重庆邮电大学

学生实验报告册

课程名称：控制系统开发技术(4)(嵌入式系统)

学　　院：自动化学院

专业班级：

姓　　名：

学　　号：

指导教师：

成　　绩：

学年学期：2022-2023 学年 □春■秋学期

重庆邮电大学教务处制

实验项目名称	嵌入式 Linux 程序开发综合实验		
实验地点	仙桃 314	实验时间	12.15
实验指导教师		成绩	

一、实验目的

1、掌握嵌入式 Linux 开发工具使用方法；

2、熟悉嵌入式 Linux 应用程序开发方法及技术；

3、熟悉嵌入式 Linux 驱动程序开发方法；

4、掌握嵌入式软硬件联合调试方法及技巧。

二、实验内容

基于实验中心提供的 NXP imx6ull 开发板基础软件代码包，开发 led 字符设备驱动与 led 控制应用程序，实现对 led 灯的开关控制，要求：

1、led 设备驱动实现 read()、write()接口，应用层通过 read()接口可获得 led 当前亮灭状态，通过 write()接口可控制 led 亮灭。

2、led 控制应用程序内创建线程 A 与 B、信号量 A’与 B’，线程 A 负责点亮 led，线程 B 负责关闭 led；A 与 B 线程运行后分别等待信号量 A’与 B’到达；当线程 A 接收到信号量 A’时点亮 led，等待 10 秒后向线程 B 发送信号量 B’，随即线程 A 又继续等待信号量 A’；当线程 B 接收到信号量 B’后关闭 led，等待 10 秒后向线程 A 发送信号量 A’，随即线程 B 又继续等待信号量 B’。如此交替往复 10 次，应用程序安全退出。

3、完成软硬件联合调试。

三、实验仪器设备、材料

硬件：嵌入式 Linux 开发板、PC 机、USB 连接线。

软件：PC 机 Linux 操作系统、嵌入式 Linux 交叉编译工具链、超级终端等工具。

四、设计方案

(一) 驱动开发

Linux 驱动分为三大类：字符设备驱动、块设备驱动、网络设备驱动。字符设备是 Linux 驱动中最基本的一类设备驱动，字符设备就是按照字节流进行读写操作的设备。本实验驱动 LED，本质上就是对一个 IO 口进行操作，对相关寄存器进行读写操作，应该归类为字符设备驱动。

Linux 将空间分为内核空间和用户空间，Linux 内核和驱动程序运行在内核空间，驱动成功加载后会在“/dev”下生成对应文件，位于用户空间的应用程序通过具体的驱动程序来实现对硬件的控制。

```
root@ATK-IMX6U:/dev# ls
autofs           loop6              ram11   tty11  tty35  tty59
block            loop7              ram12   tty12  tty36  tty6
bus              loop-control       ram13   tty13  tty37  tty60
char             mem                ram14   tty14  tty38  tty61
console          memory_bandwidth   ram15   tty15  tty39  tty62
cpu_dma_latency  mmcblk1            ram2    tty16  tty4   tty63
disk             mmcblk1boot0       ram3    tty17  tty40  tty7
dri              mmcblk1boot1       ram4    tty18  tty41  tty8
fb               mmcblk1p1          ram5    tty19  tty42  tty9
fb0              mmcblk1p2          ram6    tty2   tty43  ttymxc0
fd               mmcblk1rpmb        ram7    tty20  tty44  ttymxc2
full             mtab               ram8    tty21  tty45  ubi_ctrl
fuse             mxc_asrc           ram9    tty22  tty46  udev_network_queue
hwrng            network_latency    random  tty23  tty47  urandom
i2c-0            network_throughput rfkill  tty24  tty48  v4l
i2c-1            null               rtc     tty25  tty49  vcs
initctl          ppp                rtc0    tty26  tty5   vcs1
input            pps0               shm     tty27  tty50  vcs2
kmsg             pps1               snd     tty28  tty51  vcsa
log              ptmx               stderr  tty29  tty52  vcsa1
loop0            ptp0               stdin   tty3   tty53  vcsa2
loop1            ptp1               stdout  tty30  tty54  video0
loop2            pts                tty     tty31  tty55  watchdog
loop3            ram0               tty0    tty32  tty56  watchdog0
loop4            ram1               tty1    tty33  tty57  xconsole
loop5            ram10              tty10   tty34  tty58  zero
```

