主要分有两大块：

1. 搭建环境
2. 开发驱动并测试

作为一个记录，从搭建到开发过程细节，简单的管理内存的虚拟字符设备开发。

搭建环境

环境准备

1. Linux 源码

```
~/Desktop/workspace/virtual_chardev/linux-5.10                    11:11:41
❯ ls
arch      CREDITS        fs       Kbuild    LICENSES     net       security  virt
block     crypto         include  Kconfig   MAINTAINERS  README    sound
certs     Documentation  init     kernel    Makefile     samples   tools
COPYING   drivers        ipc      lib       mm           scripts   usr
```

2. QEMU 虚拟机(ARM)

3. 对应的 toolchains(ARM)

```
sudo apt install gcc-arm-linux-gnueabi
```

准备过程

1. QEMU ARM 版本

```
sudo apt install qemu-system-arm
```

2. 编译源码

    1. 使用默认的配置进行初始化

    ```
    make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- vexpress_defconfig
    ```

    2. 使用 `menuconfig` 进行调整裁剪不需要的模块和功能等

    ```
    make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- menuconfig
    ```

    3. 利用生成的 `.config` 进行编译

    ```
    make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- -j16
    ```

3. 测试启动内核

```
qemu-system-arm \
    -M vexpress-a9 \
    -m 1024M \
    -kernel arch/arm/boot/zImage \
    -dtb arch/arm/boot/dts/vexpress-v2p-ca9.dtb \
    -append "root=/dev/mmcblk0 rw console=ttyAMA0" \
    -serial stdio
#   -nographic 和 -serial stdio 不能同时使用
```

完善环境

测试内核启动报错

> ---[end Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(0,0)]---

由于只有内核不能完整启动，还需要一个根文件系统，选择较为简单的 `busybox` 使用。

1. 编译 `busybox`

    注意选择

    > Settings --->
    > --- Build Options
    > [*] Build static binary (no shared libs)

    `make install` 之后生成的文件和目录被生成到 `_install` 目录中

```
bin  linuxrc  sbin  usr
```

2. 制作文件镜像

```
dd if=/dev/zero of=rootfs.ext4 bs=1M count=256
mkfs.ext4 rootfs.ext4
sudo mount rootfs.ext4 /mnt
sudo cp -rf _install/* /mnt
ls /mnt
sudo umount /mnt
```

3. 启动测试

```
qemu-system-arm \
    -M vexpress-a9 \
    -m 1024M \
    -kernel arch/arm/boot/zImage \
    -dtb arch/arm/boot/dts/vexpress-v2p-ca9.dtb \
    -append "root=/dev/mmcblk0 rw console=ttyAMA0 init=/sbin/init" \
    -sd rootfs.ext4 \
    -smp 4 \
    -serial stdio
```

4. 启动成功但不完善，需要创建出这些文件

> EXT4-fs (mmcblk0): mounted filesystem with ordered data mode. Opts: (null)
> VFS: Mounted root (ext4 filesystem) on device 179:0.
> Freeing unused kernel memory: 1024K
> Run /sbin/init as init process
> random: crng init done
> can't run '/etc/init.d/rcS': No such file or directory
>
> can't open /dev/tty2: No such file or directory
> can't open /dev/tty3: No such file or directory
> can't open /dev/tty4: No such file or directory

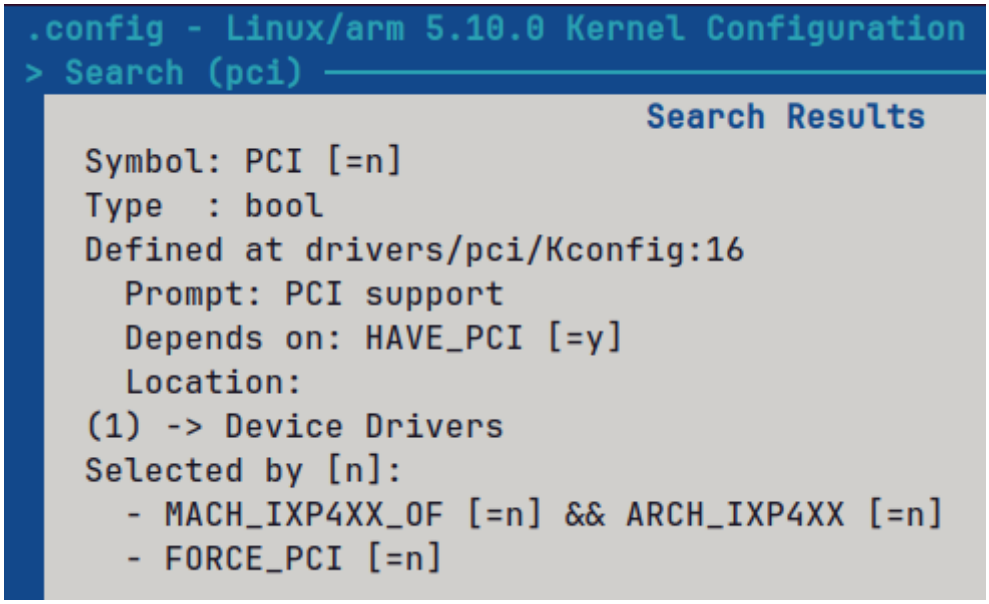
```
sudo mount rootfs.ext4 /mnt
cd /mnt
sudo mkdir proc sys dev etc etc/init.d
sudo touch etc/init.d/rcS
```

```
# etc/init.d/rcS 文件内容
#!/bin/sh
mount -t proc none /proc
mount -t sysfs none /sys
/sbin/mdev -s
```

```
sudo chmod +x etc/init.d/rcS
cd ..
sudo umount /mnt
```

再次使用 3 点的命令启动成功。

共享文件



可以在 `/etc/init.d/` 目录下添加脚本，并确保它在启动时被执行。

QEMU 启动命令添加

```
qemu-system-arm \
    -M vexpress-a9 \
    -m 1024M \
    -kernel arch/arm/boot/zImage \
    -dtb arch/arm/boot/dts/vexpress-v2p-ca9.dtb \
    -append "root=/dev/mmcblk0 rw console=ttyAMA0 init=/sbin/init" \
    -sd rootfs.ext4 \
    -smp 4 \
    -serial stdio \
    -virtfs
local,path=/home/liyunfeng/Desktop/shared,mount_tag=host0,security_model=pas
sthrough,id=host0


mount -t 9p -o trans=virtio,version=9p2000.L host0 /mnt/shared

-fsdev
local,security_model=passthrough,id=fsdev0,path=/home/liyunfeng/Desktop/shar
ed \
    -device virtio-9p-pci,id=fs0,fsdev=fsdev0,mount_tag=host0
```

虚拟字符设备驱动开发

大致流程中需要用到相关函数

虚拟字符设备主要控制了一段内核空间的内存，用户空间进行通过设备 `/dev/xxx` 进行读写，涉及到一个内核和用户之间的内存读写，主要函数 `copy_to_user` 和 `copy_from_user` 操作。

```
static inline long copy_from_user(void *to, const void __user * from,
unsigned long n)

static inline long copy_to_user(void __user *to, const void *from, unsigned
long n)
/*
to：指定目标地址，也就是数据存放的地址，

from：指定源地址，也就是数据的来源。

n：指定写入/读取数据的字节数。
*/
```

