：
        void __user *to：目标地址，应用空间地址
    参数2：
        const void *from：源地址，内核空间地址
    参数3：
        unsigned long n：个数
返回值：
    成功返回0，失败返回大于0，表示还有多少个没有拷贝完
```

将应用空间数据拷贝到驱动空间

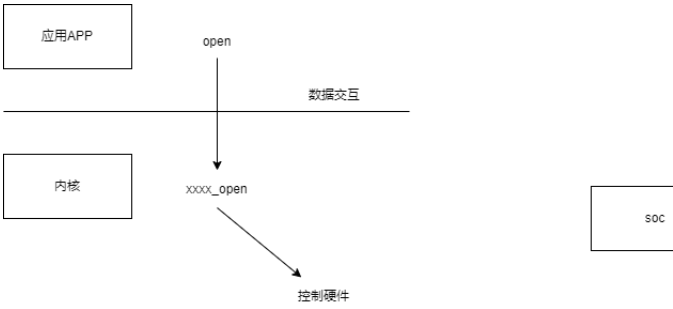
```
long copy_from_user(void * to,const void __user * from,unsigned long n)
参数：
    参数1：
        void *to：目标地址，内核空间地址
    参数2：
        const void *from：源地址，应用空间地址
    参数3：
        unsigned long n：个数
```

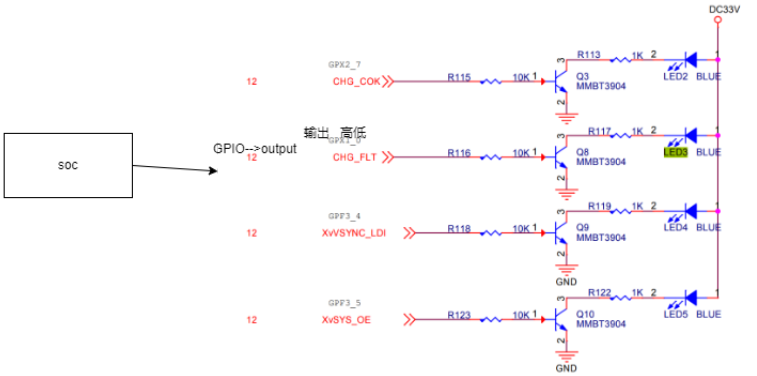
返回值：
    成功返回0，失败返回大于0，表示还有多少个没有拷贝完

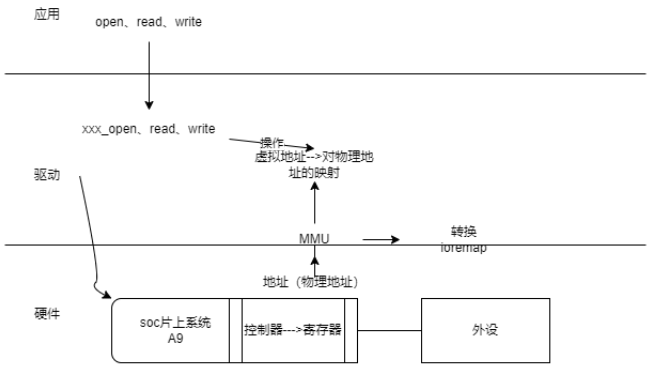
5. 控制外设，其实就是控制地址，内核驱动中通过虚拟地址操作

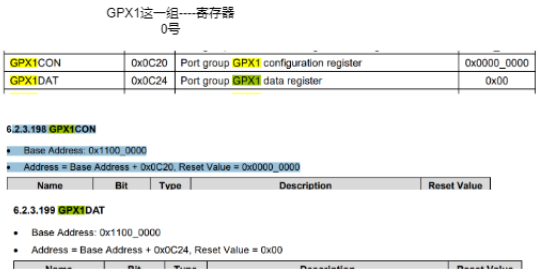
需要把外设控制的物理地址，映射到内核空间

```
//虚拟地址映射
void *ioremap(phys_addr_t offset, unsigned long size)
参数：
    参数1：
        phys_addr_t offset：物理地址
    参数2：
        unsigned long size：映射大小

返回值：
    映射之后的虚拟地址
//接触映射
void iounmap(void *addr)
```









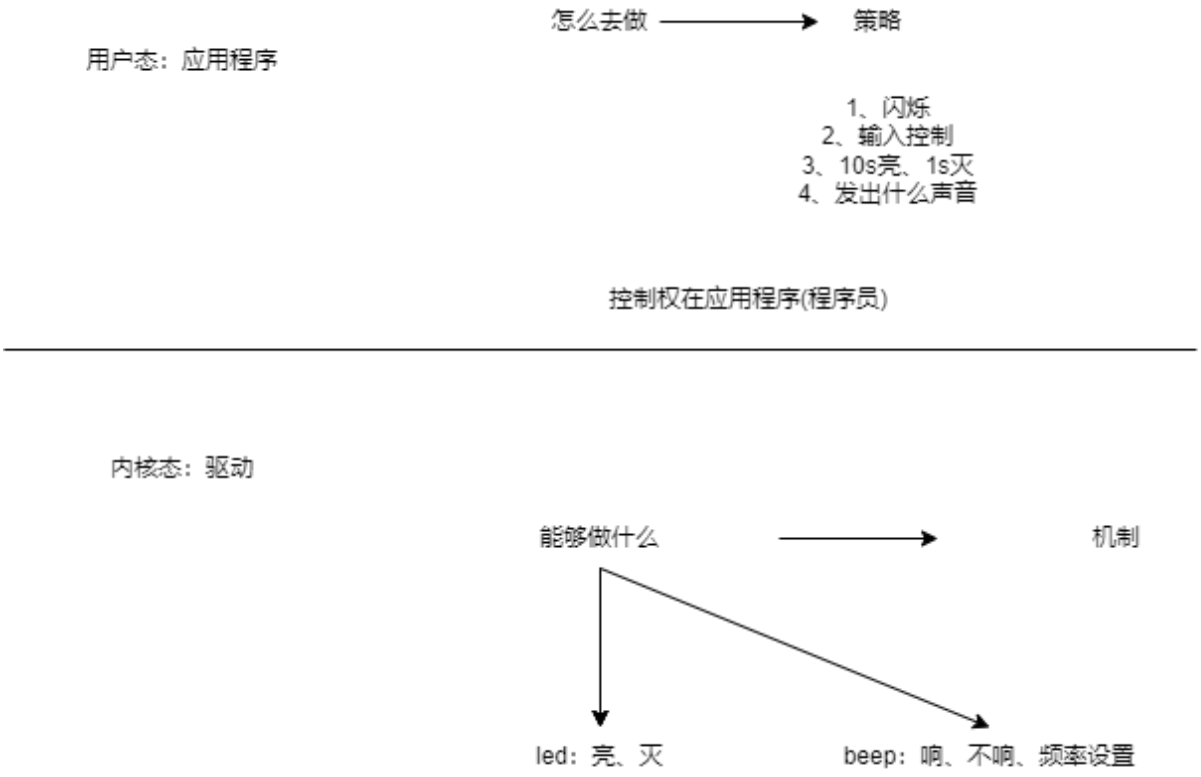
6. 操作寄存器地址的方式

    1. 通过指针取 *
    volatile unsigned int * gpx1con;
    *gpx1con---进行操作

2. readl()、writel()