应用程序可以通过系统调用的方法从应用空间“陷入”到内核空间，这样才能访问底层驱动的资源并进行操作。在 Linux 系统中，系统调用为 C 库的一部分，当应用程序调用 open、write 等函数时，本质上通过调用 C 库中的 open、write 函数来进行内核的系统调用，最终实现驱动的 open、write 函数调用。

因此，驱动中应该有与应用程序对应的函数，驱动函数的定义已经在内核文件中定义，我们只需要引用头文件并调用即可。

Linux 内核启动的时候会初始化 MMU。MMU 实现了虚拟空间到物理空间的映射，也就是地址映射。对于实验中使用的 512MB 的 DDR3 物理内存，经过 MMU 的映射就可以映射到 4GB 的空间。这就导致了我们并不能像单片机开发时那样，直接对一个物理地址进行读写操作，需要实现物理内存到虚拟内存的转换。这里我们可以通过调用 ioremap 和 iounmap 来实现。

对于控制 LED 来说，只需要拉高或者拉低 IO 口，所以我们只需要重点编写 write 函数和 init 函数。参考正点原子提供的资料，编写了以下函数。

```c
static int __init led_init(void)
{
    int retvalue = 0;
    u32 val = 0;

    /* 初始化LED */
    /* 1、寄存器地址映射 */
    IMX6U_CCM_CCGR1 = ioremap(CCM_CCGR1_BASE, 4);
    SW_MUX_GPIO1_IO03 = ioremap(SW_MUX_GPIO1_IO03_BASE, 4);
    SW_PAD_GPIO1_IO03 = ioremap(SW_PAD_GPIO1_IO03_BASE, 4);
    GPIO1_DR = ioremap(GPIO1_DR_BASE, 4);
    GPIO1_GDIR = ioremap(GPIO1_GDIR_BASE, 4);

    /* 2、使能GPIO1时钟 */
    val = readl(IMX6U_CCM_CCGR1);
    val &= ~(3 << 26);  /* 清楚以前的设置 */
    val |= (3 << 26);   /* 设置新值 */
    writel(val, IMX6U_CCM_CCGR1);

    /* 3、设置GPIO1_IO03的复用功能，将其复用为
     *    GPIO1_IO03，最后设置IO属性。
     */
    writel(5, SW_MUX_GPIO1_IO03);
    writel(0x10B0, SW_PAD_GPIO1_IO03);

    /* 4、设置GPIO1_IO03为输出功能 */
    val = readl(GPIO1_GDIR);
    val &= ~(1 << 3);   /* 清除以前的设置 */
    val |= (1 << 3);    /* 设置为输出 */
    writel(val, GPIO1_GDIR);

    /* 5、默认关闭LED */
    val = readl(GPIO1_DR);
    val |= (1 << 3);
    writel(val, GPIO1_DR);

    /* 6、注册字符设备驱动 */
    retvalue = register_chrdev(LED_MAJOR, LED_NAME, &led_fops);
    if(retvalue < 0){
        printk("register chrdev failed!\r\n");
        return -EIO;
    }
    return 0;
}
```

Led 初始化函数，通过 ioremap 获取地址映射，得到相关寄存器的地址。类似于单片机开发，IO 口的配置的流程，顺序为使能时钟，设置为复用功能，

并且注册为字符驱动。

```c
static ssize_t led_write(struct file *filp, const char __user *buf, size_t cnt, loff_t *offt)
{
    int retvalue;
    unsigned char databuf[1];
    unsigned char ledstat;

    retvalue = copy_from_user(databuf, buf, cnt);
    if(retvalue < 0) {
        printk("kernel write failed!\r\n");
        return -EFAULT;
    }

    ledstat = databuf[0];       /* 获取状态值 */

    if(ledstat == LEDON) {
        led_switch(LEDON);      /* 打开LED灯 */
    } else if(ledstat == LEDOFF) {
        led_switch(LEDOFF); /* 关闭LED灯 */
    }
    return 0;
}
```