其中像 `open`、`release`、`write`、`read` 等都是需要实现的，具体需要实现哪些函数还是要看具体的驱动要求。

驱动模块加载和卸载

1. 直接编译进内核
2. 使用模块

```
insmod xxx.ko
rmmod xxx.ko
```

加载和卸载函数

```
/* 驱动入口函数 */
static int __init xxx_init(void)
{
    /* 入口函数具体内容 */
    return 0;
}

/* 驱动出口函数 */
static void __exit xxx_exit(void)
{
    /* 出口函数具体内容 */
}
module_init(xxx_init);
//注册模块加载函数
module_exit(xxx_exit);
//注册模块卸载函数
```

设备号的分配

1. 可以静态手动分配，但不保证没有被占用
2. 动态分配，使用 `alloc_chrdev_region` 函数

```
// 在这个函数中，dev_t 是一个数据类型，通常表示设备号，由主设备号（major number）和次要设备号（minor number）组成。baseminor 是分配的起始次要编号，count 是请求分配的设备数量。name 是分配的设备区域的名称，这个名称将用于在 /dev 目录下创建相应的设备文件。如果函数执行成功，它会将分配的设备号写入到 dev 指针指向的内存位置，并返回0。如果失败，它会返回一个负数表示的错误码。
// 动态申请设备号
int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count,
            const char *name)
    /*
     * dev: 指向dev_t类型的指针，用于存储分配的设备号。
     *      如果函数调用成功，内核将在这里设置设备号。
     * baseminor: 指定分配的设备号的起始次要编号（minor number）。
     * count: 要分配的设备号的数量。
     * name: 设备的名称，用于在/dev目录下创建设备文件。
     *
     * 函数返回值：
     * 成功时返回0。
     * 失败时返回负错误码。
     */

// 注销字符设备后，要释放设备号
// register_chrdev_region函数 以及alloc_chrdev_region函数分配得到的设备编号，可以使用unregister_chrdev_region函数实现该功能。
void unregister_chrdev_region(dev_t from, unsigned count)
    /*
     * from: 要注销的字符设备号的起始设备号。
     *       这个设备号通常是由 alloc_chrdev_region 函数分配的。
     * count: 要注销的设备号的数量。
     *
     * 这个函数会释放指定的设备号区域，并且删除相应的 /dev 目录下的设备文件。
     * 它不返回任何值，即没有返回值。
     */
```

cdev 设备注册和注销

```
// 注册字符设备驱动的宏定义
// 除了上述的两种(alloc_chrdev_region/register_chrdev_region)，内核还提供了register_chrdev函数用于分配设备号。该函数是一个内联函数，它不 仅支持静态申请设备号，也支持动态申请设备号，并将主设备号返回
static inline int register_chrdev(unsigned int major, const char *name,
                  const struct file_operations *fops)
    // major: 指定设备的主要编号，用于区分不同的设备驱动。
    //        如果设置为0，内核将自动分配一个未使用的major号。
    // name: 设备的名称，用于在/dev目录下创建设备文件。
    // fops: 指向file_operations结构体的指针，包含了设备操作的函数指针集合。
    //       这个结构体定义了如何打开、读取、写入、关闭设备等操作。

// 注销字符设备驱动的宏定义
```

```
// 使用register函数申请的设备号，则应该使用unregister_chrdev函数进行注销。
static inline void unregister_chrdev(unsigned int major, const char *name)
    // major: 设备的主要编号，用于指定要注销的设备驱动。
    // name: 设备的名称，用于与注册时的名称进行匹配，确保注销正确的设备。
    // 这个宏定义会从内核中移除指定的字符设备驱动，以及删除 /dev 目录下的对应设备文件。
```

一般字符设备的注册在驱动模块的入口函数 xxx_init 中进行，字符设备的注销在驱动模块的出口函数 xxx_exit 中进行。

成功之后可以用 `cat /proc/devices` 查看注册的设备和主设备号。

实现具体的操作函数

对 `file_operations` 结构体实现里面的函数，目前只需要 `open` `release` `write` `read` 实现。

设备节点的创建和销毁

创建一个设备并将其注册到文件系统

```
struct device *device_create(struct class *class, struct device *parent,
            dev_t devt, void *drvdata, const char *fmt, ...)
/**
class：指向这个设备应该注册到的struct类的指针；

parent：指向此新设备的父结构设备（如果有）的指针；

devt：要添加的char设备的开发；

drvdata：要添加到设备进行回调的数据；

fmt：输入设备名称。

成功时返回 struct device 结构体指针，错误时返回ERR_PTR().
*/
```

删除使用device_create函数创建的设备

```
void device_destroy(struct class *class, dev_t devt)
/*
class：指向注册此设备的struct类的指针；

devt：以前注册的设备的开发；
*/
```

除了使用代码创建设备节点，还可以使用mknod命令创建设备节点。

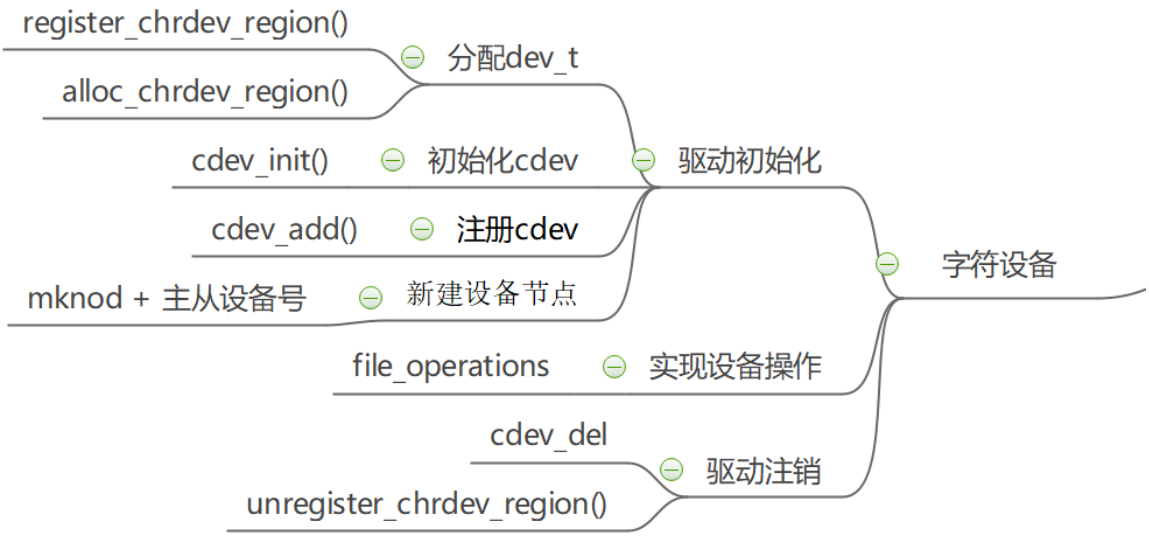
用法：mknod 设备名 设备类型 主设备号 次设备号

- b 创建(有缓冲的)区块特殊文件
- c, u 创建(没有缓冲的)字符特殊文件
- p 创建先进先出(FIFO)特殊文件

```
mknod /dev/xxx c 220 0
```