```
//从地址中读取地址空间的值
u32 readl(const volatile void *addr)

//将value值，存储到对应地址中
void writel(u32 value, volatile void *addr)
```

7. 应用程序和驱动程序扮演什么角色



8. 编写字符设备驱动的步骤

1. 实现模块加载和卸载入口函数

```
static int __init beep_init(void)//加载入口
{

}
static void __exit beep_exit(void)//卸载入口
{

}
```

```
module_init(beep_init);
module_exit(beep_exit);
```

2. 在模块加载入口
    1. 申请设备号（内核中用于区分和管理不同的字符设备驱动）
    2. 创建设备节点（为用户提供一个可操作的文件接口---用户操作文件）
    3. 实现硬件初始化
        1. 地址的映射
        2. 实现硬件的寄存器的初始化
        3. 中断的申请
    4. 实现文件io接口（struct file_operations fops结构体）

9. 规范

    1. 在加载入口实现资源申请，需要在卸载入口实现资源释放

```
//申请资源
register_chrdev();//申请设备号
class_create();
device_create();//创建设备节点
ioremap();//硬件资源映射
------------------------
//释放资源
iounmap();//解除硬件资源映射
device_destroy();//释放设备节点
class_destroy();//释放节点信息类
unregister_chrdev();//释放设备号
```

    2. 出错处理
    在某个位置出错，要将之前申请的资源进行释放

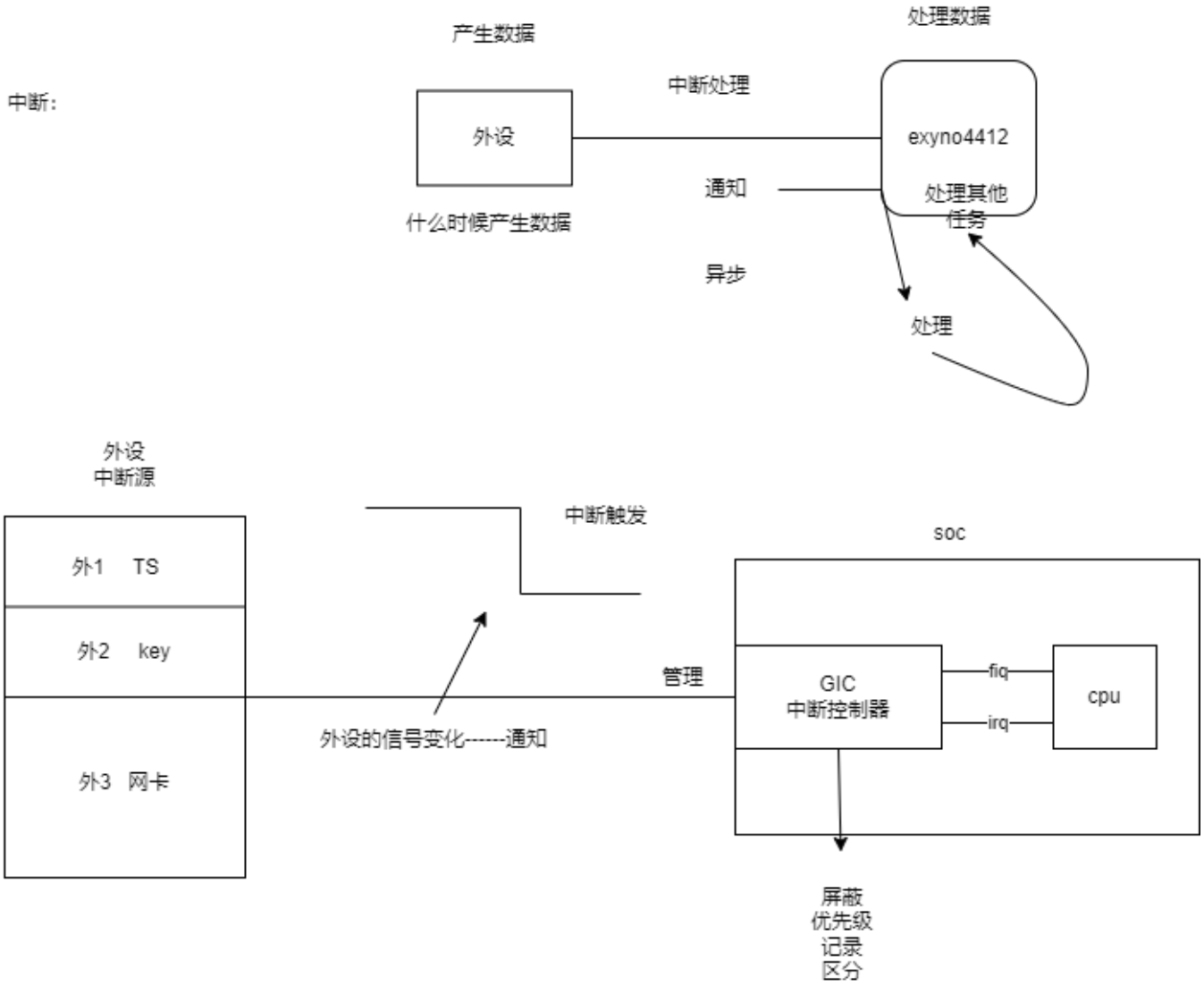
    3. 面向对象编程思想
    用一个结构体来表示一个对象
    设计一个类型，描述一个设备的信息

```
struct BEEP
{
        unsigned int major;
        struct class * cls;
        struct device * dev;
        unsigned int * pwmtcfg0;
```

```
};
struct BEEP beep;//表示一个设备对象
```

2.3 中断驱动编写(驱动之中断编程)

2.3.1 中断处理



2.3.2 中断驱动编写

1. 中断号：硬件设备连接到芯片，芯片已经对每个引脚设置好对应的中断编号，对应硬件的中断通过引脚查看
   如：k3按键
   硬件连接： key3-------——> GPX1_2-----——>XEINT10
   在Linux3.14内核中，由于存在设备树(描述设备信息)，可以在设备树中添加一个按键设备信息
   在完成按键设备信息中，添加上使用的中断号，启动内核时，内核中就有按键信息(中断号)
   修改要下载到硬件设备的设备树，做设备信息的添加
   设备树位置：
   linux-3.14-fs4412/arch/arm/boot/dts
   在编程过程中，需要添加中断设备节点信息--描述当前设备的中断号

