

姓名：\_\_\_\_章崇文\_\_\_\_ 学号：\_\_\_\_202202296\_\_\_\_ 班级：\_\_\_\_计算机 222\_\_\_\_

## 实验二、嵌入式 Linux 驱动编程实验

### 一、实验目的

编写简单的虚拟硬件驱动程序并进行调试，实验驱动的各个接口函数的实现,分析并理解驱动与应用程序的交互过程。

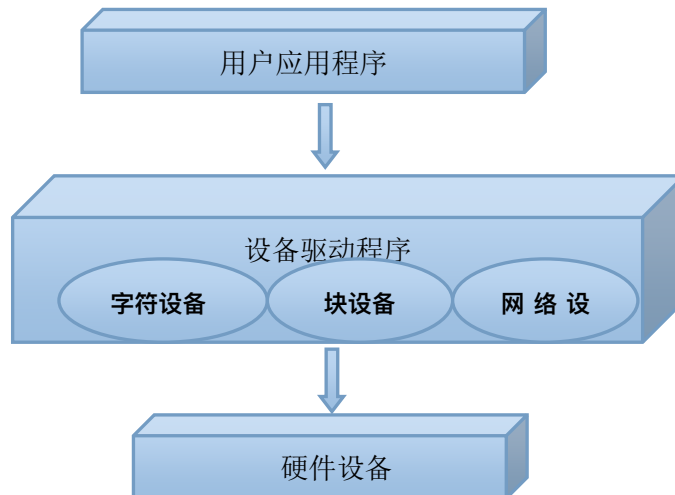
### 二、实验基本要求

1. 编写简单的虚拟硬件驱动程序并进行调试。
2. 掌握字符设备驱动程序开发的原理及步骤。
3. 编写加载驱动程序到 Ubuntu 系统或嵌入式实验箱的 Makefile 文件。

### 三、实验原理

Linux 中的驱动设计是嵌入式 Linux 开发中十分重要的部分，它要求开发者不仅要熟悉 Linux 的内核机制、驱动程序与用户级应用程序的接口关系、考虑系统中对设备的并发操作等等，而且还要非常熟悉所开发硬件的工作原理。这对驱动开发者提出了比较高的要求，这个实验主要是给大家进入驱动设计提供一个简单入门的一个实例，并不需要提供太多与硬件相关的内容，这部分应该是通过仔细阅读芯片厂家提供的资料来解决。

#### 1. Linux 设备驱动的定义



设备驱动最通俗的解释就是“驱使硬件设备行动”。用户应用程序直接对硬件操作是非常复杂的，设备驱动程序作为内核的一部分，其作用就是让驱动与底层硬件直接打交道，按照硬件设备的具体工作方式，读写设备的寄存器，完成设备的轮询、中断处理、DMA 通信，进行物理内存向虚拟内存的映射等，最终让通信设备能收发数据，让显示设备能显示文字和画面，让存

储设备能记录文件和数据。

这样设备驱动程序就是用户应用程序与硬件之间的中间软件层，用户应用程序通过间接地调用设备驱动程序来操作硬件。

另一方面，在 Linux 中，将每个硬件设备都作为一个文件对待，即每个设备有一个对应的设备文件，每个设备文件其实就是编译后的驱动程序，一旦将驱动程序以模块的方式加载到内核中，那么用户应用程序就可以通过标准的文件读写调用接口对硬件设备进行数据读取或写入。当然如果不用此设备，可以将其驱动程序从内核中卸载。

## 2. Linux 设备驱动的分类

(1) 字符设备：字符设备指那些必须以字节流的串行顺序依次进行访问的设备，如串行口、并行口、虚拟控制台（触摸屏、磁带驱动器、鼠标等）。这种设备没有缓冲区，一旦有 IO 请求，立刻发生实际的硬件 IO 操作。

(2) 块设备：块设备通常以块为单位，利用数据缓冲区进行读写，可以用以随机方式任意顺序进行访问，如硬盘、软驱等。这种设备接到 IO 请求后，只有满足一定条件时，才会发生实际的硬件 IO 操作。注：字符设备和块设备并没有明显的界限，如对于 Flash 设备，符合块设备的特点，但也支持“字节流”方式访问，所以仍然可以把它作为一个字符设备来访问。

