Table of Contents

1. 第一天

1.1. 1. 内核源码的编译

1.1.1. a. **获取源码**

```
www.kernel.org
```

1.1.2. b. **内核编译步骤**

略

1.2. 2. 编写驱动

1.2.1. a. **头文件**

```
#include <linux/init.h>
#include <linux/module.h>
```

1.2.2. b. 声明驱动模块的装载和卸载函数

```
module_init(hello_init);
module_exit(hello_exit);
```

1.2.3. c. 实现驱动模块的装载和卸载函数

1.2.4. d. 添加GPL认证

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("bot@qq.com"); //可选
MODULE_DESCRIPTION("ACPI EC sysfs access driver"); // 可选
```

1.2.5. e. 写Makefile

```
#内核源码路径
 KERNEL_DIR = /home/dwu/samba/linux-4.19.100

 #-C表示进入某个目录并执行该目录下的Makefile

 all:
     #表示进入内核目录， 并告诉内核要将当前源码编译成模块
     make -C $(KERNEL_DIR) M=$(CUR_DIR) module

 clean:
     make -C $(KERNEL_DIR) M=$(CUR_DIR) clean

 #指定编译哪个源文件
 obj-m = hello_drv.o
```

1.3. 3. 运行开发板

前提是内核已经移植好

1.4. 4. 模块特性---ko

> 模块传参
> 模块调用

1.4.1. a. **模块传参**

```
insmod hello_drv.ko number=250 name="ruhua"
```

代码中需要增加：

```
module_param(参数名, 参数类型, 权限(644));
module_param(number, int, 0644);
```

其中0644为权限值，即为下面文件的权限

```
# ls /sys/module/hello_drv_para/parameters/
name    number
-rw-r--r--    1 root     root        4.0K Jan  1 00:03 name
-rw-r--r--    1 root     root        4.0K Jan  1 00:03 number
```

1.4.2. b. **模块调用**

被调用的模块需要用EXPORT_SYMBOL导出

```
int add(int a, int b)
{
    return a+b;
}
EXPORT_SYMBOL(add);
MODULE_LICENSE("GPL");
```

2. 第二天

2.1. 5. 一个真正的设备驱动需要一些什么元素

o. 实例化全局的设备对象

```
/**
 * @brief        分配内存
 * @param[1]    分配大小
 * @param[2]    分配标志---GFP_KERNEL表示如果当前暂时没有内存，会尝试等待
 * @param[3]    文件操作对象
 * @return        如果系统分配，返回设备号，否则返回负数错误
 */
led_dev = kzalloc(sizeof (struct s5pv210_led), GFP_KERNEL);
```

a. 需要一个设备号

> 因为内核中有很多设备驱动，所以需要一个id来进行区分
> 设备号分为两个部分：
> 主设备号：某一类设备
> 此设备号：某类设备中的某个设备
> 例如：前置和后置摄像头都是camera这类设备，前置：0， 后置：1
> 在内核中：dev_t来表示设备号，为32bit的整数，高12bits：主设备号；低20bits：次设备号

```
/**
 * @brief        申请一个设备号
 * @param[1]    指定一个号码，填0表示由系统分配
 * @param[2]    字符串--描述设备驱动信息--自定义--/pro/devices文件中的名字
 * @param[3]    文件操作对象
 * @return        如果系统分配，返回设备号，否则返回负数错误
 */
int register_chrdev(unsigned int major, const char *name,
              const struct file_operations *fops);

/**
 * @brief        申请一个设备号
 * @param[1]    指定一个号码
 * @param[2]    字符串--描述设备驱动信息--自定义
 * @return        void
 */
void unregister_chrdev(unsigned int major, const char *name);
```

b. 需要一个设备文件

> linux中将所有的设备都看成是文件，操作设备其实就是操作文件；
> 设备文件称之为设备节点(/dev下面的文件)

```
ls -l /dev
crw-------    1 root     root      254,   0 Jan  1 00:00 gpiochip0
crw-------    1 root     root      254,   1 Jan  1 00:00 gpiochip1
crw-------    1 root     root      254,  10 Jan  1 00:00 gpiochip10
crw-------    1 root     root      254,  11 Jan  1 00:00 gpiochip11
```

> 如何创建设备节点：
> 1）手动创建：每次开机都要创建，/dev下面的节点斗在内存中

```
mknod 文件名       类型   主设备号   此设备号
mknod /dev/hello   c      250         0
```

2. 自动创建

/**

- @brief 创建一个类
- @param[1] 当前模块---THIS_MODULE
- @param[2] 字符串--表示类的名字
- @return struct class 指针类型 */ *struct class * class_create(owner, name);* 销毁：*class_destroy(led_dev->cls); /**
- @brief 创建一个设备节点
- @param[1] class_create返回的指针
- @param[2] 该设备的父类
- @param[3] 设备号---包含了主设备号和次设备号MKDEV
- @param[4] 私有数据---一般都填NULL
- @param[5] 设备节点的名字 --- /dev/led
- @return struct class 指针类型 */ struct device *device_create(struct class *class, struct device *parent, dev_t devt, void *drvdata, const char *fmt, ...); 销毁：device_destroy(led_dev->cls, MKDEV(led_dev->dev_major, 0));

c. 需要一个设备的操作方法

> 驱动：操作硬件，为应用服务
> 驱动提供功能，应用使用功能

```
int hello_drv_open(struct inode * inode, struct file * filep)
{
    printk(KERN_INFO"--------^_^ %s-------\n", __FUNCTION__);
    return 0;
}

static struct file_operations hello_fops = {
    .open = hello_drv_open,
};
```

> 系统调用，会从用户空间陷入内核空间，操作系统通过软中断产生异常，跳到中断处理函数，实现空间切换，使用不同的中断号区分不同的系统调用，arm软中断指令为swi，x86软中断指令为int

```
异常-->处理-->根据中断号调用-->sys_xxx(open,write...)-->区分调用哪个驱动的(open,write...)-->
```

d. 初始化硬件

```
/**
 * @brief         将物理地址转为虚拟地址
 * @param[1]    物理地址
 * @param[2]    映射长度
 * @return        映射后的虚拟地址
 */
gpj0_conf = ioremap(0xE0200240, 8);
iounmap(gpj0_conf);
```

e. 应用空间和内核空间之间的数据交互

```
/**
 * @brief        从用户空间获取数据，一般都用在驱动中的write函数的实现
 * @param[1]    目标地址---内核空间地址
 * @param[2]    源地址---用户空间的地址
 * @param[3]    拷贝数据个数
 * @return        没有拷贝成功的数据个数，所以成功返回0，出错返回大于0
 */
unsigned int copy_from_user(void * to, const void __user * from, unsigned long n);

/**
 * @brief        将内核空间数据拷贝到用户空间，一般都用在驱动中的read函数的实现
 * @param[1]    目标地址---用户空间地址
 * @param[2]    源地址---内核空间的地址
 * @param[3]    拷贝数据个数
 * @return        没有拷贝成功的数据个数，所以成功返回0，出错返回大于0
 */
unsigned int copy_to_user(void __user * to, const void * from, unsigned long n);
```

f. linux中ioctl的实现和gpio库函数的使用

> 如果需要给用户空间更多的api，可以添加一个ioctl接口：
> ioctl()用于给驱动发送指令：某个灯亮，某个灯灭，全亮，全灭
> 应用空间：

```
int ioctl(int fd, unsigned long cmd, ...);
```

> 驱动：

```
xxx_ioctl(int fd, unsigned long cmd, ...)
{
    switch(cmd)
    {
        case 命令1：
        ...
    }
}
```

> 命令如何定义：由程序员决定， 一定是一个整数

1)直接用一个整数---可能与系统中已经存在的命令冲突

```
eg:
    #define LED_ALL_ON    0x2222
    #define LED_ALL_OFF    0x3333
```

2)用内核提供接口来定义一个整数

```
用户只传命令：_IO(type, nr)    //参数1：魔幻数(0-255)，参数2：唯一的数字
用户写数据到内核：_IOW(type, nr, size) //参数3：数据类型，可以是结构体
用户从内核读数据：_IOR(type, nr, size)
用户读写内核数据s：_IOWR(type, nr, size)

eg:
    #define LED_NUM_ON    _IOW('L', 0x3456, int)
    #define LED_NUM_OFF    _IOW('L', 0x3457, int)
    #define LED_ALL_ON    _IO('L', 0x3458)
    #define LED_ALL_OFF    _IO('L', 0x3459)
```

3. gpio库函数的使用:

(1) 直接操作gpio口对应的寄存器（先看原理图---数据手册---物理地址---ioremap）
(2) gpio库函数的接口

```
//申请gpio口
gpio_request(unsigned gpio, const char * label);

//将某个gpio配置成输出功能
gpio_direction_intput(unsigned gpio, int value);
//将某个gpio配置成输出功能，并直接输出高低电平
gpio_direction_output(unsigned gpio, int value);
//获取gpio值
gpio_get_value
//设置gpio值
gpio_set_value
//通过gpio口获取到中断号码
gpio_to_irq
//释放
gpio_free(unsigned gpio)
```

3. 第三天

3.1. 作业1

PC使用QT编写界面控制开发板led灯

3.2. 作业2

两开发板互相控制led

3.3. 主要内容

1. Linux中file，cdev，inode之间的关系

> struct file对象：描述进程中打开一个文件的信息：文件名，标志(可读写)，文件偏移

```
open("/dev/led", O_RDWR, 0666);
struct file
{
    struct path        f_path;
    struct inode        *f_inode;    /* cached value */
    const struct file_operations    *f_op;//文件操作对象
    ...
    unsigned int         f_flags;
    fmode_t            f_mode;
    struct mutex        f_pos_lock;
    loff_t            f_pos;
    struct fown_struct    f_owner;
    ...
    void            *private_data;//万能指针
    ...
}
```

> struct cdev对象：描述一个字符设备对象(设备号+文件操作对象)，任何一个字符设备驱动，都有该对象, 一旦cdev被注册，就会将新建的cdev放在cdev_map全局变量中；

```
struct cdev {
    struct kobject kobj;//基类
    struct module *owner;
    const struct file_operations *ops;//文件操作对象
    struct list_head list;//链表
    dev_t dev;//设备号
    unsigned int count;
} __randomize_layout;
```

> struct inode对象：描述文件系统中的某个文件的信息(文件权限，类型，uid，gid，修改时间，大小等)；

```
struct inode {
    umode_t             i_mode;//fstat查看文件状态
    unsigned short         i_opflags;
    kuid_t             i_uid;
    kgid_t             i_gid;

    ...
    dev_t             i_rdev;//设备号
    ...
    struct timespec64    i_atime;
    struct timespec64    i_mtime;
    struct timespec64    i_ctime;
    ...
    const struct file_operations    *i_fop;
    ...
}
```

> 三者关系

```
异常-->处理-->根据中断号调用-->sys_xxx(open,write...)-->区分调用哪个驱动的(open,write...)-->
```

> 设备驱动
> register_chrdev(dev_no, fops)注册设备时会创建cdev，并将cdev中的设备号和fops初始化，放入vfs层的cdev链表

2. 新的注册字符设备的方式

   cdev_alloc(void);
   cdev_init(struct cdev * cdev, const struct file_operations * fops);
   cdev_add(struct cdev *, dev_t, unsigned);

3. 中断申请

> 中断控制器(vic)管理/筛选/设置中断优先级
> 最终中断由CPU处理（FIQ，IRQ）

中断操作函数，补----------

4. 文件io模型实现之阻塞和非阻塞

> 默认情况下大部分都是阻塞模式：
> 阻塞函数：read, accept, read/recv/recvform

> 实现阻塞：
> 0,需要一个等待队列头

```
struct wait_queue_head {
spinlock_t        lock;
struct list_head    head;
};

init_waitqueue_head(&key_dev->wq_head);
```

> 1,根据条件可以让进程进入休眠状态

```
/**
 * @brief         实现阻塞
 * @param[1]     表示等待队列头
 * @param[2]     表示一个条件---如果为假，就在此休眠，如果为真就不休眠
 * @return         映射后的虚拟地址
 */
wait_event_interruptible(wq_head, condition)
```

> 2,资源可达进行唤醒

```
wake_up_interruptible(x)
```

> 非阻塞

> 在应用中设置非阻塞模式

```
open("/dev/key0", O_RDWR | O_NONBLOCK);
```

> read() 有数据就得到数据，没有数据就得到一个出错码---EAGAIN。
> 驱动：需要区分阻塞还是非阻塞

```
xxx_read
{
    需要区分阻塞还是非阻塞
}
if ((flp->f_flags & O_NONBLOCK) && !key_dev->have_data) //如果是非阻塞且没有数据
{
    return -EAGAIN;
}
```

4. 第四天

4.0.1. 主要内容：

1. 多路复用的实现----等设备树学完回来再学这个，里面用到的中断，Linux4.19已经没有
2. mmap的实现
3. 中断下半部的实现方式
    1. a，tasklet的实现
    2. b, 工作队列的实现

> 应用调用：

```
open()
```

vfs： 搜索SYSCALL_DEFINE3 sys_open() ---fs/open.c { do_sys_open(AT_FDCWD, filename, flags, mode); //1，创建struct_file记录open中的各个参数信息，返回一个fd, 将fd和struct_file关联 查找cdev代码 fd = get_unused_fd_flags(flags); struce file * f = do_filp_open(dfd, tmp, &op, lookup); fd_install(fd, f);

