cdev、misc以及device三者之间的联系和区别

原创

leochen_career 于 2017-11-15 15:03:02 发布 5568 收藏 37
分类专栏： 嵌入式开发 linux 文章标签： 内核 驱动 linux驱动模型

本文链接：https://blog.csdn.net/leochen_career/article/details/78540916


嵌入式开发 同时被 2 个专栏收录
13 篇文章 0 订阅
订阅专栏


linux
7 篇文章 0 订阅
订阅专栏

1.从/dev目录说起

从事Linux嵌入式驱动开发的人，都很熟悉下面的一些基础知识
比如对于一个char类型的设备，我想对其进行read wirte 和ioctl操作，那么我们通常会在内核驱动中实现一个file_operations结构体，然后分配主次设备号，调用cdev_add函数进行注册。从/proc/devices下面找到注册的设备的主次设备号，在用mknod /dev/char_dev c major minor 命令行创建设备节点。在用户空间open /dev/char_dev这个设备，然后进行各种操作。
OK，字符设备模型就这么简单，很多ABC教程都是一个类似的实现。
然后我们去看内核代码时，突然一脸懵逼。。。怎么内核代码里很多常用的驱动的实现不是这个样子的？没看到有file_operations结构体，我怎么使用这些驱动？看到了/dev目录下有需要的char设备，可是怎么使用呢？我还是习惯用/dev的形式，那我该怎么办？

2.Linux驱动模型的一个重要分界线

linux2.6版本以前，普遍的用法就像我上面说的一样。但是linux2.6版本以后，引用了Linux设备驱动模型，开始使用了基于sysfs的文件系统，出现让我们不是太明白的那些Linux内核驱动了。
Linux驱动模型的相关知识，建议看下 http://www.wowotech.net/device_model/13.html
也就是说，我们熟悉的那套驱动模式是2.6版本以前的（当然这是基础，肯定要会的）
我们不熟悉的驱动模型是2.6版本以后的。

3. cdev、misc以及device

cdev和device的区别和联系

那些说cdev是device派生出来的一个子类的，应该没看过内核代码这两个结构体的实现。。。

```
struct cdev {
    struct kobject kobj;
    struct module *owner;
    const struct file_operations *ops;
    struct list_head list;
    dev_t dev;
    unsigned int count;
};

struct device {
    struct device     *parent;
    struct device_private  *p;
    struct kobject kobj;
    const char     *init_name;  /* initial name of the device */
    struct device_driver *driver; /* which driver has allocated this device */
    void        * driver_data ;   /* Driver data, set and get with dev_set/get_drvdata */
    dev_t           devt;   /* dev_t, creates the sysfs "dev" */
    u32        id; /* device instance */
    void    (*release)( struct device *dev);
    ......
};
```

通过看这两个结构体发现，其实cdev和device之间没啥直接的联系，只有一个 dev_t 和kobject 是相同的。 dev_t 这个是联系两者的一个纽带了 。通常可以这么理解： cdev是传统驱动的设备核心数据结构，device是linux设备驱动模型中的核心数据结构。
要注册一个device设备，需要调用核心函数device_register()（或者说是device_add()函数） ； 要注册一个cdev设备，需要调用核心函数register_chrdev()(或者说是cdev_add()函数)

miscdevice和device的区别和联系

```
struct miscdevice {
    int minor;
    const char *name;
    const struct file_operations *fops;
    struct list_head list;
    struct device *parent;
    struct device *this_device;
    const struct attribute_group **groups;
    const char *nodename;
    umode_t mode;
};
```

从定义可以看出，miscdevice是 device的子类，是从device派生出来的结构体，也是属于device范畴的，也就是该类设备会统一在/sys目录下进行管理了。

miscdevice和cdev的区别和联系
通过上面的数据结构可以看到，两者都有一个file_operations和 dev_t（misdevice由于主设备号固定，所以结构体里只有一个minor）。