最后添加协议信息

```
1 MODULE_LICENSE("GPL");
```



程序编写

> 我们创建一个字符设备的时候，首先要的到一个设备号，分配设备号的途径有静态分配和动态分配； 拿到设备的唯一ID，我们需要实现file_operation并保存到cdev中，实现cdev的初始化； 然后我们需要将我们所做的工作告诉内核，使用cdev_add()注册cdev； 最后我们还需要创建设备节点，以便我们后面调用file_operation接口。
>
> 注销设备时我们需释放内核中的cdev，归还申请的设备号，删除创建的设备节点。

测试程序

```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>

char *write_buf = "Test application 测试程序。\n";
char read_buf[128];

int main(void)
{
    printf("%s", write_buf);

    // open file
    int fd = open("/dev/mem_ctl", O_RDWR);
    // filename use `mknod /dev/mem_ctl c xxx 0` generate
    // xxx via `cat /proc/devices` check

    // write data
    write(fd, write_buf, strlen(write_buf));
    printf("write success 写入成功。\n");
    close(fd);

    // reopen
    fd = open("/dev/mem_ctl", O_RDWR);
    // read data
    read(fd, read_buf, 128);
    // print
    printf("Read content: %s", read_buf);
    close(fd);
    return 0;
}
```

```
qemu-aarch64 /mnt/shared # cat /proc/devices
Character devices:
  1 mem
  4 ttyS
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
 10 misc
128 ptm
136 pts
204 ttyAMA
249 mem_ctl
250 bsg
```

```
# mknod /dev/mem_ctl c 249 0
```

```
qemu-aarch64 /mnt/shared # ./app
Test application 测试程序。
[  321.317095] mem_ctl open.
write success 写入成功。
[  321.317664] mem_ctl release.
[  321.317792] mem_ctl open.
Read content: Test application 测试程序。
[  321.318065] mem_ctl release.
```

驱动程序改进

- 支持多个设备
- 自动创建设备
- `platform` 总线（设备、驱动、总线形式）

支持多个设备有两种方式，一是通过定义不同的缓冲区，通过设备的次设备号区分，然后选择初始化哪个，用到了 `filp->private_data` 的形式，这个变量是文件指针的私有数据，一般就是用来放自己想要保存使用的数据，这样的方式抽象程度不够高，每增加一个设备要增加一个缓冲区和 `switch` 选项；二是通过抽象一个自己的字符设备结构体，在里面包含 `struct cdev` 结构和数据。

主要修改代码有

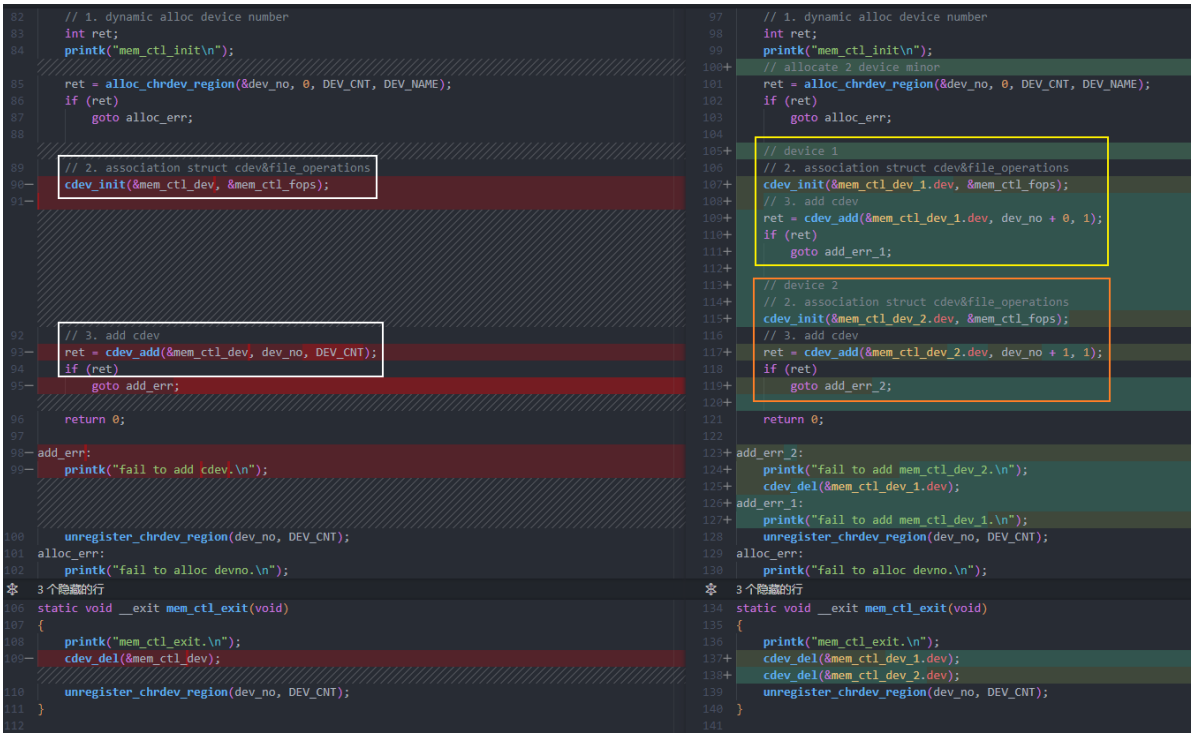
多设备

将 `cdev` 的数据抽象成一个自定义的结构体

```
#include <linux/fs.h>

static dev_t dev_no; // device number(major minor)
static struct cdev mem_ctl_dev; // cdev struct

#define DEV_NAME "mem_ctl"
#define DEV_CNT 1
#define BUF_SIZE 128

static char vbuf[BUF_SIZE];
```

```
#include <linux/fs.h>

#define DEV_NAME "mem_ctl"
#define DEV_CNT 2
#define BUF_SIZE 128

static dev_t dev_no; // device number(major minor)
// user define data struct of mem_ctl,
// container cdev structure, data area.
struct mem_ctl_dev {
    struct cdev dev;
    char vbuf[BUF_SIZE];
};
// device 1
static struct mem_ctl_dev mem_ctl_dev_1;
// device 2
static struct mem_ctl_dev mem_ctl_dev_2;
```

