数字系统II 实验 报告三

南京大学 电子科学与工程学院 吴康正 171180571
项目repo地址：https://git.nju.edu.cn/Minaduki/beaglebone_proj

实验目的

1. 学习嵌入式Linux操作系统设备驱动的方法。
2. 编写一个I/O接口的驱动模块，并且安装到系统内核中，实现通过syscall访问设备。
3. 提供一个用户空间程序，包装对接口的访问。

实验环境

1. 硬件环境

实验使用了TI的BeagleBone Black开发板作为实验环境，其参数如下：

- 处理器：基于ARM Cortex-A8架构的TI AM3359 Sitara @ 1GHz
- 内存：板载512MiB DDR3
- 存储：板载4GiB 8-bit eMMC闪存
- 拓展存储：支持micro SD存储卡
- 网络界面：RJ-45接口百兆以太网
- 数字多媒体输出：micro HDMI接口
- 拓展接口：UART、GPIO、SPI、I2C等

使用的显示器：

- 分辨率：最大1920*1080
- 色彩空间：RGB565

2. 软件环境

- 上位机：使用Ubuntu 19.10系统的x86 PC
- Linux源码：版本4.4.155
- Busybox源码：版本1.30.1
- 编译器：The GNU Compiler Collection 9.2.1
- bootloader：U-boot

3. 网络环境

- 网关：192.168.208.254
- 上位机：192.168.208.35
- 开发板：192.168.208.121

实验原理

> 关于实验原理：
> 具体的实验原理中，实验指导书有所提及的在这里不再重复叙述。这里仅进行一些补充说明。

关于内核模块

内核模块可以减小内核的体积，同时增强内核的拓展性。内核模块在加载时只会被链接到内核，因此不能使用标准glibc库的函数，应该使用内核提供的API，并在编译时提供内核源码。
内核模块可以用于实现设备驱动的功能。
内核模块驱动程序与守护进程的区别有以下几点：

1. 守护进程运行在用户空间，内核模块运行在内核空间。因此内核模块不能直接访问用户空间的指针。
2. 守护进程是后台进程，系统CPU会定期访问进程以检查是否有请求，而内核模块只能被动接受请求。

关于设备文件

设备文件使用`mknod`命令创建，通过设备号与内核设备驱动相对应，在对设备文件请求系统调用时，会根据设备文件的设备号去内核寻找注册了该设备号的模块，然后调用其注册的方法进行实际I/O操作。

关于LED与GPIO

BeagleBone Black开发板上总共有3个空闲可供使用的LED(还有一个被系统用作状态指示灯了，可以改变状态，但是会被系统状态覆盖)，这三个LED都位于GPIO1控制器上，分别对应GPIO1_24-GPIO1_22的引脚。
对于GPIO控制器，其拥有数个寄存器，其中与本次实验相关的寄存器为`GPIO_OE`与`GPIO_DATAOUT`，每个寄存器大小为4byte，对应每个GPIO控制器控制的32个引脚，其中`GPIO_OE`负责输出使能，0为输出，1为输入，而`GPIO_DATAOUT`对应每个引脚具体的输出状态。实际使用中，只需要将对应的寄存器地址使用`ioremap`映射到内存中，然后就可以进行存取。
对于GPIO1控制器，其基地址与各个寄存器的偏移地址如下：

寄存器	地址	备注
GPIO_BASE	0x4804C000	基地址，非寄存器
GPIO_OE	GPIO_BASE + 0x134	输出使能(0为输出，1为输入)
GPIO_DATAIN	GPIO_BASE + 0x138	输入值
GPIO_DATAOUT	GPIO_BASE + 0x13C	输出值

实验流程

1. 准备工作

1.1 重新配置编译内核

使用`make menuconfig ARCH=arm`对内核进行自定义，需要保证如下条件满足：
`Enable loadable module support`处于启用状态，支持在系统运行时加载/卸载驱动模块。

如有配置修改，配置完成后重新编译内核。

至此，所有准备工作完成，使用内核和根文件系统启动。

2. 确定项目结构

项目由三部分组成，分别为：

1. 内核模块文件`led.c`和`led.h`，编译后生成内核目标文件，提供syscall供用户程序调用。
2. 脚本文件`led.sh`，负责在开发板上安装内核模块，并且创建设备文件，如有必要，可以将此脚本放入初始化表或启动脚本中，实现开机自动安装模块。
3. 用户程序`ledctrl.c`，通过syscall调用内核模块驱动，提供直观的LED驱动控制。