```
    //2，查找cdev的代码---在do_filp_open
    chrdev_open查找cdev
    const struct file_operations def_chr_fops = {
        .open = chrdev_open,

            struct cdev *new = NULL;
            //根据设备号找到cdev中的obj
            kobj = kobj_lookup(cdev_map, inode->i_rdev, &idx);
            通过kobj找到cdev
            new = container_of(kobj, struct cdev, kobj);
            inode->i_cdev = p = new;
            //将cdev中fop给了file的f_op
            fops = fops_get(p->ops);
            //调用了cdev中fop的open方法
            filp->f_op->open(inode, filp);

        .llseek = noop_llseek,
    };
}
```

4.1. 1. 多路复用

对于阻塞的io，如果有多个就会阻塞，无法继续执行，这时就可以用多路复用，进行监听，当有数据的io发生时，就可以立即被处理

> select/poll/epoll监听
> select 等待数据到来，只要有数据select就返回，无法判断到底哪个io有数据，只能自己去获取数据
> select主要监控三个集合， readset, writeset, exceptionset，每个set里面有不同fd
> poll 监控每个fd的读、写、出错状态，监控到状态后，也需要查询到底哪个io状态改变

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
struct pollfd {
    int fd;    //被监控的fd
    short events; //希望监控的事件
    short revents; //用于查询，当前fd是否发生了读，写，出错
};

while (1)
{
    ret = poll(pfd, 2, 01);
}
```

> 驱动中实现epoll

```
__poll_t key_drv_poll (struct file * flp, struct poll_table_struct * pts)
{
    unsigned int mask = 0;

    //1、将当前等待队列头注册到vfs层
    /**
     * @brief        将当前等待队列头注册到vfs层
     * @param[1]    文件对象
     * @param[2]    等待队列头
     * @param[3]    与等待队列关联的表格
     * @return
     */
    poll_wait(flp, &key_dev->wq_head, pts);//这是一个注册动作

    //2、如果有数据返回一个pollin，没有数据返回一个0
    if (key_dev->have_data)
    {
        mask |= POLLIN;
    }

    return mask;
}
```

根据图分析驱动中poll的两次调用

4.2. 2. mmap的实现

1） 是文件io中的一种
2）进程空间和驱动数据交互的比较高效的方式
3）将内核空间的物理内存映射到用户空间， 可以直接操作地址

```
#include <sys/mman.h>

/**
 * @brief        内存映射
 * @param[1]    指定映射到用户空间的地址，一般都填NULL，表示系统自动分配
 * @param[2]    映射的长度
 * @param[3]    对内存的访问权限，PROT_EXEC，PROT_READ，PROT_WRITE
 * @param[4]    是都给其他进程映射，MAP_SHARED，MAP_PRIVATE
 * @param[5]    打开的文件
 * @param[6]    从内存的多少偏移量开始映射
 * @return        映射到用户空间的地址
 */
void *mmap(void *addr, size_t length, int prot, int flags,
          int fd, off_t offset);
int munmap(void *addr, size_t length);
```

> 驱动的.mmap实现只需要调用如下函数就能将驱动中的物理内存映射到用户空间：
> vma用于描述用户空间的映射需求，vma是vfs层传递过来

```
/**
 * @brief        内存映射
 * @param[1]    表示需求
 * @param[2]    映射到用户空间的起始位置，addr用kzalloc分配，为虚拟地址，因此需要转为物理地址
                使用virt_to_phys(const volatile void * x)将虚拟地址转为物理地址
 * @param[3]    被映射的物理地址的页地址，用addr/4K，即addr >> 12
 * @param[4]    映射大小，建议为页的倍数
 * @param[5]    映射的权限
 * @return        如果出错建议调用者返回-EAGAIN
 */

addr
int remap_pfn_range(struct vm_area_struct *vma, unsigned long addr,
        unsigned long pfn, unsigned long size, pgprot_t prot)
eg:
    int dt_test_mmap (struct file * flp, struct vm_area_struct * vma)
    {
        unsigned long addr = virt_to_phys(dt_test_dev->virt_mem);

        vma->vm_flags |= VM_IO;
        vma->vm_page_prot = pgprot_noncached(vma->vm_page_prot);

        if (io_remap_pfn_range(vma, vma->vm_start,
                        addr >> PAGE_SHIFT,
                        PAGE_SIZE, vma->vm_page_prot))
        {
            printk(KERN_INFO"io_remap_pfn_range error\n");
            return -EAGAIN;
        }

        return 0;
    }
```

4.3. 3. 中断下半部的实现方式

> 中断的上半部、下半部：
> 中断的特性要求中断处理时间不能太长，一般将耗时短的部分放在上半部处理，将耗时长的放在下半部处理
> Linux中断下半部实现方式：softirq软中断实现,一般在内核开发中使用，tasklet是softirq的一种，workqueue工作队列
> 中断的下半部实际是将处理任务放在一个链表（tasklet）或队列（workqueue），由一个内核线程进行调度执行
> 中断下半部的基本编程：
> 1）初始化tasklet

```
/**
 * @brief         初始化tasklet
 * @param[1]    tasklet对象
 * @param[2]    下半部任务处理函数
 * @param[3]    传递给下半部任务处理函数的参数
 * @return
 */
void tasklet_init(struct tasklet_struct * t, void(* func)(unsigned long), unsigned long d
struct tasklet_struct
{
    struct tasklet_struct *next;
    unsigned long state;
    atomic_t count;
    void (*func)(unsigned long);
    unsigned long data;
};
```

> 2）在中断的上半部将tasklet加入到内核线程 /** * @brief 启动下半部
> * @param[1] tasklet对象 * @return
> */ static inline void tasklet_schedule(struct tasklet_struct * t)

```
/**
 * @brief         销毁tasklet对象
 * @param[1]    tasklet对象
 * @return
 */
void tasklet_kill(struct tasklet_struct * t)
```

> 2）创建tasklet对象

```
 1. a，tasklet的实现
2. b，工作队列的实现
```

5. 第五天

5.1. 主要内容：平台总线

1. 平台总线的作用
2. 平台总线的编程
3. 平台总线中的自定义数据
4. 内核中的平台设备

> 如果要实现一个驱动代码兼容多个设备：
> 分离：将通用/相似的代码和差异化代码分离
> 合并：在实际运行的时候，需要将差异化代码拿出来，进行操作。
> 很多硬件都有相似的操作方法，只是操作地址不同

5.2. 1. 平台总线的作用

> 用于soc的升级/驱动相似的设备

```
soc:
    s3c2440         s3c6410         s5pv210
    arm9         arm11          A8
    gpio         gpio         gpio
    uart         uart         uart
    i2c             i2c              i2c
    ...

gpio:
    1. 配置gpio的功能
    2. 给数据寄存器赋值

uart:
    1. 设置波特率115200，8n1
    2. 设置no fifo， no AFC
    3. 发送数据---写发送数据寄存器
        接收数据---读取接收数据寄存器
不同soc中：
    操作逻辑/操作方法相似，操作的硬件地址不一样

如果要编写一个gpio驱动，能够在不同的soc中用，怎么办
    在代码编程的时候：
        操作方法的代码（通用代码）-----分离-----硬件的资源（差异化代码）
            为了更好的代码维护
    在运行的时候：
        操作方法的代码（通用代码）-----合并-----硬件的资源（差异化代码）
```

> 如果需要合并就需要总线：bus、device，driver
> 平台总线的编程
> 平台总线：struct bus_type

```
struct bus_type {
    const char        *name;
    const char        *dev_name;
    struct device        *dev_root;
    const struct attribute_group **bus_groups;
    const struct attribute_group **dev_groups;
    const struct attribute_group **drv_groups;

    int (*match)(struct device *dev, struct device_driver *drv);
    int (*uevent)(struct device *dev, struct kobj_uevent_env *env);
    int (*probe)(struct device *dev);
    int (*remove)(struct device *dev);
    void (*shutdown)(struct device *dev);

    int (*online)(struct device *dev);
    int (*offline)(struct device *dev);

    int (*suspend)(struct device *dev, pm_message_t state);
    int (*resume)(struct device *dev);

    int (*num_vf)(struct device *dev);

    int (*dma_configure)(struct device *dev);

    const struct dev_pm_ops *pm;

    const struct iommu_ops *iommu_ops;

    struct subsys_private *p;
    struct lock_class_key lock_key;

    bool need_parent_lock;
};
```

> pdev:

```
struct platform_device {    //描述一个设备的信息
    const char    *name;        //名字，用于匹配
    int        id;                //表示不同寄存器的编号
    bool        id_auto;
    struct device    dev;    //父类
    u32        num_resources;    //资源的个数
    struct resource    *resource;    //资源的详细信息，描述中断和内存资源

    const struct platform_device_id    *id_entry;
    char *driver_override; /* Driver name to force a match */

    /* MFD cell pointer */
    struct mfd_cell *mfd_cell;

    /* arch specific additions */
    struct pdev_archdata    archdata;
};

struct resource {//资源
    resource_size_t start;
    resource_size_t end;
    const char *name;
    unsigned long flags;
    unsigned long desc;
    struct resource *parent, *sibling, *child;
};

注册和注销：
    int platform_device_register(struct platform_device*);
    void platform_device_unregister(struct platform_device*);
```

> pdrv

```
struct platform_driver {
    int (*probe)(struct platform_device *);        //表示匹配之后运行的函数
    int (*remove)(struct platform_device *);
    void (*shutdown)(struct platform_device *);
    int (*suspend)(struct platform_device *, pm_message_t state);
    int (*resume)(struct platform_device *);    //父类
    struct device_driver driver;
    const struct platform_device_id *id_table;    //可以匹配列表
    bool prevent_deferred_probe;
};

注册和注销：
    int platform_driver_register(struct platform_driver*);
    void platform_driver_unregister(struct platform_driver*);
```

> probe中要做事情：
> 1.拿到pdev中的资源，对硬件进行初始化
> 2.为用户提供接口(老套路)

```
编写驱动的套路：

0、实例化全局的设备对象---kzalloc

1、申请设备号---register_chrdev

2、自动创建节点---class_create, device_create

3、初始化硬件---ioremap

4、实现file_operations
```

> 写完驱动出现编译错误error: array type has incomplete element type ‘struct platform_device_id’，原因是struct platform_device_id结构体在linux/mod_devicetable.h中定义，没有包含头文件所致
> 卸载plat_led_dev.ko时，出现Device 's5pv210_led' does not have a release() function, it is broken and must be fixed.错误。
> 原因是：platform_device->device -> release 没有初始化。
> 解决方案：使用platform_device_alloc() , platform_device_add() 替代 platform_device_register()去添加设备,如果使用platform_device_register()添加设备需要自己实现 platform_device->device -> release,3.8以上的内核使用设备树管理设备,除测试外尽量少用这种代码添加的方式.
> 注：Linux3.8之后的内核使用设备树自动生成设备，由内核统一管理，不再手动注册
> 参考： https://blog.csdn.net/x356982611/article/details/79399371

5.3. 3. 平台总线中的自定义数据

> pdev:

```
resource: 地址和中断号（资源）
定义其他类型的资源和数据

//设计一个自定义数据
struct led_platdata{
    char * name;
    int shift;//移位数据
};

struct led_platdata led_pdata = {
    .name = "gpj0_3/4/5",
    .shift = 3,
};

struct platform_device led_pdev = {
    .name = "s5pv210_led",
    .id = -1,
    .num_resources = ARRAY_SIZE(led_resource),
    .resource = led_resource,
    .dev = {
        .platform_data = &led_pdata,//<-------------
        .release = plat_led_dev_release,
    }
};
```

> 编程套路：

```
1. 获取平台自定义数据
2. 获取内存/中断资源
3. ioremap
4. 硬件初始化
```

5.4. 4. 内核中的平台设备

> 什么时候用平台总线

```
1. 只要有设备的地址和中中断都可以用平台总线
2. 如果写的驱动需要在多个平台中升级使用
3. 平台总线只是一个功能代码：将操作方法和操资源进行分离
```

6. 第六天

6.1. 主要内容：输入子系统编程---主要针对输入设备

1. 输入系统的作用和框架
2. 输入子系统的编程
3. 输入子系统和平台总线的结合
4. 输入子系统的执行流程

6.2. 1. 输入系统的作用和框架

> 针对输入设备：button，keyboard，mouse，ts，gsensor，joystick
> 按照产生数据的类型进行分类：

```
1. 按键数据----键值
    button，keyboard

2. 绝对坐标数据----有最大值，最小值
    ts，gsensor，陀螺仪

3. 相对坐标数据---下一个坐标是相对于之前的坐标
    mouse，
```

> 引入输入子系统对输入设备进行驱动管理

```
1. 为驱动定义一个标准的编程方式
2. 用户空间到数据格式是统一的
```

> 输入子系统框架

```
应用层

input handler层数据处理层：kernel/drivers/inpout/evdev.c
    1. 和用户进行交互，实现fops
    2. 不知道数据是如何得到，但是知道如何将数据交给用户

input core层：kernel/drivers/inpout/input.c
    1. 维护了两个链表
    2. 为上下两层提供接口

input device层：输入设备硬件层
    内核自带的input device代码：
        drivers/input/touchscreen等
    1. 初始化硬件，获取硬件中的数据
    2. 知道得到数据，但是不知道如何将数据交给用户

确保zImage已经包含了input core和input handler层代码
    make menuconfig
     Device Drivers  --->
        Input device support  --->
            -*- Generic input layer (needed for keyboard, mouse, ...) //input.c
                <*>   Event interface //evdev.c  input handler层的代码
```