分别初始化两个设备



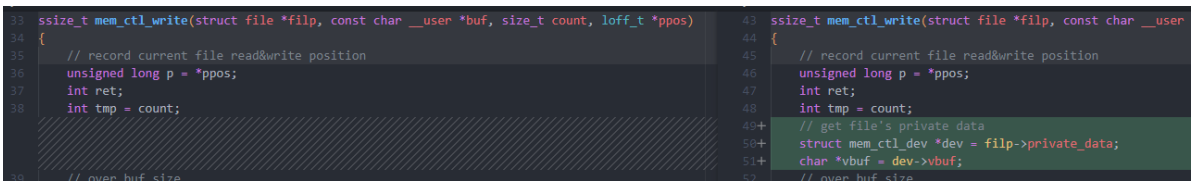
设置对应的地址

利用 `container_of` 宏通过 `inode` 中保存的 `cdev` 地址来获得自定义结构体的地址，并设置成文件（也就是应用程序中 `open` ）的私有数据成员

```c
static int mem_ctl_open(struct inode *inode, struct file *filp)
{
    // printk("mem_ctl open.\n");
    // via container_of get mem_ctl_dev address
    // inode's member i_cdev saved cdev structure address.
    filp->private_data = container_of(inode->i_cdev, struct mem_ctl_dev, dev);
    return 0;
}
```

修改 `write` 的位置

在 `write` 函数中获取到这个私有数据成员，也就是设置的 自定义结构体的地址，并获得对应的 `buf` 数据地址



驱动的测试

```
qemu-aarch64 /mnt/shared # insmod mem_ctl.ko
[59574.958641] mem_ctl_init
qemu-aarch64 /mnt/shared # mknod /dev/mem_ctl_1 c 249 0
qemu-aarch64 /mnt/shared # mknod /dev/mem_ctl_2 c 249 1
qemu-aarch64 /mnt/shared # ls -alh /dev/mem_ctl*
crw------- 1 root root 249,  0 Aug 14 02:22 /dev/mem_ctl_1
crw------- 1 root root 249,  1 Aug 14 02:22 /dev/mem_ctl_2
qemu-aarch64 /mnt/shared #
```

```
qemu-aarch64 /mnt/shared # echo "device 1 data" > /dev/mem_ctl_1
qemu-aarch64 /mnt/shared # echo "device 2 data" > /dev/mem_ctl_2
qemu-aarch64 /mnt/shared # cat /dev/mem_ctl_2
device 2 data
qemu-aarch64 /mnt/shared # cat /dev/mem_ctl_1
device 1 data
qemu-aarch64 /mnt/shared #
```

自动创建和删除设备节点

主要使用到 `class_create` 宏和 `device_create` `device_destroy` 函数

```
#define class_create(owner, name)       \
({                      \
    static struct lock_class_key __key; \
    __class_create(owner, name, &__key);    \
})
// class_create 一共有两个参数，参数 owner 一般为 THIS_MODULE，参数 name 是类名字。
// 返回值是个指向结构体 class 的指针，也就是创建的类。


/**
 * class_destroy - destroys a struct class structure
 * @cls: pointer to the struct class that is to be destroyed
 *
 * Note, the pointer to be destroyed must have been created with a call
 * to class_create().
 */
// 参数 cls 就是要删除的类。
void class_destroy(struct class *cls)
{
    if ((cls == NULL) || (IS_ERR(cls)))
        return;

    class_unregister(cls);
}
```

```
struct device *device_create(struct class *class, struct device *parent,
                 dev_t devt, void *drvdata, const char *fmt, ...)
    /*
     * class: 指向设备类（struct class）的指针，设备类定义了一组设备的共同属性。
     * parent: 指向父设备的指针（struct device），如果设备没有父设备，则此参数为
NULL。
     * devt: 设备号，一个dev_t类型的值，由主设备号和次设备号组成。
     *       内核将使用这个设备号来区分不同的设备。
     * drvdata: 驱动数据，这是驱动程序用来关联设备的一个私有数据指针。
     * fmt: 格式化字符串，用来创建设备名称。
     * ...: 可变参数列表，提供格式化字符串所需的参数。
     *
     * 函数返回值：
     * 成功时返回新创建的设备指针。
     * 失败时返回NULL或错误指针。
     */

```

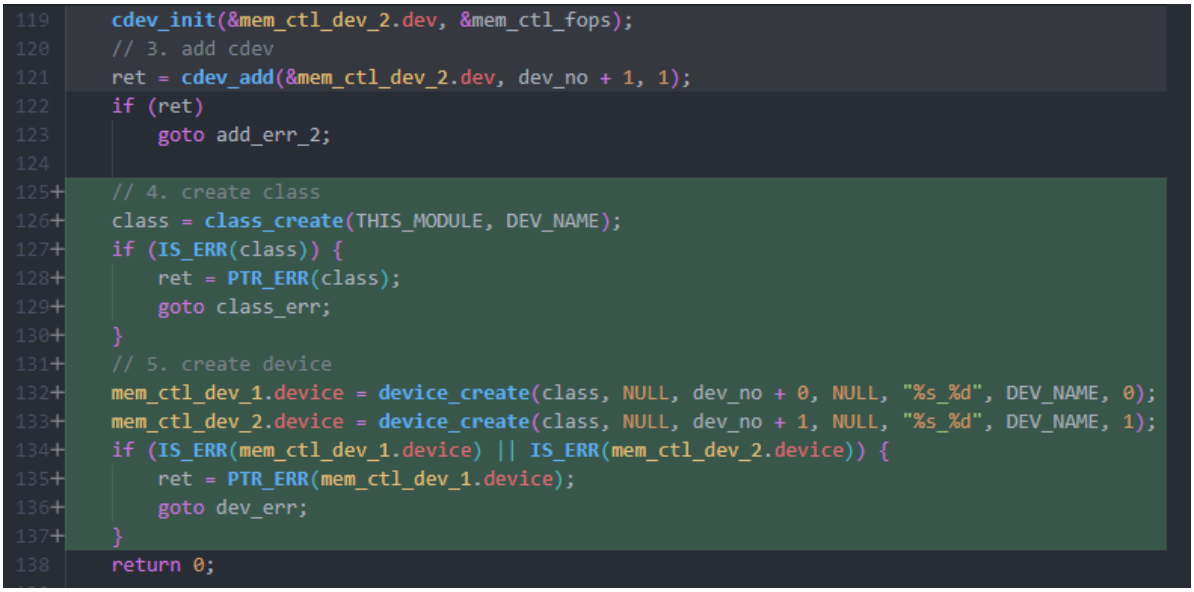
```
/**
 * device_destroy - removes a device that was created with device_create()
 * @class: pointer to the struct class that this device was registered with
 * @devt: the dev_t of the device that was previously registered
 *
 * This call unregisters and cleans up a device that was created with a
 * call to device_create().
 */
    // 参数 class 是要删除的设备所处的类，参数 devt 是要删除的设备号。
void device_destroy(struct class *class, dev_t devt)
```