从数据结构上看，miscdevice是device和cdev的结合。

4. device_register()---理解上面三者关系的关键函数
device_register()--->device_add(),然后会调用两个关键函数
device_create_file() ----将相关信息添加到/sys文件系统中
devtmpfs_create_node() ----将相关信息添加到/devtmpfs文件系统中
第一个调用不做详细解析，因为devices本来就是/sys文件系统中的重要概念
关键是devtmpfs_create_node()
先了解下Devtmpfs 的概念
Devtmpfs lets the kernel create a tmpfs very early at kernel initialization , before any driver core device is registered . Every device with a major / minor will have a device node created in this tmpfs instance . After the rootfs is mounted by the kernel , the populated tmpfs is mounted at / dev .

devtmpfs_create_node()函数的核心是调用了内核的 vfs_mknod()函数，这样就在devtmpfs系统中创建了一个设备节点，当devtmpfs被内核mount到/dev目录下时，该设备节点就存在于/dev目录下，比如/dev/char_dev之类的。
vfs_mknod()函数中会调用一个init_special_inode(),该函数实现如下：
如果node是一个char设备，会给i_fop 赋值一个默认的def_chr_fops，也就是说对该node节点，有一个默认的操作。在open一个字符设备文件时，最终总会调用chrdev_open，然后调用各个char设备自己的file_operations 定义的open函数。

```
void init_special_inode( struct inode *inode, umode_t mode, dev_t rdev)
{
    inode->i_mode = mode;
    if (S_ISCHR(mode)) {
        inode->i_fop = &def_chr_fops;
        inode->i_rdev = rdev;
    } else if (S_ISBLK(mode)) {
        inode->i_fop = &def_blk_fops;
        inode->i_rdev = rdev;
    } else if (S_ISFIFO(mode))
        inode->i_fop = &pipefifo_fops;
    else if (S_ISSOCK(mode))
        ;  /* leave it no_open_fops */
    else
        printk(KERN_DEBUG "init_special_inode: bogus i_mode (%o) for"
                " inode %s:%lu\n" , mode, inode->i_sb->s_id,
                inode->i_ino);
}
/*
 * Dummy default file-operations: the only thing this does
 * is contain the open that then fills in the correct operations
 * depending on the special file...
 */
const struct file_operations def_chr_fops = {
    .open = chrdev_open ,
    .llseek = noop_llseek,
};
```

```
static int chrdev_open( struct inode *inode, struct file *filp)
{
  ret = -ENXIO;
    fops = fops_get(p->ops);
    if (!fops)
        goto out_cdev_put;

    replace_fops (filp, fops);
    if (filp->f_op->open) {
        ret = filp->f_op-> open (inode, filp);
        if (ret)
            goto out_cdev_put;
    }

    return 0;
  out_cdev_put:
    cdev_put(p);
    return ret;
}
```

上面分析的核心意思是：device_register()函数除了注册在/sys下面外，还做了一件重要的事情，通过devtmpfs_create_node() 在/dev目录下创建了一个设备节点(inode)，这个设备节点有一个默认的file_operations操作。

4. 理解cdev_add()函数的实质

```
/**
 * cdev_add() - add a char device to the system
 * @p: the cdev structure for the device
 * @dev: the first device number for which this device is responsible
 * @count: the number of consecutive minor numbers corresponding to this
 *         device
 *
 * cdev_add() adds the device represented by @p to the system, making it
 * live immediately.  A negative error code is returned on failure.
 */
int cdev_add( struct cdev *p, dev_t dev, unsigned count)
{
    int error;

    p->dev = dev;
    p->count = count;

    error = kobj_map (cdev_map, dev, count, NULL,
            exact_match, exact_lock, p);
    if (error)
       return error;

    kobject_get(p->kobj.parent);

    return 0;
}
```

kobj_map() 函数就是用来把字符设备编号和 cdev 结构变量一起保存到 cdev_map 这个散列表里 。当后续要打开一个字符设备文件时，通过调用 kobj_lookup() 函数， 根据设备编号就可以找到 cdev 结构变量，从而取出其中的 ops 字段 。此处只是简单的一个保存过程，并没有将cdev和inode关联起来。
有了这个关联之后，当我们使用mknod 命令，就会创建一个inode节点，并且通过 dev_t将inode和 cdev_map 里面的cdev联系起来了。