> 输入子系统的编程： Documentation/input/input-programming.rst

a. 分配一个input device对象

```
struct input_dev *input_allocate_device(void)
input_free_device(button_dev);
```

b. 初始化input devide对象

```
struct input_dev {
    const char *name;
    const char *phys;
    const char *uniq;
    struct input_id id;

    unsigned long propbit[BITS_TO_LONGS(INPUT_PROP_CNT)];
    //位表----每个bit表示不同的意义
    //BITS_TO_LONGS就是计算用多少个long来表示多长的bit
    //BITS_TO_LONGS = num / 32
    //evbit表示能够长生哪些数据类型
    unsigned long evbit[BITS_TO_LONGS(EV_CNT)];
    //keybit表示能够产生哪些按键数据
    unsigned long keybit[BITS_TO_LONGS(KEY_CNT)];//keybit[BITS_TO_LONGS(768)] = keybi
    //relbit表示能够产生哪些相对坐标
    unsigned long relbit[BITS_TO_LONGS(REL_CNT)];
    //absbit表示能够产生哪些绝对坐标
    unsigned long absbit[BITS_TO_LONGS(ABS_CNT)];
    unsigned long mscbit[BITS_TO_LONGS(MSC_CNT)];
    unsigned long ledbit[BITS_TO_LONGS(LED_CNT)];
    unsigned long sndbit[BITS_TO_LONGS(SND_CNT)];
    unsigned long ffbit[BITS_TO_LONGS(FF_CNT)];
    unsigned long swbit[BITS_TO_LONGS(SW_CNT)];

    struct device dev;//父类
    struct list_head h_list;
    struct list_head node;//链表节点
    ...
};

struct input_event { //用户读取到的输入设备的数据包
#if (__BITS_PER_LONG != 32 || !defined(__USE_TIME_BITS64)) && !defined(__KERNEL__)
    struct timeval time; //时间戳
#define input_event_sec time.tv_sec
#define input_event_usec time.tv_usec
#else
    __kernel_ulong_t __sec;
#if defined(__sparc__) && defined(__arch64__)
    unsigned int __usec;
    unsigned int __pad;
#else
    __kernel_ulong_t __usec;
#endif
#define input_event_sec  __sec
#define input_event_usec __usec
```

```
    #endif
        __u16 type;    //读取到数据类型：EV_KEY, EV_ABS, EB_REL
        __u16 code;    //码值
        __s32 value;//状态
    };

c. 注册input device对象
d. 硬件初始化
```

> 看源码：
>
> 1. 看整个分层
> 2. 看应用和驱动如何交互

```
open，read()
```

> 1. 分层

```
input handler
    evdev.c
    module_init(evdev_init);
    module_exit(evdev_exit);
    static struct input_handler evdev_handler = {
        .event        = evdev_event,
        .events        = evdev_events,
        .connect    = evdev_connect,
        .disconnect    = evdev_disconnect,
        .legacy_minors    = true,
        .minor        = EVDEV_MINOR_BASE,
        .name        = "evdev",
        .id_table    = evdev_ids,//用于比较的
    };

    input_register_handler(&evdev_handler);

        INIT_LIST_HEAD(&handler->h_list);
        注册到input core层
        //将当前的evdev_handler放入到input_table
        list_add_tail(&handler->node, &input_handler_list);

        list_for_each_entry(dev, &input_dev_list, node)
                input_attach_handler(dev, handler);
                    id = input_match_device(handler, dev);
                    error = handler->connect(handler, dev, id);//一旦匹配成功，调用connect


input core
    input.c为一个单独驱动
    err = class_register(&input_class);
    err = input_proc_init();
    err = register_chrdev_region(MKDEV(INPUT_MAJOR, 0),
                 INPUT_MAX_CHAR_DEVICES, "input");
    subsys_initcall(input_init);
    module_exit(input_exit);


input device
    input_register_device();

        INIT_LIST_HEAD(&handler->h_list);
        list_add_tail(&handler->node, &input_handler_list);
        list_for_each_entry(dev, &input_dev_list, node)
        input_attach_handler(dev, handler);//匹配
            id = input_match_device(handler, dev);
            error = handler->connect(handler, dev, id);//一旦匹配成功，调用connect, input_

    调用的conncet方法：
        .connect    = evdev_connect //evdev.c
```

```
int evdev_connect(struct input_handler *handler, struct input_dev *dev,
 const struct input_device_id *id)
//获取一个新的此设备号，在64~64+32之间
minor = input_get_new_minor(EVDEV_MINOR_BASE, EVDEV_MINORS, true);
if (minor < 0) {
    error = minor;
    pr_err("failed to reserve new minor: %d\n", error);
    return error;
}

//分配一个evdev对象
evdev = kzalloc(sizeof(struct evdev), GFP_KERNEL);
if (!evdev) {
    error = -ENOMEM;
    goto err_free_minor;
}
//初始化evdev对象
INIT_LIST_HEAD(&evdev->client_list);
spin_lock_init(&evdev->client_lock);
mutex_init(&evdev->mutex);
//初始化队列头，用于阻塞
init_waitqueue_head(&evdev->wait);
evdev->exist = true;

dev_no = minor;
/* Normalize device number if it falls into legacy range */
if (dev_no < EVDEV_MINOR_BASE + EVDEV_MINORS)
    dev_no -= EVDEV_MINOR_BASE;

//利用handle记录了inputdev和input handler
evdev->handle.dev = input_get_device(dev);
evdev->handle.name = dev_name(&evdev->dev);
evdev->handle.handler = handler;
//相互留个联系方式
evdev->handle.private = evdev;

//创建设备节点；以下代码等同于 device_create
//设置设备名，event0，event1...
dev_set_name(&evdev->dev, "event%d", dev_no);
evdev->dev.devt = MKDEV(INPUT_MAJOR, minor);
evdev->dev.class = &input_class;
evdev->dev.parent = &dev->dev;
evdev->dev.release = evdev_free;
device_initialize(&evdev->dev);
//inputdev, input handler，input handle建立双向关系
input_register_handle(&evdev->handle);

    struct input_handler *handler = handle->handler;
    struct input_dev *dev = handle->dev;
    list_add_tail_rcu(&handle->d_node, &dev->h_list);
    list_add_tail_rcu(&handle->h_node, &handler->h_list);
```

```
    //将当前的evdev放入到evdev_table[minor]
    cdev_init(&evdev->cdev, &evdev_fops);
    error = cdev_device_add(&evdev->cdev, &evdev->dev);
```

> 2. 看应用和驱动如何交互

```
open("/dev/input/event0", O_RDWR);
read(fd, &event, sizeof(struct input_event));

VFS:
    sys_open
        根据设备号找到cdev
驱动层的open
    input.c
    static const struct file_operations input_devices_fileops = {
        .owner        = THIS_MODULE,
        .open        = input_proc_devices_open,
        .poll        = input_proc_devices_poll,
        .read        = seq_read,
        .llseek        = seq_lseek,
        .release    = seq_release,
    };

struct input_event event;
read(fd, &event, sizeof(struct input_event));

sys_read

    filp->f_fop->read()
evdev.c
    .read = evdev_read,

        //阻塞
        while (read + input_event_size() <= count &&
               evdev_fetch_next_event(client, &event)) {
            //拷贝给用户
            if (input_event_to_user(buffer + read, &event))
                return -EFAULT;

            read += input_event_size();
        }

        if (read)
            break;

        //如果阻塞被唤醒，会从evdev_client中获取event
        if (!(file->f_flags & O_NONBLOCK)) {
            error = wait_event_interruptible(evdev->wait,
                    client->packet_head != client->tail ||
                    !evdev->exist || client->revoked);
            if (error)
                return error;
        }

        //谁唤醒
        input_event(button_dev, EV_KEY, pdesc->code, 1);
```

```
        input_sync(button_dev);
        input_handle_event(dev, type, code, value);//跳转到evdev.c
```

7. 第七天

7.1. 主要内容

1. i2c协议讲解
2. i2c子系统框架
3. i2c子系统驱动编程--从设备at24c02
4. i2c子系统框架代码流程

> i2c驱动 i2c设备比较多：ts，camera，e2prom，gsensor，hdmi

7.2. 1. i2c协议讲解

> 1） 传输协议：master首先通过总线发送一个起始位S，然后通过总线发送一个从设备地址（7bit/10bit，一般为7bit） + 1bit R/W位，总线上的从设备比对自己的地址，如果与主机发送的地址相等，则给主机回1bit的ACK，主机收到ACK后，开始进行数据传输（8bit数据+1bit ACK）

> 2） 时序：SCK时钟线有一个固定的时钟频率，在空闲时SDA线为高电平，当要开始发数据时，主机在时钟为高电平期间将SDA拉低（S位产生），第二个周期开始的7周期（或10个周期）位从机地址，第8个周期传输读写位（R/W位如果为高则为读，为低则为写），第9个周期master强行将SDA拉高，如果从设备响应则会在SCK为高电平时把SDA强行拉低，master则认为收到响应ACK，接下来的周期为8bit数据+1bit ACK，进行数据传输。当在SCK位高电平的半周期，SDA由低变高则表示停止。

> 3. 实际中在对i2c设备进行读写时，如果进行写，则主机首先发送Sbit+7bit地址+Wbit+ACK,然后从设备(如eeprom)的内部地址若干bits+ACK+ 若干数据8bits+ACK + Pbit , 如果读，则需要，首先发送Sbit+7bit地址+Wbit+ACK, 再发送从设备(如eeprom)的内部地址若干bits+ACK+P， 然后重新发送起始位进行转换模式Sbit+7bit地址+Rbit+ACK + 若干数据8bits+ACK + Pbit。

> 注意：当在进行数据传输时，SDA上的数据只有在SCK为高电平的半周期才有效，因此数据在传输过程中，只有SCK为低电平的半周期中，SDA上的数据才可以变化，而起始和停止位（S/P）只有在SCK为高电平时SDA变化才认为是起始和停止位，（在SCK位高电平的半周期，如果SDA由高变低则产生S位，如果由低变高则产生P位）

7.3. 2. i2c子系统框架

> 应用层

> 驱动层

```
i2c driver层 （从设备驱动层，与用户交互）probe，remove，注册到driver链表
    1 负责与用户交互
    2 知道要传输给从设备的数据是什么，但不知道如何通过硬件操作给从设备

i2c core层 以名字匹配dev和drv，并执行probe
    维护了i2c总线
    drivers/i2c/i2c-core-base.c i2c-core-of.c
    drivers/i2c/busses/i2c-s3c2410.c

i2c-adapter适配器层（i2c控制器层）：
    与硬件交互，负责硬件的初始化，即初始化i2c控制器
    知道如何将数据给从设备，但是不知道数据是什么
    创建i2c_client(slaver name and address),注册到device链表

如何确保低两层内核自带，make menuconfig
    > Device Drivers
        > I2C support   //i2c-corc.c
            > I2C Hardware Bus support
                > S3C2410 I2C Driver //i2c-s3c2410.c

如何查看？ // /sys/bus/i2c
```

7.4. 3. i2c子系统驱动编程--从设备at24c02

> 间接为i2c_client提供信息，有了信息后，系统会自动创建i2c_client
> 构建i2c_driver，注册到总线中

1. 构建i2c_driver，注册到总线

```
i2c_add_driver(struct i2c_driver *);
i2c_del_drive(struct i2c_driver *);

struct i2c_driver {
    unsigned int class;

    /* Standard driver model interfaces */
    int (*probe)(struct i2c_client *, const struct i2c_device_id *);
    int (*remove)(struct i2c_client *);
    ...
    struct device_driver driver;
    const struct i2c_device_id *id_table;
    ...

};
```