所以在代码中还需要添加 `struct class *` 和 `struct device *` 两个指针

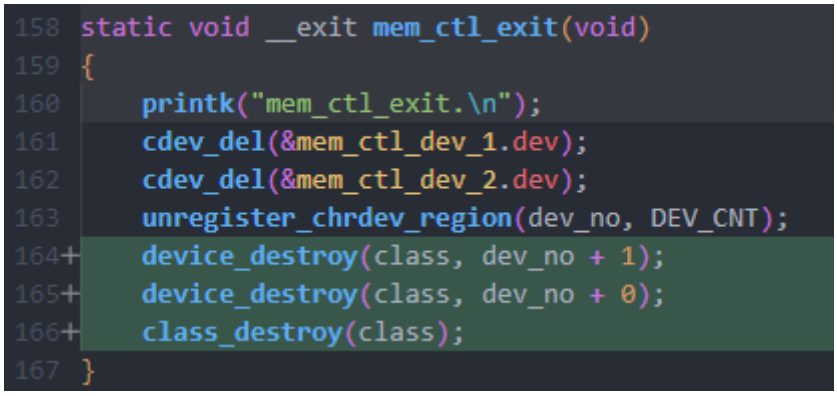
关键部分代码修改

```
static dev_t dev_no; // device number(major minor)
// user define data struct of mem_ctl,
// container cdev structure, data area.
struct mem_ctl_dev {
    struct cdev dev;
    char vbuf[BUF_SIZE];
};
```

```
static dev_t dev_no; // device number(major minor)
+static struct class *class;
// user define data struct of mem_ctl,
// container cdev structure, data area.
struct mem_ctl_dev {
    struct cdev dev;
+   struct device *device;
    char vbuf[BUF_SIZE];
};
```



```
    cdev_init(&mem_ctl_dev_2.dev, &mem_ctl_fops);
    // 3. add cdev
    ret = cdev_add(&mem_ctl_dev_2.dev, dev_no + 1, 1);
    if (ret)
        goto add_err_2;

+   // 4. create class
+   class = class_create(THIS_MODULE, DEV_NAME);
+   if (IS_ERR(class)) {
+       ret = PTR_ERR(class);
+       goto class_err;
+   }
+   // 5. create device
+   mem_ctl_dev_1.device = device_create(class, NULL, dev_no + 0, NULL, "%s_%d", DEV_NAME, 0);
+   mem_ctl_dev_2.device = device_create(class, NULL, dev_no + 1, NULL, "%s_%d", DEV_NAME, 1);
+   if (IS_ERR(mem_ctl_dev_1.device) || IS_ERR(mem_ctl_dev_2.device)) {
+       ret = PTR_ERR(mem_ctl_dev_1.device);
+       goto dev_err;
+   }
    return 0;
```



```
static void __exit mem_ctl_exit(void)
{
    printk("mem_ctl_exit.\n");
    cdev_del(&mem_ctl_dev_1.dev);
    cdev_del(&mem_ctl_dev_2.dev);
    unregister_chrdev_region(dev_no, DEV_CNT);
+   device_destroy(class, dev_no + 1);
+   device_destroy(class, dev_no + 0);
+   class_destroy(class);
}
```

可以看到加载驱动之后自动创建了两个设备

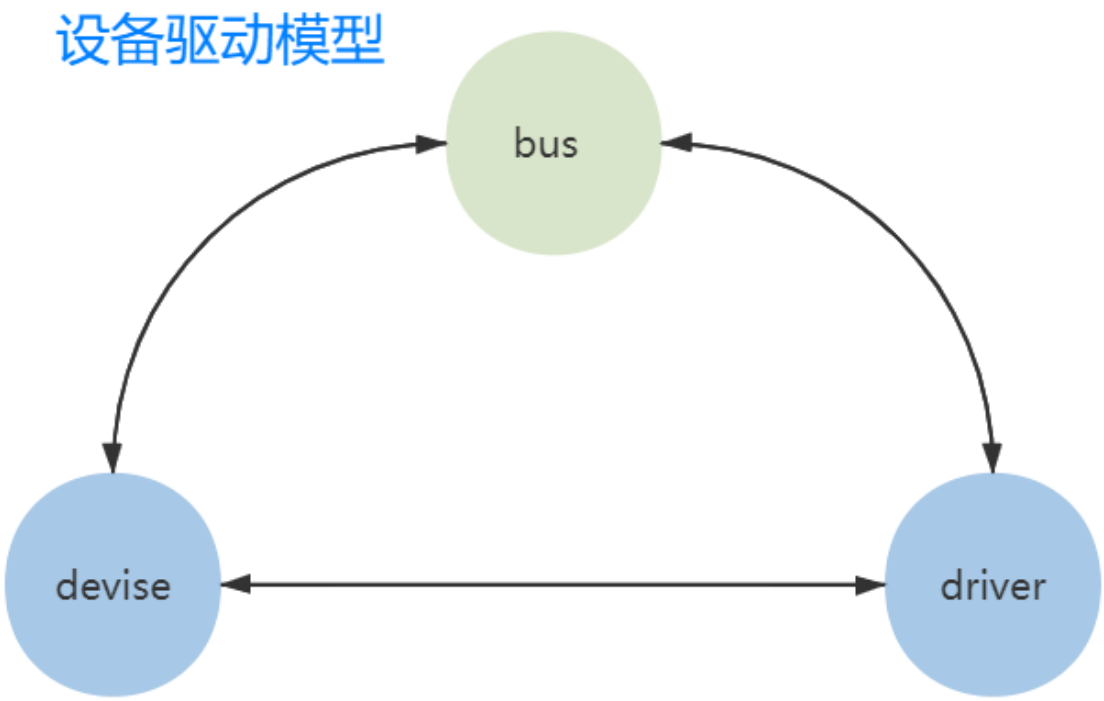
```
qemu-aarch64 /mnt/shared # ls -alh /dev/mem*
crw-r----- 1 root root 1,  1 Jan  1  1970 /dev/mem
qemu-aarch64 /mnt/shared # rmmod mem_ctl.ko
[62796.837754] mem_ctl_exit.
qemu-aarch64 /mnt/shared # lsmod | grep mem
qemu-aarch64 /mnt/shared # insmod mem_ctl.ko
[62808.965913] mem_ctl_init
qemu-aarch64 /mnt/shared # lsmod | grep mem
mem_ctl                16384  0
qemu-aarch64 /mnt/shared # ls -alh /dev/mem*
crw-r----- 1 root root   1, 1 Jan  1  1970 /dev/mem
crw-rw---- 1 root root 249, 0 Aug 14 03:15 /dev/mem_ctl_0
crw-rw---- 1 root root 249, 1 Aug 14 03:15 /dev/mem_ctl_1
qemu-aarch64 /mnt/shared #
```

效果相同

```
qemu-aarch64 /mnt/shared # cat /dev/mem_ctl_0
qemu-aarch64 /mnt/shared # cat /dev/mem_ctl_1
qemu-aarch64 /mnt/shared # echo "device 000 data" > /dev/mem_ctl_0
qemu-aarch64 /mnt/shared # echo "device 111 data" > /dev/mem_ctl_1
qemu-aarch64 /mnt/shared # cat /dev/mem_ctl_1
device 111 data
qemu-aarch64 /mnt/shared # cat /dev/mem_ctl_0
device 000 data
qemu-aarch64 /mnt/shared #
```

