第 9 章 字符设备驱动程序

字符设备通常是指只能顺序访问，数据传输量较低的设备。字符设备的类型多种多样，是系统中数量最多的设备。字符设备驱动在设备驱动数据库中由 cdev 结构体表示，字符设备驱动程序的主要工作就是创建并向内核注册 cdev 实例。

在通用驱动模型中，设备由 device 结构体（通常由另一结构体封装）表示，驱动由 device_driver 结构体（通常由另一结构体封装）表示，设备和驱动挂接到总线上。向总线注册设备或驱动时，将触发总线上设备与驱动的匹配，匹配成功将调用驱动（或总线）的 probe()函数，在此函数内将创建设备驱动程序 cdev 结构体实例并注册，这称之为加载设备驱动程序，从而进程可通过设备文件访问设备了。

本章首先介绍字符设备驱动程序通用框架，然后重点介绍几种常见字符设备驱动程序的实现。

9.1 驱动程序框架

字符设备驱动程序框架相对于块设备驱动程序来说稍微简单一些，驱动程序首先要向内核申请设备号（设备号区间），确定设备号可用，然后定义字符设备的文件操作结构 file_operations 实例，最后创建并设置 cdev 实例，最后添加到字符设备驱动数据库。

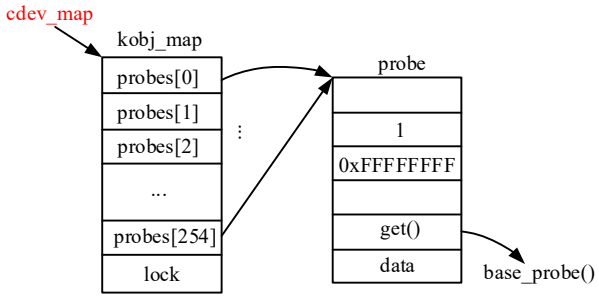
9.1.1 初始化

前面第 8 章介绍过，内核通过 kobj_map 数据结构管理设备驱动程序。内核在初始化阶段将调用函数 **chrdev_init()**为字符设备创建 kobj_map 结构体实例。

函数调用关系为：start_kernel()->vfs_caches_init()->chrdev_init()，chrdev_init()函数在/fs/char_dev.c 文件内实现：

```
static struct kobj_map *cdev_map;       /*全局变量，指向管理字符设备驱动程序的 kobj_map 实例*/

void __init chrdev_init(void)
{
    cdev_map = kobj_map_init(base_probe, &chrdevs_lock);     /*创建并初始化 kobj_map 实例*/
}
```

kobj_map_init()函数在第 8 章介绍过了，主要是创建 kobj_map 数据结构实例，实例中各指针数组项都关联初始的 probe 实例，如下图所示。



字符设备驱动数据库

初始 probe 实例中 get()函数指针为 base_probe()函数（正常是用于获取跟踪驱动数据结构实例的 kobject 实例），这里主要工作是加载模块：

```
static struct kobject *base_probe(dev_t dev, int *part, void *data)
{
```

```
    if (request_module("char-major-%d-%d", MAJOR(dev), MINOR(dev)) > 0)
            request_module("char-major-%d", MAJOR(dev));
    return NULL;
}
```

9.1.2 设备驱动程序

字符设备驱动程序的主要流程是（在 xxx_driver 实例中的 probe()函数内实现）：

（1）向内核申请或分配设备号，以保证系统内设备号的唯一性。

（2）实现字符设备文件操作结构 file_operations 实例。

（3）创建、设置并添加表示字符设备驱动的 cdev 实例。

1 申请/分配设备号

字符设备驱动程序首先需要为字符设备申请/分配设备号，确保设备号可用后（在系统内是唯一的），才可赋予 cdev 实例。由于字符设备数量和类型比较多，因此内核对字符设备号的管理稍显复杂。

内核定义了 char_device_struct 结构体用于管理字符设备的设备号，结构体定义在/fs/char_dev.c 文件内：

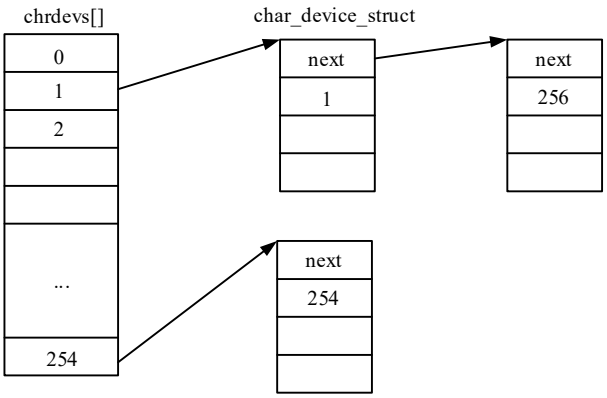
```
static struct char_device_struct {
    struct char_device_struct *next;      /*指向下一数据结构实例，实例组成单链表*/
    unsigned int major;                   /*主设备号（12bit）*/
    unsigned int baseminor;               /*起始从设备号（20bit）*/
    int minorct;                          /*从设备号数量*/
    char name[64];                        /*名称*/
    struct cdev *cdev;                    /* will die */
} *chrdevs[CHRDEV_MAJOR_HASH_SIZE];
```

每个 char_device_struct 结构体实例表示某一主设备号下已经注册的一段从设备号，从设备号范围是：[baseminor,baseminor+minorct-1]。

chrdevs[]为 char_device_struct 结构体指针数组（散列表）用于管理 char_device_struct 实例。数组项数定义在/include/linux/fs.h 头文件内：

```
#define CHRDEV_MAJOR_HASH_SIZE  255        /*散列值 major%255*/
```

char_device_struct 实例通过散列值 major%255 添加到散列表，如下图所示：

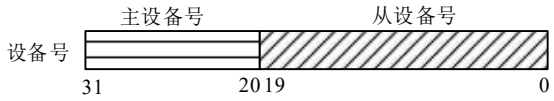


字符设备设备号支持静态申请和动态分配。静态申请需指定主设备号和从设备号范围，如果申请设备号未被使用，申请成功，否则申请失败。动态分配需指定从设备号范围，主设备号由内核动态分配，内核

会查找尚未使用的主设备号，返回调用者。

■静态申请设备号

在内核中，设备号由 32 位无符号整数表示（dev_t），如下图所示，高 12 位表示主设备号，低 20 位表示从设备号。



静态申请设备号的函数为 **register_chrdev_region**(dev_t from, unsigned count, const char *name)，此函数在/fs/char_dev.c 文件内实现，代码如下：

```
int register_chrdev_region(dev_t from, unsigned count, const char *name)
/*form：设备号，内含主设备号和起始从设备号，count：从设备号数量，name：名称字符串指针*/
{
    struct char_device_struct *cd;
    dev_t    to = from + count;            /*最大从设备号*/
    dev_t    n, next;

    for (n = from; n < to; n = next) {
        /*如果(MINOR(n)+count)大于从设备号最大值，需申请多个主设备号*/
        next = MKDEV(MAJOR(n)+1, 0);
        if (next > to)
            next = to;
        cd = __register_chrdev_region(MAJOR(n), MINOR(n),    next - n, name);
                                        /*申请同一主设备号下的一段从设备号，/fs/char_dev.c*/
        if (IS_ERR(cd))
            goto fail;
    }
    return 0;          /*申请成功返回 0*/
fail:
    ...
    return PTR_ERR(cd);       /*申请失败返回错误码*/
}
```

参数 from 包含主设备号和起始从设备号，count 表示从设备号数量，name 为名称字符串指针。函数首先判断(MINOR(n)+count)是否大于从设备号最大值（从设备号由低 20 位表示），如果是则需要申请多个主设备号。这里只考虑只需一个主设备号的情况，此时只需调用一次__register_chrdev_region()函数申请设备号。

__register_chrdev_region()函数在/fs/char_dev.c 文件内实现，用于申请（或分配）某一主设备号下的一段从设备号，函数代码如下：

```
static struct char_device_struct *__register_chrdev_region(unsigned int major, unsigned int baseminor,   \
                                                           int minorct, const char *name)
/* major：主设备号，baseminor：起始从设备号，minorct：从设备数量，name：名称字符串*/
{
    struct char_device_struct *cd, **cp;
```

```
int ret = 0;
int i;

cd = kzalloc(sizeof(struct char_device_struct), GFP_KERNEL);   /*创建 char_device_struct 实例*/
...
mutex_lock(&chrdevs_lock);

/*如果主设备号为 0 则表示动态分配主设备号，用于执行 alloc_chrdev_region()函数*/
if (major == 0) {
    for (i = ARRAY_SIZE(chrdevs)-1; i > 0; i--) {
        if (chrdevs[i] == NULL)        /*从后至前查找第一个值为空的 chrdevs[]数组项*/
            break;
    }
    /*如果内核已经使用了(0,254]的主设备号，分配将不成功，此处不正确！！*/
    /*因为主设备号由 12 位表示，可以大于 254，后续版本进行了修改！！*/
    if (i == 0) {
        ret = -EBUSY;
        goto out;
    }
    major = i;      /*数组项索引值就是动态分配的主设备号*/
}

/*主设备号已经确定，检查从设备号范围是否与现有的 char_device_struct 实例有重叠*/
cd->major = major;
cd->baseminor = baseminor;
cd->minorct = minorct;
strlcpy(cd->name, name, sizeof(cd->name));   /*复制名称字符串*/

i = major_to_index(major);      /*散列值，major%255*/

for (cp = &chrdevs[i]; *cp; cp = &(*cp)->next)     /*遍历散列链表成员，确定插入点*/
    if ((*cp)->major > major || ((*cp)->major == major &&(((*cp)->baseminor >= baseminor) ||
                                    ((*cp)->baseminor + (*cp)->minorct > baseminor))))
        break;

/*检查从设备号是否与已注册从设备号有重叠，有重叠申请将失败*/
if (*cp && (*cp)->major == major) {
    int old_min = (*cp)->baseminor;
    int old_max = (*cp)->baseminor + (*cp)->minorct - 1;
    int new_min = baseminor;
    int new_max = baseminor + minorct - 1;

    if (new_max >= old_min && new_max <= old_max) {
```

```
                ret = -EBUSY;
                goto out;
            }

            if (new_min <= old_max && new_min >= old_min) {
                ret = -EBUSY;
                goto out;
            }
        }

        cd->next = *cp;
        *cp = cd;                    /*将新创建的 char_device_struct 实例插入到散列链表*/
        mutex_unlock(&chrdevs_lock);
        return cd;                   /*成功返回 char_device_struct 实例指针*/
    out:
        ...
        return ERR_PTR(ret);       /*失败返回错误码*/
    }
```

__register_chrdev_region()函数根据 major 参数是否为 0，确定是否是动态分配主设备号，如果是动态分配主设备号，则在散列表中从后至前查找第一个未使用的主设备号。

若 major 不为 0，则申请指定的主设备号。确定主设备号后再检查从设备号范围是否与已注册从设备号有重叠，如果有重叠，则申请失败，没有重叠则将 char_device_struct 实例插入到散列链表中合适位置。

char_device_struct 实例在散列链表中按主设备号从小到大在链表中从左至右依次排序，若主设备号相同，则按从设备号从小到大依次排序。

注销设备号的函数为 unregister_chrdev_region(dev_t from, unsigned count)，源代码请读者自行阅读。

■动态分配主设备号

字符设备驱动程序也可以不指定主设备号（但需要指定从设备号区间），而由内核动态分配主设备号。动态分配主设备号的函数为 **alloc_chrdev_region()**，函数定义在/fs/char_dev.c 文件内：

```
int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count,const char *name)
/*dev：保存分配的设备号，baseminor：起始从设备号，count：从设备号数量，name：名称*/
{
    struct char_device_struct *cd;
    cd = __register_chrdev_region(0, baseminor, count, name); /*注意第一个参数为 0，上面介绍过*/
    if (IS_ERR(cd))
        return PTR_ERR(cd);
    *dev = MKDEV(cd->major, cd->baseminor);
                                /*保存分配的设备号，含主设备号和起始从设备号*/
    return 0;        /*成功返回 0*/
}
```

alloc_chrdev_region()函数内调用前面介绍的__register_chrdev_region()函数，主设备号参数为 0，表示由内核动态分配主设备号。函数成功返回 0，否则返回错误码，参数 dev 指向的 dev_t 实例保存了分配的设备号，含主设备号和起始从设备号。

动态分配设备号只能分配主设备号，不管是静态申请还是动态分配设备号函数都需要指定从设备号的区间，并且内核不会对其进行修改，如果从设备号范围与已注册的从设备号有重叠，申请/分配将失败，函数返回错误码，请读者注意。

2 数据结构

内核通过设备文件访问设备（与普通文件访问接口相同），字符设备驱动程序需要为设备实现文件操作 file_operations 结构体实例并赋予 cdev 实例，这是内核操作字符设备的接口。

字符设备在设备驱动数据库中，由 cdev 结构体表示，结构体定义在/include/linux/cdev.h 头文件：

```
struct cdev {
    struct kobject kobj;                    /*跟踪（管理）cdev 实例的 kobject 实例*/
    struct module *owner;                   /*模块指针*/
    const struct file_operations *ops;      /*文件操作结构指针*/
    struct list_head list;                  /*管理设备文件 inode 实例，每个从设备都有一个设备文件*/
    dev_t dev;                              /*设备号，含主设备号和起始从设备号*/
    unsigned int count;                     /*从设备号数量*/
};
```

cdev 结构体主要成员简介如下：

●kobj：kobject 结构体实例，用于跟踪管理 cdev 实例；

●owner：驱动程序模块指针；

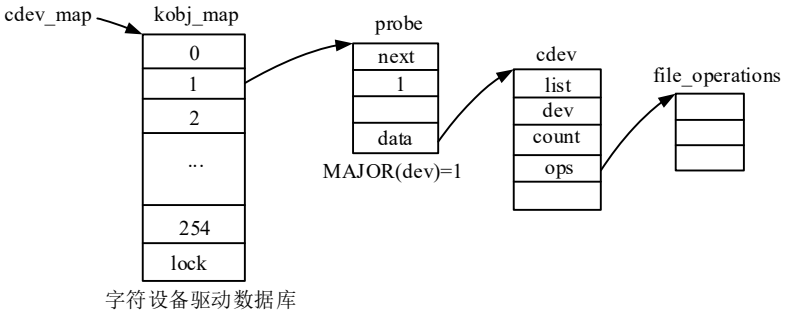
●**ops**：文件操作 file_operations 结构体指针，内核操作字符设备的接口；

●dev：设备号，含主设备号和起始从设备号；

●count：表示从设备号数量，cdev 实例表示的从设备号范围是[MINORS(dev)，MINORS(dev)+count-1]；

●list：链接 cdev 表示设备的设备文件 inode 实例，每个从设备都有一个设备文件，cdev 实例可适用于同一主设备号下的多个从设备。

内核设备驱动数据库通过 kobj_map 结构管理 cdev 实例，如下图所示，cdev 实例创建后需要添加到设备驱动数据库中，添加函数为 kobj_map()，详见第 8 章。



3 创建/添加 cdev 实例

字符设备驱动 cdev 实例可以静态定义，也可以动态创建。静态定义的 cdev 实例，需要调用 **cdev_init()** 函数进行初始化，函数定义如下：

```
void cdev_init(struct cdev *cdev, const struct file_operations *fops)
{
    memset(cdev, 0, sizeof *cdev);          /*清零*/
```

```
    INIT_LIST_HEAD(&cdev->list);
    kobject_init(&cdev->kobj, &ktype_cdev_default);
    cdev->ops = fops;       /*指向文件操作结构实例*/
}
```

动态分配 cdev 实例的函数定义如下（/fs/char_dev.c）：

```
struct cdev *cdev_alloc(void)       {
    struct cdev *p = kzalloc(sizeof(struct cdev), GFP_KERNEL);   /*从通用缓存中分配*/
    if (p) {
        INIT_LIST_HEAD(&p->list);
        kobject_init(&p->kobj, &ktype_cdev_dynamic);       /*初始化 cdev->kobj*/
    }
    return p;
}
```

动态创建的 cdev 实例，需要手动对 cdev->ops 成员赋值（file_operations 实例指针）。

初始化完成的 cdev 实例即可调用 **cdev_add()**函数添加到字符设备驱动数据库，成功返回 0，否则返回错误码，函数代码如下（/fs/char_dev.c）：

```
int cdev_add(struct cdev *p, dev_t dev, unsigned count)
/*p：cdev 实例指针，dev：设备号（含起始从设备号），count：从设备数量*/
{
    int error;
    p->dev = dev;                   /*设备号赋予 cdev 实例*/
    p->count = count;               /*从设备数量赋予 cdev 实例*/
    error = kobj_map(cdev_map, dev, count, NULL,exact_match, exact_lock, p);
                                                    /*将 cdev 实例添加到字符设备驱动数据库*/
    if (error)
        return error;
    kobject_get(p->kobj.parent);        /*增加父节点引用计数*/

    return 0;
}
```

参数 p 为 cdev 实例指针，参数 dev、count 分别表示设备号（含起始从设备号）和从设备号数量，函数内调用 kobject_map()函数（见第 8 章），用于将 cdev 实例添加到字符设备驱动数据库中。

kobject_map()函数中参数 exact_match 函数指针赋予新创建 probe 实例的 **get()**成员，用于获取 cdev 实例中 kobject 结构体成员指针。exact_match()函数定义在/fs/char_dev.c 文件内：

```
static struct kobject *exact_match(dev_t dev, int *part, void *data)
{
    struct cdev *p = data;
    return &p->kobj;        /*直接返回 cdev 实例 kobj 成员指针*/
}
```

内核在早期版本中还定义了注册字符设备驱动的接口函数 **register_chrdev()**，函数内同时完成设备号

的申请，以及 cdev 实例的创建和添加。为兼容尚未更新到新接口的驱动程序，内核仍然保留了该接口，此函数定义在/include/linux/fs.h 头文件内：

```
static inline int register_chrdev(unsigned int major, const char *name, const struct file_operations *fops)
{
    return   __register_chrdev(major, 0, 256, name, fops);   /*/fs/char_dev.c*/
}
```

参数 major 为主设备号，如果 major 为 0 则动态分配主设备号，不为 0 则申请指定主设备号。函数内部调用的__register_chrdev()函数，首先向内核申请主设备号为 major，从设备号为 0~255 的设备号区间，然后动态创建 cdev 实例并初始化（fops 赋予 cdev->ops 成员），最后调用 cdev_add()函数将 cdev 实例添加到字符设备驱动数据库。

在添加 cdev 实例的过程中，cdev.kobj 并未导出到 sysfs 文件系统，因此 cdev 实例对用户进程是不可见的，函数源代码请读者自行阅读。

至此，字符设备驱动程序框架已经介绍完了，字符设备驱动程序的流程简述如下：

●申请设备号：向内核申请指定设备号或动态分配（主）设备号。

●实现字符设备文件操作结构 file_operations 实例。

●创建并初始化 cdev 实例：（1）静态创建实例，并调用 cdev_init(cdev,fops)函数对其初始化，（2）动态创建实例，手动对其 cdev->ops 成员赋值。

●添加 cdev 实例：调用 cdev_add()函数将 cdev 实例添加到内核字符设备驱动数据库。

在通用驱动模型中，设备由 xxx_device 实例表示，内嵌表示设备的 device 实例成员。在添加 xxx_device 实例时，将为设备创建设备文件（主设备号不为 0），并查找匹配的驱动 xxx_driver 实例，在此实例的 probe()函数中实现上面介绍的驱动程序流程。

9.1.3 字符设备操作

用户进程通过设备文件操作设备，操作设备的接口与普通文件的操作接口相同，即 file_operations 结构体实例，各文件操作系统调用最终调用此结构体中的函数实现。

字符设备驱动程序中需要为设备定义 file_operations 实例并赋予 cdev 实例。下面列出 file_operations 结构体中与字符设备操作相关的主要函数指针成员：

```
struct file_operations {
    ...
    ssize_t    (*read) (struct file *, char __user *, size_t, loff_t *);              /*读操作*/
    ssize_t    (*write) (struct file *, const char __user *, size_t, loff_t *);      /*写操作*/
    ...
    unsigned int    (*poll) (struct file *, struct poll_table_struct *);           /*查询设备状态*/
    long    (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);      /*向设备发送控制命令*/
    ...
    int    (*open) (struct inode *, struct file *);          /*打开设备*/
    int    (*release) (struct inode *, struct file *);      /*与 open()对应的函数*/
    ...
};
```

file_operations 结构体主要函数指针成员简介如下：

●**open**：打开设备文件时调用此函数。

●**read**：读操作函数，从设备中获取数据，调用 copy_to_user()等函数将数据复制到用户空间。

●**write**：写操作函数，调用 copy_from_user()等函数从用户空间复制数据至内核并写入设备。

●**unlocked_ioctl**：对设备写入命令，实现对设备的控制。

●**poll**：如果设备被设置成非阻塞式操作，应用程序在操作前可使用 select()和 poll()等系统调用查询设备当前状态，以便确定是否可非阻塞地访问设备。这几个系统调用内将调用 poll()函数查询设备状态。

由于设备的读写操作与具体设备密切相关，并且前面第 7 章已经介绍了读写文件的系统调用，因此这里不再介绍 read()和 write()函数。下面主要介绍一下字符设备文件的打开、设备控制等函数的实现。

1 打开设备

内核在打开字符设备文件时，将在虚拟文件系统中为其创建 inode 实例，并在打开操作过程中对其成员进行赋值（从具体文件系统中 inode 提取信息填充）。

inode 结构体中与字符设备相关的成员如下：

```
struct inode {
    umode_t        i_mode;          /*文件访问模式*/
    ...
    dev_t          i_rdev;          /*设备号*/
    ...
    const struct file_operations    *i_fop;   /*文件操作结构指针  */
    ...
    union {
        struct pipe_inode_info   *i_pipe;
        struct block_device   *i_bdev;       /*指向块设备数据结构*/
        struct cdev             *i_cdev;     /*指向 cdev 实例*/
        char                    *i_link;
    };
    ...
    struct list_head            i_devices;   /*将 inode 添到 cdev.list 为表头的双链表*/
    ...
};
```

内核在打开文件 open()系统调用中，根据文件系统 inode 中 i_mode 成员判断文件的类型，若为特殊文件（含设备文件），将调用 **init_special_inode()**函数对 inode 实例进行初始化，此函数定义在/fs/inode.c 文件内，代码简列如下：

```
void init_special_inode(struct inode *inode, umode_t mode, dev_t rdev)
{
    inode->i_mode = mode;
    if (S_ISCHR(mode)) {          /*是否是字符设备文件*/
        inode->i_fop = &def_chr_fops;        /*字符设备文件操作结构实例*/
        inode->i_rdev = rdev;                /*写入设备号*/
    } else if (S_ISBLK(mode)) {      /*块设备文件*/
        inode->i_fop = &def_blk_fops;        /*块设备文件操作结构*/
        inode->i_rdev = rdev;                /*写入设备号*/
```

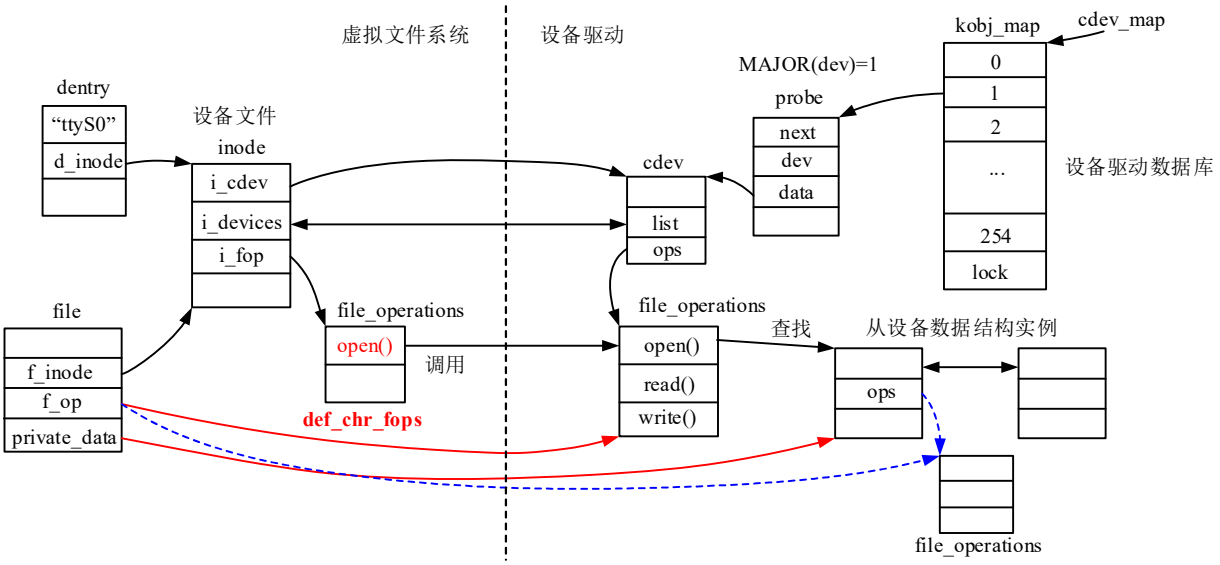
```
    } else if (S_ISFIFO(mode))          /*管道文件*/
        inode->i_fop = &pipefifo_fops;
    else if (S_ISSOCK(mode))
        ;
    else
        ...
}
```

字符设备文件 inode 实例文件操作结构指针 i_fop 指向 **def_chr_fops** 实例，实例定义在/fs/char_dev.c 文件内（由内核静态定义，适用于所有字符设备文件）：

```
const struct file_operations def_chr_fops = {
    .open = chrdev_open,            /*字符设备文件打开函数*/
    .llseek = noop_llseek,
};
```

内核在打开文件 open()系统调用中随后还将调用 **def_chr_fops** 实例中的 open()函数，即 chrdev_open()函数。

先看下图了解一下 chrdev_open()函数执行的主要工作，函数将根据 inode 中保存的设备号，查找字符设备驱动数据库，找到对应的 cdev 实例，然后将 cdev-> ops 指向的 file_operations 实例赋予打开设备文件的 file 实例（file->f_op），最后调用 cdev->ops->open()函数完成特定字符设备的打开操作（激活硬件、查找私有数据结构等）。



cdev 实例可能适用于一类设备，即有多个从设备，从设备又可以由自己的私有数据结构，以及文件操作 file_operations 实例。在 cdev 实例关联 file_operations 实例的 open()函数中可以根据从设备号，查找从设备的私有数据结构实例，并赋予 file 实例，还可以用私有的 file_operations 实例替换原 file 实例关联的文件操作结构实例，也就是使用特定于从设备的 file_operations 实例。

chrdev_open()函数定义如下（/fs/char_dev.c）：

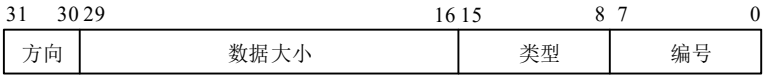
```
static int chrdev_open(struct inode *inode, struct file *filp)
{
```

```
const struct file_operations *fops;
struct cdev *p;
struct cdev *new = NULL;
int ret = 0;

spin_lock(&cdev_lock);
p = inode->i_cdev;              /*cdev 实例*/
if (!p) {                       /*若 inode->i_cdev 为 NULL，表示首次字符设备文件打开*/
    struct kobject *kobj;
    int idx;
    spin_unlock(&cdev_lock);
    kobj = kobj_lookup(cdev_map, inode->i_rdev, &idx);   /*在驱动数据库中查找 cdev 实例*/
    if (!kobj)
        return -ENXIO;
    new = container_of(kobj, struct cdev, kobj);      /*由 kobject 获取 cdev 实例指针*/
    spin_lock(&cdev_lock);
    p = inode->i_cdev;                  /*再次判断 inode->i_cdev 是否为空*/
    if (!p) {                           /*若 inode->i_cdev 为空则对其赋值*/
        inode->i_cdev = p = new;                    /*inode->i_cdev 指向 cdev 实例*/
        list_add(&inode->i_devices, &p->list);     /*inode 添加到 cdev.list 双链表头部*/
        new = NULL;
    } else if (!cdev_get(p))        /*增加 cdev 引用计数*/
        ret = -ENXIO;
} else if (!cdev_get(p))
    ret = -ENXIO;
spin_unlock(&cdev_lock);
cdev_put(new);
if (ret)
    return ret;

ret = -ENXIO;
fops = fops_get(p->ops);     /*fops=cdev->ops，cdev 定义的文件操作结构实例*/
if (!fops)
    goto out_cdev_put;

replace_fops(filp, fops);        /*file->i_fop=cdev->ops，/include/linux/fs.h*/
if (filp->f_op->open) {
    ret = filp->f_op->open(inode, filp);        /*调用 cdev->ops->open()函数*/
    if (ret)
        goto out_cdev_put;
}
return 0;
...
```

}

2 设备控制

用户进程可通过系统调用 ioctl()向设备发送命令对设备进行控制，命令的格式如下：



命令由 32 位无符号整数表示，低 16 位表示命令（高 8 位表示命令类型，低 8 位表示命令编号），命令类型通常用英文字符“A”~“Z”或“a”~“z”表示。高 16 位中的低 14 位表示参数数据大小（见下文），最高 2 位用于区别读写操作（1 表示写，2 表示读），详见/include/asm-generic/ioctl.h 头文件。

内核在/include/asm-generic/ioctl.h 头文件定义了合成命令编码的宏，例如：

```
#define _IOC(dir,type,nr,size) \            /*合成命令编码*/
        (((dir)  << _IOC_DIRSHIFT) | \      /*读写方向*/
        ((type) << _IOC_TYPESHIFT) | \      /*命令类型*/
        ((nr)    << _IOC_NRSHIFT) | \       /*命令编号*/
        ((size) << _IOC_SIZESHIFT))         /*参数数据大小*/

#define _IO(type,nr)            _IOC(_IOC_NONE,(type),(nr),0)     /*不带读写方向（0）*/
#define _IOR(type,nr,size)      _IOC(_IOC_READ,(type),(nr),(_IOC_TYPECHECK(size)))   /*读命令*/
#define _IOW(type,nr,size)      _IOC(_IOC_WRITE,(type),(nr),(_IOC_TYPECHECK(size)))   /*写命令*/
#define _IOWR(type,nr,size) _IOC(_IOC_READ|_IOC_WRITE,(type),(nr),(_IOC_TYPECHECK(size)))
...
```

另外，头文件中还定义了解析命令编码的宏，例如：

```
#define _IOC_DIR(nr)        (((nr) >> _IOC_DIRSHIFT) & _IOC_DIRMASK)
#define _IOC_TYPE(nr)       (((nr) >> _IOC_TYPESHIFT) & _IOC_TYPEMASK)
#define _IOC_NR(nr)         (((nr) >> _IOC_NRSHIFT) & _IOC_NRMASK)
#define _IOC_SIZE(nr)       (((nr) >> _IOC_SIZESHIFT) & _IOC_SIZEMASK)
```

设备命令不能随意编码，有些值内核已经使用了，不能与之重复，例如（/include/uapi/linux/fs.h）：

```
#define FIBMAP          _IO(0x00,1)     /* bmap access */
#define FIGETBSZ        _IO(0x00,2)     /* get the block size used for bmap */
#define FIFREEZE        _IOWR('X', 119, int)/* Freeze */
#define FITHAW          _IOWR('X', 120, int)/* Thaw */
#define FITRIM          _IOWR('X', 121, struct fstrim_range)     /* Trim */
...
```

内核在**/Documentation/ioctl/ioctl-number.txt** 文件内，标识了各种设备使用的命令编码方式，各设备支持的命令在设备相关的头文件中定义。

ioctl()系统调用在/fs/ioctl.c 文件内实现，代码如下：

```
SYSCALL_DEFINE3(ioctl, unsigned int, fd, unsigned int, cmd, unsigned long, arg)
/*cmd：命令参数，arg：通常是某一数据结构指针，命令编码中包含此数据结构大小*/
{
    int error;
```

```
    struct fd f = fdget(fd);

    if (!f.file)
        return -EBADF;
    error = security_file_ioctl(f.file, cmd, arg);       /*安全性检查*/
    if (!error)
        error = do_vfs_ioctl(f.file, fd, cmd, arg);    /*/fs/ioctl.c*/
    fdput(f);
    return error;
}
```

cmd 参数表示命令编码，arg 参数通常是某一数据结构实例的指针（地址），系统调用中从中读出/写入数据。ioctl()系统调用中调用 **do_vfs_ioctl()**函数执行命令，此函数充当一个命令分发器的作用，对不同的命令调用不同的执行函数，函数定义如下。

```
int do_vfs_ioctl(struct file *filp, unsigned int fd, unsigned int cmd,unsigned long arg)
{
    int error = 0;
    int __user *argp = (int __user *)arg;
    struct inode *inode = file_inode(filp);

    switch (cmd) {
    case FIOCLEX:
        set_close_on_exec(fd, 1);
        break;

    case FIONCLEX:
        set_close_on_exec(fd, 0);
        break;

    case FIONBIO:
        error = ioctl_fionbio(filp, argp);
        break;

    case FIOASYNC:
        error = ioctl_fioasync(fd, filp, argp);
        break;

    case FIOQSIZE:
        if (S_ISDIR(inode->i_mode) || S_ISREG(inode->i_mode) ||S_ISLNK(inode->i_mode)) {
            loff_t res = inode_get_bytes(inode);
            error = copy_to_user(argp, &res, sizeof(res)) ?-EFAULT : 0;
        } else
            error = -ENOTTY;
```

```
            break;

        case FIFREEZE:
            error = ioctl_fsfreeze(filp);
            break;

        case FITHAW:
            error = ioctl_fsthaw(filp);
            break;

        case FS_IOC_FIEMAP:
            return ioctl_fiemap(filp, arg);

        case FIGETBSZ:
            return put_user(inode->i_sb->s_blocksize, argp);

        default:
            if (S_ISREG(inode->i_mode))
                error = file_ioctl(filp, cmd, arg);     /*普通文件*/
            else
                error = vfs_ioctl(filp, cmd, arg);      /*设备文件，/fs/ioctl.c*/
            break;
        }
        return error;
    }
```

do_vfs_ioctl()函数内首先判断是否是内核定义的命令，调用相应的处理函数。如果不是内核定义的命令，对于普通文件调用 file_ioctl()函数处理命令，对于设备文件调用 **vfs_ioctl()**函数处理命令。

vfs_ioctl()函数调用设备文件操作结构 filp->f_op->**unlocked_ioctl**(filp, cmd, arg)函数处理命令。

不过，内核现在似乎更倾向于以设备属性通过 sysfs 文件系统中的读写属性文件来对设备进行控制。

9.2 GPIO 驱动

GPIO 是芯片中通用的输入输出端口（引脚），每个端口通过配置可设置为输出或输入端口（通常还复用其它功能）。芯片通常通过配置寄存器对 GPIO 进行配置和操作，寄存器中一个比特位对应一个 GPIO。

Linux 内核中对 GPIO 的配置寄存器进行了抽象，称它为 GPIO 控制器，用于实现对 GPIO 设置和操作。连接在 GPIO 上的设备，其驱动程序可通过 GPIO 控制器提供的接口函数操作 GPIO。

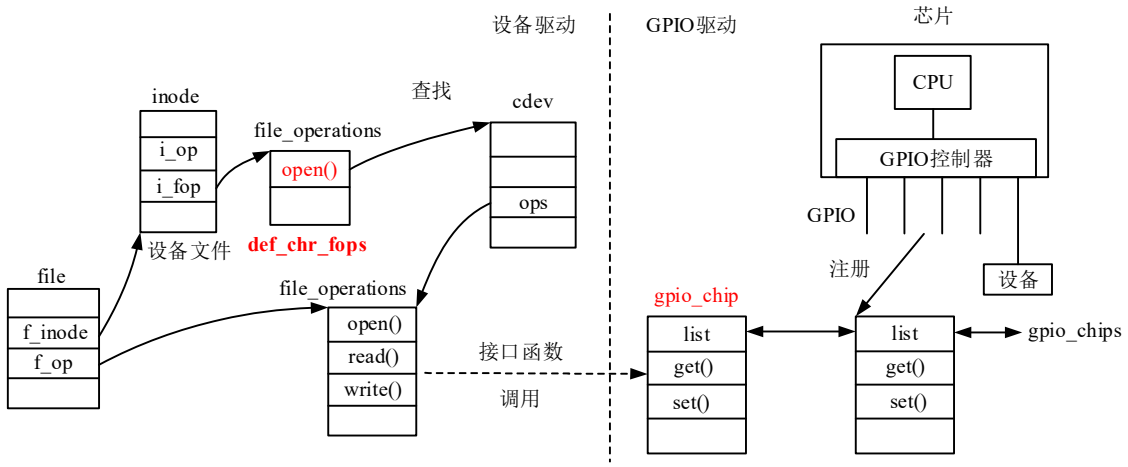
如果需要使用户能直接操作 GPIO，也可将一个或多个 GPIO 视为设备，为其创建字符设备驱动程序，从而用户可通过设备文件操作 GPIO。

GPIO 驱动相关代码位于/drivers/gpio/目录下。

9.2.1 概述

GPIO 驱动框架如下图所示，GPIO 控制器由 gpio_chip 结构体表示，结构体中包含其所管理 GPIO 的

整体信息以及操作函数指针，例如：设置引脚、读引脚值、写引脚值等。



GPIO 控制器驱动需要定义并注册 gpio_chip 实例，实例由全局双链表管理。GPIO 驱动提供了设置引脚、读写引脚的接口函数。

连接在 GPIO 上的设备，其驱动 file_operations 实例中的函数调用 GPIO 驱动提供的接口函数，实现对引脚的控制和数据传输。也可以直接将一个或多个 GPIO 当成设备，为其创建设备驱动程序，从而使用户可以直接操作 GPIO。

9.2.2 GPIO 控制器

芯片中的 GPIO 控制器由 gpio_chip 结构体表示。平台代码需要定义 gpio_chip 实例，并在初始化函数中注册实例。GPIO 驱动通用层提供了操作 GPIO 的接口函数，接口函数将调用 gpio_chip 实例中的对应函数。

1 数据结构

gpio_chip 结构体定义在/include/linux/gpio/driver.h 头文件内：

```
struct gpio_chip {
    const char          *label;         /*控制器名称*/
    struct device       *dev;           /**/
    struct device       *cdev;          /*设备类下表示控制器的 device 实例，用于 sysfs 接口*/
    struct module       *owner;         /*模块指针*/
    struct list_head            list;           /*双链表成员，用于将实例链入全局双链表*/

    int         (*request)(struct gpio_chip *chip,unsigned offset);   /*申请端口*/
    void        (*free)(struct gpio_chip *chip,unsigned offset);      /*释放端口*/
    int         (*get_direction)(struct gpio_chip *chip,unsigned offset); /*获取端口模式，0 输出，1 输入*/
    int         (*direction_input)(struct gpio_chip *chip,unsigned offset);     /*将端口设为输入*/
    int         (*direction_output)(struct gpio_chip *chip,unsigned offset, int value); /*将端口设为输出*/
    int         (*get)(struct gpio_chip *chip,unsigned offset);     /*获取端口值*/
    void        (*set)(struct gpio_chip *chip,unsigned offset, int value); /*设置端口值*/
    void        (*set_multiple)(struct gpio_chip *chip,unsigned long *mask,unsigned long *bits);
    int         (*set_debounce)(struct gpio_chip *chip,unsigned offset,unsigned debounce);
    int         (*to_irq)(struct gpio_chip *chip,unsigned offset);     /*GPIO 对应的中断号*/
```

```
    void        (*dbg_show)(struct seq_file *s,struct gpio_chip *chip);
    int             base;     /*本控制器控制的起始端口号*/
    u16             ngpio;    /*本控制器控制的端口数量*/
    struct gpio_desc    *desc;      /*指向 gpio_desc 数组，用于描述端口*/
    const char *const   *names;
    bool            can_sleep;
    bool            irq_not_threaded;

#ifdef CONFIG_GPIOLIB_IRQCHIP
    struct irq_chip     *irqchip;
    struct irq_domain   *irqdomain;
    unsigned int        irq_base;
    irq_flow_handler_t irq_handler;
    unsigned int        irq_default_type;
    int             irq_parent;
#endif

#if defined(CONFIG_OF_GPIO)         /*支持设备树*/
    struct device_node *of_node;
    int of_gpio_n_cells;
    int (*of_xlate)(struct gpio_chip *gc,const struct of_phandle_args *gpiospec, u32 *flags);
#endif
#ifdef CONFIG_PINCTRL
    struct list_head pin_ranges;
#endif
};
```

gpio_chip 结构体中主要成员简介如下：

●**base**：本控制器控制引脚的起始编号，结构体中函数内参数 offset 为引脚编号相对于 base 的偏移量，实际的端口号为 base+offset。

●**ngpio**：本控制器控制引脚的数量。

●**direction_input**：将 offset 引脚设为输入端口。

●**direction_output**：将 offset 引脚设为输出端口。

●**get**：获取 offset 引脚值。

●**set**：设置 offset 引脚值。

●**desc**：指向 gpio_desc 结构体数组，每个数组项对应一个端口，用于描述端口属性。gpio_desc 结构体定义如下（/drivers/gpio/gpiolib.h）：

```
struct gpio_desc {
    struct gpio_chip    *chip;     /*指向 GPIO 控制器*/
    unsigned long       flags;      /*标记*/
    const char          *label;
};
```

标记成员取值定义如下，主要标记端口的硬件属性：

```
#define    FLAG_REQUESTED    0
```

```
#define   FLAG_IS_OUT          1
#define   FLAG_EXPORT          2     /* protected by sysfs_lock */
#define   FLAG_SYSFS           3     /* exported via /sys/class/gpio/control */
#define   FLAG_ACTIVE_LOW      6     /* value has active low */
#define   FLAG_OPEN_DRAIN      7     /* Gpio is open drain type */
#define   FLAG_OPEN_SOURCE     8     /* Gpio is open source type */
#define   FLAG_USED_AS_IRQ     9     /* GPIO 连接到中断*/
#define   FLAG_IS_HOGGED       11    /* GPIO is hogged */
```

内核在/drivers/gpio/gpiolib.c 文件内定义了全局双链表头，用于管理 gpio_chip 实例：

```
LIST_HEAD(gpio_chips);
```

2 添加控制器

在板级相关或 GPIO 驱动程序中需要定义 gpio_chip 结构体实例，并添加到内核，**gpiochip_add()**为添加函数。

gpio_chip 结构体中的函数指针主要是对芯片配置寄存器的操作，都比较简单，且特定于芯片，这里就不讲解了。

添加 gpio_chip 实例的 gpiochip_add(struct gpio_chip *chip)函数定义如下（/drivers/gpio/gpiolib.c）：

```
int gpiochip_add(struct gpio_chip *chip)
{
    unsigned long  flags;
    int            status = 0;
    unsigned  id;
    int            base = chip->base;
    struct gpio_desc *descs;

    descs = kcalloc(chip->ngpio, sizeof(descs[0]), GFP_KERNEL);    /*为 gpio_desc 数组分配空间*/
    ...
    spin_lock_irqsave(&gpio_lock, flags);

    if (base < 0) {             /*起始编号小于 0*/
        base = gpiochip_find_base(chip->ngpio);
        if (base < 0) {
            status = base;
            spin_unlock_irqrestore(&gpio_lock, flags);
            goto err_free_descs;
        }
        chip->base = base;
    }

    status = gpiochip_add_to_list(chip);    /*将实例插入双链表合适位置，以起始端口编号排序*/
     ...
    for (id = 0; id < chip->ngpio; id++) {       /*初始化 gpio_desc 数组*/
```

```
            struct gpio_desc *desc = &descs[id];
            desc->chip = chip;
            desc->flags = !chip->direction_input ? (1 << FLAG_IS_OUT) : 0;
        }
        chip->desc = descs;
        spin_unlock_irqrestore(&gpio_lock, flags);

    #ifdef CONFIG_PINCTRL
        INIT_LIST_HEAD(&chip->pin_ranges);
    #endif

        of_gpiochip_add(chip);     /*添加设备树中定义的 GPIO 端口*/
        acpi_gpiochip_add(chip);

        status = gpiochip_sysfs_register(chip);          /*/drivers/gpio/gpiolib-sysfs.c*/
                    /*在 gpio_class 设备类对应目录下创建控制器对应的目录，并添加属性文件*/
        ...
        return 0;       /*成功返回 0*/
         ...
    }
```

gpiochip_add()函数比容易理解主要工作是为端口创建 gpio_desc 数组并初始化，将 gpio_chip 添加到全局双链表合适位置，实例在双链表中以起始端口编号大小排序。

另外，gpiochip_sysfs_register(chip)函数用于将控制器和引脚信息导出到 sysfs 文件系统，

3 接口函数

内核在/include/asm-generic/gpio.h 或/include/linux/gpio.h 头文件内定义了控制、读写 GPIO 端口的接口函数，各函数内部调用/drivers/gpio/gpiolib.c 或 gpiolib-legacy.c 内的函数实现，并最终调用控制器内定义的函数实现端口设置和操作，部分接口函数如下所示：

- int **gpio_request**(unsigned gpio, const char *label)：申请 GPIO 端口，成功返回 0，否则返回错误码；
- int gpio_request_one(unsigned gpio, unsigned long flags, const char *label)：申请 GPIO 端口，带标记；
- int **gpio_direction_input**(unsigned gpio)：将端口设为输入端口，成功返回 0，否则返回错误码；
- int **gpio_get_value**(unsigned int gpio)：获取端口值；
- int **gpio_direction_output**(unsigned gpio, int value)：设置端口为输出，初始值为 value；
- void **gpio_set_value**(unsigned int gpio, int value)：设置端口值；
- int **gpio_to_irq**(unsigned int gpio)：返回 GIPO 对应 IRQ（中断）编号。

9.2.3 GPIO 驱动示例

龙芯 1B 具有 61 个 GPIO 端口，处理器设置了两个配置寄存器用于设置端口功能（用于 GPIO 端口还是复用功能），两个输入使能寄存器用于设置 GPIO 端口用于输入或输出，两个输入寄存器用于读取输入 GPIO 端口值，两个输出寄存器用于输出端口值。龙芯 1B 处理器 GPIO 端口还可用于触发中断。

1 控制器实例

龙芯 1B 具有 61 个 GPIO 端口，编为两组，编号为 0-30 和 32-61，芯片通过两组寄存器来控制 GPIO 端口，每个端口对应寄存器中的一位，寄存器定义如下：

GPIOCFG0/GPIOCFG1：配置寄存器，置 1 表示端口为 GPIO，为 0 表示用为其它功能；

GPIOOE0/GPIOOE1：输入使能寄存器，当端口配置为 GPIO 功能时，置 1 表示 GPIO 端口为输入，0 表示端口为输出；

GPIOIN0/GPIOIN1：输入寄存器，保存端口输入值；

GPIOOUT0/GPIOOUT0：输出寄存器，写端口实现对端口输出值的控制。

龙芯 1B 开发板提供的源代码在板级相关文件 gpio.c 文件内定义了 **gpio_chip** 结构体实例：

```
static struct gpio_chip ls1x_chip[] = {          /*gpio_chip 实例数组*/
    [0] = {                                      /*控制第 1 组 GPIO*/
        .label                  = "ls1x-gpio0",
        .direction_input        = ls1x_gpio0_direction_input,    /*配置端口为输入*/
        .direction_output       = ls1x_gpio0_direction_output,   /*配置端口为输出*/
        .get                    = ls1x_gpio0_get_value,   /*读取端口值*/
        .set                    = ls1x_gpio0_set_value,    /*设置端口值*/
        .free                   = ls1x_gpio0_free,
        .to_irq                 = ls1x_gpio0_to_irq,      /*中断编号*/
        .base                   = 0,    /*起始 GPIO 编号*/
        .ngpio                  = 32,   /*GPIO 数量*/
    },
    [1] = {                              /*控制第 2 组 GPIO*/
        .label                  = "ls1x-gpio1",
        .direction_input        = ls1x_gpio1_direction_input,
        .direction_output       = ls1x_gpio1_direction_output,
        .get                    = ls1x_gpio1_get_value,
        .set                    = ls1x_gpio1_set_value,
        .free                   = ls1x_gpio1_free,
        .to_irq                 = ls1x_gpio1_to_irq,
        .base                   = 32,           /*起始 GPIO 编号*/
        .ngpio                  = 32,           /*GPIO 数量*/
    },
    ...
}
```

gpio_chip 实例中的函数都是简单的对芯片配置寄存器的读写，源代码请读者自行阅读。

在同一文件内定义了 GPIO 初始化函数，在内核启动阶段后期调用，如下所示：

```
static int __init ls1x_gpio_setup(void)
{
    gpiochip_add(&ls1x_chip[0]);    /*添加（注册）GPIO 控制器*/
    gpiochip_add(&ls1x_chip[1]);
  #if defined(CONFIG_LS1A_MACH) || defined(CONFIG_LS1C_MACH)
    gpiochip_add(&ls1x_chip[2]);
```

```
    #endif
    #if defined(CONFIG_LS1C_MACH)
        gpiochip_add(&ls1x_chip[3]);
    #endif
        return 0;
}
arch_initcall(ls1x_gpio_setup);
```

初始化函数只是向内核添加 gpio_chip 实例，在设备驱动程序中可调用 GPIO 通用接口函数申请、设置、读写 GPIO 端口。

2 GPIO 设备驱动

在开发板提供的源代码/drivers/gpio/gpio-ls1x.c 文件内，注册了一个 MISC 设备，此设备将 4 个 GPIO 引脚视为一个设备，为其创建设备驱动程序，以使用户进程可直接读取 GPIO 引脚值。

MISC 设备 miscdevice 实例（详见下节）定义如下：

```
static struct miscdevice gpio_ls1x_miscdev = {
    .minor = GPIO_LS1X_INPUT_MINOR,
    .name = "gpio_ls1x",
    .fops = &gpio_ls1x_fops     /*设备文件操作结构实例*/
};
```

初始化函数中注册了 miscdevice 实例，函数代码如下：

```
static void gpio_input_init(void)       /*申请并配置引脚*/
{
    gpio_request(LINE_DI_INPUT0, "gpio_inptu");
    gpio_request(LINE_DI_INPUT1, "gpio_inptu");
    gpio_request(LINE_DI_INPUT2, "gpio_inptu");
    gpio_request(LINE_DI_INPUT3, "gpio_inptu");

    gpio_direction_input(LINE_DI_INPUT0);
    gpio_direction_input(LINE_DI_INPUT1);
    gpio_direction_input(LINE_DI_INPUT2);
    gpio_direction_input(LINE_DI_INPUT3);
}

static int __init gpio_ls1x_init(void)
{
    gpio_input_init();      /*配置引脚为输入端*/
    return misc_register(&gpio_ls1x_miscdev);      /*注册 miscdevice 实例*/
}
module_init(gpio_ls1x_init);
```

MISC 设备文件操作结构实例定义如下：

```
static const struct file_operations gpio_ls1x_fops = {
```

```
    .owner          = THIS_MODULE,
    .open           = gpio_ls1x_open,
    .read           = gpio_ls1x_read,       /*直接对寄存器读*/
    .write          = gpio_ls1x_write,      /*直接对寄存器写*/
    .unlocked_ioctl     = gpio_ls1x_unlocked_ioctl,
    .llseek         = no_llseek,
};
```

读/写文件内容函数直接就是对配置寄存器的读写，源代码请读者自行阅读。

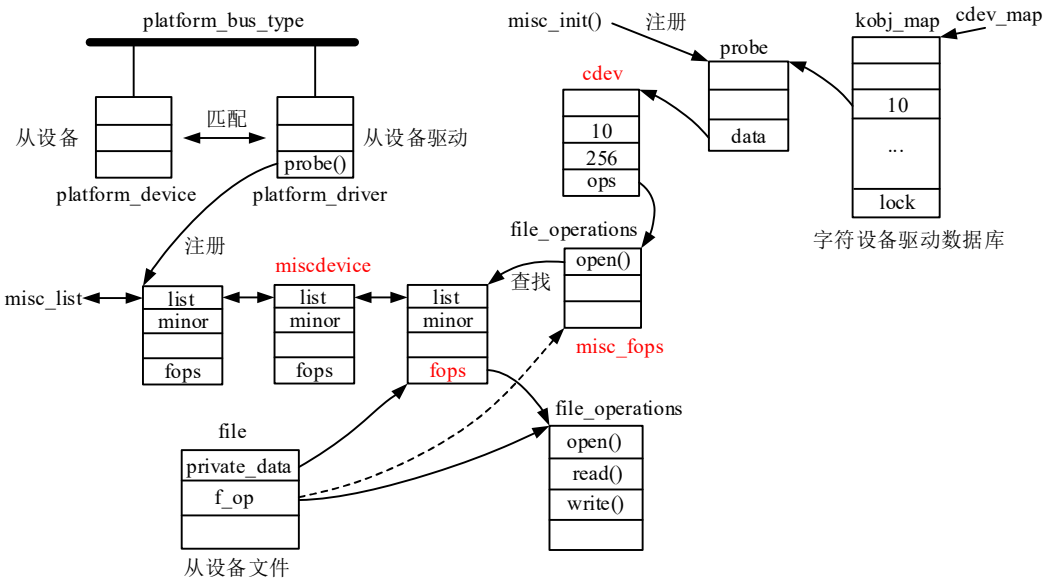
9.3 MISC 设备

MISC 设备表示一些“杂项”的字符设备，MISC 设备主设备号为 10（MISC_MAJOR），从设备号为 0~255，从设备号定义在/include/linux/miscdevice.h 头文件。misc 驱动框架源代码位于/drivers/char/misc.c 文件内，默认编译进内核。

9.3.1 驱动框架

MISC 设备驱动框架如下图所示，内核在启动阶段将注册表示 MISC 设备的 cdev 实例，主设备号为 10，从设备号数量为 256（前 64 个从设备号用于动态分配），也就是说所有的 MISC 设备驱动共用一个 cdev 实例，其文件操作结构实例为 **misc_fops**。每个具体的 MISC 从设备由 **miscdevice** 结构体实例表示，所有实例由双链表管理。

miscdevice 结构体中包含特定于从设备的文件操作结构 file_operations 实例。在打开 MISC 设备文件时，将调用公用的 misc_fops 实例的 open()函数，此函数内将根据从设备号查找 miscdevice 实例双链表，找到从设备号对应的 miscdevice 实例，并将实例中关联的文件操作结构实例指针赋予设备文件 file 实例，从而实现文件操作从通用到特定于从设备的转换，对设备文件的操作将调用特定于从设备的文件操作结构中函数实现。



MISC 设备驱动只需要实现特定于从设备的 file_operations 实例，在驱动的 probe()函数中创建表示 MISC 设备的 miscdevice 实例并关联定义的 file_operations 实例，最后调用通用的注册 miscdevice 实例的函数注册 miscdevice 实例即可。

9.3.2 初始化

MISC 设备驱动框架初始化函数 **misc_init()**定义在/drivers/char/misc.c 文件内，在内核启动后期自动调用执行：

```
static int __init misc_init(void)
{
    int err;

#ifdef CONFIG_PROC_FS
    proc_create("misc", 0, NULL, &misc_proc_fops);
#endif
    misc_class = class_create(THIS_MODULE, "misc");      /*创建设备类*/
    ...
    if (register_chrdev(MISC_MAJOR,"misc",&misc_fops))    /*注册 misc 字符设备驱动*/
        goto fail_printk;
    misc_class->devnode = misc_devnode;   /*设置设备文件名称的函数*/
    return 0;
    ...
}
subsys_initcall(misc_init);          /*内核启动阶段调用*/
```

初始化函数内主要完成两项工作：一是创建 MISC 设备类，并对 misc_class->devnode 函数指针赋值，此函数用于确定设备文件的名称，二是创建并注册 MISC 设备 cdev 实例。

1 创建设备类

内核为 MISC 设备创建了设备类实例，由全局指针 misc_class 管理，实例赋予的 **misc_devnode()**函数，用于在注册 miscdevice 实例自动创建从设备文件时确定设备文件名称，函数定义如下。

```
static char *misc_devnode(struct device *dev, umode_t *mode)
{
    struct miscdevice *c = dev_get_drvdata(dev);

    if (mode && c->mode)
        *mode = c->mode;
    if (c->nodename)
        return kstrdup(c->nodename, GFP_KERNEL);
    return NULL;
}
```

函数内判断 miscdevice 实例 **nodename** 成员是否为 NULL，不为 NULL，则将其作为设备文件名称，否则返回 NULL。如果返回 NULL，将会使用 **misc->name** 作为设备文件名称。

2 文件操作

初始化函数内调用 register_chrdev()函数注册主设备号为 10，从设备号为 0~255 的字符设备驱动，创

建并注册对应的 cdev 实例，且文件操作结构指针赋值为 **misc_fops**。

misc_fops 实例定义如下：

```
static const struct file_operations misc_fops = {
        .owner   = THIS_MODULE,
        .open    = misc_open,          /*MISC 设备打开函数*/
        .llseek  = noop_llseek,
};
```

在打开 MISC 设备文件时，将会调用 misc_fops 实例中的 open()函数，函数定义如下：

```
static int misc_open(struct inode * inode, struct file * file)
{
    int minor = iminor(inode);      /*从设备号*/
    struct miscdevice *c;
    int err = -ENODEV;
    const struct file_operations *new_fops = NULL;

    mutex_lock(&misc_mtx);

    list_for_each_entry(c, &misc_list, list) { /*搜索 miscdevice 实例双链表，由从设备号查找匹配项*/
        if (c->minor == minor) {
            new_fops = fops_get(c->fops);       /*miscdevice 实例关联的 file_operations 实例*/
            break;
        }
    }

    if (!new_fops) {      /*没找到，加载模块后再查找*/
        mutex_unlock(&misc_mtx);
        request_module("char-major-%d-%d", MISC_MAJOR, minor);      /*加载模块*/
        mutex_lock(&misc_mtx);

        list_for_each_entry(c, &misc_list, list) {     /*再次查找*/
            if (c->minor == minor) {
                new_fops = fops_get(c->fops);
                break;
            }
        }
       if (!new_fops)
            goto fail;
    }
    file->private_data = c;    /*file->private_data =miscdevice*/

    err = 0;
    replace_fops(file, new_fops);     /*file->f_op=new_fops*/
```

```
    if (file->f_op->open)
        err = file->f_op->open(inode,file);    /*调用 miscdevice->fops->open()函数*/
fail:
    mutex_unlock(&misc_mtx);
    return err;
}
```

打开操作函数比较简单，通过从设备号查找 miscdevice 实例双链表，找到匹配项并将实例中关联的文件操作结构实例指针赋予打开文件 file 实例，并调用其中的 open()函数。从而对设备的操作由通用实现转为特定于从设备的实现。

9.3.3 注册 miscdevice

MISC 设备驱动框架中，每个从设备由 miscdevice 结构体表示，结构体定义在/include/linux/miscdevice.h 头文件内：

```
struct miscdevice   {
    int    minor;                    /*从设备号，MISC_DYNAMIC_MINOR 表示由内核分配从设备号*/
    const char    *name;                    /*设备名称*/
    const struct file_operations    *fops;    /*文件操作结构指针*/
    struct list_head    list;                /*双链表元素，添加到全局双链表*/
    struct device    *parent;                /*父设备*/
    struct device    *this_device;            /*表示本设备的 device 实例指针*/
    const struct attribute_group **groups;      /*属性组*/
    const char    *nodename;            /*注册 miscdevice 实例时用于创建设备文件的名称*/
    umode_t    mode;                    /*从设备访问权限控制*/
};
```

内核中所有 miscdevice 实例由全局双链表管理，双链表头 misc_list 定义在/drivers/char/misc.c 文件内：

```
static    LIST_HEAD(misc_list);
```

从设备驱动程序需创建 miscdevice 实例，并调用 misc_register()函数向内核注册，misc_register()函数定义在/drivers/char/misc.c 文件内：

```
int misc_register(struct miscdevice * misc)
{
    dev_t dev;
    int err = 0;
    bool is_dynamic = (misc->minor == MISC_DYNAMIC_MINOR);   /*动态分配从设备号*/

    INIT_LIST_HEAD(&misc->list);
    mutex_lock(&misc_mtx);
    if (is_dynamic) {          /*如果是动态分配从设备号，从位图中查找为 0 的位，分配从设备号*/
        int i = find_first_zero_bit(misc_minors, DYNAMIC_MINORS);     /*DYNAMIC_MINORS=64*/
        ...
        misc->minor = DYNAMIC_MINORS - i - 1;     /*从设备号*/
        set_bit(i, misc_minors);                /*位图相应位置位*/
    }
```

```
        else {              /*指定从设备号*/
            struct miscdevice *c;

            list_for_each_entry(c, &misc_list, list) {          /*查找从设备 miscdevice 实例是否已存在*/
                if (c->minor == misc->minor) {   /*从设备号不能已被使用*/
                    err = -EBUSY;
                    goto out;
                }
            }
        }

        dev = MKDEV(MISC_MAJOR, misc->minor);       /*合成设备号*/

        misc->this_device =device_create_with_groups(misc_class, misc->parent, dev,
                              misc, misc->groups, "%s", misc->name);
                                                    /*创建设备 device 实例，并自动创建设备文件*/
        ...
        list_add(&misc->list, &misc_list);          /*miscdevice 实例插入到全局双链表头部*/
    out:
        mutex_unlock(&misc_mtx);
        return err;         /*成功返回 0*/
    }
```

注册 miscdevice 实例的函数比较简单，需要说明一下的是从设备号的管理。从设备号 0-63 用于动态分配，当 miscdevice 实例指定的从设备号为 MISC_DYNAMIC_MINOR（255）时，表示从设备号由内核动态分配。内核在/drivers/char/misc.c 文件内创建了 64 位的位图用于动态分配从设备号，位图清零的位表示相应的从设备号没有被分配，置 1 的位表示相应从设备号已被使用。

9.3.4 红外驱动示例

MISC 设备驱动程序位于内核源码/drivers/char/或/drivers/misc/等目录下，下面以红外接收设备为例说明驱动程序的实现。

龙芯 1B 开发板外接了 IRM-3638T 红外接收头，红外接收头有三个引脚，分别是电源端、输出端、地端，输出端接到处理器 GPIO61 上。红外发射头发射信号时增加载频（如 38KHZ），接收头检波去除载频后将信号输出，无输出时端口保持高电平。红外接收信号在每个下降沿触发中断（GPIO 中断），中断内读取端口值，根据中断之间的时间间隔来确定发送的数据。驱动程序源代码位于/drivers/char/ls1b_ir.c 文件内。