2. i2c系统中为从设备传输数据的方法

```
i2c_master_send(const struct i2c_client * client, const char * buf, int count)
i2c_master_recv(const struct i2c_client * client, char * buf, int count)

都调用了i2c_transfer(struct i2c_adapter * adap, struct i2c_msg * msgs, int num)

类似struct platform
struct i2c_client {//描述一个从设备信息的对象，里面所有的成员都是自动初始化
    unsigned short flags;        /* div., see below        */
    unsigned short addr;//从设备地址        /* chip address - NOTE: 7bit    */
                    /* addresses are stored in the    */
                    /* _LOWER_ 7 bits        */
    char name[I2C_NAME_SIZE];//名字用于和driver匹配
    struct i2c_adapter *adapter;//指向创建自己的adapter    /* the adapter we sit on
    struct device dev;        /* the device structure        */
    int init_irq;            /* irq set at initialization    */
    int irq;            /* irq issued by device        */
    struct list_head detected;
#if IS_ENABLED(CONFIG_I2C_SLAVE)
    i2c_slave_cb_t slave_cb;    /* callback for slave mode    */
#endif
};

struct i2c_adapter {
    struct module *owner;
    unsigned int class;          /* classes to allow probing for */
    const struct i2c_algorithm *algo; /* the algorithm to access the bus */
    void *algo_data;

    /* data fields that are valid for all devices    */
    const struct i2c_lock_operations *lock_ops;
    struct rt_mutex bus_lock;
    struct rt_mutex mux_lock;
```

```
    int timeout;            /* in jiffies */
    int retries;
    struct device dev;    //父类    /* the adapter device */

    int nr;
    char name[48];
    struct completion dev_released;

    struct mutex userspace_clients_lock;
    struct list_head userspace_clients;

    struct i2c_bus_recovery_info *bus_recovery_info;
    const struct i2c_adapter_quirks *quirks;

    struct irq_domain *host_notify_domain;
};

struct i2c_algorithm {
    /* If an adapter algorithm can't do I2C-level access, set master_xfer
       to NULL. If an adapter algorithm can do SMBus access, set
       smbus_xfer. If set to NULL, the SMBus protocol is simulated
       using common I2C messages */
    /* master_xfer should return the number of messages successfully
       processed, or a negative value on error */
    int (*master_xfer)(struct i2c_adapter *adap, struct i2c_msg *msgs,
               int num);//操作硬件的方法
    int (*smbus_xfer) (struct i2c_adapter *adap, u16 addr,
               unsigned short flags, char read_write,
               u8 command, int size, union i2c_smbus_data *data);

    /* To determine what the adapter supports */
    u32 (*functionality) (struct i2c_adapter *);

#if IS_ENABLED(CONFIG_I2C_SLAVE)
    int (*reg_slave)(struct i2c_client *client);
    int (*unreg_slave)(struct i2c_client *client);
#endif
};


struct i2c_msg {//传送给从设备的数据包
    __u16 addr;    //从设备的地址 /* slave address            */
    __u16 flags;//表示读还是写
    __u16 len;    //表示数据的字节数    /* msg length                */
    __u8 *buf;    //数据的缓冲指针    /* pointer to msg data            */
};
```

8. 第八天 如何跟读内核代码

1. 任务：总线匹配之后，调用drv中的probe(), 跟读probe所有代码
2. 如何跟读内核代码技巧：
    1. 看主线--主要的技术点
    2. 出错判断和变量不看
    3. 看不懂的不看
    4. 大胆去猜函数的作用
    5. 百度搜索函数
    6. 做好笔记和注释，以及总结
    7. 总结当前代码段/函数做了什么
3. i2c子系统的框架代码

    at24_drv.c:----4

    i2c-core-base.c:----2

```
如果要构建一个新的总线只需要注册一个， 就会在/sys/bus/下产生新的总线
bus_register(struct bus_type* type)
//往一个总线中增加一个设备dev，dev中的bus成员可以决定自己属于哪个总线
device_register(struct devide * dev)
//往一个总线中增加一个驱动drv，drv中的bus成员可以决定自己属于哪个总线
driver_register(struct driver * drv)

i2c_new_device是谁调用的？

struct i2c_client *
i2c_new_device(struct i2c_adapter *adap, struct i2c_board_info const *info)
    client = kzalloc(sizeof *client, GFP_KERNEL);
    client->adapter = adap;
    client->dev.platform_data = info->platform_data;
    client->flags = info->flags;
    client->addr = info->addr;
    client->init_irq = info->irq;
    status = i2c_check_addr_validity(client->addr, client->flags);
    status = i2c_check_addr_busy(adap, i2c_encode_flags_to_addr(client));

    client->dev.parent = &client->adapter->dev;
    client->dev.bus = &i2c_bus_type;
    client->dev.type = &i2c_client_type;
    client->dev.of_node = of_node_get(info->of_node);
    client->dev.fwnode = info->fwnode;

    status = device_register(&client->dev);
总结：
    i2c_new_device函数中：
        1. 构建了i2c_client
        2. 初始化i2c_client，通过i2c_board_info来初始化
        3. 注册到i2c总线

i2c_add_adapter//将adapter注册i2c总线，是谁调用该函数？应该从init里面看，由i2c-s3c2410
    i2c_register_adapter
        device_register(&adap->dev);
        i2c_scan_static_board_info
        bus_for_each_drv(&i2c_bus_type, NULL, adap, __process_new_adapter);
总结：
    adapter再注册的时候，会遍历__i2c_board_list链表
    如果adapter的编号和链表节点中的号码一致，就会构建一个i2c_client并注册client
    i2c_client中的成员的值来自board_info
```

i2c-s3c2410.c----3//平台总线的套路，既是driver也是一个adapter

```
platform_driver_register(&s3c24xx_i2c_driver);
    s3c24xx_i2c_probe
        //获取平台自定义数据
        pdata = dev_get_platdata(&pdev->dev);
        //分配一个全局的设备对象---分配i2c_adapter
        i2c = devm_kzalloc(&pdev->dev, sizeof(struct s3c24xx_i2c), GFP_KERNEL);
        //初始化adapter
        strlcpy(i2c->adap.name, "s3c2410-i2c", sizeof(i2c->adap.name));
        i2c->adap.owner = THIS_MODULE;
        i2c->adap.algo = &s3c24xx_i2c_algorithm;
        i2c->adap.retries = 2;
        i2c->adap.class = I2C_CLASS_DEPRECATED;
        i2c->tx_setup = 50;
```

mach-smdkv210.c/设备树----1

8.1. 总结：

> 所有的总线注册到内核链表的都是struct device和struct driver对象，这样便于链表的管理

8.2. LCD屏 FrameBuffer

8.3. 主要内容--framebuffer子系统--lcd屏驱动

1. FrameBuffer子系统的框架
2. LCD屏的驱动移植
3. 启动LOGO的制作
4. 应用程序控制LDC屏的方法
5. FrameBuffer子系统代码执行流程

> 补充知识点：设定自定义平台数据的方式
>
> 1. 在初始化的时候给定一个
> 2. 通过接口随时修改平台自定义数据, s3c_fb_set_platdata
> 3. 有设备树时候，自动从设备树获取

8.4. 1. FrameBuffer子系统的框架

应用层：

fb通用层：知道映射，但是不知道如何分配显存

```
1. 负责和应用层交互
2. 实现显存的映射
```

xxx-LCD控制器层：知道分配显存，但是不知道如何映射

```
1. 初始化LCD控制器
2. 实现显存的分配
```

8.5. 2. LCD屏的驱动移植

> Documentation/fb/framebuffer.txt

```
    soc             lcd屏                       驱动
```

垂直方向

```
    VSPW(无效周期)     tvpw(1<y<20)      y=10      vsync_len
    VBPD(稳定周期)     tvb-tvpw   (23-tvpw)13      upper_margin
    LINVAL(行数)       tvd                480      xres
    VFPD(返回时间)     tvfp               22       lower_margin
```

水平方向

```
    HSPW(无效周期)     thpw(1<x<40)      x=20      hsync_len
    HBPD(稳定周期)     thb-thpw   (46-thpw)26      left_margin
    HOZVAL(行数)       thd                800      xres
    HFPD(返回时间)     thfp               210      right_margin
```

移植的时候主要是配置设备树
如何确保fb通用层和lcd控制器层的驱动已经存在

```
> Device Drivers >
    Graphics support >
        Frame buffer Devices >
            Support for frame buffer devices> //fbmem.c
                Samsung S3C framebuffer support  //s3c-fb.c

Device Drivers > Graphics support > Console display driver support //将图片加载到显存中去显

> Device Drivers > Graphics support > Bootup logo
    [*]   Standard black and white Linux logo
    [*]   Standard 16-color Linux logo
    [*]   Standard 224-color Linux logo //表示的是一张图片linux-4.19.100\drivers\video\log
```

4. 应用程序控制LCD屏的方法

    1. 打开设备

```c
int fd = open("dev/fb0", O_RDWR);
```

    2. 获取到LCD屏的信息xres, yres, bpp

```c
struct fb_var_screeninfo var;
ioctl(fd, FBIOGET_VSCREENINFO, &var);
    .unlocked_ioctl = fb_ioctl,
        struct fb_info *info = file_fb_info(file);
                struct fb_info *info = registered_fb[fbidx];//从registered_fb数组中
得到fb_info
        do_fb_ioctl(info, cmd, arg);
        switch (cmd){
            case FBIOGET_VSCREENINFO:
                var = info->var;
                ret = copy_to_user(argp, &var, sizeof(var)) ? -EFAULT : 0;
```

    3. 映射显存到用户空间

```c
void *mmap(NULL, size_t length, PROT_READ|PROT_WRITE, MAP_SHARED,
                  int fd, off_t offset);
//length 一般为一帧的长度
size_t length = var.xres*var.yres * var.bits_per_pixel / 8;
```

    4. 得到图片数据

5. 高级子系统驱动套路

分配一个对象，初始化这个对象，注册这个对象(input子系统，i2c子系统，网卡驱动，块设备驱动)

9. 第九天 触摸屏

9.1. 主要内容----电容触摸屏驱动

1. 触摸屏的基本工作原理
2. 电容触摸屏驱动框架
3. 电容触摸屏读取坐标的原理和硬件初始化
4. Linux下多点触摸的协议
5. 电容触摸屏的驱动编写--gt811

9.2. 驱动编程：

1. 提供i2c client信息

arch\arm\mach-s5pv210\mach-smdkv210.c

```
static struct i2c_board_info smdkv210_i2c_devs2[] __initdata = {
    /* To Be Updated */
    { I2C_BOARD_INFO("gt811_ts", 0x5d), },
};
```

make zImage -j4

更新内核

```
[root@$x210: /]# cd /sys/bus/i2c/
[root@$x210: i2c]# ls
devices            drivers_autoprobe  uevent
drivers            drivers_probe
[root@$x210: i2c]# cd devices/
[root@$x210: devices]# ls
0-001b  0-0050  2-005d  i2c-0   i2c-1   i2c-2
[root@$x210: devices]# cd 2-005d/
[root@$x210: 2-005d]# cat
modalias    name        power/      subsystem/  uevent
[root@$x210: 2-005d]# cat name
gt811_ts
```

2. 编写从设备驱动

EINT14---GPH1_6

RESET---GPD0_3

硬件初始化：

a. 设置INT引脚为输入态，RESET设置成高电平（内部上拉）

b. RESET输出为低，延时1ms，转成输入态

c. 延时至少20ms，通过i2c寻址判断是否有响应

d. 如果有响应就分一次或多次初始化配置106个寄存器

9.3. Linux下多点触摸的协议

两点：

```
ABS_MT_POSITION_X x[0]
ABS_MT_POSITION_Y y[0]
SYN_MT_REPORT//表示第一个点上报完毕
ABS_MT_POSITION_X x[1]
ABS_MT_POSITION_Y y[1]
SYN_MT_REPORT//表示第二个点上报完毕
SYN_REPORT//所有点都上报完毕

如果用代码去实现：
input_event(dev, EV_ABS, ABS_MT_POSITION_X , 333);
input_event(dev, EV_ABS, ABS_MT_POSITION_Y , 133);
input_mt_sync(dev)
    input_event(dev, EV_SYN, SYN_MT_REPORT, 0);

input_event(dev, EV_ABS, ABS_MT_POSITION_X , 433);
input_event(dev, EV_ABS, ABS_MT_POSITION_Y , 533);
input_mt_sync(struct input_dev * dev)

input_sync(struct input_dev * dev)
```