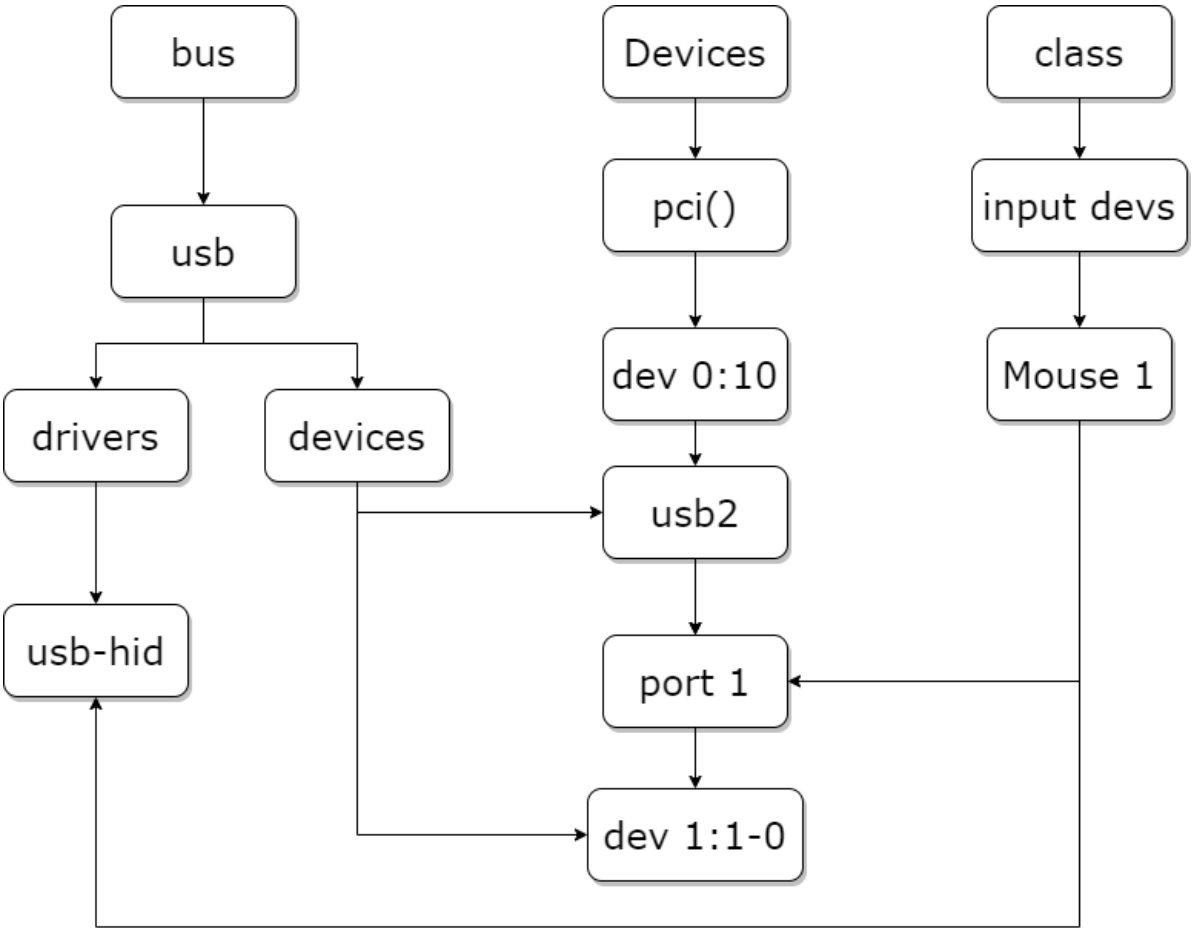
设备驱动模型 platform 总线

> 设备与驱动。设备负责提供硬件资源而驱动代码负责去使用这些设备提供的硬件资源。 并由总线将它们联系起来。



> 设备模型通过几个数据结构来反映当前系统中总线、设备以及驱动的工作状况，提出了以下几个重要概念：
>
> - **设备(device)**：挂载在某个总线的物理设备；
> - **驱动(driver)**：与特定设备相关的软件，负责初始化该设备以及提供一些操作该设备的操作方式；
> - **总线（bus)**：负责管理挂载对应总线的设备以及驱动；
> - **类(class)**：对于具有相同功能的设备，归结到一种类别，进行分类管理；

> - `/sys/bus` 目录下的每个子目录都是注册好了的总线类型。这里是设备按照总线类型分层放置的目录结构， 每个子目录(总线类型)下包含两个子目录——`devices` 和 `drivers` 文件夹；其中 `devices` 下是该总线类型下的所有设备， 而这些设备都是 **符号链接**，它们分别指向真正的设备(`/sys/devices/` 下)；如下图 `bus` 下的 `usb` 总线中的 `device` 则是Devices 目 录下 `/pci()/dev 0:10/usb2` 的符号链接。而 `drivers` 下是所有注册在这个总线上的驱动，每个 `driver` 子目录下 是一些可以观察和修改的 `driver` 参数。
>
> - `/sys/devices` 目录下是全局设备结构体系，包含所有被发现的注册在各种总线上的各种物理设备。一般来说， 所有的物理设备都按其在总线上的拓扑结构来显示。/sys/devices是内核对系统中所有设备的分层次表达模型， 也是/sys文件系统管理设备的最重要的目录结构。
>
> - `/sys/class` 目录下则是包含所有注册在kernel里面的设备类型，这是按照设备功能分类的设备模型， 我们知道每种设备都具有自己特定的功能，比如：鼠标的功能是作为人机交互的输入，按照设备功能分类无论它 挂载在哪条总线上都是归类到/sys/class/input下。

