浙江大学

本科实验报告

姓名：包博文

学院：生物医学工程与仪器科学学院

系：生物医学工程系

专业：生物医学工程

学号：3220105972

指导教师：周凡

2024 年 11 月 30 日

浙江大学实验报告

课程名称：嵌入式系统 实验类型：

实验项目名称：Linux 字符驱动开发

学生姓名：包博文 专业：生物医学工程 学号：3220105972

同组学生姓名：

指导老师：刘雪松

实验地点：创客空间 实验日期：2024 年 11 月 30 日

一、实验目的和要求

本实验通过编写 Linux 字符设备驱动程序，熟悉 Linux 内核模块开发流程及字符设备操作原理，具体目标包括：

掌握字符设备驱动程序的编写方法，实现超声波传感器的控制，完成距离测量。

了解字符设备与用户空间应用程序的交互，使用用户空间应用程序通过驱动与硬件交互。

学习 GPIO 相关函数及其在驱动程序中的应用，掌握模块加载和卸载命令及相关内核日志的调试方法。

二、实验内容和原理

1、什么是设备驱动程序？

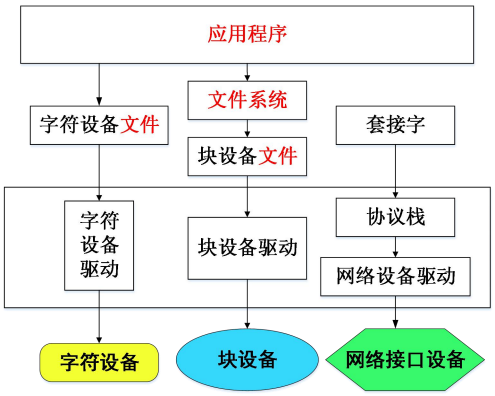
应用程序与实际设备的软件层，是 Linux 内核的一部分，是某个特定硬件响应一个定义良好的内部编程接口，该接口完全隐藏设备的工作细节。

2、设备驱动程序的作用

设备驱动程序处理如何使硬件可用的问题，而将如何使用硬件留给上层应用程序。也就是说设备驱动程序提供机制，不提供策略。

3、设备驱动调用过程

Linux 操作系统下，应用程序使用各设备过程如下图：针对不同的设备类别，用户空间的应用程序通过不同的系统调用接口调用到不同的设备驱动程序，从而控制硬件设备。



4、Linux 设备分类

Linux 系统将设备分成三种基本类型：字符设备、块设备、网络接口。相对应的，设备驱动程序也可分为字符设备驱动程序、块设备驱动程序、网络接口驱动程序。

字符设备：字符(char)设备是能够像字节流（类似文件）一样被访问的设备。字符设备可以通过文件系统节点来访问，该节点位于/dev 目录下。这些设备文件和普通文件之间的差别在于普通文件的访问可以前后移动访问位置，而大多数字符设备只能顺序呢访问。常见的字符设备有鼠标、键盘、串口、控制台和 LED 设备等。

块设备：是指可以从设备的任意位置读取一定长度数据的设备。块设备包括硬盘、磁盘、U 盘和 SD 卡等。同样可以通过文件系统节点来访问，节点位于/dev 目录下。

网络接口：任何网络事物都经过一个网络接口形成，即一个能够和其他主机交换数据的设备。由于网络设备不是面向流的设备将网络接口映射到文件系统中的节点比较困难。因此，内核访问网络接口的方法是给它们分配一个唯一的名字（如 eth0），但这个名字在文件系统内不存在对应的节点。

5、可装载模块

Linux 系统提供一个特性：内核提供的特性可以在运行时进行扩展。也就是说当系统启动并运行时，我们仍可以向内核添加功能（或移除功能）。

可在运行时添加到内核中的代码被“模块”。Linux 内核支持多种模块类型，包括但不限于设备驱动程序。每个模块由目标代码组成（没有连接成一个完成的可执行程序），可以通过命令