(3) 网络设备：通常指网络设备访问的接口，如网卡等。他们由内核中网络子系统驱动，负责接受和发送数据包。

## 3. 主设备号和次设备号

设备号是一个数字 ID，它是设备的标志。设备号有主设备号和次设备号，其中主设备号表示设备类型，对应于该类型设备的驱动程序。而次设备号标识一个具体的物理设备。

传统方式中的设备管理中,除了设备类型外,内核还需要一对称作主次设备号的参数,才能唯一标识一个设备。主设备号相同的设备使用相同的驱动程序,次设备号用于区分具体设备的实例。比如 PC 机中的 IDE 设备,一般主设备号使用 3, WINDOWS 下进行的分区,一般将主分区的次设备号为 1,扩展分区的次设备号为 2、3、4,逻辑分区使用 5、6....。设备操作宏 MAJOR()和 MINOR()可分别用于获取主次设备号,宏 MKDEV()用于将主设备号和次设备号合并为设备号,这些宏定义在 include/linux/kdev\_t.h 中。

你可以使用头文件 linux/fs.h 中的函数 register\_chrdev 来实现注册一个某个驱动模块对应的主设备号。

```
int register_chrdev(unsignedint major, constchar *name, struct file_operations *fops);
```

其中 unsigned int major 是你申请的主设备号, const char \*name 是将要在文件 /proc/devices 中 struct file\_operations \*fops 是指向你的驱动模块的 file\_operations 表的指针。负的回值意味着注册失败。注意注册并不需要提供次设备号。内核本身并不在意次设备号。

register\_chrdev 一般在 int device\_init()函数中调用,即将内核驱动模块加载入内核意味着要向内核注册自己。

现在的问题是你如何申请到一个没有被使用的主设备号？最简单的方法是查看文件 Documentation/devices.txt 从中挑选一个没有被使用的。这不是一劳永逸的方法因为你无法得知该主设备号在将来会被占用。最终的方法是让内核为你动态分配一个。

如果你向函数 `register_chrdev` 传递为 0 的主设备号，那么返回的就是动态分配的主设备号。此时应该通过命令 “`cat /proc/devices`” 命令输出得到，使用 `mknod` 调用建立设备文件。

#### 4. Linux 的驱动程序编译方法

- (1) 将其源代码添加到内核源文件中，然后静态编译整个内核，再运行新的内核来测试
- (2) 编译为模块的形式，单独加载运行调试

第一种方法效率较低，但在某些场合是唯一的方法。模块方式调试效率很高，它使用 `insmod` 工具将编译的模块直接插入内核，如果出现故障，以使用 `rmmod` 从内核中卸载模块。不需要重新启动内核，这使驱动调试效率大大提高。

#### 5. 驱动程序与应用程序的区别

- (1) 应用程序一般有一个 `main` 函数，从头到尾执行一个任务；驱动程序却不同，它没有 `main` 函数，通过使用宏 `module_init`(初始化函数名)；将初始化函数加入内核全局初始化函数列表中，在内核初始化时执行驱动的初始化函数，从而完成驱动的初始化和注册，之后驱动便停止等待被应用软件调用。驱动程序中有一个宏 `module_exit`(退出处理函数名)注册退出处理函数。它在驱动退出时被调用。
- (2) 应用程序可以和 `GLIBC` 库连接，因此可以包含标准的头文件，比如 `<stdio.h>` `<stdlib.h>`，在驱动程序中是不能使用标准 C 库的，因此不能调用所有的 C 库函数，比如输出打印函数只能使用内核的 `printk` 函数，包含的头文件只能是内核的头文件，比如: `<linux/module.h>`。