LED_write 函数主要通过另外一个封装好的 led_switch 函数进行对寄存器的操作。

```c
static ssize_t led_read(struct file *filp, char __user *buf, size_t cnt, loff_t *offt)
{
    int retvalue;
    unsigned char databuf[1];
    unsigned char ledstat;

    retvalue = copy_from_user(databuf, buf, cnt);
    if(retvalue < 0) {
        printk("kernel write failed!\r\n");
        return -EFAULT;
    }
    ledstat = databuf[0];       /* 获取状态值 */
    val =readl(GPIO1_DR);
    if (val)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
```

Read 函数在后续应用开发并没有用到，其效果就是读取寄存器 GPIO1_DR 的值。当为 0 时，代表 LED 熄灭，为 1 时，代表 LED 点亮。

在 Ubuntu 20.04.2 环境下编译，产生了后缀为.ko 的模块文件。

```
cqupt@zhang-yucong:~/C_Program/4.4$ make clean
make -C /home/cqupt/linux/linux-imx-rel_imx_4.1.15_2.1.0_ga_alientek/ M=/home/cqupt/C_Program/4.4 clean
make[1]: 进入目录“/home/cqupt/linux/linux-imx-rel_imx_4.1.15_2.1.0_ga_alientek”
  CLEAN   /home/cqupt/C_Program/4.4/.tmp_versions
  CLEAN   /home/cqupt/C_Program/4.4/Module.symvers
make[1]: 离开目录“/home/cqupt/linux/linux-imx-rel_imx_4.1.15_2.1.0_ga_alientek”
cqupt@zhang-yucong:~/C_Program/4.4$ make
make -C /home/cqupt/linux/linux-imx-rel_imx_4.1.15_2.1.0_ga_alientek/ M=/home/cqupt/C_Program/4.4 modules
make[1]: 进入目录“/home/cqupt/linux/linux-imx-rel_imx_4.1.15_2.1.0_ga_alientek”
  CC [M]  /home/cqupt/C_Program/4.4/led.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/cqupt/C_Program/4.4/led.mod.o
  LD [M]  /home/cqupt/C_Program/4.4/led.ko
make[1]: 离开目录“/home/cqupt/linux/linux-imx-rel_imx_4.1.15_2.1.0_ga_alientek”
cqupt@zhang-yucong:~/C_Program/4.4$ ls
led.c  led.ko  led.mod.c  led.mod.o  led.o  main.c  Makefile  modules.order  Module.symvers  myledApp.c  myledTest
cqupt@zhang-yucong:~/C_Program/4.4$
```

通过 MobaXterm 的 Z-Modem 方式将文件传送到开发板上，并进行模块挂载。通过 lsmod 和 cat /proc/devices 命令来查看，发现模块已经成功挂载。至此我们完成了驱动的开发。

```
root@ATK-IMX6U:/lib/modules/4.1.15# cat /proc/devices
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
 10 misc
 13 input
 14 sound
 29 fb
 81 video4linux
 89 i2c
 90 mtd
108 ppp
116 alsa
128 ptm
136 pts
166 ttyACM
180 usb
188 ttyUSB
189 usb_device
200 led
207 ttymxc
226 drm
249 ttyLP
250 iio
251 watchdog
252 ptp
253 pps
254 rtc
```

(二) 应用程序设计

在 Linux 中，线程是一个进程内部的控制序列，线程在进程内部运行，本质是在进程地址空间内运行。因为描述线程的控制块和表述进程的控制块在 Linux 是类似的，因此 Linux 并没有为线程重新设计数据，依然使用 task_struct 来模拟线程。在用户层，linux 提供了 pthread 线程库，通过在

用户层模拟实现了一套线程相关的接口。

线程中使用的信号量是一种特殊的变量，他可以被增加或者减少，但对其的关键访问是通过原子操作来实现的。只有 0 和 1 两种取值的信号量叫做二值信号量，信号量的函数都以 sem_开头，常用的有以下四个：

1) int sem_init(sem_t *sem, int pshared, unsigned int value);

   这个函数是用来创建信号量的，因为实验要求两个信号量应该互不干扰，所以应该申请为局部信号量，因此 pshared 应该为 NULL，否则信号量将会在两个进程间共享，value 为初始值。

2) int sem_wait(sem_t *sem);

   此函数实现了以原子操作的方式对信号量减 1。信号量为 1 代表后续线程可以通过，信号量为 0 时，将线程阻塞。

3) int sem_post(sem_t *sem);

   此函数实现了以原子操作的方式对信号量加 1。

4) int sem_destroy(sem_t *sem);