insmod　模块名

将模块连接到正在运行的内核。

使用命令

rmmod　模块名

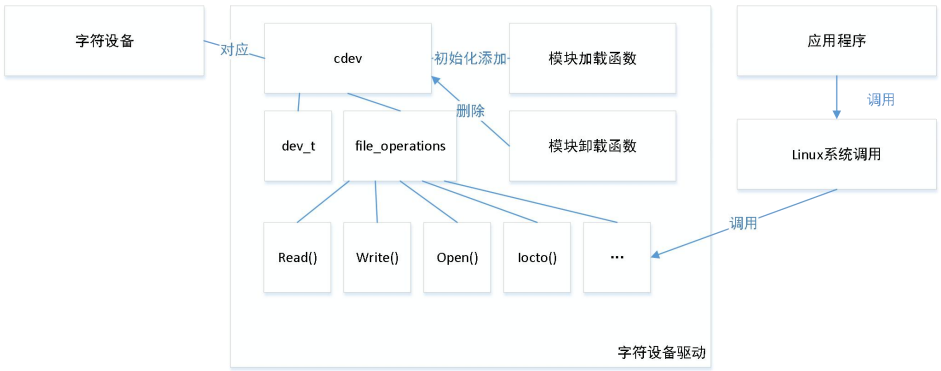
将模块移除。

可装载模块源代码的基本组成如下：

头文件	#include<linux/init.h> #include<linux/module.h>	必选
许可声明	MODULE_LICENSE(“GPL”)	必选
加载函数	static int __init init_function(void)	必选
卸载函数	static void __exit exit_function(void)	必选
模块参数	module_param(name,type,attribute)	可选
模块导出符号	EXPORT_SYMBOL(name)	可选
模块作者等信息声明	MODULE_AUTHOR(“author name”)	可选

6、字符设备、字符设备驱动与用户空间访问该设备的程序三者之间的关系

三者之间关系如下图：



如上图所示，在 Linux 内核中使用 cdev 结构体来描述字符设备，通过其成员 dev_t 来定义**设备号**（分为主、次设备号）以确定字符设备的唯一性。通过其成员 file_operations 来定义字符设备驱动提供给 VFS 的接口函数，如常见的

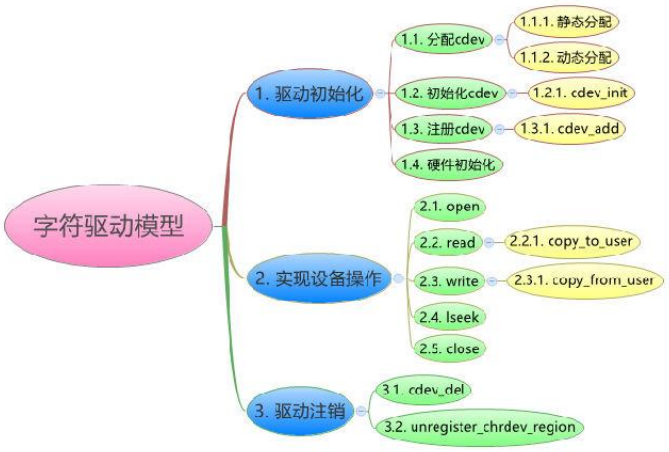
open()、read()、write()等。

在 Linux 字符设备驱动中，模块加载函数通过 register_chrdev_region( ) 或 alloc_chrdev_region( )来静态或者动态获取设备号，通过 cdev_init( )建立 cdev 与 file_operations 之间的连接,通过 cdev_add()向系统添加一个 cdev 以完成注册。模块卸载函数通过 cdev_del( )来注销 cdev，通过 unregister_chrdev_region( )来释放设备号

用户空间访问该设备的程序通过 Linux 系统调用，如 open( )、read( )、write( )，来“调用”file_operations 来定义字符设备驱动提供给 VFS 的接口函数。

7、Linux 字符设备驱动编写

Linux 字符设备驱动编写主要需完成如下工作



（1）分配 cdev

在 Linux 内核中使用 cdev 结构体来描述字符设备，在驱动中分配 cdev,主要是分配一个 cdev 结构体与申请设备号。

分配结构体

```
struct cdev cdev_name;
```

申请设备号

设备号有主次之分，在 Linux 中以主设备好表示与设备文件相连接的驱动。次设备号用来驱动操作的是哪一个设备。申请设备号存在两种方法，一种是静态申请，也就是说手动找到一个还未使用的主设备号，然后生成设备号，并注册。另一种是动态申请设备号，然后从设备号中获取主设备号，注册字符设备驱动程

序。两种方式分别如下：

静态申请设备号：

```
dev_id  = MKDEV(major,0);
register_chrdev_region(dev_id,1,”device_name”);
```