1 注册驱动

驱动程序定义了红外设备驱动 platform_driver 实例，如下：

```
static struct platform_driver ls1b_ir_driver = {
    .probe = ls1b_ir_probe,          /*空函数，板级文件内不需要定义相应的 platform_device 实例*/
    .driver = {
        .name = "ls1b_ir",
```

```
    },
};
```

红外接收驱动程序采用了 MISC 设备驱动框架，定义的 miscdevice 实例 ls1b_ir_miscdev 如下：

```
static struct miscdevice ls1b_ir_miscdev = {
    MISC_DYNAMIC_MINOR,      /*动态分配从设备号*/
    "ls1b_ir",               /*设备文件名为/dev/ls1b_ir*/
    &ls1b_ir_ops,            /*文件操作结构实例，见下文*/
};
```

在初始化函数中完成 miscdevice 实例和 platform_driver 实例的注册，函数代码如下：

```
static int __init ls1b_ir_init(void)
{
    if (misc_register(&ls1b_ir_miscdev)) {        /*注册 miscdevice 实例*/
        printk(KERN_WARNING "IR: Couldn't register device!\n ");
        return -EBUSY;
    }

    return platform_driver_register(&ls1b_ir_driver);      /*注册 platform_driver 实例*/
}
__initcall(ls1b_ir_init);
```

2 文件操作

在定义 miscdevice 实例时，其文件操作结构实例赋值为 **ls1b_ir_ops**，定义如下：

```
static const struct file_operations ls1b_ir_ops = {
    .owner = THIS_MODULE,
    .open = ls1b_ir_open,          /*打开设备*/
    .release = ls1b_ir_close,
    .read = ls1b_ir_read,          /*读数据*/
};
```

下面看一下打开设备和读数据函数的实现。

■打开设备

打开设备的函数定义如下：

```
static int ls1b_ir_open(struct inode *inode, struct file *filep)
{
    int ret;
    ret = gpio_request(GPIO_IR, "ls1x_ir");        /*申请 GPIO61 引脚，GPIO 驱动见上一节*/
    if (ret < 0)
        return ret;
    gpio_direction_input(GPIO_IR);                 /*将引脚配置成输入端口*/
```

```
    ls1b_ir_irq = gpio_to_irq(GPIO_IR);        /*GPIO 对应中断编号*/
    ret = request_irq(ls1b_ir_irq, ls1b_ir_irq_handler, IRQF_TRIGGER_FALLING, "ls1b_ir", NULL);
                                        /*申请中断，设置中断处理函数，下降沿触发*/
    ...
    return 0;
}
```

打开设备操作中，首先申请和配置 GPIO61 引脚，配置成输入端口，申请中断，中断触发方式为下降沿，在每个下降沿到来时触发中断并调用中断处理函数 ls1b_ir_irq_handler()，中断处理程序中读端口值。

■读操作

驱动程序在输出电压下降沿时（中断内）采样端口值。红外发送的数据包含启动码、系统码和数据码。驱动程序中定义以下全局变量，表示接收数据的状态及数据值：

```
#define LS1B_IR_STATE_IDLE                  0          /*状态*/
#define LS1B_IR_STATE_RECEIVESTARTCODE      1
#define LS1B_IR_STATE_RECEIVESYSTEMCODE     2
#define LS1B_IR_STATE_RECEIVEDATACODE       3

static unsigned int   ls1b_ir_irq = 0;
static unsigned int   ls1b_ir_state = LS1B_IR_STATE_IDLE;    /*状态值*/
static unsigned int   ls1b_ir_interval = 0;                  /*两个下降沿之间时间间隔*/
static unsigned int   ls1b_ir_systembit_count = 0;           /*系统码位数*/
static unsigned int   ls1b_ir_databit_count = 0;             /*数据码位数*/
static unsigned int   ls1b_ir_key_code_tmp = 0;      /*按键值，暂存*/
static unsigned int   ls1b_ir_key_code = 0;          /*按键值，暂存*/

static struct timeval   ls1b_ir_current_tv = {0, 0};    /*时间值*/
static struct timeval   ls1b_ir_last_tv = {0, 0};

DECLARE_WAIT_QUEUE_HEAD(ls1b_wate_queue);      /*睡眠进程等待队列头*/
```

输出信号下降沿触发中断，调用中断处理函数如下：

```
static irqreturn_t ls1b_ir_irq_handler(int i, void *blah)
{
    udelay(50);        /*延迟，去毛刺*/
    if (gpio_get_value(GPIO_IR))        /*获取端口值，如果为高电平，返回（认为是毛刺）*/
        return IRQ_HANDLED;

    do_gettimeofday(&ls1b_ir_current_tv);       /*读取当前时间值*/
    if (ls1b_ir_current_tv.tv_sec == ls1b_ir_last_tv.tv_sec) {       /*两次读取时间秒数相同*/
        ls1b_ir_interval = ls1b_ir_current_tv.tv_usec - ls1b_ir_last_tv.tv_usec; /*两次下降沿时间间隔*/
    } else {            /*两次读取时间秒数不同*/
        ls1b_ir_interval = 1000000 - ls1b_ir_last_tv.tv_usec + ls1b_ir_current_tv.tv_usec;
    }
```

```
ls1b_ir_last_tv = ls1b_ir_current_tv;    /*最近下降沿时间值*/

if (ls1b_ir_interval > 800 && ls1b_ir_interval < 15000) {      /*时间间间隔（us）*/
    if (ls1b_ir_interval > 11000) {          /*收到状态码*/
            ls1b_ir_state = LS1B_IR_STATE_RECEIVESTARTCODE;
            ls1b_ir_key_code_tmp = 0;
            ls1b_ir_databit_count = 0;
            ls1b_ir_systembit_count =0;
    }
    else if (ls1b_ir_state == LS1B_IR_STATE_RECEIVESTARTCODE) {
        if (ls1b_ir_systembit_count >= SYSTEMCODE_BIT_NUM - 1) {
            ls1b_ir_state = LS1B_IR_STATE_RECEIVESYSTEMCODE;
            ls1b_ir_systembit_count = 0;
        }
        else if ((ls1b_ir_interval > 800 && ls1b_ir_interval < 1300) ||
                (ls1b_ir_interval > 1900 && ls1b_ir_interval < 2400)) {
            ls1b_ir_systembit_count ++;
        }
        else
            goto receive_errerbit;
    }
    else if (ls1b_ir_state == LS1B_IR_STATE_RECEIVESYSTEMCODE) {
        if (ls1b_ir_databit_count < 8) {
            if (ls1b_ir_interval > 1900 && ls1b_ir_interval < 2400) {
                ls1b_ir_key_code_tmp |= (1 << ls1b_ir_databit_count);
                ls1b_ir_databit_count++;
            }
            else if (ls1b_ir_interval > 800 && ls1b_ir_interval < 1300) {
                ls1b_ir_databit_count++;
            }
            else
                goto receive_errerbit;
        }
        else if ((ls1b_ir_interval > 800 && ls1b_ir_interval < 1300) ||
                (ls1b_ir_interval > 1900 && ls1b_ir_interval < 2400)) {
            ls1b_ir_state = LS1B_IR_STATE_IDLE;
            ls1b_ir_key_code = ls1b_ir_key_code_tmp;
            ls1b_ir_key_code_tmp = 0;
            ls1b_ir_databit_count = 0;
            ls1b_ir_systembit_count =0;
            wake_up_interruptible(&ls1b_wate_queue);   /*唤醒睡眠等待进程*/
            ...
        }
```

```
                else
                    goto receive_errerbit;
            }
            ls1b_ir_interval = 0;
            return IRQ_HANDLED;
        }
        ...
    }
```

中断函数根据两次下降沿之间的时间来确定输出的数据，并将数据保存到全局变量 **ls1b_ir_key_code**，并唤醒等待进程。

用户进程在读红外接收数据时先进入睡眠等待，等收到数据唤醒进程时读取数据值并返回，读操作函数定义如下：

```
static ssize_t ls1b_ir_read(struct file *filp, char __user *buf, size_t count, loff_t *offp)
{
    ls1b_ir_key_code = 0;

    if (filp->f_flags & O_NONBLOCK) {     /*只能阻塞访问*/
        return -EAGAIN;
    }

    wait_event_interruptible(ls1b_wate_queue, ls1b_ir_key_code);   /*插入等待队列*/
    /*收到数据后唤醒读进程，从 ls1b_ir_key_code 全局变量读接收数据*/
    if (copy_to_user(buf, &ls1b_ir_key_code, sizeof(unsigned int))) {     /*复制数据到用户空间*/
        ...
    }
    return count;
}
```

9.4 RTC 设备

RTC 表示实时时钟设备，它可以向系统提供真实时间，在系统关机时由主板上的电池供电，以维持时间值。龙芯 1B 芯片内置 RTC 模块，由外部晶振提供时钟驱动，主板断电后可由板上电池供电，仍能正常运行。内核可通过对 RTC 控制寄存器的操作来读取设置时间值和进行设备控制。

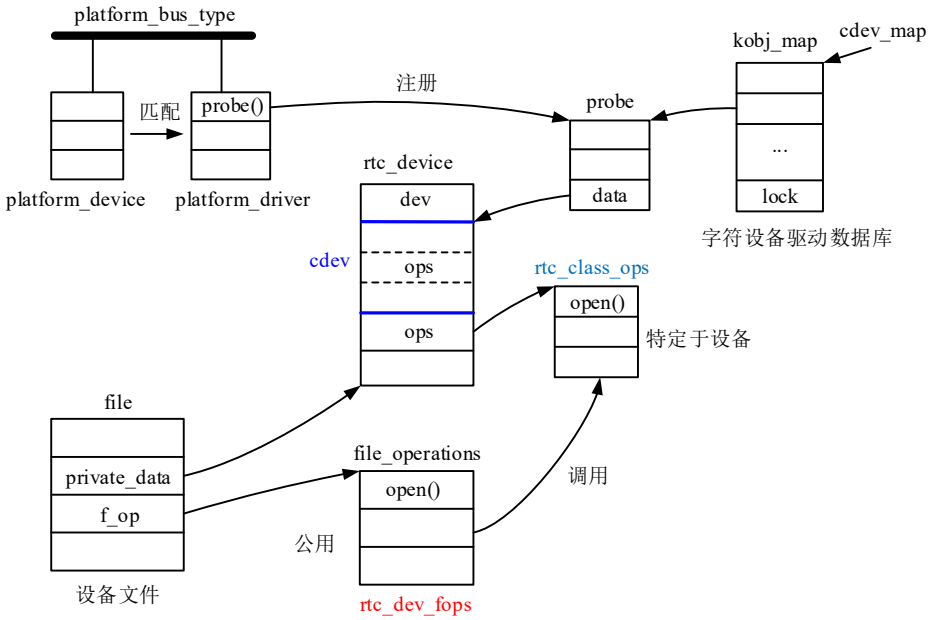
内核为 RTC 设备驱动建立了统一的框架，使用统一框架需选择配置选项 RTC_CLASS（RTC_LIB），RTC 驱动源代码位于/drivers/rtc/目录下。

9.4.1 驱动框架

RTC 设备驱动框架如下图所示，每个 RTC 设备在驱动框架中由 **rtc_device** 结构体表示，结构体内包含字符设备 cdev 结构体实例及特定于 RTC 设备的操作函数结构体 rtc_class_ops 指针成员等。驱动框架为 RTC 设备定义了公用的文件操作实例 **rtc_dev_fops**，文件操作实例中函数将调用特定于设备的 rtc_class_ops 结构体中的函数完成操作。

RTC 设备驱动的主要工作就是实现特定于设备的 **rtc_class_ops** 结构体实例，并依此在驱动 probe()函数中调用通用接口函数创建和注册 rtc_device 结构体实例。在注册 rtc_device 实例时，其内嵌 cdev 实例关

联的 file_operations 实例统一设为 **rtc_dev_fops**。



在打开 RTC 设备时，在公用文件操作结构 rtc_dev_fops 实例的 open()函数中将会根据 cdev 实例获取 rtc_device 实例指针，并赋予打开设备文件 file->private_data 成员。

内核在启动阶段后期初始化子系统时将初始化 RTC 驱动子系统，函数定义如下（/drivers/rtc/class.c）：

```
static int __init rtc_init(void)
{
    rtc_class = class_create(THIS_MODULE, "rtc");    /*创建 RTC 设备类*/
    ...
    rtc_class->pm = RTC_CLASS_DEV_PM_OPS;
    rtc_dev_init();                /*申请字符设备号，/drivers/rtc/rtc-dev.c*/
    rtc_sysfs_init(rtc_class);     /*导出至 sysfs 文件系统*/
    return 0;
}
subsys_initcall(rtc_init);
```

rtc_init()函数内主要是创建 RTC 设备类，并申请 RTC 设备号。rtc_dev_init()函数用于申请设备号，函数定义在/drivers/rtc/rtc-dev.c 文件内。

```
void __init rtc_dev_init(void)
{
    int err;
    err = alloc_chrdev_region(&rtc_devt, 0, RTC_DEV_MAX, "rtc");
                                                    /*申请设备号（动态分配主设备号）*/
    ...
}
```

全局变量 **rtc_devt** 用于保存 RTC 设备主设备号和起始从设备号，RTC_DEV_MAX 表示从设备号最大数量，定义为 16，函数内将为 RTC 设备动态申请主设备号，并申请从设备号区间。

9.4.2 注册设备

RTC 驱动框架中的 RTC 设备由 rtc_device 结构体表示，驱动框架中提供了创建和注册此实例的接口函数，供具体设备驱动的 probe()函数调用。驱动程序内需为设备定义 **rtc_class_ops** 结构体实例。

1 数据结构

RTC 驱动框架中每个 RTC 设备由 rtc_device 结构体表示，结构体定义在/include/linux/rtc.h 头文件：

```
struct rtc_device {
    struct device dev;              /*内嵌 device 实例，注册到通用驱动模型，自动创建设备文件*/
    struct module *owner;
    int id;                         /*设备编号，从设备号*/
    char name[RTC_DEVICE_NAME_SIZE];
    const struct rtc_class_ops *ops;        /*特定于设备的操作结构指针*/
    struct mutex ops_lock;
    struct cdev char_dev;                   /*内嵌字符设备 cdev 实例*/
    unsigned long flags;                    /*标记*/
    unsigned long irq_data;
    spinlock_t irq_lock;
    wait_queue_head_t irq_queue;        /*等待队列*/
    struct fasync_struct *async_queue;

    struct rtc_task *irq_task;
    spinlock_t irq_task_lock;
    int irq_freq;
    int max_user_freq;

    struct timerqueue_head timerqueue;
    struct rtc_timer aie_timer;
    struct rtc_timer uie_rtctimer;
    struct hrtimer pie_timer;
    int pie_enabled;
    struct work_struct irqwork;
    int uie_unsupported;

#ifdef CONFIG_RTC_INTF_DEV_UIE_EMUL
     ...
#endif
};
```

rtc_device 结构体主要成员简介如下：

- **dev**：device 结构体成员，用于在通用驱动模型中表示设备，注册此实例时自动创建设备文件。
- **id**：设备编号，从设备号。
- **ops**：rtc_class_ops 结构体指针，表示特定于设备的操作函数，需由具体设备驱动程序实现。
- **char_dev**：表示字符设备的 cdev 结构体成员。

●**irq_queue**：进程等待队列头，用于操作设备的进程睡眠等待操作完成（阻塞操作）。

具体设备驱动程序需要调用接口函数 **rtc_device_register()**创建并注册 rtc_device 结构体实例，后面再详细介绍此函数的实现。

下面先看一下 RTC 设备操作结构 rtc_class_ops 结构体的定义，这是一个必须有具体驱动程序实现的结构体，结构体定义如下（/include/linux/rtc.h）：

```
struct rtc_class_ops {
    int    (*open)(struct device *);                /*打开设备*/
    void    (*release)(struct device *);            /*释放设备*/
    int    (*ioctl)(struct device *, unsigned int, unsigned long);     /*设备控制操作*/
    int    (*read_time)(struct device *, struct rtc_time *);           /*读时间*/
    int    (*set_time)(struct device *, struct rtc_time *);            /*设置时间*/
    int    (*read_alarm)(struct device *, struct rtc_wkalrm *);       /*读取闹钟*/
    int    (*set_alarm)(struct device *, struct rtc_wkalrm *);        /*设置闹钟*/
    int    (*proc)(struct device *, struct seq_file *);
    int    (*set_mmss64)(struct device *, time64_t secs);
    int    (*set_mmss)(struct device *, unsigned long secs);
    int    (*read_callback)(struct device *, int data);     /*读取中断中提供的数据*/
    int    (*alarm_irq_enable)(struct device *, unsigned int enabled);
};
```

以上各函数参数类型 rtc_time 和 rtc_wkalrm 结构体定义在/include/uapi/linux/rtc.h 头文件内，此数据结构是用户进程与驱动程序交换时间信息的格式。

rtc_time 结构体表示真实的时间值：

```
struct rtc_time {
    int tm_sec;     /*秒*/
    int tm_min;     /*分*/
    int tm_hour;    /*秒*/
    int tm_mday;    /*日*/
    int tm_mon;     /*月*/
    int tm_year;    /*年*/
    int tm_wday;
    int tm_yday;
    int tm_isdst;
};
```

rtc_wkalrm 结构体表示设置的闹钟：

```
struct rtc_wkalrm {
    unsigned char enabled;     /* 0 = alarm disabled, 1 = alarm enabled */
    unsigned char pending;     /* 0 = alarm not pending, 1 = alarm pending */
    struct rtc_time time;      /* time the alarm is set to */
};
```

RTC 字符设备文件操作结构实例 rtc_dev_fops 中的函数调用特定于设备的 rtc_class_ops 实例中的函数

完成数据的传输。

2 接口函数

创建和注册 rtc_device 结构体实例的 rtc_device_register()函数定义如下（/drivers/rtc/class.c）：

```
struct rtc_device *rtc_device_register(const char *name, struct device *dev,   \
                                       const struct rtc_class_ops *ops, struct module *owner)
/*name：设备名称，dev：父设备，ops：rtc_class_ops 实例指针*/
{
    struct rtc_device *rtc;
    struct rtc_wkalrm alrm;
    int of_id = -1, id = -1, err;

    /*以下是获取设备编号（从设备号）*/
    if (dev->of_node)
        of_id = of_alias_get_id(dev->of_node, "rtc");
    else if (dev->parent && dev->parent->of_node)
        of_id = of_alias_get_id(dev->parent->of_node, "rtc");

    if (of_id >= 0) {
        id = ida_simple_get(&rtc_ida, of_id, of_id + 1,GFP_KERNEL);
        ...
    }

    if (id < 0) {
        id = ida_simple_get(&rtc_ida, 0, 0, GFP_KERNEL); /*分配 id，由 ida 结构管理*/
        ...
    }

    rtc = kzalloc(sizeof(struct rtc_device), GFP_KERNEL);      /*创建 rtc_device 实例*/
    ...
    rtc->id = id;           /*初始化成员*/
    rtc->ops = ops;         /*rtc_class_ops 实例指针*/
    rtc->owner = owner;
    rtc->irq_freq = 1;
    rtc->max_user_freq = 64;
    rtc->dev.parent = dev;
    rtc->dev.class = rtc_class;         /*设备类*/
    rtc->dev.release = rtc_device_release;

    mutex_init(&rtc->ops_lock);
    spin_lock_init(&rtc->irq_lock);
    spin_lock_init(&rtc->irq_task_lock);
    init_waitqueue_head(&rtc->irq_queue);     /*初始化等待队列*/
```

```
    /* Init timerqueue */
    timerqueue_init_head(&rtc->timerqueue);
    INIT_WORK(&rtc->irqwork, rtc_timer_do_work);
    /* Init aie timer */
    rtc_timer_init(&rtc->aie_timer, rtc_aie_update_irq, (void *)rtc);
    /* Init uie timer */
    rtc_timer_init(&rtc->uie_rtctimer, rtc_uie_update_irq, (void *)rtc);
    /* Init pie timer */
    hrtimer_init(&rtc->pie_timer, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
    rtc->pie_timer.function = rtc_pie_update_irq;
    rtc->pie_enabled = 0;

    strlcpy(rtc->name, name, RTC_DEVICE_NAME_SIZE);      /*复制名称字符串*/
    dev_set_name(&rtc->dev, "rtc%d", id);          /*设置 device 名称，设备文件名称*/

    /* Check to see if there is an ALARM already set in hw */
    err = __rtc_read_alarm(rtc, &alrm);

    if (!err && !rtc_valid_tm(&alrm.time))
        rtc_initialize_alarm(rtc, &alrm);

    rtc_dev_prepare(rtc);      /*初始化 cdev 实例，/drivers/rtc/rtc-dev.c*/

    err = device_register(&rtc->dev);       /*注册 rtc_device 内嵌 device 实例，自动创建设备文件*/
    ...

    rtc_dev_add_device(rtc);      /*注册 cdev 实例，/drivers/rtc/rtc-dev.c*/
    rtc_sysfs_add_device(rtc);
    rtc_proc_add_device(rtc);
     ...
    return rtc;      /*返回 rtc_device 实例指针*/
   ...
}
```

rtc_device_register()函数首先获取设备 id（从设备号），然后创建 rtc_device 实例并初始化成员，最后设置并向内核添加表示字符设备的 cdev 实例（内嵌在 rtc_device 实例）。

rtc_dev_prepare(rtc)和 rtc_dev_add_device(rtc)函数分别用于初始化和添加 cdev 实例，函数代码比较简单，简列如下：

```
void rtc_dev_prepare(struct rtc_device *rtc)
{
    if (!rtc_devt)      /*初始化函数 rtc_dev_init()中已经申请了设备号*/
        return;
```

```
    if (rtc->id >= RTC_DEV_MAX) {
        ...
    }

    rtc->dev.devt = MKDEV(MAJOR(rtc_devt), rtc->id);      /*合成设备号*/

#ifdef CONFIG_RTC_INTF_DEV_UIE_EMUL
    ...
#endif

    cdev_init(&rtc->char_dev, &rtc_dev_fops);    /*初始化 cdev 实例，注意 rtc_dev_fops 参数*/
    rtc->char_dev.owner = rtc->owner;
}
```

rtc_dev_prepare(rtc)函数内主要是初始化 rtc_device 实例中内嵌的 cdev 结构体实例。

```
void rtc_dev_add_device(struct rtc_device *rtc)
{
   if (cdev_add(&rtc->char_dev, rtc->dev.devt, 1))      /*添加 cdev 实例*/
      ...
   else
      ...
}
```

rtc_dev_add_device(rtc)函数用于向内核添加表示字符设备的 cdev 实例（**rtc->char_dev**）。

内核还定义了另一个创建并注册 rtc_device 实例的函数 devm_rtc_device_register()，函数定义如下：

```
struct rtc_device *devm_rtc_device_register(struct device *dev,const char *name,    \
                                            const struct rtc_class_ops *ops,struct module *owner)
/*dev：父设备的 device 实例指针，通常是 xxx_device 实例中嵌入的 device 实例*/
{
    struct rtc_device **ptr, *rtc;

    ptr = devres_alloc(devm_rtc_device_release, sizeof(*ptr), GFP_KERNEL);
    ...

    rtc = rtc_device_register(name, dev, ops, owner);    /*创建并注册 rtc_device 实例*/
    ...
    return rtc;
}
```

9.4.3 读写时间

RTC 设备文件统一的文件操作结构实例为 rtc_dev_fops，定义在/drivers/rtc/rtc-dev.c 文件内：

```
static const struct file_operations rtc_dev_fops = {
    .owner    = THIS_MODULE,
```

```
    .llseek     = no_llseek,
    .read       = rtc_dev_read,     /*读取 rtc_device.irq_data 成员值*/
    .poll       = rtc_dev_poll,
    .unlocked_ioctl= rtc_dev_ioctl,        /*设置/读取时间等操作，/drivers/rtc/rtc-dev.c*/
    .open       = rtc_dev_open,       /*调用 rtc->ops->open()函数*/
    .release    = rtc_dev_release,
    .fasync     = rtc_dev_fasync,
};
```

rtc_dev_fops 实例中的 read()函数并不是用来读取时间值，RTC 设备在中断处理程序中会设置 rtc_device 实例中的 rtc->irq_data 成员，read()函数内判断此成员是否为 0，如果为 0 则读进程进入睡眠等待；如果不为 0，则调用 rtc->ops->read_callback()函数读取 RTC 设备提供的数据并返回给读进程。

RTC 设备大部分的操作是通过 ioctl()系统调用完成，包括读写时间值，系统调用中调用 **rtc_dev_ioctl()** 函数完成相应的操作。RTC 设备 ioctl()系统调用常用命令如下（/include/uapi/linux/rtc.h）：

```
#define   RTC_ALM_SET      _IOW('p', 0x07, struct rtc_time) /* Set alarm time    */
#define   RTC_ALM_READ     _IOR('p', 0x08, struct rtc_time) /* Read alarm time */
#define   RTC_RD_TIME      _IOR('p', 0x09, struct rtc_time)    /*读时间*/
#define   RTC_SET_TIME     _IOW('p', 0x0a, struct rtc_time)    /*设置时间*/
#define   RTC_IRQP_READ       _IOR('p', 0x0b, unsigned long)       /* Read IRQ rate */
#define   RTC_IRQP_SET        _IOW('p', 0x0c, unsigned long)       /* Set IRQ rate */
#define   RTC_EPOCH_READ      _IOR('p', 0x0d, unsigned long)       /* Read epoch */
#define   RTC_EPOCH_SET       _IOW('p', 0x0e, unsigned long)       /* Set epoch */
...
```

rtc_dev_ioctl()函数相当于一个命令分配器，对不同的命令调用不同的处理函数，源代码请读者自行阅读，下面简单看一下读/写时间命令的处理。

```
static long rtc_dev_ioctl(struct file *file,unsigned int cmd, unsigned long arg)
/*cmd：命令，arg：地址，表示不同数据结构的地址*/
{
    int err = 0;
    struct rtc_device *rtc = file->private_data;
    const struct rtc_class_ops *ops = rtc->ops;
    struct rtc_time tm;
    struct rtc_wkalrm alarm;
    void __user *uarg = (void __user *) arg;
    ...
    switch (cmd) {
      ...
    case RTC_RD_TIME:     /*读时间*/
        ...
        err = rtc_read_time(rtc, &tm);     /*读时间函数*/
        ...
        if (copy_to_user(uarg, &tm, sizeof(tm)))     /*复制时间值至用户空间*/
            ...
```

```
            return err;

        case RTC_SET_TIME:      /*设置时间*/
            ...
            if (copy_from_user(&tm, uarg, sizeof(tm)))      /*复制时间值到内核空间*/
            ...
            return rtc_set_time(rtc, &tm);          /*设置时间值*/
           ...
        }
        ...
        return err;
    }
```

读写时间操作中 arg 参数是 **rtc_time** 实例的地址，读时间函数 **rtc_read_time(rtc, &tm)**中调用特定于设备的 **rtc_class_ops->read_time()**函数读取时间值至 rtc_time 实例，并将实例内容复制到用户空间。写时间操作与读时间操作相反，先从用户空间复制数据至 rtc_time 实例，再调用 **rtc_class_ops->set_time()**函数将时间值写入 RTC 设备。

9.4.4 驱动示例

龙芯 1B 处理器内置 RTC 模块，外接晶振频率 32.768KHz，处理器通过寄存器控制设备的工作，详情请参考龙芯 1B 处理器用户手册。

开发板源代码在板级相关文件中定义了 RTC 设备，并在初始化函数中注册 platform_device 实例：

```
static struct platform_device ls1x_rtc_device = {
    .name           = "ls1x-rtc",     /*匹配驱动*/
    .id             = 0,
    .num_resources  = ARRAY_SIZE(ls1x_rtc_resource),
    .resource       = ls1x_rtc_resource,
};
```

在驱动程序文件中定义了 platform_driver 实例（/drivers/rtc/rtc-ls1x.c）：

```
static struct platform_driver   ls1x_rtc_driver = {
    .driver         = {
        .name       = "ls1x-rtc",     /*匹配设备*/
    },
    .probe          = ls1x_rtc_probe,       /*探测函数*/
};
module_platform_driver(ls1x_rtc_driver);
```

驱动程序中定义的 rtc_class_ops 结构体实例 **ls1x_rtc_ops** 如下：

```
static struct rtc_class_ops   ls1x_rtc_ops = {
    .read_time = ls1x_rtc_read_time,      /*读时间*/
    .set_time = ls1x_rtc_set_time,        /*写时间*/
};
```

读写时间函数只是对 RTC 控制寄存器的操作，由于涉及到太多的细节这里就不详细介绍了，有兴趣的读者可自行参考源代码和处理器手册。

在驱动探测函数 prob()中将设置 RTC 模块，创建和注册设备对应的 rtc_device 实例，函数简列如下：

```
static int ls1x_rtc_probe(struct platform_device *pdev)
{
    ...   /*设置 RTC 模块*/

    rtcdev = devm_rtc_device_register(&pdev->dev, "ls1x-rtc",&ls1x_rtc_ops , THIS_MODULE);
    ...
    platform_set_drvdata(pdev, rtcdev);
    return 0;
    ...
}
```

9.5 简单设备驱动

本小节介绍两种简单设备的驱动程序框架。

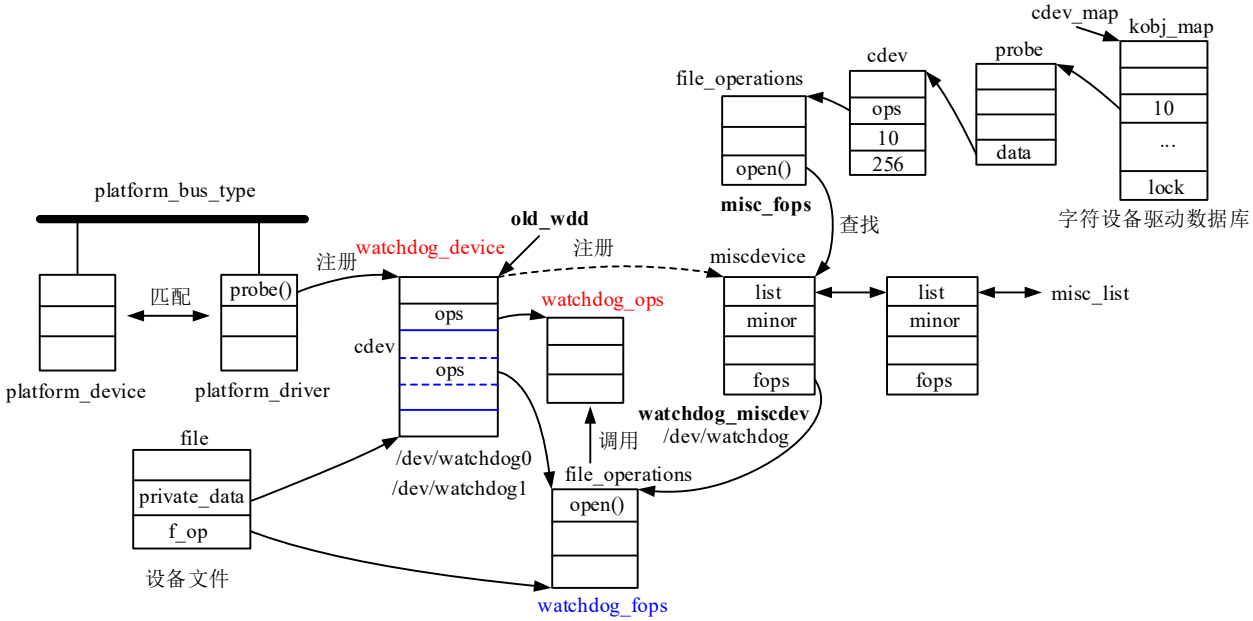
9.5.1 看门狗驱动

在系统中的看门狗定时器（WDT）实际上是一个计数器，一般给看门狗一个大数，程序开始运行后看门狗开始倒计数。如果程序运行正常，过一段时间 CPU 应发出指令使看门狗复位，重新开始倒计数。如果看门狗减到 0 就认为程序没有正常工作，强制整个系统复位。

看门狗设备驱动程序位于/drivers/watchdog/目录下。

1 驱动框架

看门狗设备驱动程序框架如下图所示，每个设备由 **watchdog_device** 结构体表示，结构体中内嵌表示字符设备的 cdev 结构体成员，以及指向 **watchdog_ops** 结构体的指针成员等，watchdog_ops 结构体包含看门狗设备的操作函数接口。



看门狗驱动程序内需要定义 watchdog_device 和 watchdog_ops 结构体实例，并将后者地址赋予前者，然后调用接口函数 **watchdog_register_device(wdd)**向驱动框架注册 watchdog_device 实例。

在注册 watchdog_device 实例时会向内核添加内嵌的 cdev 结构体成员，并将其 file_operations 文件操作结构指针赋为 **watchdog_fops**，此实例由驱动框架实现，是所有看门狗设备统一的文件操作结构实例，此实例中的函数调用看门狗操作结构 watchdog_ops 实例中的函数完成操作。

内核在初始化阶段将为看门狗设备动态申请主设备号，在注册 watchdog_device 实例时分配从设备号，以此合成设备号，对应的设备文件名称为/dev/watchdogX。

在老式的驱动框架中看门狗设备驱动采用 misc 驱动框架，为与之兼容，以上驱动框架在注册第一个 watchdog_device 实例时（编号为 0）将同时注册 miscdevice 结构体实例 watchdog_miscdev（静态定义），其文件操作结构实例也为 **watchdog_fops**，对应设备文件名称为/dev/watchdog（新框架中设备文件为 /dev/watchdog0）。全局变量 old_wdd 指向第一个 watchdog_device 实例。

在注册其它 watchdog_device 实例时，将不再创建和注册 miscdevice 实例。

看门狗驱动初始化函数 watchdog_init()定义如下（/drivers/watchdog/watchdog_core.c）：

```
static int __init watchdog_init(void)
{
    int err;

    watchdog_class = class_create(THIS_MODULE, "watchdog");     /*创建设备类*/
    ...
    err = watchdog_dev_init();   /*为看门狗设备动态申请主设备号，从设备号数量为 MAX_DOGS*/
    ...
    watchdog_deferred_registration();   /*将前期注册的 miscdevice 实例插入缓存链表，后期再注册*/
                                        /*设置 wtd_deferred_reg_done = true*/
    return 0;
}
subsys_initcall_sync(watchdog_init);
```

2 注册设备

内核中表示看门狗设备的 miscdevice 结构体定义如下（/include/linux/watchdog.h）：

```
struct watchdog_device {
    int id;          /*设备编号，从设备号*/
    struct cdev cdev;       /*表示字符设备的 cdev 结构体*/
    struct device *dev;     /*指向表示设备的 device 实例（内嵌在 xxx_device 结构体中）*/
    struct device *parent;
    const struct watchdog_info *info;
    const struct watchdog_ops *ops;     /*看门狗设备操作结构*/
    unsigned int bootstatus;
    unsigned int timeout;
    unsigned int min_timeout;
    unsigned int max_timeout;
    void *driver_data;
    struct mutex lock;
    unsigned long status;
    ...
    struct list_head deferred;
};
```

驱动程序最主要的工作就是实现特定于设备的 **watchdog_ops** 结构体实例，结构体定义如下：

```
struct watchdog_ops {
    struct module *owner;
    /*必选操作*/
    int (*start)(struct watchdog_device *);     /*启动看门狗*/
    int (*stop)(struct watchdog_device *);     /*停止看门狗*/
    /*以下是可选操作*/
    int (*ping)(struct watchdog_device *);
    unsigned int (*status)(struct watchdog_device *);
    int (*set_timeout)(struct watchdog_device *, unsigned int);
    unsigned int (*get_timeleft)(struct watchdog_device *);
    void (*ref)(struct watchdog_device *);
    void (*unref)(struct watchdog_device *);
    long (*ioctl)(struct watchdog_device *, unsigned int, unsigned long);
};
```

其中启动和停止看门狗的函数是必须实现的，其它是可选的。

驱动程序必须定义 watchdog_device 结构体及关联的 watchdog_ops 实例，在驱动探测函数中调用接口函数 **watchdog_register_device(wdd)**向内核注册，函数定义如下（/drivers/watchdog/watchdog_core.c）：

```
int watchdog_register_device(struct watchdog_device *wdd)
{
    int ret;
```

```
    mutex_lock(&wtd_deferred_reg_mutex);
    if (wtd_deferred_reg_done)
        ret = __watchdog_register_device(wdd);     /*注册 watchdog_device 实例*/
    else
        ret = watchdog_deferred_registration_add(wdd);
    mutex_unlock(&wtd_deferred_reg_mutex);
    return ret;
}
```

在__watchdog_register_device(wdd)函数中将注册 watchdog_device 实例，如果设备编号是 0（即第一个注册的看门狗设备）还将注册 miscdevice 结构体实例 watchdog_miscdev，源代码请读者自行阅读。

3 驱动示例

下面简要看一下龙芯 1B 处理器看门狗设备驱动程序的实现。

在板级相关代码中定义了表示设备的 platform_device 实例，并在初始化阶段向内核注册。

```
struct platform_device ls1x_wdt_pdev = {
    .name           = "ls1x-wdt",
    .id         = -1,
    .num_resources      = ARRAY_SIZE(ls1x_wdt_resources),
    .resource = ls1x_wdt_resources,
};
```

在驱动程序文件中定义了 platform_driver 实例和 watchdog_ops 实例：

```
static struct platform_driver ls1x_wdt_driver = {
    .probe = ls1x_wdt_probe,     /*探测函数*/
    .remove = ls1x_wdt_remove,
    .driver = {
        .name = "ls1x-wdt",     /*名称*/
    },
};
module_platform_driver(ls1x_wdt_driver);

static const struct watchdog_ops ls1x_wdt_ops = {
    .owner = THIS_MODULE,
    .start = ls1x_wdt_start,       /*对控制寄存器的操作，请读者自行阅读*/
    .stop = ls1x_wdt_stop,
    .ping = ls1x_wdt_ping,
    .set_timeout = ls1x_wdt_set_timeout,
};
```

在探测驱动函数 ls1x_wdt_probe()中将创建 watchdog_device 实例并注册，函数代码简列如下：

```
static int ls1x_wdt_probe(struct platform_device *pdev)
{
    struct ls1x_wdt_drvdata *drvdata;     /*内嵌 watchdog_device 结构体成员*/
```

```
    struct watchdog_device *ls1x_wdt;
    unsigned long clk_rate;
    struct resource *res;
    int err;

    drvdata = devm_kzalloc(&pdev->dev, sizeof(*drvdata), GFP_KERNEL);     /*创建结构体实例*/
    ...
    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    drvdata->base = devm_ioremap_resource(&pdev->dev, res);
    ...
    drvdata->clk = devm_clk_get(&pdev->dev, pdev->name);
    ...
    err = clk_prepare_enable(drvdata->clk);
    ...
    clk_rate = clk_get_rate(drvdata->clk);
    ...
    drvdata->clk_rate = clk_rate;

    ls1x_wdt = &drvdata->wdt;     /*指向 watchdog_device 结构体成员*/
    ls1x_wdt->info = &ls1x_wdt_info;
    ls1x_wdt->ops = &ls1x_wdt_ops;     /*赋予 watchdog_ops 实例地址*/
    ls1x_wdt->timeout = DEFAULT_HEARTBEAT;
    ls1x_wdt->min_timeout = 1;
    ls1x_wdt->max_hw_heartbeat_ms = U32_MAX / clk_rate * 1000;
    ls1x_wdt->parent = &pdev->dev;

    watchdog_init_timeout(ls1x_wdt, heartbeat, &pdev->dev);
    watchdog_set_nowayout(ls1x_wdt, nowayout);
    watchdog_set_drvdata(ls1x_wdt, drvdata);

    err = watchdog_register_device(&drvdata->wdt);     /*注册 watchdog_device 实例*/
    ...
    platform_set_drvdata(pdev, drvdata);
    ...
    return 0;
    ...
}
```

9.5.2 LED 驱动

Linux 内核中定义的 LED 设备专门处理各种外设的 LED 灯。内核中 LED 设备驱动并没有为其注册字符设备 cdev 实例，用户进程通过 sysfs 文件系统/sys/class/leds/目录下的文件操作 LED 设备。

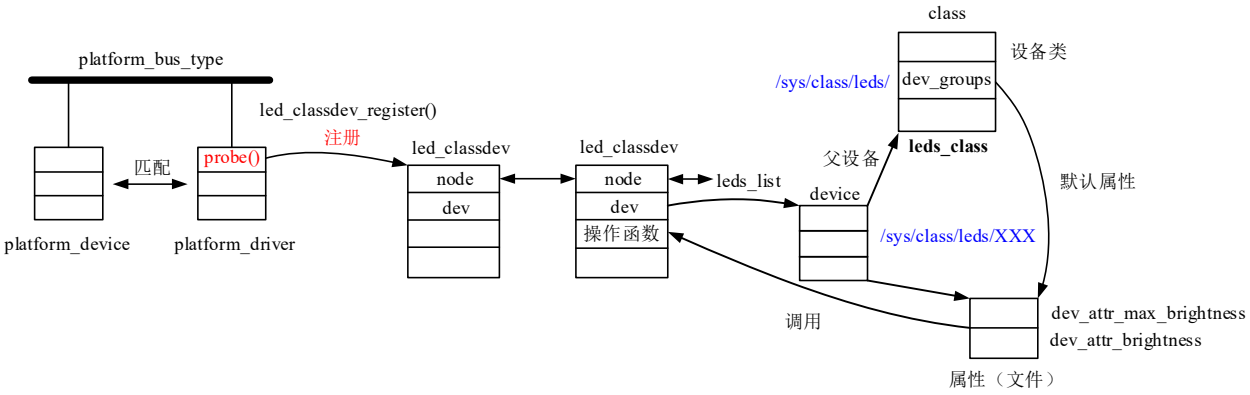
LED 设备驱动程序位于/drivers/leds/目录下。

1 驱动框架

LED 设备驱动框架如下图所示，设备驱动中通过 sysfs 文件系统中对设备属性的读写操作来操作设备，并没有使用字符设备驱动框架。

在初始化阶段，内核创建了 LED 设备类 leds_class 实例，LED 设备导出到 sysfs 文件系统中的路径为 /sys/class/leds/xxx，设备类为其下设备定义了两个默认属性 brightness 和 max_brightness（文件），用于调节 LED 亮度。

每个 LED 设备在驱动框架中由 **led_classdev** 结构体表示，LED 设备驱动在其 probe()函数中需要定义并设置 led_classdev 实例并调用接口函数 **led_classdev_register()**向内核注册。内核中 led_classdev 实例由全局双链表管理。



内核在/drivers/leds/led-class.c 文件内定义了 LED 驱动初始化函数 leds_init()，代码如下：

```
static int __init leds_init(void)
{
    leds_class = class_create(THIS_MODULE, "leds");   /*创建 LED 设备类*/
    ...
    leds_class->pm = &leds_class_dev_pm_ops;       /*电源管理操作结构*/
    leds_class->dev_groups = led_groups;           /*LED 设备默认属性组*/
    return 0;
}
subsys_initcall(leds_init);
```

leds_init()函数中创建了 LED 设备类 class 实例，并赋予设备类设备属性组 **led_groups**，属性组中主要包含以下两个属性：

```
static struct attribute *led_class_attrs[] = {
    &dev_attr_brightness.attr,
    &dev_attr_max_brightness.attr,
    NULL,
};
```

dev_attr_brightness 属性用于调节 LED 亮度（设置/显示亮度值），属性读写函数将调用 led_classdev 实例中的函数。属性 dev_attr_max_brightness 是只读属性，用于输出 LED 最大亮度值。

2 注册 LED 设备

LED 设备在驱动框架中由 led_classdev 结构体表示，结构体定义如下（/include/linux/leds.h）：

```
struct led_classdev {
    const char          *name;              /*设备名称（导出到 sysfs 文件系统）*/
    enum led_brightness     brightness;     /*当前亮度值*/
    enum led_brightness     max_brightness;  /*最大亮度值*/
    int     flags;      /*标记成员*/
    ...                 /*标记位定义*/
    void        (*brightness_set)(struct led_classdev *led_cdev,enum led_brightness brightness);
                                                                    /*设置亮度值*/
    int         (*brightness_set_sync)(struct led_classdev *led_cdev,enum led_brightness brightness);
    enum led_brightness (*brightness_get)(struct led_classdev *led_cdev);  /*获取亮度值*/

    int   (*blink_set)(struct led_classdev *led_cdev, unsigned long *delay_on,unsigned long *delay_off);

    struct device       *dev;       /*指向表示设备的 device 实例，添加到 sysfs 文件系统*/
    const struct attribute_group    **groups;   /*属性组*/

    struct list_head        node;           /*将实例添加到全局双链表 leds_list*/
    const char          *default_trigger;   /* Trigger to use */

    unsigned long           blink_delay_on, blink_delay_off;
    struct timer_list       blink_timer;
    int                 blink_brightness;
    void            (*flash_resume)(struct led_classdev *led_cdev);

    struct work_struct  set_brightness_work;
    int             delayed_set_value;

#ifdef CONFIG_LEDS_TRIGGERS
    /* Protects the trigger data below */
    struct rw_semaphore     trigger_lock;

    struct led_trigger  *trigger;
    struct list_head        trig_list;
    void            *trigger_data;
    /* true if activated - deactivate routine uses it to do cleanup */
    bool            activated;
#endif

    /* Ensures consistent access to the LED Flash Class device */
    struct mutex        led_access;
};
```

内核在/include/linux/leds.h 头文件内定义了枚举类型 led_brightness 表示 LED 设备的亮度值：

```
enum led_brightness {
    LED_OFF         = 0,
    LED_HALF    = 127,
    LED_FULL    = 255,
};
```

led_classdev 结构体中包含设置/获取 LED 亮度值的函数指针等成员。

LED 设备驱动程序需要创建并设置 led_classdev 结构体实例，尤其是要实现其中的操作函数，然后调用注册实例的接口函数 **led_classdev_register()**注册实例，函数代码简列如下（/drivers/leds/led-class.c）：

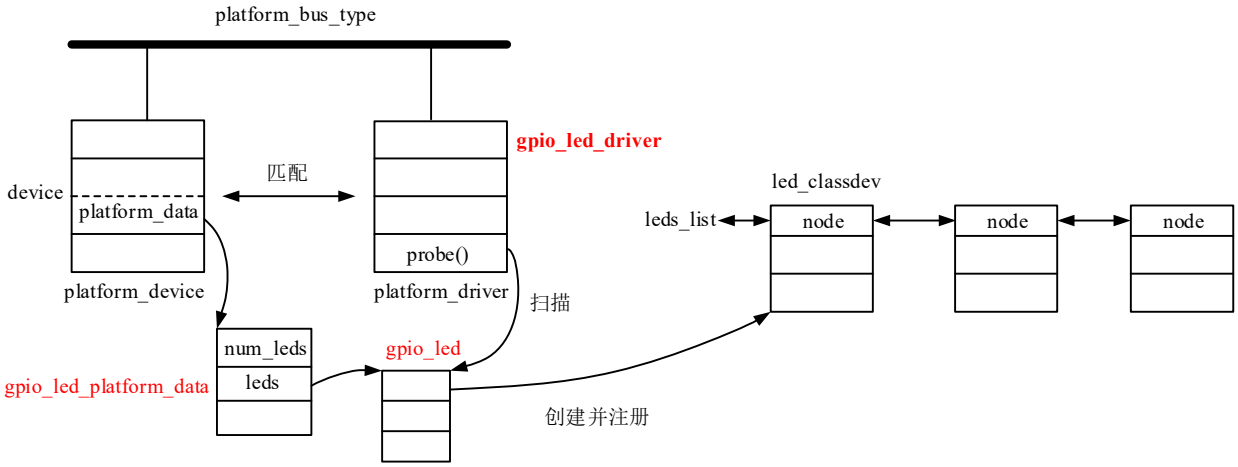
```
int led_classdev_register(struct device *parent, struct led_classdev *led_cdev)
{
    char name[64];
    int ret;

    ret = led_classdev_next_name(led_cdev->name, name, sizeof(name));     /*设备名称（编号）*/
    ...

    led_cdev->dev = device_create_with_groups(leds_class, parent, 0,
                                              led_cdev, led_cdev->groups, "%s", name);
                                                  /*创建表示设备的 device 实例*/
    ...

#ifdef CONFIG_LEDS_TRIGGERS
    init_rwsem(&led_cdev->trigger_lock);
#endif
    mutex_init(&led_cdev->led_access);
    down_write(&leds_list_lock);
    list_add_tail(&led_cdev->node, &leds_list);    /*将实例添加到全局双链表 leds_list 末尾*/
    up_write(&leds_list_lock);

    if (!led_cdev->max_brightness)
        led_cdev->max_brightness = LED_FULL;

    led_cdev->flags |= SET_BRIGHTNESS_ASYNC;

    led_update_brightness(led_cdev);       /*获取当前亮度并设置 brightness 成员*/

    INIT_WORK(&led_cdev->set_brightness_work, set_brightness_delayed);

    setup_timer(&led_cdev->blink_timer, led_timer_function, (unsigned long)led_cdev);

#ifdef CONFIG_LEDS_TRIGGERS
    led_trigger_set_default(led_cdev);
```

```
#endif
    ...
    return 0;
}
```

3 驱动示例

通常 LED 设备连接在 GPIO 上，内核实现了此类设备的通用驱动，用户不需要再编写此类设备驱动，只需要在板级相关代码中定义并注册相关的 platform_device 实例，实例中传入 GPIO 信息，并注册此实例。

内核定义并注册了通用的 platform_driver 驱动实例，在其探测函数中将创建、设置并注册 led_classdev 实例，连接在 GPIO 上的 LED 设备驱动框架如下图所示：



GPIO 信息通过 **gpio_led_platform_data** 结构体附加在 platform_device 实例，结构体中包含 GPIO 数量等信息，最重要的是 gpio_led 结构体数组，gpio_led 结构体中包含一个 GPIO 端口详细的信息。这两个结构体都定义在/include/linux/leds.h 头文件，请读者自行阅读。

龙芯 1B 开发板源码在板级文件中定义了 LED 设备相关的 platform_device 实例，如下所示：

```
static struct gpio_led_platform_data pca9555_gpio_led_info = {
    .leds       = pca9555_gpio_leds,            /*gpio_led 结构体数组*/
    .num_leds   = ARRAY_SIZE(pca9555_gpio_leds),
};

static struct platform_device pca9555_leds = {
    .name   = "leds-gpio",          /*名称，匹配驱动*/
    .id     = 0,
    .dev = {
        .platform_data= &pca9555_gpio_led_info,
    }
};
```

在初始化函数中将注册 pca9555_leds 实例。

LED 设备（GPIO）驱动在/drivers/leds/leds-gpio.c 文件内实现，定义的 platform_driver 实例如下：

```
static struct platform_driver gpio_led_driver = {       /*通用驱动*/
    .probe          = gpio_led_probe,           /*探测函数*/
```

```
    .remove         = gpio_led_remove,
    .driver         = {
        .name       = "leds-gpio",
        .of_match_table = of_gpio_leds_match,
    },
};
```

驱动探测函数 **gpio_led_probe()**中将扫描 platform_device 实例中传递的 gpio_led 结构体数组，为每个数组项（GPIO 端口）创建并注册 led_classdev 实例，实例中的操作函数调用 GPIO 驱动中的接口函数实现，源代码请读者自行阅读。

9.6 输入设备

前面几节介绍的都是一些比较简单的字符设备驱动，目的是让读者从简单的入手，理解字符设备驱动程序的实现，从本节开始将介绍几种比较复杂的字符设备驱动程序。

输入设备（如按键、键盘、鼠标等）是典型的字符设备，其一般工作原理是底层硬件有动作时产生一个中断，在中断处理程序中获取按键值、坐标值等数据，并将它们放入一个事件缓冲区。用户进程对输入设备的读写操作转换成对事件缓存区的读写。内核只负责向用户进程提供输入设备事件值，对事件的响应由用户进程完成，例如：若某用户进程在等待键盘输入，当有键按下时，驱动程序负责获取按键值存入缓存区，用户进程通过读设备操作从事件缓存区获取按键值，对按键值的响应也由用户进程实现。

内核对输入设备进行了分类，同类型输入设备事件缓存区的管理，以及与上层虚拟文件系统的接口由内核通用代码实现，具体设备驱动程序只需要在设备有动作时将事件类型和数值报告至输入设备驱动通用代码层即可。

输入设备驱动程序位于/drivers/input/目录下。

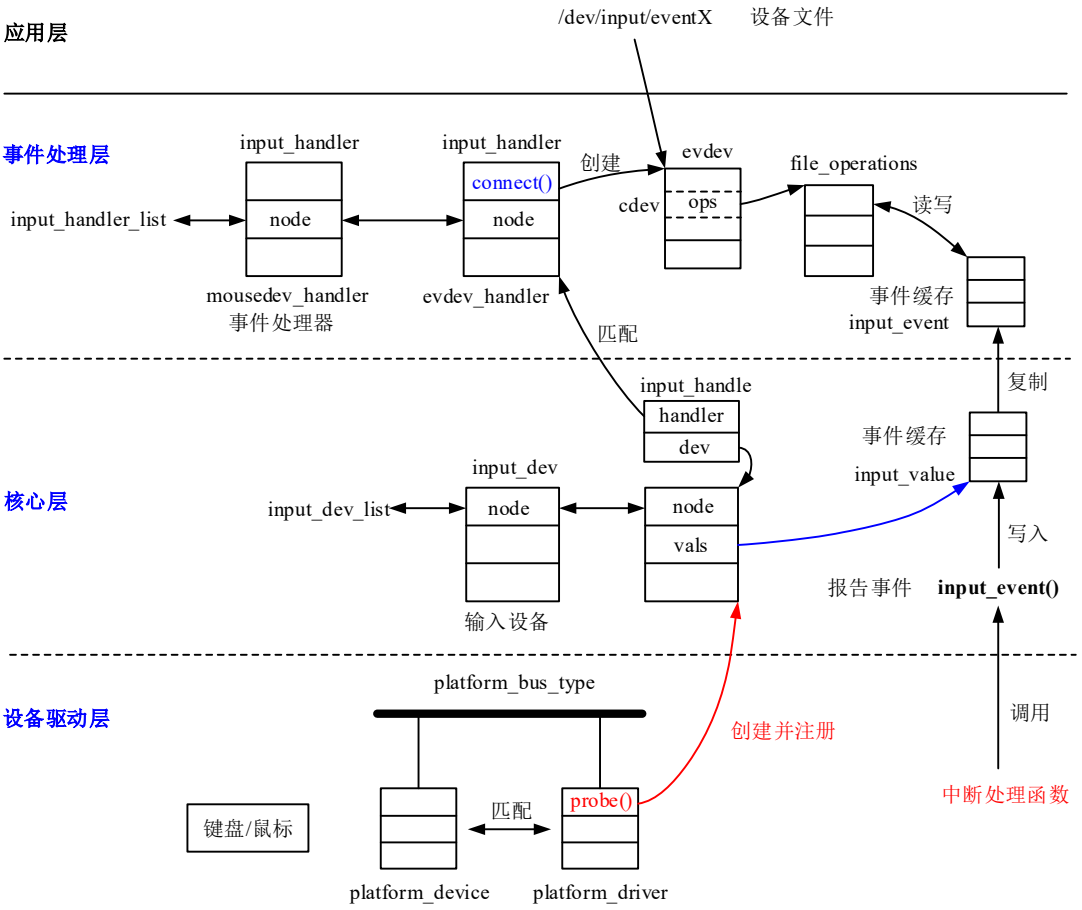
9.6.1 驱动框架

输入设备驱动框架如下图所示，驱动框架可分为三层，分别是设备驱动层、核心层和事件处理层。

最底层的设备驱动层，其驱动探测函数调用核心层函数创建并注册表示输入设备的 input_dev 结构体实例，在中断处理函数中调用接口函数向核心层报告事件（关联到 input_dev 实例）。

中间层是核心层，主要负责输入设备 input_dev 实例的管理、设备事件的缓存及建立与事件处理层事件处理器 input_handler 实例的关联。

最高层为事件处理层，主要负责管理事件处理器 input_handler 实例，建立 input_handler 与输入设备 input_dev 之间的关联（匹配两者），input_dev 与事件处理器 input_handler 匹配成功，将调用其中的连接函数，在连接函数中创建并添加表示字符设备的 cdev 实例（嵌入到另一个数据结构中）。cdev 实例中文件操作结构函数从 input_dev 实例中获取缓存事件，返回给用户进程，用户进程通过设备文件访问输入设备。

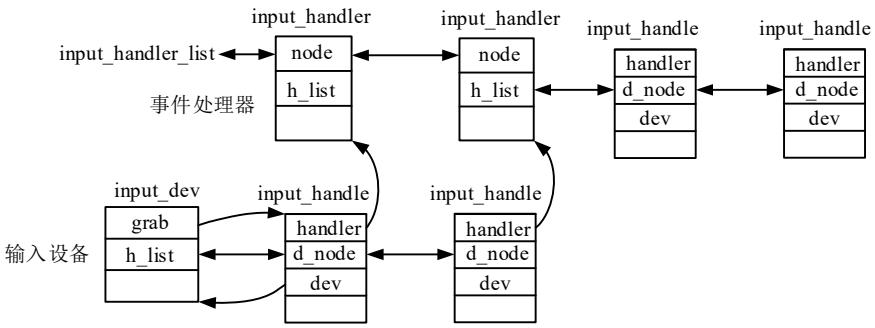


所有输入设备产生的事件将以 input_value 结构体的形式报告给输入设备 input_dev 实例，由其缓存（或处理）并提交给事件处理器，事件处理器负责将事件传递给用户进程。

输入设备驱动框架中定义了几个事件处理器实例，其中 evdev_handler 实例能匹配所有的输入设备，用户也可自行实现事件处理器实例并向内核注册，注册过程中将匹配内核中已注册的输入设备，匹配上则调用事件处理器中连接函数，为设备创建并注册 cdev 实例。

一个输入设备可以匹配多个事件处理器，事件处理器在用户看来就是操作输入设备的接口，也就是说允许用户通过多个接口操作同一个输入设备（可以有多个设备文件对应同一个输入设备）。

input_handle 结构体用来建立 input_dev 与 input_handler 实例之间的关联，如下图所示。input_dev 和 input_handler 结构体中都包含一个 input_handle 实例双链表，用于管理匹配的 input_handler 和 input_dev 实例。



输入设备主设备号为 INPUT_MAJOR（13），内核在/drivers/input/input.c 文件内定义了初始化函数：

static int __init input_init(void)

```
{
    int err;

    err = class_register(&input_class);     /*注册输入设备类*/
    ...
    err = input_proc_init();          /*在 proc 文件系统中创建目录和文件*/
    ...
    err = register_chrdev_region(MKDEV(INPUT_MAJOR, 0),   \
                    INPUT_MAX_CHAR_DEVICES, "input");      /*注册输入设备号*/
    ...
    return 0;
    ...
}
subsys_initcall(input_init);
```

注册设备号函数申请的最大从设备号数量为 INPUT_MAX_CHAR_DEVICES，取值为 1024，起始从设备号为 0。

9.6.2 设备

在输入设备驱动框架中，输入设备由 input_dev 结构体表示。在设备驱动的 probe()函数中，需要调用接口函数创建并注册 input_dev 实例。在注册实例过程中将为其查找匹配的事件处理器 **input_handler** 实例，匹配成功调用 input_handler 实例中的 **connect()**函数，为设备创建并注册 cdev 实例。

本小节介绍 input_dev 结构体的定义以及创建和注册实例的接口函数，下一小节将介绍事件处理器结构体的定义以及输入设备与事件处理器的匹配。

1 input_dev

输入设备由 input_dev 结构体表示，结构体定义如下（/include/linux/input.h）：

```
struct input_dev {
    const char *name;       /*输入设备名称*/
    const char *phys;       /*物理路径*/
    const char *uniq;       /*设备特定的标识码*/
    struct input_id   id;   /*设备 ID，input_id 实例，与 input_handler 匹配*/

    unsigned long   propbit[BITS_TO_LONGS(INPUT_PROP_CNT)];  /*设备属性（特性）位图*/

    unsigned long   evbit[BITS_TO_LONGS(EV_CNT)];      /*设备支持的事件类型位图*/
    /*以下是各类型事件中的支持事件值（事件代码）的位图*/
    unsigned long   keybit[BITS_TO_LONGS(KEY_CNT)];    /*/include/uapi/linux/input.h*/
    unsigned long   relbit[BITS_TO_LONGS(REL_CNT)];
    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)];

unsigned int hint_events_per_packet;
        /*设备一段时间内产生事件数量的平均数，事件处理器用于估算事件缓存区大小*/

unsigned int    keycodemax;     /*keycode 表大小*/
unsigned int    keycodesize;    /*keycode 表成员数*/
void *keycode;                  /*键值表指针*/

int (*setkeycode)(struct input_dev *dev,const struct input_keymap_entry *ke,   \
                                                        unsigned int *old_keycode);
int (*getkeycode)(struct input_dev *dev,struct input_keymap_entry *ke);

struct ff_device *ff;
unsigned int repeat_key;   /*保存最后的按键值，用于软件自动重复事件*/
struct timer_list    timer;   /*软件自动重复的定时器*/

int rep[REP_CNT];            /*自动重复的参数值*/
struct input_mt    *mt;        /*/include/linux/input/mt.h*/
struct input_absinfo    *absinfo;   /*input_absinfo 数组，保存坐标值信息*/

unsigned long    key[BITS_TO_LONGS(KEY_CNT)];   /*反映设备当前事件状态的位图*/
unsigned long    led[BITS_TO_LONGS(LED_CNT)];
unsigned long    snd[BITS_TO_LONGS(SND_CNT)];
unsigned long    sw[BITS_TO_LONGS(SW_CNT)];

int (*open)(struct input_dev *dev);          /*打开输入设备函数，打开设备文件时调用*/
void (*close)(struct input_dev *dev);         /*关闭设备操作函数*/
int (*flush)(struct input_dev *dev, struct file *file);   /*清除设备，刷出所有事件*/
int (*event)(struct input_dev *dev, unsigned int type, unsigned int code, int value);
                                                /*事件处理函数，直接由 input_dev 处理的事件*/
struct input_handle __rcu *grab;      /*指向 input_handle 实例，用于关联事件处理器*/

spinlock_t event_lock;
struct mutex mutex;

unsigned int users;          /*打开设备的用户数（input handlers）*/
bool going_away;           /*表示设备正在注销等非正常状态*/

struct device dev;         /*表示输入设备的 device 实例，添加到通用驱动模型中*/

struct list_head      h_list;    /*双链表头，用于管理 input_handle 实例，关联事件处理器*/
```

```
    struct list_head      node;      /*将实例添加到全局双链表 input_dev_list*/

    unsigned int num_vals;           /*表示 vals 指向数组当前使用的项数*/
    unsigned int max_vals;           /*最大项数，注册实例时设置*/
    struct input_value *vals;        /*指向 input_value 数组，缓存报告的事件，注册实例时分配空间*/

    bool devres_managed;             /*设备被管理不可注销或释放*/
};
```

input_dev 结构体主要成员简介如下：

● **evbit[]**：事件位图，标记设备支持的事件。内核为输入设备产生的事件定义了不同事件类型，如同步事件、按键事件、坐标事件等，输入设备事件类型及取值定义在/include/uapi/linux/input.h 头文件，如：

```
#define   EV_SYN          0x00        /*同步事件*/
#define   EV_KEY          0x01        /*按键值*/
#define   EV_REL          0x02        /*相对坐标值*/
#define   EV_ABS          0x03        /*绝对坐标值*/
#define   EV_MSC          0x04        /*其它类（杂项）*/
#define   EV_SW           0x05        /*开关事件*/
#define   EV_LED          0x11        /*LED 或其它指示设备*/
#define   EV_SND          0x12        /*声音输出，如蜂鸣器*/
#define   EV_REP          0x14        /*重复事件*/
#define   EV_FF           0x15        /*力反馈*/
#define   EV_PWR          0x16        /*电源管理*/
#define   EV_FF_STATUS        0x17    /*力反馈状态*/
#define   EV_MAX              0x1f
#define   EV_CNT          (EV_MAX+1)
```