   此函数实现了对 sem_init 创建的信号量的回收。

借助信号量，我们可以实现进程的同步。但是我们需要依靠 pthread 库来实现多线程的基本操作。在 Linux 下，与本实验多线程相关的 API 函数接口有：

1) int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg);
2) int pthread_join(pthread_t thread, void **retval);
3) void pthread_exit（void *retval）;

pthread_create 函数用于创建线程，pthread_join 函数用于在主线程中结束线程并且确认线程退出的状态，pthread_exit 函数用于在子线程中结束本进程。

了解了以上的函数，就可以进行 App 函数的编写，但是在实际操作中发

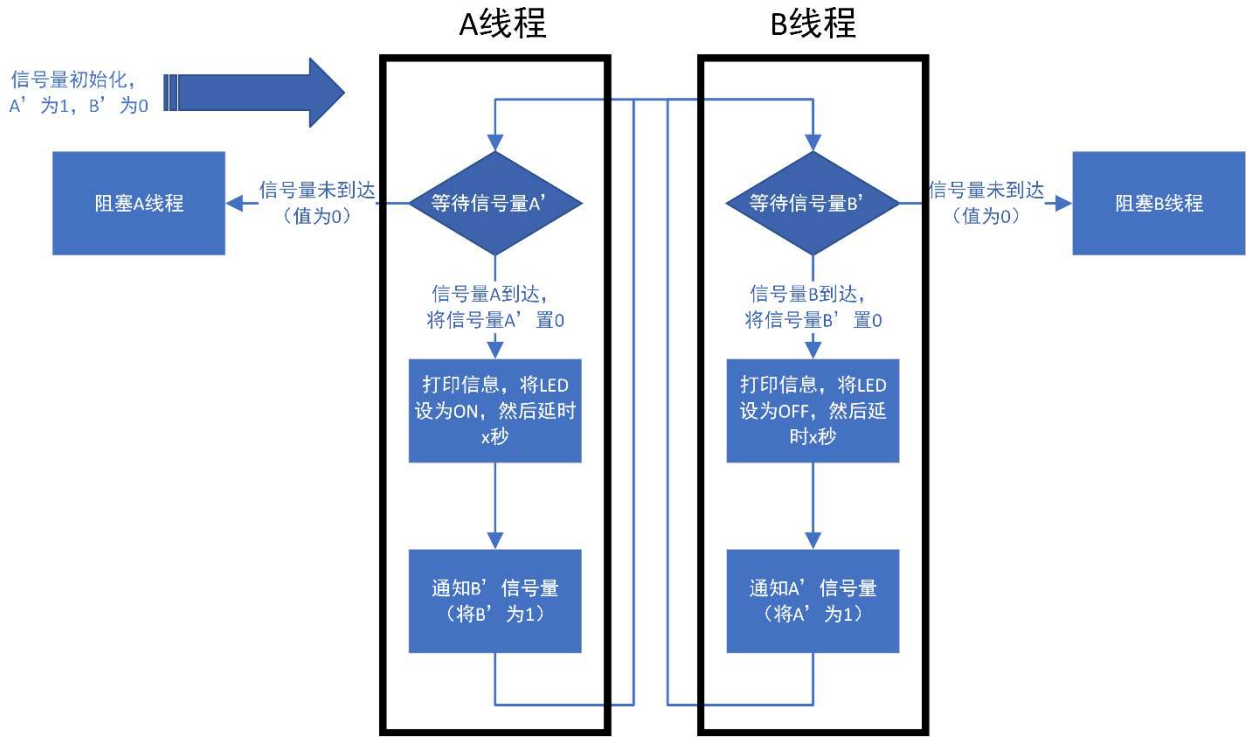
现，还需要注意两点问题：

1）开发板的出厂固件中将 led 设置为心跳灯，我们应该通过以下命令将心跳灯关闭：echo none > /sys/class/leds/sys-led/trigger

2）只在 App.c 中加入<pthread.h>函数并不能通过 gcc 编译，我们需要将其链接 libpthread.so 这个库。同时，需要将 gcc 的编译标准改为 c99。以上两点均需要在编译时注意。最终的编译命令应该为：

```
cqupt@zhang-yucong:~/C_Program/4.4$ arm-linux-gnueabihf-gcc main.c -std=c99 -o ledTest -lpthread
cqupt@zhang-yucong:~/C_Program/4.4$
```