```
key_int_node {
    compatible = "key3";
    interrupt-parent = <&gpx1>;
    interrupts = <2 4>;
};
```

2. 在驱动中获取到中断号，并申请中断处理
   1. 获取到中断号

```
//根据路径获取设备树中的节点
struct device_node *of_find_node_by_path(const char *path);
参数：
    参数1：
        const char *path：设备树中节点路径
返回值：
    获取到的节点

//根据节点获取到节点的中断号
unsigned int irq_of_parse_and_map(struct device_node *dev,int index)
参数：
    参数1：
        struct device_node *dev：设备节点
    参数2：
        int index：获取节点中第几个中断号
返回值：
    获取到的中断号
```

   2. 申请中断处理(内核检查到对应的中断，驱动申请进行处理)

```
//申请中断：对应终端号出现对应的触发方式，就调用申请中指定的函数进行处理
int request_irq(unsigned int irq, irq_handler_t handler, unsigned long
flags,const char *name, void *dev)
参数：
    参数1：
        unsigned int irq：设备对应的中断号
    参数2：
        irq_handler_t handler：中断的处理函数
            typedef irqreturn_t (*irq_handler_t)(int, void *);
    参数3：
        unsigned long flags：中断的触发方式
        #define IRQF_TRIGGER_NONE       0x00000000
        #define IRQF_TRIGGER_RISING     0x00000001
        #define IRQF_TRIGGER_FALLING    0x00000002
        #define IRQF_TRIGGER_HIGH       0x00000004
        #define IRQF_TRIGGER_LOW        0x00000008
    参数4：
        const char *name：中断的描述
    参数5：
        void *dev：传递给参数2这个函数指针的参数
返回值：
    成功返回0，失败返回非0

//释放中断
void free_irq(unsigned int irq,void * dev_id);
与申请中断，参数1和参数5一致
```

3. 驱动中将硬件产生的数据传递给用户

    1. 硬件如何获取到数据
       如：按键
       key：按下和抬起-——→ 1/0
       读取key对应的gpio的状态——→gpio数据寄存器的值
    2. 驱动如何传递用户
       应用程序和驱动通过文件io接口进行关联，驱动中只要把数据通过文件io接口传递
       驱动实现xxx_read等文件io接口
    3. 用户如何拿到数据---调用对应的文件io函数

```
open();
read();---
close();
```

4. 文件io模型实现

文件io模型： 阻塞 、非阻塞 、io多路复用 、异步通知

1．阻塞：等同于休眠，当进程在读取外部设备的资源(数据)，如果资源没有准备，进行就会休眠等待

在linux应用中，大部分的函数接口都是阻塞的

scanf();read();write();accept();

在驱动中实现文件IO模型功能

驱动中如何写代码：

1、等待队列头

//根据等待队列头创建等待队列

//等待队列头类型：wait_queue_head_t

init_waitqueue_head(wait_queue_head_t * q);

2、在需要的位置进行休眠等待

wait_event_interruptible(wait_queue_head_t wq,condition)

参数1：

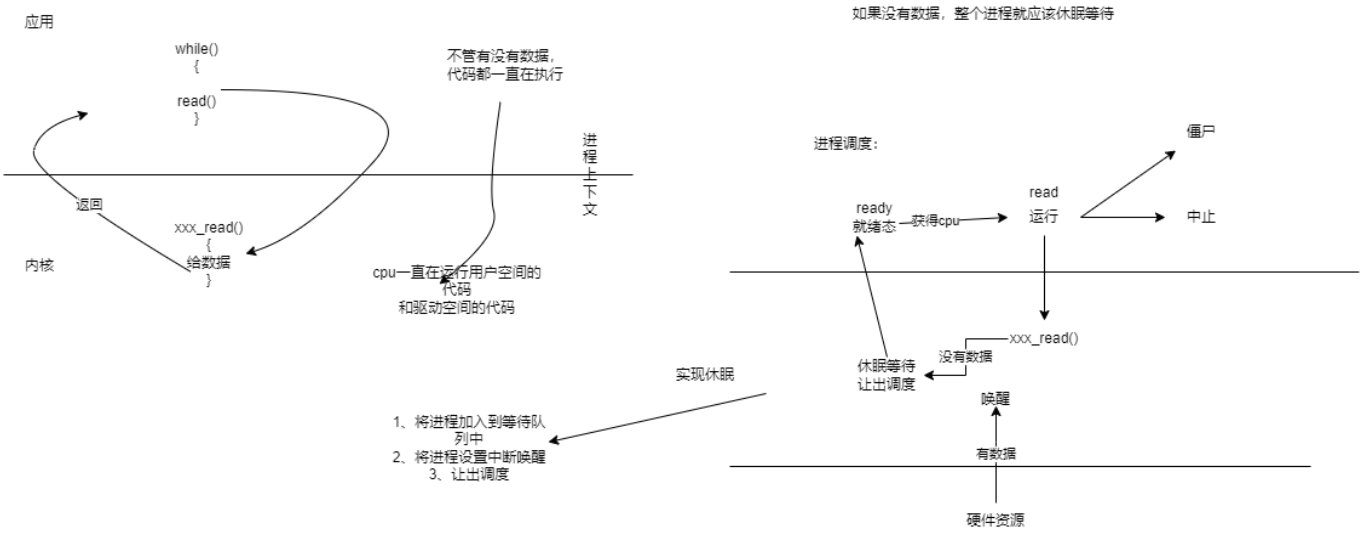
wait_queue_head_t wq：等待队列头

参数2：

condition：条件，为假，就会等待；为真，就不会等待

3、在合适的位置(有数据)，进行进程唤醒

wake_up_interruptible(wait_queue_head_t * q);



2．非阻塞：在读写的时候，如果没有数据，就立即返回

应用需要设置为非阻塞：

```
int fd = open("/dev/key3",O_RDONLY|O_NONBLOCK);

int status = fcntl(fd,F_GETFL);
```

```
status = status | O_NONBLOCK;
fcntl(fd,F_SETFL,status);
```

驱动中需要区分当前模式是否为非阻塞模式：

如果是非阻塞模式，且没有数据就立即返回

```
if(filep->f_flags & O_NONBLOCK && !key.key_state)//非阻塞
{
        return -EAGAIN;
}
```

3. 异步通知：当有数据的时候，驱动就会发送信号给(SIGIO)给应用，应用就可以异步去读写数据，不用主动读写

驱动：发送信号

1、需要和进程进行关联---信号发送给谁

实现一个fasync接口