其中同步事件包含以下几种（事件代码），其它事件类型包含的事件代码定义请读者自行查阅：

```
#define   SYN_REPORT          0
#define   SYN_CONFIG          1
#define   SYN_MT_REPORT       2
#define   SYN_DROPPED         3
#define   SYN_MAX             0xf
#define   SYN_CNT             (SYN_MAX+1)
```

input_dev 结构体中的位图表示了设备支持的事件类型、事件代码，以及当前设备事件状态等。设备支持某一类型事件则置位相应的事件位图，当某一类型事件发生时则置位事件状态位图中相应的位。

●**vals**：硬件设备在接收到某一事件时将产生中断，在中断处理函数中需要将事件报告给输入设备，即 input_dev 实例，输入设备以 input_value 结构体的形式缓存报告的事件。

vals 成员指向 input_value 结构体数组，input_value 结构体定义在/include/linux/input.h 头文件：

```
struct input_value {
    __u16 type;          /*事件类型，如：EV_SYN */
```

```
    __u16 code;          /*事件代码（每个事件类型有一套代码）*/
    __s32 value;         /*事件取值*/
};
```

input_dev 结构体中 max_vals 成员表示 input_value 数组大小，在注册 input_dev 实例时设置 max_vals 成员并依此为 input_value 数组分配空间，num_vals 成员表示 input_value 数组项当前使用的项数（写入了事件值）。

●**id**：input_id 结构体实例，结构体定义在/include/uapi/linux/input.h 头文件，用于与 input_handler 实例中 input_device_id 数组成员匹配，以判断是否匹配：

```
struct input_id {
    __u16 bustype;       /*总线类型*/
    __u16 vendor;        /*生产商*/
    __u16 product;
    __u16 version;       /*版本*/
};
```

●**event**：处理事件的函数指针，有些事件直接由输入设备处理，而不需要提交到事件处理器。

●**open**：打开设备文件时调用的函数。

●**h_list**：双链表头，管理 input_handle 实例，用于关联事件处理器。输入设备可以关联多个事件处理器。

●**grab**：当前 input_handle 结构体指针，用于关联 input_handler 实例。一个输入设备可关联到多个事件处理器，某一时刻设备事件将提交到当前 input_handle 关联的 input_handler 实例处理，后面再做介绍。

●**node**：双链表成员，将 input_dev 实例添加到全局双链表。内核定义了全局双链表用于管理 input_dev 实例（/drivers/input/input.c）：

```
static LIST_HEAD(input_dev_list);          /*全局双链表头*/
```

2 创建/注册 input_dev

input_dev 结构体实例由设备驱动程序创建并向输入设备驱动框架注册。

设备驱动中的探测函数调用 **input_allocate_device()**接口函数创建表示输入设备的 input_dev 结构体实例，接口函数 **input_register_device()**用于注册 input_dev 实例，下文将分别介绍这两个函数的实现。

input_allocate_device()函数定义如下（/drivers/input/input.c）：

```
struct input_dev *input_allocate_device(void)
{
    static atomic_t input_no = ATOMIC_INIT(-1);      /*顺序（原子）递增值*/
    struct input_dev *dev;

    dev = kzalloc(sizeof(struct input_dev), GFP_KERNEL);    /*分配 input_dev 实例*/
    if (dev) {
        dev->dev.type = &input_dev_type;     /*设备类型*/
        dev->dev.class = &input_class;       /*设备类*/
        device_initialize(&dev->dev);        /*初始化内嵌 device 结构体成员*/
        mutex_init(&dev->mutex);
```

```
        spin_lock_init(&dev->event_lock);
        init_timer(&dev->timer);
        INIT_LIST_HEAD(&dev->h_list);
        INIT_LIST_HEAD(&dev->node);

        dev_set_name(&dev->dev, "input%lu",(unsigned long)atomic_inc_return(&input_no));
                                                        /*设置设备名称为 inputX*/
        __module_get(THIS_MODULE);     /*增加模块引用计数*/
    }
    return dev;
}
```

设备驱动程序（probe()函数内）在创建 input_dev 实例后还需要对其进行初始化，尤其是支持事件位图的成员。随后，需要注册 input_dev 实例，注册函数为 **input_register_device()**，函数定义如下：

```
int input_register_device(struct input_dev *dev)
{
    struct input_devres *devres = NULL;
                    /*只包含指向 input_dev 实例的 input 指针成员，/drivers/input/input.c*/
    struct input_handler *handler;
    unsigned int packet_size;
    const char *path;
    int error;

    if (dev->devres_managed) {
        devres = devres_alloc(devm_input_device_unregister,sizeof(struct input_devres), GFP_KERNEL);
        if (!devres)
            return -ENOMEM;
        devres->input = dev;
    }

    /*设置支持 EV_SYN/SYN_REPORT 类型事件*/
    __set_bit(EV_SYN, dev->evbit);

    /*取消 KEY_RESERVED 事件支持*/
    __clear_bit(KEY_RESERVED, dev->keybit);

    input_cleanse_bitmasks(dev);
                /*设备不支持的事件类型，相应*bit[]成员清零，表示不支持此类型下的所有事件*/
    packet_size = input_estimate_events_per_packet(dev);     /*估算事件缓存区大小*/
    if (dev->hint_events_per_packet < packet_size)
        dev->hint_events_per_packet = packet_size;

    dev->max_vals = dev->hint_events_per_packet + 2;   /*缓存事件最大值*/
```

```
    dev->vals = kcalloc(dev->max_vals, sizeof(*dev->vals), GFP_KERNEL);
                                                /*为 input_val 数组分配空间，缓存事件*/
    if (!dev->vals) {
        ...
    }

    if (!dev->rep[REP_DELAY] && !dev->rep[REP_PERIOD]) {
        dev->timer.data = (long) dev;
        dev->timer.function = input_repeat_key;       /*默认定时器处理函数*/
        dev->rep[REP_DELAY] = 250;
        dev->rep[REP_PERIOD] = 33;
    }

    if (!dev->getkeycode)
        dev->getkeycode = input_default_getkeycode;       /*设置默认函数指针值*/

    if (!dev->setkeycode)
        dev->setkeycode = input_default_setkeycode;

    error = device_add(&dev->dev);
                        /*添加 input_dev.dev（没有挂载到总线），仅导出到 sysfs 文件系统*/
    ...
    list_add_tail(&dev->node, &input_dev_list);     /*将实例添加到全局双链表 input_dev_list 末尾*/

    list_for_each_entry(handler, &input_handler_list, node)   /*扫描 input_handler 双链表*/
        input_attach_handler(dev, handler);
                                    /*查找与 input_dev 匹配的 input_handler 实例（事件处理器）*/
    input_wakeup_procfs_readers();
    mutex_unlock(&input_mutex);
    if (dev->devres_managed) {
        dev_dbg(dev->dev.parent, "%s: registering %s with devres.\n",__func__, dev_name(&dev->dev));
        devres_add(dev->dev.parent, devres);
    }
    return 0;
    ...
}
```

注册函数完成 input_dev 实例部分成员初始化，为 input_val 数组分配内存空间，用于缓存设备提交的事件，最后扫描全局 input_handler 实例双链表，查找匹配的事件处理器。

下一小节将介绍匹配函数 **input_attach_handler**(dev, handler)的实现。

9.6.3 事件处理器

在输入设备驱动框架中，核心层主要管理 input_dev 实例，负责从设备驱动层接收提交的事件信息，然后将事件信息提交到事件处理层。事件处理层主要管理事件处理器实例，事件处理器由 input_handler 结

构体表示，可理解成输入设备 input_dev 的驱动。一个事件处理器可对应多个 input_dev 实例，一个 input_dev 实例也可以匹配多个事件处理器。事件处理器对下接收核心层提交的事件信息并进行处理（缓存事件），对上通过实现字符设备驱动程序来实现与 VFS 层的对接。

1 数据结构

事件处理器 input_handler 结构体定义如下（/include/linux/input.h）：

```
struct input_handler {
    void *private;      /*指向私有数据结构*/
    void (*event)(struct input_handle *handle, unsigned int type, unsigned int code, int value);
                                        /*事件处理函数，核心层在关闭中断状态下调用此函数*/
    void (*events)(struct input_handle *handle,const struct input_value *vals, unsigned int count);
                                        /*多事件处理函数，核心层在关闭中断状态下调用此函数*/
    bool (*filter)(struct input_handle *handle, unsigned int type, unsigned int code, int value);
                                        /*类似 event()函数，事件可由其处理时返回非零值*/
    bool (*match)(struct input_handler *handler, struct input_dev *dev);
                                        /*input_handler 与 input_dev 匹配函数，匹配成功返回 true，否则返回 false*/
    int (*connect)(struct input_handler *handler, struct input_dev *dev, const struct input_device_id *id);
                                        /*input_handler 与 input_dev 匹配成功调用此函数*/
    void (*disconnect)(struct input_handle *handle);   /*input_handler 与 input_dev 断开连接*/
    void (*start)(struct input_handle *handle);   /*input_dev 与 input_handler 匹配关联时调用此函数*/

    bool legacy_minors;       /*合法的从设备号*/
    int minor;                /*起始从设备号*/
    const char *name;         /*事件处理器名称*/

    const struct input_device_id *id_table;   /*匹配列表，/include/linux/mod_devicetable.h*/

    struct list_head    h_list;     /*双链表头，管理 input_handle 实例，关联匹配的 input_dev 实例*/
    struct list_head    node;       /*将实例添加到全局双链表 input_handler_list*/
};
```

input_handler 结构体主要成员简介如下：

●**event()**：事件处理器处理单个事件的函数。

●**events()**：事件处理器处理多个事件的函数。

●**filter()**：事件处理器单个事件处理函数，与 event()函数相似，filter()在 event()之前处理事件，可由 filter()处理的事件返回值为非零时，不再由 event()函数对其进行处理，返回值为零的事件将继续由 event()处理。

●**match()**：注册 input_handler 或 input_dev 实例时，将调用此函数查找匹配的 input_dev 或 input_handler 实例。

●**connect()**：input_handler 与 input_dev 实例匹配成功时，将调用此函数，完成两个实例之间的关联，并完成字符设备驱动程序 **cdev** 实例的创建和注册。

●**id_table**：指向 input_device_id 结构体数组，结构体定义在/include/linux/mod_devicetable.h 头文件：

```
struct input_device_id {
    kernel_ulong_t  flags;      /*标记，匹配时需检查的项目，如总线类型、支持事件等*/
```

```
    __u16 bustype;
    __u16 vendor;
    __u16 product;
    __u16 version;
                                                    /*事件位图*/
    kernel_ulong_t  evbit[INPUT_DEVICE_ID_EV_MAX / BITS_PER_LONG + 1];
    kernel_ulong_t  keybit[INPUT_DEVICE_ID_KEY_MAX / BITS_PER_LONG + 1];
    kernel_ulong_t  relbit[INPUT_DEVICE_ID_REL_MAX / BITS_PER_LONG + 1];
    kernel_ulong_t  absbit[INPUT_DEVICE_ID_ABS_MAX / BITS_PER_LONG + 1];
    kernel_ulong_t  mscbit[INPUT_DEVICE_ID_MSC_MAX / BITS_PER_LONG + 1];
    kernel_ulong_t  ledbit[INPUT_DEVICE_ID_LED_MAX / BITS_PER_LONG + 1];
    kernel_ulong_t  sndbit[INPUT_DEVICE_ID_SND_MAX / BITS_PER_LONG + 1];
    kernel_ulong_t  ffbit[INPUT_DEVICE_ID_FF_MAX / BITS_PER_LONG + 1];
    kernel_ulong_t  swbit[INPUT_DEVICE_ID_SW_MAX / BITS_PER_LONG + 1];

    kernel_ulong_t  driver_info;
};
```

flags 标记成员表示在匹配 input_dev 实例 id 成员与 input_device_id 数组项时，检查哪些成员值，如总线类型、支持事件等，标记位取值定义在/include/linux/mod_devicetable.h 头文件。

匹配操作中将 input_dev 实例 id 成员与 input_handler 实例中 id_table 指向数组项逐个进行比较，判断两者之间是否匹配。

2 注册实例

input_handler 实例由输入设备驱动通用代码实现，通常每个实例用于处理一类输入设备事件，实例在创建后需要向内核注册，注册函数定义如下（/drivers/input/input.c）：

```
int input_register_handler(struct input_handler *handler)
{
    struct input_dev *dev;
    int error;

    error = mutex_lock_interruptible(&input_mutex);
    ...
    INIT_LIST_HEAD(&handler->h_list);

    list_add_tail(&handler->node, &input_handler_list);
                                        /*将实例添加到全局双链表 input_handler_list 末尾*/
    list_for_each_entry(dev, &input_dev_list, node)
        input_attach_handler(dev, handler);      /*扫描 input_dev 实例双链表查找匹配设备*/

    input_wakeup_procfs_readers();
    mutex_unlock(&input_mutex);
    return 0;
}
```

注册 input_handler 实例函数将扫描 input_dev 实例双链表，调用 **input_attach_handler()**函数判断事件处理器与 input_dev 实例是否匹配，此函数内将调用 input_handler 实例中的 match()函数判断 input_handler 实例与当前 input_dev 实例是否匹配，如果匹配还将调用 input_handler 实例中的 **connect()**函数建立 input_dev 实例与 input_handler 实例的关联，并创建和注册表示字符设备的 cdev 实例。

在前面注册 input_dev 实例的函数中，将扫描 input_handler 双链表，也调用 **input_attach_handler()**函数判断匹配性，下文将介绍此函数的实现。

3 匹配函数

input_attach_handler()函数中先检查 input_handler 实例与 input_dev 实例是否匹配，如果匹配还将调用 input_handler 实例中的 **connect()**函数。

input_attach_handler()函数定义如下（/drivers/input/input.c）：

```
static int input_attach_handler(struct input_dev *dev, struct input_handler *handler)
{
    const struct input_device_id *id;
    int error;

    id = input_match_device(handler, dev);     /*检查匹配性*/
    if (!id)                    /*id 为空则返回错误码*/
        return -ENODEV;

    error = handler->connect(handler, dev, id);     /*匹配成功再调用执行 connect()函数*/
    ...
    return error;
}
```

以上函数内调用 input_match_device(handler, dev)函数检查 input_handler 与 input_dev 实例的匹配性，如果匹配则再调用 input_handler 实例的 connect()函数。

input_match_device()函数定义如下，如果匹配成功则返回 input_device_id 实例指针，否则返回 NULL：

```
static const struct input_device_id *input_match_device(struct input_handler *handler,struct input_dev *dev)
{
    const struct input_device_id *id;
    for (id = handler->id_table; id->flags || id->driver_info; id++) {    /*扫描 input_device_id 列表*/

        if (id->flags & INPUT_DEVICE_ID_MATCH_BUS)   /*需要匹配的项目逐个判断是否匹配*/
            if (id->bustype != dev->id.bustype)
                continue;

        if (id->flags & INPUT_DEVICE_ID_MATCH_VENDOR)
            if (id->vendor != dev->id.vendor)
                continue;

        if (id->flags & INPUT_DEVICE_ID_MATCH_PRODUCT)
            if (id->product != dev->id.product)
```

```
                continue;

        if (id->flags & INPUT_DEVICE_ID_MATCH_VERSION)
            if (id->version != dev->id.version)
                continue;

        if (!bitmap_subset(id->evbit, dev->evbit, EV_MAX))      /*支持事件匹配*/
            continue;

        if (!bitmap_subset(id->keybit, dev->keybit, KEY_MAX))
            continue;

        if (!bitmap_subset(id->relbit, dev->relbit, REL_MAX))
            continue;

        if (!bitmap_subset(id->absbit, dev->absbit, ABS_MAX))
            continue;

        if (!bitmap_subset(id->mscbit, dev->mscbit, MSC_MAX))
            continue;

        if (!bitmap_subset(id->ledbit, dev->ledbit, LED_MAX))
            continue;

        if (!bitmap_subset(id->sndbit, dev->sndbit, SND_MAX))
            continue;

        if (!bitmap_subset(id->ffbit, dev->ffbit, FF_MAX))
            continue;

        if (!bitmap_subset(id->swbit, dev->swbit, SW_MAX))
            continue;

        if (!handler->match || handler->match(handler, dev))
                                /*input_handler 实例匹配函数不为空则调用执行，否则直接返回 id*/
            return id;      /*返回 input_device_id 实例指针*/
    }
    /*如果 id->driver_info 非零，flags 为 0 且事件处理器没有定义 match()函数，则与设备匹配*/
    return NULL;
}
```

匹配函数扫描 input_handler 实例 input_device_id 列表（id_table 成员指向数组），首先进行 input_dev 实例 id 成员与列表项的匹配性检查，如果匹配再进行支持事件的匹配检查等，以上都匹配后，若 match()函数指针为空或执行 match()函数返回非零值，则表示 input_dev 与当前 input_device_id 列表项匹配，返回

input_device_id 列表项指针。

如果扫描完 input_device_id 列表都没有找到匹配的项，则函数返回 NULL。

如果 input_dev 实例与 input_handler 实例匹配，则 **input_attach_handler()**函数调用 input_handler 实例的 connect()函数，函数内主要建立 input_dev 与 input_handler 实例之间的关联，以及字符设备驱动程序 cdev 实例的创建和添加。

注意：在注册 input_dev 实例和 input_handler 实例的函数中，会扫描内核 input_handler 实例和 input_dev 实例双链表，查找所有的匹配项。在注册一个 input_handler 实例时，会查找所有匹配的 input_dev 实例，在注册 input_dev 实例时也会查找所有匹配的 input_handler 实例，而并不是匹配上一个 input_handler 实例就结束，input_dev 实例和 input_handler 实例是多对多的关系。

内核输入设备驱动通用层代码实现并注册了几个事件处理器实例，每个实例适用于某一类（或所有）输入设备，后面将会介绍一个具体的事件处理器的实现。

9.6.4 报告设备事件

硬件设备在有输入信号时，将产生中断，在中断处理程序中需要将事件报告给设备对应的 input_dev 实例。中断处理程序中以 input_value 结构体实例的形式报告事件，input_dev 实例缓存报告的事件，然后将事件提交给事件处理器处理（也可以直接由设备处理）。

1 报告事件

硬件设备驱动程序，在中断中向输入设备驱动核心层报告事件的形式为 input_value 结构体实例，结构体定义如下（/include/linux/input.h）：

```
struct input_value {
    __u16 type;          /*事件类型，如：EV_SYN */
    __u16 code;          /*事件代码*/
    __s32 value;         /*取值*/
};
```

input_value 结构体中事件类型编码、事件代码、取值与 input_dev 结构体中对应成员的取值相同。

input_dev 结构体中包含指向 input_val 结构体数组的成员，数组用于缓存事件，报告事件函数将事件提交至 input_dev 事件缓存。当发出同步事件时，input_dev 缓存区事件将提交至 input_handler 实例，并由其处理，具体处理方法后面再做介绍，这里先介绍事件报告的机制。

输入设备驱动核心层提供了报告事件的通用 API 函数，具体设备驱动程序在外设有动作或中断时，调用通用 API 函数报告事件即可。**input_event()**函数是报告事件的核心函数，其它 API 函数是对 input_event()函数的封装。下面我们先来看一下 input_event()函数的实现（/drivers/input/input.c）：

```
void input_event(struct input_dev *dev,unsigned int type, unsigned int code, int value)
/*参数表示事件属性*/
{
    unsigned long flags;

    if (is_event_supported(type, dev->evbit, EV_MAX)) {   /*检查 input_dev 是否支持本事件*/
        spin_lock_irqsave(&dev->event_lock, flags);
        input_handle_event(dev, type, code, value);     /*/drivers/input/input.c*/
```

```
            spin_unlock_irqrestore(&dev->event_lock, flags);
        }
    }
```

input_event()函数内调用 input_handle_event()函数完成事件报告工作，函数内首先要确定事件交由谁处理。内核在/drivers/input/input.c 文件内定义了下列宏，用于表示事件交由谁处理：

```
#define   INPUT_IGNORE_EVENT          0    /*事件被忽略*/
#define   INPUT_PASS_TO_HANDLERS      1    /*事件交由 input_handler 处理*/
#define   INPUT_PASS_TO_DEVICE        2    /*事件交由 input_dev 处理*/
#define   INPUT_SLOT                  4    /*群聚事件，如触模屏*/
#define   INPUT_FLUSH                 8    /*刷新事件，立即处理所有事件（提交 input_dev 缓存事件）*/
#define   INPUT_PASS_TO_ALL    (INPUT_PASS_TO_HANDLERS | INPUT_PASS_TO_DEVICE)
                                           /*事件由 input_dev 和 input_handler 处理*/
```

input_get_disposition()函数根据事件属性确定事件交由谁处理，返回值为以上宏，源代码如下：

```
static int input_get_disposition(struct input_dev *dev,unsigned int type, unsigned int code, int *pval)
{
    int disposition = INPUT_IGNORE_EVENT;          /*保存返回值，事件由谁处理*/
    int value = *pval;      /*value 保存输入事件值*/

    switch (type) {         /*事件类型*/
    case EV_SYN:            /*同步事件*/
        switch (code) {
        case SYN_CONFIG:
            disposition = INPUT_PASS_TO_ALL;
            break;

        case SYN_REPORT:        /*报告事件且同步事件*/
            disposition = INPUT_PASS_TO_HANDLERS | INPUT_FLUSH;
            break;
        case SYN_MT_REPORT:
            disposition = INPUT_PASS_TO_HANDLERS;
            break;
        }
        break;

    case EV_KEY:      /*按键事件*/
        if (is_event_supported(code, dev->keybit, KEY_MAX)) {
            if (value == 2) {
                disposition = INPUT_PASS_TO_HANDLERS;
                break;
            }

            if (!!test_bit(code, dev->key) != !!value) {
```

```
                __change_bit(code, dev->key);
                disposition = INPUT_PASS_TO_HANDLERS;
            }
        }
        break;

case EV_SW:       /*开关事件*/
        if (is_event_supported(code, dev->swbit, SW_MAX) &&!!test_bit(code, dev->sw) != !!value) {
            __change_bit(code, dev->sw);
            disposition = INPUT_PASS_TO_HANDLERS;
        }
        break;

case EV_ABS:   /*绝对值事件*/
        if (is_event_supported(code, dev->absbit, ABS_MAX))
            disposition = input_handle_abs_event(dev, code, &value);   /*处理绝对值事件，input.c*/
        break;

case EV_REL:     /*相对值事件*/
        if (is_event_supported(code, dev->relbit, REL_MAX) && value)
            disposition = INPUT_PASS_TO_HANDLERS;
        break;

case EV_MSC:
        if (is_event_supported(code, dev->mscbit, MSC_MAX))
            disposition = INPUT_PASS_TO_ALL;
        break;

case EV_LED:
        if (is_event_supported(code, dev->ledbit, LED_MAX) &&!!test_bit(code, dev->led) != !!value) {
            __change_bit(code, dev->led);
            disposition = INPUT_PASS_TO_ALL;
        }
        break;

case EV_SND:
        if (is_event_supported(code, dev->sndbit, SND_MAX)) {
            if (!!test_bit(code, dev->snd) != !!value)
                __change_bit(code, dev->snd);
            disposition = INPUT_PASS_TO_ALL;
        }
        break;
```

```
    case EV_REP:
        if (code <= REP_MAX && value >= 0 && dev->rep[code] != value) {
            dev->rep[code] = value;
            disposition = INPUT_PASS_TO_ALL;
        }
        break;

    case EV_FF:
        if (value >= 0)
            disposition = INPUT_PASS_TO_ALL;
        break;

    case EV_PWR:
        disposition = INPUT_PASS_TO_ALL;
        break;
    }
    *pval = value;    /*恢复原来值*/
    return disposition;        /*返回事件由谁处理标记*/
}
```

下面来看一下报告事件 input_event()函数中调用的 **input_handle_event()**函数的实现：

```
static void input_handle_event(struct input_dev *dev,unsigned int type, unsigned int code, int value)
{
    int disposition;

    disposition = input_get_disposition(dev, type, code, &value);
                                        /*确定事件交由谁处理，/drivers/input/input.c*/

    if ((disposition & INPUT_PASS_TO_DEVICE) && dev->event)
                                        /*由 input_dev 处理且 dev->event 非空*/
        dev->event(dev, type, code, value);          /*调用 input_dev 定义的 event()函数*/

    if (!dev->vals)    /*如果 dev->vals 为 NULL，函数返回，注册 input_dev 实例时为数组分配空间*/
        return;

    /*事件需要由 input_handler 处理*/
    if (disposition & INPUT_PASS_TO_HANDLERS) {
        struct input_value *v;
        if (disposition & INPUT_SLOT) {        /*群聚事件，如触模屏，事件信息写入缓存区*/
            v = &dev->vals[dev->num_vals++];
            v->type = EV_ABS;
            v->code = ABS_MT_SLOT;
            v->value = dev->mt->slot;    /*传递事件值*/
```

```
        }

        v = &dev->vals[dev->num_vals++];     /*将事件信息添加到 input_dev 事件缓存区*/
        v->type = type;
        v->code = code;
        v->value = value;
    }

    if (disposition & INPUT_FLUSH) {   /*刷新事件，处理所有缓存事件*/
        if (dev->num_vals >= 2)
            input_pass_values(dev, dev->vals, dev->num_vals);
                                        /*将缓存事件提交到 input_handler 处理*/
        dev->num_vals = 0;
    } else if (dev->num_vals >= dev->max_vals - 2) {     /*缓存事件较多*/
        dev->vals[dev->num_vals++] = input_value_sync;       /*内核定义的同步事件，input.c*/
        input_pass_values(dev, dev->vals, dev->num_vals); /*将缓存事件提交到 input_handler 处理*/
        dev->num_vals = 0;
    }
}
```

input_handle_event()函数首先判断事件将交由谁处理，如果交由设备处理且 input_dev 实例 event()函数不为空则事件由设备处理；再判断是否是需由 input_handler 处理，其中又要区分群聚事件和一般事件，处理事件需要将事件信息写入设备事件缓存区；最后判断事件处理是不是 INPUT_FLUSH 或设备事件缓存区事件数量超过一定值，是则调用 **input_pass_values()**函数，将事件打包提交到 input_handler 事件处理器。

input_pass_values()函数定义如下：

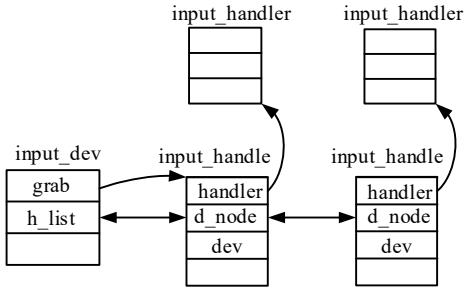
```
static void input_pass_values(struct input_dev *dev,struct input_value *vals, unsigned int count)
/*dev：指向输入设备，vals：事件缓存区指针，count：事件数量*/
{
    struct input_handle *handle;
    struct input_value *v;

    if (!count)
        return;

    rcu_read_lock();

    handle = rcu_dereference(dev->grab);     /*当前 input_handle 实例*/
    if (handle) {   /*如果设备当前有关联的 input_handle 实例则将事件交给其关联的 input_handler*/
        count = input_to_handler(handle, vals, count);   /*事件交由事件处理器处理*/
    } else { /*当前关联的 input_handle 为 NULL 则将事情发送给设备关联的所有 input_handle 实例*/
        list_for_each_entry_rcu(handle, &dev->h_list, d_node)
            if (handle->open) {
                count = input_to_handler(handle, vals, count);
                if (!count)
```

```
                    break;
                }
        }
        rcu_read_unlock();
        add_input_randomness(vals->type, vals->code, vals->value);

        /*触发键盘事件的重复操作*/
        if (test_bit(EV_REP, dev->evbit) && test_bit(EV_KEY, dev->evbit)) {
            for (v = vals; v != vals + count; v++) {
                if (v->type == EV_KEY && v->value != 2) {
                    if (v->value)
                        input_start_autorepeat(dev, v->code);
                    else
                        input_stop_autorepeat(dev);
                }
            }
        }
    }
```

input_pass_values()函数主要工作是将 input_dev 实例中缓存的事件提交给 input_handler 实例处理。

input_dev 结构体中 h_list 成员用于链接 input_handle 实例，每个 input_handle 实例关联一个事件处理器 input_handler 实例，也就是说一个设备可关联多个 input_handler 实例，如下图所示。input_dev 结构体中 grab 成员指向的 input_handle 实例关联的事件处理器 input_handler 实例是当前处理设备事件的事件处理器。

如果 grab 成员不为空则将事件交由当前关联的事件处理器处理，如果 grab 为 NULL，则将事件发送给设备关联的所有 input_handler 实例。



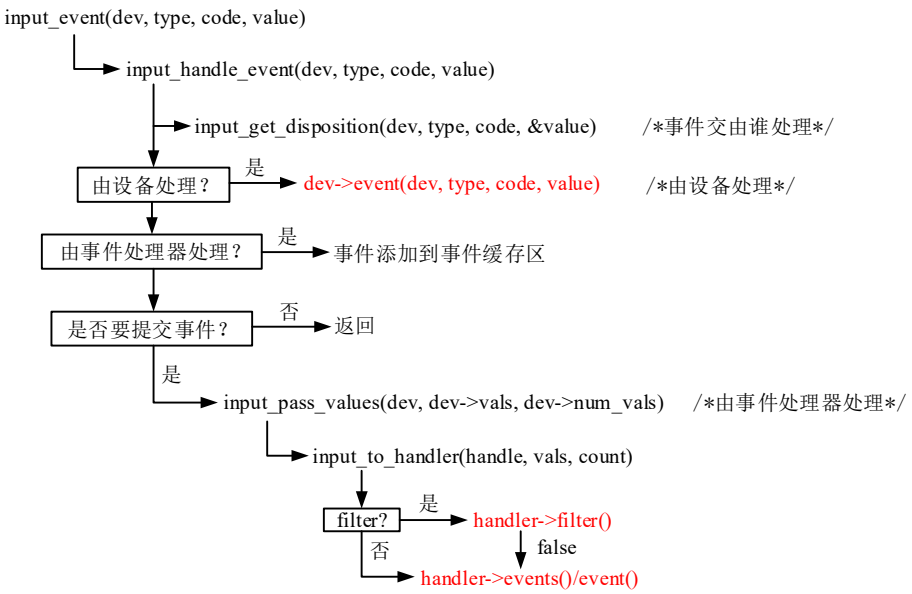
input_to_handler()函数负责将设备缓存事件提交给事件处理器处理，函数定义如下：

```
static unsigned int input_to_handler(struct input_handle *handle,struct input_value *vals, unsigned int count)
{
    struct input_handler *handler = handle->handler;     /*事件处理器*/
    struct input_value *end = vals;
    struct input_value *v;

    /*扫描事件列表，filter()函数处理返回值为 true 的删除，留下需要继续处理的在列表中*/
    if (handler->filter) {
        for (v = vals; v != vals + count; v++) {
            if (handler->filter(handle, v->type, v->code, v->value))
                continue;
```

```
                if (end != v)
                    *end = *v;     /*留下需要继续处理的事件*/
                end++;
            }
            count = end - vals;
        }

        if (!count)
            return 0;

        /*有需要由事件处理器继续处理的事件*/
        if (handler->events)          /*同时处理多个事件*/
            handler->events(handle, vals, count);
        else if (handler->event)     /*逐个处理事件*/
            for (v = vals; v != vals + count; v++)
                handler->event(handle, v->type, v->code, v->value);
        return count;       /*事件处理器处理事件的数量*/
    }
```

input_to_handler()函数内扫描输入设备 input_value 数组，对每个数组项调用 filter()函数，如果函数返回值为非零，则表示事件（数组项）已由 filter()函数处理，不需要再往下传递，如果函数返回值为 0 则数组项需要留在数组内，往下交由 events()/event()函数处理。

由上可知，报告事件函数执行流程简列如下所示：



2 接口函数

内核定义了许多封装 input_event()函数的报告事件 API 函数，用于报告不同类型的事件，供设备驱动程序调用，相关 API 函数定义在/include/linux/input.h 头文件，例如：

```
static inline void input_sync(struct input_dev *dev)
{                                   /*同步事件，立即处理缓存区中所有事件*/
```

```
    input_event(dev, EV_SYN, SYN_REPORT, 0);
}

static inline void input_report_key(struct input_dev *dev, unsigned int code, int value)
{                                       /*报告按键值*/
    input_event(dev, EV_KEY, code, !!value);
}

static inline void input_report_rel(struct input_dev *dev, unsigned int code, int value)
{                                           /*报告相对值*/
    input_event(dev, EV_REL, code, value);
}

static inline void input_report_abs(struct input_dev *dev, unsigned int code, int value)
{                                           /*报告绝对值*/
    input_event(dev, EV_ABS, code, value);
}

static inline void input_report_ff_status(struct input_dev *dev, unsigned int code, int value)
{
    input_event(dev, EV_FF_STATUS, code, value);
}

static inline void input_report_switch(struct input_dev *dev, unsigned int code, int value)
{                                           /*报告开关量*/
    input_event(dev, EV_SW, code, !!value);
}
static inline void input_mt_sync(struct input_dev *dev)
{
    input_event(dev, EV_SYN, SYN_MT_REPORT, 0);
}
```

由前面介绍报告事件函数可知，事件报告后不一定马上就能被处理，可能只是保存在设备事件缓存区中，如果再调用 **input_sync**(struct input_dev *dev)函数同步事件，则会立即对事件进行处理。所以，在设备驱动程序中报告完事件后，通常紧接着调用 input_sync()函数，以便能及时对事件进行处理。

9.6.5 通用事件处理器

通常每种类型的输入设备有其对应的事件处理器 input_handler 实例，用于处理设备事件，例如：用于鼠标设备的事件处理器 mousedev_handler 实例。另外，内核定义了适用于所有输入设备的 evdev_handler 事件处理器实例。

接口函数 input_register_handler()用于向内核注册 input_handler 实例，在注册 input_handler 实例时将会查找匹配的 input_dev 实例，匹配成功将调用事件处理器的 connect()函数。

本小节介绍通用事件处理器 evdev_handler 实例的实现。

1 概述

内核在/drivers/input/evdev.c 文件内实现了通用事件处理器 evdev_handler 实例（需选择 INPUT_EVDEV 配置选项）。通用事件处理器框架如下图所示，它可以匹配所有的输入设备，匹配成功则调用实例的 connect()函数。

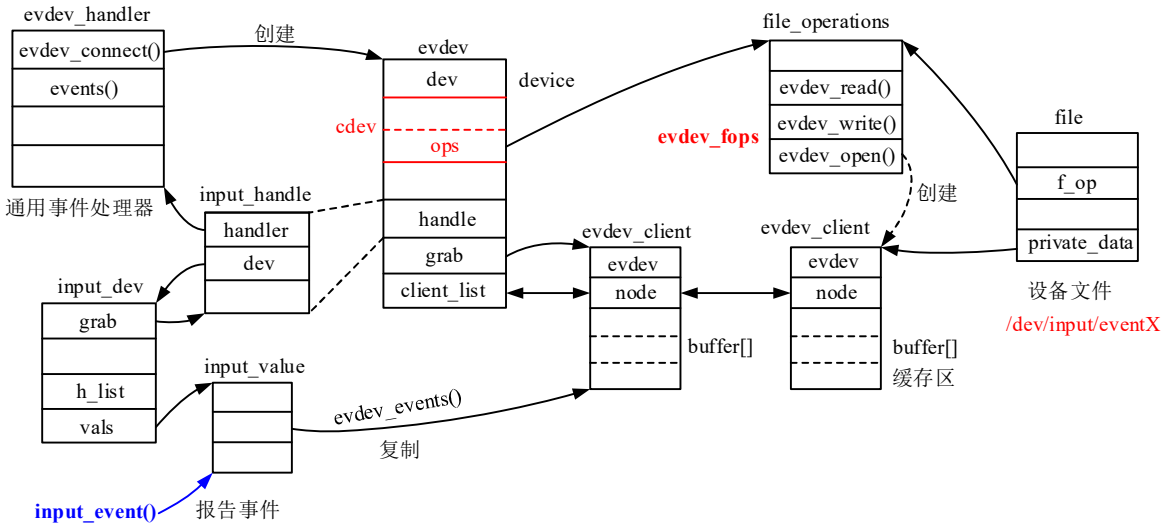
connect()函数内将为输入设备创建对应的 evdev 结构体实例，evdev 结构体中内嵌表示字符设备的 cdev 结构体以及 input_handle 结构体等成员，其中 cdev 实例关联的文件操作 file_operations 实例为 **evdev_fops**，适用于对所有输入设备的操作。evdev 结构体中内嵌 device 结构体成员，用于导出到通用驱动模型和创建设备文件等。

在打开输入设备文件，调用 evdev_fops->open()函数时，将会为设备文件创建 evdev_client 结构体实例，它是特定于设备文件的私有数据结构。同一个设备文件可同时被多个进程打开，因此设备文件可能对应多个 evdev_client 实例（打开一次创建一个）。

evdev 结构体中 grab 成员指向的 evdev_client 实例表示当前独占式地处理设备提交的事件，即此时设备事件只由此 evdev_client 实例对应的设备文件接收处理。

evdev_client 结构体中包含事件缓存区，事件处理器的 events()/event()函数负责将输入设备事件缓存区的事件复制到 evdev_client 结构体的事件缓存区。evdev_fops 实例中的读写函数即是对 evdev_client 结构体中事件缓存区的读写。

内核在/drivers/input/evdev.c 文件内定义了 evdev_init()初始化函数，在加载模块或内核启动阶段调用，用于向内核注册 evdev_handler 实例。



2 事件处理器实例

通用事件处理器实例 evdev_handler 定义如下（/drivers/input/evdev.c）：

```
static struct input_handler evdev_handler = {
    .event          = evdev_event,          /*单个事件处理函数，调用 evdev_events()*/
    .events         = evdev_events,         /*多个事件处理函数*/
    .connect        = evdev_connect,        /*连接函数（与输入设备匹配时调用）*/
    .disconnect     = evdev_disconnect,
    .legacy_minors      = true,
    .minor          = EVDEV_MINOR_BASE,     /*64，从设备号起于 64，最大数量 32*/
    .name           = "evdev",
```

```
    .id_table = evdev_ids,                    /*input_device_id 列表，可匹配任何设备*/
};
```

evdev_handler 实例中没有定义 match()函数，input_device_id 列表定义如下：

```
static const struct input_device_id evdev_ids[] = {
    { .driver_info = 1 },    /*可匹配所有设备*/
    { },                     /* Terminating zero entry */
};
```

列表中只有一项，input_device_id 实例 flags 成员和各事件位图成员都为 0，driver_info 成员值为 1，且事件处理器没有定义 match()函数，由前面的匹配函数可知，此 input_device_id 实例能匹配所有的 input_dev 实例。

■数据结构

在介绍各函数的实现前，先介绍几个相关数据结构的定义。

在 evdev_handler 实例的连接函数 evdev_connect()内将为输入设备创建 **evdev** 结构体实例，结构体定义在/drivers/input/evdev.c 文件内：

```
struct evdev {
    int open;                          /*设备文件打开次数*/
    struct input_handle    handle;     /*input_handle 实例，连接输入设备和事件处理器*/
    wait_queue_head_t    wait;         /*等待进程队列，等待输入设备事件的进程*/
    struct evdev_client __rcu *grab;   /*当前 evdev_client 实例指针*/
    struct list_head client_list;      /*evdev_client 实例双链表*/
    spinlock_t client_lock;            /*保护双链表 client_list 的自旋锁*/
    struct mutex mutex;
    struct device dev;                 /*device 实例，导出到 sysfs，用于创建设备文件*/
    struct cdev cdev;                  /*表示字符设备的 cdev 实例*/
    bool exist;                        /*是否存在关联设备*/
};
```

evdev 结构体主要成员简介如下：

●**wait**：等待队列头。

●**cdev**：表示字符设备的 cdev 实例，添加到字符设备驱动数据库。

●**dev**： device 结构体成员，在通用驱动模型中表示输入设备。

●**handle**：input_handle 结构体成员，用于建立 input_dev 与 input_handler 实例之间关联，见下文。

●**grab**：指向当前 evdev_client 结构体实例，主要用于事件处理，见下文。

●**client_list**：evdev_client 实例双链表头。

（1）input_handle

input_handle 结构体用于建立 input_dev 实例与 input_handler 实例之间的关联。一个 input_dev 实例可通过多个 input_handle 实例（组成双链表）关联到多个 input_handler 实例，也说是用户进程可通过多个接口（设备文件）操作同一个输入设备（input_handler 实例就是用户进程操作输入设备的接口）。

某一时刻设备事件将提交给 input_dev.grab 指向的 input_handle 实例关联的 input_handler 实例处理，如果 input_dev.grab 为 NULL，则设备事件将提交给输入设备关联的所有 input_handle 实例处理。

input_handle 结构体定义如下（/include/linux/input.h）：

```
struct input_handle {
    void *private;      /*私有数据，指向 evdev 实例*/
    int open;           /*打开次数*/
    const char *name;
    struct input_dev *dev;              /*关联的 input_dev 实例*/
    struct input_handler *handler;      /*关联的 input_handler 实例*/
    struct list_head    d_node;     /*将实例链接到 input_dev.h_list 双链表*/
    struct list_head    h_node;     /*将实例链接到 input_handler.h_list 双链表*/
};
```

evdev 结构体内嵌了 input_handle 结构体成员。

input_register_handle()函数用于注册 input_handle 实例，input_handle 注册前需要关联 input_dev 和 input_handler 实例，注册函数定义如下：

```
int input_register_handle(struct input_handle *handle)
{
    struct input_handler *handler = handle->handler;
    struct input_dev *dev = handle->dev;
    int error;

    error = mutex_lock_interruptible(&dev->mutex);
    if (error)
        return error;
    if (handler->filter)        /*将 input_handle 实例添加到 input_dev 实例中双链表*/
        list_add_rcu(&handle->d_node, &dev->h_list);    /*添加到双链表头部*/
    else
        list_add_tail_rcu(&handle->d_node, &dev->h_list);    /*添加到双链表末尾*/

    mutex_unlock(&dev->mutex);

    list_add_tail_rcu(&handle->h_node, &handler->h_list); /*添加到 input_handler 实例双链表末尾*/

    if (handler->start)
        handler->start(handle);     /*调用事件处理器的 start()函数*/
    return 0;
}
```

注册函数将 input_handle 实例分别插入到 input_dev 和 input_handler 实例中的双链表，调用 input_handler 实例内的 start()函数。

（2）evdev_client

evdev_client 结构体主要用于保存输入设备事件，用于建立设备事件与打开设备文件之间的关联。由于同一个设备文件可被多个进程打开，每打开一次就为进程文件创建一个 evdev_client 实例，evdev 结构体通过双链表管理 evdev_client 实例。

evdev 结构体中 grab 成员指向的 evdev_client 表示当前接收设备事件实例，也就是建立设备与某个打

开进程文件独占式的关联，设备事件由此关联进程文件接收处理，其它打开进程（设备）文件不接收事件。用户进程通过 ioctl()系统调用建立与设备的独占式的关联（见下文）。

evdev_client 结构体定义如下（/drivers/input/evdev.c）：

```
struct evdev_client {
    unsigned int    head;              /*事件缓存数组起始索引*/
    unsigned int    tail;              /*事件缓存数组结束索引*/
    unsigned int    packet_head;     /* [future] position of the first element of next packet */
    spinlock_t    buffer_lock;       /* protects access to buffer, head and tail */
    struct fasync_struct *fasync;
    struct evdev *evdev;             /*指向 evdev 实例*/
    struct list_head    node;        /*链接到 evdev.client_list 双链表*/
    int    clk_type;
    bool    revoked;
    unsigned int    bufsize;              /*buffer[]缓存区大小*/
    struct input_event    buffer[];     /*事件缓存区*/
};
```

evdev_client 结构体中的事件缓存区是 input_event 结构体数组，结构体定义在/include/uapi/linux/input.h 头文件：

```
struct input_event {
    struct timeval time;     /*时间戳*/
    __u16 type;
    __u16 code;
    __s32 value;
};
```

输入设备提交事件的形式为 input_value 结构体，通用事件处理器的事件处理函数 events()/event()需要将 input_value 形式的事件转换成 input_event 结构体形式并写入 evdev_client 结构体事件缓存区。

■连接函数

事件处理器的连接函数在 input_dev 与 input_handler 匹配时调用。

下面来看一下 evdev_handler 实例中连接函数 **evdev_connect()**的实现（/drivers/input/evdev.c）：

```
static int evdev_connect(struct input_handler *handler, struct input_dev *dev,   \
                                                        const struct input_device_id *id)
{
    struct evdev *evdev;
    int minor;
    int dev_no;
    int error;

    minor = input_get_new_minor(EVDEV_MINOR_BASE, EVDEV_MINORS, true);
                                /*分配从设备号，输入设备从设备号由 ida 结构管理*/
    if (minor < 0) {
        ...
```

```
}
evdev = kzalloc(sizeof(struct evdev), GFP_KERNEL);     /*创建 evdev 实例*/
if (!evdev) {
        ...
}

INIT_LIST_HEAD(&evdev->client_list);       /*初始化双链表头成员*/
spin_lock_init(&evdev->client_lock);
mutex_init(&evdev->mutex);
init_waitqueue_head(&evdev->wait);       /*初始化等待队列*/
evdev->exist = true;

dev_no = minor;     /*从设备号*/
/* Normalize device number if it falls into legacy range */
if (dev_no < EVDEV_MINOR_BASE + EVDEV_MINORS)
        dev_no -= EVDEV_MINOR_BASE;
dev_set_name(&evdev->dev, "event%d", dev_no); /*设置设备名称为 eventX，X 为从设备号*/

evdev->handle.dev = input_get_device(dev);
evdev->handle.name = dev_name(&evdev->dev);
evdev->handle.handler = handler;
evdev->handle.private = evdev;            /*私有数据*/

evdev->dev.devt = MKDEV(INPUT_MAJOR, minor);   /*设备号，INPUT_MAJOR 为主设备号*/
evdev->dev.class = &input_class;         /*设备类定义了设置设备文件名称的函数*/
                                         /*设备文件名为：input/dev_name(dev)*/
evdev->dev.parent = &dev->dev;
evdev->dev.release = evdev_free;
device_initialize(&evdev->dev);          /*初始化 device 实例*/

error = input_register_handle(&evdev->handle);   /*注册 input_handle 实例*/
if (error)
        goto err_free_evdev;

cdev_init(&evdev->cdev, &evdev_fops);          /*文件操作结构实例为 evdev_fops*/
evdev->cdev.kobj.parent = &evdev->dev.kobj;
error = cdev_add(&evdev->cdev, evdev->dev.devt, 1);   /*注册字符设备驱动*/
if (error)
        goto err_unregister_handle;

error = device_add(&evdev->dev);   /*添加设备，将创建设备文件，文件名/dev/input/eventX*/
if (error)
        goto err_cleanup_evdev;
```

```
    return 0;
    ...
}
```

通用事件处理器匹配的输入设备起始从设备号为 EVDEV_MINOR_BASE（64），从设备号数量为 EVDEV_MINORS（32）。

连接函数主要完成的工作是：为设备分配从设备号，创建 evdev 实例并初始化，注册 input_handle 实例，设置并添加 cdev 实例（文件操作结构实例为 **evdev_fops**），添加设备并创建设备文件等。

■事件处理函数

evdev_handler 实例事件处理函数为 **evdev_events()**，在报告事件 input_event()函数中调用，定义如下：

```
static void evdev_events(struct input_handle *handle,const struct input_value *vals, unsigned int count)
{
    struct evdev *evdev = handle->private;     /*指向 evdev 实例*/
    struct evdev_client *client;
    ktime_t ev_time[EV_CLK_MAX];

    ev_time[EV_CLK_MONO] = ktime_get();
    ev_time[EV_CLK_REAL] = ktime_mono_to_real(ev_time[EV_CLK_MONO]);
    ev_time[EV_CLK_BOOT] = ktime_mono_to_any(ev_time[EV_CLK_MONO],TK_OFFS_BOOT);

    rcu_read_lock();
    client = rcu_dereference(evdev->grab);     /*当前独占的 evdev_client 实例*/
    if (client)
        evdev_pass_values(client, vals, count, ev_time);
                    /*将事件信息提交到当前 evdev_client，并唤醒等待进程*/
    else
        list_for_each_entry_rcu(client, &evdev->client_list, node)    /*传递给所有 evdev_client*/
            evdev_pass_values(client, vals, count, ev_time);

    rcu_read_unlock();
}
```

evdev_events()函数调用 evdev_pass_values()函数将传递的输入设备事件填充到关联的 evdev_client 实例 input_event 缓存数组，并唤醒在 evdev 上等待的进程。

evdev_handler 实例中单个事件处理函数 evdev_event()内部调用 evdev_events()函数执行，只不过将参数事件数量 count 设为 1。

evdev_pass_values()函数定义如下：

```
static void evdev_pass_values(struct evdev_client *client,     \
                const struct input_value *vals, unsigned int count,ktime_t *ev_time)
{
    struct evdev *evdev = client->evdev;
    const struct input_value *v;
    struct input_event event;
```

```
        bool wakeup = false;

        if (client->revoked)
                return;

        event.time = ktime_to_timeval(ev_time[client->clk_type]);   /*时间戳*/

        spin_lock(&client->buffer_lock);

        for (v = vals; v != vals + count; v++) {     /*复制事件信息*/
                event.type = v->type;
                event.code = v->code;
                event.value = v->value;
                __pass_event(client, &event);            /*传给 evdev_client 实例*/
                if (v->type == EV_SYN && v->code == SYN_REPORT)
                        wakeup = true;
        }

        spin_unlock(&client->buffer_lock);

        if (wakeup)
                wake_up_interruptible(&evdev->wait);     /*唤醒等待事件的进程*/
}
```

3 设备操作

通用事件处理器可匹配所有的输入设备，并创建/dev/input/eventX 设备文件，用户进程可通过此设备文件操作输入设备，对应的文件操作结构实例为 evdev_fops，定义如下：

```
static const struct file_operations evdev_fops = {
    .owner      = THIS_MODULE,
    .read       = evdev_read,          /*读操作*/
    .write      = evdev_write,         /*写操作*/
    .poll       = evdev_poll,
    .open       = evdev_open,          /*打开设备*/
    .release    = evdev_release,       /*释放设备函数*/
    .unlocked_ioctl = evdev_ioctl,     /*设备控制函数*/
#ifdef CONFIG_COMPAT
    .compat_ioctl   = evdev_ioctl_compat,
#endif
    .fasync     = evdev_fasync,
    .flush      = evdev_flush,
    .llseek     = no_llseek,
};
```

下面简单看一下 evdev_fops 实例中打开函数、读写函数以及设备控制函数的实现。

■**打开操作**

打开操作函数 evdev_open()在打开设备文件的 open()系统调用中被调用，函数定义如下：

```
static int evdev_open(struct inode *inode, struct file *file)
{
    struct evdev *evdev = container_of(inode->i_cdev, struct evdev, cdev);
    unsigned int bufsize = evdev_compute_buffer_size(evdev->handle.dev);   /*计算事件缓存区大小*/
    unsigned int size = sizeof(struct evdev_client) +bufsize * sizeof(struct input_event);
                                                                    /*evdev_client 实例大小*/
    struct evdev_client *client;
    int error;

    client = kzalloc(size, GFP_KERNEL | __GFP_NOWARN);      /*创建 evdev_client 实例*/
    ...
    client->bufsize = bufsize;
    spin_lock_init(&client->buffer_lock);
    client->evdev = evdev;          /*指向 evdev 实例*/
    evdev_attach_client(evdev, client);      /*evdev_client 添加到 evdev.client_list 双链表末尾*/

    error = evdev_open_device(evdev);     /*调用 input_dev.open()函数打开设备（硬件操作）*/
    if (error)
        goto err_free_client;

    file->private_data = client;        /*建立与 file 实例与 evdev_client 实例关联*/
    nonseekable_open(inode, file);
    return 0;
    ...
}
```

打开操作函数主要完成的工作是创建 evdev_client 实例，建立其与 evdev 实例和进程文件 file 实例之间的关联，调用 input_dev 实例中定义的 open()函数（执行硬件激活操作等）。每次打开设备文件时都会创建一个 evdev_client 实例（设备文件可同时被多个进程打开），但是只有第一次打开设备文件时才会调用输入设备 input_dev 实例中的 open()函数。

■**读操作**

输入设备文件读操作函数为 evdev_read()，定义如下：

```
static ssize_t evdev_read(struct file *file, char __user *buffer,size_t count, loff_t *ppos)
{
    struct evdev_client *client = file->private_data;
    struct evdev *evdev = client->evdev;
    struct input_event event;
    size_t read = 0;
    int error;
```

```
    if (count != 0 && count < input_event_size())
        return -EINVAL;

    for (;;) {
        if (!evdev->exist || client->revoked)
            return -ENODEV;

        if (client->packet_head == client->tail &&(file->f_flags & O_NONBLOCK))
            return -EAGAIN;         /*非阻塞操作且事件缓存为空，返回错误码*/

        if (count == 0)
            break;

        while (read + input_event_size() <= count &&evdev_fetch_next_event(client, &event)) {
            if (input_event_to_user(buffer + read, &event))          /*/drivers/input/input-compat.c*/
                return -EFAULT;
            read += input_event_size();     /*将 evdev_client 内 buffer[]缓存复制到用户空间*/
        }
        if (read)       /*读到了数据，跳出循环，函数返回*/
            break;
        if (!(file->f_flags & O_NONBLOCK)) {    /*没读到数据且没有设置非阻塞操作*/
            error = wait_event_interruptible(evdev->wait,client->packet_head != client->tail ||   \
                                             !evdev->exist || client->revoked);        /*加入等待队列，等待*/
            if (error)
                return error;
        }
    }       /*循环结束*/
    return read;
}
```

读操作函数根据 count 值，以 input_event 结构体大小为单位，将 evdev_client 内 buffer[]缓存区内容复制到用户空间 buffer 地址。如果读操作没有指定 O_NONBLOCK 标记，则当缓存区为空时，用户进程进入睡眠等待，直到设备提交事件时将其唤醒。

读操作函数只是将 input_event 数组数据返回给用户进程，致以如何处理这些事件，做出何种反应，将由用户进程决定。

■写操作

对输入设备文件进行写操作，相当于在用户空间向核心层报告事件，可理解成模拟设备驱动层报告事件。输入设备文件写操作函数为 evdev_write()，定义如下：

```
static ssize_t evdev_write(struct file *file, const char __user *buffer,size_t count, loff_t *ppos)
{
    struct evdev_client *client = file->private_data;
    struct evdev *evdev = client->evdev;
    struct input_event event;     /*input_event 实例，缓存数据*/
```

```
    int retval = 0;

    if (count != 0 && count < input_event_size())
        return -EINVAL;

    retval = mutex_lock_interruptible(&evdev->mutex);
    if (retval)
        return retval;

    if (!evdev->exist || client->revoked) {
        retval = -ENODEV;
        goto out;
    }

    while (retval + input_event_size() <= count) {    /*循环开始*/
        if (input_event_from_user(buffer + retval, &event)) {    /*复制数据至 event 实例*/
            retval = -EFAULT;
            goto out;
        }
        retval += input_event_size();
        input_inject_event(&evdev->handle,event.type, event.code, event.value);
                                /*调用 input_handle_event()函数处理，报告事件*/
    }    /*循环结束*/
 out:
    mutex_unlock(&evdev->mutex);
    return retval;
}
```

写操作函数将用户空间 buffer 指向缓存区的数据以 input_event 实例的形式逐个复制到 event 实例中，并调用 input_inject_event()函数处理事件，函数内调用 input_handle_event()函数模拟报告设备事件。

■设备控制

用户进程可通过 ioctl()系统调用对设备发送命令，对设备（文件）进行控制，系统调用最终会调用文件操作结构中的 unlocked_ioctl()函数，执行命令。输入设备文件操作结构实例中此函数为 evdev_ioctl()，函数内最终调用 **evdev_do_ioctl()**函数对输入命令进行处理。

evdev_do_ioctl()函数充当分配器的角色，根据不同的命令调用不同的处理函数，源代码请读者自行阅读。内核在/include/uapi/linux/input.h 头文件定义了命令取值，例如：

```
#define    EVIOCGVERSION            _IOR('E', 0x01, int)        /* get driver version */
#define    EVIOCGID          _IOR('E', 0x02, struct input_id)    /* get device ID */
#define    EVIOCGREP         _IOR('E', 0x03, unsigned int[2])    /* get repeat settings */
#define    EVIOCSREP         _IOW('E', 0x03, unsigned int[2])   /* set repeat settings */

#define    EVIOCGKEYCODE             _IOR('E', 0x04, unsigned int[2])            /* get keycode */
#define    EVIOCGKEYCODE_V2          _IOR('E', 0x04, struct input_keymap_entry)
```

```
#define   EVIOCSKEYCODE          _IOW('E', 0x04, unsigned int[2])        /* set keycode */
#define   EVIOCSKEYCODE_V2       _IOW('E', 0x04, struct input_keymap_entry)

#define   EVIOCGNAME(len)    _IOC(_IOC_READ, 'E', 0x06, len)          /* get device name */
#define   EVIOCGPHYS(len)    _IOC(_IOC_READ, 'E', 0x07, len)          /* get physical location */
#define   EVIOCGUNIQ(len)    _IOC(_IOC_READ, 'E', 0x08, len)      /* get unique identifier */
#define   EVIOCGPROP(len)    _IOC(_IOC_READ, 'E', 0x09, len)     /* get device properties */

#define   EVIOCGMTSLOTS(len)     _IOC(_IOC_READ, 'E', 0x0a, len)

#define   EVIOCGKEY(len)     _IOC(_IOC_READ, 'E', 0x18, len)          /* get global key state */
#define   EVIOCGLED(len)     _IOC(_IOC_READ, 'E', 0x19, len)          /* get all LEDs */
#define   EVIOCGSND(len)     _IOC(_IOC_READ, 'E', 0x1a, len)      /* get all sounds status */
#define   EVIOCGSW(len)      _IOC(_IOC_READ, 'E', 0x1b, len)          /* get all switch states */

#define   EVIOCGBIT(ev,len)      _IOC(_IOC_READ, 'E', 0x20 + (ev), len) /* get event bits */
#define   EVIOCGABS(abs)     _IOR('E', 0x40 + (abs), struct input_absinfo)/* get abs value/limits */
#define   EVIOCSABS(abs)     _IOW('E', 0xc0 + (abs), struct input_absinfo)      /* set abs value/limits */

#define   EVIOCSFF     _IOC(_IOC_WRITE, 'E', 0x80, sizeof(struct ff_effect))
                                                  /* send a force effect to a force feedback device */
#define   EVIOCRMFF        _IOW('E', 0x81, int)                 /* Erase a force effect */
#define   EVIOCGEFFECTS        _IOR('E', 0x84, int)
                                       /* Report number of effects playable at the same time */
#define   EVIOCGRAB          _IOW('E', 0x90, int)                  /*进程要独占设备*/
#define   EVIOCREVOKE        _IOW('E', 0x91, int)                  /* Revoke device access */
#define   EVIOCSCLOCKID      _IOW('E', 0xa0, int)
```

_IOR()和_IOW()等宏定义在/include/uapi/asm-generic/ioctl.h 头文件，用于将参数组合成命令。各种命令的处理函数请读者自行阅读内核源代码。

9.6.6 触摸屏驱动程序

四线式电阻式触摸屏是嵌入式设备中广泛使用的触摸屏，它是一种实际的输入设备。本小节先介绍电阻式触摸屏及其控制器的基本原理，再介绍触摸屏设备驱动程序的实现。

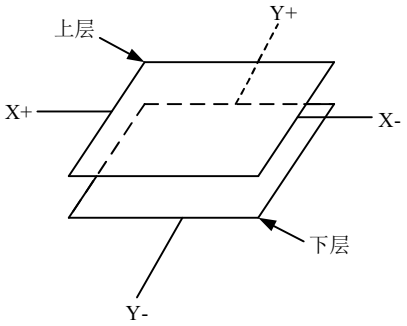
触摸屏驱动采用输入设备驱动框架。触摸屏控制器在感知到触摸屏被按下时，产生中断通知处理器，在中断处理程序中向控制器发出测量按点坐标值的命令，在收到坐标值后调用输入设备驱动的事件报告函数，向输入设备核心层报告事件，事件处理器将事件传递给用户进程，最终由用户进程对事件作出响应。