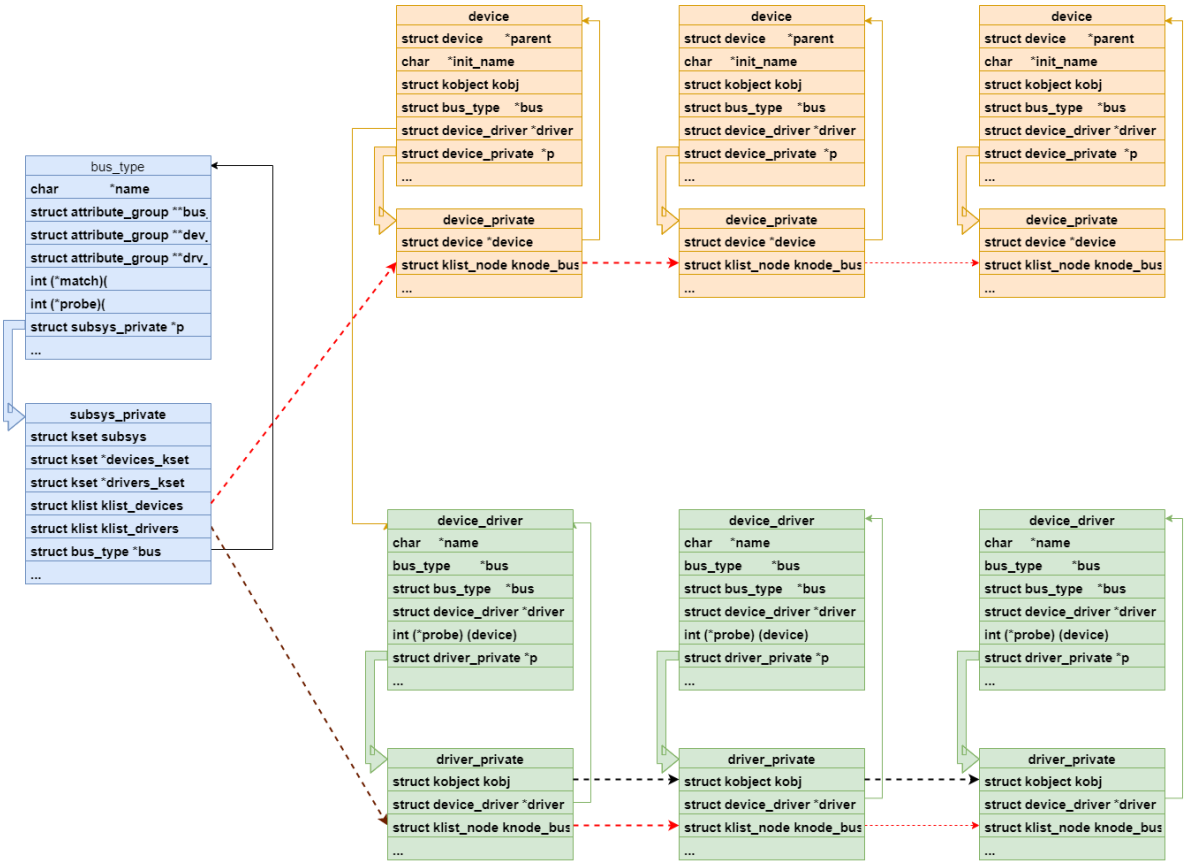


> 在总线上管理着两个链表，分别管理着设备和驱动，当我们向系统注册一个驱动时，便会向驱动的管理链表插入我们的新驱动， 同样当我们向系统注册一个设备时，便会向设备的管理链表插入我们的新设备。在插入的同时总线会执行一个bus_type结构体中match的方法对新插入的设备/驱动进行匹配。(它们之间最简单的匹配方式则是对比名字，存在名字相同的设备/驱动便成功匹配)。在匹配成功的时候会调用驱动device_driver结构体中probe方法(通常在probe中获取设备资源，

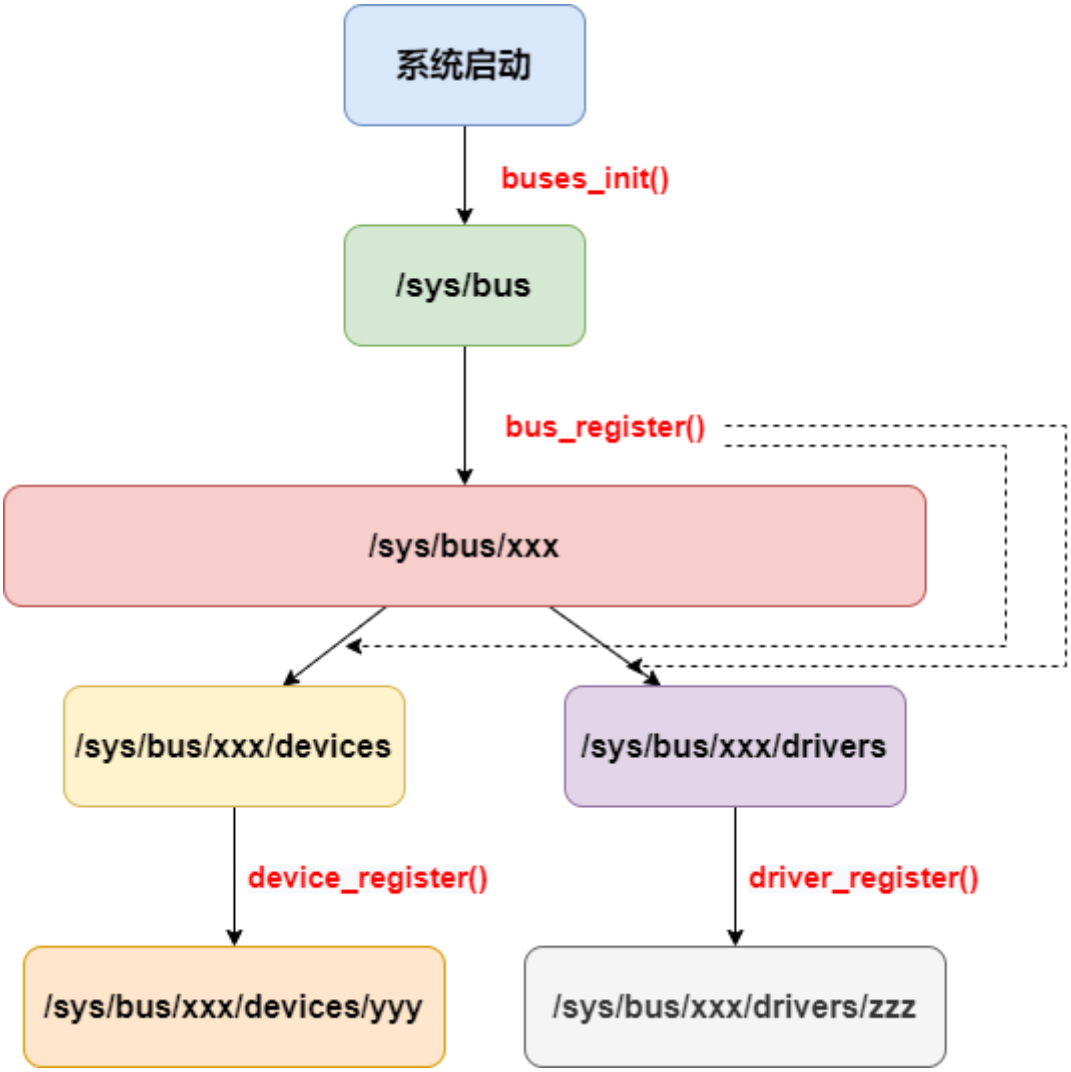
> 具体的功能可由驱动编写人员自定义)， 并且在移除设备或驱动时，会调用device_driver结构体中remove方法。

以上只是设备驱动模型的 **机制** ，上面的match、probe、remove等方法需要自己实现功能。

总线关联上设备与驱动之后的数据结构关系图



系统启动之后会调用buses_init函数创建/sys/bus文件目录，这部分系统在开机时已经帮我们准备好了，接下去就是通过总线注册函数bus_register进行总线注册，注册完总线后在总线的目录下生成devices文件夹和drivers文件夹， 最后分别通过device_register以及driver_register函数注册相对应的设备和驱动。



Linux 已经创建好了 `platform_bus_type` 的类型，同时也提供了一些对应的函数比如重要的匹配(match) 函数，platform总线提供了四种匹配方式，并且这四种方式存在着优先级：设备树机制>ACPI匹配模式>id_table方式>字符串比较。

驱动需要 实现 probe 函数，当 platform 总线成功匹配驱动和设备时，则会调用驱动的 probe 函数，在该函数中使用 `platform_get_resource` 或 `dev_get_platdata` 函数接口来获取资源， 以初始化设备，最后填充结构体 platform_driver ， 调用 platform_driver_register 进行注册。