```
int key_fasync (int fd, struct file * filep, int on)
{
    return fasync_helper(fd,filep,on,&(key.fasync));
    }
```

2、在某个特定的时刻发送信号，有数据的时候

```
//发送信号
    kill_fasync(&(key.fasync),SIGIO,POLLIN);
```

应用程序：处理信号，主要是读写数据

1、设置信号怎么处理

```
    signal(SIGIO,catch_signal);
  void catch_signal(int signo)
{
    if(signo == SIGIO)
    {
            int value = -1;
            read(fd,&value,4);
            printf("program data is %d\n",value);
    }
}
```

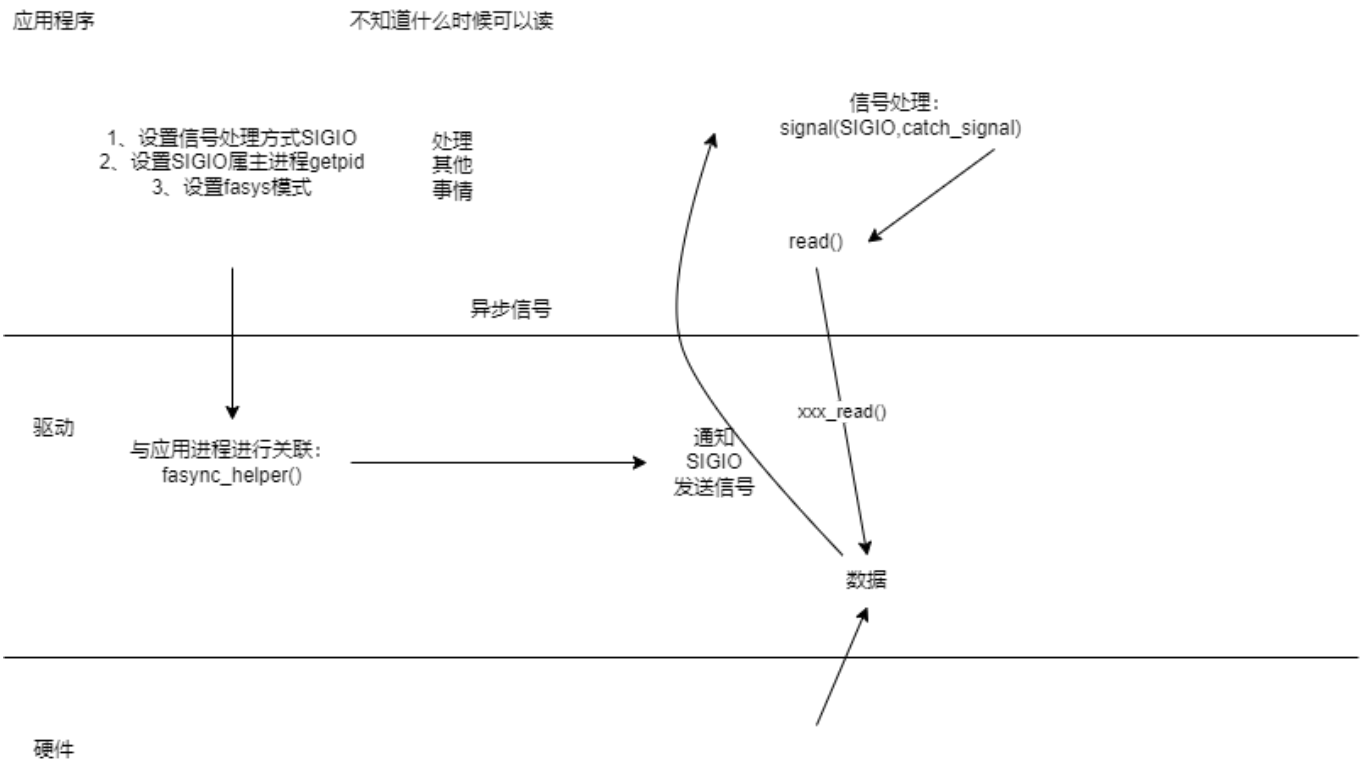
2、将当前进程设置为SIGIO的属主进程

```
//设置当前进程为信号的属主进程
    fcntl(fd,F_SETOWN,getpid());
```

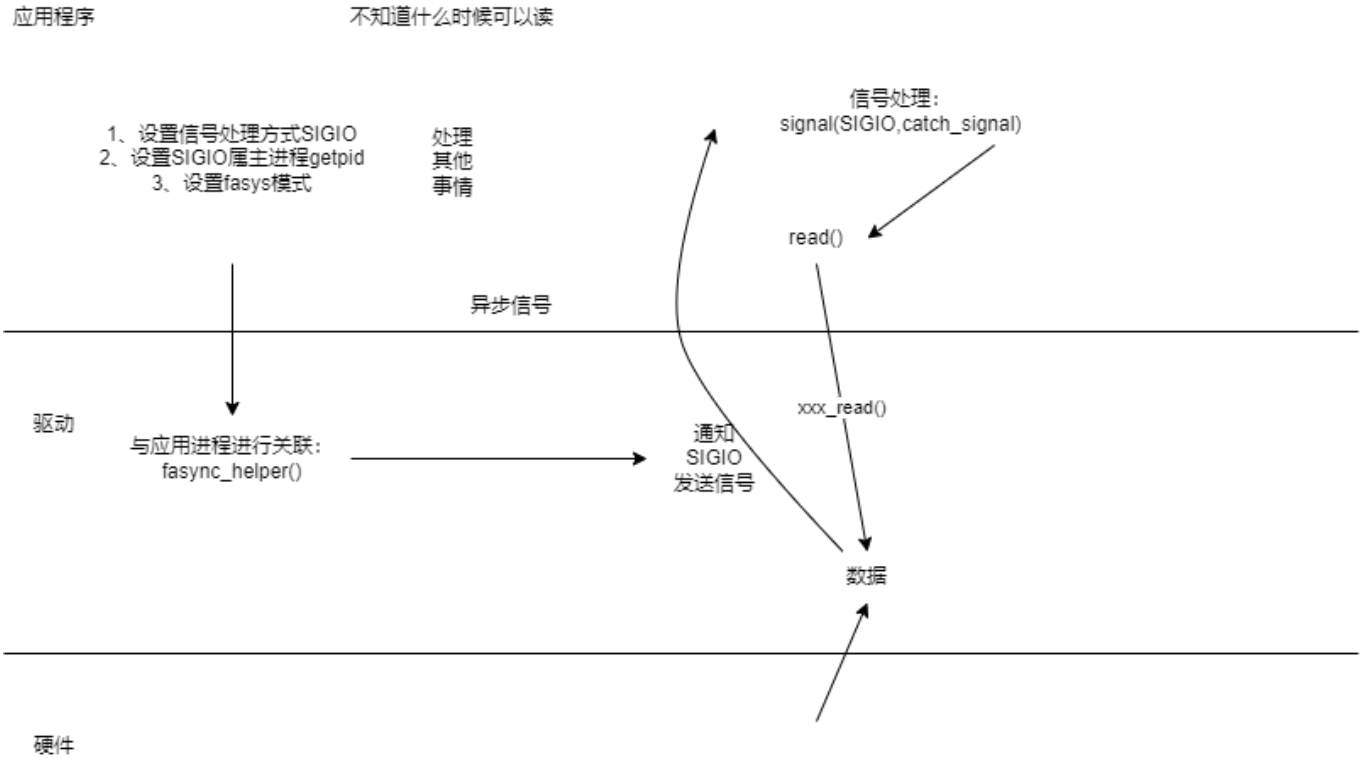
3、将io模式设置为异步模式