1 电阻屏控制器

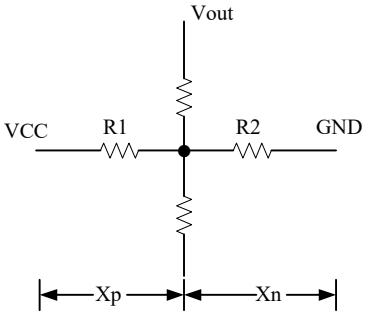
通常我们说的触摸屏与显示屏在电气上是完全隔离的器件，只不过触摸屏是贴在显示屏上面的，其实触摸屏相当于键盘，用于感知输入事件，显示屏负责显示。

嵌入式系统中常用的四线式电阻式触摸屏结构如下图所示，它由两层隔离的导电薄膜组成，贴在显示

屏上面。当触摸屏没有被按下时，上下两层导电层是绝缘隔离的。当某个位置被按下时，在按下点上下两层导电层接触导通，此时在上层 X+与 X-之间加上电压，从 Y+（或 Y-）测量电压（测量设备输入阻抗要大），由于导电薄层材质是均匀的，电阻分布也是均匀的，由测量的电压值和加在 X+与 X-之间的电压即可计算得按下点在 X 轴上的相对位置。同理，在 Y+与 Y-之间加上电压，测量 X+（X-）的电压值，可得 Y 轴上相对位置，获取按下点坐标值需经过两次加电测量操作。

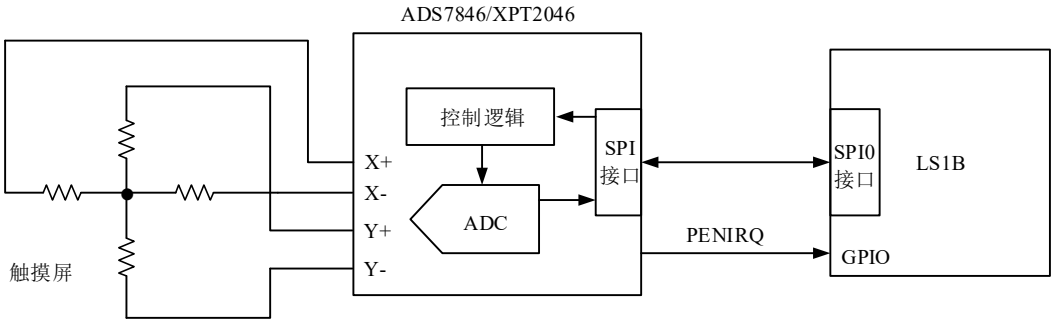


测量 X 轴坐标值的等效电路原理图如下：



由于导电薄膜电阻分布的均匀性，按点两端电阻值之间的比值 R1/R2 与按点到两端的距离比值 Xp/Xn 相等，而 R1/R2 之间的比值与电压比值(VCC-Vout)/Vout 相等，因此由电压值可计算出按点坐标值。Y 轴坐标值的测量原理相同。

电阻触摸屏有专门的控制器，例如：TI 公司的 ADS7846，国产的 XPT2046 芯片，龙芯 1B 开发板触摸屏采用的是 XPT2046 芯片，由于其与 ADS7846 兼容，所以本节介绍的是 ADS7846 芯片的驱动代码，配置时需选择 TOUCHSCREEN_ADS7846 选项。控制芯片原理框图如下图所示，详细信息请参考芯片数据手册。



控制器通过 SPI 总线与龙芯 1B（SPI0）连接，通过总线传输数据，当触摸屏被按下时，将产生笔中断（PENIRQ）通过 GPIO 传送到处理器，中断处理程序中通过 SPI 接口向控制器发送测量坐标值的命令，控制器收到命令后产生控制逻辑，给触摸屏某一导电层加电，并从另一层取电压值，送入 ADC 转换器，将电压值转换成数字量，通过 SPI 接口发送给处理器。获取一个坐标点值需要发送两个命令，分别是读取 X 轴坐标值和 Y 轴坐标值。控制器除了可获取坐标值外，还可以获取按键压力值、温度等传感器量，具体信息请读者参数芯片数据手册。

主机通过发送命令的方式与控制器进行数据传输，下面来看一下命令字的格式：

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
S	A2	A1	A0	MODE	SER/$\overline{\text{DFR}}$	PD1	PD0

●S：起始位，置 1 表示数据传输开始，没有输入数据时，输入线置 0；

●A2-A0：选择 ADC 转换器输入信号，测 X 轴坐标值时取值为 101，测 Y 轴坐标值时取值为 001；

●MODE：ADC 转换器 12 位/8 位转换模式，0 表示 12 位，1 表示 8 位；

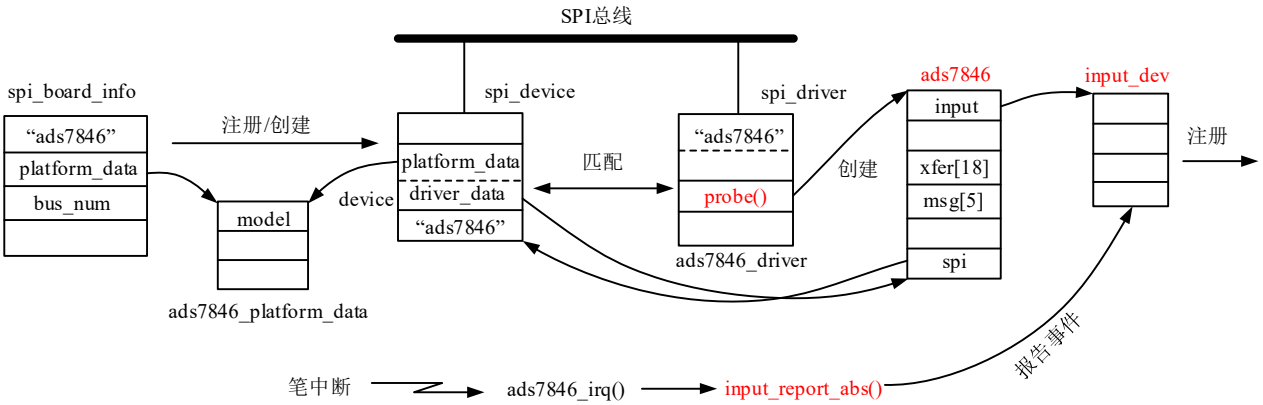
●SER/DFR：ADC 工作模式，单端模式（1），差分模式（0），坐标转换时选择差分模式；

●PD1，PD0：选择节能模式，00 表示在两次转换之间自动进入节能模式，笔中断使能，01 表示内部参考电压关闭，ADC 打开，笔中断关闭，10 表示内部参考电压打开，ADC 关闭，笔中断打开，11 表示器件始终处于上电状态，参考电压和 ADC 打开，笔中断关闭。

触摸屏驱动程序在笔中断处理程序中发送命令读取 X 轴坐标值，再发送一个命令读取 Y 轴坐标值，然后向输入设备驱动核心层报告事件即可。读取操作的时序请读者查看芯片数据手册。

2 驱动框架

触摸屏驱动框架如下图所示，控制器通过 SPI 总线与处理器连接，因此控制器由 spi_device 实例表示。第 8 章介绍 SPI 总线时我们知道，spi_device 是由内核动态创建的，板级文件需要定义 spi_board_info 结构实例，并向内核注册。在注册 spi_board_info 时会创建并注册 spi_device 实例，设备注册后会扫描 SPI 总线驱动链表，查找匹配的驱动（名称匹配），匹配上之后调用 spi_driver 实例定义的 probe()函数，函数内创建表示控制器 ads7846 结构体实例，内含输入设备 input_dev 实例。



spi_board_info 结构表示的板级 SPI 总线的物理特性，ads7846_platform_data 结构体表示控制器的物理特性，这两个结构实例都定义在板级相关的文件内。

spi_driver 结构体实例的 probe()函数内除了创建控制器驱动相关的结构实例外，还有一个重要的任务就是注册笔中断处理函数，中断处理函数中调用输入设备驱动的 API 函数向内核报告绝对值事件，事件由通用事件处理器处理，前面已经介绍过了。

3 驱动实现

驱动实现主要包括触摸屏设备的注册，以及驱动的注册（SPI 总线设备与驱动）。

■**设备**

驱动框架中 ads7846_platform_data 结构体定义在/include/linux/spi/ads7846.h 头文件，用于表示触摸屏控制器的配置信息，结构体成员简列如下：

```
struct ads7846_platform_data {
    u16  model;                 /*模式 7843, 7845, 7846, 7873，表示不同的芯片*/
    u16  vref_delay_usecs;   /* 0 for external vref; etc */
    u16  vref_mv;               /* ads7846: if 0, use internal vref，使用内部参考电压  */
    bool keep_vref_on;        /*在差分模式下仍保持内部参考电压有效*/
    bool swap_xy;               /*交换 X 轴和 Y 轴从标值*/

    u16  settle_delay_usecs;
    u16  penirq_recheck_delay_usecs;   /*笔中断核查延迟时间，防止毛刺*/

    u16  x_plate_ohms;
    u16  y_plate_ohms;

    u16  x_min, x_max;        /*坐标值最大值和最小值*/
    u16  y_min, y_max;
    u16  pressure_min, pressure_max;

    u16  debounce_max;      /* */
    u16  debounce_tol;        /* tolerance used for filtering */
    u16  debounce_rep;          /* */
    int    gpio_pendown;              / *用于笔中断的 GPIO 编号*/
    int    gpio_pendown_debounce;     /**/
    int    (*get_pendown_state)(void);
    int    (*filter_init)    (const struct ads7846_platform_data *pdata,void **filter_data);
    int    (*filter)    (void *filter_data, int data_idx, int *val);
    void (*filter_cleanup)(void *filter_data);
    void (*wait_for_sync)(void);
    bool wakeup;
    unsigned long irq_flags;
};
```

spi_board_info 和 ads7846_platform_data 结构体实例在板级相关的文件内定义，龙芯 1B 开发板实例定义如下：

```
#include <linux/spi/spi_ls1x.h>
static struct spi_board_info ls1x_spi0_devices[] = {
    ...
    #ifdef CONFIG_TOUCHSCREEN_ADS7846
    {
        .modalias = "ads7846",          /*名称，匹配驱动*/
        .platform_data = &ads_info,      /*ads7846_platform_data 实例*/
```

```
        .bus_num        = 0,
        .chip_select    = SPI0_CS1,
        .max_speed_hz   = 2500000,
        .mode           = SPI_MODE_1,
        .irq            = LS1X_GPIO_FIRST_IRQ + ADS7846_GPIO_IRQ,
    },
    #endif
    ...
}
#ifdef CONFIG_TOUCHSCREEN_ADS7846
#include <linux/spi/ads7846.h>
#define   ADS7846_GPIO_IRQ   60 /*  开发板触摸屏使用的外部中断  */
static struct ads7846_platform_data ads_info __maybe_unused = {
    .model          = 7846,
    .vref_delay_usecs   = 1,
    .keep_vref_on       = 0,
    .settle_delay_usecs = 20,
    .pressure_min       = 0,
    .pressure_max       = 2048,
    .debounce_rep       = 3,
    .debounce_max       = 10,
    .debounce_tol       = 50,
    .get_pendown_state= NULL,
    .gpio_pendown     = ADS7846_GPIO_IRQ,     /*笔中断 GPIO 编号*/
    .filter_init        = NULL,
    .filter             = NULL,
    .filter_cleanup     = NULL,
};
#endif    /* TOUCHSCREEN_ADS7846 */
```

■驱动

在驱动程序中控制器由 ads7846 结构体表示，结构体定义在/drivers/input/touchscreen/ads7846.c 文件内，简列如下：

```
struct ads7846 {
    struct input_dev    *input;     /*指向 input_dev 实例*/
    char            phys[32];
    char            name[32];

    struct spi_device   *spi;     /*指向 SPI 设备*/
    struct regulator    *reg;

#if IS_ENABLED(CONFIG_HWMON)
    struct device       *hwmon;
```

```
#endif

    u16             model;
    u16             vref_mv;
    u16             vref_delay_usecs;
    u16             x_plate_ohms;
    u16             pressure_max;
    bool            swap_xy;
    bool            use_internal;
    struct ads7846_packet   *packet;    /*保存读取得坐标值*/
    struct spi_transfer   xfer[18];    /*内嵌数组，spi 串口信息传输*/
    struct spi_message  msg[5];
    int             msg_count;
    wait_queue_head_t   wait;    /*等待队列*/
    bool            pendown;
    int             read_cnt;
    int             read_rep;
    int             last_read;
    u16             debounce_max;
    u16             debounce_tol;
    u16             debounce_rep;
    u16             penirq_recheck_delay_usecs;
    struct mutex            lock;
    bool            stopped;  /* P: lock */
    bool            disabled; /* P: lock */
    bool            suspended;     /* P: lock */

    int             (*filter)(void *data, int data_idx, int *val);
    void            *filter_data;
    void            (*filter_cleanup)(void *data);
    int             (*get_pendown_state)(void);
    int             gpio_pendown;
    void            (*wait_for_sync)(void);
};
```

触摸屏控制器驱动 spi_driver 实例定义如下（/drivers/input/touchscreen/ads7846.c）：

```
static struct spi_driver ads7846_driver = {
    .driver = {
        .name   = "ads7846",        /*名称，用于匹配 spi_device*/
        .owner  = THIS_MODULE,
        .pm  = &ads7846_pm,
        .of_match_table = of_match_ptr(ads7846_dt_ids),
    },
```

```
    .probe          = ads7846_probe,      /*探测函数*/
    .remove         = ads7846_remove,
};
```

驱动探测函数 **ads7846_probe()**源代码简列如下（/drivers/input/touchscreen/ads7846.c）：

```
static int ads7846_probe(struct spi_device *spi)
{
    const struct ads7846_platform_data *pdata;
    struct ads7846 *ts;
    struct ads7846_packet *packet;
    struct input_dev *input_dev;
    unsigned long irq_flags;
    int err;
    ...
    /*不能超过最大采样速率*/
    if (spi->max_speed_hz > (125000 * SAMPLE_BITS)) {
        ...
    }

    spi->bits_per_word = 8;     /*设置 spi_device 实例*/
    spi->mode = SPI_MODE_0;
    err = spi_setup(spi);
    if (err < 0)
        return err;

    ts = kzalloc(sizeof(struct ads7846), GFP_KERNEL);       /*分配 ads7846 实例*/
    packet = kzalloc(sizeof(struct ads7846_packet), GFP_KERNEL);
    input_dev = input_allocate_device();     /*分配 input_dev 实例*/
    ...

    spi_set_drvdata(spi, ts);     /*spi_device.dev.driver_data=ts（ads7846 实例）*/

    ts->packet = packet;
    ts->spi = spi;
    ts->input = input_dev;

    mutex_init(&ts->lock);
    init_waitqueue_head(&ts->wait);

    pdata = dev_get_platdata(&spi->dev);     /*获取 ads7846_platform_data 实例指针*/
    ...
    ts->model = pdata->model ? : 7846;            /*硬件参数值传递*/
    ts->vref_delay_usecs = pdata->vref_delay_usecs ? : 100;
```

```
ts->x_plate_ohms = pdata->x_plate_ohms ? : 400;
ts->pressure_max = pdata->pressure_max ? : ~0;

ts->vref_mv = pdata->vref_mv;
ts->swap_xy = pdata->swap_xy;

if (pdata->filter != NULL) {
        if (pdata->filter_init != NULL) {
                err = pdata->filter_init(pdata, &ts->filter_data);
                if (err < 0)
                        goto err_free_mem;
        }
        ts->filter = pdata->filter;
        ts->filter_cleanup = pdata->filter_cleanup;
  } else if (pdata->debounce_max) {
        ts->debounce_max = pdata->debounce_max;
        if (ts->debounce_max < 2)
                ts->debounce_max = 2;
        ts->debounce_tol = pdata->debounce_tol;
        ts->debounce_rep = pdata->debounce_rep;
        ts->filter = ads7846_debounce_filter;      /*默认 filter 函数*/
        ts->filter_data = ts;
} else {
        ts->filter = ads7846_no_filter;
}

err = ads7846_setup_pendown(spi, ts, pdata);      /*设置笔中断 GPIO*/
if (err)
        goto err_cleanup_filter;

if (pdata->penirq_recheck_delay_usecs)
        ts->penirq_recheck_delay_usecs =pdata->penirq_recheck_delay_usecs;

ts->wait_for_sync = pdata->wait_for_sync ? : null_wait_for_sync;      /*空操作*/

snprintf(ts->phys, sizeof(ts->phys), "%s/input0", dev_name(&spi->dev));
snprintf(ts->name, sizeof(ts->name), "ADS%d Touchscreen", ts->model);

input_dev->name = ts->name;
input_dev->phys = ts->phys;
input_dev->dev.parent = &spi->dev;

input_dev->evbit[0] = BIT_MASK(EV_KEY) | BIT_MASK(EV_ABS);      /*输入设备支持的事件*/
```

```
input_dev->keybit[BIT_WORD(BTN_TOUCH)] = BIT_MASK(BTN_TOUCH);
input_set_abs_params(input_dev, ABS_X,     /*创建并设置 input_absinfo 数组*/
            pdata->x_min ? : 0,
            pdata->x_max ? : MAX_12BIT,      /*X 轴坐标值信息*/
            0, 0);
input_set_abs_params(input_dev, ABS_Y,     /*Y 轴坐标值信息*/
            pdata->y_min ? : 0,
            pdata->y_max ? : MAX_12BIT,
            0, 0);
input_set_abs_params(input_dev, ABS_PRESSURE,        /*压力值信息*/
            pdata->pressure_min, pdata->pressure_max, 0, 0);

ads7846_setup_spi_msg(ts, pdata);     /*设置 SPI 消息数据结构*/

ts->reg = regulator_get(&spi->dev, "vcc");
...

err = regulator_enable(ts->reg);
...

irq_flags = pdata->irq_flags ? : IRQF_TRIGGER_FALLING;     /*中断标记*/
irq_flags |= IRQF_ONESHOT;

err = request_threaded_irq(spi->irq, ads7846_hard_irq, ads7846_irq,   \     /*申请中断*/
                                        irq_flags, spi->dev.driver->name, ts);
if (err && !pdata->irq_flags) {
      dev_info(&spi->dev,"trying pin change workaround on irq %d\n", spi->irq);
      irq_flags |= IRQF_TRIGGER_RISING;
      err = request_threaded_irq(spi->irq,ads7846_hard_irq, ads7846_irq,irq_flags, \
                                                spi->dev.driver->name, ts);
}
...
err = ads784x_hwmon_register(spi, ts);
...

dev_info(&spi->dev, "touchscreen, irq %d\n", spi->irq);

if (ts->model == 7845)
      ads7845_read12_ser(&spi->dev, PWRDOWN);
else
      (void) ads7846_read12_ser(&spi->dev, READ_12BIT_SER(vaux));

err = sysfs_create_group(&spi->dev.kobj, &ads784x_attr_group);
```

```
    if (err)
        goto err_remove_hwmon;

    err = input_register_device(input_dev);     /*注册输入设备，匹配通用事件处理器*/
    if (err)
        goto err_remove_attr_group;

    device_init_wakeup(&spi->dev, pdata->wakeup);

    if (!dev_get_platdata(&spi->dev))
        devm_kfree(&spi->dev, (void *)pdata);

    return 0;
    ...
}
```

探测函数内创建了 ads7846 结构体实例，并从 ads7846_platform_data 实例获取信息填充结构成员，随后初始化 spi 信息传输的 xfer[18]和 msg[5]成员，创建并注册输入设备 input_dev 实例，申请中断，中断处理函数为 ads7846_hard_irq()，后面再介绍此函数。

探测函数还有一项重要的工作是为结构体实例分配 input_absinfo 结构体数组成员，input_absinfo 结构体用于表示最近一次设备报告坐标的数值，定义如下（/include/uapi/linux/input.h）：

```
struct input_absinfo {
    __s32 value;            /*最近一次报告的坐标值*/
    __s32 minimum;          /*坐标最小值*/
    __s32 maximum;          /*坐标最大值*/
    __s32 fuzz;             /*fuzz 值用于过滤噪声*/
    __s32 flat;
    __s32 resolution;
};
```

input_set_abs_params()函数定义在/drivers/input/input.c 文件内，函数首先判断 input_dev.absinfo 指针成员是否为空，如果为空则创建 input_absinfo 结构体数组，数组项数为 ABS_CNT，为每个绝对值类型创建 input_absinfo 实例（ABS_CNT 及绝对值类型都定义在/include/uapi/linux/input.h 头文件）。

■事件报告

触摸屏控制器的笔中断处理函数为 ads7846_hard_irq(int irq, void *handle)，参数 handle 指向 ads7846 实例，中断处理线程函数为 ads7846_irq()。在中断处理程序中将调用 ads7846_hard_irq()函数，而后在中断线程中调用 ads7846_irq()函数。

ads7846_hard_irq()函数定义如下：

```
static irqreturn_t ads7846_hard_irq(int irq, void *handle)
{
    struct ads7846 *ts = handle;
    return   get_pendown_state(ts) ? IRQ_WAKE_THREAD : IRQ_HANDLED;
}
```

```
static int get_pendown_state(struct ads7846 *ts)
{
    if (ts->get_pendown_state)
        return ts->get_pendown_state();

    return !gpio_get_value(ts->gpio_pendown);
}
```

get_pendown_state()函数返回非零值，表示有笔按下，需要唤醒中断线程，否则中断处理完成不需要唤醒中断线程。

在有笔按下时，中断处理函数中将唤醒中断线程，线程调用 ads7846_irq()函数用于测量坐标值，函数定义如下：

```
static irqreturn_t ads7846_irq(int irq, void *handle)
{
    struct ads7846 *ts = handle;

    msleep(TS_POLL_DELAY);     /*稍做延迟*/

    while (!ts->stopped && get_pendown_state(ts)) {

        ads7846_read_state(ts);    /*笔按下测量坐标值*/

        if (!ts->stopped)
            ads7846_report_state(ts);     /*报告事件*/

        wait_event_timeout(ts->wait, ts->stopped,msecs_to_jiffies(TS_POLL_PERIOD));
                                                                    /*等待一段时间*/
    }

    if (ts->pendown) {
        struct input_dev *input = ts->input;

        input_report_key(input, BTN_TOUCH, 0);     /*报告事件*/
        input_report_abs(input, ABS_PRESSURE, 0);
        input_sync(input);

        ts->pendown = false;
        dev_vdbg(&ts->spi->dev, "UP\n");
    }

    return IRQ_HANDLED;
}
```

驱动程序向控制器发送命令的操作保存在 ads7846 实例 xfer[18]和 msg[5]成员内，ads7846_read_state(ts)函数负责发送命令并读取测量数值。ads7846_report_state(ts)负责报告事件，函数定义如下：

```
static void ads7846_report_state(struct ads7846 *ts)
{
    ...
    input_report_abs(input, ABS_X, x);      /*报告 X 轴坐标*/
    input_report_abs(input, ABS_Y, y);      /*报告 Y 轴坐标*/
    input_report_abs(input, ABS_PRESSURE, ts->pressure_max - Rt);    /*报告压力值*/

    input_sync(input);
    ...
}
```

报告绝对值函数在前面已经介绍过了，最终的效果是将事件发送给事件处理器，由事件处理器传递给用户进程。

9.7 帧缓存设备

帧缓存设备是指 LCD 显示屏等显示设备，LCD 显示屏像素点的颜色值保存在内存中（帧缓存），由 LCD 控制器将数值写入到显示屏，对于用户进程来说只需要将显示内容写到缓存内存即可。

帧缓存设备通用代码位于/drivers/video/fbdev/core/目录下。本节先介绍帧缓存设备通用层代码，然后介绍龙芯 1B 开发板显示屏驱动程序的实现。

9.7.1 概述

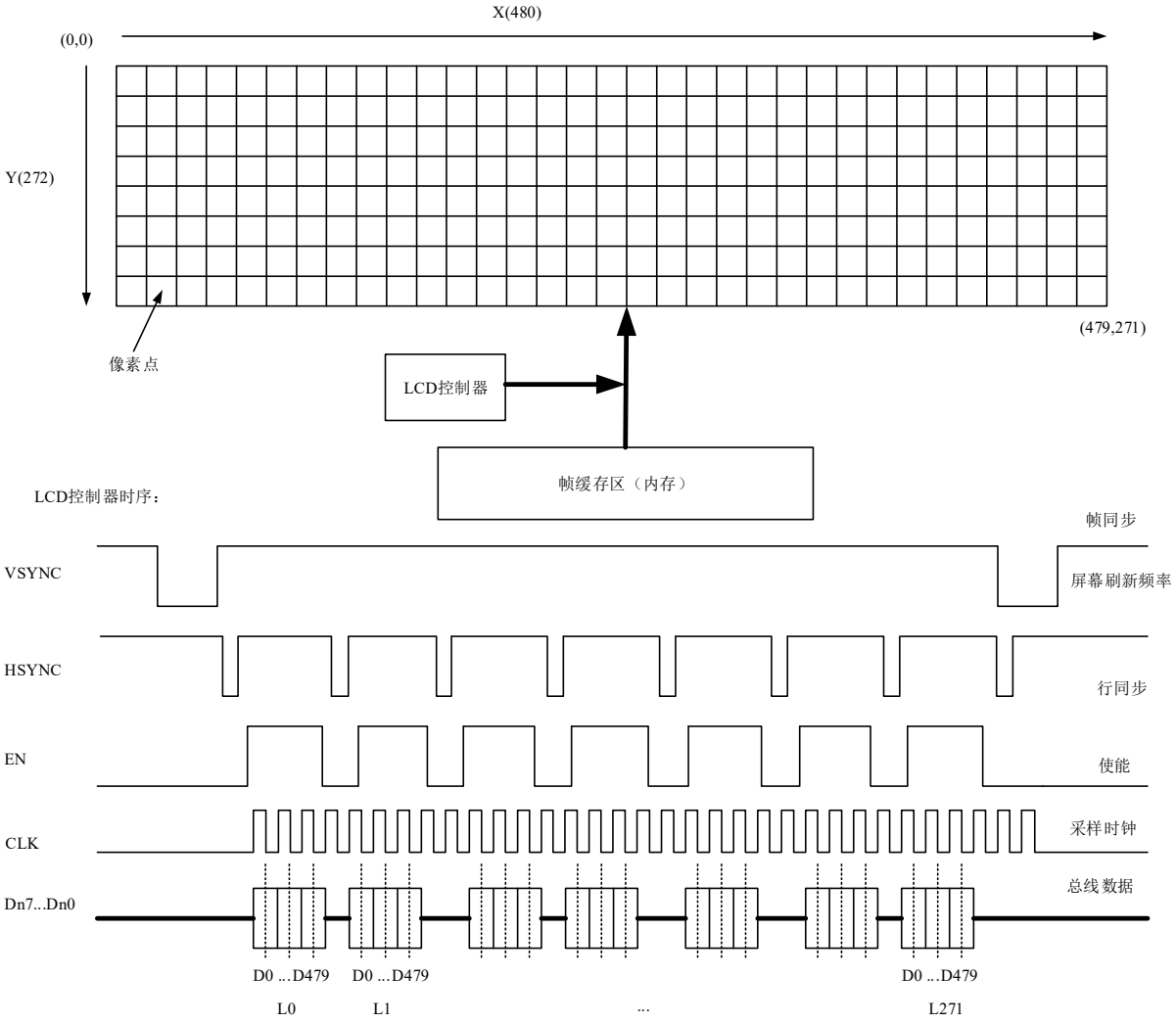
1 LCD 显示原理

LCD 显示屏由许多像素点组成，如 480×272 像素 TFT 显示屏表示长度方向上有 480 个像素点，宽度方向上有 272 个像素点。如果需要某一像素点显示则要对其写入颜色值，颜色值由 16/24bit（RGB）等二进制数表示。如需显示屏显示内容则需要从上至下，从左至右对每个像素点写入颜色值，写满一屏表示一帧。对用户进程而言相当于 LCD 显示屏具有一个内存缓存区，进程将需要显示的颜色值写入缓存区，LCD 控制器会自动将缓存数据写入 LCD 显示屏。

LCD 控制器在时钟、列同步、行同步信号的作用下，完成缓存区与 LCD 中像素点之间的数据传输，并保证将正确的数据写到正确的像素点上。

对于驱动程序来说只需为 LCD 控制器分配内存缓存区，根据硬件属性配置好 LCD 控制器，并激活控制器使其工作，控制器就会在时序控制下自动将缓存区数据传输到 LCD 屏，处理器无需干涉。对用户进程而言只需将要显示的内容写入缓存区即可，LCD 对其来说就是一块内存，LCD 控制器会自动完成数据同步。

LCD 控制器工作时序如下图所示（480×272）：



龙芯 1B 芯片 LCD 控制器输出信号主要有：

●VSYNC：帧同步信号，下降沿表示新的一帧开始。

●HSYNC：行同步信号，下降沿表示新的一行开始。

●EN：使能信号，高电平时允许 LCD 采样总线数据。

●CLK：数据采样时钟。

●R0...R7,G0...G7,B0...B7：数据总线，分别表示红、绿、蓝三基色的颜色值，像素点的颜色由三种颜色混合而成，对数据总线一次的采样表示一个像素点的颜色值。

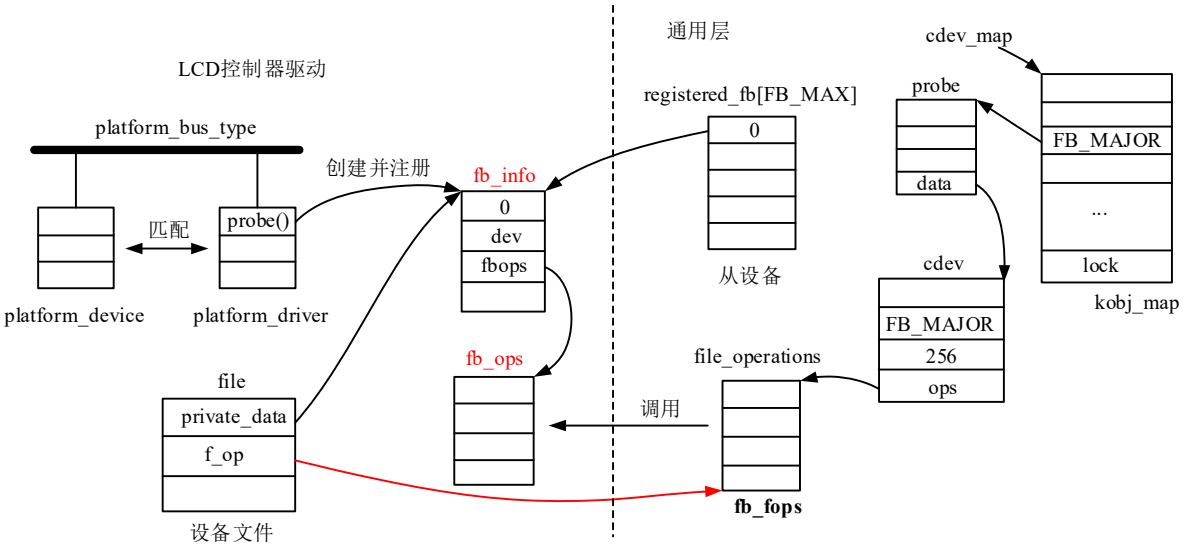
帧同步信号两次下降沿之间完成一帧数据的传递，也就是对整个屏幕打点一次，顺序是从上至下，从左至右依次进行（也可以有其它的顺序），在行同步信号两次下降沿之间完成一行像素的打点，使能信号用于允许 LCD 在采样时钟下降沿对数据总线进行采样。如屏幕像素为 480×272，则帧同步信号周期内最少有 272 个行同步周期，而行同步周期内最少需 480 个采样周期完成一行数据的采集。

2 驱动框架

帧缓存设备驱动框架如下图所示，通用层代码在内核初始化时（或加载模块时）为帧缓存设备注册了 cdev 实例，主设备号为 FB_MAJOR（29），定义了通用的文件操作结构 file_operations 实例 **fb_fops**。

具体的帧缓存设备由 **fb_info** 结构体表示，结构体中包含特定于设备控制器的操作结构 **fb_ops** 结构体成员。控制器驱动程序需实现 fb_ops 结构体实例，在 probe()函数中调用接口函数创建并注册 fb_info 结构体实例，定义的 fb_ops 结构体实例将赋予 fb_info 实例。

在驱动通用层由指针数组管理注册的 fb_info 实例，数组项数为 FB_MAX（32），数组项索引为从设备号。通用的帧缓存设备文件操作结构 file_operations 实例 fb_fops 中函数将调用从设备定义的 fb_ops 结构体实例中的函数完成操作。



3 初始化

内核在初始化阶段（或加载模块时）将调用 fbmem_init(void)函数初始化帧缓存设备驱动，初始化函数定义在/drivers/video/fbdev/core/fbmem.c 文件内：

```
static int __init    fbmem_init(void)
{
    proc_create("fb", 0, NULL, &fb_proc_fops);

    if (register_chrdev(FB_MAJOR,"fb",&fb_fops))      /*注册字符设备驱动，注册 cdev 实例*/
        printk("unable to get major %d for fb devs\n", FB_MAJOR);

    fb_class = class_create(THIS_MODULE, "graphics");   /*创建类*/
    ...
    return 0;
}

#ifdef MODULE
module_init(fbmem_init);
...
#else
subsys_initcall(fbmem_init);
#endif
```

初始化函数主要完成帧缓存设备字符设备驱动 cdev 实例创建和添加，以及设备类的创建。在注册字符设备驱动时，其文件操作结构 file_operations 实例为 **fb_fops**，其中定义了设备文件操作的接口函数。

9.7.2 数据结构

帧缓存设备驱动主要的数据结构有 fb_info 和 fb_ops，内核建立了指针数组 registered_fb[FB_MAX]，用于管理 fb_info 实例，实例在数组中的索引值表示从设备号，通用层提供了创建和注册 fb_info 实例的接口函数，fb_ops 实例需要具体设备控制器驱动程序定义。

1 fb_info

系统内每个帧缓存设备由 fb_info 结构体表示，结构体定义在/include/linux/fb.h 头文件内：

```
struct fb_info {
    atomic_t count;          /*引用计数*/
    int node;                /*节点号，从设备号*/
    int flags;               /*标记成员*/
    struct mutex lock;           /*互斥锁*/
    struct mutex mm_lock;        /* Lock for fb_mmap and smem_* fields */
    struct fb_var_screeninfo  var;    /*可变的屏幕参数*/
    struct fb_fix_screeninfo  fix;    /*固定的屏幕参数*/
    struct fb_monspecs  monspecs;  /*当前显示器规格*/
    struct work_struct  queue;    /* Framebuffer 事件列表*/
    struct fb_pixmap pixmap;     /*图像映射表*/
    struct fb_pixmap sprite;     /*光标映射表*/
    struct fb_cmap cmap;         /*当前 cmap */
    struct list_head  modelist;       /*模式列表*/
    struct fb_videomode *mode;    /*当前模式*/

#ifdef CONFIG_FB_BACKLIGHT
    struct backlight_device *bl_dev;  /*背光设备*/
    struct mutex bl_curve_mutex;
    u8 bl_curve[FB_BACKLIGHT_LEVELS];
#endif
#ifdef CONFIG_FB_DEFERRED_IO
    struct delayed_work deferred_work;
    struct fb_deferred_io *fbdefio;
#endif

    struct fb_ops *fbops;        /*帧缓存设备操作结构指针*/
    struct device *device;       /*父设备*/
    struct device *dev;          /*本设备*/
    int class_flag;              /*私有 sysfs 标志*/
#ifdef CONFIG_FB_TILEBLITTING
    struct fb_tile_ops *tileops;     /* Tile Blitting */
#endif
```

```
    char __iomem *screen_base;  /*缓存区起始地址*/
    unsigned long screen_size;      /* Amount of ioremapped VRAM or 0 */
    void *pseudo_palette;           /* Fake palette of 16 colors */

#define FBINFO_STATE_RUNNING    0
#define FBINFO_STATE_SUSPENDED  1
    u32 state;                  /*硬件状态，0 运行，1 暂停*/
    void *fbcon_par;            /* fbcon 私有数据，指向 fbcon_ops 实例*/

    void *par;              /*指向 fb_info 实例之后私有，在分配 fb_info 时分配*/
    struct apertures_struct {
        unsigned int count;
        struct aperture {
            resource_size_t base;
            resource_size_t size;
        } ranges[0];
    } *apertures;

    bool skip_vt_switch;        /* no VT switch on suspend/resume required */
};
```

fb_info 结构体中主要成员简介如下：

●**var**：fb_var_screeninfo 结构体成员，表示可变的屏幕参数，结构体定义在/include/uapi/linux/fb.h 头文件：

```
struct fb_var_screeninfo {
    __u32 xres;             /*可视像素分辨率*/
    __u32 yres;
    __u32 xres_virtual;     /*虚拟分辨率*/
    __u32 yres_virtual;
    __u32 xoffset;          /*可视分辨率相当于虚拟分辨率的偏移量*/
    __u32 yoffset;          /* resolution*/

    __u32 bits_per_pixel;       /**/
    __u32 grayscale;        /* 0 = color, 1 = grayscale,*/
                            /* >1 = FOURCC*/
    struct fb_bitfield red;         /* /include/uapi/linux/fb.h，bitfield in fb mem if true color, */
    struct fb_bitfield green;       /* else only length is significant */
    struct fb_bitfield blue;
    struct fb_bitfield transp;      /* transparency*/

    __u32 nonstd;           /* != 0 Non standard pixel format */
    __u32 activate;             /* see FB_ACTIVATE_**/
    __u32 height;           /*高度（mm） */
    __u32 width;            /*宽度（mm）*/
```

```
    __u32 accel_flags;      /* (OBSOLETE) see fb_info.flags */

    /* Timing: All values in pixclocks, except pixclock (of course) */
    __u32 pixclock;             /* pixel clock in ps (pico seconds) */
    __u32 left_margin;      /* time from sync to picture    */
    __u32 right_margin;         /* time from picture to sync    */
    __u32 upper_margin;         /* time from sync to picture    */
    __u32 lower_margin;
    __u32 hsync_len;        /* length of horizontal sync    */
    __u32 vsync_len;        /* length of vertical sync*/
    __u32 sync;             /* see FB_SYNC_*        */
    __u32 vmode;            /* see FB_VMODE_*           */
    __u32 rotate;           /* angle we rotate counter clockwise */
    __u32 colorspace;       /* colorspace for FOURCC-based modes */
    __u32 reserved[4];      /* Reserved for future compatibility */
};
```

●**fix**：fb_fix_screeninfo 结构体成员，表示固定的屏幕参数，定义在/include/uapi/linux/fb.h 头文件：

```
struct fb_fix_screeninfo {
    char id[16];                /*字符标识*/
    unsigned long smem_start;   /*帧缓存内存起始地址  */
    __u32 smem_len;             /*缓存区长度*/
    __u32 type;                 /*类型，形如  FB_TYPE_* */
    __u32 type_aux;             /* Interleave for interleaved Planes */
    __u32 visual;               /* see FB_VISUAL_*    */
    __u16 xpanstep;             /* zero if no hardware panning    */
    __u16 ypanstep;             /* zero if no hardware panning    */
    __u16 ywrapstep;        /* zero if no hardware ywrap       */
    __u32 line_length;      /* length of a line in bytes       */
    unsigned long mmio_start;   /*内存映射 I/O 起始地址  */
    __u32 mmio_len;             /*内存映射 I/O 长度*/
    __u32 accel;            /*指示具备的芯片/卡类型，FB_ACCEL_* */
    __u16 capabilities;     /* see FB_CAP_*   */
    __u16 reserved[2];      /* Reserved for future compatibility */
};
```

●**pixmap,sprite**：fb_pixmap 结构体成员，表示从通用框架至具体硬件的数据转换的数据结构，结构体定义在/include/linux/fb.h 头文件：

```
struct fb_pixmap {
    u8    *addr;        /*指向字节的指针*/
    u32 size;           /* size of buffer in bytes */
    u32 offset;         /* current offset to buffer*/
    u32 buf_align;      /* byte alignment of each bitmap   */
    u32 scan_align;         /* alignment per scanline*/
```

```
    u32 access_align;          /* alignment per read/write (bits)   */
    u32 flags;                   /* FB_PIXMAP_* */
    u32 blit_x;                  /* supported bit block dimensions (1-32)*/
    u32 blit_y;                  /* Format: blit_x = 1 << (width - 1)       */

    /*访问方式*/
    void (*writeio)(struct fb_info *info, void __iomem *dst, void *src, unsigned int size);
    void (*readio) (struct fb_info *info, void *dst, void __iomem *src, unsigned int size);
};
```

●**mode**：fb_videomode 结构体指针，结构体表示 LCD 显示器物理属性，定义在/include/linux/fb.h 头文件：

```
struct fb_videomode {
    const char *name;   /*名称*/
    u32 refresh;          /*刷新频率*/
    u32 xres;            /*X 轴分辨率*/
    u32 yres;             /*Y 轴分辨率*/
    u32 pixclock;
    u32 left_margin;
    u32 right_margin;
    u32 upper_margin;
    u32 lower_margin;
    u32 hsync_len;
    u32 vsync_len;
    u32 sync;
    u32 vmode;
    u32 flag;
};
```

驱动程序在/drivers/video/fbdev/core/modedb.c 文件内定义了标准显示屏对应的结构体实例数组。

●**modelist**：双链表成员用于管理 fb_modelist 结构体实例，结构体定义如下：

```
struct fb_modelist {
    struct list_head list;
    struct fb_videomode mode;
};
```

2 fb_ops

fb_ops 是一个非常重要的结构体，它是开发帧缓存设备驱动程序的核心结构，结构体定义如下：

```
struct fb_ops {
    struct module *owner;
    int (*fb_open)(struct fb_info *info, int user);          /*打开设备*/
    int (*fb_release)(struct fb_info *info, int user);       /*释放设备*/
    /*以下两个函数是为不支持 mmap 的设备提供的读写接口*/
    ssize_t (*fb_read)(struct fb_info *info, char __user *buf,size_t count, loff_t *ppos);    /*读缓存区*/
```

```
    ssize_t (*fb_write)(struct fb_info *info, const char __user *buf,size_t count, loff_t *ppos);
                                                                        /*写缓存区*/
    int (*fb_check_var)(struct fb_var_screeninfo *var, struct fb_info *info);   /*参数检查*/
    int (*fb_set_par)(struct fb_info *info);       /*设置模式*/
    int (*fb_setcolreg)(unsigned regno, unsigned red, unsigned green,unsigned blue,   \
                                                    unsigned transp, struct fb_info *info);

    int (*fb_setcmap)(struct fb_cmap *cmap, struct fb_info *info);
    int (*fb_blank)(int blank, struct fb_info *info);          /*清屏*/

    int (*fb_pan_display)(struct fb_var_screeninfo *var, struct fb_info *info);

    void (*fb_fillrect) (struct fb_info *info, const struct fb_fillrect *rect);       /*画矩形*/
    void (*fb_copyarea) (struct fb_info *info, const struct fb_copyarea *region);       /*复制数据*/
    void (*fb_imageblit) (struct fb_info *info, const struct fb_image *image);          /*显示图像*/

    int (*fb_cursor) (struct fb_info *info, struct fb_cursor *cursor);       /*显示光标*/
    void (*fb_rotate)(struct fb_info *info, int angle);       /*旋转图像*/
    int (*fb_sync)(struct fb_info *info);
    int (*fb_ioctl)(struct fb_info *info, unsigned int cmd,unsigned long arg);     /*设备控制*/

    int (*fb_compat_ioctl)(struct fb_info *info, unsigned cmd,unsigned long arg);

    int (*fb_mmap)(struct fb_info *info, struct vm_area_struct *vma);     /*特殊映射函数*/

    void (*fb_get_caps)(struct fb_info *info, struct fb_blit_caps *caps,struct fb_var_screeninfo *var);
    void (*fb_destroy)(struct fb_info *info);

    int (*fb_debug_enter)(struct fb_info *info);
    int (*fb_debug_leave)(struct fb_info *info);
};
```

具体设备驱动程序需要实现 **fb_ops** 实例并赋予设备对应的 fb_info 实例。fb_info 实例的创建和注册见下一小节。

9.7.3 分配/注册设备

帧缓存设备控制器驱动的 probe()函数需要创建并设置 fb_info 实例，并最后向帧缓存设备驱动通用层注册实例。

1 分配 fb_info

帧缓存设备驱动通用层提供了创建 fb_info 结构体实例的函数（/drivers/video/fbdev/core/fbsysfs.c）：

```
struct fb_info *framebuffer_alloc(size_t size, struct device *dev)
/*size：驱动私有数据大小，dev：指向父设备 device 实例*/
```

```
{
  #define BYTES_PER_LONG    (BITS_PER_LONG/8)
  #define PADDING    (BYTES_PER_LONG - (sizeof(struct fb_info) % BYTES_PER_LONG))
     int fb_info_size = sizeof(struct fb_info);
     struct fb_info *info;
     char *p;

     if (size)
          fb_info_size += PADDING;    /*字对齐*/

     p = kzalloc(fb_info_size + size, GFP_KERNEL);    /*创建结构体实例，加上 size 大小*/
     if (!p)
          return NULL;

     info = (struct fb_info *) p;    /*指向分配空间起始地址*/

     if (size)
          info->par = p + fb_info_size;      /*指同 fb_info 实例之后的内存地址*/

     info->device = dev;    /*父设备*/

  #ifdef CONFIG_FB_BACKLIGHT
      mutex_init(&info->bl_curve_mutex);
  #endif

      return info;       /*返回 fb_info 实例指针*/
  #undef PADDING
  #undef BYTES_PER_LONG
}
```

2 注册 fb_info

创建 fb_info 实例后，控制器驱动还需要设置 fb_info 实例，例如对其 fbops 指针成员赋值（指向 fb_ops 结构体），最后向内核注册实例。

内核在/drivers/video/fbdev/core/fbmem.c 文件内定义了指针数组用于管理注册的 fb_info 实例：

```
struct fb_info *registered_fb[FB_MAX] __read_mostly;     /*指针数组，FB_MAX=32*/
int num_registered_fb    __read_mostly;     /*当前注册的帧缓存设备数量*/
```

注册 fb_info 实例函数定义如下（/drivers/video/fbdev/core/fbmem.c）：

```
int register_framebuffer(struct fb_info *fb_info)
{
     int ret;

     mutex_lock(&registration_lock);
```

```
    ret = do_register_framebuffer(fb_info);     /*/drivers/video/fbdev/core/fbmem.c*/
    mutex_unlock(&registration_lock);

    return ret;
}
```

注册函数内调用 do_register_framebuffer(fb_info)函数完成注册工作，代码如下：

```
static int do_register_framebuffer(struct fb_info *fb_info)
{
    int i, ret;
    struct fb_event event;
    struct fb_videomode mode;     /*定义实例，局部变量*/

    if (fb_check_foreignness(fb_info))
        return -ENOSYS;

    ret = do_remove_conflicting_framebuffers(fb_info->apertures,fb_info->fix.id,     \
                                             fb_is_primary_device(fb_info));
    ...
    if (num_registered_fb == FB_MAX)
        return -ENXIO;

    num_registered_fb++;
    for (i = 0 ; i < FB_MAX; i++)     /*查找为 NULL 的数组项*/
        if (!registered_fb[i])
            break;

    fb_info->node = i;            /*从设备号*/
    atomic_set(&fb_info->count, 1);
    mutex_init(&fb_info->lock);
    mutex_init(&fb_info->mm_lock);

    fb_info->dev = device_create(fb_class, fb_info->device,MKDEV(FB_MAJOR, i), NULL, "fb%d", i);
                                        /*创建表示设备本身的 device 实例，并添加*/
    if (IS_ERR(fb_info->dev)) {
        ...
    } else
        fb_init_device(fb_info);    /*为 device 实例创建属性文件等，/drivers/video/fbdev/core/fbsysfs.c*/

    if (fb_info->pixmap.addr == NULL) {     /*没有分配缓存区，则分配*/
        fb_info->pixmap.addr = kmalloc(FBPIXMAPSIZE, GFP_KERNEL);     /*1024*8*/
        if (fb_info->pixmap.addr) {
            fb_info->pixmap.size = FBPIXMAPSIZE;    /*8KB*/
```

```
                fb_info->pixmap.buf_align = 1;
                fb_info->pixmap.scan_align = 1;
                fb_info->pixmap.access_align = 32;
                fb_info->pixmap.flags = FB_PIXMAP_DEFAULT;
        }
    }
    fb_info->pixmap.offset = 0;

    if (!fb_info->pixmap.blit_x)
        fb_info->pixmap.blit_x = ~(u32)0;

    if (!fb_info->pixmap.blit_y)
        fb_info->pixmap.blit_y = ~(u32)0;

    if (!fb_info->modelist.prev || !fb_info->modelist.next)
        INIT_LIST_HEAD(&fb_info->modelist);

    if (fb_info->skip_vt_switch)
        pm_vt_switch_required(fb_info->dev, false);
    else
        pm_vt_switch_required(fb_info->dev, true);

    fb_var_to_videomode(&mode, &fb_info->var);
        /*fb_var_screeninfo 结构体信息赋予 fb_videomode 结构体，/drivers/video/fbdev/core/modedb.c*/
    fb_add_videomode(&mode, &fb_info->modelist);
       /*创建 fb_modelist 实例，添加到 fb_info->modelist 链表，/drivers/video/fbdev/core/modedb.c*/

    registered_fb[i] = fb_info;        /*关联指针数组项*/

    event.info = fb_info;
    console_lock();
    ...

    fb_notifier_call_chain(FB_EVENT_FB_REGISTERED, &event);
    unlock_fb_info(fb_info);
    console_unlock();
    return 0;
}
```

注册函数首先搜索 registered_fb[]指针数组，找到第一个空闲的数组项，数组项索引值作为从设备号，然后为设备创建添加 device 实例并创建属性文件，为帧缓存区分配内存空间，最后根据 fb_var_screeninfo 成员信息创建 fb_modelist 实例，添加到 fb_info->modelist 链表。

9.7.4 设备文件操作

在前面介绍的帧缓存设备初始化时会注册字符设备 cdev 实例，cdev 实例赋予的文件操作结构实例为 fb_fops，这是所有帧缓存设备文件统一的操作接口，实例定义如下（/drivers/video/fbdev/core/fbmem.c）：

```
static const struct file_operations fb_fops = {
    .owner =  THIS_MODULE,
    .read =   fb_read,        /*读缓存区*/
    .write =  fb_write,       /*写缓存区*/
    .unlocked_ioctl = fb_ioctl,      /*设备控制*/
#ifdef CONFIG_COMPAT
    .compat_ioctl = fb_compat_ioctl,
#endif
    .mmap =       fb_mmap,   /*映射操作，将缓存区映射到用户空间*/
    .open =       fb_open,       /*打开设备*/
    .release = fb_release,
#ifdef HAVE_ARCH_FB_UNMAPPED_AREA
    .get_unmapped_area = get_fb_unmapped_area,
#endif
#ifdef CONFIG_FB_DEFERRED_IO
    .fsync =  fb_deferred_io_fsync,
#endif
    .llseek =  default_llseek,
};
```

下面简要介绍 fb_fops 实例中的几个函数。

1 打开设备

打开设备函数 fb_open()定义如下：

```
static int fb_open(struct inode *inode, struct file *file)
__acquires(&info->lock)
__releases(&info->lock)
{
    int fbidx = iminor(inode);
    struct fb_info *info;
    int res = 0;

    info = get_fb_info(fbidx);     /*从 registered_fb[]指针数组获取 fb_info 实例*/
    ...       /*错误处理*/

    mutex_lock(&info->lock);
    if (!try_module_get(info->fbops->owner)) {
        ...
    }
    file->private_data = info;       /*file 私有数据结构指针指向 fb_info 实例*/
```

```
    if (info->fbops->fb_open) {        /*调用 fb_ops 结构体 open()函数*/
        res = info->fbops->fb_open(info,1);
        if (res)
            module_put(info->fbops->owner);
    }
#ifdef CONFIG_FB_DEFERRED_IO
    if (info->fbdefio)
        fb_deferred_io_open(info, inode, file);
#endif
out:
    mutex_unlock(&info->lock);
    if (res)
        put_fb_info(info);
    return res;
}
```

设备文件打开操作主要是由从设备号查找 registered_fb[]指针数组获取 fb_info 实例，并调用对应 fb_ops 实例中定义的 fb_open()函数，设备文件 file->private_data 成员指向 fb_info 实例。

2 读写操作

帧缓存设备文件读写函数直接就是对帧缓存内存区的读写。读操作函数从帧缓存区中读取数据（截屏），函数定义如下：

```
static ssize_t fb_read(struct file *file, char __user *buf, size_t count, loff_t *ppos)
{
    unsigned long p = *ppos;   /*当前文件位置*/
    struct fb_info *info = file_fb_info(file);   /*打开设备时赋值 file->private_data = info*/
    u8 *buffer, *dst;
    u8 __iomem *src;
    int c, cnt = 0, err = 0;
    unsigned long total_size;

    if (!info || ! info->screen_base)
        return -ENODEV;

    if (info->state != FBINFO_STATE_RUNNING)
        return -EPERM;

    if (info->fbops->fb_read)      /*若未定义特定读操作函数，则采用通用操作*/
        return info->fbops->fb_read(info, buf, count, ppos);   /*调用特定设备读操作函数*/

    /*以下为通用读操作内容，从帧缓存区读数据*/
    total_size = info->screen_size;

    if (total_size == 0)
```

```
        total_size = info->fix.smem_len;     /*帧缓存区大小*/

    if (p >= total_size)
        return 0;

    if (count >= total_size)
        count = total_size;

    if (count + p > total_size)
        count = total_size - p;

    buffer = kmalloc((count > PAGE_SIZE) ? PAGE_SIZE : count,GFP_KERNEL);
                                                /*内核空间分配内存，用于暂存数据*/
    if (!buffer)
        return -ENOMEM;
    src = (u8 __iomem *) (info->screen_base + p);     /*缓存区起始地址加偏移量*/

    if (info->fbops->fb_sync)
        info->fbops->fb_sync(info);

    while (count) {
        c    = (count > PAGE_SIZE) ? PAGE_SIZE : count;
        dst = buffer;
        fb_memcpy_fromfb(dst, src, c);     /*从帧缓存区复制数据到暂存区*/
        dst += c;
        src += c;

        if (copy_to_user(buf, buffer, c)) {     /*数据从暂存区复制到用户空间*/
            err = -EFAULT;
            break;
        }
        *ppos += c;
        buf += c;
        cnt += c;
        count -= c;
    }

    kfree(buffer);
    return (err) ? err : cnt;
}
```

帧缓存设备文件写操作函数定义如下，与读操作类似：

```
static ssize_t fb_write(struct file *file, const char __user *buf, size_t count, loff_t *ppos)
```

```
{
    unsigned long p = *ppos;
    struct fb_info *info = file_fb_info(file);      /*获取 fb_info 实例*/
    u8 *buffer, *src;
    u8 __iomem *dst;
    int c, cnt = 0, err = 0;
    unsigned long total_size;

    if (!info || !info->screen_base)
        return -ENODEV;

    if (info->state != FBINFO_STATE_RUNNING)
        return -EPERM;

    if (info->fbops->fb_write)          /*如果定义了特定的写操作则调用*/
        return info->fbops->fb_write(info, buf, count, ppos);

    total_size = info->screen_size;

    if (total_size == 0)
        total_size = info->fix.smem_len;

    if (p > total_size)
        return -EFBIG;

    if (count > total_size) {
        err = -EFBIG;
        count = total_size;
    }

    if (count + p > total_size) {
        if (!err)
            err = -ENOSPC;

        count = total_size - p;
    }

    buffer = kmalloc((count > PAGE_SIZE) ? PAGE_SIZE : count,GFP_KERNEL); /*分配暂存区*/
    if (!buffer)
        return -ENOMEM;

    dst = (u8 __iomem *) (info->screen_base + p);
```

```
    if (info->fbops->fb_sync)
        info->fbops->fb_sync(info);

    while (count) {
        c = (count > PAGE_SIZE) ? PAGE_SIZE : count;
        src = buffer;

        if (copy_from_user(src, buf, c)) {     /*用户空间数据复制到暂存区*/
            err = -EFAULT;
            break;
        }

        fb_memcpy_tofb(dst, src, c);          /*暂存区复制到帧缓存区*/
        dst += c;
        src += c;
        *ppos += c;
        buf += c;
        cnt += c;
        count -= c;
    }
    kfree(buffer);
    return (cnt) ? cnt : err;
}
```

3 设备映射

帧缓存设备还有一个比较重要的操作就是映射操作，可以将缓存区映射到用户进程空间，映射完后用户进程可以将对帧缓存区的操作直接转换成对自身地址空间内存的操作。映射函数定义如下：

```
static int fb_mmap(struct file *file, struct vm_area_struct * vma)
{
    struct fb_info *info = file_fb_info(file);
    struct fb_ops *fb;
    unsigned long mmio_pgoff;
    unsigned long start;
    u32 len;

    if (!info)
        return -ENODEV;
    fb = info->fbops;
    if (!fb)
        return -ENODEV;
    mutex_lock(&info->mm_lock);
    if (fb->fb_mmap) {     /*如果定义了 fb->fb_mmap()函数，没有定义则采用通用映射操作*/
        int res;
```

```
            res = fb->fb_mmap(info, vma);
            mutex_unlock(&info->mm_lock);
            return res;
        }
        /*通用映射操作*/
        start = info->fix.smem_start;
        len = info->fix.smem_len;
        mmio_pgoff = PAGE_ALIGN((start & ~PAGE_MASK) + len) >> PAGE_SHIFT;
        if (vma->vm_pgoff >= mmio_pgoff) {
            if (info->var.accel_flags) {
                mutex_unlock(&info->mm_lock);
                return -EINVAL;
            }

            vma->vm_pgoff -= mmio_pgoff;
            start = info->fix.mmio_start;
            len = info->fix.mmio_len;
        }
        mutex_unlock(&info->mm_lock);

        vma->vm_page_prot = vm_get_page_prot(vma->vm_flags);
        fb_pgprotect(file, vma, start);

        return vm_iomap_memory(vma, start, len);
    }
```

4 设备控制

设备控制函数 fb_ioctl()用于向设备发送命令，实现对设备的控制，函数定义如下：

```
static long fb_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
    struct fb_info *info = file_fb_info(file);

    if (!info)
        return -ENODEV;
    return do_fb_ioctl(info, cmd, arg);
}
```

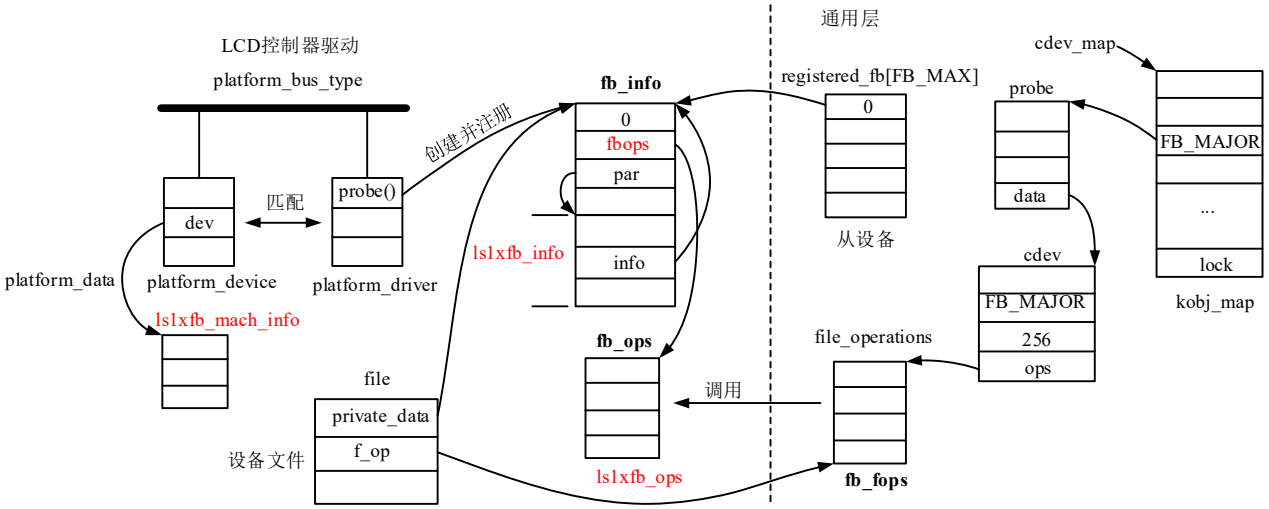
do_fb_ioctl()函数是一个命令分配器，根据不同的命令调用不同的函数实现，源代码请读者自行阅读。命令参数定义在/include/uapi/linux/fb.h 头文件内，由于命令的发送属用户空间编程的内容。本书只讲解内核的实现机制，具体命令定义及响应请读者自行阅读源代码。

9.7.5 驱动示例

龙芯 1x 处理器内置了 LCD 控制器，通过对寄存器的配置实现对 LCD 控制器的控制。LCD 控制器视

为设备挂接在 platform 总线，在板级代码中需定义并注册表示 LCD 控制器的 platform_device 实例。驱动程序中定义并注册了 platform_driver 实例，在其 probe()函数中将创建并注册帧缓存设备 fb_info 结构体实例。

龙芯 1x 帧缓存设备驱动框架如下图所示：



ls1xfb_mach_info 结构体用于向驱动传递 LCD 控制器信息，ls1xfb_info 结构体是驱动中 fb_info 实例的私有数据。驱动程序中定义了 fb_ops 结构体实例 **ls1xfb_ops**。

1 控制器信息

驱动程序定义了 ls1xfb_mach_info 结构体用于表示 LCD 控制器的板级信息，结构体定义如下：

```
struct ls1xfb_mach_info {                    /*/include/video/ls1xfb.h*/
    char   id[16];              /*fb_videomode 实例名称*/
    int         num_modes;
    struct fb_videomode *modes;
    unsigned pix_fmt;           /*数据格式*/
    unsigned invert_pixclock:1;
    unsigned invert_pixde:1;
    unsigned de_mode:1;
    unsigned enable_lcd:1;
};
```