分析实验要求，利用信号量和 pthread 库，我们可以画出以下的流程图：



在初始化信号量时，我将 A’设为 1，B’为 0，这样就可以使线程开始运行时，A 线程先工作，B 线程阻塞。随后 A 线程阻塞，B 线程工作，使用简单的 for 循环来实现 10 个来回。往返 10 个来回后，线程通过 pthread_exit 退出。在主线程 main 中通过 pthread_join 中确保线程已经安全退出，然后使用 sem_destroy 将信号量回收。在主线程中，需要申请 2 个 pthread_t 结构体，为 pthread_create 函数提供入口参数。pthread_t 是一种用于表示线程的数据类型，每一个 pthread_t 类型的变量都可以表示一个线程。同时，信

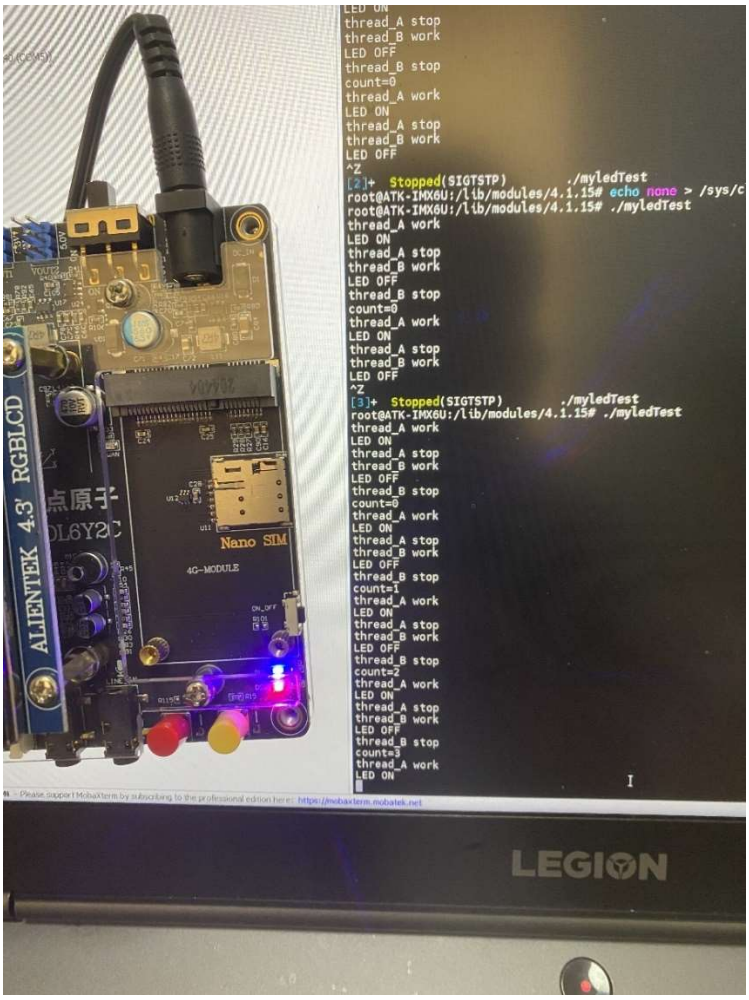
号量的申请也需要在主函数中进行操作，通过 sem_init 函数，对两个信号量进行赋值。

```c
sem_t number_A,number_B;

void *thread_A(void *arg)
{
    for(int i = 0; i <10;i++)
    {
        sem_wait(&number_A);//信号量为1可以进行，为0阻塞
        printf("thread_A work\n");
        Led_Status_Switch(LEDON);
        sleep(2);
        sem_post(&number_B);
        printf("thread_A stop\n");
    }
    printf("thread_A exited\n");
    pthread_exit(NULL);
    return NULL;
}

void *thread_B(void *arg)
{
    for(int i = 0; i <10;i++)
    {
        sem_wait(&number_B);
        printf("thread_B work\n");
        Led_Status_Switch(LEDOFF);
        sleep(2);
        sem_post(&number_A);
        printf("thread_B stop\n");
    }
    printf("thread_B exited\n");
    pthread_exit(NULL);
    return NULL;
}

int main()
{
    pthread_t pid,cid;
    sem_init(&number_A,0,1);
    sem_init(&number_B,0,0);

    pthread_create(&pid,NULL,thread_A,NULL);
    pthread_create(&cid,NULL,thread_B,NULL);
    pthread_join(pid,NULL);
    pthread_join(cid,NULL);
    sem_destroy(&number_A);
    sem_destroy(&number_B);
    pthread_exit(NULL);
    return 0;
}
```