动态申请设备号：

```
alloc_chardev_region(&dev_id,0,1,”device_name”);
major = MAJOR(dev_id)
```

（2）初始化 cdev

```
void cdev_init(struct cdev* ,struct file_operations *);
```

cdev_init()函数用于初始化 cdev 的成员，并建立 cdev 和 file_oprations 之间的联系。

（3） 注册 cdev

```
int  cdev_add (struct cdev *, dev_t, unsigned);
```

cdev_add（）函数向系统添加一个 cdev，完成字符设备的注册

（4） 实现设备操作

用户空间的程序以访问文件的形式访问字符设备，通常进行 open，read，write，close 等系统调用。而这些系统调用最终调用的是 file_operations 结构体中的成员函数。它们是字符设备驱动与内核的接口。对应的系统调用会调用到结构体中对应名称的成员函数。

```
static struct file_operaions fops_name = {
        .owner = THIS_MODULE,
        .open   = open_name,
        .close  =  close_name,
       …
}
```

（5）注销驱动

当使用 rmmod 命令时，会调用驱动的退出函数，该函数中需要完成在清理注册过程中使用的资源，删除字符设备，释放设备号等。

也就是说，初始化函数与退出函数是一个反向操作。在初始化使用的资

源必须在退出函数中释放。

删除字符设备

```
void cdev_del(struct cdev *);
```

释放设备号

```
void unregister_chrdev_region(dev_t from, unsigned count);
```

8、Linux 内核中 gpio 相关函数

（1）**gpio_set_value(unsigned gpio, int value)**用来设置 gpio 寄存器的值；

（2）**gpio_direction_output(unsigned gpio, int value)**用来设置 gpio 为输出功能，同时设置 gpio 输出的值。

一般来说，设置一个 GPIO 口为输出，先执行一次 gpio_direction_output，然后接下来只需执行 gpio_set_value 就行了。

（3）**gpio_direction_input(unsigned gpio)**用来设置 gpio 为输入功能

（4）**gpio_get_value(unsigned gpio)**用来获取 gpio 口的输入的值；

（5）在使用 gpio 口之前，先用 **gpio_request(unsigned gpio, const char* label)**申请 gpio 口的使用，若申请成功，则说明该 gpio 口未被使用。

（6）**gpio_request_one(unsigned gpio, unsigned long flags, const char *label)** ，request a single GPIO with initial configuration，flags 包括 GPIOF_DIR_IN，GPIOF_DIR_OUT，GPIOF_INIT_LOW，GPIOF_INIT_HIGH，GPIOF_IN，GPIOF_OUT_INIT_LOW，GPIOF_OUT_INIT_HIGH

（7）**gpio_to_irq(unsigned gpio)** 将一个 GPIO 映射为中断，返回的值即中断编号

（8）在使用完 gpio 口之后，用 **gpio_free(unsigned gpio)**释放 gpio 口。

9、Linux 内核中时间相关函数

```
udelay(n)          mdelay(n)
ndelay(n)          do_gettimeofday(struct timeval *tv)
```

10、内核态与用户态交互函数

从用户态传递到内核态：

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

从内核态传递到用户态：

copy_to_user(void __user *to, const void *from, unsigned long n);

11、内核中中断相关函数

enable_irq(unsigned int irq)使能指定的中断

disable_irq(unsigned int irq)禁止指定的中断

request_irq(unsigned int irq,irq_handler_t handler,unsigned long fl
ags,const char *name , void *dev) 注册中断处理的函数

free_irq(unsigned int irq,void *dev_id)注销相应的中断处理程序

12、树莓派与虚拟机 ubuntu 之间的文件传输

scp 命令用于 Linux 之间复制文件和目录。scp 是 secure copy 的缩写，scp 是 linux 系统下基于 ssh 登陆进行安全的远程文件拷贝命令。语法如下：

scp [-1246BCpqrv] [-c cipher] [-F ssh_config] [-i identity_file]
[-l limit] [-o ssh_option] [-P port] [-S program]
[[user@]host1:]file1 [...] [[user@]host2:]file2