5.dev_t---联系inode cdev device三者的纽带

在创建device设备时，如果定义了dev_t，那么会创建一个inode节点，并且绑定一个cdev,以及该cdev的一个默认的file_operations;如果针对该dev_t,定义了自己的file_operations，再调用cdev_add()，那么就会用自己定义的file_operations替换掉那个默认的file_operations.这样 device和cdev以及自己定义的file_operations就关联起来了。
大致代码流程如下：

```
struct  class  *my_class;
struct  cdev my_cdev[N_MINORS];
dev_t dev_num;

static  int  __init my_init( void )
{
    int  i;
    dev_t curr_dev;

    /* Request the kernel for N_MINOR devices */
    alloc_chrdev_region(&dev_num, 0, N_MINORS,  "my_driver" );

    /* Create a class : appears at /sys/class */
    my_class = class_create(THIS_MODULE,  "my_driver_class" );

    /* Initialize and create each of the device(cdev) */
    for  (i = 0; i < N_MINORS; i++) {

        /* Associate the cdev with a set of file_operations */
        cdev_init(&my_cdev[i], &fops);

        /* Build up the current device number. To be used further */
        curr_dev = MKDEV(MAJOR(dev_num), MINOR(dev_num) + i);

        /* Create a device node for this device. Look, the class is
         * being used here. The same class is associated with N_MINOR
         * devices. Once the function returns, device nodes will be
         * created as /dev/my_dev0, /dev/my_dev1,... You can also view
         * the devices under /sys/class/my_driver_class.
         */
        device_create (my_class, NULL,  curr_dev , NULL,  "my_dev%d" , i);

        /* Now make the device live for the users to access */
        cdev_add (&my_cdev[i],  curr_dev , 1);
    }

    return  0;
}
```

5.将device和cdev关联起来，并定义自己的file_operations

实现该模式的典型设备就是misdevice，它同时具有device的标准设备模型，也定义了自己的file_operations。

按照上面的思路，推测出misdevice的注册过程应该是如下流程：

1.调用核心device注册函数device_add()

2.调用核心cdev注册函数cdev_add()

但是分析misc_register(),函数发现只调用了device_add()函数，并没有调用cdev_add()，那么自己定义的file_operations是如何和cdev关联起来的呢？

其实misc.c中用了另外一套思路，但是原理是一样的。

a)在misc_init()中， 通过register_chrdev()函数将主设备号为0，次设备号为0-255的所有cdev都通过cdev_add()进行了注册，也就是说将256个cdev放入到了cdev_map中，然后绑定的file_operations为默认的misc_open操作函数；

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

    if  ( register_chrdev (MISC_MAJOR, "misc" ,&misc_fops))
        goto  fail_printk;
    misc_class->devnode = misc_devnode;
    return  0;

    ...
}

int  __register_chrdev(unsigned  int  major, unsigned  int  baseminor,
            unsigned  int  count,  const  char  *name,
            const  struct  file_operations *fops)
{



    cd = __register_chrdev_region(major, baseminor, count, name);
    cdev = cdev_alloc();
    cdev->ops = fops;
    err =  cdev_add (cdev, MKDEV(cd->major, baseminor), count);
    ...
}
```

b)当调用misc_register()时，内核通过维护一个 misc_list 链表，misc设备在misc_register注册的时候链接到这个链表。

```
/**
 * misc_register   -   register a miscellaneous device
 * @misc: device structure
 *
 * Register a miscellaneous device with the kernel. If the minor
 * number is set to %MISC_DYNAMIC_MINOR a minor number is assigned
 * and placed in the minor field of the structure. For other cases
 * the minor number requested is used.
 *
 * The structure passed is linked into the kernel and may not be
 * destroyed until it has been unregistered. By default, an open()
 * syscall to the device sets file->private_data to point to the
 * structure. Drivers don't need open in fops for this.
 *
 * A zero is returned on success and a negative errno code for
 * failure.
 */

int  misc_register( struct  miscdevice * misc)
{
    dev_t dev;
    int  err = 0;

    INIT_LIST_HEAD(&misc->list);


    dev = MKDEV(MISC_MAJOR, misc->minor);

    misc->this_device =
        device_create_with_groups(misc_class, misc->parent, dev,
                      misc, misc->groups,  "%s" , misc->name);