将代码编译生成测试 App，通过串口发送给 IMX6ULL，通过 chmod 777 myledTest 命令给代码执行命令，在终端上输入执行命令，运行通过。

五、实验结果及分析（或设计总结）

```
thread_B stop
count=6
thread_A work
LED ON
thread_A stop
thread_B work
LED OFF
thread_B stop
count=7
thread_A work
LED ON
thread_A stop
thread_B work
LED OFF
thread_B stop
count=8
thread_A work
LED ON
thread_A stop
thread_A exited
thread_B work
LED OFF
thread_B stop
count=9
thread_B exited
root@ATK-IMX6U:/lib/modules/4.1.15#
```



经过 10 个来回，代码打印出信息并退出。

附录(关键源程序)

```
//以下为 App 应用程序
#include "stdio.h"
#include "unistd.h"
#include "sys/types.h"
#include "sys/stat.h"
#include "fcntl.h"
#include "stdlib.h"
#include "string.h"
#include "pthread.h"
#include "semaphore.h"

#define LEDOFF        0
#define LEDON    1

sem_t number_A,number_B;

int Led_Status_Switch(char status)
{
    int fd, retvalue;
    char *filename;
    unsigned char databuf[1];

    filename = "/dev/led";

    /* 打开 led 驱动 */
    fd = open(filename, O_RDWR);
    if(fd < 0){
        printf("file %s open failed!\r\n",filename);
        return -1;
    }

    databuf[0] = status;    /* 要执行的操作：打开或关闭 */

    /* 向/dev/led 文件写入数据 */
    retvalue = write(fd, databuf, sizeof(databuf));
    if(retvalue < 0){
        printf("LED Control Failed!\r\n");
        close(fd);
        return -1;
    }
    if(status)
    printf("LED ON\n");
```

```
    else
    printf("LED OFF\n");

    retvalue = close(fd); /* 关闭文件 */
    if(retvalue < 0){
        printf("file %s close failed!\r\n",filename);
        return -1;
    }
    return 0;
}

void *thread_A(void *arg)
{
     for(int i = 0; i <10;i++)
     {
        sem_wait(&number_A);
           printf("thread_A work\n");
        Led_Status_Switch(LEDON);
           sleep(2);
           sem_post(&number_B);
           printf("thread_A stop\n");
     }
    printf("thread_A exited\n");
    pthread_exit(NULL);
    return NULL;
}

void *thread_B(void *arg)
{
     for(int i = 0; i <10;i++)
     {
        sem_wait(&number_B);
           printf("thread_B work\n");
        Led_Status_Switch(LEDOFF);
           sleep(2);
           sem_post(&number_A);
           printf("thread_B stop\n");
           printf("count=%d\n",i);
     }
    printf("thread_B exited\n");
    pthread_exit(NULL);
    return NULL;
}
```

```
int main()
{
      pthread_t pid,cid;
      sem_init(&number_A,0,1);
      sem_init(&number_B,0,0);
      pthread_create(&pid,NULL,thread_A,NULL);
      pthread_create(&cid,NULL,thread_B,NULL);
      pthread_join(pid,NULL);
      pthread_join(cid,NULL);
      sem_destroy(&number_A);
      sem_destroy(&number_B);
     pthread_exit(NULL);
     return 0;
}

//以下为 LED 驱动程序
#include <linux/types.h>
#include <linux/kernel.h>
#include <linux/delay.h>
#include <linux/ide.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/errno.h>
#include <linux/gpio.h>
#include <asm/mach/map.h>
#include <asm/uaccess.h>
#include <asm/io.h>

#define LED_MAJOR         200      /* 主设备号 */
#define LED_NAME          "led"    /* 设备名字 */

#define LEDOFF   0                 /* 关灯 */
#define LEDON    1                 /* 开灯 */

/* 寄存器物理地址 */