scp 可从本地复制到远程，也可从远程复制到本地

从本地复制到远程：

scp local_file remote_username@remote_ip:remote_folder

scp -r local_folder remote_username@remote_ip:remote_folder

从远程复制到本地：

scp remote_username@remote_ip:remote_folder local_file

scp -r remote_username@remote_ip:remote_folder local_folder

应用实例:从虚拟机 ubuntu 复制文件到树莓派：

scp /home/test pi@192.168.137.250:/home/pi

13、超声波传感器简介

接口定义：Vcc(5V)、 Trig（控制端）、 Echo（接收端）、 Gnd

工作原理：

（1）采用 IO 触发测距，给至少 10us 的高电平信号；

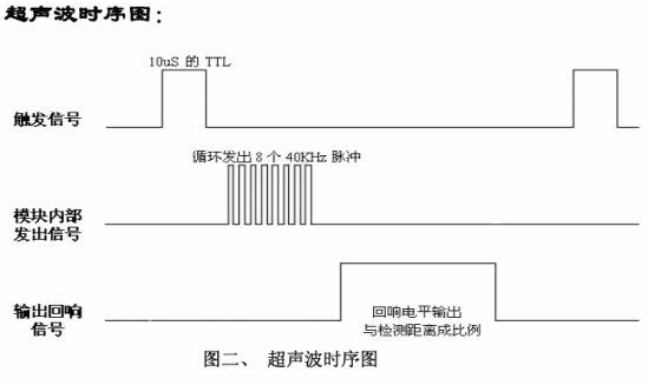
（2）模块自动发送 8 个 40khz 的方波，自动检测是否有信号返回；

（3）有信号返回，通过 IO 输出一高电平，高电平持续的时间就是超声

波从发射到返回的时间. 测试距离=(高电平时间*声速(340m/s))/2;

使用方法：控制端发一个 10us 以上的高电平,在接收端等待高电平输出.一有输出就可以进行计时,当此口变为低电平时停止计时,此时间差就为此次测距的时间,可根据时间算出距离。

注意事项：此模块不宜带电连接，如果要带电连接，则先让模块的 Gnd 端先连接。否则会影响模块工作。



图二、 超声波时序图

三、主要仪器设备

树莓派开发板

超声波传感器

杜邦线×4

配备 Ubuntu 系统的 PC（WSL2）

四、操作方法和实验步骤

1、将树莓派的环境还原到实验二

2、编译运行所给样例驱动程序（通过按键控制 LED 点亮），验证实验现象，熟悉驱动开发过程。编译时注意修改 Makefile 中的相关路径

（1）将编译得到的可执行文件 demo 和模块 demo.ko 放入树莓派 SD 卡（用 scp 命令拷贝）

（2）运行树莓派，用串口或 SSH 登录到树莓派，使用 insmod 命令加载模块(sudo insmod demo.ko)

（3）将按键与 LED 连接到树莓派，运行 sudo ./demo 观察现象

3、修改样例驱动程序，实现超声波传感器控制。

测试距离=（高电平时间*声速（340M/S））/2

五、实验数据记录和处理

```
bubblevan@Bubbles:~/lab2_zju/code$ cd sensor/app
bubblevan@Bubbles:~/lab2_zju/code/sensor/app$ make
/home/bubblevan/lab2_zju/gcc-linaro-7.5.0-2019.12-x86_64_aarch64-linux-gnu/bin/aarch64-linux-gnu-gcc    -c -o demo.o dem
o.c
/home/bubblevan/lab2_zju/gcc-linaro-7.5.0-2019.12-x86_64_aarch64-linux-gnu/bin/aarch64-linux-gnu-gcc -o demo demo.o -sta
tic
bubblevan@Bubbles:~/lab2_zju/code/sensor/app$ cd ../driver
bubblevan@Bubbles:~/lab2_zju/code/sensor/driver$ make
make CROSS_COMPILE=/home/bubblevan/lab2_zju/gcc-linaro-7.5.0-2019.12-x86_64_aarch64-linux-gnu/bin/aarch64-linux-gnu- ARC
H=arm64 -C /home/bubblevan/lab2_zju/linux M=/home/bubblevan/lab2_zju/code/sensor/driver modules
make[1]: Entering directory '/home/bubblevan/lab2_zju/linux'
  CC [M]  /home/bubblevan/lab2_zju/code/sensor/driver/demo.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/bubblevan/lab2_zju/code/sensor/driver/demo.mod.o
  LD [M]  /home/bubblevan/lab2_zju/code/sensor/driver/demo.ko
make[1]: Leaving directory '/home/bubblevan/lab2_zju/linux'
```

```
bubblevan@Bubbles:~/lab2_zju/code/sensor/driver$ scp /home/bubblevan/lab2_zju/code/sensor/driver/demo.ko bubblevan@192.1
68.43.36:/home/bubblevan/
bubblevan@192.168.43.36's password:
demo.ko                                                                   100%   12KB 764.3KB/s   00:00
bubblevan@Bubbles:~/lab2_zju/code/sensor/driver$ scp /home/bubblevan/lab2_zju/code/sensor/app/demo bubblevan@192.168.43.
36:/home/bubblevan/
bubblevan@192.168.43.36's password:
demo                                                                      100% 4783KB   2.7MB/s   00:01
```