在板级相关文件内需定义 ls1xfb_mach_info 实例以及表示控制器设备的 platform_device：

```
static struct resource ls1x_fb0_resource[] = {          /*资源*/
    [0] = {
        .start = LS1X_DC0_BASE,         /*控制寄存器基址*/
        .end   = LS1X_DC0_BASE + 0x0010 - 1,   /* 1M? */
        .flags = IORESOURCE_MEM,
    },
};

struct ls1xfb_mach_info ls1x_lcd0_info = {          /*ls1xfb_mach_info 实例*/
    .id             = "Graphic lcd",
```

```
    .modes              = video_modes,
    .num_modes          = ARRAY_SIZE(video_modes),
    .pix_fmt        = PIX_FMT_RGB565,
    .de_mode        = 0, /* 注意：lcd 是否使用 DE 模式 */
        /* 根据 lcd 屏修改 invert_pixclock 和 invert_pixde 参数(0 或 1)，部分 lcd 可能显示不正常 */
    .invert_pixclock    = 0,
    .invert_pixde   = 0,
};
```

LCD 控制器 platform_device 实例定义如下：

```
struct platform_device ls1x_fb0_device = {
    .name               = "ls1x-fb",        /*名称，用于匹配驱动*/
    .id                 = 0,
    .num_resources      = ARRAY_SIZE(ls1x_fb0_resource),
    .resource       = ls1x_fb0_resource,            /*资源*/
    .dev            = {
        .platform_data = &ls1x_lcd0_info,           /*控制器信息*/
    }
};
```

在板级定义的初始化函数中将向内核注册 **ls1x_fb0_device** 实例。

2 LCD 控制器驱动

龙芯 1x 帧缓存设备驱动中 fb_info 私有数据由 ls1xfb_info 结构体表示，定义如下：

```
struct ls1xfb_info {
    struct device   *dev;
    struct clk      *clk;
    struct fb_info      *info;      /*指向 fb_info 实例*/
    struct ls1xfb_i2c_chan    chan;
    unsigned char *edid;
    void __iomem        *reg_base;
    dma_addr_t          fb_start_dma;
    u32             pseudo_palette[16];
    int             pix_fmt;
    unsigned        de_mode:1;
};
```

驱动程序中实现的 fb_ops 结构体实例如下：

```
static struct fb_ops ls1xfb_ops = {
    .owner          = THIS_MODULE,
    .fb_check_var = ls1xfb_check_var,
    .fb_set_par     = ls1xfb_set_par,
    .fb_setcolreg   = ls1xfb_setcolreg,
//  .fb_blank = ls1xfb_blank,
```

```
    .fb_pan_display     = ls1xfb_pan_display,
    .fb_fillrect        = cfb_fillrect,
    .fb_copyarea        = cfb_copyarea,
    .fb_imageblit       = cfb_imageblit,
//  .fb_ioctl           = ls1xfb_ioctl,
                /* 可用于 LCD 控制器的 Switch Panel 位，实现显示单元 0 和 1 的相互复制 */
};
```

ls1xfb_ops 实例中各函数请读者自行参考源代码。

LCD 控制器驱动 platform_driver 实例定义如下：

```
static struct platform_driver ls1xfb_driver = {
    .driver         = {
        .name       = "ls1x-fb",        /*匹配设备*/
        .owner      = THIS_MODULE,
    },
    .probe      = ls1xfb_probe,         /*探测函数*/
    .remove     = __devexit_p(ls1xfb_remove),
    .suspend    = ls1xfb_suspend,
    .resume     = ls1xfb_resume,
};
```

在初始化函数（模块加载函数）中将注册 ls1xfb_driver 实例，如下所示：

```
static int __init ls1xfb_init(void)
{
    ...
    return platform_driver_register(&ls1xfb_driver);        /*注册 ls1xfb_driver 实例*/
}
module_init(ls1xfb_init);
```

ls1xfb_driver 实例中探测函数定义简列如下：

```
static int __devinit ls1xfb_probe(struct platform_device *pdev)
{
    struct ls1xfb_mach_info *mi;
    struct fb_info *info = 0;
    struct ls1xfb_info *fbi = 0;
    struct resource *res;
    struct clk *clk;
    int ret;

    mi = pdev->dev.platform_data;       /*ls1xfb_mach_info 结构指针*/
    ...
```

```
    ...
    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);     /*获取控制寄存器资源*/
    ...
    info = framebuffer_alloc(sizeof(struct ls1xfb_info), &pdev->dev);
                                                    /*分配 fb_info 实例，后接 ls1xfb_info 实例*/
    ...
    /*初始化实例*/
    fbi = info->par;
    fbi->info = info;
    fbi->clk = clk;
    fbi->dev = info->dev = &pdev->dev;
    fbi->de_mode = mi->de_mode;
    ...
    info->fbops = &ls1xfb_ops;          /*赋值 fb_ops 实例指针*/
    info->pseudo_palette = fbi->pseudo_palette;
    fbi->reg_base = ioremap_nocache(res->start, resource_size(res));     /*控制寄存器基址*/
    ...
    /*分配缓存空间*/
    if (unlikely(vga_mode)) {
        info->fix.smem_len = PAGE_ALIGN(1920 * 1080 * 4); /*分配足够的显存，用于切换分辨率*/
    } else {
        info->fix.smem_len = PAGE_ALIGN(default_xres * default_yres * 4);
    }

    info->screen_base = dma_alloc_coherent(fbi->dev, info->fix.smem_len,&fbi->fb_start_dma, \
                                                                        GFP_KERNEL);
    ...
    info->fix.smem_start = (unsigned long)fbi->fb_start_dma;
    set_graphics_start(info, 0, 0);
    /*设置显示模式*/
    ...
    /* init video mode data.*/
    ls1xfb_init_mode(info, mi);
    ret = ls1xfb_check_var(&info->var, info);
    ...
    /*初始化 LCD 控制器*/
    ...
    ret = register_framebuffer(info);     /*注册 fb_info 实例*/
    ...
    platform_set_drvdata(pdev, fbi);
    return 0;
    ...
}
```

由于作者水平有限，以上只是简单介绍了帧缓存设备的驱动框架，还有很多细节并没有提及，有兴趣的读者可自行研究。

9.8 终端设备

早期计算机比较昂贵，不能每个人都拥有计算机。用户通过终端连接计算机主机，终端只具有显示和输入功能，没有计算功能。主机通过连接多个终端来实现多用户多任务。

历史上，用户接入一个 UNIX 系统都是利用终端通过串行线（RS-232 连接）连接到主机。终端由阴极射线管（CRT）组成，能够显示字符。甚至更早的时期，终端有时候还是硬拷贝电传设备（Teletype）。我们常用到的 tty 是 Teletype 的缩写，也作为终端设备的缩写。

如今传统型的终端已经不常见了，因为用户都可以独占地拥有计算机了，不需要通过传统终端接入主机。现在常用的用户接口是图形界面中的窗口，视为终端模拟器，称之为虚拟终端设备，系统内可以同时开启多个虚拟终端。

总之，终端是用来实现人机交互的设备，进程通过终端设备向用户输入和输出信息。在 Linux 系统中设备文件/dev/ttySn（n 为编号）表示传统的串口终端，/dev/ttyn（n 为编号）表示虚拟终端。

本节主要介绍终端设备驱动的实现框架，以及串口终端设备驱动程序的实现，虚拟终端设备驱动到下一节再介绍。终端设备驱动代码位于/drivers/tty/目录下。

9.8.1 概述

终端设备与用户进程（或内核）之间通过终端协议传输数据，主要包含特殊字符的含义等，例如移动光标到一行的开头用什么字符表示等。

早期的终端设备没有统一的标准（什么动作用什么字符定义），但最终 Digitals 的 VT-100 成了事实上的标准，也成了 ANSI 标准。

根据终端设备与主机的连接方式，Linux 中终端设备主要包含串口终端设备、虚拟终端设备和伪终端设备等。终端设备的主设备号为 TTY_MAJOR（4），虚拟终端设备从设备号为 0~63，串口终端从设备号为 64~255。

（1）虚拟终端设备（**/dev/ttyn**）

虚拟终端设备主要是本机的键盘和显示器，主机通过终端协议访问虚拟终端设备。虚拟终端设备可理解成系统中一个窗口。由于它不是传统上的终端设备但是又使用了终端的传输协议，因此称为虚拟终端设备。虚拟终端设备主设备号为 TTY_MAJOR（4），从设备号为 1~63，设备文件名为 tty1~tty63，tty0 表示当前虚拟终端。

（2）串口终端设备（**/dev/ttySn**）

串口终端设备是使用 UART 串行端口与主机相连的终端设备，其主设备号为 TTY_MAJOR（4），从设备号为 64~255，设备文件名为 ttyS0~ttyS191。

（3）伪终端设备

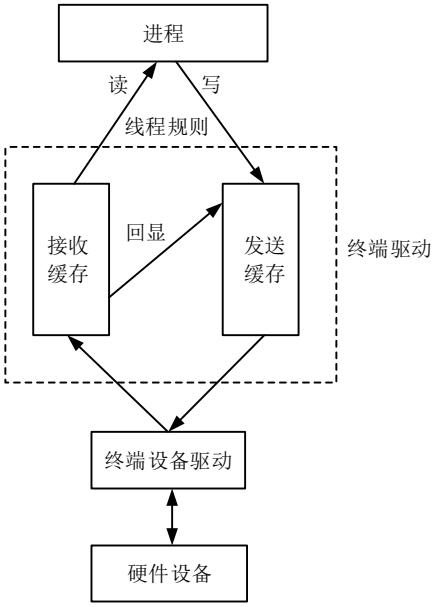
伪终端设备是一种虚拟设备（由软件模拟），它提供一个 IPC 通道，用作进程间通信的逻辑设备，伪终端设备总是成对出现，本节暂不介绍伪终端设备。

1 驱动框架

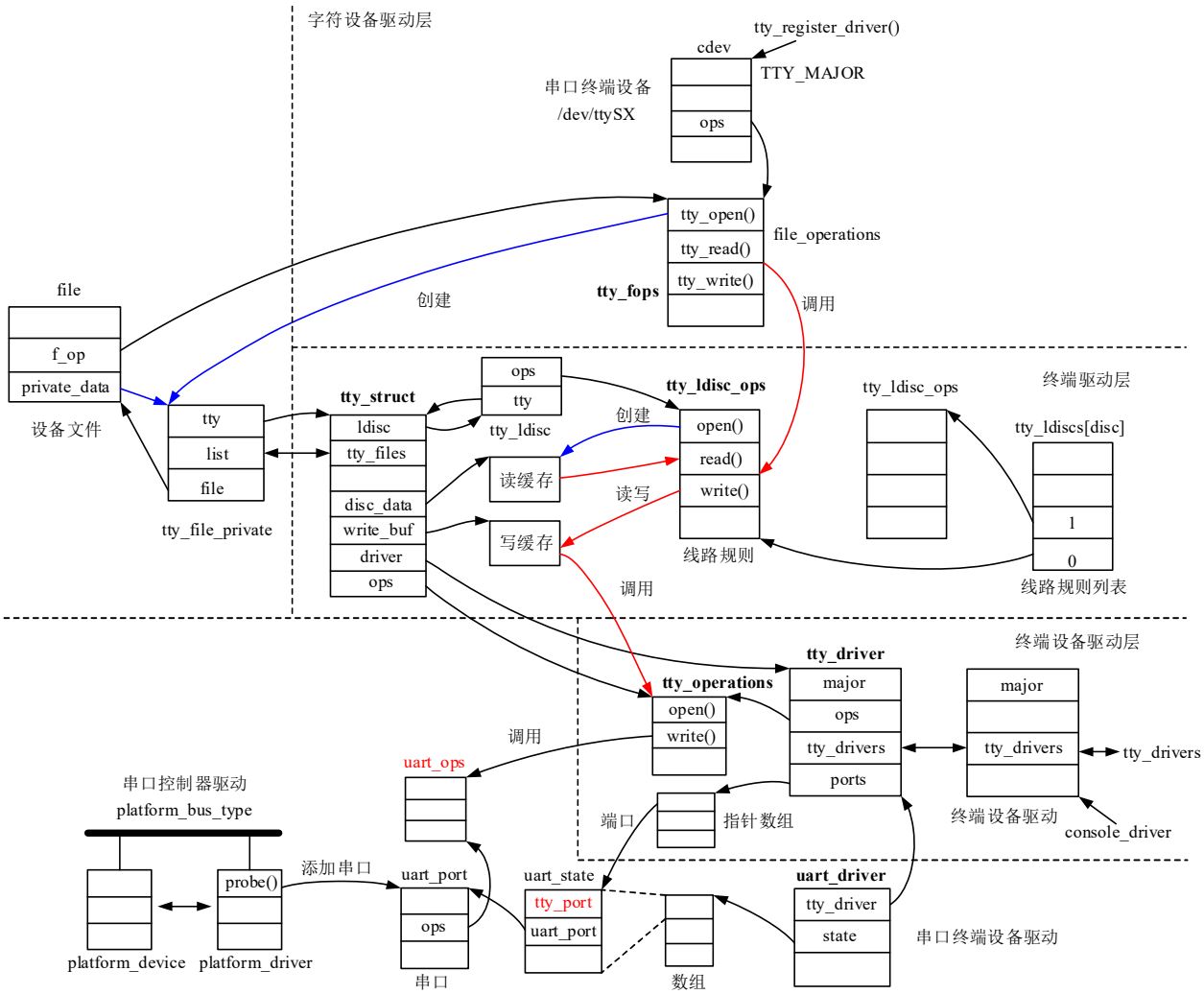
终端可以理解成一种数据传输协议，协议中规定了控制终端设备行为的字符等，例如换行操作用什么字符表示。按照终端传输协议进行数据传输的设备称之为终端设备。

终端驱动主要实现数据的缓存，以及与用户数据之间的交互，如下图所示。终端驱动调用终端设备驱动中的操作函数将缓存中数据写入硬件设备或从设备接收数据写入缓存。

终端设备驱动实现底层的数据传输和控制。



终端驱动、终端设备驱动框架如下图所示：



终端设备驱动由 tty_driver 结构体表示，一个驱动适用于一类终端设备，如串口设备终端驱动共用一个 tty_driver 实例。tty_driver 结构体中 ports 成员指向一个指针数组，数组项指向端口 tty_port 结构体。tty_port

表示终端一个通道，或者说从设备。tty_driver 关联的 tty_operations 结构体用于实现底层硬件的数据传输和控制。

tty_driver 和 tty_operations 实例需要具体终端设备驱动定义，并注册。例如，串口终端驱动中通过注册 uart_driver 实例来注册 tty_driver 实例。串口终端驱动中 uart_port 结构体表示串口，与终端驱动中端口 tty_port 结构体一一对应。tty_operations 实例调用串口操作结构中的函数实现底层硬件操作。

在具体终端设备驱动的初始化函数（模块加载函数）中需要注册 tty_driver 实例，在具体终端设备控制器驱动的 probe()函数中需要注册端口 tty_port 实例（激活通道）。

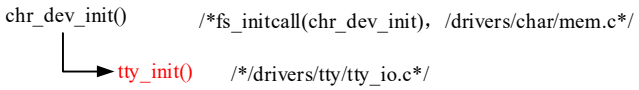
在注册 tty_driver 或 tty_port 实例时，将为端口（从设备）创建和添加字符设备驱动 cdev 实例以及对应的 device 实例。打开终端设备文件时，将为终端设备创建 tty_struct 实例，并将其关联到默认线路规则和终端设备驱动 tty_driver 实例（由设备号查找 tty_driver 实例）。

tty_struct 结构体中主要包含读写数据的缓存区，线路规则用于实现进程与 tty_struct 数据缓存区之间的数据传输。写数据时，线路规则先将用户数据写入 tty_struct 写数据缓存区，然后调用 tty_operations 实例中的 write()函数写入硬件设备。读数据时，线路规则从 tty_struct 读数据缓存区获取数据返回给用户进程，若读缓存区没有数据（或不够）可能睡眠等待。

终端设备在收到数据时（中断处理函数中），将数据缓存到端口缓存区，然后由线路规则将数据填充到 tty_struct 读数据缓存区，并唤醒读数据睡眠等待进程。

2 初始化

TTY 驱动初始化函数 **tty_init()**主要完成 TTY 设备字符设备驱动 cdev 实例的注册，函数调用关系如下图所示：



tty_init()函数定义在/drivers/tty/tty_io.c 文件内，代码如下：

```
int __init tty_init(void)
{
    cdev_init(&tty_cdev, &tty_fops);     /*初始化/dev/tty 设备文件对应的 cdev 实例（固定设备号）*/
    if (cdev_add(&tty_cdev, MKDEV(TTYAUX_MAJOR, 0), 1) ||        /*添加 cdev，1 个设备*/
       register_chrdev_region(MKDEV(TTYAUX_MAJOR, 0), 1, "/dev/tty") < 0)
            panic("Couldn't register /dev/tty driver\n");

    device_create(tty_class, NULL, MKDEV(TTYAUX_MAJOR, 0), NULL, "tty");
                                        /*创建添加 device 实例，创建/dev/tty 设备文件*/

    /*初始化并添加/dev/console（当前系统控制台）设备文件对应的 cdev 实例*/
    cdev_init(&console_cdev, &console_fops);
    if (cdev_add(&console_cdev, MKDEV(TTYAUX_MAJOR, 1), 1) ||
        register_chrdev_region(MKDEV(TTYAUX_MAJOR, 1), 1, "/dev/console") < 0)     /*申请设备号*/
         panic("Couldn't register /dev/console driver\n");

    consdev = device_create_with_groups(tty_class, NULL,
                        MKDEV(TTYAUX_MAJOR, 1), NULL,cons_dev_groups, "console");
                                            /*创建设备文件/dev/console*/
```

```
    if (IS_ERR(consdev))
        consdev = NULL;

#ifdef CONFIG_VT     /*如果支持虚拟终端*/
    vty_init(&console_fops);     /*虚拟终端初始化，见下一节*/
#endif
    return 0;
}
```

初始化函数中主要是创建并添加/dev/tty 和/dev/console 设备文件对应的 device 实例及 cdev 实例，设备文件操作结构 file_operations 实例分别为 tty_fops 和 console_fops。

/dev/tty 表示当前进程关联（绑定）的 tty 设备（current->signal->tty），current->signal->tty 在打开终端设备文件时设置。

/dev/console 表示当前系统控制台，相关内容下一节再做介绍。

9.8.2 线路规则

线路规则是用户进程与终端设备之间的中间层，在两者之间实现数据传输，内核定义了多个线路规则实例。终端设备初始都关联默认的线路规则，但用户可以修改。

1 数据结构

线路规则（line discipline）是 TTY 文件层与 TTY 设备驱动层的中间层，主要负责 TTY 协议的实现。线路规则使用 **tty_ldisc_ops** 结构体描述，结构体中包含两类接口，一类是面向上层 TTY 文件层提供调用函数接口，另一类是面向下层提供 TTY 设备驱动调用的函数接口。

tty_ldisc_ops 结构体定义如下（/include/linux/tty_ldisc.h）：

```
struct tty_ldisc_ops {
    int   magic;     /*魔数*/
    char *name;      /*名称*/
    int   num;       /*线路规则实例编号*/
    int   flags;     /*标记，只定义了 LDISC_FLAG_DEFINED 标记位*/

    /*以下函数由上层调用（文件操作结构）*/
    int   (*open)(struct tty_struct *);     /*tty_ldisc_ops 实例与 tty_struct 实例关联时调用*/
    void (*close)(struct tty_struct *);   /*tty_ldisc_ops 实例与 tty_struct 实例断开关联时调用*/
    void (*flush_buffer)(struct tty_struct *tty);     /*冲刷数据缓存区*/
    ssize_t    (*chars_in_buffer)(struct tty_struct *tty);          /*输入字符队列长度*/
    ssize_t    (*read)(struct tty_struct *tty, struct file *file,unsigned char __user *buf, size_t nr);
                                                          /*从 tty 缓存中读数据给用户进程*/
    ssize_t    (*write)(struct tty_struct *tty, struct file *file,const unsigned char *buf, size_t nr);
                                                        /*从用户进程向 tty 缓存写数据*/
    int   (*ioctl)(struct tty_struct *tty, struct file *file,unsigned int cmd, unsigned long arg);   /*设备控制*/
    long (*compat_ioctl)(struct tty_struct *tty, struct file *file,unsigned int cmd, unsigned long arg);
    void (*set_termios)(struct tty_struct *tty, struct ktermios *old); /*通知线路规则 termios 发生了改变*/
```

```
    unsigned int (*poll)(struct tty_struct *, struct file *,struct poll_table_struct *);   /*查询设备状态*/
    int   (*hangup)(struct tty_struct *tty);   /*停止 IO 操作*/

    /*以下函数由下层调用（终端设备驱动层）*/
    void (*receive_buf)(struct tty_struct *, const unsigned char *cp,char *fp, int count);
                                                    /*设备驱动层通知线路规则设备收到了数据*/
    void (*write_wakeup)(struct tty_struct *);      /*通知线路规则需要发送更多数据给设备驱动*/
    void (*dcd_change)(struct tty_struct *, unsigned int);
    void (*fasync)(struct tty_struct *tty, int on);     /*信号驱动 IO 使能或关闭时通知线路规则*/
    int   (*receive_buf2)(struct tty_struct *, const unsigned char *cp,char *fp, int count);
                                                    /*发送接收到的数据给线路规则*/
    struct   module *owner;      /*模块指针*/
    int refcount;
};
```

tty_ldisc_ops 结构体中主要包含对上层和对下层的操作函数接口，以上已做了注释，不再介绍了。

内核对每个线路规则实例赋予一个编号，编号定义在/include/uapi/linux/tty.h 头文件内：

```
#define NR_LDISCS          30     /*线路规则最大数量*/

#define   N_TTY        0        /*默认线路规则*/
#define   N_SLIP       1
#define   N_MOUSE      2
#define   N_PPP        3
#define   N_STRIP      4
#define   N_AX25       5
#define   N_X25        6    /* X.25 async */
#define   N_6PACK      7
#define   N_MASC       8    /* Reserved for Mobitex module <kaz@cafe.net> */
#define   N_R3964      9    /* Reserved for Simatic R3964 module */
#define   N_PROFIBUS_FDL     10   /* Reserved for Profibus */
#define   N_IRDA             11   /* Linux IrDa - http://irda.sourceforge.net/ */
#define   N_SMSBLOCK         12   /* SMS block mode - for talking to GSM data */
                                  /* cards about SMS messages */
#define   N_HDLC             13   /* synchronous HDLC */
#define   N_SYNC_PPP         14   /* synchronous PPP */
#define   N_HCI              15   /* Bluetooth HCI UART */
#define   N_GIGASET_M101     16   /* Siemens Gigaset M101 serial DECT adapter */
#define   N_SLCAN            17   /* Serial / USB serial CAN Adaptors */
#define   N_PPS              18   /* Pulse per Second */
#define   N_V253             19   /* Codec control over voice modem */
#define   N_CAIF             20        /* CAIF protocol for talking to modems */
#define   N_GSM0710          21   /* GSM 0710 Mux */
#define   N_TI_WL            22   /* for TI's WL BT, FM, GPS combo chips */
```

```
#define   N_TRACESINK    23   /* Trace data routing for MIPI P1149.7 */
#define   N_TRACEROUTER     24   /* Trace data routing for MIPI P1149.7 */
#define   N_NCI                    25   /* NFC NCI UART */
```

内核在/drivers/tty/tty_ldisc.c 文件内定义了指针数组，用于管理 tty_ldisc_ops 实例：

```
static struct tty_ldisc_ops *tty_ldiscs[NR_LDISCS];
```

其中 **N_TTY** 号线路规则由内核通用代码实现，是终端设备默认使用的线路规则。

2 注册线路规则

注册线路规则的函数为 **tty_register_ldisc()**，函数定义如下（/drivers/tty/tty_ldisc.c）：

```
int tty_register_ldisc(int disc, struct tty_ldisc_ops *new_ldisc)
/*disc：线路规则编号，new_ldisc：指向线路规则实例*/
{
    unsigned long flags;
    int ret = 0;

    if (disc < N_TTY || disc >= NR_LDISCS)
        return -EINVAL;

    raw_spin_lock_irqsave(&tty_ldiscs_lock, flags);
    tty_ldiscs[disc] = new_ldisc;     /*关联到指针数组项*/
    new_ldisc->num = disc;       /*线路规则编号*/
    new_ldisc->refcount = 0;
    raw_spin_unlock_irqrestore(&tty_ldiscs_lock, flags);
    return ret;
}
```

注册线路规则实例时必须指定其编号，注册函数只是简单地将实例与对应的 tty_ldiscs[]指针数组项关联。

3 初始化

内核在/drivers/tty/n_*.c 文件内实现了各线路规则实例（蓝牙、网络等驱动中也有注册线路规则），根据配置选项确定是否编译入内核。其中在 n_tty.c 文件内实现的 **tty_ldisc_N_TTY** 线路规则（**N_TTY** 号线路规则）是终端设备默认采用的线路规则，永久编译入内核，线路规则实例如下：

```
struct tty_ldisc_ops tty_ldisc_N_TTY = {
    .magic                = TTY_LDISC_MAGIC,
    .name                 = "n_tty",
    .open                 = n_tty_open,
    .close                = n_tty_close,
    .flush_buffer         = n_tty_flush_buffer,
    .chars_in_buffer = n_tty_chars_in_buffer,
    .read                 = n_tty_read,
```

```
    .write              = n_tty_write,
    .ioctl              = n_tty_ioctl,
    .set_termios        = n_tty_set_termios,
    .poll               = n_tty_poll,
    .receive_buf        = n_tty_receive_buf,
    .write_wakeup       = n_tty_write_wakeup,
    .fasync             = n_tty_fasync,
    .receive_buf2       = n_tty_receive_buf2,
};      /*线路规则中函数请读者自行阅读*/
```

在内核初始化阶段将注册 **tty_ldisc_N_TTY** 线路规则，函数调用关系为：start_kernel()->console_init()，**console_init()**函数定义如下（/drivers/tty/tty_io.c）：

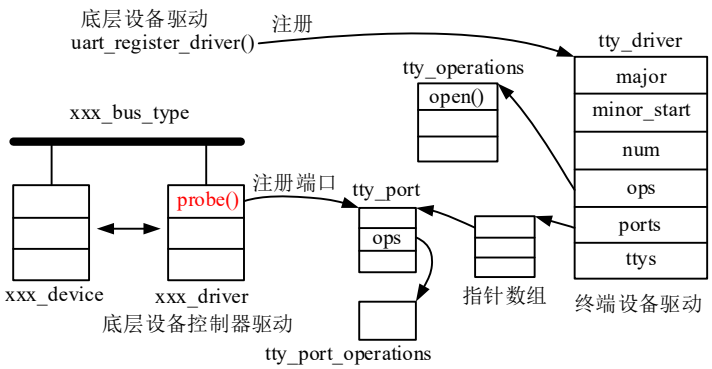
```
void __init console_init(void)
{
    initcall_t *call;

    tty_ldisc_begin();      /*注册 tty_ldisc_N_TTY 实例，/drivers/tty/tty_ldisc.c*/

    call = __con_initcall_start;
    while (call < __con_initcall_end) {     /*调用 console_initcall(fn)声明的初始化函数*/
        (*call)();
        call++;
    }
}
```

9.8.3 终端设备驱动

终端设备是按功能分类的一类设备，具体有多种设备可充当终端设备，如串口。每种类型的终端设备其驱动由 tty_driver 结构体表示，tty_driver 结构体关联 tty_operations 结构体用于实现底层硬件设备的数据传输以及设备控制。

终端设备中的一个通道，称为端口（或者说设备）由 tty_port 结构体表示，关联到 tty_driver 实例。例如，系统中可能有多个串口通道，每个通道就是一个端口。

终端设备驱动框架如下图所示：



终端设备驱动程序需要定义 tty_driver 和 tty_operations 实例，在驱动初始化函数中需要注册 tty_driver 实例（关联 tty_operations 实例）。

在设备控制器驱动的 probe()函数中需检测系统可用的终端通道，向 tty_driver 驱动注册端口 tty_port 实例，并同时创建和添加对应的 cdev 和 devcie 实例（含创建设备文件），随后用户进程就可以通过设备文件操作终端设备了。

1 数据结构

tty_driver 表示终端设备驱动，通常每个 tty_driver 实例适用一类终端设备。tty_driver 实例中需指定适用设备的主设备号（也可不指定）、起始从设备号、从设备数量、设备文件名称等。

在注册 tty_driver 实例时将为终端设备创建并添加表示字符设备的 cdev 实例，并根据标记取值为每个从设备创建并添加 device 实例，以及创建设备文件等。

tty_driver 结构体定义在/include/linux/tty_driver.h 头文件内：

```
struct tty_driver {
    int    magic;          /*魔数*/
    struct kref kref;      /*引用计数*/
    struct cdev *cdevs;    /*指向 cdev 实例（数组）*/
    struct module  *owner;
    const char     *driver_name;      /*驱动名称*/
    const char     *name;      /*名称，用于设备文件名称*/
    int    name_base;    /*设备文件名称中起始编号（通常为 0），不是起始从设备号*/
    int    major;          /*主设备号*/
    int    minor_start;    /*起始从设备号*/
    unsigned int    num;      /*支持的通道数量（从设备数量）*/
    short      type;          /*tty 设备驱动类型*/
    short      subtype;       /*子类型*/
    struct ktermios    init_termios;   /*设备属性*/
    unsigned long flags;               /*标记成员*/
    struct proc_dir_entry *proc_entry;      /* /proc fs entry */
    struct tty_driver *other;      /*PTY driver 专用*/


    struct tty_struct **ttys;     /*指向 tty_struct 指针数组，管理打开设备的 tty_struct 实例*/
    struct tty_port **ports;      /*端口指针数组，一个端口就是一个终端设备*/
    struct ktermios **termios;
    void *driver_state;

    const struct tty_operations *ops;     /*tty_operations 实例指针*/
    struct list_head tty_drivers;      /*将实例添加到 tty_drivers 全局双链表*/
};
```

tty_driver 结构体主要成员简介如下：

●**cdevs**：指向 cdev 实例或数组，在注册 tty_driver 实例或注册端口时，添加到字符设备驱动数据库。

●**major、minor_start、num**：驱动所支持设备的主设备号、起始从设备号和从设备号数量，在打开设备文件时，由此查找对应的 tty_driver 实例。

●**tty_drivers**：双链表成员，全局双链表 tty_drivers 用于管理注册的 tty_driver 实例。

●**ttys**：指向 tty_struct 指针数组，每个打开的端口关联一个 tty_struct 实例，见下文。

●**flags**：标记成员，取值如下：

```
#define   TTY_DRIVER_INSTALLED          0x0001    /*tty_driver 实例已注册*/
#define   TTY_DRIVER_RESET_TERMIOS      0x0002
#define   TTY_DRIVER_REAL_RAW           0x0004
#define   TTY_DRIVER_DYNAMIC_DEV        0x0008    /*是否动态创建从设备 device 实例*/
#define   TTY_DRIVER_DEVPTS_MEM         0x0010    /*需要伪终端内存*/
#define   TTY_DRIVER_HARDWARE_BREAK     0x0020
#define   TTY_DRIVER_DYNAMIC_ALLOC      0x0040   /*是否所有从设备共用一个 cdev 实例*/
#define   TTY_DRIVER_UNNUMBERED_NODE    0x0080   /*设备文件名后面是否不加序号，如 tty1*/
```

●**type、subtype**：驱动类型、子类型，定义如下：

```
#define   TTY_DRIVER_TYPE_SYSTEM        0x0001      /*系统*/
#define   TTY_DRIVER_TYPE_CONSOLE       0x0002      /*控制台*/
#define   TTY_DRIVER_TYPE_SERIAL        0x0003      /*串口*/
#define   TTY_DRIVER_TYPE_PTY           0x0004      /*PTY*/
#define   TTY_DRIVER_TYPE_SCC           0x0005     /* scc driver */
#define   TTY_DRIVER_TYPE_SYSCONS       0x0006

/*系统类型驱动子类型*/
#define   SYSTEM_TYPE_TTY               0x0001
#define   SYSTEM_TYPE_CONSOLE           0x0002
#define   SYSTEM_TYPE_SYSCONS           0x0003
#define   SYSTEM_TYPE_SYSPTMX           0x0004

/* pty 子类型*/
#define   PTY_TYPE_MASTER             0x0001
#define   PTY_TYPE_SLAVE              0x0002

/*串口设备子类型*/
#define   SERIAL_TYPE_NORMAL      1
```

●**init_termios**：ktermios 结构体成员，结构体定义如下（/include/uapi/asm-generic/termbits.h）：

```
struct ktermios {               /*记录终端各项属性*/
    tcflag_t c_iflag;           /*输入模式标记*/
    tcflag_t c_oflag;           /*输出模式标记*/
    tcflag_t c_cflag;           /*控制模式标记*/
    tcflag_t c_lflag;           /*控制终端输入的用户界面标记*/
    cc_t c_line;                /*线路规则*/
    cc_t c_cc[NCCS];            /*控制字符*/
    speed_t c_ispeed;           /*输入速率*/
    speed_t c_ospeed;           /*输出速率*/
};
```

ktermios 结构体标记了终端设备的各项属性，用户进程可通过系统调用设置，各成员标记位定义也在

同一头文件内。

●**ports**：指向 tty_port 结构体指针数组，tty_port 结构体表示一个端口（从设备），详见下文。

●**ops**：指向 **tty_operations** 结构体，用于实现底层数据传输和设备控制等，结构体定义如下：

```
struct tty_operations {                  /*/include/linux/tty_driver.h*/
    struct tty_struct * (*lookup)(struct tty_driver *driver,struct inode *inode, int idx);
                                                          /*由 idx 查找 tty_struct 实例*/
    int    (*install)(struct tty_driver *driver, struct tty_struct *tty);     /*tty_struct 关联 tty_driver 时调用*/
    void (*remove)(struct tty_driver *driver, struct tty_struct *tty);   /*断开 tty_struct 与 tty_driver 关联*/
    int    (*open)(struct tty_struct * tty, struct file * filp);
                                                  /*打开设备文件时调用，激活设备，申请中断等*/
    void (*close)(struct tty_struct * tty, struct file * filp);       /*关闭设备文件时调用*/
    void (*shutdown)(struct tty_struct *tty);
    void (*cleanup)(struct tty_struct *tty);
    int    (*write)(struct tty_struct * tty,const unsigned char *buf, int count); /*写入一串字符到终端设备*/
    int    (*put_char)(struct tty_struct *tty, unsigned char ch);     /*写单个字符到终端设备*/
    void (*flush_chars)(struct tty_struct *tty);   /*刷出 tty_struct 缓存数据至设备*/
    int    (*write_room)(struct tty_struct *tty);
    int    (*chars_in_buffer)(struct tty_struct *tty);
    int    (*ioctl)(struct tty_struct *tty,unsigned int cmd, unsigned long arg);     /*设备控制*/
    long (*compat_ioctl)(struct tty_struct *tty,unsigned int cmd, unsigned long arg);
    void (*set_termios)(struct tty_struct *tty, struct ktermios * old);   /*设置 ktermios*/
    void (*throttle)(struct tty_struct * tty);   /*通知 tty_driver 缓存已满*/
    void (*unthrottle)(struct tty_struct * tty);
    void (*stop)(struct tty_struct *tty);   /*通知 tty_driver 停止向设备发送数据*/
    void (*start)(struct tty_struct *tty);   /*通知 tty_driver 继续向设备发送数据*/
    void (*hangup)(struct tty_struct *tty);     /*通知 tty_driver 挂起设备*/
    int (*break_ctl)(struct tty_struct *tty, int state);
    void (*flush_buffer)(struct tty_struct *tty);
    void (*set_ldisc)(struct tty_struct *tty);
    void (*wait_until_sent)(struct tty_struct *tty, int timeout);     /*等待设备发送完数据*/
    void (*send_xchar)(struct tty_struct *tty, char ch);
    int (*tiocmget)(struct tty_struct *tty);
    int (*tiocmset)(struct tty_struct *tty,unsigned int set, unsigned int clear);
    int (*resize)(struct tty_struct *tty, struct winsize *ws);
    int (*set_termiox)(struct tty_struct *tty, struct termiox *tnew);
    int (*get_icount)(struct tty_struct *tty,struct serial_icounter_struct *icount);
#ifdef CONFIG_CONSOLE_POLL
    int (*poll_init)(struct tty_driver *driver, int line, char *options);
    int (*poll_get_char)(struct tty_driver *driver, int line);
    void (*poll_put_char)(struct tty_driver *driver, int line, char ch);
#endif
    const struct file_operations *proc_fops;
};
```

2 创建终端驱动

终端设备驱动程序需要实现特定于设备类型的 tty_operations 实例，调用接口函数 **tty_alloc_driver()**创建驱动 tty_driver 实例，并将 tty_operations 实例赋予 tty_driver 实例，最后调用接口函数 **tty_register_driver()** 注册 tty_driver 实例。

tty_alloc_driver()函数用于分配一个 tty_driver 实例，函数定义如下（/include/linux/tty_driver.h）：

```
#define tty_alloc_driver(lines, flags) \
        __tty_alloc_driver(lines, THIS_MODULE, flags)
```

lines 表示从设备号数量，flags 为标记，取值为 TTY_DRIVER_*（见上文）。

__tty_alloc_driver()函数定义如下（/drivers/tty/tty_io.c）：

```
struct tty_driver *__tty_alloc_driver(unsigned int lines, struct module *owner,unsigned long flags)
{
    struct tty_driver *driver;
    unsigned int cdevs = 1;   /*初始化只创建一个 cdev 实例*/
    int err;

    if (!lines || (flags & TTY_DRIVER_UNNUMBERED_NODE && lines > 1))
        return ERR_PTR(-EINVAL);

    driver = kzalloc(sizeof(struct tty_driver), GFP_KERNEL);     /*分配 tty_driver 实例*/
    if (!driver)
        return ERR_PTR(-ENOMEM);

    kref_init(&driver->kref);
    driver->magic = TTY_DRIVER_MAGIC;     /*魔数*/
    driver->num = lines;     /*从设备号数量*/
    driver->owner = owner;
    driver->flags = flags;

    if (!(flags & TTY_DRIVER_DEVPTS_MEM)) {
        driver->ttys = kcalloc(lines, sizeof(*driver->ttys),GFP_KERNEL);/*分配 tty_struct 指针数组*/
        driver->termios = kcalloc(lines, sizeof(*driver->termios),GFP_KERNEL);
        ...
    }

    /*没有设置 TTY_DRIVER_DYNAMIC_ALLOC 标记，分配 lines 个 cdev 实例*/
    if (!(flags & TTY_DRIVER_DYNAMIC_ALLOC)) {   /*为每个从设备创建 cdev 实例*/
        driver->ports = kcalloc(lines, sizeof(*driver->ports),GFP_KERNEL);
                                        /*为从设备分配 tty_port 结构体指针数组*/
        ...
        cdevs = lines;
    }
```

```
    driver->cdevs = kcalloc(cdevs, sizeof(*driver->cdevs), GFP_KERNEL); /*为设备分配 cdev 实例*/
    ...
    return driver;
    ...
}
```

__tty_alloc_driver()函数的主要工作是创建并初始化 tty_driver 实例，为设备（端口）创建 cdev 实例。如果标记参数 flags 设置了 TTY_DRIVER_DYNAMIC_ALLOC 标记位，则所有从设备共用一个 cdev 实例，否则为每个从设备创建 cdev 实例。

3 注册设备驱动

终端设备驱动在创建了 tty_driver 实例后，还需要对实例进行设置，其中最重要的就是调用接口函数 **tty_set_operations**(struct tty_driver *driver,const struct tty_operations *op)将 tty_operations 实例指针赋予驱动 tty_driver 实例。最后，终端设备驱动需要调用 **tty_register_driver()**函数注册 tty_driver 实例。

tty_register_driver()函数定义如下（/drivers/tty/tty_io.c）：

```
int tty_register_driver(struct tty_driver *driver)
{
    int error;
    int i;
    dev_t dev;
    struct device *d;

    if (!driver->major) {    /*主设备号为 0，动态分配主设备号*/
        error = alloc_chrdev_region(&dev, driver->minor_start,driver->num, driver->name);
        ...
    } else {    /*已指定主设备号*/
        dev = MKDEV(driver->major, driver->minor_start);          /*生成起始设备号*/
        error = register_chrdev_region(dev, driver->num, driver->name);   /*申请设备号*/
    }
    ...          /*错误处理*/
    if (driver->flags & TTY_DRIVER_DYNAMIC_ALLOC) {   /*所有从设备共用一个 cdev 实例*/
        error = tty_cdev_add(driver, dev, 0, driver->num);     /*添加 cdev 实例（只有一个）*/
        ...
    }

    mutex_lock(&tty_mutex);
    list_add(&driver->tty_drivers, &tty_drivers);     /*将 tty_driver 实例添加到全局双链表*/
    mutex_unlock(&tty_mutex);

    if (!(driver->flags & TTY_DRIVER_DYNAMIC_DEV)) {   /*创建并注册从设备 device 实例*/
        for (i = 0; i < driver->num; i++) {
            d = tty_register_device(driver, i, NULL);
                    /*创建并注册表示设备的 device 实例，创建设备文件*/
```

```
                ...
            }
        }
        proc_tty_register_driver(driver);
        driver->flags |= TTY_DRIVER_INSTALLED;   /*tty_driver 实例已注册*/
        return 0;
        ...
}
```

注册 tty_driver 实例函数主要工作如下：

（1）为终端设备申请设备号，或动态分配主设备号。

（2）如果驱动设置了 TTY_DRIVER_DYNAMIC_ALLOC 标记位，则调用 **tty_cdev_add()**函数设置并向内核添加适用于所有从设备的 cdev 实例（含对其文件操作结构实例指针赋值）。若没有设置此标记位，则跳过本步骤。

（3）将 tty_driver 实例插入到全局双链表头部。

（4）如果驱动没有设置 TTY_DRIVER_DYNAMIC_DEV 标记位，则调用 **tty_register_device()**函数为每个从设备创建并添加表示从设备的 device 实例并创建设备文件。若没有设置此标记位则跳过本步骤。

由上可知，TTY_DRIVER_DYNAMIC_ALLOC 标记位表示是否所有从设备共用同一个 cdev 实例，置位则在注册 tty_driver 实例时初始化并添加唯一的 cdev 实例。若没有设置此标记位则表示每个从设备对应一个 cdev 实例，在注册端口（设备）时初始化并添加表示设备的 cdev 实例。

TTY_DRIVER_DYNAMIC_DEV 标记位表示是否动态为每个从设备创建并添加 device 实例，如果此标记位置位，则在注册端口（设备）时创建并添加表示设备的 device 实例。如果没有设置此标记位，则在注册 tty_driver 实例时为每个从设备创建并添加 device 实例。

下面将分别介绍 tty_cdev_add()和 tty_register_device()函数的实现。

■添加字符设备驱动

tty_cdev_add()函数用于初始化和添加设备对应的 cdev 实例，函数定义如下（/drivers/tty/tty_io.c）：

```
static int tty_cdev_add(struct tty_driver *driver, dev_t dev,unsigned int index, unsigned int count)
/*dev：设备号，index：在驱动 cdev 实例数组中的索引值，count：表示的从设备号数量*/
{
    cdev_init(&driver->cdevs[index], &tty_fops);     /*文件操作结构实例为 tty_fops*/
    driver->cdevs[index].owner = driver->owner;
    return cdev_add(&driver->cdevs[index], dev, count);     /*添加 cdev 实例*/
}
```

若参数 index 为 0，表示只有一个 cdev 实例。如果驱动没有设置 TTY_DRIVER_DYNAMIC_ALLOC 标记位，则在注册端口（设备）的 tty_port_register_device()、tty_register_device()等函数中创建并添加 cdev 实例。

■注册终端设备

在注册 tty_driver 实例时，如果实例没有设置 TTY_DRIVER_DYNAMIC_DEV 标记位，则对每个从设备调用以下函数，创建/添加 device 实例并创建设备文件，函数定义如下（/drivers/tty/tty_io.c）：

```
struct device *tty_register_device(struct tty_driver *driver, unsigned index,struct device *device)
```

```
/*
*index：设备序号，驱动支持的设备从 0 顺序编号，(driver->minor_start+ index)为从设备号。
*device：父设备。
*/
{
        return tty_register_device_attr(driver, index, device, NULL, NULL);
}

struct device *tty_register_device_attr(struct tty_driver *driver,    \
                unsigned index, struct device *device,void *drvdata,const struct attribute_group **attr_grp)
{
        char name[64];
        dev_t devt = MKDEV(driver->major, driver->minor_start) + index;      /*合成设备号*/
        struct device *dev = NULL;
        int retval = -ENODEV;
        bool cdev = false;

        if (index >= driver->num) {
                ...
        }

        if (driver->type == TTY_DRIVER_TYPE_PTY)    /*伪终端驱动*/
                pty_line_name(driver, index, name);
        else
                tty_line_name(driver, index, name);         /*/drivers/tty/tty_io.c*/
                        /*由 driver->name 生成设备文件名称，保存于 name[]数组*/

        if (!(driver->flags & TTY_DRIVER_DYNAMIC_ALLOC)) {      /*每个从设备对应一个 cdev 实例*/
                retval = tty_cdev_add(driver, devt, index, 1);      /*设置/添加从设备的 cdev 实例*/
                if (retval)
                        goto error;
                cdev = true;
        }

        dev = kzalloc(sizeof(*dev), GFP_KERNEL);      /*创建 device 实例*/
        ...
        dev->devt = devt;       /*设备号*/
        dev->class = tty_class;      /*设备类*/
        dev->parent = device;       /*父设备*/
        dev->release = tty_device_create_release;
        dev_set_name(dev, "%s", name);       /*设备名称，用于设备文件名称，名称来自于 driver->name*/
        dev->groups = attr_grp;          /*此处为 NULL*/
        dev_set_drvdata(dev, drvdata);    /*drvdata 此处为 NULL*/
```

```
    retval = device_register(dev);     /*注册设备，创建设备文件，例如：/dev/tty1*/
    ...
    return dev;
    ...
}
```

tty_register_device_attr()函数主要工作如下：

（1）由 driver->name 生成设备文件名称，保存于 name[]数组。

（2）如果驱动没有设置 TTY_DRIVER_DYNAMIC_ALLOC 标记，则调用 tty_cdev_add()函数为从设备设置并添加 cdev 实例，否则跳过本步骤。

（3）为从设备创建、设置并注册 device 实例，将会创建设备文件。

4 注册端口

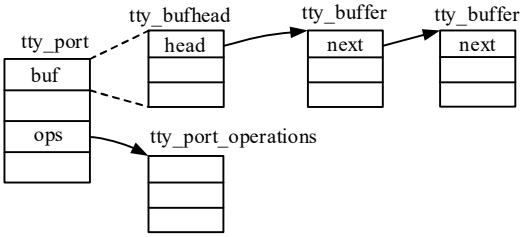
底层硬件设备的每个通道作为一个终端设备（端口）由 tty_port 结构体表示。在分配 tty_driver 实例时，如果实例标记成员没有设置 TTY_DRIVER_DYNAMIC_DEV 标记位（默认没有设置），将为终端设备分配 tty_port 结构体指针数组，并赋予 tty_driver->ports 成员。

底层硬件设备控制器驱动的 probe()函数需要为每个通道创建并注册 tty_port 实例。

tty_port 结构体定义在/include/linux/tty.h 头文件。

```
struct tty_port {
    struct tty_bufhead  buf;        /* Locked internally */
    struct tty_struct   *tty;       /*关联的 tty_struct 实例*/
    struct tty_struct   *itty;          /* internal back ptr */
    const struct tty_port_operations *ops;  /*端口操作结构*/
    spinlock_t          lock;       /* Lock protecting tty field */
    int             blocked_open;   /* Waiting to open */
    int             count;          /* Usage count */
    wait_queue_head_t   open_wait;  /* Open waiters，等待队列*/
    wait_queue_head_t   close_wait; /* Close waiters，等待队列 */
    wait_queue_head_t   delta_msr_wait;  /* Modem status change，等待队列 */
    unsigned long       flags;          /*标记，TTY flags ASY_*/
    unsigned char       console:1,      /*端口是否是控制台*/
                    low_latency:1;      /* optional: tune for latency */
    struct mutex        mutex;          /* Locking */
    struct mutex        buf_mutex;      /* Buffer alloc lock */
    unsigned char       *xmit_buf;      /*可选的缓存*/
    unsigned int        close_delay;    /* Close port delay */
    unsigned int        closing_wait;   /* Delay for output */
    int             drain_delay;    /* Set to zero if no pure time
                                       based drain is needed else
                                       set to size of fifo */
    struct kref         kref;       /* Ref counter */
};
```

tty_port 结构体主要成员简介如下：



●**buf**：tty_bufhead 结构体成员，定义如下（用于接收数据）：

```
struct tty_bufhead {
    struct tty_buffer *head;      /*指向 tty_buffer 实例链表头实例*/
    struct work_struct work;      /*将缓存中数据提交到线路规则，执行函数为 flush_to_ldisc()*/
    struct mutex        lock;
    atomic_t      priority;
    struct tty_buffer sentinel;
    struct llist_head free;          /*释放的 tty_buffer 实例双链表*/
    atomic_t      mem_used;      /* In-use buffers excluding free list */
    int           mem_limit;
    struct tty_buffer *tail;    /*末尾 tty_buffer 实例*/
};
```

tty_bufhead 结构体是一个缓存队列头（单链表），队列成员为 tty_buffer 结构体实例，定义如下：

```
struct tty_buffer {
    union {
        struct tty_buffer *next;
        struct llist_node free;
    };
    int used;
    int size;
    int commit;
    int read;
    int flags;
    /*数据*/
    unsigned long data[0];
};
```

tty_buffer 结构体用于缓存通道接收的数据，之后将由线路规则提交到 tty_struct 实例的读缓存区。

●**ops**：指向 tty_port_operations 结构体实例，表示端口操作接口，结构体定义如下：

```
struct tty_port_operations {
    int (*carrier_raised)(struct tty_port *port);
    void (*dtr_rts)(struct tty_port *port, int raise);
    void (*shutdown)(struct tty_port *port);
    int (*activate)(struct tty_port *port, struct tty_struct *tty);          /*激活端口*/
    void (*destruct)(struct tty_port *port);
};
```

具体终端设备驱动程序需要定义 tty_port 实例及其关联的 tty_port_operations 实例。

在/drivers/tty/tty_port.c 文件内，定义了向 tty_driver 实例注册端口的接口函数，供设备控制器驱动的 probe()函数调用，例如：

●void **tty_port_init**(struct tty_port *port)：初始化端口。

●void **tty_port_link_device**(struct tty_port *port,struct tty_driver *driver, unsigned index)：将端口关联到驱动 tty_driver 中的端口指针数据组（index 为数组项索引值）。

●struct device ***tty_port_register_device()**：关联端口到 tty_driver 实例，并添加对应的 cdev 和 device 实例，函数定义如下：

```
struct device *tty_port_register_device(struct tty_port *port,
                                        struct tty_driver *driver, unsigned index,struct device *device)
{
    tty_port_link_device(port, driver, index);      /*关联 tty_driver->ports 指针数组项*/
    return tty_register_device(driver, index, device);    /*添加 cdev 和 device 实例*/
}
```

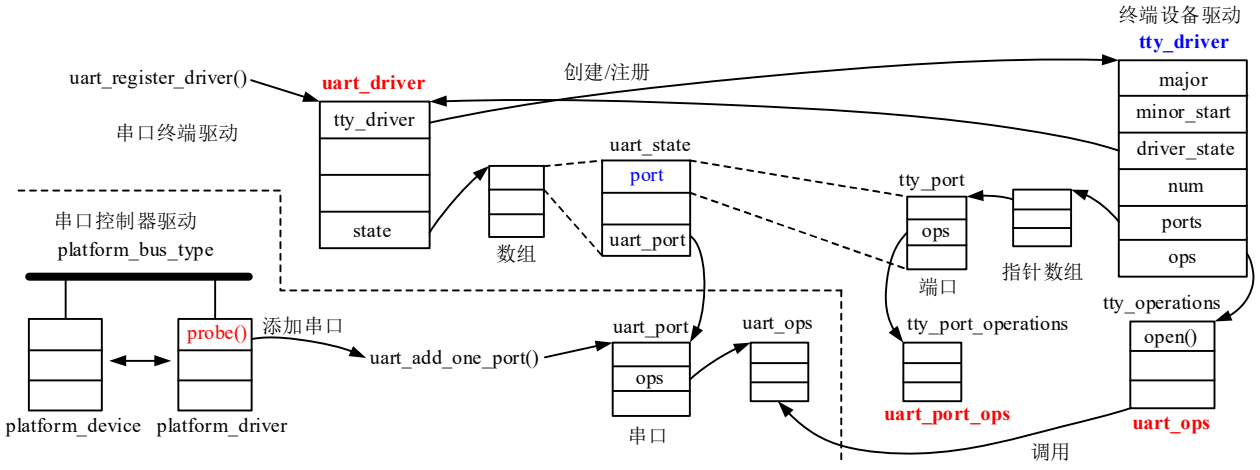
●struct device ***tty_port_register_device_attr**(struct tty_port *port,
struct tty_driver *driver, unsigned index,struct device ***device**, void *drvdata,
const struct attribute_group **attr_grp)：与 tty_port_register_device()类似，只不过带了设备属性。

9.8.4 串口终端设备驱动

对于嵌入式系统来说，最普遍采用的终端设备就是 UART 串行端口，简称串口，串口是终端设备的一个实例。本小节将简要介绍串口终端设备驱动的实现。

1 驱动框架

串口终端设备驱动框架如下图所示：



串口设备控制器假设挂载在平台总线，串口设备通常包含多个通道。为了与终端驱动中端口概念区分，串口通道就称它为串口，串口是端口的一个实例。

串口终端驱动程序需定义 uart_driver 结构体实例，其中包含的 uart_state 实例（数组）表示了串口信息，uart_state 结构体内嵌端口 tty_port 结构体成员。在模块加载（初始化）函数中，需要注册 uart_driver 实例。

在注册 uart_driver 实例的接口函数 uart_register_driver(struct uart_driver *drv)中将为 uart_driver 实例创

建、设置并注册对应的终端设备驱动 tty_driver 实例。tty_driver 实例中 ops 成员（tty_operations 指针）指向通用的 **uart_ops** 实例。uart_driver 实例中包含的 uart_state 实例（数组）中的 tty_port 实例也将关联到 tty_driver 实例，tty_port 实例 ops 成员指向 uart_port_ops 实例（tty_port_operations 结构体）。

串口控制器驱动的 probe()函数，从板级文件中获取串口信息，调用 uart_add_one_port(struct uart_driver *drv, struct uart_port *uport)接口函数向 uart_driver 实例添加表示串口的 uart_port 实例，uart_port 实例将关联到 uart_state 实例中 uart_port 指针成员，也就是建立串口 uart_port 实例与端口 tty_port 实例之间的对应关系。uart_port 结构体中 ops 成员指向串口操作 uart_ops 结构体实例。

tty_driver 实例关联 tty_operations 实例中的函数将调用串口操作 uart_ops 结构体中的函数，完成底层数据操作和控制。uart_ops 结构体实例需由串口控制器驱动程序实现。

串口终端驱动公共层代码在/drivers/tty/serial/serial_core.c 文件实现，主要实现 uart_driver 注册函数、添加 uart_port 端口函数、tty_operations 结构体实例 uart_ops、tty_port_operations 结构体实例 uart_port_ops 的定义，以及串口设备控制接口函数等。

串口终端设备驱动位于/drivers/tty/serial/目录下。

2 数据结构

uart_driver 结构体表示串口终端驱动，uart_port 结构体表示一个串口。uart_driver 和 uart_port 结构体都定义在/include/linux/serial_core.h 头文件。

■驱动

uart_driver 结构体定义如下：

```
struct uart_driver {          /*串口终端驱动*/
    struct module       *owner;
    const char          *driver_name;    /*驱动名称*/
    const char          *dev_name;       /*串口设备文件名称，"ttyS"*/
    int                 major;           /*主设备号*/
    int                 minor;           /*起始从设备号*/
    int                 nr;              /*串口数量，state 指向数组的项数*/
    struct console      *cons;           /*串口驱动注册的控制台，/include/linux/console.h*/

    struct uart_state   *state;          /*指向 uart_state 结构体数组，数组项数为 nr，包含串口信息*/
    struct tty_driver   *tty_driver;     /*指向 tty_driver 实例*/
};
```

uart_driver 结构体主要成员简介如下：

●**tty_driver**：指向对应的终端设备驱动 tty_driver 实例，在注册 uart_driver 实例时创建并注册。

●**state**：指向 uart_state 结构体数组，数组项数由 nr 成员决定，表示串口信息，结构体定义如下：

```
struct uart_state {
    struct tty_port     port;        /*tty_port 结构体成员*/
    enum uart_pm_state  pm_state;
    struct circ_buf     xmit;

    struct uart_port    *uart_port;  /*串口 uart_port 结构体指针*/
};
```

port 是 tty_port 结构体成员，表示终端驱动中端口，uart_port 是 uart_port 结构体指针，表示串口信息，结构体定义见下文。

■串口

串口 uart_port 结构体定义简列如下：

```
struct uart_port {
    spinlock_t          lock;           /* port lock */
    unsigned long       iobase;             /*IO 基地址*/
    unsigned char __iomem  *membase;            /* read/write[bwl] */
    unsigned int        (*serial_in)(struct uart_port *, int);
    void            (*serial_out)(struct uart_port *, int, int);
    void            (*set_termios)(struct uart_port *,struct ktermios *new,struct ktermios *old);
    void            (*set_mctrl)(struct uart_port *, unsigned int);
    int             (*startup)(struct uart_port *port);
    void            (*shutdown)(struct uart_port *port);
    void            (*throttle)(struct uart_port *port);
    void            (*unthrottle)(struct uart_port *port);
    int             (*handle_irq)(struct uart_port *);
    void            (*pm)(struct uart_port *, unsigned int state,unsigned int old);
    void            (*handle_break)(struct uart_port *);
    int             (*rs485_config)(struct uart_port *,struct serial_rs485 *rs485);
    unsigned int        irq;            /*中断号*/
    unsigned long       irqflags;       /*中断标志*/
    unsigned int        uartclk;        /*时钟*/
    unsigned int        fifosize;       /*发送 fifo 大小*/
    unsigned char       x_char;             /* xon/xoff char */
    unsigned char       regshift;       /*寄存器偏移地址*/
    unsigned char       iotype;             /*IO 访问类型*/
    unsigned char       unused1;
    ...     /*宏定义*/
    unsigned int        read_status_mask;       /* driver specific */
    unsigned int        ignore_status_mask;     /* driver specific */
    struct uart_state   *state;             /*指向对应的 uart_state 实例*/
    struct uart_icount  icount;             /* statistics */

    struct console      *cons;              /*控制台*/
#if defined(CONFIG_SERIAL_CORE_CONSOLE) || defined(SUPPORT_SYSRQ)
    unsigned long       sysrq;              /* sysrq timeout */
#endif

    /* flags must be updated while holding port mutex */
    upf_t               flags;
    ...     /*宏定义*/
```

```
    upstat_t          status;
    ...     /*宏定义*/
    int               hw_stopped;       /* sw-assisted CTS flow state */
    unsigned int      mctrl;            /* current modem ctrl settings */
    unsigned int      timeout;          /* character-based timeout */
    unsigned int      type;             /*串口类型*/
    const struct uart_ops     *ops;     /*串口操作结构*/
    unsigned int      custom_divisor;
    unsigned int      line;             /*串口序号*/
    unsigned int      minor;
    resource_size_t       mapbase;      /* for ioremap */
    resource_size_t       mapsize;
    struct device     *dev;             /*父设备 device 实例*/
    unsigned char     hub6;             /*8250 驱动私有数据*/
    unsigned char     suspended;
    unsigned char     irq_wake;
    unsigned char     unused[2];
    struct attribute_group    *attr_group;        /*端口属性*/
    const struct attribute_group **tty_groups;    /* all attributes (serial core use only) */
    struct serial_rs485       rs485;
    void              *private_data;    /* generic platform data pointer */
};
```

uart_port 结构体中 ops 成员指向串口操作 uart_ops 结构体，包含串口的底层操作函数。uart_ops 结构体实例由串口控制器驱动程序实现，通过控制器实现底层数据传输和控制，结构体定义如下：

```
struct uart_ops {
    unsigned int    (*tx_empty)(struct uart_port *);
    void        (*set_mctrl)(struct uart_port *, unsigned int mctrl);
    unsigned int    (*get_mctrl)(struct uart_port *);
    void        (*stop_tx)(struct uart_port *);     /*停止传输*/
    void        (*start_tx)(struct uart_port *);    /*开始传输*/
    void        (*throttle)(struct uart_port *);
    void        (*unthrottle)(struct uart_port *);
    void        (*send_xchar)(struct uart_port *, char ch);   /*发送字符*/
    void        (*stop_rx)(struct uart_port *);      /*接收停止*/
    void        (*enable_ms)(struct uart_port *);
    void        (*break_ctl)(struct uart_port *, int ctl);
    int         (*startup)(struct uart_port *);
    void        (*shutdown)(struct uart_port *);
    void        (*flush_buffer)(struct uart_port *);
    void        (*set_termios)(struct uart_port *, struct ktermios *new,struct ktermios *old);
    void        (*set_ldisc)(struct uart_port *, struct ktermios *);
    void        (*pm)(struct uart_port *, unsigned int state, unsigned int oldstate);
```

```
    const char      *(*type)(struct uart_port *);   /*返回串口类型*/
    void        (*release_port)(struct uart_port *);   /*释放串口*/
    int         (*request_port)(struct uart_port *);
    void        (*config_port)(struct uart_port *, int);   /*配置串口*/
    int         (*verify_port)(struct uart_port *, struct serial_struct *);
    int         (*ioctl)(struct uart_port *, unsigned int, unsigned long);
#ifdef CONFIG_CONSOLE_POLL
    int         (*poll_init)(struct uart_port *);
    void        (*poll_put_char)(struct uart_port *, unsigned char);
    int         (*poll_get_char)(struct uart_port *);
#endif
};
```

3 接口函数

串口终端驱动需定义 uart_driver 实例，并在初始化（或模块加载）函数调用 uart_register_driver(struct uart_driver *drv)函数注册 uart_driver 实例。在控制器驱动的 probe()函数中需获取串口硬件信息，添加表示可用串口的 uart_port 实例。下面介绍相关接口函数的实现。