```
//将io模式设置为异步模式
```

```
int stats = fcntl(fd,F_GETFL);
stats = stats | FASYNC;
fcntl(fd,F_SETFL,stats);
```



5. 中断下半部分



1. sorfirq：处理速度比较块，但是是内核级别的机制，要修改内核源码，，不推荐也不常用 2. tasklet：内部实现调用sortirq 3. workqueue：工作队列

a、tasklet：

```
1. 初始化，设置tasklet任务结构体
struct tasklet_struct tasklet;
tasklet_init(&任务结构体的地址,中断下半部分执行函数,函数参数);
2. 启动中断下半部分----把任务放入到内核线程中
//中断下半部分启动--放入内核线程
tasklet_schedule(&tasklet);
```

b、workqueue

```
1. 初始化，设置workqueue任务结构体
struct work_struct workqueue;
INIT_WORK(&任务结构体的地址,中断下半部分执行函数);
2. 启动中断下半部分----把任务放入到内核线程中
//中断下半部分启动--放入内核线程
schedule_work(&workqueue);
```

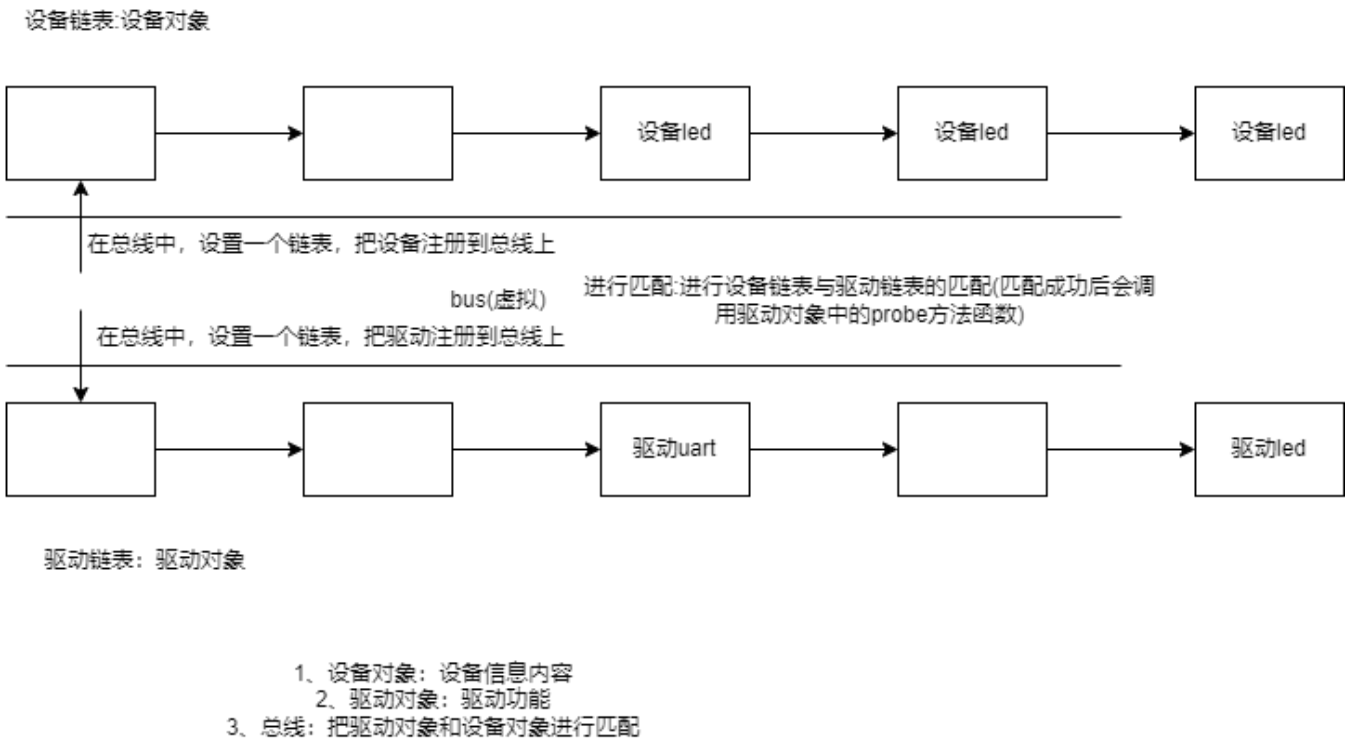
tasklet和workqueue区别：tasklet在中断上下文执行，workqueue在进程上下文；tasklet不可休眠，workqueue可以休眠

3. 总线模型

1.设备驱动模型

1. 实现加载入口函数xxx_init()和卸载入口函数xxx_exit()
2. 申请设备号 register_chrdev()(与内核相关)
3. 利用udev和mdev机制创建设备文件(节点)class_create、device_create
4. 硬件部分初始化
    1. io资源映射--ioremap
    2. 注册中断
5. 构建file_operation结构
6. 实现具体的硬件操作方法：xxx_read、xxx_open

2.总线模型



1、设备对象：设备信息内容
2、驱动对象：驱动功能
3、总线：把驱动对象和设备对象进行匹配

总线设备驱动模型：bus、driver、device

总线bus：

```
//总线对象,描述一个总线，管理driver和device，匹配
struct bus_type
{
        const char *name;//总线名字
        int (*match)(struct device *dev, struct device_driver *drv);//总线进行匹配的
```

```
函数
        //匹配时机：当总线上添加了新设备或新驱动，内核会调用一次或多次这个函数；匹配成功：通过匹配函数的返回值来表示是否成功，返回值为0表示不成功，1表示成功
};
//在内核中注册总线
int bus_register(struct bus_type *bus)
参数:
    struct bus_type *bus:结构体的地址，结构体类型就是总线对象(描述总线)
返回值：
    注册总线：成功返回0，失败返回错误码