项目通过Makefile进行管理，调用内核Makefile将`led.c`和`led.h`编译为内核目标文件`led.o`，Makefile中的变量`obj-m = led.o`中的m代表编译为可卸载的模块而非整合进入内核。

```
PWD = $(shell pwd)
KERNELDIR = /home/minaduki/Desktop/Beaglebone_Proj/src
ARCH_TYPE = arm
PREFIX = arm-linux-gnueabihf-
obj-m = led.o
APP = ledctrl
APP_SRC = ledctrl.c

default: $(APP)
    $(MAKE) CROSS_COMPILE=$(PREFIX) ARCH=$(ARCH_TYPE) -C $(KERNELDIR) SUBDIF

$(APP): $(APP_SRC)
    $(PREFIX)gcc -o $@ $^

copy: $(APP) led.ko
    cp $^ /home/minaduki/Desktop/nfsroot/nfs4/

clean:
    $(MAKE) CROSS_COMPILE=$(PREFIX) ARCH=$(ARCH_TYPE) -C $(KERNELDIR) SUBDIF
    rm $(APP)
```

其中，各个目标的作用如下：

- $(target): 用于生成全部程序，包含内核目标文件
- $(APP): 用于生成LED控制器的用户程序
- copy: 用于将生成的文件复制到NFS根目录，供开发板运行
- clean: 用于清理

3. 编写程序

> 仅包含部分关键代码，完整源码请查看此处。

3.1 内核模块

1. 头文件led.h

头文件中包含了定义，包括寄存器的地址、数据结构和文件操作的重写：
寄存器地址之前已经提及，不再重复；
定义一个用于描述gpio控制器状态的结构体gpio_t，包含以下内容：

```
typedef struct
{
    volatile int* pConf;
    volatile int* pDataIn;
    volatile int* pDataOut;
} gpio_t;
```

三个条目分别指向重映射后的输出使能寄存器、输入寄存器和输出寄存器。 对文件操作中的open、write、read和release方法进行重写，分别对应驱动文件中的方法：

```
struct file_operations led_fops =
{
    .open = led_open,
    .write = led_write,
    .read = led_read,
    .release = led_release
};
```

2. 源文件led.c

源文件中首先定义了一个静态变量major，用于存储分配到的主设备号：

```
static unsigned int major = 0; //主设备号（用于区分设备类）
```

接下来是模块的加载与卸载函数，一般而言，要求函数名为init_module和cleanup_module，但是处于方便考虑，也可以使用自定义的函数名，比如这里使用led_init作为初始化函数的名称，并在文件中声明module_init(led_init);，使系统在加载内核时能找到正确的函数。

```
major = register_chrdev(0, "LED", &led_fops);
```

在加载内核模块时，需要调用register_chrdev向系统申请注册一个字符设备，这里指定了设备名称为"LED"，不指定设备号，这样系统就会自动分配一个设备号，并返回存储在major变量中。然后调用printk函数将设备号打印在内核调试信息里，这样就可以通过dmesg命令查看得到的设备号，以在稍后的步骤中创建设备文件。

为了在不需要控制的时候节约系统资源(虽然也节约不了多少，而且代价是调用时响应时间的增加)，因此在安装内核时不进行内存的分配和I/O的重映射，而是在打开设备时执行。

```
int led_open(struct inode* inode, struct file* filp)
{
    gpio_t* gpio;

    filp->private_data = kmalloc(sizeof(gpio_t), GFP_KERNEL);
```

```
    gpio = (gpio_t*)(filp->private_data);

    gpio->pConf = ioremap(GPIO_OE, 4); /* 映 射 方 向 寄 存 器 */
    gpio->pDataIn = ioremap(GPIO_IN, 4); /* 映 射 输 入 寄 存 器 */
    gpio->pDataOut = ioremap(GPIO_OUT, 4); /* 映 射 输 出 寄 存 器 */
    *(gpio->pConf) &= ~(7 << 22);

    return 0;
}
```

对于每个模块，系统都有一个file结构体，其中的private_data指针可以用于管理模块需要存储的私有数据。在内核中分配一段空间用于存放gpio的重映射指针，并将private_data指针指向这段分配的空间，然后对于每一个重映射指针，使用ioremap函数将对应的GPIO寄存器的地址重映射到指针上，最后将GPIO的输出使能对应24、23、22引脚的bit复位，代表设置这三个引脚位输出模式，这样就完成了LED控制驱动的初始化。