    /*
     * Add it to the front, so that later devices can "override"
     * earlier defaults
     */
    list_add (&misc->list, &misc_list);

}
```

c)通过步骤b),可以操作对应dev_t的某个cdev的inode了，但此时的open为绑定的file_operations提供的默认的misc_open，在此函数中，会根据dev_t在内核的misc_list中进行查找，然后用自己定义的file_operations替换到misc提供的那个默认的file_operations。

```
static  int  misc_open( struct  inode * inode,  struct  file * file)
{
    int  minor = iminor(inode);
    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) {
        if  (c->minor == minor) {
            new_fops = fops_get(c->fops);
            break ;
        }
    }

    if  (!new_fops) {
        mutex_unlock(&misc_mtx);
        request_module( "char-major-%d-%d" , MISC_MAJOR, minor);
        mutex_lock(&misc_mtx);

        list_for_each_entry(c, &misc_list, list) {
            if  (c->minor == minor) {
                new_fops = fops_get(c->fops);
                break ;
            }
        }
        if  (!new_fops)
            goto  fail;
    }

    /*
     * Place the miscdevice in the file's
     * private_data so it can be used by the
     * file operations, including f_op->open below
     */
    file->private_data = c;

    err = 0;
    replace_fops(file, new_fops);
    if  (file->f_op->open)
        err = file->f_op->open(inode,file);
fail:
    mutex_unlock(&misc_mtx);
    return  err;
}
```

miscdevice通过上面的a) b) c)三部，实现了device cdev和自定义 file_operations的关联。

6.一个简单的小测试

通过命令mknod建立一个misc设备，然后去打开该设备，看下返回值，如下

```
root@imx6qsabresd:/dev# mknod test c 10 100
root@imx6qsabresd:/dev# cat /dev/test
cat: can't open '/dev/test': No such device
```

通过上面的分析，我们知道cat时，调用的open函数会最后调到misc_open(),该函数中返回的错误就是 -ENODEV，和看到的现象一致。

通过mknod建立一个设备，主次设备号随便，看下返回值，如下