#### 6. 设备驱动程序接口

结构体 `file_operations` 在头文件 `linux/fs.h` 中定义，用来存储驱动内核模块提供的对设备进行各种操作的功能接口函数的指针。该结构体的每个域都对应着驱动内核模块用来处理某个被请求的事物的函数的地址。

```
struct file_operations {
    struct module *owner;
    loff_t(*llseek) (struct file *, loff_t, int);
    ssize_t(*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t(*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
    ssize_t(*write) (struct file *, constchar __user *, size_t, loff_t *);
    ssize_t(*aio_write) (struct kiocb *, constchar __user *, size_t,
        loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsignedint (*poll) (struct file *, struct poll_table_struct *);
```

```

int (*ioctl) (struct inode *, struct file *, unsignedint,
              unsignedlong);
int (*mmap) (struct file *, struct vm_area_struct *);
int (*open) (struct inode *, struct file *);
int (*flush) (struct file *);
int (*release) (struct inode *, struct file *);
int (*fsync) (struct file *, struct dentry *, int datasync);
int (*aio_fsync) (struct kiocb *, int datasync);
int (*fasync) (int, struct file *, int);
int (*lock) (struct file *, int, struct file_lock *);
ssize_t(*readv) (struct file *, conststruct iovec *, unsignedlong,
                 loff_t *);
ssize_t(*writev) (struct file *, conststruct iovec *, unsignedlong,
                  loff_t *);
ssize_t(*sendfile) (struct file *, loff_t *, size_t, read_actor_t,
                    void __user *);
ssize_t(*sendpage) (struct file *, struct page *, int, size_t,
                    loff_t *, int);
unsignedlong (*get_unmapped_area) (struct file *, unsignedlong,
                                   unsignedlong, unsignedlong,
                                   unsignedlong);
};

```

驱动内核模块是不需要实现每个函数的。像视频卡的驱动就不需要从目录的结构中读取数据。那么，相对应的 `file_operations` 重的项就为 `NULL`。gcc 还有一个方便使用这种结构体的扩展。在较现代的驱动内核模块中见到。使用这种结构体的新的方式如下：

```

struct file_operations fops = {
    .read = device_read,
    .write = device_write,
    .open = device_open,
    .release = device_release
};

```

在用户应用程序利用系统调用对设备文件进行操作时，系统调用首先通过设备文件的主设备号找到相应的驱动程序，然后读取这个结构体相应的函数指针，借着把控制权交给驱动的这个函数。

例如：当应用程序调用 `read(设备文件名,,)` 函数时，系统会根据主设备号找到其 `struct file_operations` 中 `device_read` 功能接口函数。应用程序调用 `open()` 文件操作是通过调用 `file_operations` 结构的 `device_open` 函数接口而实现的。

## 7. 字符驱动程序编写流程

- (1) 包含字符设备驱动程序所必需包含的头文件
- (2) 定义字符设备文件名、主设备号或由系统动态分配设备号
- (3) 编写 `file_operations` 结构体中功能接口函数，如以上 `device_read`、`device_write`、`device_open`
- (4) 定义 `file_operations` 结构变量，建立用户程序调用函数与驱动功能接口的对应关系。
- (5) 编写字符设备驱动程序加载函数，完成设备文件注册及必要的初始化
- (6) 编写字符设备驱动程序卸载函数，完成注销及必要的环境恢复。
- (7) 编写 Makefile 文件，使用 `make` 来编译 Makefile，生成`****.ko` 内核模块文件
- (8) 使用 “`insmod ****.ko`” 加载设备驱动程序，并调用 `dmesg` 来查看输出结果
- (9) 使用 “`cat /proc/devices`” 查看获取主设备号
- (10) 使用 “`mknod /dev/设备文件名 c 主设备号 0`” 来创建设备文件
- (11) 编写用户应用程序，调用用户调用接口对驱动功能接口进行测试。
- (12) 使用 “`rmmmod /dev/设备文件名`” 卸载驱动模块，并调用 `dmesg` 查看