10. 第十天

10.1. 主要内容---内核工作原理解析

1. 内核的编译步骤

2. Kconfig和Makefile的使用

3. 内核的裁剪---make menuconfig

4. 内核的工作原理

   a. 内核的内部构造

   b. 内核的启动步骤

   c. 内核中module_init是什么

   d. 挂载是什么意思

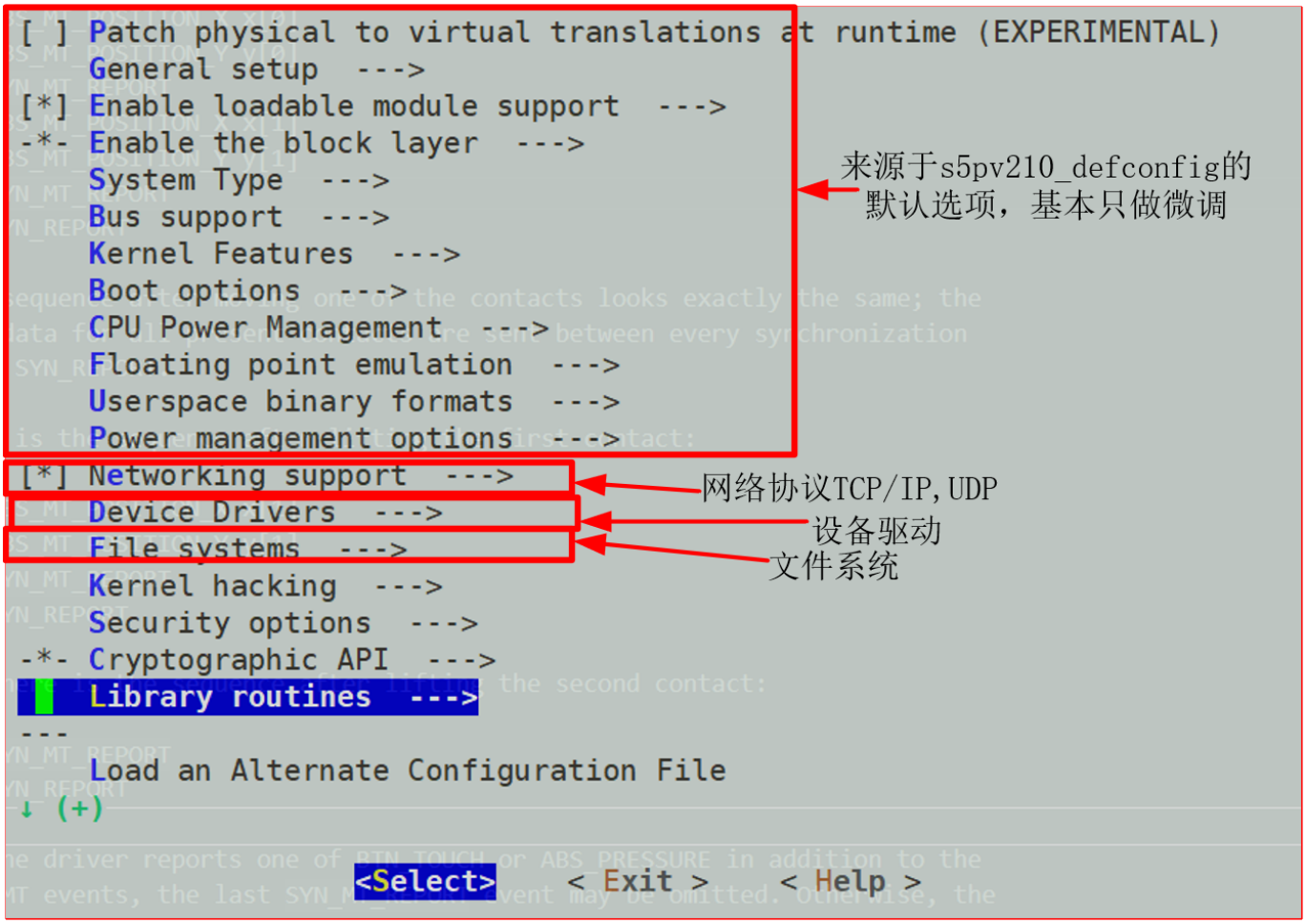
   e. 祖先进程init是如何启动的

10.2. 1. 内核的编译步骤

> step1:设置交叉编译工具链
>
> ```
> ARCH=arm
> CROSS_COMPILE=arm-none-linux-gnueabi-
> ```
>
> step2:选择当前的soc---s5pv210
>
> ```
> make s5pv210_defconfig
> ```
>
> step3:make menuconfig---内核剪裁
>
> ```
> System Type  --->
>     (2) S3C UART to use for low-level messages
> ```
>
> step4:make zImage或make uImage
>
> ```
> make uImage
> ```

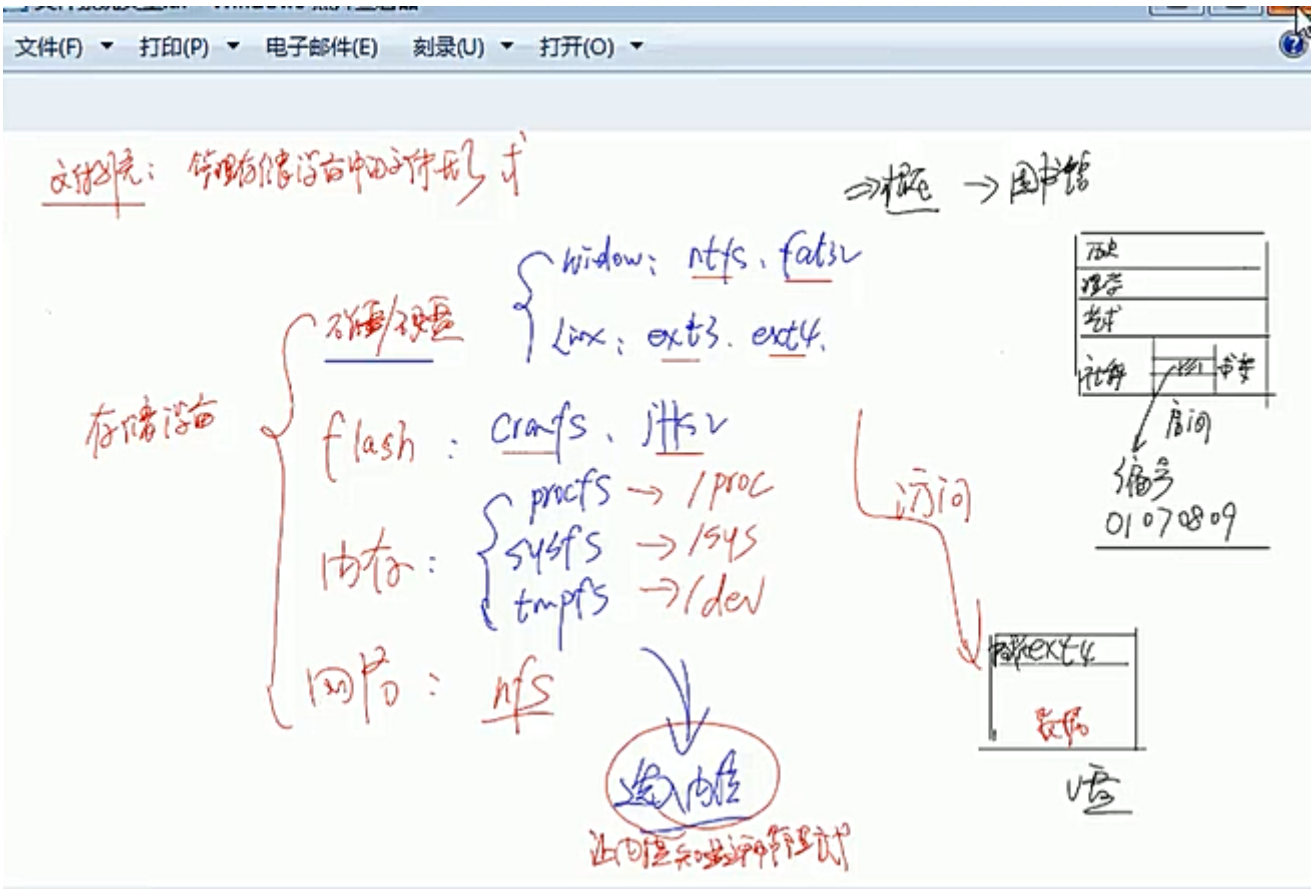
10.3. 2. Kconfig和Makefile的使用

10.4. 3. 内核的裁剪---make menuconfig

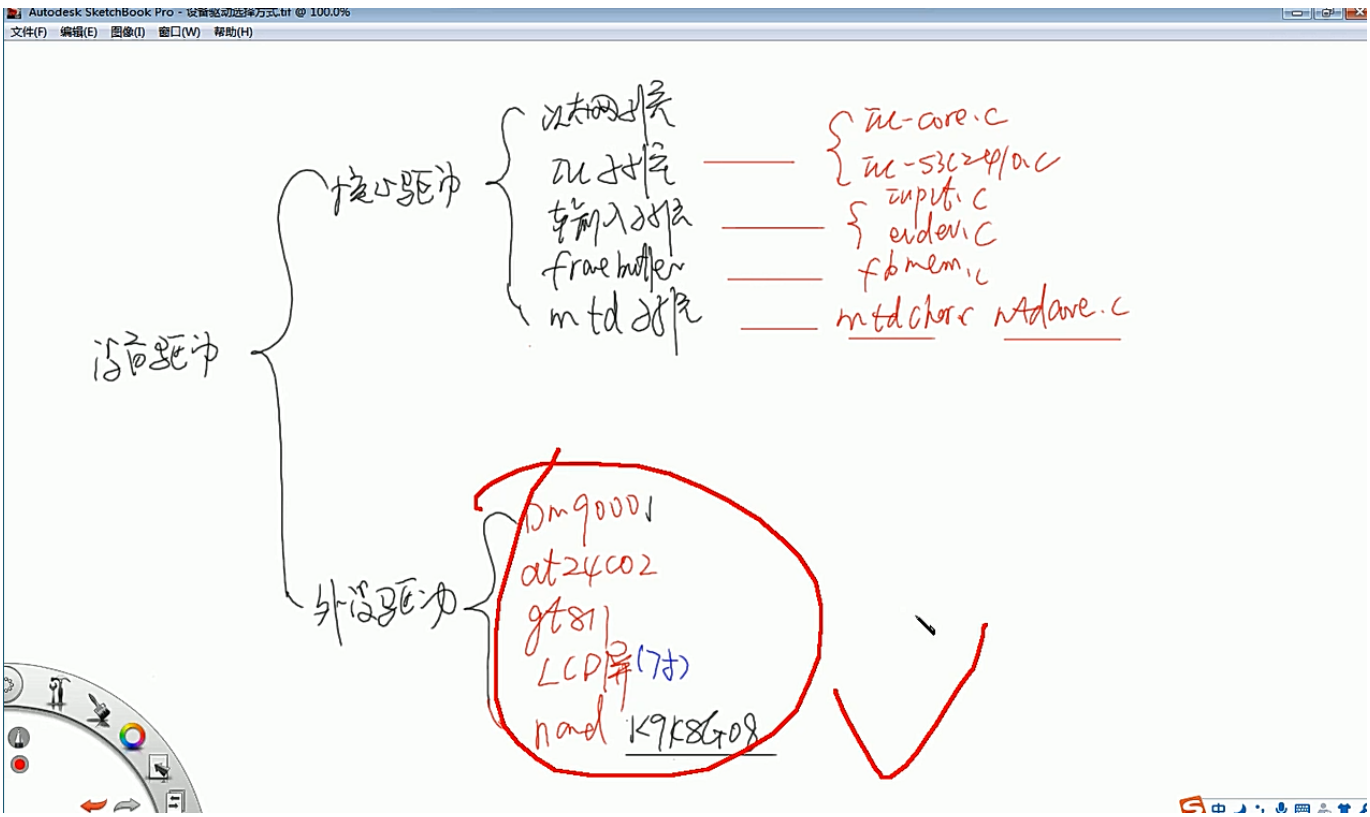

[ ] Patch physical to virtual translations at runtime (EXPERIMENTAL)
General setup --->
[*] Enable loadable module support --->
-*- Enable the block layer --->
System Type --->
Bus support --->
Kernel Features --->
Boot options --->
CPU Power Management --->
Floating point emulation --->
Userspace binary formats --->
Power management options --->
[*] Networking support --->
Device Drivers --->
File systems --->
Kernel hacking --->
Security options --->
-*- Cryptographic API --->
Library routines --->

Load an Alternate Configuration File
↓ (+)
<Select> < Exit > < Help >
来源于s5pv210_defconfig的
默认选项，基本只做微调
网络协议TCP/IP,UDP
设备驱动
文件系统

文件系统类型图



设备驱动选择方式图



10.5. 4. 内核的工作原理

> uImage = 64字节头 + zImage

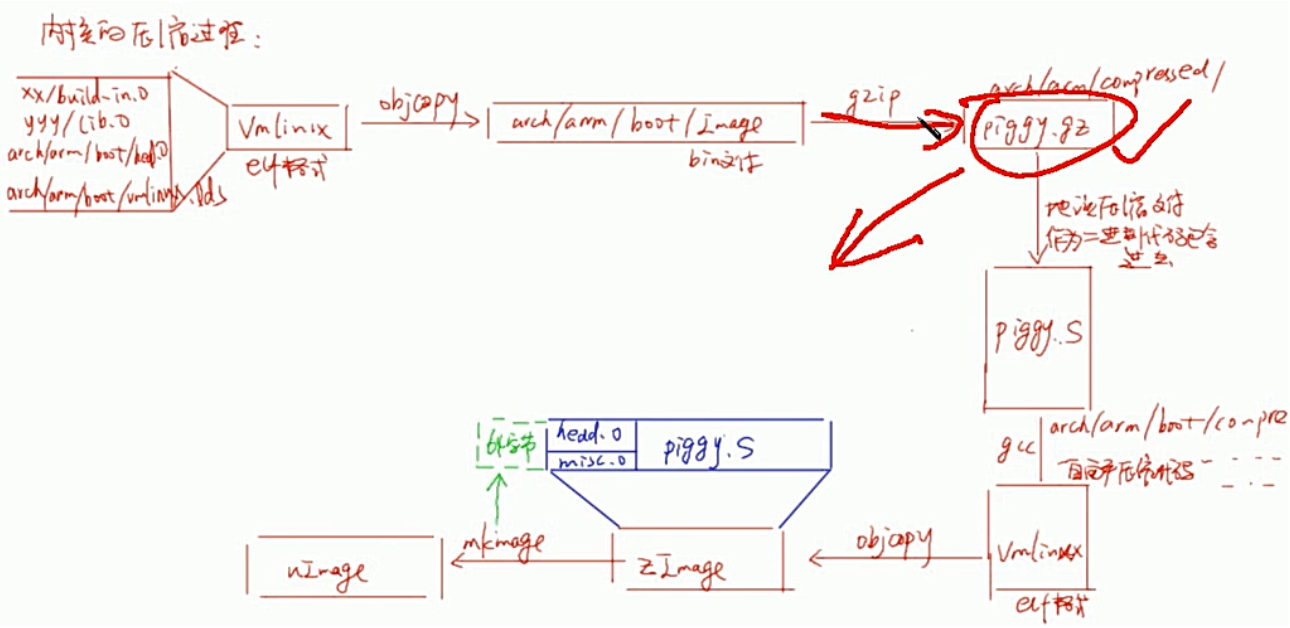
> uImage表示u-boot启动的内核---Linux，bsd，VxWorks，但是启动方式不一样
>
> Linux：arm，x86，mips等架构的内核
>
> 为何u-boot可以启动zImage：因为u-boot做了移植，默认认为zImage就是架构的Linux内核
>
> 64字节头包含了描述内核的信息，u-boot在启动内核时会根据64字节的信息启动不同系统不同架构的内核
>
> mkimage工具：增加64字节的头信息，mkimage是由u-boot提供的。