安装完成之后却出现了问题：

```
bubblevan@raspberrypi:~ $ sudo insmod /home/bubblevan/demo.ko
bubblevan@raspberrypi:~ $ sudo ./demo
device init done
distance 0 cm
distance 0 cm
distance 0 cm
distance 0 cm
distance 0 cm
distance 0 cm
distance 0 cm
^C
```

被迫删除内核重新查看代码，甚至怀疑是不是 10us 设的太短了。

```
bubblevan@raspberrypi:~ $ lsmod | grep demo
demo                   16384  0
bubblevan@raspberrypi:~ $ sudo rmmod demo
bubblevan@raspberrypi:~ $ lsmod | grep demo
bubblevan@raspberrypi:~ $ ls
Bookshelf  demo  demo.ko  Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos
bubblevan@raspberrypi:~ $ rm demo demo.ko
```

第二次安装，加上 printk 后通过 dmesg 查看日志

```
bubblevan@raspberrypi:~ $ sudo insmod /home/bubblevan/demo.ko
bubblevan@raspberrypi:~ $ sudo ./demo
device init done
distance 0 cm
distance 0 cm
distance 0 cm
distance 0 cm
distance 0 cm
distance 0 cm
distance 0 cm
distance 0 cm
^C
bubblevan@raspberrypi:~ $ dmesg | tail
[ 3894.162911] [demo_ioctl 144]:Invalid cmd: 100
[ 3895.163078] demo_ioctl 1
[ 3895.163093] [demo_ioctl 144]:Invalid cmd: 1
[ 3897.163287] demo_ioctl 100
[ 3897.163294] [demo_ioctl 144]:Invalid cmd: 100
[ 3898.163446] demo_ioctl 1
[ 3898.163454] [demo_ioctl 144]:Invalid cmd: 1
[ 3900.163608] demo_ioctl 100
[ 3900.163615] [demo_ioctl 144]:Invalid cmd: 100
[ 3900.766460] Demo release
```

原来是我修改了 driver/demo.c 的魔数，却忘记把 app/demo.c 的魔数修改了

第三次安装成功

```
bubblevan@raspberrypi:~ $ sudo rmmod demo
bubblevan@raspberrypi:~ $ rm demo demo.ko
bubblevan@raspberrypi:~ $ sudo insmod /home/bubblevan/demo.ko
bubblevan@raspberrypi:~ $ sudo ./demo
device init done
distance 4 cm
distance 9 cm
distance 6 cm
distance 6 cm
distance 16 cm
distance 13 cm
distance 9 cm
distance 23 cm
distance 18 cm
distance 17 cm
distance 17 cm
distance 10 cm
distance 10 cm
```

六、实验结果与分析

在本实验中，app 和 driver 分别是用户空间和内核空间的两个程序模块：

app（用户空间应用程序）：

从用户角度与设备交互负责。

它通过文件操作（如 open、ioctl、read、write 等）与字符设备 ioctl 驱动交互。readwrite

app/demo.c 用于从/dev/Demo 设备文件访问字符设备，触发探针操作，并获取距离测量结果。

app 的 Makefile 用于编译用户空间程序，生成可执行文件；因为用户空间程序运行在 ARM 平台上，Makefile 需要指定交叉编译器（如 arm-linux-gnueabihf-gcc）。