## 四、实验内容

- (1) 新建一个 `driver` 目录，将 `Dev_frame.c` 拷贝到其中。
- (2) 包含字符设备驱动程序所必需包含的头文件。
- (3) 定义字符设备文件名、主设备号或由系统动态分配设备号。
- (4) 编写 `file_operations` 结构体中功能接口函数，如以上 `device_read`、`device_write`、`device_open`
- (5) 定义 `file_operations` 结构变量，建立用户程序调用接口与驱动功能接口的对应关系。
- (6) 编写字符设备驱动程序加载函数，完成设备文件注册及必要的初始化。
- (7) 编写字符设备驱动程序卸载函数，完成注销及必要的环境恢复。
- (8) 在 `driver` 目录下编写 Makefile 文件。
- (9) 使用 `make` 来编译 Makefile，生成`****.ko` 内核模块文件。
- (10) 使用 “`insmod ****.ko`” 加载设备驱动程序，并调用 `dmesg` 来查看内核驱动输出结果。
- (11) 使用 “`cat /proc/devices`” 查看获取主设备号。
- (12) 使用 “`mknod /dev/设备文件名 c 主设备号 0`” 在目录`/dev` 下创建设备文件
- (13) 编写用户应用程序 `m_test.c`，调用标准的文件操作 `open()`、`read()`、`write()` 等对驱动功能接口进行测试。并 `dmesg` 来查看内核驱动输出结果。
- (14) 使用 “`rmmmod /dev/设备文件名`” 卸载驱动模块，并调用 `dmesg` 查看

## 五、实验结果及分析：

### 1. 请简述设备驱动程序的作用

设备驱动程序的作用是**充当操作系统与硬件设备之间的桥梁**，实现用户程序对硬件的间接访问。它的核心功能可以概括为以下几点：

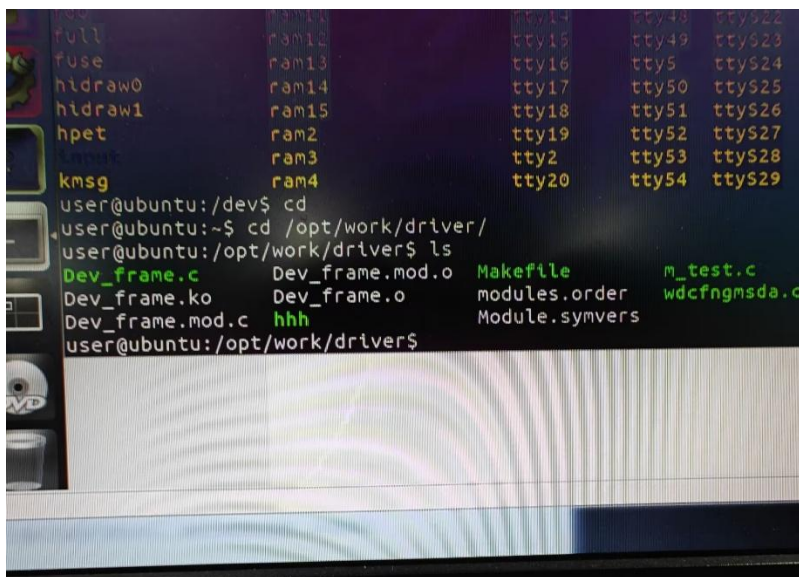
- **屏蔽硬件细节**：用户应用程序不需要关心硬件设备的寄存器、通信方式等底层细节，只需要通过文件操作接口（如 `open`、`read`、`write`）访问设备。
- **提供统一接口**：在 Linux 中，设备以文件形式存在（如 `/dev/mydevice`），用户通过标准系统调用访问设备文件，驱动程序接收这些调用并与硬件交互。
- **管理设备资源**：驱动程序负责设备的初始化、数据传输、中断处理等，包括 DMA、内存映射等资源管理工作。
- **实现设备功能接口**：通过实现 `file_operations` 结构体中的函数，如 `device_open()`、`device_read()`、`device_write()` 等，驱动提供与设备交互的具体方式。
- **提高系统模块化与可维护性**：驱动模块可以独立开发、编译并通过 `insmod` 和 `rmmod` 动态加载与卸载，提升了开发与调试的灵活性。

简而言之，设备驱动程序**使操作系统可以以标准的方式控制各种不同的硬件设备**，**同时将复杂的硬件操作对用户进行封装和抽象**，是嵌入式 Linux 系统开发中至关重要的一环。