> 可以手动使用mkimage对zImage进行添加64字节的头

```
dwu@atc-dwu:~$ mkimage
Error: Missing output filename
Usage: mkimage -l image
          -l ==> list image header information
       mkimage [-x] -A arch -O os -T type -C comp -a addr -e ep -n name -d
data_file[:data_file...] image
          -A ==> set architecture to 'arch' //设定架构：arm，mips，x86
          -O ==> set operating system to 'os' //操作系统类型：linux，bsd，VxWorks
          -T ==> set image type to 'type' //镜像类型：kernel，ramdisk
          -C ==> set compression type 'comp' //压缩类型：gzip2，bzip2，none
          -a ==> set load address to 'addr' (hex) //内核被加载到内存的地址(参考
值):0x20008000
          -e ==> set entry point to 'ep' (hex) //内核被执行的地址(参考值):
0x20008000
          -n ==> set image name to 'name' //描述语句：自定义
          -d ==> use image data from 'datafile' //指定增加头部信息的文件---zImage
          -x ==> set XIP (execute in place)
       mkimage [-D dtc_options] [-f fit-image.its|-f auto|-F] [-b <dtb> [-b
<dtb>]] [-i <ramdisk.cpio.gz>] fit-image
           <dtb> file is used with -f auto, it may occur multiple times.
          -D => set all options for device tree compiler
          -f => input filename for FIT source
          -i => input filename for ramdisk file
Signing / verified boot options: [-E] [-B size] [-k keydir] [-K dtb] [ -c
<comment>] [-p addr] [-r] [-N engine]
          -E => place data outside of the FIT structure
          -B => align size in hex for FIT structure and header
          -k => set directory containing private keys
          -K => write public keys to this .dtb file
          -c => add comment in signature node
          -F => re-sign existing FIT image
          -p => place external data at a static position
          -r => mark keys used as 'required' in dtb
          -N => openssl engine to use for signing
       mkimage -V ==> print version information and exit
Use '-T list' to see a list of available image types


dwu@atc-dwu:~/samba/tftproot$ mkimage -A arm -O linux -T kernel -C none -a
0x30008000 -e 0x30008000 -n "linux 3.0.8-jihq" -d zImage uImage_hq
Image Name:   linux 3.0.8-jihq
Created:      Fri Sep 30 11:57:40 2022
Image Type:   ARM Linux Kernel Image (uncompressed)
Data Size:    2077840 Bytes = 2029.14 KiB = 1.98 MiB
Load Address: 30008000
Entry Point:  30008000


x210 # tftp 40008000 uImage_hq
dm9000 i/o: 0x88000300, id: 0x90000a46
```

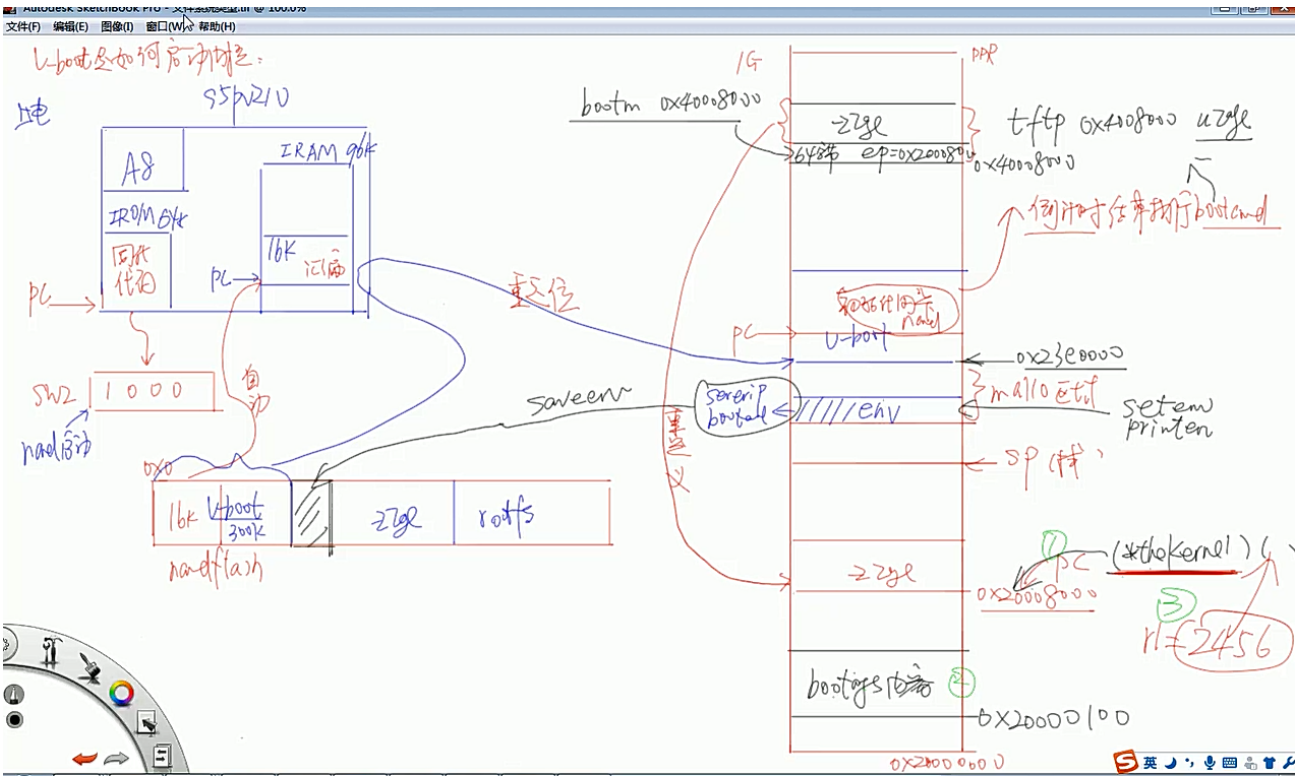
```
DM9000: running in 16 bit mode
MAC: 00:40:5c:26:0a:5b
operating at 100M full duplex mode
TFTP from server 192.100.100.14; our IP address is 192.100.100.34
Filename 'uImage_hq'.
Load address: 0x40008000
Loading: #################################################################
     #################################################################
     #############
done
Bytes transferred = 2077904 (0x1fb4d0)
x210 # bootm 0x40008000
get_format
-------- 1 --------
## Booting kernel from Legacy Image at 40008000 ...
   Image Name:   linux 3.0.8-jihq
   Created:      2022-09-30  11:57:40 UTC
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    2077840 Bytes =  2 MB
   Load Address: 30008000
   Entry Point:  30008000
   Verifying Checksum ... OK
get_format
-------- 1 --------
   Loading Kernel Image ... OK
OK

Starting kernel ...

Uncompressing Linux... done, booting the kernel.
```

Linux内核的压缩过程



u-boot是如何启动内核的?

u-boot启动内核的过程图



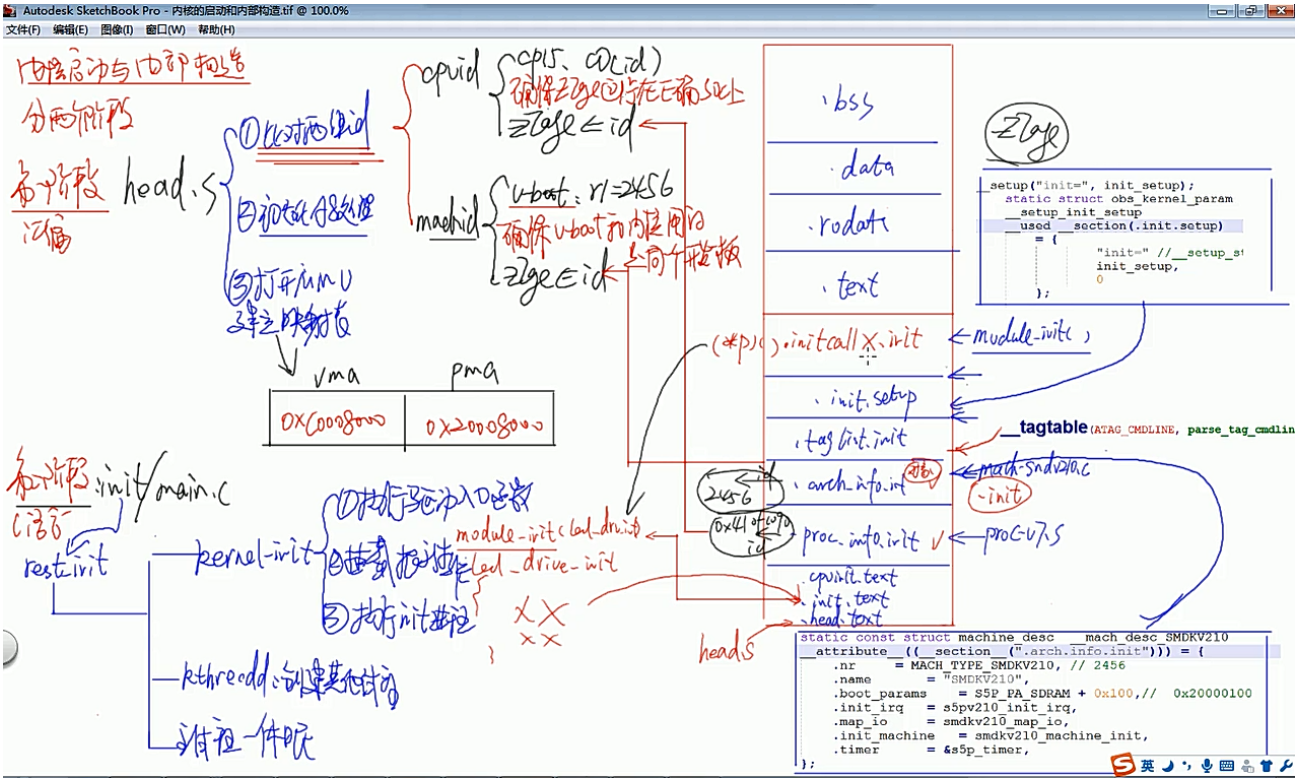
u-boot在交控制权之前，会将内核参数放在物理内存起始+0x100处，以及机器ID传给r1寄存器，然后调用

```
void (*thekernel)(0, machineid, boot_params);
```

```
MACHINE_START(SMDKV210, "SMDKV210")
    /* Maintainer: Kukjin Kim <kgene.kim@samsung.com> */
    .boot_params= S5P_PA_SDRAM + 0x100, //0x30000000 + 0x100，参数固定放在物理内存起始+0x100处
    .init_irq    = s5pv210_init_irq,
    .map_io        = smdkv210_map_io,
    .init_machine    = smdkv210_machine_init,
    .timer        = &s5p_timer,
MACHINE_END
```