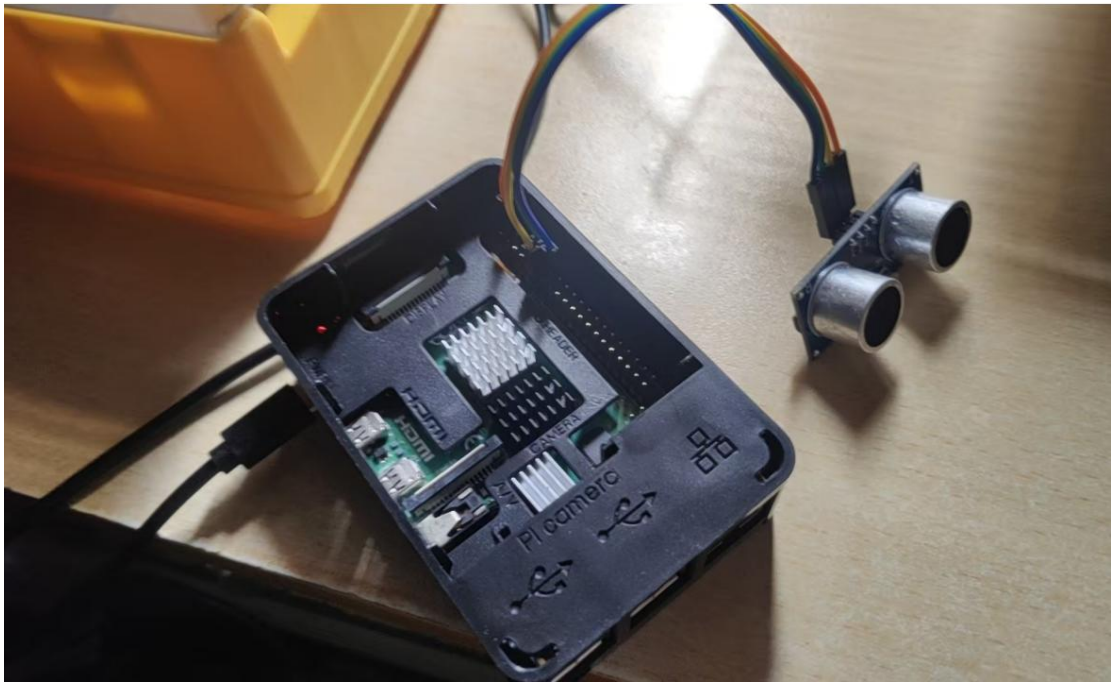
driver（内核空间设备驱动）：

负责直接控制硬件设备（如 GPIO）和处理底层操作（如中断处理）。

提供一个统一的接口（通过/dev/Demo）供用户空间程序访问。

driver/demo.c 是用于控制 GPIO 并实现探针传感器功能的字符设备驱动。

驱动程序的 Makefile 用于编译设备驱动程序，生成.ko 内核模块文件;其通常包含编译器源码路径（KDIR）和交叉编译工具链路径。



七、讨论、心得

在本实验中，我们实现了基于树莓派的超声波传感器控制，通过编写 Linux 字符设备驱动和用户空间应用程序，实现了距离测量的功能。最开始使用 irq_handler 时，未正确实现对回波信号的上升沿和下降沿的时间记录，随后使用 do_gettimeofday 精确记录时间差，并根据超声波传播公式计算距离。然后，在驱动和应用程序中使用了不一致的命令魔数(CMD_INIT, CMD_TRIG, CMD_READ)，导致 ioctl 调用无法正确执行，可谓命途多舛。

当然最后，还是成功让我更深入地理解了 Linux 字符设备驱动的开发流程，并学会了如何将用户空间应用程序与内核空间驱动程序结合起来。实验完成后，我对 Linux 内核模块的加载和卸载机制、GPIO 的使用方法以及超声波传感器的工作原理有了更清晰的理解。

八、代码附件

app/demo.c

```c
// sensor/app/demo.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/time.h>
#include <sys/ioctl.h>
#include <errno.h>
#include <fcntl.h>

// 定义命令魔数
#define CMD_INIT 0x100   // 初始化传感器
#define CMD_TRIG 0x101   // 触发超声波传感器
#define CMD_READ 0x200   // 读取传感器数据
// TODO: change the gpios
// #define GPIO_TRIG 2
// #define GPIO_ECHO 3
```

```c
#define GPIO_TRIG 17
#define GPIO_ECHO 18
struct sensor_config
{
    int trig;
```

```
    int echo;
    int output;
};
int main(int argc, char **argv)
{
    int fd;
    struct sensor_config config;
    config.trig = GPIO_TRIG;
    config.echo = GPIO_ECHO;
    fd = open("/dev/Demo", O_RDWR);
    if (fd < 0) {
        perror("/dev/Demo");
        exit(0);
    }
    ioctl(fd, CMD_INIT, &config); // 初始化传感器
    printf("device init done\n");
    while(1) {
        ioctl(fd, CMD_TRIG, &config); // 触发测量
        sleep(2);
        ioctl(fd, CMD_READ, &config); // 获取测量值
        printf("distance %d cm\n", config.output);
        sleep(1);
    }
    close(fd);
    return 0;
}
```