//注销总线
void bus_unregister(struct bus_type * bus)
```

设备device：

```
//设备对象，描述一个设备，有设备信息内容，包括地址，中断号，甚至是其他自定义信息
struct device
{
    struct kobject kobj;//所有对象的父对象
    const char * init_name;//设备名字
    struct bus_type     *bus;//在设备对象中描述要注册到哪条总线（总线对象的地址）
    void * platform_data;//自定义数据的地址，指向任意类型(表示设备信息内容)
    void (*release)(struct device *dev);//在注销卸载设备时(device_unregister函数)，就会调用这个release函数
};

//注册设备到总线中
int device_register(struct device *dev)
参数：
    struct device *dev：结构体的地址，结构体类型是设备对象(描述设备，包含设备信息)
返回值：
    注册设备：成功返回0，失败返回错误码

//从总线注销设备
void device_unregister(struct device *dev)
```

驱动driver：

```
//驱动对象，描述设备驱动的方法（代码逻辑）
struct device_driver
{
    const char * name;//驱动名字
    struct bus_type * bus;//在驱动对象中描述要注册到哪条总线（总线对象的地址）
```

```
    int (*probe) (struct device *dev);//如果device和driver匹配成功，driver要做的事情(驱动的方法：申请设备号、设备节点等)
        int (*remove) (struct device *dev);//如果device或driver从总线移除，driver要做的事情(驱动的方法：注销设备号、设备节点等)
};
//注册驱动到总线中
int driver_register(struct device_driver *drv)
参数：
    struct device_driver *drv：结构体的地址，结构体类型是驱动对象(描述驱动)
返回值：
    注册驱动：成功返回0，失败返回错误码

//从总线注销驱动
void driver_unregister(struct device *dev)
```

3. 平台总线

三星设备：2410、2440、6410、s5pc100、4412
硬件平台升级后，部分的模块控制方式，基本上是类似的
但是模块的地址可能不同
gpio控制逻辑：

1. 配置gpio的控制寄存器输入输出功能gpxxcon
2. 给gpio数据寄存器设置(获取)高低电平gpxxdat

硬件版本升级后，逻辑控制操作基本上是一样的
但是地址不同

uart控制：

1. 设置 115200、8n1：UCON、ULCON、UDIV

硬件版本升级后，逻辑控制操作基本上是一样的
但是地址不同

问题： 当soc（片上系统）升级的时候，对于相似的设备驱动，需要编写多次
只要完成硬件产品的升级，驱动就需要重新编写
但是存在大量重复的代码

平台总线：
driver(操作逻辑)和device(中断/地址)分离，在升级时，只需要修改device中信息既可

平台总线三要素：

1. bus总线
   platform_bus：平台总线，不需要我们自己进行创建，在内核中进行了调用，开机的时候自动创建

```
//在内核代码，已经创建了平台总线对象，对平台总线进行了描述
struct bus_type platform_bus_type = {
        .name           = "platform",
        .dev_groups     = platform_dev_groups,
        .match          = platform_match,
        .uevent         = platform_uevent,
        .pm             = &platform_dev_pm_ops,
};
//内核调用 注册平台总线 操作
bus_register(&platform_bus_type);

匹配方式：
struct platform_device *pdev = to_platform_device(dev);
struct platform_driver *pdrv = to_platform_driver(drv);

if (pdrv->id_table)//:在驱动中存在支持的平台信息，使用驱动中的平台信息与设备进行匹配
                return platform_match_id(pdrv->id_table, pdev) != NULL;
```

2. device设备

```
//平台总线设备对象,描述一个平台的设备
struct platform_device {
    const char  *name;//设备名字，用于匹配
    int id;//一般直接写-1
    struct device dev;//继承了device父类，包含设备对象的所有信息
    u32 num_resources;//资源的个数
    struct resource     *resource;//设备信息资源的首地址
};

//注册设备到平台总线
int platform_device_register(struct platform_device * pdev)
//从平台总线注销设备
void platform_device_unregister(struct platform_device * pdev)
```

3. driver驱动

```
//平台总线驱动对象，描述一个平台的对应驱动操作

struct platform_driver {
        int (*probe)(struct platform_device *);//匹配成功调用
```

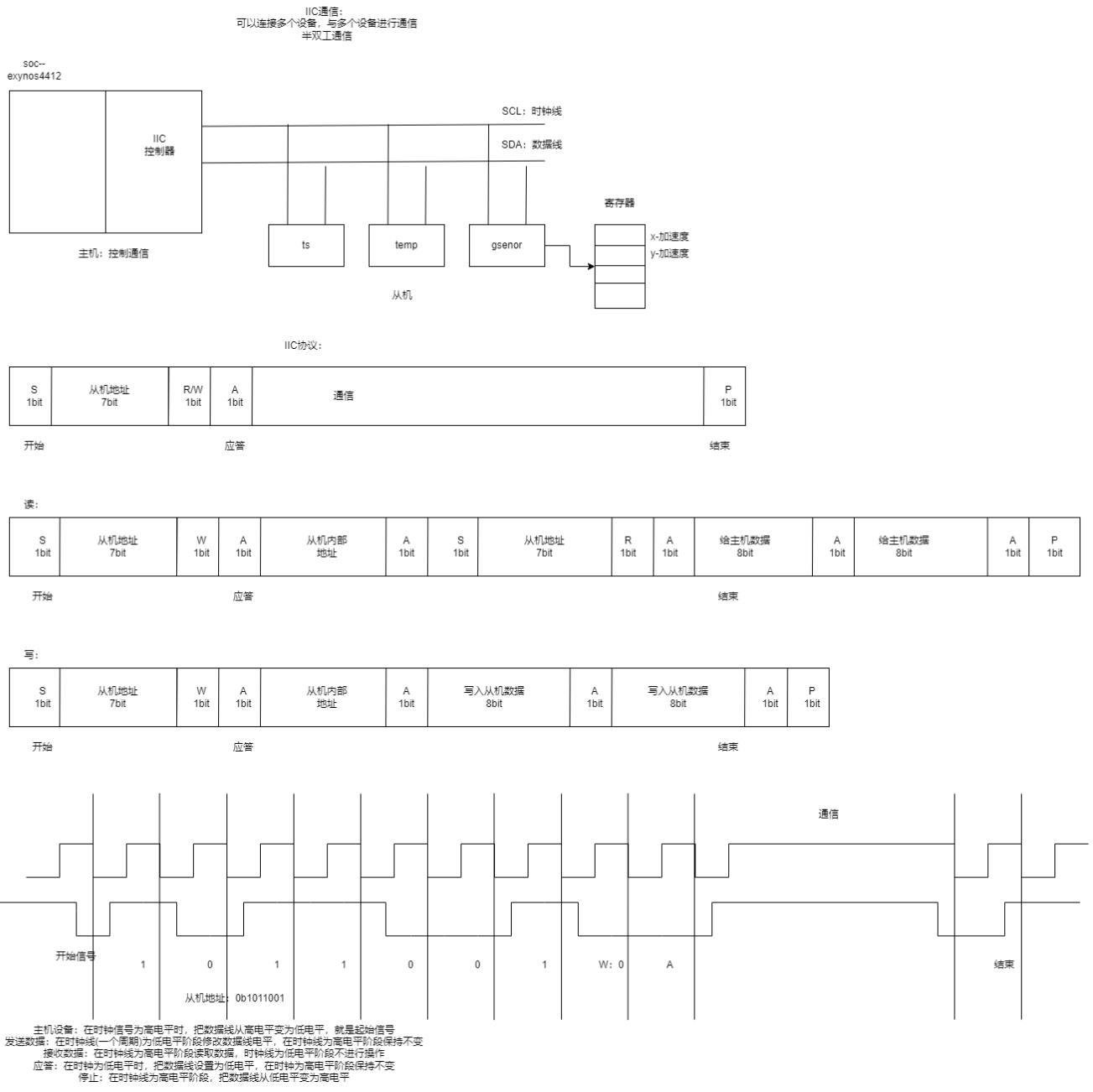
```
        int (*remove)(struct platform_device *);//移除时调用
        struct device_driver driver;//平台驱动继承的父类（通用的驱动对象信息）
        |
        const char              *name;//驱动名字
        const struct platform_device_id *id_table;//如果driver支持多个平台，在列表中列举支持平台，是一个数组，数组中每一个元素就是一个支持的平台信息，id_table就是存储首地址
};

