# 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。



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\_i**r", /\*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;

}

returnplatform\_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 devic**e 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：



●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()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2hwr，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 \***tabl**e, 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。