> 为什么左移22?
> 查阅资料发现，GPIO的引脚编号是从0开始计算的，因此22-24对应的应该是从1开始计算的第23-25位。

对应的，当已经没有人在访问设备时，应该释放设备资源，而设备资源的释放不应该影响硬件状态，因此在release函数中，不对设备寄存器进行操作，使用ioummap取消对I/O的映射，然后释放private_data中分配到的内存。

在读取设备的状态时，由于具体的模块驱动程序是一个LED的驱动而不是I/O驱动，因此此处不应该切换GPIO的模式，也不应该访问输入寄存器，而应该将输出寄存器中的值返回给用户，以"读取"此时LED的发光状态。使用copy_to_user函数将从输出寄存器中取出的值送入用户空间内：

```
ssize_t led_read(struct file* filp, char* buff, size_t count, loff_t* f_pos)
{
    gpio_t* gpio = (gpio_t*)(filp->private_data);
    int data = *(gpio->pDataOut);
    int uncopied = copy_to_user(buff, &data, count);

    return sizeof(char) - uncopied;
}
```

在写入设备状态时，为了确保使用一些基本busybox程序如echo能够正常控制设备，因此规定写入格式为一个0-7之间的数字，代表从000到111中的8种状态，如果输入的值是ASCII码的数字，由于程序取低三位进行判断，因此不会有影响。然后，进行移位操作，取出每个LED应该对应的状态。

```
ssize_t led_write(struct file* filp, const char* buff, size_t count, loff_t*
{
    char c;
    gpio_t* gpio = (gpio_t*)(filp->private_data);

    int uncopied = copy_from_user(&c, buff, sizeof(buff));

    int temp[3];
```

```
    int i = 0;

    temp[0] = ((unsigned int)(c)) & (1 << 0);
    temp[1] = ((unsigned int)(c)) & (1 << 1);
    temp[2] = ((unsigned int)(c)) & (1 << 2);

    for (i = 0; i < 3; ++i)
    {
        if (temp[i])
        {
            printk("Turning LED %d on\n", i);
            led_on(i + 1);
        }
        else
        {
            printk("Turning LED %d off\n", i);
            led_off(i + 1);
        }
    }

    printk("Write finished\n");
    return sizeof(char) - uncopied;
}
```

根据每一位的状态分别调用不同的语句向映射的寄存器中输入不同的值，led_on和led_off是定义在头文件里的宏，结构如下：

```
#define led_on(index) (*(gpio->pDataOut) |= (1 << (21 + index)))
#define led_off(index) (*(gpio->pDataOut) &= ~(1 << (21 + index)))
```

最后是移除模块时的操作，我们希望在移除模块后LED不在工作，因此首先通过I/O重映射将所有的输出寄存器复位，然后再取消映射，同时使用unregister_chrdev(major, "LED");注销设备号。

3.2 用户程序

用户程序提供了一种访问控制LED的方法，同时用户也可以通过其他程序使用指定的方式调用syscall，从而完成对LED的控制。

程序支持输入参数，可以按照ledctrl <LED0> <LED1> <LED2>的方法对三个LED的状态进行控制，如果不输入参数，则会询问输入的值，然后使用write函数送入设备文件中。

```
    int fd = open("/dev/led", O_RDWR);
    char buf = 0;
    for (int i = 0; i < 3; ++i)
    {
        buf |= (argv[i + 1][0] - 0x30) << i;
    }
    printf("Writing %d to device\n", (int)buf);
    write(fd, &buf, 1);
    close(fd);
    return 0;
```

4. 运行效果

4.1 编译程序

使用make编译程序，然后使用make copy复制到根文件系统，准备测试。

4.2 安装模块与创建设备文件

文件中已经准备了led.sh用于安装模块和创建，但首次使用还是手动安装，以进行直观感受：

```
/mnt # insmod led.ko
[   61.160530] led: loading out-of-tree module taints kernel.
[   61.166134] led: module license 'MIT' taints kernel.
[   61.171141] Disabling lock debugging due to kernel taint
[   61.177251] Device registered with MAJOR = 245
[   61.181816] do_init_module: 'led'->init suspiciously returned 245, it shc
follow 0/-E convention
[   61.181816] do_init_module: loading module anyway...
[   61.195911] CPU: 0 PID: 138 Comm: insmod Tainted: P           O    4.4.15
11
[   61.203250] Hardware name: Generic AM33XX (Flattened Device Tree)
[   61.209409] [<c001b1cc>] (unwind_backtrace) from [<c00158d4>] (show_stack
0/0x24)
[   61.217205] [<c00158d4>] (show_stack) from [<c052809c>] (dump_stack+0x8c/
)
[   61.224474] [<c052809c>] (dump_stack) from [<c00d658c>] (do_init_module+0
0x1e8)
[   61.232169] [<c00d658c>] (do_init_module) from [<c00d892c>] (load_module+
90/0x2594)
[   61.240212] [<c00d892c>] (load_module) from [<c00d8e70>] (SyS_init_module
40/0x1f4)
[   61.248174] [<c00d8e70>] (SyS_init_module) from [<c0010f20>] (ret_fast_sy
l+0x0/0x50)
```