■注册 uart_driver

注册串口终端驱动 uart_driver 实例的 **uart_register_driver()**函数定义如下：

```
int uart_register_driver(struct uart_driver *drv)
{
    struct tty_driver *normal;
    int i, retval;

    BUG_ON(drv->state);

    drv->state = kzalloc(sizeof(struct uart_state) * drv->nr, GFP_KERNEL);   /*分配 uart_state 数组*/
    ...     /*错误处理*/

    normal = alloc_tty_driver(drv->nr);     /*分配 tty_driver 实例*/
    ...     /*错误处理*/

    drv->tty_driver = normal;       /*指向 tty_driver 实例*/

    normal->driver_name    = drv->driver_name;      /*驱动名称*/
    normal->name       = drv->dev_name;     /*设备（文件）名称*/
    normal->major          = drv->major;    /*主设备号*/
    normal->minor_start    = drv->minor;    /*起始从设备号/
    normal->type       = TTY_DRIVER_TYPE_SERIAL;
    normal->subtype        = SERIAL_TYPE_NORMAL;
    normal->init_termios   = tty_std_termios;
```

```
    normal->init_termios.c_cflag = B9600 | CS8 | CREAD | HUPCL | CLOCAL;
    normal->init_termios.c_ispeed = normal->init_termios.c_ospeed = 9600;
    normal->flags        = TTY_DRIVER_REAL_RAW | TTY_DRIVER_DYNAMIC_DEV;
    normal->driver_state      = drv;       /*指向 uart_driver 实例*/
    tty_set_operations(normal, &uart_ops);   /*设置 tty_operations 指针成员为 uart_ops*/

    for (i = 0; i < drv->nr; i++) {              /*uart_state 中端口关联到 tty_driver，并初始化*/
        struct uart_state *state = drv->state + i;
        struct tty_port *port = &state->port;        /*端口*/

        tty_port_init(port);        /*初始化端口，/drivers/tty/tty_port.c*/
        port->ops = &uart_port_ops;     /*设置端口操作结构*/
    }

    retval = tty_register_driver(normal);     /*注册 tty_driver 实例*/
    if (retval >= 0)
        return retval;
    ...
}
```

uart_register_driver()函数主要是为 uart_driver 实例创建、设置并注册对应的 tty_driver 实例。tty_driver 实例关联的 tty_operations 实例为 **uart_ops**。将 uart_driver 实例中 uart_state 数组项中端口关联到 tty_driver，并初始化，端口 tty_port 关联的 tty_port_operations 实例设为 **uart_port_ops**。

■添加串口

串口控制器驱动 probe()函数需要根据板级信息获取串口信息，创建并设置串口 uart_port 实例（含关联的 uart_ops 实例）并向串口终端驱动添加。添加串口 uart_port 实例的接口函数为 **uart_add_one_port()**，函数内将 uart_port 实例关联到 uart_driver 实例中 uart_state 数组项，并为数组项内嵌的端口 tty_port 实例创建、注册 cdev、device 实例等。

uart_add_one_port()函数定义如下（/drivers/tty/serial/serial_core.c）：

```
int uart_add_one_port(struct uart_driver *drv, struct uart_port *uport)
/*drv：串口终端驱动，uport：串口*/
{
    struct uart_state *state;
    struct tty_port *port;
    int ret = 0;
    struct device *tty_dev;
    int num_groups;

    BUG_ON(in_interrupt());

    if (uport->line >= drv->nr)
        return -EINVAL;
```

```
state = drv->state + uport->line;   /*串口对应的 uart_state 实例*/
port = &state->port;     /*uart_state 实例中内嵌 tty_port 结构体成员*/

mutex_lock(&port_mutex);
mutex_lock(&port->mutex);
...
state->uart_port = uport;           /*关联串口 uart_port 实例*/
uport->state = state;               /*关联 uart_state 实例*/

state->pm_state = UART_PM_STATE_UNDEFINED;
uport->cons = drv->cons;
uport->minor = drv->tty_driver->minor_start + uport->line;   /*从设备号*/

/*如果串口是控制台*/
if (!(uart_console(uport) && (uport->cons->flags & CON_ENABLED))) {
        spin_lock_init(&uport->lock);
        lockdep_set_class(&uport->lock, &port_lock_key);
}
if (uport->cons && uport->dev)
        of_console_check(uport->dev->of_node, uport->cons->name, uport->line);

uart_configure_port(drv, state, uport);     /*配置串口*/

num_groups = 2;
if (uport->attr_group)
        num_groups++;

uport->tty_groups = kcalloc(num_groups, sizeof(*uport->tty_groups),GFP_KERNEL);
...
uport->tty_groups[0] = &tty_dev_attr_group;
if (uport->attr_group)
        uport->tty_groups[1] = uport->attr_group;

/*创建/添加串口对应端口的 device（cdev）实例，创建设备文件等，端口已关联 tty_driver 实例*/
tty_dev = tty_port_register_device_attr(port, drv->tty_driver,
                                        uport->line, uport->dev, port, uport->tty_groups);      /*见上文*/
if (likely(!IS_ERR(tty_dev))) {
        device_set_wakeup_capable(tty_dev, 1);
} else {
        dev_err(uport->dev, "Cannot register tty device on line %d\n",uport->line);
}
uport->flags &= ~UPF_DEAD;
```

```
out:
    mutex_unlock(&port->mutex);
    mutex_unlock(&port_mutex);

    return ret;
}
```

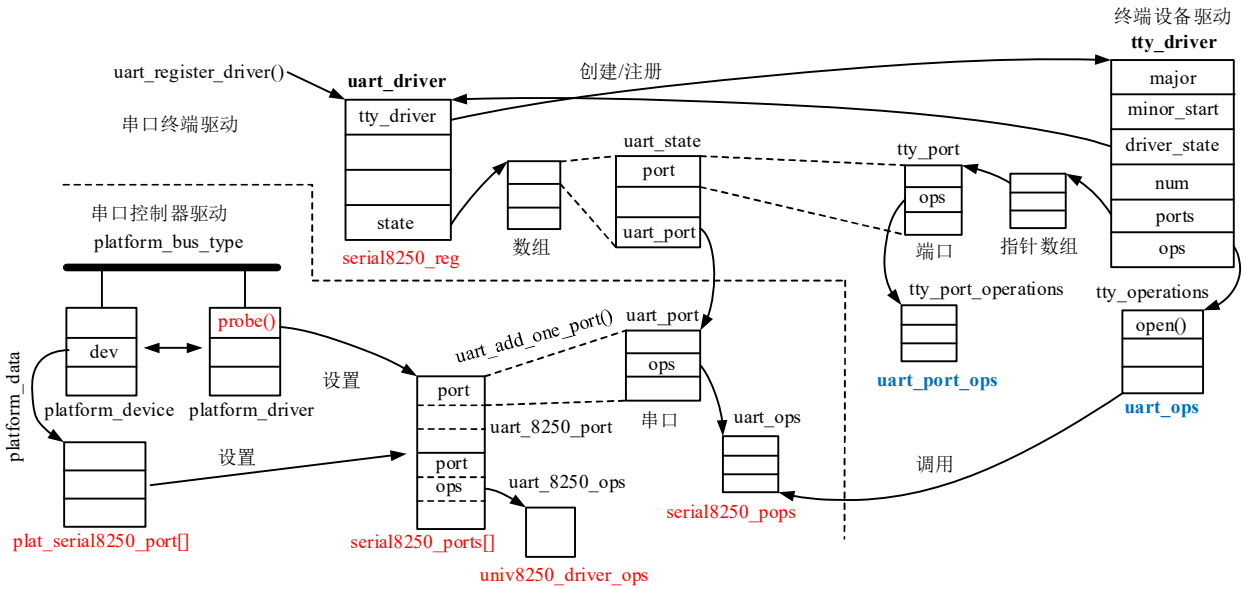
串口控制器驱动程序需定义表示串口的 uart_port 实例，实现底层串口操作结构 **uart_ops** 实例，并赋予 uart_port 实例，然后才能调用 uart_add_one_port()函数向串口终端驱动添加串口。

在添加串口的操作函数中将为串口创建、添加 cdev/device 实例，并自动创建设备文件，设备文件名称来源于 tty_driver->name（uart_driver->dev_name）后接串口序号，如 ttyS0、ttyS1 等。

4 驱动示例

下面以/drivers/tty/serial/8250/8250_core.c 文件内实现的 8250 串口控制器（芯片）驱动程序为例简要说明串口驱动程序的实现，假设串口控制器挂载在平台总线上。

8250 串口驱动框架如下图所示：



串口终端设备驱动定义了 uart_driver 实例 **serial8250_reg**，在初始化（模块加载）函数中调用接口函数 **uart_register_driver()**函数注册 serial8250_reg 实例。注册函数将创建和注册终端设备驱动 tty_driver 实例，serial8250_reg 实例中 uart_state 数组项中内嵌的 tty_port 结构体成员（端口）关联到 tty_driver 实例。

驱动中定义了 uart_8250_port 结构体数组 serial8250_ports[]，结构体中内嵌 uart_port 结构体成员，用于表示 8250 串口。在初始化函数中将初始化并添加 serial8250_ports[]数组中串口，以激活某些隐含的串口（应该不用这种方式了）。串口 uart_port 关联的 uart_ops 实例为 **serial8250_pops**，此实例中函数用于实现底层数据传输和串口控制。

在初始化函数中还创建并添加了表示串口控制器驱动的 platform_driver 实例 **serial8250_isa_driver**。串口控制器设备 platform_device 实例通过 plat_serial8250_port 结构体，向驱动传递串口通道的底层硬件信息。

serial8250_isa_driver 实例的 probe()函数将根据 plat_serial8250_port 结构体数组项设置 serial8250_ports[]数组项，并添加其中的串口 uart_port 结构体成员。

8250 串口终端设备驱动 uart_driver 结构体实例 serial8250_reg 定义如下：

```
static struct uart_driver serial8250_reg = {
    .owner              = THIS_MODULE,
```

```
    .driver_name        = "serial",
    .dev_name           = "ttyS",    /*串口终端设备文件名称为 ttySX*/
    .major              = TTY_MAJOR,      /*主设备号*/
    .minor              = 64,             /*起始从设备号*/
    .cons               = SERIAL8250_CONSOLE,   /*控制台*/
};
```

■初始化

8250 串口驱动中，由 uart_8250_port 结构体表示串口，定义如下（/include/linux/serial_8250.h）：

```
struct uart_8250_port {
    struct uart_port    port;           /*通用串口*/
    struct timer_list   timer;          /* "no irq" timer */
    struct list_head    list;           /* ports on this IRQ */
    unsigned short      capabilities;   /* port capabilities */
    ...
    struct uart_8250_dma    *dma;
    const struct uart_8250_ops *ops;
    ...         /*特有的回调函数*/
};
```

在/drivers/tty/serial/8250/8250_core.c 文件内定义了此结构体数组 serial8250_ports[UART_NR]。数组项数 UART_NR 值由配置选项 SERIAL_8250_NR_UARTS 确定，默认为 4。

8250 串口驱动初始化函数定义如下：

```
static int __init serial8250_init(void)
{
    int ret;
    /*nr_uarts 由 SERIAL_8250_RUNTIME_UARTS 选项确定（4），表示启动阶段可注册串口数*/
    if (nr_uarts == 0)
        return -ENODEV;

    serial8250_isa_init_ports();        /*初始化 serial8250_ports[]数组项中 uart_port 成员，并添加*/

    ...     /*输出信息*/

#ifdef CONFIG_SPARC
    ret = sunserial_register_minors(&serial8250_reg, UART_NR);
#else
    serial8250_reg.nr = UART_NR;            /*serial8250_reg 实例支持的串口数量*/
    ret = uart_register_driver(&serial8250_reg);        /*注册 uart_driver 结构体实例 serial8250_reg*/
#endif
    ...
    ret = serial8250_pnp_init();        /*没有选择 SERIAL_8250_PNP 选项为空*/
    ...
```

```
    serial8250_isa_devs = platform_device_alloc("serial8250",PLAT8250_DEV_LEGACY);
                                                    /*分配 platform_device 实例*/
    ...
    ret = platform_device_add(serial8250_isa_devs);        /*添加 platform_device 实例*/
    ...
    serial8250_register_ports(&serial8250_reg, &serial8250_isa_devs->dev);
            /*添加 serial8250_ports[]数组项内嵌串口，为对应端口创建/添加 cdev、device 实例*/

    ret = platform_driver_register(&serial8250_isa_driver);
                    /*注册串口控制器驱动 platform_driver 实例 serial8250_isa_driver*/
    ...
out:
    return ret;        /*成功返回 0*/
}
module_init(serial8250_init);          /*初始化或模块加载函数*/
module_exit(serial8250_exit);          /*serial8250_exit()函数源代码请读者自行阅读*/
```

在初始化函数中已经创建和注册了串口对应终端设备驱动的 tty_driver 实例，初始化了 serial8250_ports[]数组项中串口成员，并且添加了对应的端口 tty_port 实例，即关联到 tty_driver 实例，创建、添加 cdev、device 实例。

serial8250_ports[]数组项中串口 uart_port 结构体成员关联的 uart_ops 实例为 **serial8250_pops**，此实例中函数用于实现底层数据传输和串口控制。serial8250_pops 实例中函数通过对串口控制器寄存器的操作实现数据传输和控制，函数源代码请读者自行阅读。

■串口控制器驱动

串口控制器驱动需要从设备中获取串口通道信息，设置对应 serial8250_ports[]数组项，并添加其中串口 uart_port 结构体成员。

●通道信息

plat_serial8250_port 结构体用于平台代码向串口控制器驱动传递串口通道信息，结构体定义如下：

```
struct plat_serial8250_port {            /*/include/linux/serial_8250.h*/
    unsigned long iobase;            /* io base address */
    void __iomem *membase;       /**/
    resource_size_t     mapbase;      /*控制寄存器基址*/
    unsigned int    irq;          /*中断编号*/
    unsigned long irqflags;   /*中断标记*/
    unsigned int    uartclk;   /*时钟频率*/
    void            *private_data;
    unsigned char regshift;         /* register shift */
    unsigned char iotype;           /* UPIO_* */
    unsigned char hub6;
    upf_t           flags;          /* UPF_* flags */
```

```
    unsigned int    type;           /*串口类型*/
    ...
};
```

例如，龙芯 1B 平台代码中定义的 plat_serial8250_port 结构体数组如下：

```
#define LS1X_UART(_id)              \
    {                               \
        .mapbase= LS1X_UART ## _id ## _BASE,    \   /*控制寄存器基地址*/
        .irq        = LS1X_UART ## _id ## _IRQ,  \
        .iotype     = UPIO_MEM,                  \
        .flags      = UPF_IOREMAP | UPF_FIXED_TYPE, \
        .type       = PORT_16550A,               \
    }

static struct plat_serial8250_port    ls1x_serial8250_pdata[] = {       /*4 个通道*/
    LS1X_UART(0),
    LS1X_UART(1),
    LS1X_UART(2),
    LS1X_UART(3),
    {},
};

struct platform_device ls1x_uart_pdev = {                       /*串口控制器设备*/
    .name       = "serial8250",         /*匹配驱动 serial8250_isa_driver*/
    .id         = PLAT8250_DEV_PLATFORM,
    .dev        = {
        .platform_data = ls1x_serial8250_pdata,    /*指向 plat_serial8250_port 数组*/
    },
};
```

串口控制器驱动 platform_driver 实例 **serial8250_isa_driver** 定义如下：

```
static struct platform_driver serial8250_isa_driver = {
    .probe          = serial8250_probe,       /*探测函数，添加串口*/
    .remove         = serial8250_remove,
    .suspend  = serial8250_suspend,
    .resume         = serial8250_resume,
    .driver         = {
        .name       = "serial8250",         /*匹配设备*/
    },
};
```

●探测函数

驱动 serial8250_isa_driver 实例探测函数 serial8250_probe()中将扫描设备传递的 plat_serial8250_port 结

构体数组，用于初始化 **uart_8250_port[]**数组，并添加数组项中 port 成员（uart_port 结构体）。

serial8250_probe()函数代码简列如下：

```
static int serial8250_probe(struct platform_device *dev)
{
    struct plat_serial8250_port *p = dev_get_platdata(&dev->dev);   /*dev->dev->platform_data*/
    struct uart_8250_port uart;
    int ret, i, irqflag = 0;

    memset(&uart, 0, sizeof(uart));          /*uart_8250_port 实例清 0*/

    if (share_irqs)
        irqflag = IRQF_SHARED;

    for (i = 0; p && p->flags != 0; p++, i++) {     /*扫描 plat_serial8250_port[]数组*/
        uart.port.iobase      = p->iobase;          /*设置 uart_port 成员*/
        uart.port.membase  = p->membase;
        uart.port.irq         = p->irq;
        uart.port.irqflags    = p->irqflags;
        uart.port.uartclk     = p->uartclk;
        uart.port.regshift    = p->regshift;
        uart.port.iotype      = p->iotype;
        uart.port.flags       = p->flags;
        uart.port.mapbase     = p->mapbase;
        uart.port.hub6        = p->hub6;
        uart.port.private_data     = p->private_data;
        uart.port.type        = p->type;
        uart.port.serial_in   = p->serial_in;
        uart.port.serial_out = p->serial_out;
        uart.port.handle_irq= p->handle_irq;
        uart.port.handle_break    = p->handle_break;
        uart.port.set_termios      = p->set_termios;
        uart.port.pm          = p->pm;
        uart.port.dev         = &dev->dev;
        uart.port.irqflags    |= irqflag;
        ret = serial8250_register_8250_port(&uart);     /*注册串口*/
        ...
    }
    return 0;
}
```

serial8250_register_8250_port()函数用于注册 8250 串口，函数定义如下：

```
int serial8250_register_8250_port(struct uart_8250_port *up)
{
```

```
struct uart_8250_port *uart;
int ret = -ENOSPC;

if (up->port.uartclk == 0)
        return -EINVAL;

mutex_lock(&serial_mutex);

uart = serial8250_find_match_or_unused(&up->port);       /*查找 serial8250_ports[]中匹配数组项*/
if (uart && uart->port.type != PORT_8250_CIR) {          /*设置 serial8250_ports[]中匹配数组项*/
        if (uart->port.dev)
                uart_remove_one_port(&serial8250_reg, &uart->port);

        uart->port.iobase             = up->port.iobase;
        uart->port.membase            = up->port.membase;
        uart->port.irq                = up->port.irq;
        uart->port.irqflags           = up->port.irqflags;
        uart->port.uartclk            = up->port.uartclk;
        uart->port.fifosize           = up->port.fifosize;
        uart->port.regshift           = up->port.regshift;
        uart->port.iotype             = up->port.iotype;
        uart->port.flags              = up->port.flags | UPF_BOOT_AUTOCONF;
        uart->bugs                    = up->bugs;
        uart->port.mapbase            = up->port.mapbase;
        uart->port.mapsize            = up->port.mapsize;
        uart->port.private_data = up->port.private_data;
        uart->tx_loadsz               = up->tx_loadsz;
        uart->capabilities    = up->capabilities;
        uart->port.throttle   = up->port.throttle;
        uart->port.unthrottle       = up->port.unthrottle;
        uart->port.rs485_config = up->port.rs485_config;
        uart->port.rs485      = up->port.rs485;
        uart->dma             = up->dma;

        /* Take tx_loadsz from fifosize if it wasn't set separately */
        if (uart->port.fifosize && !uart->tx_loadsz)
                uart->tx_loadsz = uart->port.fifosize;

        if (up->port.dev)
                uart->port.dev = up->port.dev;

        if (skip_txen_test)
                uart->port.flags |= UPF_NO_TXEN_TEST;
```

```
        if (up->port.flags & UPF_FIXED_TYPE)
            uart->port.type = up->port.type;

        serial8250_set_defaults(uart);

        /* Possibly override default I/O functions.    */
        if (up->port.serial_in)
            uart->port.serial_in = up->port.serial_in;
        if (up->port.serial_out)
            uart->port.serial_out = up->port.serial_out;
        if (up->port.handle_irq)
            uart->port.handle_irq = up->port.handle_irq;
        /*    Possibly override set_termios call */
        if (up->port.set_termios)
            uart->port.set_termios = up->port.set_termios;
        if (up->port.set_mctrl)
            uart->port.set_mctrl = up->port.set_mctrl;
        if (up->port.startup)
            uart->port.startup = up->port.startup;
        if (up->port.shutdown)
            uart->port.shutdown = up->port.shutdown;
        if (up->port.pm)
            uart->port.pm = up->port.pm;
        if (up->port.handle_break)
            uart->port.handle_break = up->port.handle_break;
        if (up->dl_read)
            uart->dl_read = up->dl_read;
        if (up->dl_write)
            uart->dl_write = up->dl_write;

        if (serial8250_isa_config != NULL)
            serial8250_isa_config(0, &uart->port,&uart->capabilities);

        ret = uart_add_one_port(&serial8250_reg, &uart->port);       /*添加串口*/
        if (ret == 0)
            ret = uart->port.line;
    }
    mutex_unlock(&serial_mutex);

    return ret;             /*返回最大串口序号*/
}
```

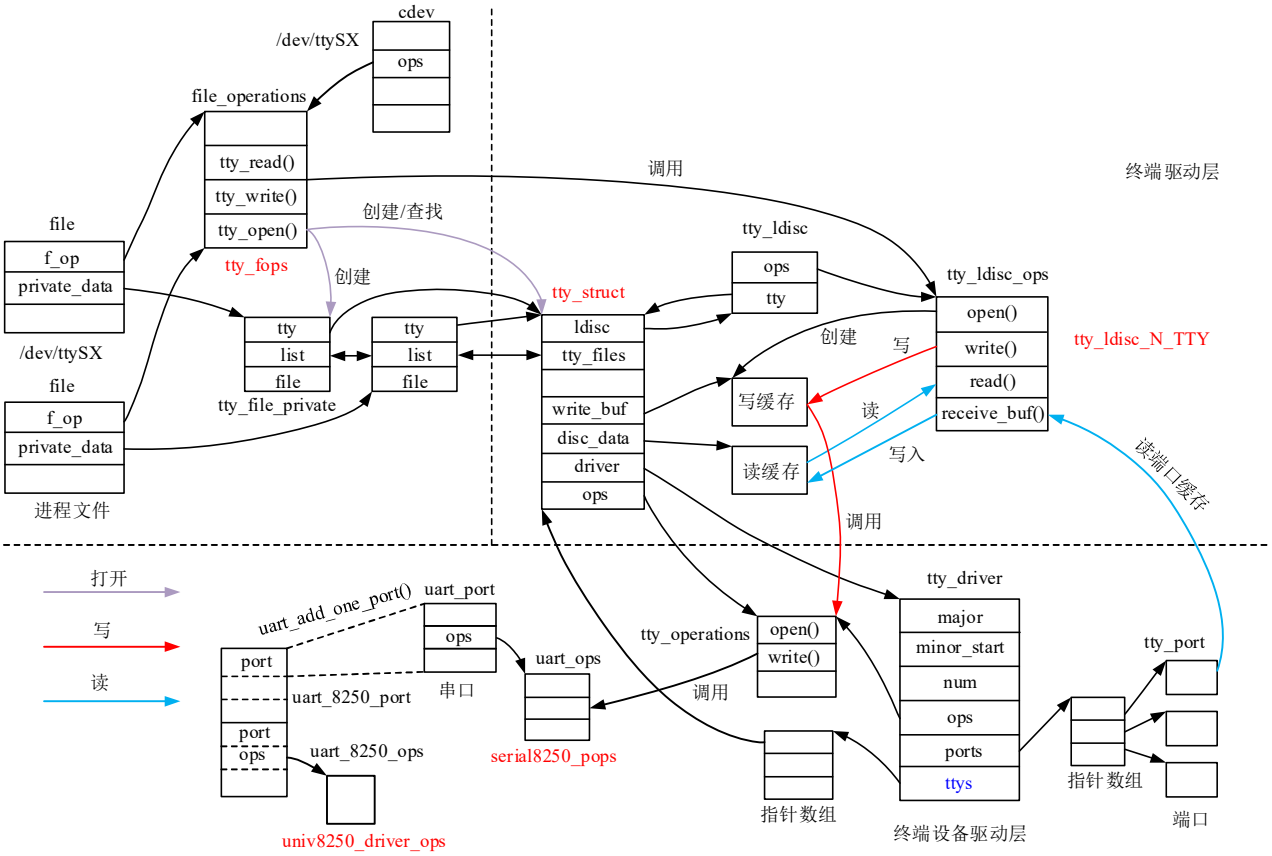
9.8.5 终端设备操作

终端设备的一个通道（终端设备驱动中称为端口），例如一个串口，就是一个终端设备，具有一个设备文件。在注册终端设备驱动 tty_driver 实例，或者添加端口 tty_port 实例时，将定义并添加 cdev 实例，创建并添加表示终端设备的 device 实例（创建设备文件）。用户可通过终端设备文件对设备进行操作。

串口终端设备文件名为/dev/ttySX（X 为串口通道编号），下面以/dev/ttySX 文件操作为例，说明终端设备的操作。

1 概述

终端设备文件操作流程如下图所示：



在创建并添加终端设备对应的 cdev 实例时，实例关联的文件操作结构 file_operations 实例为 **tty_fops**，也就是说进程通过 tty_fops 实例操作终端设备。

打开操作是终端设备文件操作的基础，在打开设备文件 open()系统调用中，将会根据其设备号查找 cdev 实例，将其关联的 file_operations 实例赋予 file 实例，并调用其中的 cdev->ops->open()函数，对于终端设备时就是 **tty_open()**函数。

tty_open()函数将创建或查找在 VFS 中表示打开终端设备的 tty_struct 结构体实例，每个打开的终端设备对应一个实例。由于一个终端设备文件可以被多个进程打开，因此 tty_struct 中包含一个 tty_file_private 结构体实例链表，用于建立 tty_struct 与多个 file 实例之间的关联。

在 tty_open()函数中将根据终端设备号，查找终端驱动 tty_driver 实例，并联到 tty_struct 实例。tty_struct 实例默认关联到 **tty_ldisc_N_TTY** 线路规则。

在 tty_open()函数中还将为 tty_struct 实例创建输入数据的缓存（输出缓存在写函数中创建）。设备文件的读写操作函数调用线路规则中的读写函数，线路规则中的写函数将为 tty_struct 实例分配写缓存，并分批将数据写入缓存并调用 tty_operations 实例中 write()函数将数据写入终端设备。

线路规则中的读函数将从 tty_struct 实例读缓存中复制数据至用户空间。终端设备在收到数据时（中断处理函数中），将数据缓存在端口数据缓存中，在适当的时机将调用线路规则的 receive_buf()或 receive_buf2()函数，将端口缓存中数据写入 tty_struct 实例中的读缓存区中，然后唤醒读等待进程，进程就可以继续读数据了。

在串口终端的 tty_operations 实例的 open()函数中将为端口申请中断，在中断处理函数中将从设备读取的数据通过线路规则写入 tty_struct 实例读缓存。

串口终端的 tty_operations 实例中的函数将调用串口操作 uart_ops 结构体中的函数完成底层操作。

终端设备文件关联的 file_operations 结构体实例 **tty_fops** 定义在/drivers/tty/tty_io.c 文件内：

```
static const struct file_operations tty_fops = {
    .llseek     = no_llseek,
    .read       = tty_read,       /*读操作*/
    .write      = tty_write,      /*写操作*/
    .poll       = tty_poll,
    .unlocked_ioctl= tty_ioctl,     /*设备控制*/
    .compat_ioctl  = tty_compat_ioctl,
    .open       = tty_open,       /*打开操作*/
    .release    = tty_release,
    .fasync     = tty_fasync,
};
```

默认的 tty_ldisc_N_TTY 线路规则定义在/drivers/tty/n_tty.c 文件内，如下所示：

```
struct tty_ldisc_ops tty_ldisc_N_TTY = {
    .magic           = TTY_LDISC_MAGIC,
    .name            = "n_tty",
    .open            = n_tty_open,          /*打开函数，为 tty_struct 分配读缓存区等*/
    .close           = n_tty_close,
    .flush_buffer    = n_tty_flush_buffer,
    .chars_in_buffer = n_tty_chars_in_buffer,
    .read            = n_tty_read,
    .write           = n_tty_write,       /*写（输出）函数*/
    .ioctl           = n_tty_ioctl,
    .set_termios     = n_tty_set_termios,
    .poll            = n_tty_poll,
    .receive_buf     = n_tty_receive_buf,       /*接收端口缓存数据*/
    .write_wakeup    = n_tty_write_wakeup,
    .fasync          = n_tty_fasync,
    .receive_buf2    = n_tty_receive_buf2,          /*接收端口缓存数据*/
};
```

2 打开操作

下面先介绍终端设备文件的打开操作，打开操作由 tty_fops 实例的 open()函数完成。在介绍函数实现前先看一下打开操作中需要创建/查找的数据结构定义。

■数据结构

tty_struct 结构体表示已打开的终端设备，结构体定义如下（/include/linux/tty.h）：

```
struct tty_struct {
    int    magic;        /*魔数 TTY_MAGIC*/
    struct kref kref;          /*引用计数*/
    struct device *dev;
    struct tty_driver *driver;              /*指向终端设备驱动*/
    const struct tty_operations *ops;     /*指向终端设备操作结构*/
    int index;

    struct ld_semaphore ldisc_sem;
    struct tty_ldisc *ldisc;        /*指向 tty_ldisc 结构体，用于关联线路规则*/

    ...          /*各种锁成员*/
    struct ktermios termios, termios_locked;          /*设备控制结构，见 tty_driver 结构体定义*/
    struct termiox *termiox;   /* May be NULL for unsupported */
    char name[64];
    struct pid *pgrp;           /* Protected by ctrl lock */
    struct pid *session;
    unsigned long flags;       /*标记成员*/
    int count;
    struct winsize winsize;          /* winsize_mutex */
    unsigned long stopped:1,       /* flow_lock */
                  flow_stopped:1,
                  unused:BITS_PER_LONG - 2;
    int hw_stopped;
    unsigned long ctrl_status:8,   /* ctrl_lock */
                  packet:1,
                  unused_ctrl:BITS_PER_LONG - 9;
    unsigned int receive_room;    /* Bytes free for queue */
    int flow_change;

    struct tty_struct *link;    /* pair tty for pty/tty pairs*/
    struct fasync_struct *fasync;
    int alt_speed;          /* For magic substitution of 38400 bps */
    wait_queue_head_t write_wait;     /*写操作等待队列*/
    wait_queue_head_t read_wait;     /*读操作等待队列*/
    struct work_struct hangup_work;
    void *disc_data;              /*线路规则数据，指向 n_tty_data 结构体，含读缓存区，/drivers/tty/n_tty.c*/
    void *driver_data;          /*驱动私有数据*/
    struct list_head tty_files;   /*双链表头，管理 tty_file_private 实例*/

#define N_TTY_BUF_SIZE    4096        /*缓存区大小*/
```

```
    int closing;
    unsigned char *write_buf;       /*写缓存*/
    int write_cnt;
    /* If the tty has a pending do_SAK, queue it here - akpm */
    struct work_struct SAK_work;
    struct tty_port *port;     /*对应的端口（从设备），见 tty_driver 结构体定义*/
};
```

tty_struct 结构体主要成员简介如下：

●**driver**：指向终端设备驱动 tty_driver 结构体实例，由设备号查找 tty_driver 实例。

●**ops**：指向终端设备操作 tty_operations 结构体实例，由查找的 tty_driver 实例获取。

●**read_wait、write_wait**：读写操作等待进程队列。

●**disc_data**：指向由线路规则的 open()函数分配的数据，包含接收数据（读操作）缓存区。

●**write_buf**：指向发送数据（写操作）缓存区。

●**port**：指向驱动 tty_driver 中设备对应的端口 tty_port 实例。

●**flags**：标记成员，取值定义如下：

```
#define  TTY_THROTTLED          0    /* Call unthrottle() at threshold min */
#define  TTY_IO_ERROR           1    /* Cause an I/O error (may be no ldisc too) */
#define  TTY_OTHER_CLOSED       2    /* Other side (if any) has closed */
#define  TTY_EXCLUSIVE          3    /*独占式打开设备*/
#define  TTY_DEBUG              4    /* Debugging */
#define  TTY_DO_WRITE_WAKEUP    5     /* Call write_wakeup after queuing new */
#define  TTY_OTHER_DONE         6    /* Closed pty has completed input processing */
#define  TTY_LDISC_OPEN         11   /*线路规则已打开*/
#define  TTY_PTY_LOCK           16   /* pty private */
#define  TTY_NO_WRITE_SPLIT     17   /* Preserve write boundaries to driver */
#define  TTY_HUPPED             18   /* Post driver->hangup() */
#define  TTY_LDISC_HALTED       22   /* Line discipline is halted */
```

●**tty_files**：双链表头，链接打开设备文件关联的 tty_file_private 实例，进程每打开一次终端设备文件就会为其创建一个 tty_file_private 实例。

```
struct tty_file_private {          /*/include/linux/tty.h*/
    struct tty_struct *tty;      /*指向 tty_struct 实例*/
    struct file *file;           /*指向 file 实例*/
    struct list_head list;      /*将实例添加到 tty_struct.tty_files 双链表*/
};
```

●**ldisc**：指向 tty_ldisc 结构体，用于关联线路规则。

```
struct tty_ldisc {     /*/include/linux/tty_ldisc.h*/
    struct tty_ldisc_ops *ops;     /*指向线路规则*/
    struct tty_struct *tty;          /*指向 tty_struct 实例*/
};
```

终端设备驱动 tty_driver 实例中 ttys 成员指向指针数组，数组项关联 tty_struct 实例，tty_struct 实例与端口 tty_port 实例是一一对应的关系，即一个端口对应一个设备。

■打开函数

终端设备文件操作的打开函数 **tty_open()**定义如下（/drivers/tty/tty_io.c）：

```
static int tty_open(struct inode *inode, struct file *filp)
{
    struct tty_struct *tty;
    int noctty, retval;
    struct tty_driver *driver = NULL;
    int index;
    dev_t device = inode->i_rdev;     /*设备号*/
    unsigned saved_flags = filp->f_flags;
    nonseekable_open(inode, filp);     /*设置 filp->f_mode，/fs/open.c*/

retry_open:
    retval = tty_alloc_file(filp);       /*创建 tty_file_private 实例，并赋予 file->private_data 成员*/
    ...
    noctty = filp->f_flags & O_NOCTTY;
    index  = -1;
    retval = 0;

    tty = tty_open_current_tty(device, filp);     /*是否是打开/dev/tty 文件，/drivers/tty/tty_io.c*/
    /*打开/dev/tty 设备文件时，返回 current->signal->tty，并设置非阻塞操作等，其它文件返回 NULL*/
    if (!tty) {     /*如果 tty 为 NULL，即不是打开/dev/tty 设备文件*/
        mutex_lock(&tty_mutex);
        driver = tty_lookup_driver(device, filp, &noctty, &index);   /*index 为设备在驱动中的编号*/
                        /*由设备号查找 tty_driver 实例，包含了对虚拟终端（控制台）的处理*/
        ...
        tty = tty_driver_lookup_tty(driver, inode, index);   /*查找驱动是否已关联 tty_struct 实例*/
        ...                                                  /*index 为指针数组索引值*/

        if (tty) {     /*找到了 tty_struct 实例（设备文件已被其它进程打开）*/
            mutex_unlock(&tty_mutex);
            tty_lock(tty);
            tty_kref_put(tty);
            retval = tty_reopen(tty);   /*增加引用计数等*/
            ...
        } else {     /*设备文件第一次被打开*/
            tty = tty_init_dev(driver, index);
            /*创建 tty_struct 实例，关联 tty_driver 实例，关联默认的 tty_ldisc_N_TTY 线路规则*/
            /*tty.port 指向驱动 tty_driver 中设备对应的端口 tty_port 实例等，见下文*/
            mutex_unlock(&tty_mutex);
```

```
            }

            tty_driver_kref_put(driver);
        }       /*if (!tty)结束*/
        ...
        tty_add_file(tty, filp);     /*建立 tty_file_private 实例与 file、tty_struct 实例的关联*/

        check_tty_count(tty, __func__);
        if (tty->driver->type == TTY_DRIVER_TYPE_PTY &&
                                    tty->driver->subtype == PTY_TYPE_MASTER)
            noctty = 1;
        ...
        if (tty->ops->open)
            retval = tty->ops->open(tty, filp);     /*调用 tty_operations 实例中的 open()函数*/
                                                    /*完成硬件设备的打开等*/
        else
            retval = -ENODEV;
        filp->f_flags = saved_flags;
        ...         /*错误处理*/
        clear_bit(TTY_HUPPED, &tty->flags);

        read_lock(&tasklist_lock);
        spin_lock_irq(&current->sighand->siglock);
        if (!noctty &&current->signal->leader &&!current->signal->tty &&tty->session == NULL) {
            if (filp->f_mode & FMODE_READ)
                __proc_set_tty(tty);       /*设置进程控制终端，current->signal->tty*/
        }
        spin_unlock_irq(&current->sighand->siglock);
        read_unlock(&tasklist_lock);
        tty_unlock(tty);
        return 0;
        ...
    }
```

tty_open()函数的主要工作如下：

（1）创建 tty_file_private 实例，并赋予 file->private_data 成员。

（2）判断是否是打开/dev/tty 设备文件，且返回值 current->signal->tty 不为 NULL，若是则跳至步骤（5），否则继续往下执行。

（3）由设备号查找对应的 tty_driver 实例，但是对/dev/tty0，/dev/console 设备文件需要特别处理，详见下节介绍的 tty_lookup_driver()函数。

（4）查找 tty_driver 实例是否关联了对应的 tty_struct 实例，如果没有则创建并初始化，包括关联线路规则，调用线路规则的 open()函数等，详见下文。

（5）建立 tty_file_private 实例与 file、tty_struct 实例的关联。

（6）调用终端设备驱动关联 tty_operations 实例中的 open()函数等（tty->ops->open(tty, filp)）。

下面简要介绍一下创建并初始化 tty_struct 实例的 **tty_init_dev()**函数的实现。

●创建/初始化 tty_struct

在查找到设备文件对应的 tty_driver 实例后，需要在实例中查找设备对应的 tty_struct 实例。驱动适用的 tty_struct 实例由 tty_driver->ttys 指向的指针数组管理。

如果 tty_struct 实例尚不存在则调用 **tty_init_dev()**函数创建并初始化 tty_struct 实例，函数定义如下：

```
struct tty_struct *tty_init_dev(struct tty_driver *driver, int idx)        /*/drivers/tty/tty_io.c*/
/*driver：设备对应的 tty_driver 实例，idx：设备在驱动中的序号*/
{
    struct tty_struct *tty;
    int retval;

    if (!try_module_get(driver->owner))
        return ERR_PTR(-ENODEV);

    tty = alloc_tty_struct(driver, idx);
            /*创建并初始化 tty_struct 实例，关联 tty_ldisc_N_TTY 线路规则（N_TTY 编号）等*/
    ...

    tty_lock(tty);
    retval = tty_driver_install_tty(driver, tty);
             /*调用 driver->ops->install()函数，建立 tty_driver 与 tty_struct 实例之间的关联。*/
    if (retval < 0)
        goto err_deinit_tty;

    if (!tty->port)
        tty->port = driver->ports[idx];    /*关联 tty_port*/

    WARN_RATELIMIT(!tty->port,
            "%s: %s driver does not set tty->port. This will crash the kernel later. Fix the driver!\n",
            __func__, tty->driver->name);

    tty->port->itty = tty;          /*指向 tty_struct 实例*/
    retval = tty_ldisc_setup(tty, tty->link);   /*调用线路规则的 open()函数等，/drivers/tty/tty_ldisc.c*/
    if (retval)
        goto err_release_tty;
    return tty;
    ...
}
```

tty_init_dev()函数主要工作如下：

（1）调用 alloc_tty_struct()创建并初始化 tty_struct 实例，关联默认的 **tty_ldisc_N_TTY** 线路规则。

（2）调用 driver->ops->install()函数，建立 tty_driver 与 tty_struct 实例之间的关联。

（3）tty_struct 实例关联端口 tty_port 实例。

（4）调用线路规则定义的 open()函数等。

由 tty_open()函数的调用关系可知，线路规则的 open()函数在 tty_operations 实例的 open()函数之前调用。tty_struct 实例默认关联的是 tty_ldisc_N_TTY 线路规则，其 open()函数为 n_tty_open()。

n_tty_open()函数主要是为 tty_struct 实例分配并设置 n_tty_data 实例（tty->disc_data，内含读缓存区）等，源代码请读者自行阅读（/drivers/tty/n_tty.c）。

3 写操作

写操作就是进程通过写终端设备文件，将信息输出到终端设备，以使用户能看到进程信息。终端设备文件写操作函数 tty_write()定义如下（/drivers/tty/tty_io.c）：

```
static ssize_t tty_write(struct file *file, const char __user *buf,size_t count, loff_t *ppos)
{
    struct tty_struct *tty = file_tty(file);
    struct tty_ldisc *ld;
    ssize_t ret;

    if (tty_paranoia_check(tty, file_inode(file), "tty_write"))
        return -EIO;
    if (!tty || !tty->ops->write ||(test_bit(TTY_IO_ERROR, &tty->flags)))
        return -EIO;                          /*tty_operations 实例必须定义 write()函数*/

    if (tty->ops->write_room == NULL)
        ...                                   /*输出信息*/
    ld = tty_ldisc_ref_wait(tty);      /*获取 tty_ldisc 实例，获取锁，/drivers/tty/tty_ldisc.c*/
    if (!ld->ops->write)      /*如果线路规则 tty_ldisc_ops 没有定义 write()函数*/
        ret = -EIO;
    else
        ret = do_tty_write(ld->ops->write, tty, file, buf, count);     /*调用线路规则中的 write()函数*/
    tty_ldisc_deref(ld);          /*释放 tty_ldisc 实例，释放锁*/
    return ret;
}
```

tty_write()函数主要是调用 do_tty_write()函数执行写操作，函数内为 tty_struct 实例分配写缓存区（赋予 tty->write_buf），然后将用户空间数据复制到该缓存区，最后调用线路规则的 **ld->ops->write()**函数执行写操作。

do_tty_write()函数代码简列如下（/drivers/tty/tty_io.c）：

```
static inline ssize_t do_tty_write(
    ssize_t (*write)(struct tty_struct *, struct file *, const unsigned char *, size_t),
    struct tty_struct *tty,struct file *file,const char __user *buf,size_t count)
{
    ssize_t ret, written = 0;
    unsigned int chunk;
```

```
ret = tty_write_lock(tty, file->f_flags & O_NDELAY);        /*获取锁*/
if (ret < 0)
        return ret;

chunk = 2048;        /*缓存块大小，默认 2KB*/
if (test_bit(TTY_NO_WRITE_SPLIT, &tty->flags))
        chunk = 65536;
if (count < chunk)
        chunk = count;

/* write_buf/write_cnt is protected by the atomic_write_lock mutex */
if (tty->write_cnt < chunk) {
        unsigned char *buf_chunk;

        if (chunk < 1024)
                chunk = 1024;

        buf_chunk = kmalloc(chunk, GFP_KERNEL);       /*分配写缓存区*/
        ...
        kfree(tty->write_buf);
        tty->write_cnt = chunk;
        tty->write_buf = buf_chunk;       /*指向写缓存区*/
}

/*执行写操作*/
for (;;) {
        size_t size = count;
        if (size > chunk)
                size = chunk;
        ret = -EFAULT;
        if (copy_from_user(tty->write_buf, buf, size))        /*用户空间复制到缓存*/
                break;
        ret = write(tty, file, tty->write_buf, size);       /*线路规则的 write()函数*/
        if (ret <= 0)
                break;
        written += ret;
        buf += ret;
        count -= ret;
        if (!count)
                break;
        ret = -ERESTARTSYS;
        if (signal_pending(current))
                break;
```

```
            cond_resched();
        }
        if (written) {
            tty_update_time(&file_inode(file)->i_mtime);
            ret = written;
        }
    out:
        tty_write_unlock(tty);
        return ret;
    }
```

do_tty_write()函数简单来说就是为 tty_struct 实例分配一块固定大小的写缓存区，然后将用户数据复制到缓存区，最后调用线路规则定义的 write()函数将数据写入终端设备。如果用户写入的数据较大，则将用户数据按缓存区大小进行划分，分批写出。

默认线路规则 tty_ldisc_N_TTY 实例的 write()函数将调用 tty->ops->write()函数（tty_operations）将缓存区中数据输出到终端设备，见下文。

■线路规则写函数

默认线路规则 tty_ldisc_N_TTY 实例写函数 n_tty_write()代码简列如下（/drivers/tty/n_tty.c）：

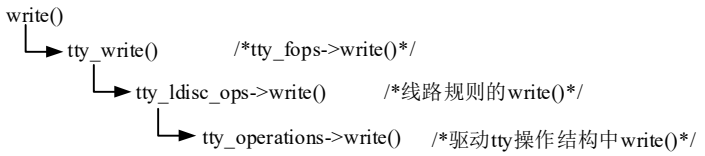
```
static ssize_t n_tty_write(struct tty_struct *tty, struct file *file,const unsigned char *buf, size_t nr)
{
    const unsigned char *b = buf;
    DEFINE_WAIT_FUNC(wait, woken_wake_function);
    int c;
    ssize_t retval = 0;

    /* Job control check -- must be done at start (POSIX.1 7.1.1.4). */
    if (L_TOSTOP(tty) && file->f_op->write != redirected_tty_write) {
        retval = tty_check_change(tty);
        if (retval)
            return retval;
    }

    down_read(&tty->termios_rwsem);

    /* Write out any echoed characters that are still pending */
    process_echoes(tty);        /*写出上次未写出的数据，/drivers/tty/n_tty.c*/

    add_wait_queue(&tty->write_wait, &wait);       /*当前进程添加到 tty_struct 实例的写等待队列*/
    while (1) {
        if (signal_pending(current)) {          /*是否有挂起信号*/
            retval = -ERESTARTSYS;
            break;
        }
```

```
if (tty_hung_up_p(file) || (tty->link && !tty->link->count)) {
        retval = -EIO;
        break;
}
if (O_OPOST(tty)) {       /*检测 tty->termios.c_oflag 标记位，/include/linux/tty.h*/
        while (nr > 0) {
                ssize_t num = process_output_block(tty, b, nr);
                if (num < 0) {
                        if (num == -EAGAIN)
                                break;
                        retval = num;
                        goto break_out;
                }
                b += num;
                nr -= num;
                if (nr == 0)
                        break;
                c = *b;
                if (process_output(c, tty) < 0)
                        break;
                b++; nr--;
        }
        if (tty->ops->flush_chars)
                tty->ops->flush_chars(tty);
} else {
        struct n_tty_data *ldata = tty->disc_data;

        while (nr > 0) {
                mutex_lock(&ldata->output_lock);
                c = tty->ops->write(tty, b, nr);          /*调用 tty_operations 实例中的 write()函数*/
                mutex_unlock(&ldata->output_lock);
                ...
                if (!c)
                        break;
                b += c;
                nr -= c;
        }
}
if (!nr)
        break;
if (file->f_flags & O_NONBLOCK) {
        retval = -EAGAIN;
        break;
```

```
            }
            up_read(&tty->termios_rwsem);

            wait_woken(&wait, TASK_INTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);

            down_read(&tty->termios_rwsem);
        }
    break_out:
        remove_wait_queue(&tty->write_wait, &wait);
        if (b - buf != nr && tty->fasync)
            set_bit(TTY_DO_WRITE_WAKEUP, &tty->flags);
        up_read(&tty->termios_rwsem);
        return (b - buf) ? b - buf : retval;
    }
```

n_tty_write()函数简单地说就是将 tty_struct 实例写缓存中的数据调用 tty_operations 实例中的 write()函数输出到终端设备。

因此，终端设备写操作流程可简要概括如下：



对于串口设备驱动 tty_driver 实例关联的 tty_operations 实例，其 write()函数将调用串口操作结构中的函数通过串口将缓存区中数据发送出去。

4 读操作

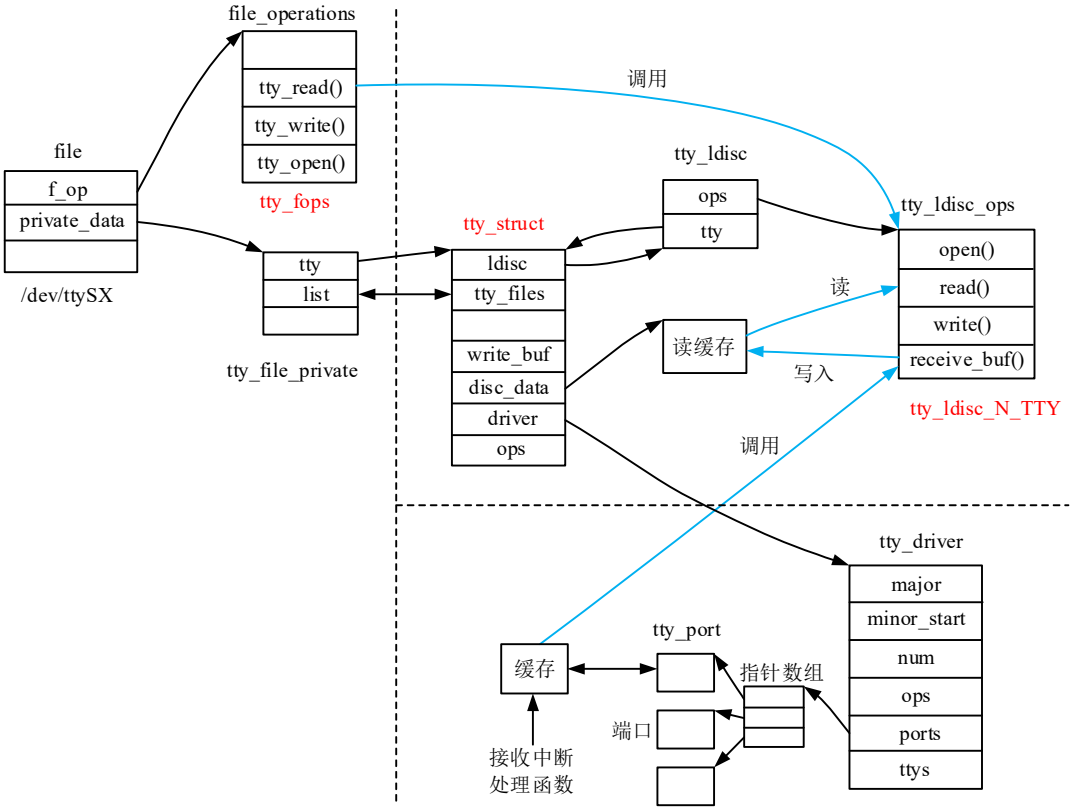
终端设备文件读操作函数 **tty_read()**定义如下（/drivers/tty/tty_io.c）：

```
static ssize_t tty_read(struct file *file, char __user *buf, size_t count,loff_t *ppos)
{
    int i;
    struct inode *inode = file_inode(file);
    struct tty_struct *tty = file_tty(file);    /*关联 tty_struct 实例*/
    struct tty_ldisc *ld;

    if (tty_paranoia_check(tty, inode, "tty_read"))
        return -EIO;
    if (!tty || (test_bit(TTY_IO_ERROR, &tty->flags)))
        return -EIO;

    ld = tty_ldisc_ref_wait(tty);      /*关联线路规则*/
    if (ld->ops->read)
        i = ld->ops->read(tty, file, buf, count);    /*调用线路规则的读操作函数*/
    else
        i = -EIO;
```

```
    tty_ldisc_deref(ld);

    if (i > 0)
        tty_update_time(&inode->i_atime);
    return i;
}
```

终端设备文件读操作函数主要就是调用线路规则定义的读操作函数。默认线路规则 tty_ldisc_N_TTY 实例在 open()函数中将为 tty_struct 实例分配读缓存区（赋予 tty->disc_data），读操作函数内主要工作是将缓存区中数据复制到用户进程空间。如果没有足够的数据，读进程将可能进入睡眠等待。

那么读缓存区中的数据是什么时候写入的呢？串口设备在接收到数据后，一般会产生中断，在中断处理函数中会将接收到的数据写入端口缓存，当端口缓存中数据达到一定数量后，将调用线路规则中定义的 receive_buf()或 receive_buf2()函数，将端口缓存中数据写入 tty_struct 实例中的读缓存区，然后唤醒读等待进程，进程就可以继续读数据了，相关源代码请读者自行阅读。



5 设备控制

终端设备控制命令定义在/include/uapi/asm-generic/ioctls.h 头文件，例如：

```
#define  TCGETS      0x5401
#define  TCSETS      0x5402
#define  TCSETSW     0x5403
#define  TCSETSF     0x5404
#define  TCGETA      0x5405
#define  TCSETA      0x5406
#define  TCSETAW     0x5407
...
```

```
#define   TIOCNOTTY      0x5422     /*设置控制终端，current->signal->tty*/
...
```

终端设备文件控制操作函数 **tty_ioctl()**定义如下（/drivers/tty/tty_io.c）：

```
long tty_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
    struct tty_struct *tty = file_tty(file);
    struct tty_struct *real_tty;
    void __user *p = (void __user *)arg;
    int retval;
    struct tty_ldisc *ld;

    if (tty_paranoia_check(tty, file_inode(file), "tty_ioctl"))
        return -EINVAL;

    real_tty = tty_pair_get_tty(tty);       /*tty_struct 实例*/

    switch (cmd) {       /*预处理命令*/
    case TIOCSETD:
    case TIOCSBRK:
    case TIOCCBRK:
    case TCSBRK:
    case TCSBRKP:
        retval = tty_check_change(tty);
        if (retval)
            return retval;
        if (cmd != TIOCCBRK) {
            tty_wait_until_sent(tty, 0);
            if (signal_pending(current))
                return -EINTR;
        }
        break;
    }

    switch (cmd) {     /*处理其它命令*/
    case TIOCSTI:
        return tiocsti(tty, p);
    case TIOCGWINSZ:
        return tiocgwinsz(real_tty, p);
    ...
    }

    /*处理剩余命令*/
```

```
    if (tty->ops->ioctl) { /*先调用 tty_operations 中命令处理函数，如果没有则调用线路规则中函数*/
        retval = tty->ops->ioctl(tty, cmd, arg);
        if (retval != -ENOIOCTLCMD)
            return retval;
    }
    ld = tty_ldisc_ref_wait(tty);
    retval = -EINVAL;
    if (ld->ops->ioctl) {       /*调用线路规则中 ioctl()函数*/
        retval = ld->ops->ioctl(tty, file, cmd, arg);
        if (retval == -ENOIOCTLCMD)
            retval = -ENOTTY;
    }
    tty_ldisc_deref(ld);
    return retval;
}
```

tty_ioctl()函数主要是充当命令分发器的角色，某些命令由公用层代码处理，其它命令需由终端设备驱动或线路规则的 IO 控制函数处理，具体命令的含义和处理方法，请读者参考内核源代码及系统编程方面的书籍。

9.9 虚拟终端与控制台

本节是前一节的延续。传统的终端设备是串口，对于串口终端设备，终端设备驱动只需要从串口接收和发送数据即可，至于字符如何显示，如何从键盘获取输入，那是硬件终端设备的事，主机不需要管。

现代计算机基本不需要以前的 CRT 终端了，进程与用户之间的交互通过主机系统中的窗口（或者说显示屏）和键盘实现，这称之为虚拟终端。系统中可以运行多个虚拟终端，即可以有多个终端窗口。

虚拟终端设备驱动比串口终端设备驱动要复杂，因为它不仅要接收进程发送的数据，还要在窗口中将数据显示出来，还得通过键盘获取用户输入。

控制台最初是用于内核（系统）输出信息，内核代码中通过 printk()等函数将信息输出到控制台。一般硬件终端设备可同时注册为控制台。控制台的操作直接与设备底层交互，不经过终端驱动。

在 Linux 系统中将当前控制台通过/dev/console 设备文件导出到用户空间，使用户进程也可以通过控制台输入、输出进程信息。

在 Linux 系统中将虚拟终端也称为控制台，为了与内核使用的控制台相区分，在本书中内核使用的控制台称之为系统控制台，虚拟终端就称虚拟终端，不称之为控制台，也就是说本书中控制台就是指系统控制台。

本节介绍虚拟终端设备驱动与系统控制台的实现。虚拟终端设备驱动公共层代码位于/drivers/tty/vt/目录下，具体虚拟终端设备驱动程序位于/drivers/video/console/目录下。

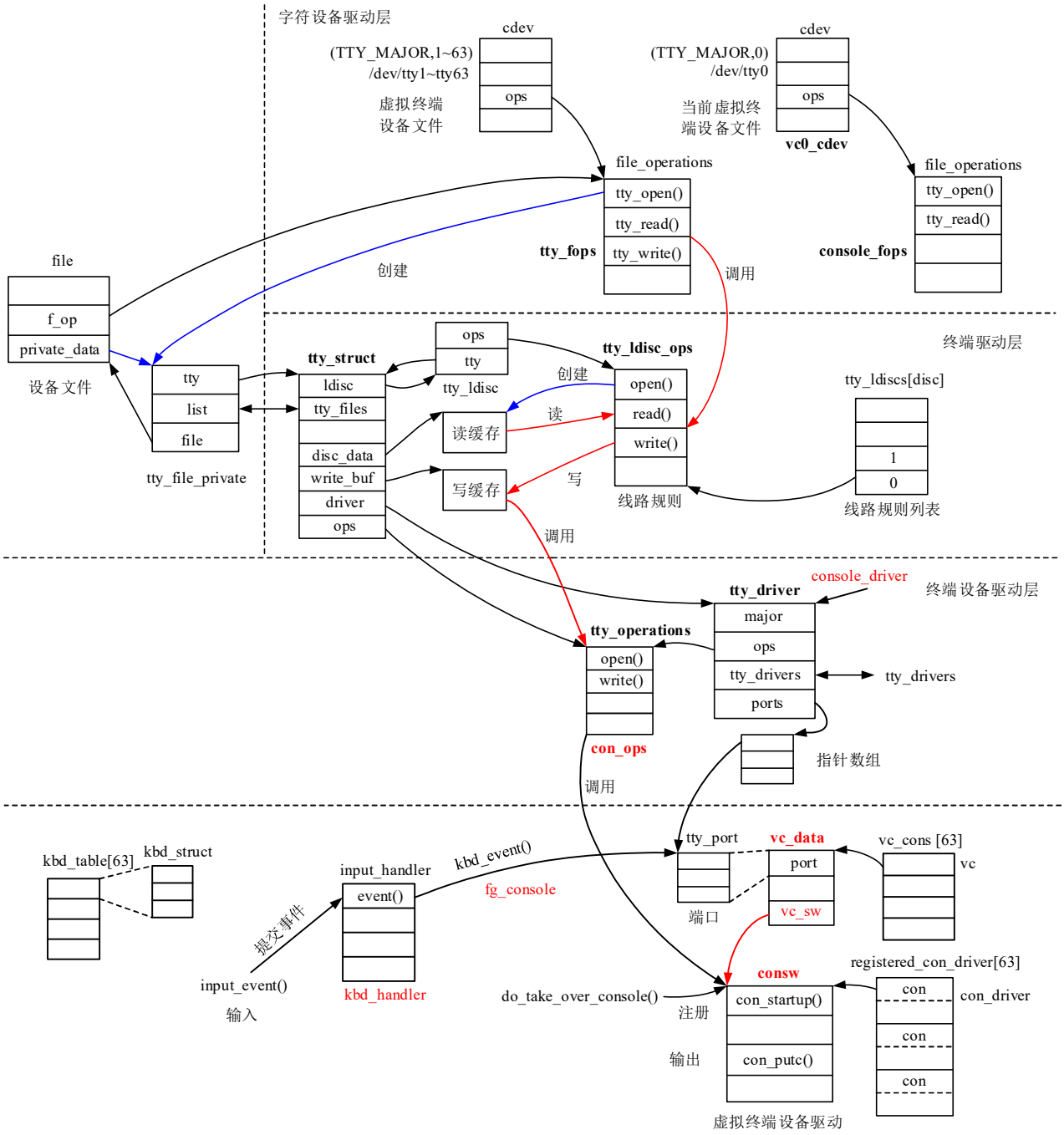
系统控制台公共层代码位于/kernel/printk/printk.c 文件内。

9.9.1 虚拟终端设备驱动

虚拟终端设备可理解成由本机的显示器、键盘按终端协议实现数据传输的设备，虚拟终端可理解成系统中的一个窗口。本小节主要介绍虚拟终端设备驱动的实现。

1 驱动框架

虚拟终端设备驱动框架如下图所示：



在内核初始化阶段，将创建并注册虚拟终端驱动 tty_driver 实例，关联的 tty_operations 实例为 **con_ops**，全局指针变量 console_driver 指向虚拟终端驱动 tty_driver 实例。

在注册 tty_driver 实例时将设置并添加虚拟终端设备字符设备驱动 cdev 实例，TTY_MAJOR 为主设备号为，从设备号为 1~63，设备文件名称为 tty1~63。

在初始化函数中还将设置并添加当前虚拟终端设备对应的 cdev 实例 vc0_cdev，TTY_MAJOR 为主设备号，从设备号为 0，设备文件名为 tty0。

consw 结构体表示一个虚拟终端设备（从设备、端口），就像串口设备中的一个通道，对应一个设备文件，最多可以向内核注册 63 个虚拟终端设备。

内核中虚拟终端最小数量为 1，最大为 63，相关宏定义如下（/include/uapi/linux/vt.h）：

```
#define   MIN_NR_CONSOLES     1          /*最小为 1*/
```

```
#define   MAX_NR_CONSOLES    63      /*最大为 63*/
#define   MAX_NR_USER_CONSOLES    63
```

注册虚拟终端设备需要创建并注册 consw 结构体实例，结构体中包含对终端设备进行操作的函数指针，例如向终端输出字符串并显示。注册的 consw 实例由 con_driver 结构体数组 registered_con_driver[63]通过指针管理。

在打开虚拟终端设备的操作中，将调用 tty_driver 驱动关联 tty_operations 实例的 con_ops->install()函数，在此函数中将为设备创建 vc_data 结构体实例。vc_data 结构体中包含表示终端端口的通用 tty_port 结构体成员，并关联到 consw 实例。vc_data 结构体还描述了虚拟终端设备的属性，如屏幕的行数、列数等。

vc_data 结构体实例由静态数组 vc_cons [63]管理，与 registered_con_driver[63]数组管理的 consw 实例一一对应。

tty_operations 实例 con_ops 中的操作函数将调用 vc_data->vc_sw 指向的 consw 实例中的函数完成具体操作。

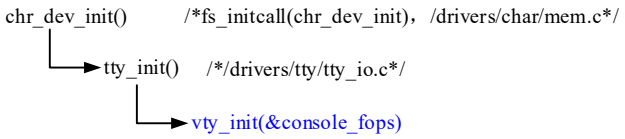
以上介绍的主要是虚拟终端的输出通道，虚拟终端的输入来自系统键盘。内核定义了 kbd_struct 结构体数组 kbd_table[63]，用于标识虚拟终端对应键盘的信息。