```
root@imx6qsabresd:/dev# mknod aaa c 1 0
```

root@imx6qsabresd:/dev# cat /dev/aaa
cat: can't open '/dev/aaa': No such device or address
通过3的分析，我们知道，调用的open函数会最后调用到chrdev_open，该函数中返回的错误就是-ENXIO，和看到的错误提示一致。

7.linux设备驱动模型下的cdev
通过上面的分析，我们知道当device_add()注册device时，会调用devtmpfs_create_node(),但是这个调用是有个约束条件的，这个约束条件是device中必须定义了devt这个设备号。
所以对于 很多的平台设备platform_devices（也就是那些在dts文件中定义的devices）,在进行platform_device_add（）时,并不会在/dev下面出现inode节点。

但是i2c控制器，作为一个平台设备，确在/dev下面出现了设备节点，比如/dev/i2c-0 /dev/i2c-1
这是什么原因的?分析内核代码，我们发现i2c-dev.c这个文件，其实这个文件就是用来创建/dev/i2c-0这些设备的，它是一个我们熟悉的cdev设备驱动。i2c_dev_init()函数中使用了一种 内核通知机制，通过回调，在i2cdev_attach_adapter()中创建了一个带devt的device,那么自然会出现/dev/i2c-0节点了。内核通知机制的具体原理没有研究。

8.linux设备驱动模型下的eeprom驱动
我们传统的用法是习惯在/dev下注册eeprom设备，也就是所谓的cdev设备，然后操作。
但是内核中用的是基于/sys系统的devices驱动模型，使用这个模型时，我们在dts文件中，在相应的i2c控制器下配置好好，就能在通过/sys文件系统进行访问了。比如如下的dts配置，我再i2c4下挂了一个eeprom芯片。内核会在初始化时将device于at24的驱动进行绑定（具体过程是dts以及platform设备模型的相关知识，不熟悉的可以参考链接
http://www.wowotech.net/device_model/13.html）
&i2c4{
eerpom: at24c04@54{
compatible = "24c04";
reg = <0x54>;
status = "okay";
};
};
设备注册成功后，我们能看到如下信息
root@imx6qsabresd:/sys/class/i2c-dev/i2c-3/device/3-0054# ls
driver modalias of_node subsystem eeprom name power uevent
该目录下有很多属性文件，如果我们想读写eeprom，那么我们可以通过操作eeprom这个文件实现读写，这些都是设备驱动模型和sys文件系统的相关知识了。
那么，如果我想基于这个架构，增加一个/dev/eeprom设备，该怎么实现呢？
首先，需要清楚，在内核调用at24_probe()函数之前，eprom作为devices，已经被注册到了系统中（这个过程是内核在解析dts配置文件时自动完成的，不是使用者自己注册的）。
由于在内核注册devices时，并没有分配dev_t这个设备号，所以不会在/dev下面创建设备节点。那么现在想创建一个cdev设备，但是这个设备号内核注册时并没有分配给这个device,怎么办？
有两种方案实现：
方案一：
在at24_probe()函数中，直接注册一个misc设备，实现file_operations，从而能够建/dev/eeprom节点。使用该方法时，需要理解一点，其实你是将eeprom这个芯片注册了两次设备（devices）,一次是作为i2c的下挂的设备，由平台自动注册的；还有一次是自己注册的misc设备。这两个设备都在/sys系统下存在。这两个设备对sys来说是完全独立的，只是因为我们自己将他们的驱动写在了一起实现。
root@imx6qsabresd:/sys/class/i2c-dev/i2c-3/device/3-0054# ls
driver modalias of_node subsystem eeprom name power uevent

root@imx6qsabresd:/sys/devices/virtual/misc/eeprom# ls
dev power subsystem uevent
root@imx6qsabresd:/# ls -al /dev/eeprom
crw------- 1 root root 10, 100 Jan 1 1970 /dev/eeprom

方案二：
在at24_probe()函数中，给入参i2c_client里面的device里的dev_t分配一个设备号，然后通过cdev_init(),cdev_add()函数将该设备号与file_operations进行关联，再通过mknod创建节点。
使用此方法，实现了在/sys系统中只注册了一个device，但既能够通过sys系统访问，也能够通过/dev/eeprom设备节点的形式访问了。
下面是自己尝试在at24.c里面增加的代码

```
static int eeprom_open( struct inode *inode, struct file *filp) {
    unsigned int major, minor;
    major = imajor(inode);
    minor = iminor(inode);

    printk( "%s, major = %d, minor = %d!\n" , __func__, major, minor);
    return 0;
}
```

```
static ssize_t eeprom_read( struct file *filp, char *buf,
                    size_t len, loff_t *offset) {
    printk( "%s, len = %d\n" , __func__, len);
    return 0;
}

static ssize_t eeprom_write( struct file *filp, const char *buf,
                    size_t len, loff_t *offset) {
    printk( "%s, len = %d\n" , __func__, len);
    return 0;
}

static int eeprom_close( struct inode *inode, struct file *filp) {
    printk( "%s\n" , __func__);
    return 0;
}


static struct file_operations eeprom_fops = {
    .open      = eeprom_open,
    .read      = eeprom_read,
    .write       = eeprom_write,
    .release   = eeprom_close,
};




static int at24_probe( struct i2c_client *client, const struct i2c_device_id *id)
{

     ......

    /*
     * add the char device to system
     */
    dev = &client->dev;

    dev->devt = MKDEV(100, 0);
    err = register_chrdev_region(dev->devt, 1, "eeprom" );
    if (err < 0) {
       dev_err(&client->dev, "Can't static register chrdev region!\n" );
       return err;
    }
    cdev_init(&eeprom_cdev, &eeprom_fops);
    err = cdev_add(&eeprom_cdev, dev->devt, 1);
    if (err < 0) {
       dev_err(&client->dev, "Can't add char device!\n" );
       return err;
    }
    dev_err(&client->dev, "add char eeprom device!\n" );

    return 0;
}
```

```
root@imx6qsabresd:/sys/class/i2c-dev/i2c-3/device/3-0054# ls
driver modalias of_node subsystem eeprom name power uevent
root@imx6qsabresd:/# ls -al /dev/eeprom
crw------- 1 root root 100, 0 Jan 1 1970 /dev/eeprom
```