```c
/*
 * Aaron Chan
 * ECE 373 (Spring 2017)
 * Assignment #3
 *
 * This module will initialize a PCI driver
 * so that we can write to the Atom Box's
 * LED control register to turn it on and off.
 */

#include <linux/module.h>
#include <linux/types.h>
#include <linux/kdev_t.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/slab.h>
#include <linux/uaccess.h>
#include <linux/pci.h>

#define DEVCNT 5
#define DEVNAME "my_pci_dev"

#define VENDOR_ID 0x8086
#define DEVICE_ID 0x150c
#define LED_REG   0x0E00

static struct mydev_dev {
    struct cdev cdev;
    int syscall_val; // not used in this assignment
    int led_val;    // store value for LED register
    void *hw_addr;  // base address of driver

} mydev;

static dev_t mydev_node;
static char *device_name = "my_pci_dev";

static DEFINE_PCI_DEVICE_TABLE(pci_test_tbl) = {
    { PCI_DEVICE(VENDOR_ID, DEVICE_ID) },
    { }, /* must have an empty at the end! */
};

static int my_pci_probe(struct pci_dev *pdev, const struct pci_device_id *ent)
{
    resource_size_t mmio_start, mmio_len;
    int bars, err;

    /* this is where I'd map BAR's for access, save stuff off, etc. */
    printk(KERN_INFO "It's dangerous to go alone, take this with you.\n");

    err = pci_enable_device_mem(pdev);

    // set up pci pci connections
    bars = pci_select_bars(pdev,IORESOURCE_MEM);
    err = pci_request_selected_regions(pdev,bars,device_name);

    pci_set_master(pdev);

    // map memory
    mmio_start = pci_resource_start(pdev, 0);
    mmio_len = pci_resource_len(pdev,0);
    mydev.hw_addr = ioremap(mmio_start,mmio_len);

    /* 0 means success */
    return 0;

```

```
}
// Assignment #3 pci remove function
static void my_pci_remove(struct pci_dev *pdev)
{
    iounmap(mydev.hw_addr);
    pci_release_selected_regions(pdev, pci_select_bars(pdev, IORESOURCE_MEM));
    pci_disable_device(pdev);

    printk(KERN_INFO "So long!!\n");
}

// Information about my PCI driver
static struct pci_driver my_pci_driver = {
    .name = DEVNAME,
    .id_table = pci_test_tbl,
    .probe = my_pci_probe,
    .remove = my_pci_remove,
};

// Open function
static int pci_hw3_open(struct inode *inode, struct file *file)
{
    printk(KERN_INFO "(my_pci_driver)successfully opened!\n");
    return 0;
}

// Read function
static ssize_t pci_hw3_read(struct file *file, char __user *buf,
                             size_t len, loff_t *offset)
{
    /* Get a local kernel buffer set aside */
    int ret;

    if (*offset >= sizeof(int))
        return 0;

    /* Make sure our user wasn't bad... */
    if (!buf) {
        ret = -EINVAL;
        goto out;
    }

    // Read the value in the LED register
    mydev.led_val = readl(mydev.hw_addr + LED_REG);

    // Pass LED value to userspace
    if (copy_to_user(buf, &mydev.led_val, sizeof(unsigned int))) {
        ret = -EFAULT;
        goto out;
    }
    ret = sizeof(unsigned int);
    *offset += len;

    /* Good to go, so printk the thingy */
    printk(KERN_INFO "(my_pci_driver:read)User got from us %d\n",mydev.led_val);

out:
    return ret;
}

// Write function
static ssize_t pci_hw3_write(struct file *file, const char __user *buf,
                              size_t len, loff_t *offset)
{
    int ret;

    /* Make sure our user isn't bad... */
    if (!buf) {
        ret = -EINVAL;
        goto out;
    }

    /* Copy from the user-provided buffer */
    if (copy_from_user(&mydev.led_val, buf, len)) {
        /* uh-oh... */
        ret = -EFAULT;
        goto out;
    }
    ret = len;

    // Display value we are going to write to LED control register
    printk(KERN_INFO "(my_pci_driver:write)Value to write to LED_CTRL register: 
    %d\n",mydev.led_val);

    // Write to LED control register
    writel((unsigned int)mydev.led_val,mydev.hw_addr + LED_REG);

out:
    return ret;
}

/* File operations for our device */
static struct file_operations mydev_fops = {
    .owner = THIS_MODULE,
    .open = pci_hw3_open,
    .read = pci_hw3_read,
    .write = pci_hw3_write,
};

// Initialization
static int __init pci_hw3_init(void)
{
    int ret;
    printk(KERN_INFO "(my_pci_driver) Registering PCI Driver...\n");
    ret = pci_register_driver(&my_pci_driver);

    printk(KERN_INFO "(my_pci_driver) module loading...\n");

    if (alloc_chrdev_region(&mydev_node, 0, DEVCNT, DEVNAME)) {
        printk(KERN_ERR "alloc_chrdev_region() failed!\n");
        return -1;
    }

    printk(KERN_INFO "Allocated %d devices at major: %d\n", DEVCNT,
           MAJOR(mydev_node));

    /* Initialize the character device and add it to the kernel */
    cdev_init(&mydev.cdev, &mydev_fops);
    mydev.cdev.owner = THIS_MODULE;

    if (cdev_add(&mydev.cdev, mydev_node, DEVCNT)) {
        printk(KERN_ERR "cdev_add() failed!\n");
        /* clean up chrdev allocation */
        unregister_chrdev_region(mydev_node, DEVCNT);

        return -1;
    }
        printk(KERN_INFO "(my_pci_driver)Tried to register pci driver. Return = 
        %d\n",ret);
    return ret;
}

```

```c
// Clean up when removing driver
static void __exit pci_hw3_exit(void)
{
    /* destroy the cdev */
    cdev_del(&mydev.cdev);

    /* Unregister PCI Driver*/
    pci_unregister_driver(&my_pci_driver);

    /* clean up the devices */
    unregister_chrdev_region(mydev_node, DEVCNT);
    printk(KERN_INFO "(my_pci_driver) module unloaded!\n");
}

MODULE_AUTHOR("Aaron Chan");
MODULE_LICENSE("GPL");
MODULE_VERSION("0.2");
module_init(pci_hw3_init);
module_exit(pci_hw3_exit);

```