在初始化函数中，注册了输入设备事件处理器 input_handler 结构体实例 **kbd_handler**。它将接收键盘的输入，在其事件处理函数 kbd_event()中将根据 fg_console 值（当前虚拟终端序号）检索 vc_cons []数组，获取对应的 vc_data 实例，将输入字符提交给其内嵌的 tty_port 成员，完成输入操作。

虚拟终端设备驱动的主要工作就是创建并注册 consw 结构体实例。下面将简要介绍虚拟终端设备驱动的实现，以及虚拟终端设备文件操作等。

2 初始化

虚拟终端设备驱动初始化函数为 vty_init()，函数调用关系如下：



vty_init(&console_fops)函数定义在/drivers/tty/vt/vt.c 文件内，参数 console_fops 实例与 tty_fops 实例基本相同（/drivers/tty/tty_io.c）。

```
int __init vty_init(const struct file_operations *console_fops)
{
    cdev_init(&vc0_cdev, console_fops);      /*初始化 cdev 实例*/
    if (cdev_add(&vc0_cdev, MKDEV(TTY_MAJOR, 0), 1) ||          /*主 TTY_MAJOR，从 0*/
            register_chrdev_region(MKDEV(TTY_MAJOR, 0), 1, "/dev/vc/0") < 0)
                panic("Couldn't register /dev/tty0 driver\n");

    tty0dev = device_create_with_groups(tty_class, NULL,MKDEV(TTY_MAJOR, 0), NULL,
                                        vt_dev_groups, "tty0");      /*设备文件/dev/tty0*/
    ...     /*错误处理*/

    vcs_init();   /*屏幕设备初始化，/drivers/tty/vt/vc_screen.c*/
```

```
    console_driver = alloc_tty_driver(MAX_NR_CONSOLES);     /*分配 tty_driver 实例*/
        ...      /*错误处理*/                                    /*MAX_NR_CONSOLES 为 63*/

    console_driver->name = "tty";       /*设备文件名*/
    console_driver->name_base = 1;
    console_driver->major = TTY_MAJOR;        /*主设备号，同串口终端设备*/
    console_driver->minor_start = 1;     /*从设备号从 1 开始，数量为 63*/
    console_driver->type = TTY_DRIVER_TYPE_CONSOLE;
    console_driver->init_termios = tty_std_termios;
    if (default_utf8)
        console_driver->init_termios.c_iflag |= IUTF8;
    console_driver->flags = TTY_DRIVER_REAL_RAW | TTY_DRIVER_RESET_TERMIOS;
    tty_set_operations(console_driver, &con_ops);       /*设置 tty_operations 实例*/
    if (tty_register_driver(console_driver))                /*注册 tty_driver 实例*/
        panic("Couldn't register console driver\n");
    kbd_init();          /*键盘设备初始化，/drivers/tty/vt/keyboard.c*/
    console_map_init();      /*设置已有 vc_data 实例的字体 Unicode 映射，/drivers/tty/vt/consolemap.c*/
  #ifdef CONFIG_MDA_CONSOLE
    mda_console_init();
  #endif
    return 0;
}
```

vty_init()初始化函数主要完成以下工作：

（1）初始化并添加 cdev 实例 **vc0_cdev**，关联的 file_operations 实例为 console_fops，其适用的设备号为 MKDEV(TTY_MAJOR, 0)，设备文件名称为/dev/tty0，表示当前虚拟终端。

（2）初始化虚拟终端中屏幕设备驱动，见下文。

（3）为/dev/tty1~63 虚拟终端设备（共 63 个从设备）分配并注册 tty_driver 实例，其关联的 tty_operations 实例为 **con_ops**。

（4）键盘设备初始化，见下文。

（5）设置已有 vc_data 实例的字体 Unicode 映射

■屏幕设备初始化

虚拟终端的屏幕可视为一个设备，内核为其注册了驱动并创建立了设备文件。虚拟终端屏幕设备的主设备号为 VCS_MAJOR（7）。

初始化函数 **vcs_init()**定义如下（/drivers/tty/vt/vc_screen.c）：

```
int __init vcs_init(void)
{
    unsigned int i;

    if (register_chrdev(VCS_MAJOR, "vcs", &vcs_fops))          /*创建并添加 cdev 实例*/
        panic("unable to get major %d for vcs device", VCS_MAJOR);
    vc_class = class_create(THIS_MODULE, "vc");          /*创建设备类*/
```

```
    device_create(vc_class, NULL, MKDEV(VCS_MAJOR, 0), NULL, "vcs");       /*设备文件/dev/vcs*/
    device_create(vc_class, NULL, MKDEV(VCS_MAJOR, 128), NULL, "vcsa"); /*设备文件/dev/vcsa*/
    for (i = 0; i < MIN_NR_CONSOLES; i++)
        vcs_make_sysfs(i);              /*创建/dev/vcsi，/dev/vcsai，/drivers/tty/vt/vc_screen.c*/
    return 0;
}
```

vcs_init()函数创建/添加了主设备号为 VCS_MAJOR 的 cdev 实例，关联 file_operations 实例为 vcs_fops，其操作函数主要是对屏幕进行读写。

vcs_init()函数中创建的设备文件及其从设备号对应关系如下：

/dev/vcs：0。

/dev/vcs1~63：1~63

/dev/vcsa：128

/dev/vcsa1~63：129~191

■键盘设备初始化

虚拟终端中的键盘信息由 kbd_struct 结构体表示，内核在/drivers/tty/vt/keyboard.c 文件内定义了结构体数组 kbd_table[MAX_NR_CONSOLES]，每个数组项对应一个虚拟终端设备。

kbd_struct 结构体定义如下（/include/linux/kbd_kern.h）：

```
struct kbd_struct {
    unsigned char lockstate;
    ...         /*宏定义，下同*/
    unsigned char slockstate;       /* for `sticky' Shift, Ctrl, etc. */
    unsigned char ledmode:1;
    ...
    unsigned char ledflagstate:4;  /* flags, not lights */
    unsigned char default_ledflagstate:4;
    ...
    unsigned char kbdmode:3;     /* one 3-bit value */
    ...
    unsigned char modeflags:5;
    ...
};
```

在/drivers/tty/vt/keyboard.c 文件定义了 kbd_struct 结构体数组，如下所示：

```
static struct kbd_struct kbd_table[MAX_NR_CONSOLES];
static struct kbd_struct *kbd = kbd_table;
```

虚拟终端键盘信息初始化函数 kbd_init()定义如下（/drivers/tty/vt/keyboard.c）：

```
int __init kbd_init(void)
{
    int i;
    int error;
```

```
    for (i = 0; i < MAX_NR_CONSOLES; i++) {     /*初始化 kbd_table[]数组*/
        kbd_table[i].ledflagstate = kbd_defleds();
        kbd_table[i].default_ledflagstate = kbd_defleds();
        kbd_table[i].ledmode = LED_SHOW_FLAGS;
        kbd_table[i].lockstate = KBD_DEFLOCK;
        kbd_table[i].slockstate = 0;
        kbd_table[i].modeflags = KBD_DEFMODE;
        kbd_table[i].kbdmode = default_utf8 ? VC_UNICODE : VC_XLATE;
    }

    kbd_init_leds();

    error = input_register_handler(&kbd_handler);       /*注册输入事件处理器*/
    if (error)
        return error;

    tasklet_enable(&keyboard_tasklet);       /*小任务执行函数为 kbd_bh()，用于更新 LED 状态*/
    tasklet_schedule(&keyboard_tasklet);

    return 0;
}
```

kbd_init()函数内初始化了 kbd_table[63]数组项，并注册了输入设备事件处理器 kbd_handler 实例。

kbd_handler 实例定义在/drivers/tty/vt/keyboard.c 文件内：

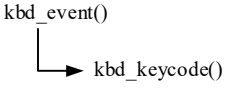
```
static struct input_handler kbd_handler = {
    .event          = kbd_event,        /*处理键盘输入事件，/drivers/tty/vt/keyboard.c*/
    .match          = kbd_match,
    .connect  = kbd_connect,        /*建立输入设备 input_dev 与 kbd_handler 实例关联*/
    .disconnect     = kbd_disconnect,
    .start      = kbd_start,
    .name           = "kbd",
    .id_table = kbd_ids,        /*匹配键盘设备*/
};
```

在键盘设备驱动程序中，将注册表示键盘的输入设备 input_dev 实例，它将匹配 kbd_handler 实例，并调用其中的 kbd_connect()函数。kbd_connect()函数建立 input_dev 实例与 kbd_handler 实例之间的关联。

当键盘有输入，报告输入事件时，将调用 kbd_handler 实例的 event()函数，即 kbd_event()函数处理。

内核在/drivers/tty/vt/vt.c 文件内定义了 **fg_console** 全局变量（初始为 0，0~62），标识当前虚拟终端，即输入由此终端接收。fg_console 用来检索 vc_cons [63]和 registered_con_driver[63]数组项。

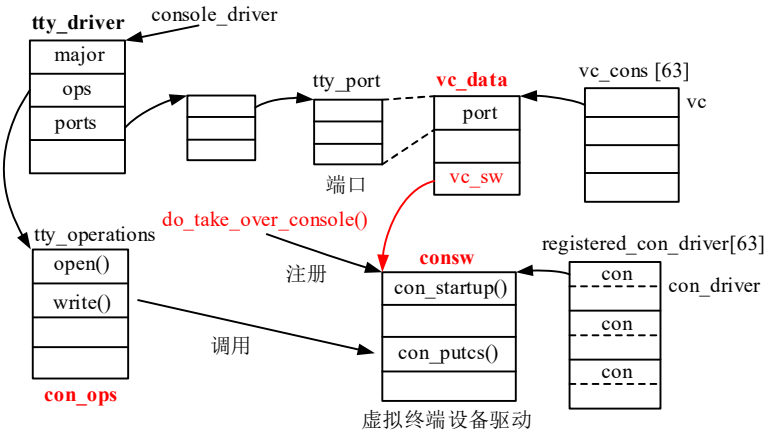
键盘报告事件时，kbd_event()函数处理将调用 kbd_keycode()函数（/drivers/tty/vt/keyboard.c），处理输入字符。



kbd_keycode()函数根据 fg_console 值检索 vc_cons []数组，获取对应的 vc_data 实例，将输入字符提交给其内嵌的 tty_port 成员，源代码请读者自行阅读。

3 驱动实现

虚拟终端设备驱动框架如下图所示：



consw 结构体可视为具体虚拟终端设备驱动，它主要包含一些屏幕操作函数，如将输出数据在屏幕上显示等，它相当于串口终端设备驱动中的 uart_port 结构体。在内核由全局数组 registered_con_driver[63]通过指针管理注册的 consw 实例。若要注册虚拟终端则需定义并注册 consw 实例。

每个 consw 实例对应一个 vc_data 结构体实例，表示窗口（物理）信息，也可以视为一个终端通道的信息，内嵌表示 tty 端口的 tty_prot 结构体成员，它相当于串口终端驱动中的 uart_state 结构体。vc_data 实例由 vc_cons [63]全局数组通过指针管理。vc_data 实例在打开虚拟终端设备文件时动态创建。

registered_con_driver[63]和 vc_cons [63]数组项是一一对应的关系，每个数组项代表一个虚拟终端，数组项索引值是虚拟终端的序号。在注册 consw 实例时，将依次查找 registered_con_driver[63]数组项，关联到第一个空闲的数组项。

虚拟终端设备驱动关联 tty_operations 实例 con_ops 中的函数调用 consw 实例中的函数完成终端的显示和控制等操作。

下面先介绍 consw 和 vc_data 结构体定义，然后介绍注册 consw 实例的接口函数以及相关的初始化函数。

■数据结构

虚拟终端驱动主要的数据结构包括 consw 和 vc_data 结构体，下面分别进行介绍。

●consw

虚拟终端设备驱动 consw 结构体定义在/include/linux/console.h 头文件：

```
struct consw {
    struct module *owner;
    const char *(*con_startup)(void);       /*启动函数，在注册是调用*/
    void (*con_init)(struct vc_data *, int);       /*初始化函数，在打开虚拟终端时调用*/
    void (*con_deinit)(struct vc_data *);
    void (*con_clear)(struct vc_data *, int, int, int, int);     /*清屏*/
    void (*con_putc)(struct vc_data *, int, int, int);           /*输出一个字符*/
```

```
    void (*con_putcs)(struct vc_data *, const unsigned short *, int, int, int);     /*输出字符串*/
    void (*con_cursor)(struct vc_data *, int);
    int   (*con_scroll)(struct vc_data *, int, int, int, int);
    void (*con_bmove)(struct vc_data *, int, int, int, int, int, int);
    int   (*con_switch)(struct vc_data *);
    int   (*con_blank)(struct vc_data *, int, int);
    int   (*con_font_set)(struct vc_data *, struct console_font *, unsigned);
    int   (*con_font_get)(struct vc_data *, struct console_font *);
    int   (*con_font_default)(struct vc_data *, struct console_font *, char *);
    int   (*con_font_copy)(struct vc_data *, int);
    int   (*con_resize)(struct vc_data *, unsigned int, unsigned int,unsigned int);
    int   (*con_set_palette)(struct vc_data *, unsigned char *);
    int   (*con_scrolldelta)(struct vc_data *, int);
    int   (*con_set_origin)(struct vc_data *);
    void (*con_save_screen)(struct vc_data *);
    u8   (*con_build_attr)(struct vc_data *, u8, u8, u8, u8, u8, u8);
    void (*con_invert_region)(struct vc_data *, u16 *, int);
    u16       *(*con_screen_pos)(struct vc_data *, int);
    unsigned long (*con_getxy)(struct vc_data *, unsigned long, int *, int *);

    int   (*con_debug_enter)(struct vc_data *);
    int   (*con_debug_leave)(struct vc_data *);
};
```

内核在/drivers/tty/vt/vt.c 文件内定义了 con_driver 结构体及实例数组，用于管理 consw 实例。con_driver 结构体定义如下：

```
struct con_driver {
    const struct consw *con;              /*指向 consw 实例*/
    const char *desc;
    struct device *dev;         /*指向 device 实例，对应设备文件名为/dev/vtconX*/
    int node;
    int first;
    int last;
    int flag;
};
```

flag 标记成员取值定义如下：

```
#define   CON_DRIVER_FLAG_MODULE   1
#define   CON_DRIVER_FLAG_INIT      2
#define   CON_DRIVER_FLAG_ATTR      4
#define   CON_DRIVER_FLAG_ZOMBIE    8
```

内核定义了全局 con_driver 结构体数组 **registered_con_driver**[MAX_NR_CON_DRIVER]用于管理注册的 consw 实例。数组项索引值就是虚拟终端设备的序号。

内核在初始化阶段会对数组进行初始化，初始化函数 con_init()定义如下（/drivers/tty/vt/vt.c）：

```
static int __init con_init(void)
{
    const char *display_desc = NULL;
    struct vc_data *vc;
    unsigned int currcons = 0, i;

    console_lock();

    if (conswitchp)          /*全局变量，初始值为 NULL，由体系结构相关代码赋值，如 setup_arch()*/
        display_desc = conswitchp->con_startup();
    if (!display_desc) {          /*display_desc 为 NULL，返回 0*/
        fg_console = 0;          /*第一个虚拟终端设为当前虚拟终端*/
        console_unlock();
        return 0;
    }
    /*以下是 display_desc 非 NULL 的情形*/
    ...              /*用 conswitchp 指向实例设置 registered_con_driver[]数组*/

#ifdef CONFIG_VT_CONSOLE
    register_console(&vt_console_driver);       /*注册系统控制台*/
#endif
    return 0;
}
console_initcall(con_init);   /*在 console_init()函数中调用执行*/
```

conswitchp 是指向 consw 实例的全局指针变量，初始值为 NULL。体系结构相关代码可以对 conswitchp 变量赋值（如在 setup_arch()函数中），这表示系统默认使用的虚拟终端，在 con_init()函数中会用此实例设置 registered_con_driver[]数组项。

如果需要将虚拟终端作为系统控制台，则需将对应 consw 实例赋予全局变量 conswitchp，系统控制台见下一小节。

●vc_data

vc_data 结构体表示虚拟终端设备的屏幕信息等，结构体定义如下（/include/linux/console_struct.h）：

```
struct vc_data {
    struct tty_port port;          /*通用 tty 端口，关联到 tty_driver 实例*/

    unsigned short  vc_num;          /*虚拟终端编号，0~62，vc_cons []数组索引值*/
    unsigned int    vc_cols;         /*显示屏列数*/
    unsigned int    vc_rows;         /*显示屏行数*/
    unsigned int    vc_size_row;     /*行字节数*/
    unsigned int    vc_scan_lines;   /* # of scan lines */
    unsigned long vc_origin;         /* [!] Start of real screen */
    unsigned long vc_scr_end;        /* [!] End of real screen */
```

```
unsigned long vc_visible_origin;   /* [!] Top of visible window */
unsigned int    vc_top, vc_bottom; /* Scrolling region */
const struct consw *vc_sw;              /*指向 consw 实例*/
unsigned short *vc_screenbuf;           /*屏幕缓存*/
unsigned int    vc_screenbuf_size;
unsigned char vc_mode;       /* KD_TEXT, ... */
/*屏幕中所有字符的属性*/
unsigned char vc_attr;              /*Current attributes */
unsigned char vc_def_color;         /* Default colors */
unsigned char vc_color;             /* Foreground & background */
unsigned char vc_s_color;           /* Saved foreground & background */
unsigned char vc_ulcolor;           /* Color for underline mode */
unsigned char     vc_itcolor;
unsigned char vc_halfcolor;         /* Color for half intensity mode */
/*光标*/
unsigned int    vc_cursor_type;
unsigned short  vc_complement_mask; /* [#] Xor mask for mouse pointer */
unsigned short  vc_s_complement_mask;     /* Saved mouse pointer mask */
unsigned int    vc_x, vc_y;         /* Cursor position */
unsigned int    vc_saved_x, vc_saved_y;
unsigned long vc_pos;               /*当前光标位置*/
/*字体*/
unsigned short vc_hi_font_mask;   /* [#] Attribute set for upper 256 chars of font or 0 if not supported */
struct console_font vc_font;        /* Current VC font set */
unsigned short vc_video_erase_char;     /* Background erase character */
/*虚拟终端数据*/
unsigned int    vc_state;        /* Escape sequence parser state */
unsigned int    vc_npar,vc_par[NPAR];       /* Parameters of current escape sequence */
/* data for manual vt switching */
struct vt_mode  vt_mode;     /*/include/uapi/linux/vt.h*/
struct pid        *vt_pid;
int           vt_newvt;
wait_queue_head_t paste_wait;
/*模式标记*/
unsigned int    vc_charset      : 1;   /* Character set G0 / G1 */
unsigned int    vc_s_charset    : 1;   /* Saved character set */
unsigned int    vc_disp_ctrl    : 1;   /* Display chars < 32? */
unsigned int    vc_toggle_meta      : 1;   /* Toggle high bit? */
unsigned int    vc_decscnm      : 1;   /* Screen Mode */
unsigned int    vc_decom        : 1;   /* Origin Mode */
unsigned int    vc_decawm       : 1;   /* Autowrap Mode */
unsigned int    vc_deccm        : 1;   /* Cursor Visible */
unsigned int    vc_decim: 1;        /* Insert Mode */
```

```
    unsigned int    vc_deccolm    : 1;  /* 80/132 Column Mode */
    /* attribute flags */
    unsigned int    vc_intensity    : 2;  /* 0=half-bright, 1=normal, 2=bold */
    unsigned int      vc_italic:1;
    unsigned int    vc_underline   : 1;
    unsigned int    vc_blink  : 1;
    unsigned int    vc_reverse      : 1;
    unsigned int    vc_s_intensity : 2;  /* saved rendition */
    unsigned int      vc_s_italic:1;
    unsigned int    vc_s_underline      : 1;
    unsigned int    vc_s_blink      : 1;
    unsigned int    vc_s_reverse   : 1;
    /* misc */
    unsigned int    vc_ques         : 1;
    unsigned int    vc_need_wrap : 1;
    unsigned int    vc_can_do_color    : 1;
    unsigned int    vc_report_mouse : 2;
    unsigned char vc_utf            : 1;  /* Unicode UTF-8 encoding */
    unsigned char vc_utf_count;
    int    vc_utf_char;
    unsigned int    vc_tab_stop[8];             /* Tab stops. 256 columns. */
    unsigned char     vc_palette[16*3];           /* Colour palette for VGA+ */
    unsigned short * vc_translate;
    unsigned char       vc_G0_charset;
    unsigned char       vc_G1_charset;
    unsigned char       vc_saved_G0;
    unsigned char       vc_saved_G1;
    unsigned int      vc_resize_user;               /* resize request from user */
    unsigned int    vc_bell_pitch;       /* Console bell pitch */
    unsigned int    vc_bell_duration;    /* Console bell duration */
    unsigned short vc_cur_blink_ms;   /* Cursor blink duration */
    struct vc_data **vc_display_fg;           /* [!] Ptr to var holding fg console for this display */
    struct uni_pagedir *vc_uni_pagedir;
    struct uni_pagedir **vc_uni_pagedir_loc; /* [!] Location of uni_pagedir variable for this console */
    bool vc_panic_force_write; /* when oops/panic this VC can accept forced output/blanking */
    /* additional information is in vt_kern.h */
};
```

内核通过 vc 结构体数组 **vc_cons** [MAX_NR_CONSOLES]管理 vc_data 结构体实例，vc 结构体定义如下（/include/linux/console_struct.h）：

```
struct vc {
    struct vc_data *d;      /*指向 vc_data 结构体*/
    struct work_struct SAK_work;
```

```
};
```

vc_cons []数组项与管理 consw 实例的 registered_con_driver[]数组项一一对应，MAX_NR_CONSOLES 数组项数为 63，定义在/include/uapi/linux/vt.h 头文件内。

vc_allocate(unsigned int currcons)函数用于为 vc_cons []数组中 currcons 数组项创建（关联）并初始化 vc_data 实例，这在打开虚拟终端设备文件时进行，下文将介绍此函数的实现。

■注册 consw 实例

虚拟终端设备驱动程序的主要工作就是实现 consw 实例，并向内核注册。

注册 consw 实例函数 **do_take_over_console()**定义如下（/drivers/tty/vt/vt.c）：

```
int do_take_over_console(const struct consw *csw, int first, int last, int deflt)
{
    int err;

    err = do_register_con_driver(csw, first, last);     /*注册实例，/drivers/tty/vt/vt.c*/
    if (err == -EBUSY)
        err = 0;
    if (!err)
        do_bind_con_driver(csw, first, last, deflt);   /*完成注册后的工作，/drivers/tty/vt/vt.c*/

    return err;
}
```

注册工作主要由 do_register_con_driver()函数完成，代码如下：

```
static int do_register_con_driver(const struct consw *csw, int first, int last)
{
    struct module *owner = csw->owner;
    struct con_driver *con_driver;
    const char *desc;
    int i, retval = 0;

    WARN_CONSOLE_UNLOCKED();

    if (!try_module_get(owner))
        return -ENODEV;

    for (i = 0; i < MAX_NR_CON_DRIVER; i++) {     /*检查虚拟终端驱动是否已经注册*/
        con_driver = &registered_con_driver[i];
        if (con_driver->con == csw)
            retval = -EBUSY;
    }

    ...
```

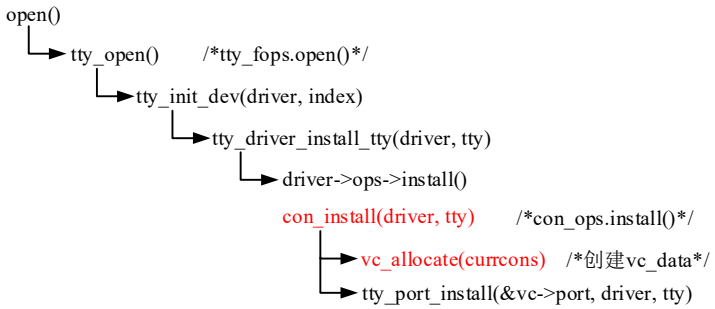
```
        desc = csw->con_startup();   /*调用启动函数*/
        ...
        retval = -EINVAL;

        for (i = 0; i < MAX_NR_CON_DRIVER; i++) {   /*查找第一个空闲 registered_con_driver[]数组项*/
            con_driver = &registered_con_driver[i];

            if (con_driver->con == NULL &&!(con_driver->flag & CON_DRIVER_FLAG_ZOMBIE)) {
                con_driver->con = csw;              /*指向 consw 实例*/
                con_driver->desc = desc;
                con_driver->node = i;
                con_driver->flag = CON_DRIVER_FLAG_MODULE |CON_DRIVER_FLAG_INIT;
                con_driver->first = first;
                con_driver->last = last;
                retval = 0;
                break;
            }
        }
        ...
        con_driver->dev =device_create_with_groups(vtconsole_class, NULL,
                                MKDEV(0, con_driver->node),
                                con_driver, con_dev_groups,
                                "vtcon%i", con_driver->node);      /*创建添加 device 实例，创建设备文件*/
        if (IS_ERR(con_driver->dev)) {
            ...
        } else {
            vtconsole_init_device(con_driver);   /*设置 CON_DRIVER_FLAG_ATTR 标记位*/
        }

    err:
        module_put(owner);
        return retval;
    }
```

do_register_con_driver()函数比较好理解，首先检查 consw 实例是否已经注册，如果没有则调用实例中 con_startup()函数（激活虚拟终端），然后查找 registered_con_driver[]数组中空闲的项，关联到时 consw 实例，最后创建并添加对应的 device 实例。

注销 consw 实例的函数为 do_unregister_con_driver(const struct consw *csw)，源代码请读者自行阅读。

■创建 vc_data 实例

前面介绍了注册 consw 实例的函数，但并没有创建对应的 vc_data 实例，此实例在何时创建呢？在注册虚拟终端驱动 tty_driver 实例时，将为虚拟终端设备创建设备文件。在进程第一次打开虚拟终端设备文件时，将调用 tty_operations 实例的 install()函数，调用关系如下图所示：



对于虚拟终端设备驱动 tty_driver 实例，其关联的 tty_operations 实例为 **con_ops**，实例中 install()函数为 **con_install()**。con_install()函数内将调用 vc_allocate(currcons)函数（currcons 为虚拟终端序号）创建 vc_data 实例，调用 tty_port_install()函数建立 vc_data 实例中内嵌 tty_port 实例与 tty_driver 与 tty_struct 实例之间的关联。

下面主要看一下 **vc_allocate**(unsigned int currcons)函数的实现，如下所示（/drivers/tty/vt/vt.c）：

```
int vc_allocate(unsigned int currcons)
/*currcons：vc_cons []数组项索引值，成功返回 0*/
{
    WARN_CONSOLE_UNLOCKED();

    if (currcons >= MAX_NR_CONSOLES)
        return -ENXIO;
    if (!vc_cons[currcons].d) {    /*如果数组项尚未关联 vc_data 实例（若已关联直接返回 0）*/
        struct vc_data *vc;
        struct vt_notifier_param param;

        if (currcons >= MAX_NR_USER_CONSOLES && !capable(CAP_SYS_RESOURCE))
            return -EPERM;

        param.vc = vc = kzalloc(sizeof(struct vc_data), GFP_KERNEL);    /*分配 vc_data 实例*/
        if (!vc)
            return -ENOMEM;
        vc_cons[currcons].d = vc;      /*关联全局数组项*/
        tty_port_init(&vc->port);      /*初始化 tty_port 成员*/
        INIT_WORK(&vc_cons[currcons].SAK_work, vc_SAK);
        visual_init(vc, currcons, 1);    /*初始化部分成员，调用 vc->vc_sw->con_init(vc, init)函数等*/
        if (!*vc->vc_uni_pagedir_loc)
            con_set_default_unimap(vc);
        vc->vc_screenbuf = kmalloc(vc->vc_screenbuf_size, GFP_KERNEL);    /*分配屏幕缓存*/
        ...       /*错误处理*/

        if (global_cursor_default == -1)
            global_cursor_default = 1;

        vc_init(vc, vc->vc_rows, vc->vc_cols, 1);    /*继续初始化 vc_data 成员，行数、列数等*/
```

```
            vcs_make_sysfs(currcons);
                                /*创建添加 device 实例（创建设备文件），/drivers/tty/vt/vc_screen.c*/
            atomic_notifier_call_chain(&vt_notifier_list, VT_ALLOCATE, &param);
        }
        return 0;
    }
```

4 虚拟终端设备驱动示例

帧缓存设备通常作为虚拟终端的显示屏，选择 FRAMEBUFFER_CONSOLE 配置选项，帧缓存设备将注册为虚拟终端设备。帧缓存设备虚拟终端驱动程序代码主要位于/drivers/video/console/fbcon.c 文件内，在此文件内定义了 consw 结构体实例 fb_con（实例中函数源代码请有兴趣的读者自行阅读）：

```
static const struct consw fb_con = {
    .owner              = THIS_MODULE,
    .con_startup        = fbcon_startup,
    .con_init           = fbcon_init,          /*显示 logo*/
    .con_deinit         = fbcon_deinit,
    .con_clear          = fbcon_clear,
    .con_putc           = fbcon_putc,          /*输出字符*/
    .con_putcs          = fbcon_putcs,
    .con_cursor         = fbcon_cursor,
    .con_scroll         = fbcon_scroll,
    .con_bmove          = fbcon_bmove,
    .con_switch         = fbcon_switch,
    .con_blank          = fbcon_blank,
    .con_font_set       = fbcon_set_font,
    .con_font_get           = fbcon_get_font,
    .con_font_default   = fbcon_set_def_font,
    .con_font_copy          = fbcon_copy_font,
    .con_set_palette    = fbcon_set_palette,
    .con_scrolldelta    = fbcon_scrolldelta,
    .con_set_origin     = fbcon_set_origin,
    .con_invert_region      = fbcon_invert_region,
    .con_screen_pos     = fbcon_screen_pos,
    .con_getxy          = fbcon_getxy,
    .con_resize         = fbcon_resize,
    .con_debug_enter    = fbcon_debug_enter,
    .con_debug_leave    = fbcon_debug_leave,
};
```

初始化函数 **fb_console_init()**中将注册 fb_con 实例，函数定义如下（/drivers/video/console/fbcon.c）：

```
static int __init fb_console_init(void)
{
    int i;
```

```
    console_lock();
    fb_register_client(&fbcon_event_notifier);
                                /*向 fb_notifier_list 通知链注册通知，/drivers/video/fbdev/core/fb_notify.c*/
    fbcon_device = device_create(fb_class, NULL, MKDEV(0, 0), NULL,"fbcon");   /*创建 device 实例*/

    if (IS_ERR(fbcon_device)) {
        ...
    } else
        fbcon_init_device();     /*为 fbcon_device 指向实例在 sysfs 文件系统中创建属性文件*/

    for (i = 0; i < MAX_NR_CONSOLES; i++)
        con2fb_map[i] = -1;

    console_unlock();
    fbcon_start();     /*注册  consw 结构体 fb_con 实例等，/drivers/video/console/fbcon.c*/
    return 0;
}
fs_initcall(fb_console_init);          /*内核启动后期调用*/
```

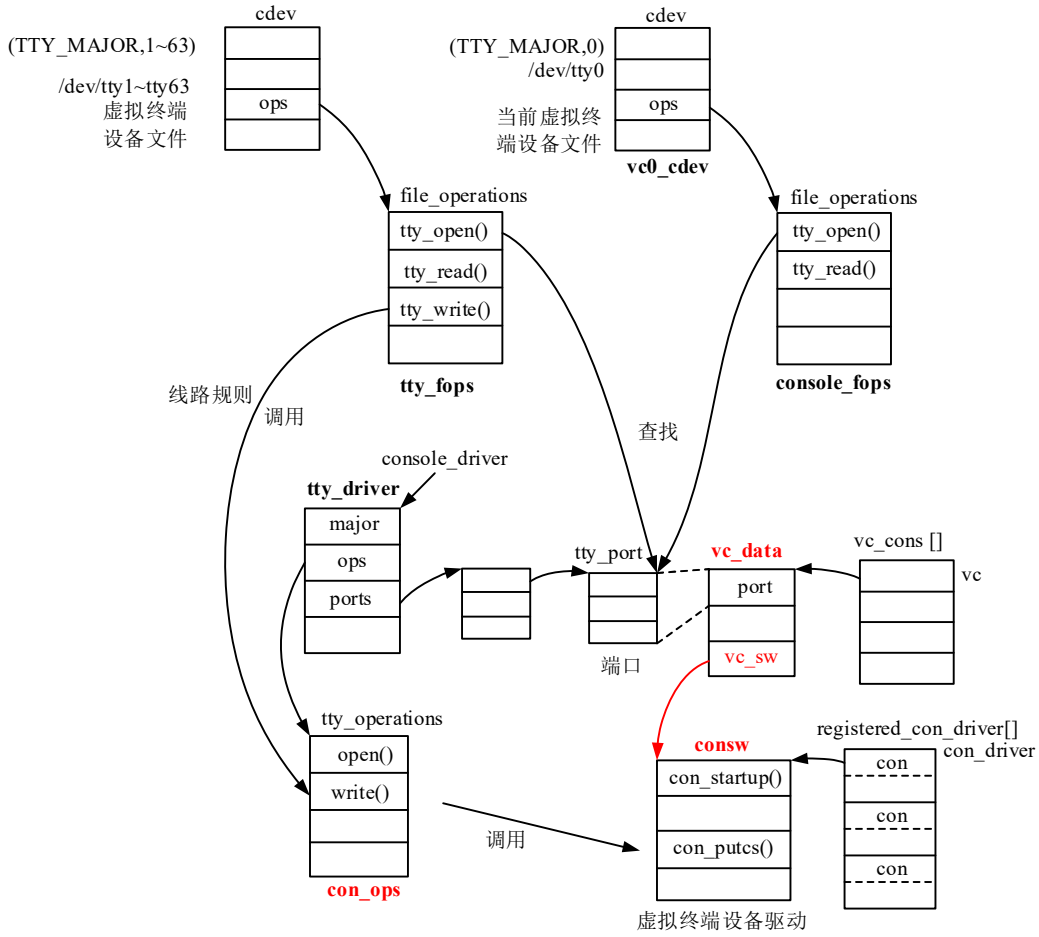
在 fbcon_start()函数内将调用前面介绍的 do_take_over_console()函数，注册 consw 结构体实例 **fb_con**，函数代码如下：

```
static void fbcon_start(void)
{
    if (num_registered_fb) {
        int i;
        console_lock();

        for (i = 0; i < FB_MAX; i++) {          /*查找第一个注册 fb_info 实例的序号*/
            if (registered_fb[i] != NULL) {
                info_idx = i;
                break;
            }
        }
        do_fbcon_takeover(0);        /*注册 consw 结构体实例 fb_con*/
        console_unlock();
    }
}
static int do_fbcon_takeover(int show_logo)
/*show_logo：是否显示 logo，这里为 0*/
{
    int err, i;
```

```
    if (!num_registered_fb)
        return -ENODEV;

    if (!show_logo)
        logo_shown = FBCON_LOGO_DONTSHOW;

    for (i = first_fb_vc; i <= last_fb_vc; i++)
        con2fb_map[i] = info_idx;

    err = do_take_over_console(&fb_con, first_fb_vc, last_fb_vc,fbcon_is_default);
                                                        /*注册 consw 结构体实例 fb_con*/

    if (err) {
        ...
    } else {
        fbcon_has_console_bind = 1;
    }
    return err;
}
```

5 虚拟终端设备操作

在注册虚拟终端设备驱动的 tty_driver 实例时，将为/dev/tty1~63 虚拟终端设备设置并注册 cdev 实例，关联的 file_operations 实例为 tty_fops，并创建设备文件。在 vty_init()函数中将为/dev/tty0 设备文件设置并添加 cdev 实例，关联的 file_operations 实例为 console_fops，并创建设备文件，如下图所示。



对于/dev/tty1~63 设备文件，其关联 file_operations 实例为 tty_fops，前面已经介绍过了，文件操作流程与前面串口终端设备操作类似，不再解释了。

对于/dev/tty0 设备文件，其关联 file_operations 实例为 console_fops。/dev/tty0 设备文件与/dev/tty1~63 设备文件的主要不同是在打开操作时，并不是按从设备号查找虚拟终端设备，而是关联当前虚拟终端设备。详见下面由设备文件查找 tty_driver 实例的函数代码：

```
static struct tty_driver *tty_lookup_driver(dev_t device, struct file *filp,int *noctty, int *index)
{
    struct tty_driver *driver;

    switch (device) {
#ifdef CONFIG_VT
    case MKDEV(TTY_MAJOR, 0): {          /*/dev/tty0 设备文件*/
        extern struct tty_driver *console_driver;
        driver = tty_driver_kref_get(console_driver);
        *index = fg_console;          /*index 为虚拟终端序号，设为当前虚拟终端*/
        *noctty = 1;
        break;
    }
#endif
    ...
    default:
```

```
        driver = get_tty_driver(device, index);
        if (!driver)
            return ERR_PTR(-ENODEV);
        break;
    }
    return driver;
}
```

/dev/tty0 设备文件与/dev/tty1~63 设备文件的其它操作相同，这里不再解释了。

虚拟终端设备控制处理函数位于/drivers/tty/vt/vt_ioctl.c 文件内，请读者自行阅读源代码。

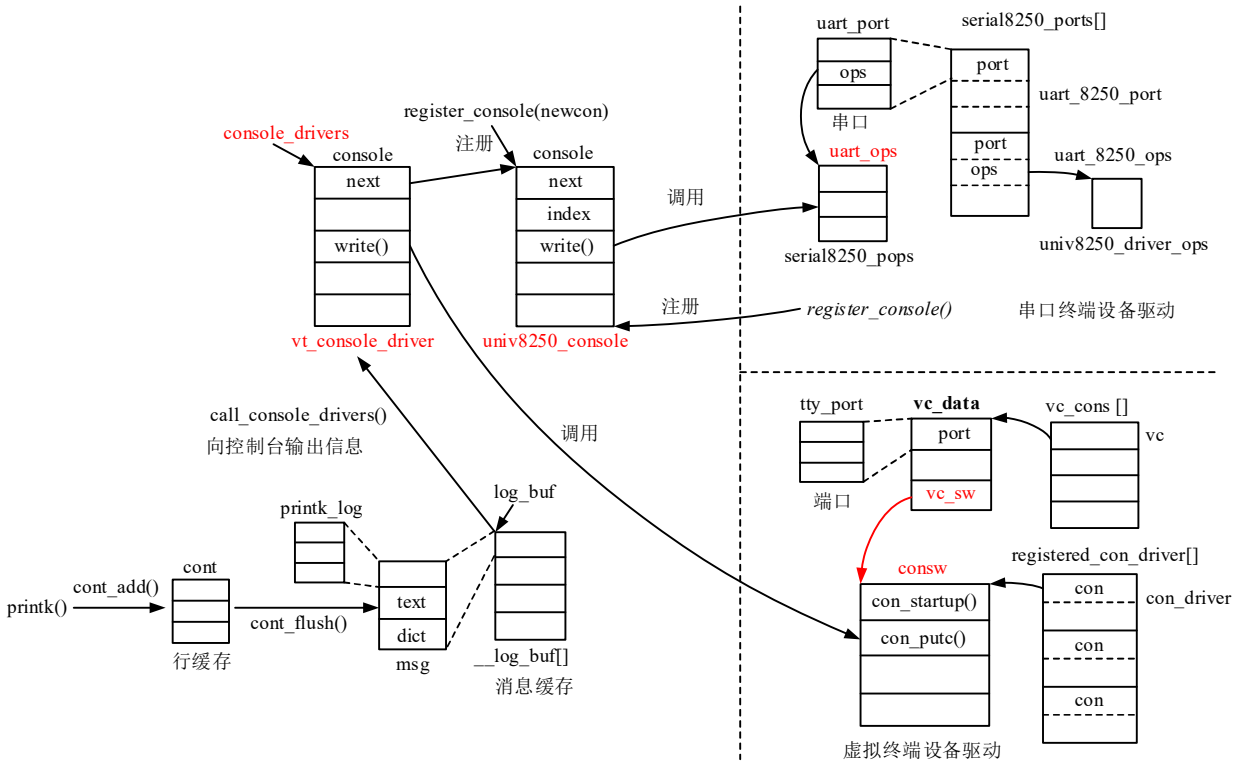
9.9.2 系统控制台

系统控制台是指用于内核输出信息的设备。内核代码中通常通过 printk()函数及其封装函数向系统控制台输出信息。系统控制台的实现代码位于/kernel/printk/printk.c 文件内。

前面介绍的串口终端设备和虚拟终端设备等，都可以注册为系统控制台，用于输出内核信息。

1 概述

系统控制台用来输出内核（系统）信息，内核通过 printk()等函数向系统控制台输出字符串信息。系统控制台在内核中由 console 结构体表示，系统控制台实现框架如下图所示：



注册的系统控制台 console 实例在内核中组成单链表，console_drivers 指向第一个成员。console 实例中的 index 成员表示终端设备通道编号，例如，将串口用用系统控制台时 index 表示串口通道编号。console 实例中的 write()函数负责把字符串输出到硬件设备。

printk()函数中通过 cont 结构体来缓存一行数据，行数据再刷出到消息数组中。消息数组中每个数组项

开头是一个 printk_log 结构体实例，后面接要输出的数据。

消息具有日志级别属性，系统控制台也具有日志级别属性，只有日志级别值小于系统控制台日志级别的消息才会被调用 console 实例中的 write()函数输出到系统控制台。

printk()函数输出的消息会输出到所有注册的系统控制台。

2 注册系统控制台

系统控制台由 console 结构体表示，定义如下（/include/console.h）：

```
struct console {
    char name[16];          /*名称*/
    void (*write)(struct console *, const char *, unsigned);     /*向系统控制台写出数据*/
    int   (*read)(struct console *, char *, unsigned);           /*从系统控制台读数据*/
    struct tty_driver *(*device)(struct console *, int *);       /*由控制台查找对应的 tty_driver 实例*/
    void (*unblank)(void);
    int   (*setup)(struct console *, char *);          /*启动函数，注册 console 实例时调用*/
    int   (*match)(struct console *, char *name, int idx, char *options);     /*系统控制台匹配/查找函数*/
    short      flags;          /*标记*/
    short      index;          /*设备通道号*/
    int   cflag;
    void *data;                /*系统控制台私有数据*/
    struct   console *next;        /*下一个注册的系统控制台*/
};
```

console 结构体中主要成员简介如下：

●**index**：作为系统控制台的通道号，例如，将串口注册为系统控制台时，串口控制器中可能有多个通道，index 表示哪个通道用于系统控制台，由“console=”命令行参数设置，对应 ttySX 设备文件。若没有设置则默认使用通道 0。

●**write()**：向系统控制台输出字符串的函数。

●**device()**：查找系统控制台设备对应终端驱动 tty_driver 实例，在打开/dev/console 设备文件时调用。

●**flags**：标记，取值如下：

```
#define   CON_PRINTBUFFER   (1)
#define   CON_CONSDEV       (2)   /*最后一个"console=***"命令行参数对应的系统控制台*/
#define   CON_ENABLED       (4)   /*系统控制台已使能，可用*/
#define   CON_BOOT          (8)     /*只在内核启动阶段使用的系统控制台*/
#define   CON_ANYTIME       (16)  /* Safe to call when cpu is offline */
#define   CON_BRL           (32)  /* Used for a braille device */
#define   CON_EXTENDED      (64)  /* Use the extended output format a la /dev/kmsg */
```

■注册函数

内核中有两种类型的 console，一种是启动 console，另一种是真实的 console。

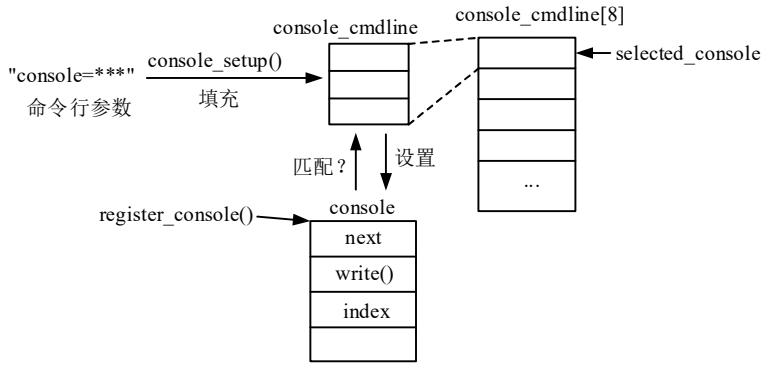
启动 console 需选择 EARLY_PRINTK 配置选项，early_console 全局指针指向启动 console。启动 console 用于在内核启动早期输出内核信息，由体系结构相关代码定义并赋予 early_console 指针（设置 CON_BOOT

标记），这里就不做介绍了。

当注册的真实的 console 后，所有启动 console 都将注销，且不能再注册启动 console 了。注册真实 console 实例的函数为 register_console(struct console *newcon)。

在介绍注册 console 实例的函数前，先看一下"console=***"命令行参数的处理，如下图所示。“console=”命令行参数用于指定用作系统控制台的设备及参数，例如，“console=ttyS0,115200”，表示串口 0 设为系统控制台，波特率为 115200。

“console=”命令行参数处理函数 console_setup()定义在/kernel/printk/printk.c 文件内。



内核定义了 console_cmdline 结构体数组（项数为 8）用于保存"console="命令行参数传递的系统控制台参数，命令行参数值通过 console_setup()函数依次写入 console_cmdline 结构体数组项中。

console_cmdline 结构体定义在/kernel/printk/console_cmdline.h 头文件：

```
struct console_cmdline
{
    char name[16];              /*系统控制台设备对应的 tty_driver 驱动名称*/
    int  index;                 /*设备中用作系统控制台的通道号，如串口通道编号*/
    char *options;          /*参数*/
#ifdef CONFIG_A11Y_BRAILLE_CONSOLE
    char *brl_options;          /* Options for braille driver */
#endif
};
```

全局变量 preferred_console 和 selected_console 的初始值为-1。selected_console 参数在处理"console="命令行参数时，设为最后填充数组项的索引值。

例如，如果命令行参数中包含一个"console=***"，则 selected_console 值为 0。preferred_console 值通常为最后一个"console="命令参数对应 console_cmdline 在数组中的索引值。

在注册 console 实例时，将会在 console_cmdline 结构体数组中查找是否有与之匹配的数组项，如果有则用其中参数设置 console 实例（系统控制台设备）。"console=***"命令行参数的处理先于 console 实例的注册进行。

内核中所有注册的 console 实例组成一个单链表，全局指针 console_drivers 指向单链表中第一个成员。注册 console 实例的函数为 **register_console()**，定义如下（/kernel/printk/printk.c）：

```
void register_console(struct console *newcon)
/*newcon：指向注册的 console 实例*/
{
    int i;
```

```
unsigned long flags;
struct console *bcon = NULL;
struct console_cmdline *c;

...       /*检查 console 实例是否已经注册*/

/*如果是注册 CON_BOOT 控制台，需保证现在还没有真实的系统控制台*/
if (console_drivers && newcon->flags & CON_BOOT) {
    for_each_console(bcon) {        /*遍历已注册的 console 实例*/
        if (!(bcon->flags & CON_BOOT)) {
            pr_info("Too late to register bootconsole %s%d\n",newcon->name, newcon->index);
            return;
        }
    }
}

if (console_drivers && console_drivers->flags & CON_BOOT)       /*注册启动 console*/
    bcon = console_drivers;

if (preferred_console < 0 || bcon || !console_drivers)
    preferred_console = selected_console;

if (preferred_console < 0) {        /*没有传递"console="命令行参数*/
    if (newcon->index < 0)
        newcon->index = 0;        /*默认使用通道 0*/
    if (newcon->setup == NULL ||newcon->setup(newcon, NULL) == 0) {
        newcon->flags |= CON_ENABLED;        /*激活系统控制台设备*/
        if (newcon->device) {
            newcon->flags |= CON_CONSDEV;
            preferred_console = 0;
        }
    }
}

/*遍历 console_cmdline 实例，检查是否是"console=***"命令行参数指定的控制台*/
for (i = 0, c = console_cmdline;i < MAX_CMDLINECONSOLES && c->name[0];i++, c++) {
    if (!newcon->match ||newcon->match(newcon, c->name, c->index, c->options) != 0) {
        /*如果没有定义 match()函数，或 match()返回非 0（不匹配），则执行以下匹配操作*/

        BUILD_BUG_ON(sizeof(c->name) != sizeof(newcon->name));
        if (strcmp(c->name, newcon->name) != 0)        /*名称不同，不匹配，相同则匹配*/
            continue;
        if (newcon->index >= 0 &&newcon->index != c->index)     /*通道编号不同，不匹配*/
```

```
                continue;
            if (newcon->index < 0)          /*名称和通道匹配（-1）*/
                newcon->index = c->index;          /*设置通道编号*/

            if (_braille_register_console(newcon, c))
                return;

            if (newcon->setup &&newcon->setup(newcon, c->options) != 0)       /*调用 setup()函数*/
                break;
        }

        /*以下是注册的 console 实例与 console_cmdline 实例匹配的情况，match()返回 0*/
        newcon->flags |= CON_ENABLED;          /*控制台已使能*/
        if (i == selected_console) {
            newcon->flags |= CON_CONSDEV;
            preferred_console = selected_console;
        }
        break;
    }       /*for 循环结束*/

    if (!(newcon->flags & CON_ENABLED))
        return;

    if (bcon && ((newcon->flags & (CON_CONSDEV | CON_BOOT)) == CON_CONSDEV))
        newcon->flags &= ~CON_PRINTBUFFER;

    console_lock();
    if ((newcon->flags & CON_CONSDEV) || console_drivers == NULL) {
        newcon->next = console_drivers;          /*插入单链表头部*/
        console_drivers = newcon;
        if (newcon->next)
            newcon->next->flags &= ~CON_CONSDEV;
    } else {
        newcon->next = console_drivers->next;       /*插入单链表中第二个位置*/
        console_drivers->next = newcon;
    }

    if (newcon->flags & CON_EXTENDED)
        if (!nr_ext_console_drivers++)
            pr_info("printk: continuation disabled due to ext consoles, expect more fragments in \
                    /dev/kmsg\n");

    if (newcon->flags & CON_PRINTBUFFER) {
```

```
            raw_spin_lock_irqsave(&logbuf_lock, flags);
            console_seq = syslog_seq;
            console_idx = syslog_idx;
            console_prev = syslog_prev;
            raw_spin_unlock_irqrestore(&logbuf_lock, flags);
            exclusive_console = newcon;
        }
        console_unlock();          /*输出缓存信息*/
        console_sysfs_notify();
        ...
        if (bcon &&((newcon->flags & (CON_CONSDEV | CON_BOOT)) == CON_CONSDEV) &&
                    !keep_bootcon) {
            for_each_console(bcon)
                if (bcon->flags & CON_BOOT)        /*注销 CON_BOOT 系统控制台*/
                    unregister_console(bcon);
        }
    }
```

register_console(struct console *newcon)函数内要做一些检查，简单地说就是将 newcon 实例插入到全局单链表 console_drivers 中。如果 newcon 实例设置了 CON_CONSDEV 标记（最后"console=***"参数指定的控制台），则添加到单链表头部，否则添加到单链表中第二个位置。注册函数还要激活系统控制台设备。

通常 console_drivers 单链表中第一个实例表示当前系统控制台，对应/dev/console 设备文件关联的系统控制台，详见下一小节。

■系统控制台示例

注册 console 实例的函数为 register_console()。通常终端设备可用于系统控制台，例如，在串口终端设备驱动程序中若要将串口注册为系统控制台，则需要定义并注册 console 实例。

8250 串口驱动注册 console 实例如下所示（/drivers/tty/serial/8250/8250_core.c）：

```
static struct console univ8250_console = {     /*console 实例，需选择 SERIAL_8250_CONSOLE 选项*/
    .name           = "ttyS",
    .write          = univ8250_console_write,
    .device         = uart_console_device,
    .setup          = univ8250_console_setup,
    .match          = univ8250_console_match,
    .flags          = CON_PRINTBUFFER | CON_ANYTIME,
    .index          = -1,              /*由“console=”命令行参数确定 index 值，或为 0*/
    .data       = &serial8250_reg,          /*uart_driver 实例，私有数据*/
};

static int __init univ8250_console_init(void)
{
    if (nr_uarts == 0)
        return -ENODEV;
```

```
    serial8250_isa_init_ports();          /*初始化串口列表*/
    register_console(&univ8250_console);          /*注册 console 实例*/
    return 0;
}
console_initcall(univ8250_console_init);          /*初始化函数指针放入指定段*/
```

univ8250_console_init()函数调用 register_console()函数注册 console 实例，此函数指针存入内核镜像指定段中，那么这个函数在什么时候被调用呢？

在初始化函数 **console_init()**函数中将调用通过 console_initcall()宏存入指定段的函数（只存放了函数入口地址），函数调用关系为：start_kernel()->console_init()。

console_init()函数定义如下（/drivers/tty/tty_io.c）：

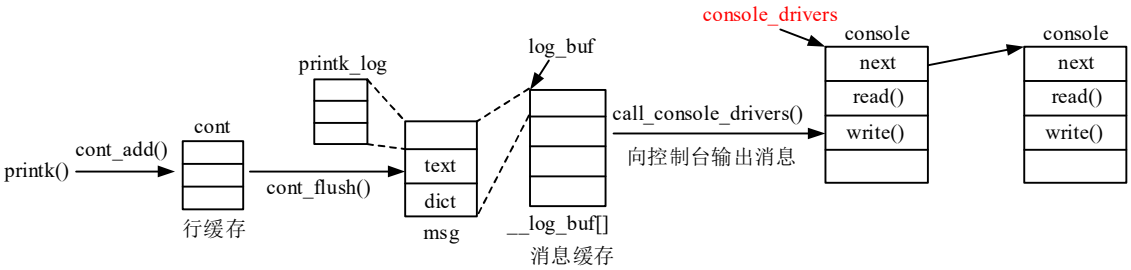
```
void __init console_init(void)
{
    initcall_t *call;

    tty_ldisc_begin();    /*注册 tty_ldisc_N_TTY 实例，/drivers/tty/tty_ldisc.c*/

    call = __con_initcall_start;
    while (call < __con_initcall_end) {    /*调用 console_initcall(fn)宏声明的初始化函数*/
        (*call)();
        call++;
    }
}
```

console_initcall(fn)宏声明的函数（用于注册 console 实例）在"console=***"命令行参数处理之后调用。

3 printk()

内核通常通过 printk()函数向系统控制台输出信息，其它内核输出信息的函数都是 printk()函数的包装器，详见/include/linux/printk.h 头文件。printk()函数实现在/kernel/printk/printk.c 文件内。

下面先看一下 printk()函数执行的流程，如下图所示：



内核定义了 cont 结构体（只有一个实例）用于缓存一行数据，有时一行数据可能分多次输入，cont 结构体用于收集同一行的数据。当收集到一行数据时，将写入__log_buf[]数组表示的消息数组项中。

__log_buf[]数组中每个消息开头是一个 printk_log 结构体实例，其后是消息数据和参数值等信息。数组__log_buf[]中的消息数据随后由各系统控制台 console 实例中的 write()函数输出到设备中。

需要注意的是__log_buf[]数组中的消息有一个日志级别属性，系统控制台也有一个级别属性，只有消息日志级别数值小于系统控制台级别数值的消息才会输出到系统控制台，详见下文。

■数据结构

内核为输出信息定义了日志级别的概念，可视为信息的优先级，它由一个数值来表示，值越小表示优先级越高。系统控制台也被赋予一个日志级别值，只有级别值小于等于系统控制台级别值的消息才会输出到系统控制台。

printk()函数在参数中可通指定信息的日志级别，若不指定则使用默认的级别值，如下：

```
#define MESSAGE_LOGLEVEL_DEFAULT    CONFIG_MESSAGE_LOGLEVEL_DEFAULT
                /*/include/linux/printk.h，配置选项位于/lib/Kconfig.debug，范围是 1~7，默认为 4*/
```

在/include/linux/printk.h 头文件中定义了系统控制台日志级别值的宏：

```
#define    CONSOLE_LOGLEVEL_SILENT          0   /*不输出任何消息*/
#define    CONSOLE_LOGLEVEL_MIN             1   /*可用的系统控制台最小日志级别值*/
#define    CONSOLE_LOGLEVEL_QUIET           4   /* Shhh ..., when booted with "quiet" */
#define    CONSOLE_LOGLEVEL_DEFAULT         7   /*输出比 KERN_DEBUG 级别高的信息*/
#define    CONSOLE_LOGLEVEL_DEBUG           10 /*可输出调试信息*/
#define    CONSOLE_LOGLEVEL_MOTORMOUTH    15      /* You can't shut this one up */
```

内核在/kernel/printk/printk.c 文件内定义了并初始化了 console_printk[4]数组，用于保存各日志级别值：

```
int console_printk[4] = {                   /*每个数组项保存了一个日志级别值*/
   CONSOLE_LOGLEVEL_DEFAULT,    /*系统控制台的日志级别，console_loglevel 宏表示*/
   MESSAGE_LOGLEVEL_DEFAULT,    /*消息默认的日志级别，default_message_loglevel*/
   CONSOLE_LOGLEVEL_MIN,        /*系统控制台最小日志级别，minimum_console_loglevel */
   CONSOLE_LOGLEVEL_DEFAULT,    /*系统控制台默认的日志级别，default_console_loglevel */
};
```

cont 结构体及其实例用于缓存行数据，定义如下（/kernel/printk/printk.c）：

```
static struct cont {
    char buf[LOG_LINE_MAX];          /*缓存数据大小（1024-32），前 32 字节为前缀*/
    size_t len;                 /* length == 0 means unused buffer */
    size_t cons;                /*已写入系统控制台的字节数*/
    struct task_struct *owner;     /*输出数据的进程（内核线程）*/
    u64 ts_nsec;                /*输出时间戳，纳秒*/
    u8 level;                   /*消息的日志级别*/
    u8 facility;                /*消息由谁产生（用户、内核）*/
    enum log_flags flags;           /*标记*/
    bool flushed:1;                 /*是否已提交到__log_buf[]数组*/
} cont;         /*同名的实例*/
```

cont 结构体中部分成员简介如下：

●**facility**：表示消息由谁产生（内核用、户），0 表示由内核产生（LOG_KERN），1 表示由用户空间产生（LOG_USER）。

●**flags**：标记，取值定义如下（/kernel/printk/printk.c）：

```
enum log_flags {
    LOG_NOCONS    = 1,      /*已经刷出，不需要写入系统控制台*/
    LOG_NEWLINE   = 2,      /* text ended with a newline */
    LOG_PREFIX    = 4,      /*输出数据有前缀*/
    LOG_CONT   = 8,         /* text is a fragment of a continuation line，是连续行的拆分*/
};
```

cont 结构体实例中缓存的行，需要刷出到__log_buf[]数组项中。__log_buf[]数组项中的每个消息开头是一个 printk_log 结构体实例，定义如下（/kernel/printk/printk.c）：

```
struct printk_log {
    u64 ts_nsec;        /*时间戳*/
    u16 len;            /*整个消息的长度*/
    u16 text_len;       /*text 的长度*/
    u16 dict_len;       /*dict 的长度*/
    u8 facility;        /*同 cont 结构体*/
    u8 flags:5;         /* internal record flags */
    u8 level:3;         /*日志级别*/
};
```

__log_buf[]数组项中 printk_log 结构体实例之后是消息数据（text），最后是 dict 数据（键值对数据）。

■函数调用关系

printk()函数调用关系简列如下：

```
printk(const char *fmt, ...)          /*/kernel/printk/printk.c*/
    └► vprintk_default(const char *fmt, va_list args)
        └► vprintk_emit(0, LOGLEVEL_DEFAULT, NULL, 0, fmt, args)
            ├► vscnprintf(text, sizeof(textbuf), fmt, args)     /*/tools/include/linux/kernel.h*/
            │   └► vsnprintf(buf, size, fmt, args)              /*/lib/vsprintf.c，将字符串进行格式化*/
            ├► cont_add()/cont_flush()     /*将字符串写入cont实例和__log_buf[]数组*/
            └► console_unlock(void)        /*遍历log_buf[]各数组项，/kernel/printk/printk.c*/
                └► call_console_drivers()  /*将数组项中信息写出控制台*/
                    └► console->write()    /*遍历console实例链表，调用各实例的write()函数*/
```

printk()函数主要的流程就是调用 cont_add()函数将输出信息写入 cont 实例，然后调用 cont_flush()函数将 cont 实例中信息写入__log_buf[]数组项，最后遍历__log_buf[]数组项将信息输出到所有注册的系统控制台中。

console_unlock()函数遍历__log_buf[]数组项，对每个数组项调用 call_console_drivers()函数，只有日志级别小于系统控制台日志级别的消息才会调用所有注册的（可用的）console 实例的 write()，将数组项信息输出到系统控制台。

这里还有一个问题需要说明一下。默认情况下直接调用 printk()函数时，函数参数中没有指定消息的日志级别，在 vprintk_emit()函数中会将消息设置为默认的日志级别（MESSAGE_LOGLEVEL_DEFAULT）。

在/include/linux/printk.h 头文件中定义的其它的输出信息函数，将加入消息日志级别参数，例如：

```
#define pr_info(fmt, ...) \
    printk(KERN_INFO pr_fmt(fmt), ##__VA_ARGS__)
```

以上日志级别 KERN_INFO 宏定义在/include/linux/kern_levels.h 头文件。

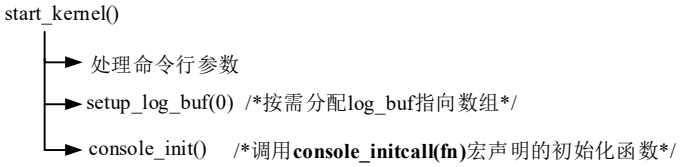
pr_info()宏的意思是将日志级别值插入到输出信息的开头，也就是说日志级别信息保存在输出字符串的开头两个字节中，其中第一个字节是标记（标识开头两个字节是表示日志级别信息），第二个字节表示日志级别值。