#define CCM_CCGR1_BASE                (0X020C406C)
#define SW_MUX_GPIO1_IO03_BASE        (0X020E0068)
#define SW_PAD_GPIO1_IO03_BASE        (0X020E02F4)
#define GPIO1_DR_BASE                 (0X0209C000)
#define GPIO1_GDIR_BASE               (0X0209C004)

/* 映射后的寄存器虚拟地址指针 */
static void __iomem *IMX6U_CCM_CCGR1;
```

```
static void __iomem *SW_MUX_GPIO1_IO03;
static void __iomem *SW_PAD_GPIO1_IO03;
static void __iomem *GPIO1_DR;
static void __iomem *GPIO1_GDIR;

void led_switch(u8 sta)
{
    u32 val = 0;
    if(sta == LEDON) {
        val = readl(GPIO1_DR);
        val &= ~(1 << 3);
        writel(val, GPIO1_DR);
    }else if(sta == LEDOFF) {
        val = readl(GPIO1_DR);
        val|= (1 << 3);
        writel(val, GPIO1_DR);
    }
}

static int led_open(struct inode *inode, struct file *filp)
{
    return 0;
}

static ssize_t led_read(struct file *filp, char __user *buf, size_t cnt, loff_t *offt)
{
//因为应用程序并未使用 read，所以此处留成空
    return 0;
}

static ssize_t led_write(struct file *filp, const char __user *buf, size_t cnt, loff_t
*offt)
{
    int retvalue;
    unsigned char databuf[1];
    unsigned char ledstat;

    retvalue = copy_from_user(databuf, buf, cnt);
    if(retvalue < 0) {
        printk("kernel write failed!\r\n");
        return -EFAULT;
    }

    ledstat = databuf[0];        /* 获取状态值 */
```

```
    if(ledstat == LEDON) {
        led_switch(LEDON);      /* 打开 LED 灯 */
    } else if(ledstat == LEDOFF) {
        led_switch(LEDOFF); /* 关闭 LED 灯 */
    }
    return 0;
}

static int led_release(struct inode *inode, struct file *filp)
{
    return 0;
}

/* 设备操作函数 */
static struct file_operations led_fops = {
    .owner = THIS_MODULE,
    .open = led_open,
    .read = led_read,
    .write = led_write,
    .release =  led_release,
};

static int __init led_init(void)
{
    int retvalue = 0;
    u32 val = 0;
    IMX6U_CCM_CCGR1 = ioremap(CCM_CCGR1_BASE, 4);
    SW_MUX_GPIO1_IO03 = ioremap(SW_MUX_GPIO1_IO03_BASE, 4);
    SW_PAD_GPIO1_IO03 = ioremap(SW_PAD_GPIO1_IO03_BASE, 4);
    GPIO1_DR = ioremap(GPIO1_DR_BASE, 4);
    GPIO1_GDIR = ioremap(GPIO1_GDIR_BASE, 4);
    val = readl(IMX6U_CCM_CCGR1);
    val &= ~(3 << 26);  /* 清楚以前的设置 */
    val |= (3 << 26);   /* 设置新值 */
    writel(val, IMX6U_CCM_CCGR1);
    writel(5, SW_MUX_GPIO1_IO03);
    writel(0x10B0, SW_PAD_GPIO1_IO03);
    val = readl(GPIO1_GDIR);
    val &= ~(1 << 3);   /* 清除以前的设置 */
    val |= (1 << 3);    /* 设置为输出 */
    writel(val, GPIO1_GDIR);
    val = readl(GPIO1_DR);
    val |= (1 << 3);
```

```
    writel(val, GPIO1_DR);

    retvalue = register_chrdev(LED_MAJOR, LED_NAME, &led_fops);
    if(retvalue < 0){
        printk("register chrdev failed!\r\n");
        return -EIO;
    }
    return 0;
}

static void __exit led_exit(void)
{
    /* 取消映射 */
    iounmap(IMX6U_CCM_CCGR1);
    iounmap(SW_MUX_GPIO1_IO03);
    iounmap(SW_PAD_GPIO1_IO03);
    iounmap(GPIO1_DR);
    iounmap(GPIO1_GDIR);

    unregister_chrdev(LED_MAJOR, LED_NAME);
}

module_init(led_init);
module_exit(led_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("zhangyucong");
```