内核的内部构造

全局图像



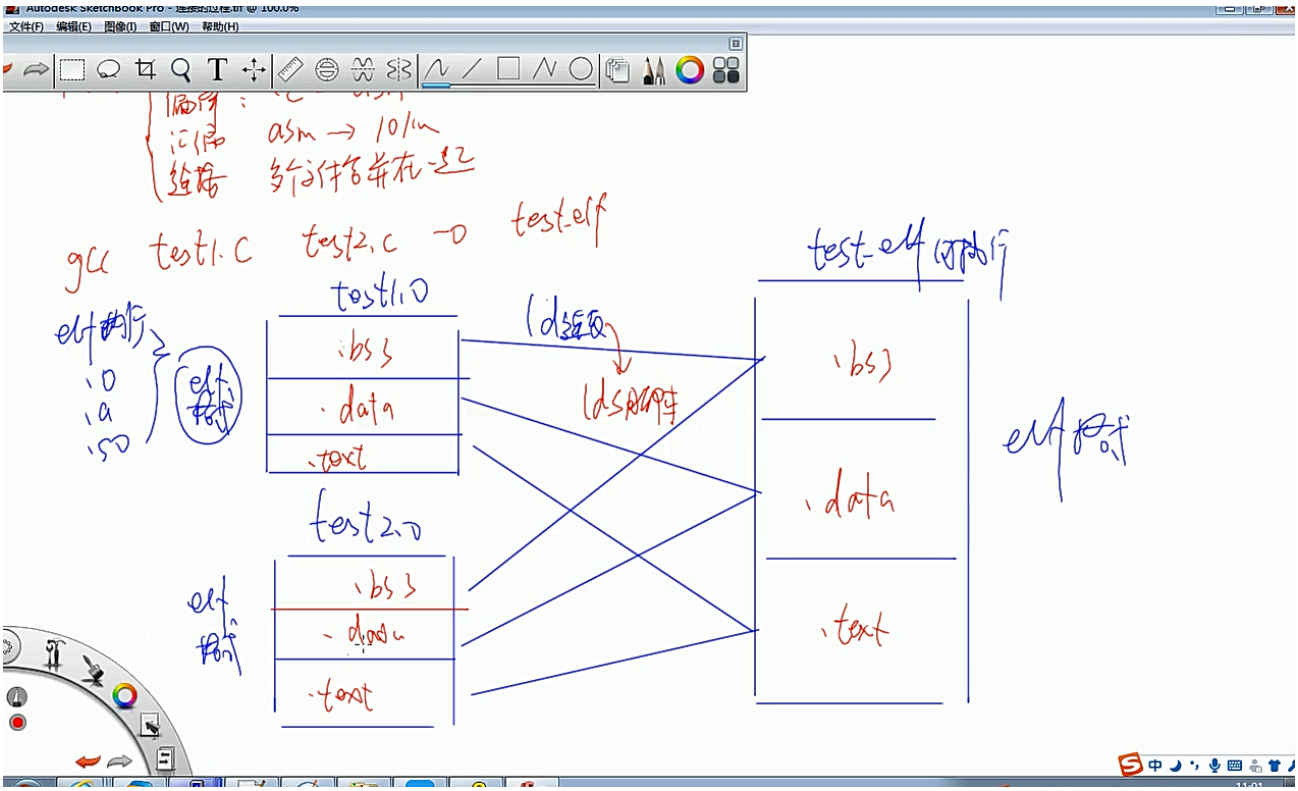
首先要看连接脚本

```
vim Makefile
#搜索vmlinux
560 all: vmlinux
731 # Build vmlinux
732 # ---------------------------------------------------------------------------
733 # vmlinux is built from the objects selected by $(vmlinux-init) and
734 # $(vmlinux-main). Most are built-in.o files from top-level directories
735 # in the kernel tree, others are specified in arch/$(ARCH)/Makefile.
736 # Ordering when linking is important, and $(vmlinux-init) must be first.
737 #
738 # vmlinux
739 #   ^
740 #   |
741 #   +-< $(vmlinux-init)
742 #   |   +--< init/version.o + more
743 #   |
744 #   +--< $(vmlinux-main)
745 #   |    +--< driver/built-in.o mm/built-in.o + more
746 #   |
747 #   +-< kallsyms.o (see description in CONFIG_KALLSYMS section)

758 vmlinux-init := $(head-y) $(init-y)
759 vmlinux-main := $(core-y) $(libs-y) $(drivers-y) $(net-y)
760 vmlinux-all  := $(vmlinux-init) $(vmlinux-main)
761 vmlinux-lds  := arch/$(SRCARCH)/kernel/vmlinux.lds
```

所以链接脚本在arch/arm/kernel/vmlinux.lds 链接就是将elf文件相同的段(.bss, .data, .text)合并在一起 合并到同一文件的源elf文件的排列顺序默认按照先后顺序排，链接脚本可以改变这种顺序

有链接脚本，先后顺序按链接脚本的排列

链接过程图


文件(F) 编辑(E) 图像(I) 窗口(W) 帮助(H)
asm → 10/1m
gcc test1.c test2.c -o test.elf
test1.o
.bss
.data
.text
test2.o
.bss
.data
.text
.bss
.data
.text

基本的链接脚本

```
ld -T连接脚本.lds -Ttext 0x30008000 *.o -o elfname
-T 指定链接脚本
-Ttext 指定text段的链接地址，可以覆盖链接脚本中的text段的起始地址


tftp 40008000 uImage 是将uImage暂存在0x40008000处，暂存地址随意，但不可以是0x30008000，
因为u-boot读取ep入口地址为0x30008000时，会将zImage拷贝到30008000的位置，可能会把后面的
zImage覆盖掉
一般将uImage下载到入口地址 + zImage长度的任意位置
```

```
OUTPUT_ARCH(arm) #输出格式为arm架构
ENTRY(_start)    #表示指定入口标签，第一个执行函数
SECTIONS        #指定目标文件所有段的排布
{
    . = 0x30008000; # . 表示当前的位置，依次从当前位置网上堆叠
    .text : {         #表示目标文件的text段由哪些elf文件的text段组成
        start.o (.text) #强行指定start.o的text段排在最前面
        ledc.o (.text)
        * (.text)    # * 表示其他的文件，先后顺序按makefile的先后排布
    }
    .rodata : {
        * (.rodata)
    }
    .data : {
        * (.data)
    }
    .bss : {
        __bss_start = .; #记录bss段的起始位置
        * (.bss)
        __bss_end = .; #记录bss段的结束位置
        #记录bss段的起始和结束位置是为了清bss使用
    }
}
```

内核的启动及其内部构造

```
OUTPUT_ARCH(arm)
ENTRY(stext)
jiffies = jiffies_64;
SECTIONS
{
 . = 0xC0000000 + 0x00008000;
 .init : { /* Init code and data        */
  _stext = .;
  _sinittext = .;
   *(.head.text)
   *(.init.text) *(.cpuinit.text) *(.meminit.text)

  _einittext = .;
  __proc_info_begin = .;
    *(.proc.info.init) #处理器信息段
  __proc_info_end = .;
  __arch_info_begin = .;
   *(.arch.info.init) #机器信息段,对象
  __arch_info_end = .;
  __tagtable_begin = .;
   *(.taglist.init) #u-boot传递bootargs数据的处理段
  __tagtable_end = .;
  __pv_table_begin = .;
   *(.pv_table)
  __pv_table_end = .;
  . = ALIGN(16);
  __setup_start = .;
  *(.init.setup)    # booargs中各个参数的处理段
  __setup_end = .;
  __initcall_start = .;
  *(.initcallearly.init)

  #驱动函数执行段，这个段全部是函数指针
  #如果批量执行这里的所有函数指针，驱动的入口函数就会被执行
  #内核启动时，通过二级指针遍历这些函数指针进行执行，则所有驱动就会运行
  __early_initcall_end = .;
  *(.initcall0.init) *(.initcall0s.init)
  *(.initcall1.init) *(.initcall1s.init)
  *(.initcall2.init) *(.initcall2s.init)
  *(.initcall3.init) *(.initcall3s.init)
  *(.initcall4.init) *(.initcall4s.init) # subsys_initcall
  *(.initcall5.init) *(.initcall5s.init)
  *(.initcallrootfs.init)
  *(.initcall6.init) *(.initcall6s.init) #module_init
  *(.initcall7.init) *(.initcall7s.init)
  __initcall_end = .;
  __con_initcall_start = .; *(.con_initcall.init) __con_initcall_end = .;
  __security_initcall_start = .; *(.security_initcall.init)
__security_initcall_end = .;
  . = ALIGN(4); __initramfs_start = .; *(.init.ramfs) . = ALIGN(8); *
```

```
(.init.ramfs.info)
  __init_begin = _stext;
  *(.init.data) *(.cpuinit.data) *(.meminit.data) *(.init.rodata) *
(.cpuinit.rodata) *(.meminit.rodata) . = ALIGN(32); __dtb_start = .; *
(.dtb.init.rodata) __dtb_end = .;

 }
 . = ALIGN((1 << 12)); .data..percpu : AT(ADDR(.data..percpu) - 0) {
__per_cpu_load = .; __per_cpu_start = .; *(.data..percpu..first) . = ALIGN((1 <<
12)); *(.data..percpu..page_aligned) . = ALIGN(32); *(.data..percpu..readmostly)
. = ALIGN(32); *(.data..percpu) *(.data..percpu..shared_aligned) __per_cpu_end =
.; }
 . = ALIGN((1 << 12));
 __init_end = .;
 /*
     * unwind exit sections must be discarded before the rest of the
     * unwind sections get included.
     */
 /DISCARD/ : {
  *(.ARM.exidx.exit.text)
  *(.ARM.extab.exit.text)
  *(.ARM.exidx.cpuexit.text)
  *(.ARM.extab.cpuexit.text)
 }
 .text : { /* Real text segment        */
  _text = .; /* Text and read-only data    */
   __exception_text_start = .;
   *(.exception.text)
   __exception_text_end = .;

   . = ALIGN(8); *(.text.hot) *(.text) *(.ref.text) *(.devinit.text) *
(.devexit.text) *(.text.unlikely)
   . = ALIGN(8); __sched_text_start = .; *(.sched.text) __sched_text_end = .;
   . = ALIGN(8); __lock_text_start = .; *(.spinlock.text) __lock_text_end = .;
   . = ALIGN(8); __kprobes_text_start = .; *(.kprobes.text) __kprobes_text_end =
.;
   *(.fixup)
   *(.gnu.warning)
   *(.rodata)
   *(.rodata.*)
   *(.glue_7)
   *(.glue_7t)
  . = ALIGN(4);
  *(.got) /* Global offset table        */

 }
 . = ALIGN(((1 << 12))); .rodata : AT(ADDR(.rodata) - 0) { __start_rodata = .; *
(.rodata) *(.rodata.*) *(__vermagic) . = ALIGN(8); __start___tracepoints_ptrs =
.; *(__tracepoints_ptrs) __stop___tracepoints_ptrs = .; *(__markers_strings) *
(__tracepoints_strings) } .rodata1 : AT(ADDR(.rodata1) - 0) { *(.rodata1) }
.pci_fixup : AT(ADDR(.pci_fixup) - 0) { __start_pci_fixups_early = .; *
```

```
(.pci_fixup_early) __end_pci_fixups_early = .; __start_pci_fixups_header = .; *
(.pci_fixup_header) __end_pci_fixups_header = .; __start_pci_fixups_final = .; *
(.pci_fixup_final) __end_pci_fixups_final = .; __start_pci_fixups_enable = .; *
(.pci_fixup_enable) __end_pci_fixups_enable = .; __start_pci_fixups_resume = .; *
(.pci_fixup_resume) __end_pci_fixups_resume = .; __start_pci_fixups_resume_early
= .; *(.pci_fixup_resume_early) __end_pci_fixups_resume_early = .;
__start_pci_fixups_suspend = .; *(.pci_fixup_suspend) __end_pci_fixups_suspend =
.; } .builtin_fw : AT(ADDR(.builtin_fw) - 0) { __start_builtin_fw = .; *
(.builtin_fw) __end_builtin_fw = .; } .rio_ops : AT(ADDR(.rio_ops) - 0) {
__start_rio_switch_ops = .; *(.rio_switch_ops) __end_rio_switch_ops = .; }
__ksymtab : AT(ADDR(__ksymtab) - 0) { __start___ksymtab = .; *
(SORT(___ksymtab+*)) __stop___ksymtab = .; } __ksymtab_gpl :
AT(ADDR(__ksymtab_gpl) - 0) { __start___ksymtab_gpl = .; *
(SORT(___ksymtab_gpl+*)) __stop___ksymtab_gpl = .; } __ksymtab_unused :
AT(ADDR(__ksymtab_unused) - 0) { __start___ksymtab_unused = .; *
(SORT(___ksymtab_unused+*)) __stop___ksymtab_unused = .; } __ksymtab_unused_gpl :
AT(ADDR(__ksymtab_unused_gpl) - 0) { __start___ksymtab_unused_gpl = .; *
(SORT(___ksymtab_unused_gpl+*)) __stop___ksymtab_unused_gpl = .; }
__ksymtab_gpl_future : AT(ADDR(__ksymtab_gpl_future) - 0) {
__start___ksymtab_gpl_future = .; *(SORT(___ksymtab_gpl_future+*))
__stop___ksymtab_gpl_future = .; } __kcrctab : AT(ADDR(__kcrctab) - 0) {
__start___kcrctab = .; *(SORT(___kcrctab+*)) __stop___kcrctab = .; }
__kcrctab_gpl : AT(ADDR(__kcrctab_gpl) - 0) { __start___kcrctab_gpl = .; *
(SORT(___kcrctab_gpl+*)) __stop___kcrctab_gpl = .; } __kcrctab_unused :
AT(ADDR(__kcrctab_unused) - 0) { __start___kcrctab_unused = .; *
(SORT(___kcrctab_unused+*)) __stop___kcrctab_unused = .; } __kcrctab_unused_gpl :
AT(ADDR(__kcrctab_unused_gpl) - 0) { __start___kcrctab_unused_gpl = .; *
(SORT(___kcrctab_unused_gpl+*)) __stop___kcrctab_unused_gpl = .; }
__kcrctab_gpl_future : AT(ADDR(__kcrctab_gpl_future) - 0) {
__start___kcrctab_gpl_future = .; *(SORT(___kcrctab_gpl_future+*))
__stop___kcrctab_gpl_future = .; } __ksymtab_strings : AT(ADDR(__ksymtab_strings)
- 0) { *(__ksymtab_strings) } __init_rodata : AT(ADDR(__init_rodata) - 0) { *
(.ref.rodata) *(.devinit.rodata) *(.devexit.rodata) } __param : AT(ADDR(__param)
- 0) { __start___param = .; *(__param) __stop___param = .; } __modver :
AT(ADDR(__modver) - 0) { __start___modver = .; *(__modver) __stop___modver = .; .
= ALIGN(((1 << 12))); __end_rodata = .; } . = ALIGN(((1 << 12)));
 /*
     * Stack unwinding tables
     */
 . = ALIGN(8);
 .ARM.unwind_idx : {
  __start_unwind_idx = .;
  *(.ARM.exidx*)
  __stop_unwind_idx = .;
 }
 .ARM.unwind_tab : {
  __start_unwind_tab = .;
  *(.ARM.extab*)
  __stop_unwind_tab = .;
 }
 _etext = .; /* End of text and rodata section */
```

```
  . = ALIGN(8192);
  __data_loc = .;
  .data : AT(__data_loc) {
   _data = .; /* address in memory */
   _sdata = .;
   /*
          * first, the init task union, aligned
          * to an 8192 byte boundary.
          */
   . = ALIGN(8192); *(.data..init_task)
   . = ALIGN((1 << 12)); __nosave_begin = .; *(.data..nosave) . = ALIGN((1 <<
12)); __nosave_end = .;
   . = ALIGN(32); *(.data..cacheline_aligned)
   . = ALIGN(32); *(.data..read_mostly) . = ALIGN(32);
   /*
          * The exception fixup table (might need resorting at runtime)
          */
   . = ALIGN(32);
   __start___ex_table = .;
   *(__ex_table)
   __stop___ex_table = .;
   /*
          * and the usual data section
          */
   *(.data) *(.ref.data) *(.data..shared_aligned) *(.devinit.data) *
(.devexit.data) . = ALIGN(32); *(__tracepoints) . = ALIGN(8);
__start___jump_table = .; *(__jump_table) __stop___jump_table = .; . = ALIGN(8);
__start___verbose = .; *(__verbose) __stop___verbose = .;
   CONSTRUCTORS
   _edata = .;
  }
  _edata_loc = __data_loc + SIZEOF(.data);
  .notes : AT(ADDR(.notes) - 0) { __start_notes = .; *(.note.*) __stop_notes = .;
}
  . = ALIGN(0); __bss_start = .; . = ALIGN(0); .sbss : AT(ADDR(.sbss) - 0) { *
(.sbss) *(.scommon) } . = ALIGN(0); .bss : AT(ADDR(.bss) - 0) { *
(.bss..page_aligned) *(.dynbss) *(.bss) *(COMMON) } . = ALIGN(0); __bss_stop = .;
  _end = .;
  .stab 0 : { *(.stab) } .stabstr 0 : { *(.stabstr) } .stab.excl 0 : { *
(.stab.excl) } .stab.exclstr 0 : { *(.stab.exclstr) } .stab.index 0 : { *
(.stab.index) } .stab.indexstr 0 : { *(.stab.indexstr) } .comment 0 : { *
(.comment) }
  .comment 0 : { *(.comment) }
  /* Default discards */
  /DISCARD/ : { *(.exit.text) *(.cpuexit.text) *(.memexit.text) *(.exit.data) *
(.cpuexit.data) *(.cpuexit.rodata) *(.memexit.data) *(.memexit.rodata) *
(.exitcall.exit) *(.discard) *(.discard.*) }
  /DISCARD/ : {
   *(.alt.smp.init)
  }
}
```

```
/*
 * These must never be empty
 * If you have to comment these two assert statements out, your
 * binutils is too old (for other reasons as well)
 */
ASSERT((__proc_info_end - __proc_info_begin), "missing CPU support")
ASSERT((__arch_info_end - __arch_info_begin), "no machine record defined")
```