//注册驱动到平台总线
int platform_driver_register(struct platform_driver * pdrv)
//从平台总线注销驱动
void platform_driver_unregister(struct platform_driver * drv)
```

实现：编写一个能够在多个平台上使用的led驱动

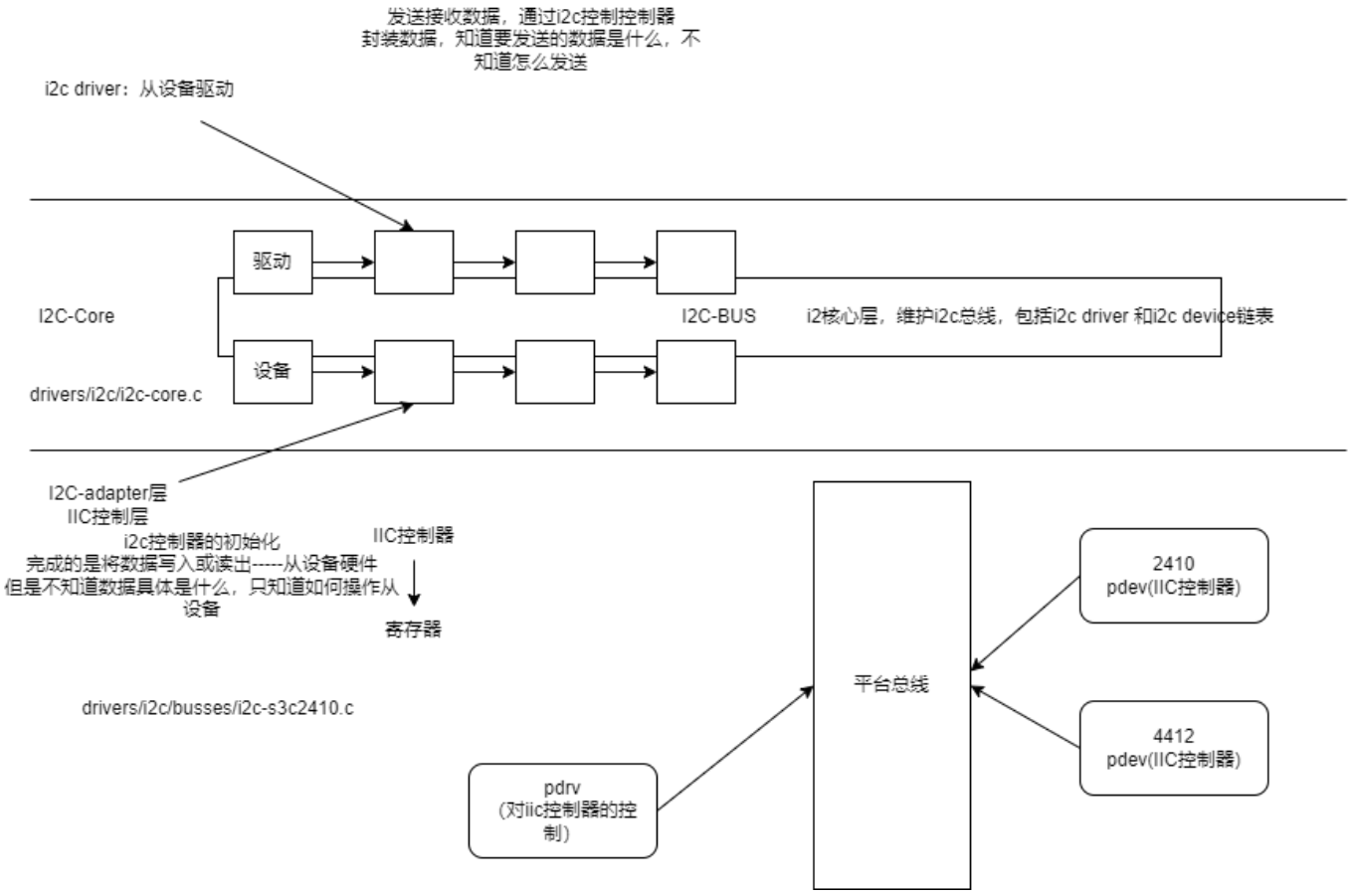
1. 注册platform_driver，实现操作设备方法 const struct platform_device_id *id_table：平台设备支持
   注册完毕，匹配成功调用probe函数：
   probe方法： 对硬件进行操作
   申请设备号
   设备节点
   初始化硬件
   实现文件IO接口
2. 注册platform_device，设备信息资源
   struct resource * resource;//设备信息资源的首地址

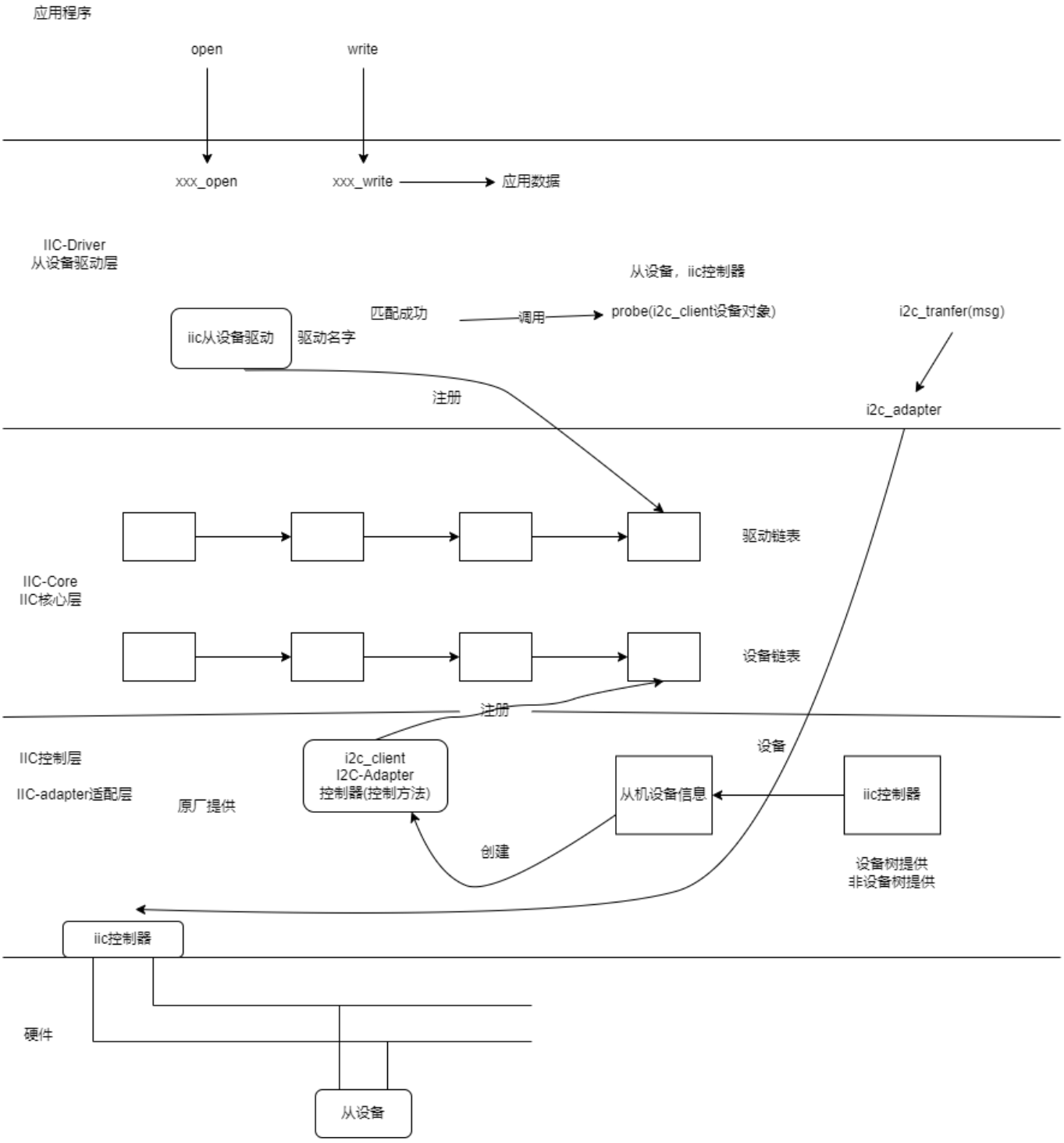
3. IIC子系统(IIC总线驱动)

是芯片与外部设备进行通信的一种方式，采用某种格式来进行收发数据



IIC子系统框架(Linux IIC驱动)


发送接收数据，通过i2c控制控制器
封装数据，知道要发送的数据是什么，不
知道怎么发送
i2c driver：从设备驱动
驱动
设备
I2C-Core
I2C-BUS
i2核心层，维护i2c总线，包括i2c driver 和i2c device链表
drivers/i2c/i2c-core.c
I2C-adapter层
IIC控制层
i2c控制器的初始化
完成的是将数据写入或读出-----从设备硬件
但是不知道数据具体是什么，只知道如何操作从
设备
IIC控制器
寄存器
drivers/i2c/busses/i2c-s3c2410.c
pdrv
(对iic控制器的控
制)
平台总线
2410
pdev(IIC控制器)
4412
pdev(IIC控制器)



1. IIC子系统设备信息

设备信息：设备树添加 从机设备：mpu6050 soc--------I2C5 IIC控制器： 0x1386_0000 0×1387_0000 0×1388_0000 0×1389_0000 0×138A_0000 0×138B_0000-----IIC5------- mpu6050 0×138C_0000 0×138D_0000 0×138E_0000

控制器
在设备树中：exynos4.dtsi
存在对iic控制器的描述