在 vprintk_emit()函数中将根据开头两个字节，识别出消息的日志级别，以填充至 cont 实例和__log_buf[]数组项中。

■初始化

针对系统控制台有一些初始化工作以及命令行参数，下面做简单介绍。

先看内核启动函数 start_kernel()中与系统控制台相关的函数：



__log_buf[__LOG_BUF_LEN]数组大小由 LOG_BUF_SHIFT 配置选项决定（默认为 17，即 128KB，/init/Kconfig）。log_buf 全局变量指向__log_buf[]数组。命令行参数"log_buf_len"可用于设置__log_buf[]数组大小，不过需要比默认大小大才有效。

setup_log_buf(0)函数定义在/kernel/printk/printk.c 文件内，用于为__log_buf[]数组重新分配空间，如果需要的话。另外，如果是 SMP 系统还需要考虑 CPU 额外空间的影响。

console_init()函数前面介绍过，就是注册默认的线路规则，并调用 console_initcall()宏声明的函数，这些函数主要是激活系统控制台，注册 console 实例等。

4 当前系统控制台

Linux 将（当前）系统控制台通过设备文件导出到用户空间，使用户进程也可以通过系统控制台输入输出信息。当前系统控制台由/dev/console 设备文件表示，在 tty_init()初始化函数中为其设置并添加了 cdev 实例并申请了设备号，如下所示。

```
int __init tty_init(void)
{
    ...
    /*初始化并添加/dev/console 设备文件对应的 cdev 实例（固定设备号）*/
    cdev_init(&console_cdev, &console_fops);
    if (cdev_add(&console_cdev, MKDEV(TTYAUX_MAJOR, 1), 1) ||
        register_chrdev_region(MKDEV(TTYAUX_MAJOR, 1), 1, "/dev/console") < 0)    /*申请设备号*/
            panic("Couldn't register /dev/console driver\n");
    ...
}
```

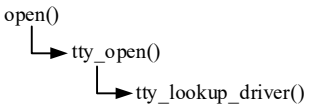
/dev/console 设备文件主设备号为 TTYAUX_MAJOR，从设备号为 1。cdev 实例关联的 file_operations 实例为 console_fops，定义如下（/drivers/tty/tty_io.c）：

```
static const struct file_operations console_fops = {
```

```
    .llseek     = no_llseek,
    .read       = tty_read,
    .write      = redirected_tty_write,
    .poll       = tty_poll,
    .unlocked_ioctl     = tty_ioctl,
    .compat_ioctl  = tty_compat_ioctl,
    .open           = tty_open,
    .release        = tty_release,
    .fasync         = tty_fasync,
};
```

console_fops 实例中的函数与前面介绍的 tty_fops 实例中的函数基本相同。这里主要来看一下 tty_open()函数如何为**/dev/console** 设备文件查找终端驱动 tty_driver 实例。

在 open()系统调用中将调用 console_fops 实例中的 open()函数 tty_open()，在此函数中将为设备文件查找对应的终端驱动 tty_driver 实例。



tty_lookup_driver()函数代码简列如下（/drivers/tty/tty_io.c）：

```
static struct tty_driver *tty_lookup_driver(dev_t device, struct file *filp,int *noctty, int *index)
{
    struct tty_driver *driver;

    switch (device) {       /*设备号*/
#ifdef CONFIG_VT        /*支持虚拟终端*/
    ...
#endif
    case MKDEV(TTYAUX_MAJOR, 1): {          /*/dev/console 设备文件*/
        struct tty_driver *console_driver = console_device(index);
            /*console_drivers 单链表中第一个具有 tty_driver 实例的成员，/kernel/printk/printk.c*/
        if (console_driver) {
            driver = tty_driver_kref_get(console_driver);       /*增加引用计数*/
            if (driver) {
                filp->f_flags |= O_NONBLOCK;        /*文件为非阻塞操作*/
                *noctty = 1;
                break;
            }
        }
        return ERR_PTR(-ENODEV);
    }
    default:
        driver = get_tty_driver(device, index);    /*根据设备号查找 tty_driver 实例*/
        ...
        break;
```

```
    }
    return driver;
}
```

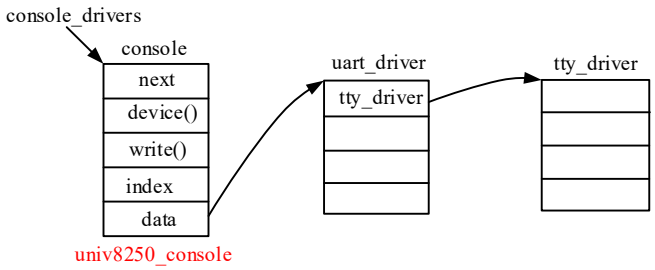
对于/dev/console 设备文件，将调用 **console_device**(index)函数查找对应的 tty_driver 实例，index 参数是一个指针，指向的地址用来保存/dev/console 设备文件对应 tty_driver 实例中的端口序号。

console_device()函数定义如下（/kernel/printk/printk.c）：

```
struct tty_driver *console_device(int *index)
{
    struct console *c;
    struct tty_driver *driver = NULL;

    console_lock();
    for_each_console(c) {          /*遍历 console 实例链表*/
        if (!c->device)
            continue;
        driver = c->device(c, index);        /*tty_driver 实例，c->index 确定端口号*/
        if (driver)
            break;
    }
    console_unlock();        /*输出缓存数据*/
    return driver;
}
```

console_device()函数遍历 console 实例链表，返回第一个能通过 c->device()函数获取的 tty_driver 实例。也就是说/dev/console 设备文件关联到第一个同时注册为系统控制台和终端设备的设备。

如下图所示，8250 串口注册的 console 实例，其 data 成员指同 uart_driver 实例，deivce()函数以此查找到 uart_driver 实例，其 tty_driver 成员指向终端驱动 tty_driver 实例。



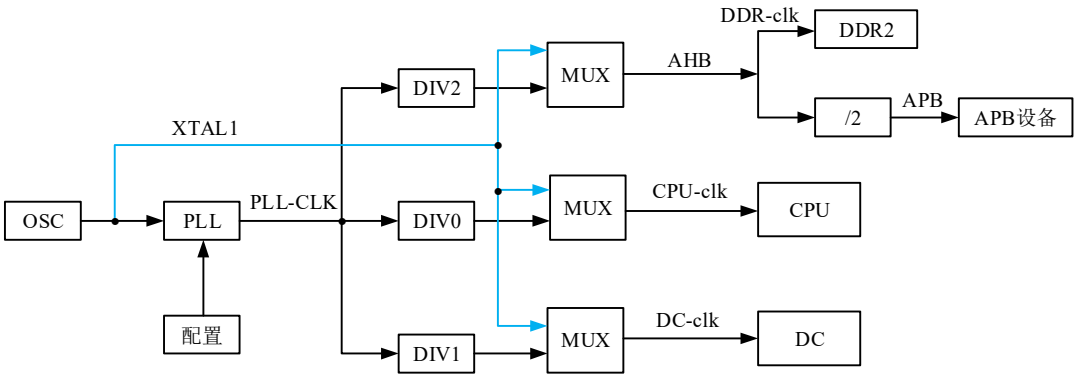
内核启动后期将创建 kernel_init 线程，它是第一个用户进程的前身，在 kernel_init 线程中将打开设备文件/dev/console 作为线程的标准输入、标准输出和标准错误输出文件，这三个文件将会传递给第一个用户进程。也就是说，默认情况下用户进程将标准输入、标准输出和标准错误输出定位到当前系统控制台，当然用户进程可以进行更改。

9.10 硬件时钟框架

现代处理器都有复杂的硬件时钟（Clock）模块，用于向 CPU 和各外设模块提供工作时钟。内核需要对时钟模块进行控制，因此定义了 CCF 框架（Common Clock Framework），用于提供硬件时钟操作接口。

9.10.1 CCF 框架

处理器 Clock 模块中通常包含产生基准时钟的振荡器 OSC（有源振荡器、无源振荡器）、用于倍频的 PLL（锁相环）、分频器 divider、多路选择器 MUX（时钟选择）等。例如：下图为龙芯 1B 处理器时钟 Clock 模块结构图：



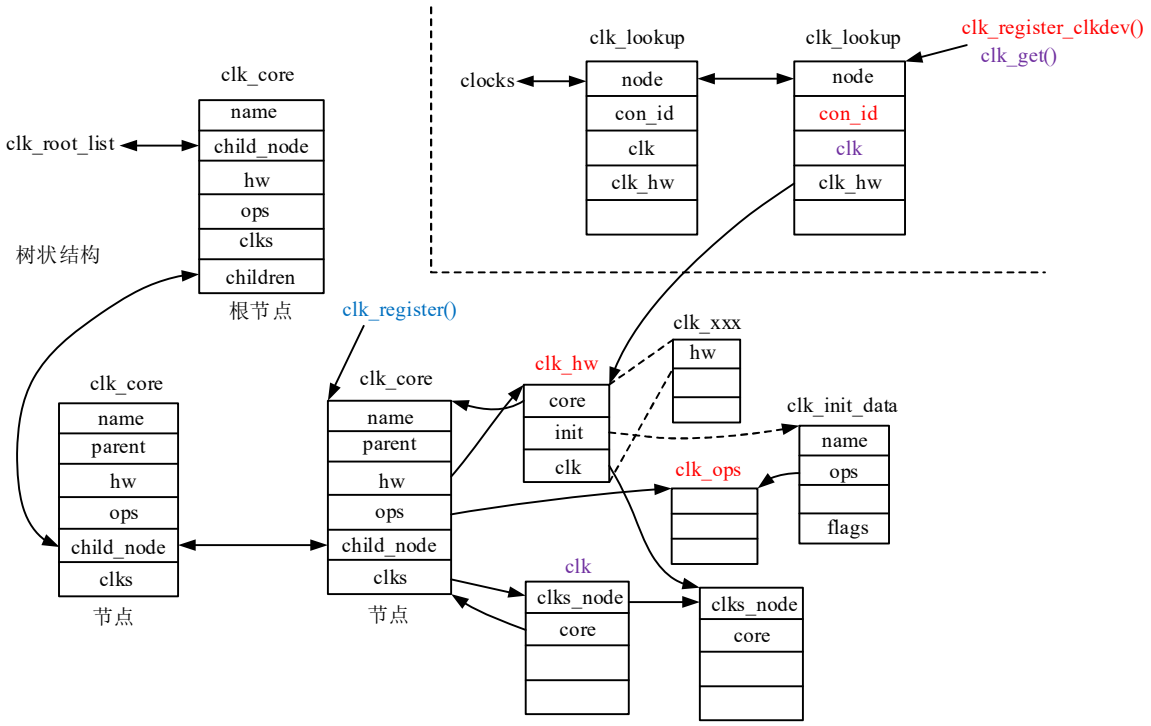
振荡器 OSC 产生固定的硬件时钟通过 PLL 倍频，倍频后再经过分频器（DIV）到达多路选择器（MUX）多路选择器另一路输入直接来自振荡器，被选中的时钟提供给外设模块。处理器可配置 PLL 输出频率、分频器分频系数以及多路选择器选择哪一路时钟作为输出等。龙芯 1B 中 DDR2 使用 AHB 总线时钟频率，APB 总线时钟频率固定是 AHB 总线频率的一半。

内核定义了通用时钟框架（CCF，需选择 COMMON_CLK 配置选项，龙芯 1B 默认选择），CCF 中将 CLOCK 模块中的 OSC、PLL、MUX 等各个部件视为节点，并构成树状结构，树状结构中每个节点由结构体 clk_core 表示。节点输出的时钟由 clk 结构体表示，一个节点输出的时钟可供多个设备使用，每个使用节点输出时钟的设备对应一个 clk 实例（设备时钟）。设备驱动程序可获取 clk 实例，通过此实例实现获取时钟频率和设置时钟频率操作等。

CCF 相关代码位于/drivers/clk/目录下。

1 概述

Linux 内核通用时钟框架如下图所示：



Clock 模块中各节点由 clk_core 结构体表示，并依实际硬件结构构成父子的树状结构。子节点的输入来自父节点的输出，多路选择器节点可以有多个父节点。

不同类型的节点由 clk_xxx 结构体表示，结构体中第一个成员是通用的 clk_hw 结构体，表示节点的硬件特性。clk_ops 结构体是节点操作函数的集合，如获取时钟频率，设置时钟频率等操作。

注册节点时，需为节点创建并设置 clk_xxx 结构体实例，含重要的 clk_ops 结构体实例，然后调用通用的 clk_register()函数创建并注册 clk_core 实例（含关联的初始 clk 实例）。上图中 clk_init_data 是一个临时的结构体，实例只在注册函数时使用，注册完成后就释放了。

若要使节点输出时钟对使用此时钟的设备可见（可对节点进行操作），需调用 clk_register_clkdev(struct clk ***clk**, const char ***con_id**,const char *dev_fmt, ...)函数（clk 指向 clk_core 实例中初始 clk 实例）为输出时钟创建并注册用于查找的 clk_lookup 实例（由全局双链表管理），clk_lookup 实例对应使用此时钟的设备。

设备驱动程序可通过 clk_get(struct device *dev, const char ***con_id**)函数查找 clk_lookup 实例（con_id 为关键字），并为设备创建 clk 实例，添加到 clk_core 实例散列链表中，最后返回 clk 实例指针。随后设备驱动程序就可以通过返回的 clk 实例，执行获取、设置时钟频率等操作。

2 数据结构

下面简要介绍一下 CCF 中相关数据结构的定义。

■clk_core/clk

clk_core 是组成 CCF 树状结构的核心数据结构，表示 CLOCK 模块中的部件，结构体定义如下：

```
struct clk_core {                    /*/drivers/clk/clk.c*/
    const char          *name;      /*名称*/
    const struct clk_ops *ops;      /*来自到 hw 成员指向的 clk_hw 实例*/
    struct clk_hw       *hw;        /*指向 clk_hw 结构体，见下文*/
    struct module       *owner;
    struct clk_core     *parent;    /*指向父节点 clk_core 实例*/
```

```
    const char          **parent_names;    /*父节点名称字符指针数组*/
    struct clk_core     **parents;         /*指向父节点指针数组，数组项指向 clk_core 实例*/
    u8                  num_parents;       /*父节点数量，若是根节点则为 0*/
    u8                  new_parent_index;
    unsigned long       rate;              /*节点当前实际输出时钟频率*/
    unsigned long       req_rate;          /*最近重设操作设置的频率，与 rate 可能有误差*/
    unsigned long       new_rate;
    struct clk_core     *new_parent;
    struct clk_core     *new_child;
    unsigned long       flags;     /*标记成员，/include/linux/clk-provider.h*/
    unsigned int        enable_count;      /*使能时钟计数*/
    unsigned int        prepare_count;
    unsigned long       accuracy;          /*精度*/
    int                 phase;
    struct hlist_head   children;          /*子节点链表头*/
    struct hlist_node   child_node;    /*链接兄弟节点，表头为父节点的 children 成员*/
    struct hlist_head   clks;          /*散列链表头，链接 clk 实例*/
    unsigned int        notifier_count;
    ...
    struct kref         ref;       /*引用计数，增加一个 clk 实例时，引用计数加 1*/
};
```

clk_core 结构体主要成员简介如下：

●**name**：名称。

●**ops**：指向 clk_ops 结构体，结构体实例来自于 clk_hw 实例，见下文。

●**hw**：指向 clk_hw 结构体，此结构体用于区分不同类型的部件，见下文。

●**parent**：指向父节点 clk_core 实例。

●**parent_names**：指向父节点名称的指针数组。

●**parents**：指向指针数组，数组项指向父节点 clk_core 实例。

●**children**：子节点散列链表头。

●**child_node**：链接兄弟节点，表头为父节点的 children 成员，或添加到全局双链表。

●**rate**：当前输出时钟频率。

●**clks**：散列链表头，链接 clk 实例。由于一个节点的输出可供多个设备使用，因此有多个 clk 实例。clk 可认为是节点输出到设备的时钟（可见时钟）。

●**flags**：标记成员，标记值定义在/include/linux/clk-provider.h 头文件：

```
#define   CLK_SET_RATE_GATE       BIT(0)   /* must be gated across rate change */
#define   CLK_SET_PARENT_GATE     BIT(1)   /* must be gated across re-parent */
#define   CLK_SET_RATE_PARENT     BIT(2)   /* propagate rate change up one level */
#define   CLK_IGNORE_UNUSED       BIT(3)   /* do not gate even if unused */
#define   CLK_IS_ROOT             BIT(4)   /*CCF 中根节点，没有父节点*/
#define   CLK_IS_BASIC            BIT(5)   /*基础时钟，如固定频率时钟源*/
#define   CLK_GET_RATE_NOCACHE          BIT(6) /* do not use the cached clk rate */
#define   CLK_SET_RATE_NO_REPARENT      BIT(7) /* don't re-parent on rate change */
#define   CLK_GET_ACCURACY_NOCACHE      BIT(8) /* do not use the cached clk accuracy */
```

```
#define    CLK_RECALC_NEW_RATES            BIT(9) /* recalc rates after notifications */
```

内核定义了以下两个全局双链表：

```
static HLIST_HEAD(clk_root_list);     /*链接根节点，可遍历树状结构中节点，clk_core->child_node*/
static HLIST_HEAD(clk_orphan_list);   /*链接所有孤儿节点，没有父点，clk_core->child_node*/
```

clk_root_list 双链表用于管理根节点，clk_orphan_list 双链表用于管理不是根节点且没有父节点的节点。

clk 结构体定义如下：

```
struct clk {
    struct clk_core*core;     /*指向 clk_core 结构体*/
    const char *dev_id;       /*使用时钟的设备名称，可以为 NULL*/
    const char *con_id;       /*标识时钟，查找时使用的关键字*/
    unsigned long min_rate;     /*最小频率*/
    unsigned long max_rate;     /*最大频率*/
    struct hlist_node clks_node;     /*散列链表节点，将实例添加到 clk_core 实例中链表*/
};
```

■clk_hw/clk_ops

clk_hw 和 clk_ops 是与硬件特性相关的数据结构。clk_hw 结构体用于标识不同类型的时钟模块（节点），clk_ops 表示时钟模块的操作函数集合。

clk_hw 结构体定义如下（/include/linux/clk-provider.h）：

```
struct clk_hw {
    struct clk_core *core;      /*指向 clk_core 实例*/
    struct clk *clk;            /*指向 clk 实例*/
    const struct clk_init_data *init;     /*指向 clk_init_data 结构体（注册节点时临时使用）*/
};
```

init 成员指向 clk_init_data 结构体，定义如下（/include/linux/clk-provider.h）：

```
struct clk_init_data {
    const char          *name;     /*节点名称*/
    const struct clk_ops*ops;      /*硬件节点操作结构*/
    const char * const *parent_names;      /*字符指针数组，指向父节点名称字符*/
    u8                  num_parents;       /*父节点数量，多路选择器可以有多个父节点*/
    unsigned long       flags;     /*标记成员，赋予 clk_core 结构体中 flags 成员*/
};
```

clk_hw 结构体通常嵌入到 clk_xxx 结构体中（第一个成员），clk_xxx 结构体表示真实的硬件模块。在注册节点的函数中需要将 clk_hw 实例指针作为参数。

clk_init_data 结构体是一个在注册 clk_core 实例过程中临时使用的数据结构，用来传递信息。

clk_init_data 结构体中 ops 成员指向的 **clk_ops** 结构体是一个非常重要的结构体，结构体中包含了硬件模块的操作函数指针集合。CCF 为每种类型的模块（节点）定义了对应的 clk_ops 实例。

clk_ops 结构体定义如下（/include/linux/clk-provider.h）：

```
struct clk_ops {
    int         (*prepare)(struct clk_hw *hw);      /*使能时钟前的准备工作*/
    void        (*unprepare)(struct clk_hw *hw);    /*使能之后的工作，通常为空*/
    int         (*is_prepared)(struct clk_hw *hw);  /*检查是否硬件模块是否准备好*/
    void        (*unprepare_unused)(struct clk_hw *hw);
    int         (*enable)(struct clk_hw *hw);     /*使能时钟，原子操作*/
    void        (*disable)(struct clk_hw *hw);    /*禁止时钟，原子操作*/
    int         (*is_enabled)(struct clk_hw *hw);    /*查询硬件模块是否使能*/
    void        (*disable_unused)(struct clk_hw *hw);
    unsigned long (*recalc_rate)(struct clk_hw *hw,unsigned long parent_rate);  /*重新计算时钟频率*/
    long        (*round_rate)(struct clk_hw *hw, unsigned long rate,unsigned long *parent_rate);
                                                    /*返回与 rate 最接近的，硬件支持的频率*/
    long        (*determine_rate)(struct clk_hw *hw,unsigned long rate,unsigned long min_rate,
                        unsigned long max_rate,unsigned long *best_parent_rate,
                        struct clk_hw **best_parent_hw);
                                                /*返回与 rate 最接近的，硬件支持的频率，*/
    int         (*set_parent)(struct clk_hw *hw, u8 index);       /*设置父节点*/
    u8          (*get_parent)(struct clk_hw *hw);                 /*获取父节点*/
    int         (**set_rate**)(struct clk_hw *hw, unsigned long rate,unsigned long parent_rate); /*设置频率*/
    int         (*set_rate_and_parent)(struct clk_hw *hw,unsigned long rate,
                                                    unsigned long parent_rate, u8 index);
    unsigned long (*recalc_accuracy)(struct clk_hw *hw,unsigned long parent_accuracy);
                                                                    /*重新计算精度*/
    int         (*get_phase)(struct clk_hw *hw);      /*获取相位*/
    int         (*set_phase)(struct clk_hw *hw, int degrees);      /*设置相位*/
    void        (*init)(struct clk_hw *hw);               /*初始化函数*/
    int         (*debug_init)(struct clk_hw *hw, struct dentry *dentry);
};
```

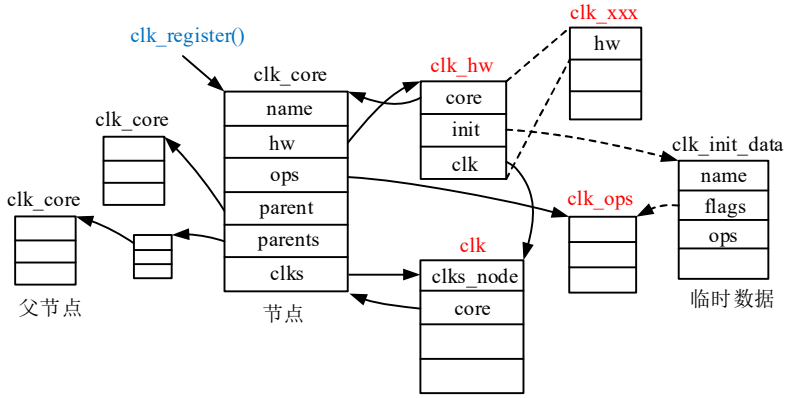
9.10.2 注册时钟部件

本小节主要介绍向 CCF 注册各类型部件（模块）的接口函数，平台（处理器）相关代码需要向 CCF 注册各部件。

1 注册节点

CCF 框架中每个节点由 clk_core 结构体表示，这是一个通用的结构体。不同类型的节点通过 clk_hw 结构体区分，clk_hw 结构体关联到通用节点 clk_core 结构体。clk_hw 结构体通常嵌入到 clk_xxx 结构体中，也就是说不同类型节点由 clk_xxx 结构体表示。

注册通用节点的函数 **clk_register()**执行结果如下图所示：



clk_register(dev,hw)函数定义如下（/drivers/clk/clk.c）：

```
struct clk *clk_register(struct device *dev, struct clk_hw *hw)
/*dev：代表节点的 device 实例，hw：指向 clk_hw 实例*/
{
    int i, ret;
    struct clk_core *core;

    core = kzalloc(sizeof(*core), GFP_KERNEL);    /*分配 clk_core 实例*/
    ...
    core->name = kstrdup_const(hw->init->name, GFP_KERNEL);    /*设置名称*/
    ...
    core->ops = hw->init->ops;          /*指向 clk_ops 实例*/
    if (dev && dev->driver)
        core->owner = dev->driver->owner;        /*模块指针*/
    core->hw = hw;                      /*指向 clk_hw 实例*/
    core->flags = hw->init->flags;          /*标记*/
    core->num_parents = hw->init->num_parents;          /*父节点数量*/
    hw->core = core;                    /*指向 clk_core*/

    /*分配字符指针数组，指向父节点名称字符串*/
    core->parent_names = kcalloc(core->num_parents, sizeof(char *),GFP_KERNEL);
    ...

    /* copy each string name in case parent_names is __initdata */
    for (i = 0; i < core->num_parents; i++) {
        core->parent_names[i] = kstrdup_const(hw->init->parent_names[i],GFP_KERNEL);
        ...
    }

    INIT_HLIST_HEAD(&core->clks);

    hw->clk = __clk_create_clk(hw, NULL, NULL);      /*创建 clk 实例，/drivers/clk/clk.c*/
    ...
```

```
    ret = __clk_init(dev, hw->clk);    /*初始化 clk_core 实例，插入树状结构等，/drivers/clk/clk.c*/
    if (!ret)
        return hw->clk;            /*返回 clk 实例指针*/
    ...
}
```

clk_register()函数将分配 clk_core 实例，并用 clk_hw 实例初始化 clk_core 实例，调用__clk_create_clk()函数创建 clk 实例，调用__clk_init()函数初始化 clk 实例关联的 clk_core 实例。下面看一下这两个函数的实现。

■创建 clk

__clk_create_clk()函数用于创建 clk 实例，函数定义如下：

```
struct clk *__clk_create_clk(struct clk_hw *hw, const char *dev_id,const char *con_id)
/*dev_id 和 con_id 这里都为 NULL*/
{
    struct clk *clk;

    if (!hw || IS_ERR(hw))
        return (struct clk *) hw;

    clk = kzalloc(sizeof(*clk), GFP_KERNEL);       /*分配 clk 实例*/
    if (!clk)
        return ERR_PTR(-ENOMEM);

    clk->core = hw->core;          /*指向 clk_core 实例*/
    clk->dev_id = dev_id;
    clk->con_id = con_id;
    clk->max_rate = ULONG_MAX;

    clk_prepare_lock();
    hlist_add_head(&clk->clks_node, &hw->core->clks);       /*添加到 clk_core 实例中的散列链表*/
    clk_prepare_unlock();

    return clk;            /*返回 clk 实例指针*/
}
```

■初始化 clk_core

__clk_init()函数用于初始化 clk 实例关联的 clk_core 实例，函数代码简列如下（/drivers/clk/clk.c）：

```
static int __clk_init(struct device *dev, struct clk *clk_user)
/*dev：代表节点的 device 实例，clk_user：指向 clk 实例*/
{
    int i, ret = 0;
```

```
struct clk_core *orphan;
struct hlist_node *tmp2;
struct clk_core *core;
unsigned long rate;
...

core = clk_user->core;      /*指向 clk_core 实例*/
clk_prepare_lock();
if (clk_core_lookup(core->name)) {     /*查找是否有同名的 clk_core 实例存在*/
      ...     /*错误处理*/
}

/*检查 clk_ops 实例*/
if (core->ops->set_rate &&!((core->ops->round_rate || core->ops->determine_rate) &&
          core->ops->recalc_rate)) {
      ...     /*错误处理*/
}
/*检查 clk_ops 实例，定义了 set_parent()函数就必须定义 get_parent()函数*/
if (core->ops->set_parent && !core->ops->get_parent) {
      ...     /*错误处理*/
}
/*检查 clk_ops 实例*/
if (core->ops->set_rate_and_parent &&!(core->ops->set_parent && core->ops->set_rate)) {
      ...     /*错误处理*/
}

for (i = 0; i < core->num_parents; i++)     /*父节点名称不能为空（告警）*/
      ...     /*错误处理*/

if (core->num_parents > 1 && !core->parents) {     /*查找并关联父节点 clk_core 实例*/
      core->parents = kcalloc(core->num_parents, sizeof(struct clk *),GFP_KERNEL);
                                          /*分配指针数组，不应该是指向 clk_core 的指针吗？ */
      if (core->parents)
            for (i = 0; i < core->num_parents; i++)
                  core->parents[i] =clk_core_lookup(core->parent_names[i]);     /*查找父 clk_core 实例*/
}

core->parent = __clk_init_parent(core);      /*查找父节点，根节点没有父节点*/
                              /*如果有多个父点，调用 core->ops->get_parent()查找父节点*/

if (core->parent)          /*将 clk_core 实例添加到管理结构*/
      hlist_add_head(&core->child_node,&core->parent->children); /*加入父节点的子节点链表*/
else if (core->flags & CLK_IS_ROOT)
```

```
    hlist_add_head(&core->child_node, &clk_root_list);     /*根节点添加到 clk_root_list 链表*/
else
    hlist_add_head(&core->child_node, &clk_orphan_list);
                                        /*没有父节点且不是根节点，添加到 clk_orphan_list 链表*/
if (core->ops->recalc_accuracy)        /*设置节点时钟精度*/
    core->accuracy = core->ops->recalc_accuracy(core->hw,__clk_get_accuracy(core->parent));
else if (core->parent)
    core->accuracy = core->parent->accuracy;
else
    core->accuracy = 0;

if (core->ops->get_phase)
    core->phase = core->ops->get_phase(core->hw);
else
    core->phase = 0;

if (core->ops->recalc_rate)          /*节点输出时钟频率值*/
    rate = core->ops->recalc_rate(core->hw,clk_core_get_rate_nolock(core->parent));
else if (core->parent)
    rate = core->parent->rate;
else
    rate = 0;
core->rate = core->req_rate = rate;       /*设置节点输出时钟频率值*/

/*查找孤儿 clk_core 链表，如果新 clk_core 实例是孤儿节点的父节点，则设置其父节点*/
hlist_for_each_entry_safe(orphan, tmp2, &clk_orphan_list, child_node) {
    if (orphan->num_parents && orphan->ops->get_parent) {
        i = orphan->ops->get_parent(orphan->hw);
        if (!strcmp(core->name, orphan->parent_names[i]))
            clk_core_reparent(orphan, core);
        continue;
    }

    for (i = 0; i < orphan->num_parents; i++)
        if (!strcmp(core->name, orphan->parent_names[i])) {
            clk_core_reparent(orphan, core);
            break;
        }
 }

 if (core->ops->init)
    core->ops->init(core->hw);     /*调用 clk_ops 实例中 init()函数*/
```

```
        kref_init(&core->ref);
    out:
        clk_prepare_unlock();
        if (!ret)
            clk_debug_register(core);
        return ret;
}
```

__clk_init()函数主要是对 clk_core 实例进行初始化，并将其添加到管理结构中，函数比较简单，就不再解释了。

2 注册固定频率时钟源

前面介绍的 clk_register(dev,hw)函数是用于注册通用的节点 clk_core 实例，而 CCF 中不同类型的部件由 clk_hw 结构体表示，clk_hw 结构体通常嵌入到 clk_xxx 结构体中。clk_xxx 结构体才是真正表示不同硬件部件的结构体。

在注册硬件部件的函数中，需要创建并初始化 clk_xxx 实例，然后调用 clk_register(dev,hw)函数注册节点，hw 参数指向 clk_xxx 实例中内嵌的 clk_hw 结构体成员。

固定频率时钟源通常是振荡器模块，提供固定的基准时钟。固定频率时钟源通常是 CCF 中的根节点。固定频率时钟源由 clk_fixed_rate 结构体表示，定义如下（/include/linux/clk-provider.h）：

```
struct clk_fixed_rate {
    struct    clk_hw    hw;      /*内嵌 clk_hw 结构体*/
    unsigned long fixed_rate;    /*时钟频率*/
    unsigned long fixed_accuracy;        /*精度*/
    u8        flags;     /*标记*/
};
```

clk_fixed_rate 实例 hw 成员关联的 clk_ops 实例（时钟操作结构）定义如下：

```
const struct clk_ops clk_fixed_rate_ops = {          /*/drivers/clk/clk-fixed-rate.c*/
    .recalc_rate = clk_fixed_rate_recalc_rate,           /*返回 clk_fixed_rate.fixed_rate*/
    .recalc_accuracy = clk_fixed_rate_recalc_accuracy,   /*返回 clk_fixed_rate.fixed_accuracy*/
};
```

注册固定频率时钟源的 clk_register_fixed_rate()函数定义如下（/drivers/clk/clk-fixed-rate.c）：

```
struct clk *clk_register_fixed_rate(struct device *dev, const char *name,
            const char *parent_name, unsigned long flags,unsigned long fixed_rate)
/*name：时钟源名称，parent_name：父节点名称，flags：标记，fixed_rate：时钟频率*/
{
    return clk_register_fixed_rate_with_accuracy(dev, name, parent_name,flags, fixed_rate, 0);
}
```

clk_register_fixed_rate_with_accuracy()函数定义如下：

```
struct clk *clk_register_fixed_rate_with_accuracy(struct device *dev,
        const char *name, const char *parent_name, unsigned long flags,
        unsigned long fixed_rate, unsigned long fixed_accuracy)
```

```
/*fixed_accuracy：0*/
{
    struct clk_fixed_rate *fixed;
    struct clk *clk;
    struct clk_init_data init;    /*clk_init_data 实例，临时数据，注册完就释放了*/

    fixed = kzalloc(sizeof(*fixed), GFP_KERNEL);      /*分配 clk_fixed_rate 实例*/
    ...
    init.name = name;
    init.ops = &clk_fixed_rate_ops;          /*clk_ops 实例*/
    init.flags = flags | CLK_IS_BASIC;
    init.parent_names = (parent_name ? &parent_name: NULL);
    init.num_parents = (parent_name ? 1 : 0);        /*没有父节点，或只有一个父节点*/

    /*设置 clk_fixed_rate 实例成员*/
    fixed->fixed_rate = fixed_rate;
    fixed->fixed_accuracy = fixed_accuracy;
    fixed->hw.init = &init;        /*指向 clk_init_data 实例*/

    clk = clk_register(dev, &fixed->hw);    /*注册节点*/
    ...
    return clk;
}
```

注册固定频率时钟源可能有一个或没有父节点，clk_core 实例设置了 CLK_IS_BASIC 标记位。

3 注册分频器

CCF 中分频器由 clk_divider 结构体表示，定义如下（/include/linux/clk-provider.h）：

```
struct clk_divider {
    struct clk_hw hw;    /*内嵌 clk_hw 实例*/
    void __iomem *reg;   /*分频器寄存器地址*/
    u8          shift;       /*读寄存器时需右移位数*/
    u8          width;       /*掩码*/
    u8          flags;    /*标记，/include/linux/clk-provider.h*/
    const struct clk_div_table     *table;    /*指向 clk_div_table 数组，/include/linux/clk-provider.h*/
    spinlock_t      *lock;
};
```

clk_divider 结构体部分成员简介如下：

●**reg**：分频器寄存器地址，

●**flags**：标记，取值如下：

```
#define   CLK_DIVIDER_ONE_BASED          BIT(0)    /*置位表示读取的原始时钟频率*/
#define   CLK_DIVIDER_POWER_OF_TWO   BIT(1)   /*置位表示除数是 2^hwr，hwr 是硬件寄存器值*/
#define   CLK_DIVIDER_ALLOW_ZERO       BIT(2)   /*若 bit0 置 1，且本位置 1，表示除数允许为 0*/
```

```
#define   CLK_DIVIDER_HIWORD_MASK    BIT(3)   /*硬件寄存器低 16 位有效，高 16 位为掩码*/
#define   CLK_DIVIDER_ROUND_CLOSEST  BIT(4)
#define   CLK_DIVIDER_READ_ONLY      BIT(5)   /*只读，分频率不可配置*/
```

●**table**：指向 clk_div_table 结构体数组，表示分频表，结构体定义如下：

```
struct clk_div_table {
    unsigned int    val;
    unsigned int    div;
};
```

CCF 为分频器定义的 clk_ops 实例如下（/drivers/clk/clk-divider.c）：

```
const struct clk_ops clk_divider_ops = {
    .recalc_rate = clk_divider_recalc_rate,
    .round_rate = clk_divider_round_rate,
    .set_rate = clk_divider_set_rate,        /*设置时钟频率*/
};
```

注册分频器的接口函数 clk_register_divider()定义如下（/drivers/clk/clk-divider.c）：

```
struct clk *clk_register_divider(struct device *dev, const char *name,
    const char *parent_name, unsigned long flags,
    void __iomem *reg, u8 shift, u8 width,u8 clk_divider_flags, spinlock_t *lock)
{
    return _register_divider(dev, name, parent_name, flags, reg, shift,
                                        width, clk_divider_flags, NULL, lock);
}
```

_register_divider()函数定义如下：

```
static struct clk *_register_divider(struct device *dev, const char *name,
    const char *parent_name, unsigned long flags,void __iomem *reg, u8 shift, u8 width,
    u8 clk_divider_flags, const struct clk_div_table *table,spinlock_t *lock)
/*table：这里为 NULL*/
{
    struct clk_divider *div;
    struct clk *clk;
    struct clk_init_data init;

    if (clk_divider_flags & CLK_DIVIDER_HIWORD_MASK) {
        if (width + shift > 16) {
            ...        /*错误处理*/
        }
    }

    div = kzalloc(sizeof(*div), GFP_KERNEL);     /*分配 clk_divider 实例*/
```

```
    ...        /*错误处理*/
    init.name = name;
    init.ops = &clk_divider_ops;        /*clk_ops 实例*/
    init.flags = flags | CLK_IS_BASIC;
    init.parent_names = (parent_name ? &parent_name: NULL);
    init.num_parents = (parent_name ? 1 : 0);

    div->reg = reg;
    div->shift = shift;
    div->width = width;
    div->flags = clk_divider_flags;
    div->lock = lock;
    div->hw.init = &init;
    div->table = table;

    clk = clk_register(dev, &div->hw);     /*注册节点*/
    ...
    return clk;       /*返回 clk 实例指针*/
}
```

4 注册多路选择器

CCF 中多路选择器由 clk_mux 结构体表示，定义如下（/include/linux/clk-provider.h）：

```
struct clk_mux {
    struct clk_hw hw;
    void __iomem *reg;        /*硬件寄存器地址*/
    u32        *table;
    u32        mask;
    u8         shift;
    u8         flags;     /*标记成员，/include/linux/clk-provider.h*/
    spinlock_t      *lock;
};
```

clk_mux 结构体与 clk_divider 结构体类似，标记成员 flags 取值定义如下：

```
#define   CLK_MUX_INDEX_ONE          BIT(0)   /*置位表示寄存器索引值从 1 开始，不是 0*/
#define   CLK_MUX_INDEX_BIT          BIT(1)   /*寄存器索引值是 2 的幂*/
#define   CLK_MUX_HIWORD_MASK    BIT(2) /*置位表示寄存器低 16 位可设置，高 16 位为掩码*/
#define   CLK_MUX_READ_ONLY          BIT(3)       /*只读不可更改*/
#define   CLK_MUX_ROUND_CLOSEST      BIT(4)
```

CCF 为多路选择器定义的 clk_ops 实例如下（/drivers/clk/clk-mux.c）：

```
const struct clk_ops clk_mux_ops = {
    .get_parent = clk_mux_get_parent,
    .set_parent = clk_mux_set_parent,
```

```
    .determine_rate = __clk_mux_determine_rate,
};

const struct clk_ops clk_mux_ro_ops = {        /*适用于只读多路选择器*/
    .get_parent = clk_mux_get_parent,
};
```

注册多路选择器的接口函数为 **clk_register_mux()**，定义如下（/drivers/clk/clk-mux.c）：

```
struct clk *clk_register_mux(struct device *dev, const char *name,
        const char * const *parent_names, u8 num_parents,unsigned long flags,
        void __iomem *reg, u8 shift, u8 width,u8 clk_mux_flags, spinlock_t *lock)
/*parent_names：指向父节点名称字符指针数组，num_parents：父节点数量，clk_mux 有多个父节点*/
{
    u32 mask = BIT(width) - 1;
    return clk_register_mux_table(dev, name, parent_names, num_parents,
                                  flags, reg, shift, mask, clk_mux_flags,NULL, lock);
}
```

clk_register_mux_table()函数定义如下：

```
struct clk *clk_register_mux_table(struct device *dev, const char *name,
        const char * const *parent_names, u8 num_parents,
        unsigned long flags,void __iomem *reg, u8 shift, u32 mask,
        u8 clk_mux_flags, u32 *table, spinlock_t *lock)
/*talbe：这里为 NULL*/
{
    struct clk_mux *mux;
    struct clk *clk;
    struct clk_init_data init;
    u8 width = 0;

    if (clk_mux_flags & CLK_MUX_HIWORD_MASK) {
        width = fls(mask) - ffs(mask) + 1;
        if (width + shift > 16) {
            pr_err("mux value exceeds LOWORD field\n");
            return ERR_PTR(-EINVAL);
        }
    }

    mux = kzalloc(sizeof(struct clk_mux), GFP_KERNEL);    /*分配 clk_mux 实例*/
    ...
    init.name = name;
    if (clk_mux_flags & CLK_MUX_READ_ONLY)
        init.ops = &clk_mux_ro_ops;      /*clk_ops 实例*/
```

```
        else
                init.ops = &clk_mux_ops;
        init.flags = flags | CLK_IS_BASIC;
        init.parent_names = parent_names;
        init.num_parents = num_parents;

        mux->reg = reg;
        mux->shift = shift;
        mux->mask = mask;
        mux->flags = clk_mux_flags;
        mux->lock = lock;
        mux->table = table;
        mux->hw.init = &init;

        clk = clk_register(dev, &mux->hw);      /*注册节点*/
        ...
        return clk;
}
```

5 注册固定倍频/分频器

CLOCK 模块中有的部件是对输入时钟进行固定的倍频或分频，这类部件由 clk_fixed_factor 结构体表示，定义如下（/include/linux/clk-provider.h）：

```
struct clk_fixed_factor {
        struct clk_hw hw;       /*内嵌 clk_hw 结构体*/
        unsigned int    mult;   /*倍频系数*/
        unsigned int    div;    /*分频系数*/
};
```

固定倍频/分频器的输出频率=输入频繁（父节点频率）/div*mult。

CCF 为固定倍频/分频器定义的 clk_ops 实例如下（/drivers/clk/clk-fixed-factor.c）：

```
const struct clk_ops clk_fixed_factor_ops = {
        .round_rate = clk_factor_round_rate,
        .set_rate = clk_factor_set_rate,
        .recalc_rate = clk_factor_recalc_rate,
};
```

注册固定倍频/分频器的 clk_register_fixed_factor()函数定义如下（/drivers/clk/clk-fixed-factor.c）：

```
struct clk *clk_register_fixed_factor(struct device *dev, const char *name,
                const char *parent_name, unsigned long flags,unsigned int mult, unsigned int div)
{
        struct clk_fixed_factor *fix;
        struct clk_init_data init;
        struct clk *clk;
```

```
    fix = kmalloc(sizeof(*fix), GFP_KERNEL);      /*创建 clk_fixed_factor 实例*/
    ...

    fix->mult = mult;
    fix->div = div;
    fix->hw.init = &init;

    init.name = name;
    init.ops = &clk_fixed_factor_ops;     /*clk_ops 实例*/
    init.flags = flags | CLK_IS_BASIC;
    init.parent_names = &parent_name;
    init.num_parents = 1;

    clk = clk_register(dev, &fix->hw);     /*注册节点*/
    ...
    return clk;
}
```

CCF 中还有其它的部件，如控制门等，相关的数据结构和注册函数与前面介绍的类似，不再一一列举了，相关代码位于/drivers/clk/clk-xxx.c 文件内。

9.10.3 时钟接口

前面介绍的 CCF 实现了时钟部件和时钟的管理，那么使用时钟的设备如何从 CCF 中获取、设置时钟呢？

CCF 中需要导出（对内核其它部分可见）的时钟 clk，需要为其注册 clk_lookup 结构体实例。导出后内核其它部分通过查找 clk_lookup 实例，获取时钟所在的节点，并创建/添加 clk 实例，然后就可以通过此 clk 实例对时钟进行操作（对节点进行操作）。也就是说 clk_lookup 是查找/获取 clk 的入口，只有注册了 clk_lookup 的节点才可以被内核其它部分访问。

一个节点输出时钟可供多个设备使用，可为每个设备设置一个查找入口，即 clk_lookup 实例。设备驱动在依 clk_lookup 实例查找 clk 实例时，会为每个 clk_lookup 实例创建新的 clk 实例，通过此实例操作节点时钟。

clk_lookup 结构体定义如下（/include/linux/clkdev.h）：

```
struct clk_lookup {
    struct list_head    node;      /*将实例添加到全局双链表 clocks*/
    const char          *dev_id;    /*使用时钟的设备名称*/
    const char          *con_id;    /*用于查找 clk_core 实例*/
    struct clk      *clk;           /*指向 clk 实例，在 clk_get()函数中新建*/
    struct clk_hw       *clk_hw;   /*指向 clk_hw 实例*/
};
```

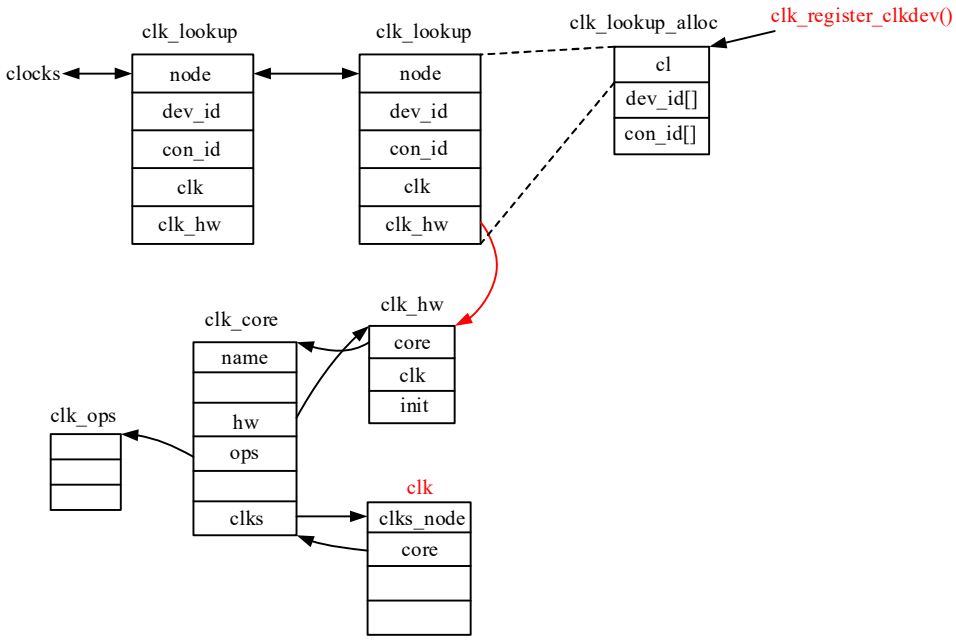
内核在/drivers/clk/clkdev.c 文件内定义了全局双链表 clocks 用于管理 clk_lookup 实例：

```
static LIST_HEAD(clocks);
```

CCF 中间部件（节点），如 PLL、分频器等，如果要设置成可访问的，也需要注册 clk_lookup 实例，以便对其进行操作。

1 导出输出时钟

对于节点输出的时钟，需要为其注册 clk_lookup 实例，才能使设备驱动程序查找到其 clk 实例，然后通过 clk 实例对节点时钟进行操作。为节点注册的 clk_lookup 实例如下图所示：



注册函数为 clk_register_clkdev(struct clk ***clk**, const char ***con_id**,const char *dev_fmt, ...)，clk 参数指向 clk 实例（以此找到 clk_core 实例），**con_id** 为查找 clk_lookup 实例时使用的字符串，dev_fmt 为设备名称。

clk_register_clkdev()函数内将分配 clk_lookup_alloc 结构体实例，其第一个成员为 clk_lookup 结构体，第二个成员保存 dev_fmt 字符（最多 20 个字符），第三个成员保存 con_id 字符（最多 16 个字符）。

clk_lookup 结构体实例将添加到 clocks 双链表末尾，其 dev_id 和 con_id 成员指向字符串复制于 dev_fmt 和 con_id 参数，clk_hw 成员指向 clk->core->hw（clk_hw 实例）。注意，此时 clk_lookup 实例 clk 成员还没有指向 clk 实例。

clk_register_clkdev()函数定义如下（/drivers/clk/clkdev.c）：

```
int clk_register_clkdev(struct clk *clk, const char *con_id,const char *dev_fmt, ...)
{
    struct clk_lookup *cl;
    va_list ap;
    ...
    va_start(ap, dev_fmt);
    cl = vclkdev_create(__clk_get_hw(clk), con_id, dev_fmt, ap);   /*创建并添加 clk_lookup 实例*/
    va_end(ap);

    return cl ? 0 : -ENOMEM;          /*成功返回 0*/
}
```

vclkdev_create()函数定义如下：

```
static struct clk_lookup *vclkdev_create(struct clk_hw *hw, const char *con_id, const char *dev_fmt,
                                         va_list ap)
/*hw：clk->core->hw*/
{
    struct clk_lookup *cl;

    cl = vclkdev_alloc(hw, con_id, dev_fmt, ap);   /*创建 clk_lookup 实例（clk_lookup_alloc 实例）*/
    if (cl)
        __clkdev_add(cl);     /*将 clk_lookup 实例添加到 clocks 双链表末尾*/

    return cl;          /*返回 clk_lookup 实例指针*/
}
```

2 获取时钟

导出节点输出时钟后，使用时钟的设备驱动程序可通过 clk_lookup.**con_id** 字符串，在 clocks 双链表中查找到 clk_lookup 实例，进而创建/添加新的 clk 实例，然后就可以通过此 clk 实例对节点进行操作。

clk_get()函数用于查找 clk_lookup 实例，并创建/添加 clk 实例，定义如下（/drivers/clk/clkdev.c）：

```
struct clk *clk_get(struct device *dev, const char *con_id)
/*dev：表示设备的 device 实例，可以为 NULL，con_id：用于查找 clk_lookup 实例*/
{
    const char *dev_id = dev ? dev_name(dev) : NULL;       /*设备名称*/
    struct clk *clk;

    if (dev) {
        clk = __of_clk_get_by_name(dev->of_node, dev_id, con_id);     /*需要设备树支持*/
        if (!IS_ERR(clk) || PTR_ERR(clk) == -EPROBE_DEFER)
            return clk;
    }
    return clk_get_sys(dev_id, con_id);       /*创建 clk 实例，返回实例指针*/
}
```

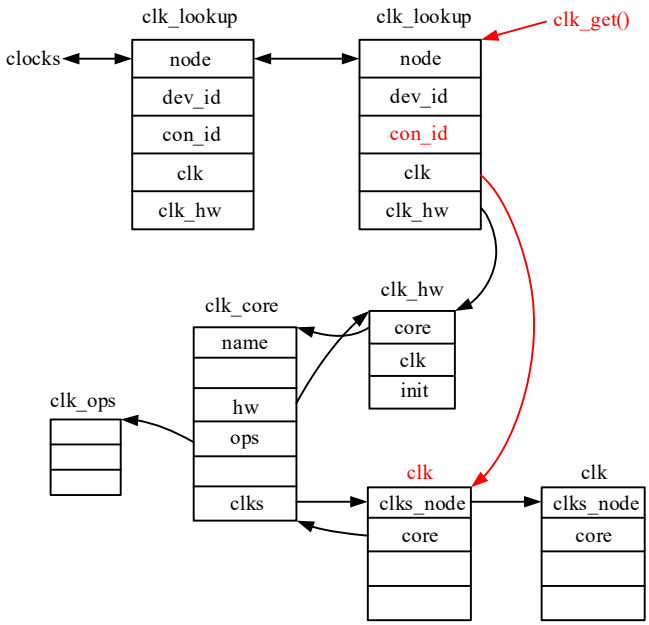
clk_get_sys()函数定义如下：

```
struct clk *clk_get_sys(const char *dev_id, const char *con_id)
{
    struct clk_lookup *cl;
    struct clk *clk = NULL;

    mutex_lock(&clocks_mutex);

    cl = clk_find(dev_id, con_id);     /*在 clocks 双链表中查找 clk_lookup 实例*/
    ...
    clk = __clk_create_clk(cl->clk_hw, dev_id, con_id);       /*创建并初始化 clk 实例，/drivers/clk/clk.c*/
    ...
```

```
        if (!__clk_get(clk)) {      /*增加 clk_core 引用计数，/drivers/clk/clk.c*/
            ...
        }
    out:
        mutex_unlock(&clocks_mutex);

        return cl ? clk : ERR_PTR(-ENOENT);
    }
```

clk_get_sys()函数内调用__clk_create_clk()函数创建并设置 clk 实例（已经存在就不创建了），添加到 clk_core 实例中散列链表头部，调用__clk_get()函数增加 clk_core 实例的引用计数，源代码请读者自行阅读。

clk_get()函数执行结果如下图所示，函数返回新创建的 clk 实例指针：



3 其它操作接口

设备驱动程序在调用 clk_get()函数获取 clk 实例后，就可以调用其它接口函数，以 clk 实例为参数，对节点时钟进行操作了，例如：

●int **clk_get_rate(struct clk *clk)**：获取当前时钟频率，频率值保存在 clk 实例关联 clk_core 实例的 **rate** 成员中，函数定义在/drivers/clk/clk.c 文件内，源代码请读者自行阅读。

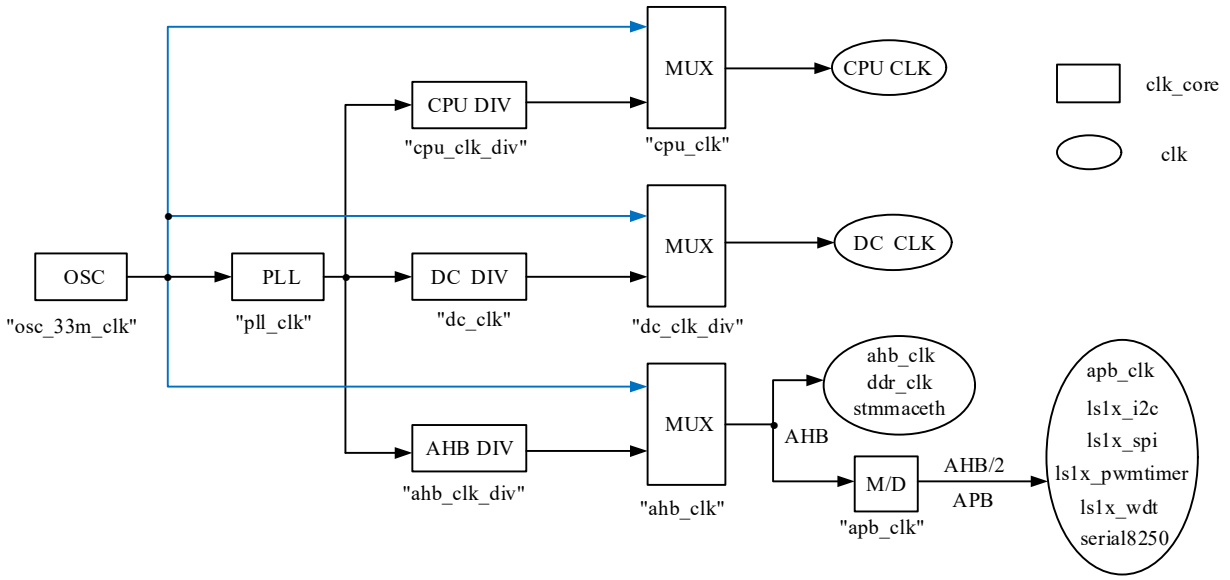
●int **clk_set_rate**(struct clk *clk, unsigned long rate)：设置时钟频率，成功返回 0，否则返回错误码（负值），函数定义在/drivers/clk/clk.c 文件内。

●int **clk_enable**(struct clk *clk)：使能时钟（输出），成功返回 0，否则返回错误码（负值），函数定义在/drivers/clk/clk.c 文件内。

●void **clk_disable**(struct clk *clk)：禁止时钟（不输出），函数定义在/drivers/clk/clk.c 文件内。

9.10.4 龙芯 1B 时钟示例

在本节的开头列出了龙芯 1B 的时钟模块框图，下图示意了依硬件时钟模块结构实现的 CCF 框架图，图中方框表示节点，椭圆形表示时钟或者说使用节点输出时钟的设备。



上图中 CPU CLK、DC CLK（显示控制）和 AHB CLK 来自 OSC 或 PLL 输出的分频，DDR 使用的是 AHB CLK，APB CLK 固定为 AHB CLK 的一半（AHB 和 APB 为芯片内部总线名称）。

龙芯 1B 处理器 CCF 实现代码位于/drivers/clk/clk-ls1x.c 文件内，主要代码简列如下：

```
static const char * const cpu_parents[] = { "cpu_clk_div", "osc_33m_clk", };     /*CPU MUX 的父节点*/
static const char * const ahb_parents[] = { "ahb_clk_div", "osc_33m_clk", };     /*AHB MUX 的父节点*/
static const char * const dc_parents[] = { "dc_clk_div", "osc_33m_clk", };       /*DC MUX 的父节点*/
```

在初始化函数 ls1x_clk_init()中将建立 CCF 实例，函数代码简列如下：

```
void __init ls1x_clk_init(void)        /*由 plat_time_init()函数调用*/
{
    struct clk *clk;

    clk = clk_register_fixed_rate(NULL, "osc_33m_clk", NULL, CLK_IS_ROOT,OSC);
                                                        /*注册固定频率时钟源，根节点*/
    clk_register_clkdev(clk, "osc_33m_clk", NULL);     /*导出固定时钟输出*/

    /*CCF 没有实现注册 PLL 的函数，clk_register_pll()函数由平台实现*/
    clk = clk_register_pll(NULL, "pll_clk", "osc_33m_clk", 0);  /*注册 PLL 节点，由处理器代码实现*/
    clk_register_clkdev(clk, "pll_clk", NULL);       /*导出 PLL 时钟输出，以便设置时钟*/


    /*注册 CPU DIV 节点，父节点为 PLL 节点*/
    clk = clk_register_divider(NULL, "cpu_clk_div", "pll_clk",
                    CLK_GET_RATE_NOCACHE, LS1X_CLK_PLL_DIV,
                    DIV_CPU_SHIFT, DIV_CPU_WIDTH,
                    CLK_DIVIDER_ONE_BASED |
                    CLK_DIVIDER_ROUND_CLOSEST, &_lock);     /*注册分频器*/
    clk_register_clkdev(clk, "cpu_clk_div", NULL);       /*导出 CPU DIV 时钟输出*/
    clk = clk_register_mux(NULL, "cpu_clk", cpu_parents,
```

```
                    ARRAY_SIZE(cpu_parents),          /*父节点*/
                    CLK_SET_RATE_NO_REPARENT, LS1X_CLK_PLL_DIV,
                    BYPASS_CPU_SHIFT, BYPASS_CPU_WIDTH, 0, &_lock);
                                                      /*注册 CPU CLK 多路选择器*/
    clk_register_clkdev(clk, "cpu_clk", NULL);    /*导出 CPU CLK，就是 CPU MUX 的输出*/

    /*注册 DC DIV 和 DC MUX*/
    clk = clk_register_divider(NULL, "dc_clk_div", "pll_clk",
                    0, LS1X_CLK_PLL_DIV, DIV_DC_SHIFT,
                    DIV_DC_WIDTH, CLK_DIVIDER_ONE_BASED, &_lock);
    clk_register_clkdev(clk, "dc_clk_div", NULL);
    clk = clk_register_mux(NULL, "dc_clk", dc_parents,
                    ARRAY_SIZE(dc_parents),           /*父节点*/
                    CLK_SET_RATE_NO_REPARENT, LS1X_CLK_PLL_DIV,
                    BYPASS_DC_SHIFT, BYPASS_DC_WIDTH, 0, &_lock);
    clk_register_clkdev(clk, "dc_clk", NULL);

    /*注册 AHB DIV 和 AHB MUX*/
    clk = clk_register_divider(NULL, "ahb_clk_div", "pll_clk",
                    0, LS1X_CLK_PLL_DIV, DIV_DDR_SHIFT,
                    DIV_DDR_WIDTH, CLK_DIVIDER_ONE_BASED,
                    &_lock);
    clk_register_clkdev(clk, "ahb_clk_div", NULL);
    clk = clk_register_mux(NULL, "ahb_clk", ahb_parents,
                    ARRAY_SIZE(ahb_parents),          /*父节点*/
                    CLK_SET_RATE_NO_REPARENT, LS1X_CLK_PLL_DIV,
                    BYPASS_DDR_SHIFT, BYPASS_DDR_WIDTH, 0, &_lock);
    clk_register_clkdev(clk, "ahb_clk", NULL);      /*导出 AHB MUX 时钟输出*/
    clk_register_clkdev(clk, "stmmaceth", NULL);

    /*注册固定倍频/分频器，父节点为 AHB MUX，输入频率减半，输出为 APB CLK*/
    clk = clk_register_fixed_factor(NULL, "apb_clk", "ahb_clk", 0, 1,DIV_APB);
    /*导出 APB CLK 时钟（使用 APB CLK 的设备）*/
    clk_register_clkdev(clk, "apb_clk", NULL);
    clk_register_clkdev(clk, "ls1x_i2c", NULL);
    clk_register_clkdev(clk, "ls1x_pwmtimer", NULL);
    clk_register_clkdev(clk, "ls1x_spi", NULL);
    clk_register_clkdev(clk, "ls1x_wdt", NULL);
    clk_register_clkdev(clk, "serial8250", NULL);
}
```

在设备驱动程序中，可通过 clk_get(struct device *dev, const char ***con_id**)函数获取设备时钟对应的 clk 实例，然后以 clk 实例为参数对时钟进行操作，如获取时钟频率、设置时钟频率等。

9.11 小结

字符设备杂而多，内核在通用字符设备驱动框架下为各类型字符设备实现了各自的私有框架，具体设备驱动只需要实现私有框架中与底层硬件操作相关的部分即可。

本章介绍了通用字符设备驱动框架，重点介绍了几类常用字符设备驱动框架的实现，例如：GPIO、MISC 设备、RTC 设备、输入设备、帧缓存设备、终端设备等，最后介绍了硬件时钟 CCF 的实现。

由于作者时间、能力有限，只能介绍各类型设备驱动的框架，对具体设备细节研究不深，在驱动程序移植时读者可参考处理器手册和内核源代码。如果内核中没有所需设备的驱动程序，可在内核源文件中查找一个类似或相同设备的驱动程序进行修改移植。