### 2. 写出驱动开发的一般步骤。

#### 1. 创建驱动源文件目录并准备代码文件

- 新建一个如 `driver/` 的目录，将驱动源文件（如 `Dev_frame.c`）拷贝或创建在其中。



```
user@ubuntu:~$ cd /opt/work/driver/
user@ubuntu:~/opt/work/driver$ ls
Dev_frame.c      Dev_frame.mod.o  Makefile         m_test.c
Dev_frame.ko     Dev_frame.o      modules.order    wdcfngmsda.c
Dev_frame.mod.c  hhh              Module.symvers
user@ubuntu:~/opt/work/driver$
```

#### 2. 包含必要的头文件



- 如：#include <linux/module.h>、#include <linux/fs.h>、#include <linux/uaccess.h> 等。

### 3. 定义设备名称和设备号

- 定义字符设备名称，如 #define DEVICE\_NAME "mydev"
- 主设备号可以手动指定，也可以使用 register\_chrdev(0, ...); 动态申请。

```

#include <linux/module.h>
#include <linux/fs.h>
#include <linux/uaccess.h>

#define DEVICE_NAME "Dev_frame"
int DEVICE_MAJOR = 0;

#define M_LENGTH 10
char core_data0[M_LENGTH][M_LENGTH];
char core_data1[M_LENGTH][M_LENGTH];

int device_open(struct inode *inode, struct file *file)
{
    char c = 'a';
    int i, j;
    printk("user open device %s\n", DEVICE_NAME);
}

```

### 4. 实现 file\_operations 接口函数

- 如 device\_open()、device\_read()、device\_write()、device\_release() 等。

### 5. 定义并初始化 file\_operations 结构体

```

6. struct file_operations fops = {
7.     .open = device_open,
8.     .read = device_read,
9.     .write = device_write,
10.    .release = device_release,
11. };

```

### 12. 编写模块加载函数 (init 函数)

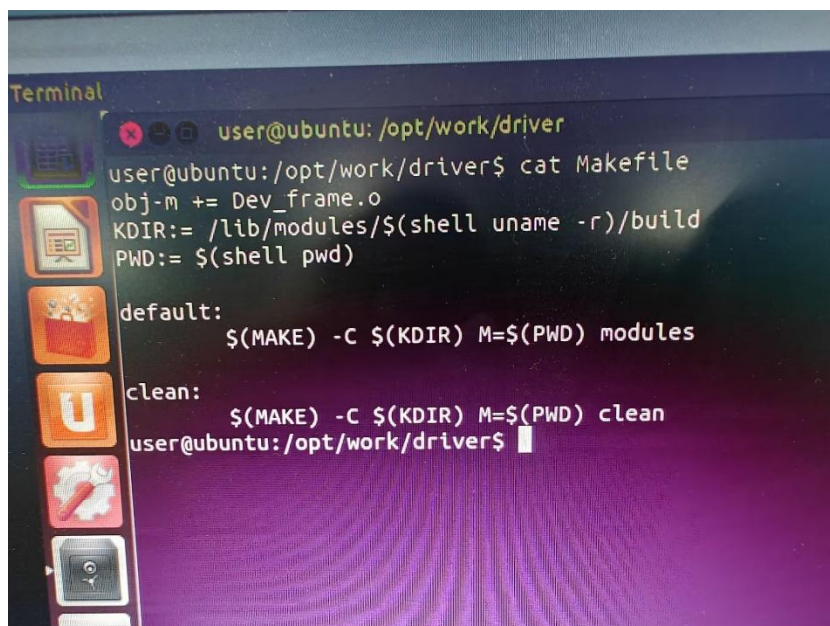
- 使用 register\_chrdev() 注册设备，完成必要的初始化。
- 使用 module\_init(mydev\_init); 注册加载函数。