从内核调试信息输出可得，分配到的主设备号为245，根据此设备号，使用mknod命令创建一个设备led：

```
/ # mknod /dev/led c 245 0
/ # ls -al /dev/led
crw-r--r--    1 0        0         245,   0 Jan  1 01:01 /dev/led
```

成功创建了一个主设备号为245，次设备号为0的块设备。

4.3 调用驱动

1. 首先，使用echo命令访问设备文件，并输入一些数值进行测试：

```
/ # echo -n 7 > /dev/led
[  168.642410] Turning LED 0 on
[  168.645427] Turning LED 1 on
[  168.648345] Turning LED 2 on
[  168.651235] Write finished
[  168.654059] Releasing

/ # echo -n 0 > /dev/led
[  193.203207] Turning LED 0 off
[  193.206309] Turning LED 1 off
[  193.209309] Turning LED 2 off
[  193.212304] Write finished
[  193.215116] Releasing

/ # echo -n 9 > /dev/led
[  274.992720] Turning LED 0 on
[  274.995638] Turning LED 1 off
[  274.998671] Turning LED 2 off
[  275.001672] Write finished
[  275.004491] Releasing
```

可以看见，设备驱动正确解析了输入的数字，并点亮了对应的LED，实际LED的点亮情况与内核输出一致。
对于输入9的情况，由于9转换为二进制是1001，而驱动只会处理后三位，因此输入9与输入1等价。

> 关于`echo -n`:
> `-n`选项用于表示“不输出换行符”，如果不加上这个选项的话，输出会变成这样的：

```
/ # echo 7 > /dev/led
[  206.883079] Turning LED 0 on
[  206.886118] Turning LED 1 on
[  206.889038] Turning LED 2 on
[  206.891927] Write finished
[  206.894813] Turning LED 0 off
[  206.897839] Turning LED 1 on
[  206.900757] Turning LED 2 off
[  206.903734] Write finished
[  206.907139] Releasing
```

> 这是由于换行符对应的ASCII码为0x0A，即二进制1010，与输入2(010)等价。

1. 使用提供的用户程序`lcdctrl`控制LED状态，可以在命令行指定LED状态：

```
/mnt # ./ledctrl 1 0 1
Writing 5 to device[  244.743726] Turning LED 0 on

[  244.748612] Turning LED 1 off
[  244.751595] Turning LED 2 on
[  244.754518] Write finished
[  244.757741] Releasing
```

3. 也可以直接打开程序，通过输入不同的值来切换LED状态：

```
/mnt # ./ledctrl
Please input 0 to 7 to change status of 3 leds, an input of 8 or larger will
considered as exiting
Please input:1
[  315.227594] Turning LED 0 on
[  315.230577] Turning LED 1 off
[  315.233574] Turning LED 2 off
[  315.236566] Write finished
Please input 0 to 7 to change status of 3 leds, an input of 8 or larger will
considered as exiting
Please input:4
[  317.243592] Turning LED 0 off
[  317.246697] Turning LED 1 off
[  317.249692] Turning LED 2 on
[  317.252598] Write finished
Please input 0 to 7 to change status of 3 leds, an input of 8 or larger will
considered as exiting
Please input:3
[  319.611692] Turning LED 0 on
[  319.614722] Turning LED 1 on
[  319.617634] Turning LED 2 off
[  319.620629] Write finished
Please input 0 to 7 to change status of 3 leds, an input of 8 or larger will
considered as exiting
Please input:7
[  321.499534] Turning LED 0 on
[  321.502567] Turning LED 1 on
[  321.505473] Turning LED 2 on
[  321.508379] Write finished
Please input 0 to 7 to change status of 3 leds, an input of 8 or larger will
considered as exiting
Please input:8
[  322.780032] Releasing
```

4.4 卸载设备

使用rmmod命令卸载设备，释放设备号：

```
/mnt # rmmod led
[  340.321003] LED module reset and removed.
```

LED灯全部关闭，同时设备号245也被释放。