app/Makefile

```
CCC = /home/bubblevan/lab2_zju/gcc-linaro-7.5.0-2019.12-x86_64_aarch64-linux-gnu/bin/aarch64-linux-gnu-gcc

all: demo
demo: demo.o
    $(CC) -o $@ $^ -static
clean:
    -rm *.o demo
```

driver/demo.c

```
// sensor/driver/demo.c
#include <linux/init.h>
#include <linux/module.h>
```

```c
#include <linux/device.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/cdev.h>
#include <linux/ioctl.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <linux/uaccess.h>
#include <linux/time.h>
#include <linux/delay.h>

#define DRIVER_NAME "Demo"
#define DEVICE_NAME "Demo"
// 定义命令魔数
#define CMD_INIT 0x100   // 初始化传感器
#define CMD_TRIG 0x101   // 触发超声波传感器
#define CMD_READ 0x200   // 读取传感器数据
static dev_t demo_devno; // device number
static struct class *demo_class;
static struct cdev demo_dev; // char device struct
static int flag = 0;
static struct sensor_config {
    int trig; // control end gpio
    int echo; // receive end gpio
    int output; // output
} config;
static int distance = 0;
// 中断处理函数
static irqreturn_t irq_handler(int irq, void* dev) {
    static struct timeval start_time, end_time;
    static int start_flag = 0;
    unsigned long time_diff;
    if (flag) {
        // TODO: The echo GPIO receives a high level, measures the time and calculates the distance
        // The result is stored in the "distance" variable.
        // Hint: do_gettimeofday(struct timeval* tv)
        if (gpio_get_value(config.echo) == 1) {
            // Rising edge (开始接收回波)
            do_gettimeofday(&start_time);
            start_flag = 1;
            printk(KERN_INFO "Echo rising edge detected\n");
        } else {
            // Falling edge (结束接收回波)
            do_gettimeofday(&end_time);
```

```
            start_flag = 0;
            // 计算时间差（单位：微秒）
            time_diff = (end_time.tv_sec - start_time.tv_sec) * 1000000 + (end_time.tv_usec -
start_time.tv_usec);
            // 计算距离，超声波的传播速度大约是 343 米/秒（34300 厘米/秒）
            distance = time_diff * 34300 / 2000000; // 将时间差转换为厘米
            printk(KERN_INFO "Distance: %d cm\n", distance);
        }
        // end for TODO
    }
    return IRQ_HANDLED;
}
// The open function is called when the application calls the open system call.
static int demo_open(struct inode *inode, struct file *filp) {
    printk(KERN_INFO"Demo open\n");
    return 0;
}
// The release function is called when the application calls the close system call
static int demo_release(struct inode *inode, struct file *flip) {
    if (flag) {
        free_irq(gpio_to_irq(config.echo), NULL);
        gpio_free(config.trig);
        gpio_free(config.echo);
    }
    flag = 0;
    printk(KERN_INFO"Demo release\n");
    return 0;
}
static long demo_ioctl(struct file *filp, unsigned int cmd, unsigned long arg) {
    printk(KERN_INFO"demo_ioctl %d\n", cmd);

    switch (cmd) {
        case CMD_INIT: { // Init the sensor
            // Step1: obtain configuration data from the user program
            //TODO: copy_from_user
            if(copy_from_user(&config, (void*) arg, sizeof(struct sensor_config))){
                return -EFAULT;
            }
            // Step2: request GPIOs
            // TODO: TRIG: output, ECHO: input
            if(gpio_request(config.trig, "trig") < 0){
                printk(KERN_ERR "request gpio trig failed\n");
                return -EBUSY;
            }
```

```
            gpio_direction_output(config.trig, 0); // 设置为输出，初始化为低电平
            if(gpio_request(config.echo, "echo") < 0){
                printk(KERN_ERR "request gpio echo failed\n");