### 13. 编写模块卸载函数 (exit 函数)

- 使用 unregister\_chrdev() 注销设备，释放资源。
- 使用 module\_exit(mydev\_exit); 注册卸载函数。

### 14. 编写 Makefile 文件

- 编写简单的 Makefile，用于将驱动编译为内核模块 (.ko 文件)。



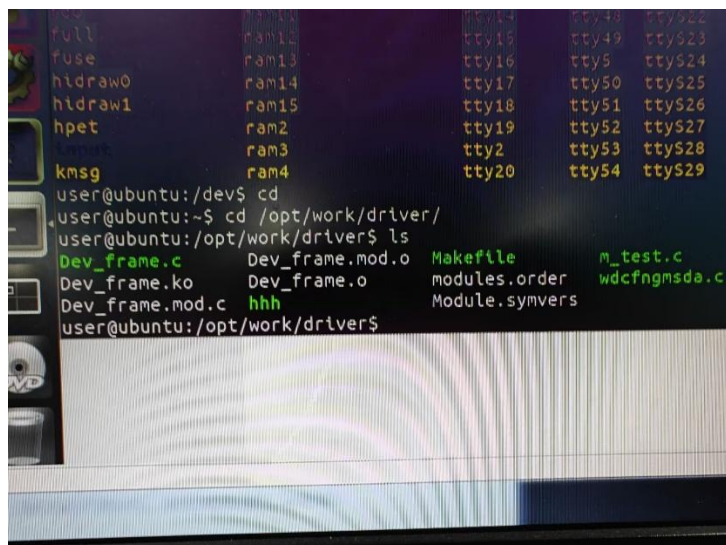
```
Terminal
user@ubuntu: /opt/work/driver
user@ubuntu:/opt/work/driver$ cat Makefile
obj-m += Dev_frame.o
KDIR:= /lib/modules/$(shell uname -r)/build
PWD:= $(shell pwd)

default:
    $(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
    $(MAKE) -C $(KDIR) M=$(PWD) clean
user@ubuntu:/opt/work/driver$
```

## 15. 使用 make 命令编译模块

- 执行 make 生成 .ko 模块文件。



```
user@ubuntu:/dev$ cd
user@ubuntu:~$ cd /opt/work/driver/
user@ubuntu:/opt/work/driver$ ls
Dev_frame.c      Dev_frame.mod.o  Makefile         m_test.c
Dev_frame.ko     Dev_frame.o      modules.order    wdcfngmsda.c
Dev_frame.mod.c  hhh             Module.symvers
user@ubuntu:/opt/work/driver$
```

## 16. 加载驱动模块并测试

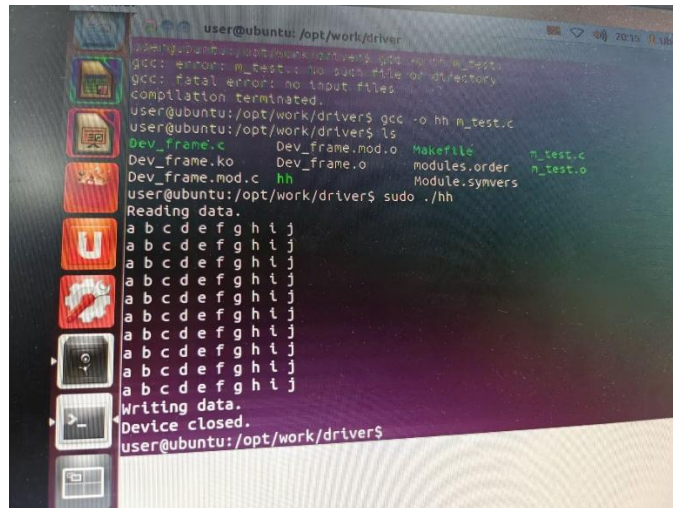
- 使用 insmod xxx.ko 加载模块
- 查看 dmesg 输出是否成功
- 使用 cat /proc/devices 查看分配的主设备号
- 使用 mknod 创建设备文件:
- Sudo mknod /dev/mydev c 主设备号 0





### 17. 编写用户测试程序

- 编写测试程序 m\_test.c, 使用 open()、read()、write() 等标准文件操作测试驱动功能。



### 18. 卸载驱动模块

- 使用 rmmod xxx 卸载模块
- 再次查看 dmesg 确认卸载信息

```
2566.135815- user read data from driver.
2566.135816 user write data to driver.
2566.135818 aaaaaaaaaa
2566.135817 bbbbbbbbbb
2566.135819 cccccccccc
2566.135821 dddddddddd
2566.135823 eeeeeeeeee
2566.135825 ffffffffff
2566.135827 gggggggggg
2566.135829 hhhhhhhhhh
2566.135831 iiiiiiii
2566.135832 jjjjjjjj
2566.135888 device release I
2587.907914 unregister chrdev ok!
user@ubuntu:/opt/work/driver$
```