此时代码应该有两部分，同时也有两个模块，一个设备，一个驱动

1. 编写第一个内核模块，设备模块
2. 在模块中定义一个 platform 设备，并填充相关设备信息
3. 在该模块入口函数，注册/挂载这个平台设备
4. 编写第二个内核模块，驱动模块
5. 在模块中定义一个平台驱动，在 probe 函数中完成字符设备驱动的创建
6. 在该模块入口函数，注册/挂载这个平台驱动

因为是虚拟的字符设备，没有资源，`resource` 定义成 1 个中断号，定义私有数据为一个 `int` 变量

此时未加载任何设备和驱动

```
qemu-aarch64 /mnt/shared # ls -alh /dev/mem_ctl*
ls: cannot access '/dev/mem_ctl*': No such file or directory
qemu-aarch64 /mnt/shared # lsmod | grep mem_ctl
qemu-aarch64 /mnt/shared # cat /proc/devices | grep mem_ctl
qemu-aarch64 /mnt/shared #
```

```
qemu-aarch64 /mnt/shared # ls /sys/bus/platform/drivers
 ARM-CCI         basic-mmio-gpio   of_fixed_factor_clk   syscon
'ARM-CCI PMU'    dw-apb-uart       of_serial             virtio-mmio
 alarmtimer      gpio-clk          pci-host-generic      xgene-gpio
 arm-ccn         gpio-dwapb        sbsa-uart
 armv8-pmu       of_fixed_clk      serial8250
qemu-aarch64 /mnt/shared # ls /sys/bus/platform/devices/
0.flash               a001600.virtio_mmio  a003200.virtio_mmio
4010000000.pcie       a001800.virtio_mmio  a003400.virtio_mmio
9020000.fw-cfg        a001a00.virtio_mmio  a003600.virtio_mmio
a000000.virtio_mmio   a001c00.virtio_mmio  a003800.virtio_mmio
a000200.virtio_mmio   a001e00.virtio_mmio  a003a00.virtio_mmio
a000400.virtio_mmio   a002000.virtio_mmio  a003c00.virtio_mmio
a000600.virtio_mmio   a002200.virtio_mmio  a003e00.virtio_mmio
a000800.virtio_mmio   a002400.virtio_mmio  alarmtimer.0.auto
a000a00.virtio_mmio   a002600.virtio_mmio  gpio-keys
a000c00.virtio_mmio   a002800.virtio_mmio  platform@c000000
a000e00.virtio_mmio   a002a00.virtio_mmio  pmu
a001000.virtio_mmio   a002c00.virtio_mmio  psci
a001200.virtio_mmio   a002e00.virtio_mmio  serial8250
a001400.virtio_mmio   a003000.virtio_mmio  timer
qemu-aarch64 /mnt/shared #
```

首先加载设备的模块

可以看到已经注册进 platform 设备中，模块也正确加载

```
qemu-aarch64 /mnt/shared # insmod mem_ctl_pdev.ko
[80946.192672] mem_ctl_pdev_init.
qemu-aarch64 /mnt/shared # ls /sys/bus/platform/devices/
0.flash               a001800.virtio_mmio  a003600.virtio_mmio
4010000000.pcie       a001a00.virtio_mmio  a003800.virtio_mmio
9020000.fw-cfg        a001c00.virtio_mmio  a003a00.virtio_mmio
a000000.virtio_mmio   a001e00.virtio_mmio  a003c00.virtio_mmio
a000200.virtio_mmio   a002000.virtio_mmio  a003e00.virtio_mmio
a000400.virtio_mmio   a002200.virtio_mmio  alarmtimer.0.auto
a000600.virtio_mmio   a002400.virtio_mmio  gpio-keys
a000800.virtio_mmio   a002600.virtio_mmio  mem_ctl_pdev.0
a000a00.virtio_mmio   a002800.virtio_mmio  platform@c000000
a000c00.virtio_mmio   a002a00.virtio_mmio  pmu
a000e00.virtio_mmio   a002c00.virtio_mmio  psci
a001000.virtio_mmio   a002e00.virtio_mmio  serial8250
a001200.virtio_mmio   a003000.virtio_mmio  timer
a001400.virtio_mmio   a003200.virtio_mmio
a001600.virtio_mmio   a003400.virtio_mmio
qemu-aarch64 /mnt/shared # ls -alh /dev/mem_ctl*
ls: cannot access '/dev/mem_ctl*': No such file or directory
qemu-aarch64 /mnt/shared # lsmod | grep mem_ctl
mem_ctl_pdev           16384  0
qemu-aarch64 /mnt/shared # cat /proc/devices | grep mem_ctl
qemu-aarch64 /mnt/shared #
```

再加载驱动的模块

可以看到成功匹配并打印出提示信息，同时自动创建了设备

```
qemu-aarch64 /mnt/shared # insmod mem_ctl_pdrv.ko
[  372.236169] mem_ctl_drv_init
[  372.237450] mem_ctl_probe means the match.
[  372.237644] mem_ctl_probe: res->start = 1,                    res->e
nd = 1,                       private_data = ffff
qemu-aarch64 /mnt/shared # lsmod | grep mem_ctl
mem_ctl_pdrv           16384  0
mem_ctl_pdev           16384  0
qemu-aarch64 /mnt/shared # ls /sys/bus/platform/drivers
 ARM-CCI        basic-mmio-gpio   of_fixed_clk          serial8250
'ARM-CCI PMU'   dw-apb-uart       of_fixed_factor_clk   syscon
 alarmtimer     gpio-clk          of_serial             virtio-mmio
 arm-ccn        gpio-dwapb        pci-host-generic      xgene-gpio
 armv8-pmu      mem_ctl_pdev      sbsa-uart
qemu-aarch64 /mnt/shared # ls /sys/class/
bdi     devlink  mem_ctl_class  pci_bus       rtc           spi_master  watchdog
block   gpio     misc           phy           scsi_device   tty
bsg     mem      net            power_supply  scsi_host     wakeup
qemu-aarch64 /mnt/shared # ls -alh /dev/mem_ctl*
crw-rw---- 1 root root 249, 0 Aug 14 08:47 /dev/mem_ctl_0
qemu-aarch64 /mnt/shared # cat /proc/devices | grep mem_ctl
249 mem_ctl
qemu-aarch64 /mnt/shared #
```

测试正常读写

```
qemu-aarch64 /mnt/shared # cat /dev/mem_ctl_0
qemu-aarch64 /mnt/shared # echo 'virtual character driver' > /dev/mem_ctl_0
qemu-aarch64 /mnt/shared # cat /dev/mem_ctl_0
virtual character driver
qemu-aarch64 /mnt/shared #
```

卸载驱动

首先卸载 platform 设备的模块，可以看到 设备部分调用了 `exit` 的函数，然后 platform 驱动部分执行了 `remove` 的方法，最后调用了 `release` 方法释放。

然后卸载 platform 驱动的模块。

```
qemu-aarch64 /mnt/shared # rmmod mem_ctl_pdev.ko
[  677.525614] mem_ctl_pdev_exit.
[  677.525979] mem_ctl_remove
[  677.529469] mem_ctl_pdev_release 00000000ae1fbd57.
qemu-aarch64 /mnt/shared # rmmod mem_ctl_pdrv.ko
[  718.644272] mem_ctl_drv_exit
qemu-aarch64 /mnt/shared #
```