module_init解析

```
module_init(led_drv_init); //是一个函数指针，指针指向参数
#define module_init(x)    __initcall(x);
  #define __initcall(fn) device_initcall(fn)
    #define device_initcall(fn)        __define_initcall("6",fn,6)
      #define __define_initcall(level,fn,id) \
    static initcall_t __initcall_##fn##id __used \
    __attribute__((__section__(".initcall" level ".init"))) = fn

typedef int (*initcall_t)(void);
static initcall_t __initcall_led_drv_init6 __used
__attribute__((__section__(".initcall6.init"))) = led_drv_init
```

```
MACHINE_START(SMDKV210, "SMDKV210")
    /* Maintainer: Kukjin Kim <kgene.kim@samsung.com> */
    .boot_params= S5P_PA_SDRAM + 0x100, //0x30000000 + 0x100，参数固定放在物理内存
起始+0x100处
    .init_irq    = s5pv210_init_irq,
    .map_io        = smdkv210_map_io,
    .init_machine    = smdkv210_machine_init,
    .timer        = &s5p_timer,
MACHINE_END

#define MACHINE_START(_type,_name)            \
static const struct machine_desc __mach_desc_##_type    \
 __used                                \
 __attribute__((__section__(".arch.info.init"))) = {    \
    .nr        = MACH_TYPE_##_type,        \
    .name        = _name,

#define MACHINE_END                \
};


static const struct machine_desc __mach_desc_SMDKV210 \
__used __attribute__((__section__(".arch.info.init"))) = {
    .nr        = MACH_TYPE_SMDKV210,        \
    .name        = "SMDKV210",
    .boot_params= S5P_PA_SDRAM + 0x100, //0x30000000 + 0x100，参数固定放在物理内存
起始+0x100处
    .init_irq    = s5pv210_init_irq,
    .map_io        = smdkv210_map_io,
    .init_machine    = smdkv210_machine_init,
    .timer        = &s5p_timer,
}
.section ".proc.info.init", #alloc, #execinstr

    .type   __v7_ca9mp_proc_info, #object
__v7_ca9mp_proc_info:
    .long    0x410fc090        @ Required ID value
    .long    0xff0ffff0        @ Mask f        or ID
dwu@atc-dwu:~/samba/linux-3.0.8$ grep MACH_TYPE_SMDKV210 -rHn ./include/
./include/generated/mach-types.h:417:#define MACH_TYPE_SMDKV210            2456
./include/generated/mach-types.h:5937:#  define machine_arch_type
MACH_TYPE_SMDKV210
./include/generated/mach-types.h:5939:# define machine_is_smdkv210()
(machine_arch_type == MACH_TYPE_SMDKV210)

struct machine_desc {
    unsigned int        nr;        /* architecture number    */
    const char        *name;        /* architecture name    */
    unsigned long        boot_params;    /* tagged list        */
    const char        **dt_compat;    /* array of device tree
```

```
                         * 'compatible' strings    */

    unsigned int        nr_irqs;    /* number of IRQs */

    unsigned int        video_start;    /* start of video RAM    */
    unsigned int        video_end;    /* end of video RAM    */

    unsigned int        reserve_lp0 :1;    /* never has lp0    */
    unsigned int        reserve_lp1 :1;    /* never has lp1    */
    unsigned int        reserve_lp2 :1;    /* never has lp2    */
    unsigned int        soft_reboot :1;    /* soft reboot        */
    void            (*fixup)(struct machine_desc *,
                     struct tag *, char **,
                     struct meminfo *);
    void            (*reserve)(void);/* reserve mem blocks    */
    void            (*map_io)(void);/* IO mapping function    */
    void            (*init_early)(void);
    void            (*init_irq)(void);
    struct sys_timer    *timer;        /* system tick timer    */
    void            (*init_machine)(void);
#ifdef CONFIG_MULTI_IRQ_HANDLER
    void            (*handle_irq)(struct pt_regs *);
#endif
};
```

```
#define __setup(str, fn)                    \
    __setup_param(str, fn, fn, 0)

#define __setup_param(str, unique_id, fn, early)            \
    static const char __setup_str_##unique_id[] __initconst    \
        __aligned(1) = str; \
    static struct obs_kernel_param __setup_##unique_id    \
        __used __section(.init.setup)            \
        __attribute__((aligned((sizeof(long)))))    \
        = { __setup_str_##unique_id, fn, early }

__setup("init=", init_setup);
  static const char __setup_str_init_setup[] \
    __section(.init.rodata)\
    __aligned(1) = "init="; \
static struct obs_kernel_param __setup_init_setup    \
        __used __section(.init.setup)            \
        __attribute__((aligned((sizeof(long)))))    \
        = { "init=", init_setup, 0}
```

/proc/cmdline文件存放u-boot存的参数

内核启动流程

第一阶段head.S

1. 比对两组ID
2. 初始化A8处理器
3. 打开mmu，建立映射表

```
//第一阶段：arch\arm\kernel\head.S
    __HEAD
ENTRY(stext)
    setmode    PSR_F_BIT | PSR_I_BIT | SVC_MODE, r9 @ ensure svc mode
                        @ and irqs disabled
    mrc    p15, 0, r9, c0, c0        @ get processor id
    bl    __lookup_processor_type        @ r5=procinfo r9=cpuid
    bl    __create_page_tables//建立mmu映射表

    /*
     * The following calls CPU specific code in a position independent
     * manner.  See arch/arm/mm/proc-*.S for details.  r10 = base of
     * xxx_proc_info structure selected by __lookup_processor_type
     * above.  On return, the CPU will be ready for the MMU to be
     * turned on, and r0 will hold the CPU control register value.
     */
ldr    r13, =__mmap_switched        @ address to jump to after
__mmap_switched: //head_common.S, 会跳到第二阶段 b start_kernel
                        @ mmu has been enabled
    adr    lr, BSYM(1f)            @ return (PIC) address
    mov    r8, r4                @ set TTBR1 to swapper_pg_dir
 ARM(    add    pc, r10, #PROCINFO_INITFUNC    )
 THUMB(    add    r12, r10, #PROCINFO_INITFUNC    )
 THUMB(    mov    pc, r12                )
1:    b    __enable_mmu
ENDPROC(stext)
    .ltorg
#ifndef CONFIG_XIP_KERNEL
2:    .long    .
    .long    PAGE_OFFSET
#endif
```

第二阶段：init/main.c

start_kernel

```
asmlinkage void __init start_kernel(void)
 printk(KERN_NOTICE "%s", linux_banner);
 setup_arch(&command_line); //初始化cpu和机器代码，找到机器描述对象
    //通过machineid找到对应的struct machine_desc
    //从0x30000100中获取到了u-boot传递过来的参数
    mdesc = setup_machine_tags(machine_arch_type);//machine_arch_type=2456
        //__arch_info_begin和__arch_info_end在链接脚本中定义
        for (p = __arch_info_begin; p < __arch_info_end; p++) {
            if (nr == p->nr) {
                printk("Machine: %s\n", p->name);
                mdesc = p;
                break;
            }
        }
        return mdesc

    machine_desc = mdesc;//将遍历到的desc 赋值给machine_desc
 printk(KERN_NOTICE "Kernel command line: %s\n", boot_command_line);
    parse_early_param();
    //分析u-boot传进来的每个参数
    //Kernel command line: console=ttySAC2,115200 root=/dev/nfs rw
nfsroot=192.100.100.14:/home/dwu/samba/nfsroot/rootfs,v3
ip=192.100.100.44:192.100.100.14:192.100.100.1:255.255.255.0::eth0:off
earlyprintk init=/linuxrc
    //有一个应用：在bootargs中可以自定义传递参数，一般用于开发u-boot，给内核传各种参数
    //set bootargs myname=ruhua myage=20
    /*
     内核中需要拿到ruhua和20，就必须在任何一个文件中添加如下内容
     eg:在init/main.c中
    static int __init myname_setup(char *str)
    {
        printk("myname = %s \n", str);
        return 1;
    }
    __setup("myname=", myname_setup);

    static int __init myage_setup(char *str)
    {
        int age = simple_strtoul(str);
        printk("age= %d \n", age );
        return 1;
    }
    __setup("myage=", myage_setup);
    */

    parse_args("Booting kernel", static_command_line, __start___param,
           __stop___param - __start___param,
           &unknown_bootoption);
...
```

```
rest_init();//开辟3个线程
//1. kernel_init
    //a. 执行所有驱动入口函数
    //b. 挂载根文件系统
    //c. 执行祖先init进程

    kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND);
        static int __init kernel_init(void * unused)
        //a. 执行所有驱动入口函数
            do_basic_setup();//执行所有驱动入口函数
                do_initcalls();
                    for (fn = __early_initcall_end; fn < __initcall_end; fn++)
                        do_one_initcall(*fn);
                            ret = fn(); //批量的执行了initcallX.init段中所有的函数指
针
                                        //也就是执行了所有的驱动入口函数

dwu@atc-dwu:~/samba/linux-3.0.8$ grep -rHn ">init_machine" arch/arm/
arch/arm/kernel/setup.c:732:    if (machine_desc->init_machine)
arch/arm/kernel/setup.c:733:        machine_desc->init_machine();

static int __init customize_machine(void)
{
    /* customizes platform devices, or adds new ones */
    if (machine_desc->init_machine)
        machine_desc->init_machine();
    return 0;
}
arch_initcall(customize_machine);//在do_basic_setup();中调用

    //b. 挂载根文件系统
    prepare_namespace(); //挂载根文件系统

    //c. 执行祖先init进程
    init_post(); //执行祖先init进程
        if (execute_command) { //init="/linuxrc"  execute_command = "/linuxrc"
            //优先执行bootargs中指定的祖先进程
            run_init_process(execute_command);
            printk(KERN_WARNING "Failed to execute %s.  Attempting "
                    "defaults...\n", execute_command);
        }
        //如果没有执行，就执行以下备胎；成功了就没后面的事了，因为会将该程序的各段加载
到内存
        run_init_process("/sbin/init");
        run_init_process("/etc/init");
        run_init_process("/bin/init");
        run_init_process("/bin/sh");

        panic("No init found.  Try passing init= option to kernel. "
              "See Linux Documentation/init.txt for guidance.");
```

```
//2. kthreadd: 创建其他线程
pid = kernel_thread(kthreadd, NULL, CLONE_FS | CLONE_FILES);
create_kthread(create);//死循环，创建其他内核线程
//3. 主线程：睡眠
schedule();//让出调度权
```

调用机器初始化的过程