```
    i2c_0: i2c@13860000 {
            #address-cells = <1>;
            #size-cells = <0>;
            compatible = "samsung,s3c2440-i2c";
            reg = <0x13860000 0x100>;
            interrupts = <0 58 0>;
            clocks = <&clock 317>;
            clock-names = "i2c";
            pinctrl-names = "default";
            pinctrl-0 = <&i2c0_bus>;
            status = "disabled";
    };

    i2c_5: i2c@138B0000 {
            #address-cells = <1>;
            #size-cells = <0>;
            compatible = "samsung,s3c2440-i2c";
            reg = <0x138B0000 0x100>;
            interrupts = <0 63 0>;
            clocks = <&clock 322>;
            clock-names = "i2c";
            status = "disabled";
    };
```

从机设备
对iic从设备的信息描述进行添加
新增一个i2c5的从设备 exynos4412-fs4412.dts 添加包括i2c5控制器及从设备信息

```
i2c@138B0000 {
            #address-cells = <1>;
            #size-cells = <0>;
            samsung,i2c-sda-delay = <100>;
            samsung,i2c-max-bus-freq = <20000>;
            pinctrl-0 = <&i2c5_bus>;
            pinctrl-names = "default";
            status = "okay";

            mpu6050@68 {
                    compatible = "invensense,mpu6050";
                    reg = <0x68>;
```

```
            };
    };
```

2. IIC子系统从设备驱动

a、构造i2c_driver，注册到i2c总线

```
//i2c驱动对象
struct i2c_driver {
    int (*probe)(struct i2c_client *, const struct i2c_device_id *);
        int (*remove)(struct i2c_client *);
    struct device_driver driver;//继承的父类(i2c驱动对象中包含的父类信息)
        |
        const char * name;//驱动名字
        const struct of_device_id       *of_match_table;//设备树的设备匹配
    const struct i2c_device_id *id_table;//用于与设备对象进行匹配，非设备树
};

//注册iic驱动到iic总线
int i2c_add_driver(struct i2c_driver *driver);
//从i2c总线注销i2c驱动
void i2c_del_driver(struct i2c_driver * driver)
```

b、实现probe函数
申请设备号
创建设备节点
iic控制器与从设备通信
实现文件io接口