3. 写出在 Ubuntu 编译驱动程序的 makefile 文件。

```
嵌入式系统原理与应用 > chaoxing > Experiment 2 > Makefile
1 # 要编译的内核模块对象
2 MODULE_NAME := Dev_frame
3 obj-m += $(MODULE_NAME).o
4
5 # 内核源代码目录
6 KDIR := /lib/modules/$(shell uname -r)/build
7 # 当前工作目录
8 PWD := $(shell pwd)
9
10 # 默认目标: 编译内核模块
11 default:
12     @echo "开始编译内核模块..."
13     $(MAKE) -C $(KDIR) M=$(PWD) modules || { echo "编译内核模块失败! "; exit 1; }
14     @echo "内核模块编译成功! "
15
16 # 清理目标: 清理编译生成的文件
17 clean:
18     @echo "开始清理编译生成的文件..."
19     $(MAKE) -C $(KDIR) M=$(PWD) clean || { echo "清理编译生成的文件失败! "; exit 1; }
20     @echo "编译生成的文件清理成功! "
```

附录: Dev\_frame.c 部分代码

```
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <asm/uaccess.h>

#define DEVICE_NAME "Dev_frame"
```

```
int DEVICE_MAJOR = 0;
```

```
#define M_LENGTH 10
```

```
char core_data0[M_LENGTH][M_LENGTH];
```

```
char core_data1[M_LENGTH][M_LENGTH];
```

```
int device_open(struct inode *inode, struct file *filp)
```

```
{
    char c = 'a';
    int i, j;
    printk("user open device.\n");
    for (i = 0; i < M_LENGTH; i++)
    {
        for (j = 0; j < M_LENGTH; j++)
        {
            core_data0[i][j] = c + j;
        }
    }
    return 0;
}
```

```
ssize_t device_write(struct file *filp, const char *buffer, size_t count, loff_t *f_pos)
```

```
{
    int i, j;
    printk("user write data to driver.\n");
    copy_from_user(core_data1, buffer, sizeof(core_data1));
    for (i = 0; i < M_LENGTH; i++)
    {
        for (j = 0; j < M_LENGTH; j++)
        {
            printk("%c", core_data1[i][j]);
        }
        printk("\n");
    }
    return count;
}
```

```
ssize_t device_read(struct file *filp, char *buffer, size_t count, loff_t *f_pos)
```

```
{
    printk("user read data from driver.\n");
    copy_to_user(buffer, core_data0, sizeof(core_data0));
}
```

```

    return count;
}

```

```

int device_ioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg)
{
    printk("user ioctl running.\n");
    switch (cmd)
    {
        case 1:
            printk("cmd = %d.\n", cmd);
            break;
        case 2:
            printk("cmd = %d.\n", cmd);
            break;
        case 3:
            printk("cmd = %d.\n", cmd);
            break;
        case 4:
            printk("cmd = %d.\n", cmd);
            break;
        case 5:
            printk("cmd = %d.\n", cmd);
            break;
    }
    return cmd;
}

```

```

int device_release(struct inode *inode, struct file *filp)
{
    printk("device release\n");
    return 0;
}

```

```

struct file_operations device_fops = {
    .owner = THIS_MODULE,
    .read = device_read,
    .open = device_open,
    .write = device_write,
    .release = device_release,
    // .ioctl = device_ioctl, // 根据需求决定是否取消注释
};

```

```
int device_init(void)
{
    int ret;
    ret = register_chrdev(0, DEVICE_NAME, &device_fops);
    if (ret < 0)
    {
        printk("register chrdev failure!\n");
        return ret;
    }
    else
    {
        printk("register chrdev ok!\n");
        DEVICE_MAJOR = ret;
    }
    return 0;
}
```

```
void device_exit(void)
{
    unregister_chrdev(DEVICE_MAJOR, DEVICE_NAME);
    printk("unregister chrdev ok!\n");
}
```

```
module_init(device_init);
module_exit(device_exit);
```

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