                gpio_free(config.trig);
                return -EBUSY;
            }
            gpio_direction_input(config.echo); // 设置为输入

            // Step3: Configure the irq
            // TODO: Set ECHO to rising-edge trigger
            if(request_irq(gpio_to_irq(config.echo), irq_handler, IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING,
"ultrasonic_echo", NULL) < 0){
                printk(KERN_ERR "request irq failed\n");
                gpio_free(config.trig);
                gpio_free(config.echo);
                return -EBUSY;
            }
            flag = 1;
            printk(KERN_INFO"CMD_INIT executed\n");
            break;
        } case CMD_TRIG: { // trig
            // TODO: Let the trig GPIO release a high level of more than 10us
            gpio_set_value(config.trig, 1);
            udelay(20);
            gpio_set_value(config.trig, 0); // 这样一来就是间隔 20us 的高电平脉冲
            printk(KERN_INFO"CMD_TRIG executed\n");
            break;
        } case CMD_READ: { // get output
            config.output = distance;
            if (copy_to_user((void*) arg, &config, sizeof(struct sensor_config))) {
                printk(KERN_ERR"[%s %d]  : copy_to user failed !\n", __func__, __LINE__);
                return -EFAULT;
            }
            printk(KERN_INFO"CMD_READ executed\n");
            break;
        } default:
            printk(KERN_INFO"[%s %d]:Invalid cmd: %d\n", __func__, __LINE__, cmd);
            break;
    }
    return 0;
}
// 好像 Linux 神奇之处在于所有东西都可以当成文件来操作？
static struct file_operations demo_fops = {
    .owner = THIS_MODULE,
```

```
    .open = demo_open,
    .release = demo_release,
    .unlocked_ioctl = demo_ioctl,
};
// 初始化驱动
static int __init demo_init(void) {
    int err;
    printk(KERN_INFO"Demo Init \n");
    err = alloc_chrdev_region(&demo_devno, 0, 1, DRIVER_NAME);
    if (err < 0) {
        goto err;
    }
    cdev_init(&demo_dev, &demo_fops);
    err = cdev_add(&demo_dev, demo_devno, 1);
    if (err < 0) {
        printk(KERN_ERR"[%s,%d]add cdev failed\n",__func__,__LINE__);
        goto FREE_DEVNO;
    }
    // Automatically generate device files in the /dev directory with the file name DEVICE_ NAME
    demo_class = class_create(THIS_MODULE, DEVICE_NAME);
    if(IS_ERR(demo_class)) {
        printk(KERN_ERR"[%s,%d]class create failed\n", __func__, __LINE__);
        goto DEV_FREE;
    }
    device_create(demo_class, NULL, demo_devno, NULL, DEVICE_NAME);
    return 0;
DEV_FREE:
    cdev_del(&demo_dev);
FREE_DEVNO:
    unregister_chrdev_region(demo_devno, 1);
err:
    return err;
}
static void demo_exit(void) {
    if (flag) {
        free_irq(gpio_to_irq(config.echo),NULL);
        gpio_free(config.echo);
        gpio_free(config.trig);
    }
    device_destroy(demo_class, demo_devno);
    class_destroy(demo_class);
    cdev_del(&demo_dev);
    unregister_chrdev_region(demo_devno, 1);
    printk(KERN_INFO"Demo exit\n");
```

```
}
module_init(demo_init);
module_exit(demo_exit);
MODULE_AUTHOR("bubblevan");
MODULE_DESCRIPTION("Ultrasonic Sensor Driver");
MODULE_LICENSE("GPL");
```

driver/Makefile

```
ARCH=arm64
CROSS_COMPILE=/home/bubblevan/lab2_zju/gcc-linaro-7.5.0-2019.12-x86_64_aarch64-linux-gnu/bin/aarch64-linux-gnu-

ifneq ($(KERNELRELEASE),)
    obj-m := demo.o
else
    KERNELDIR := /home/bubblevan/lab2_zju/linux
    PWD := $(shell pwd)
default:
    $(MAKE) CROSS_COMPILE=$(CROSS_COMPILE) ARCH=$(ARCH) -C $(KERNELDIR) M=$(PWD) modules
clean:
    rm -rf *.o *.ko *.mod.c *.order *.symvers
endif
```