```c
/*
 * Aaron Chan
 * ECE 373 (Spring 2017)
 * Assignment #4
 *
 * This module will initialize a PCI driver
 * so that we can blink LEDs on the Atom Box.
 * We will set a default blinking rate and include
 * the capability to change the blink rate with
 * values passed in from userspace.
 */

#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>
#include <linux/time.h>

#define DEVCNT 5
#define DEVNAME "my_timer_blink"
#define DEV_NODE_NAME "hw4_led"

#define LED_ON 78   //LED0 on value
#define LED_OFF 15  //LED0 off value

#define VENDOR_ID 0x8086
#define DEVICE_ID 0x150c
#define LED_REG   0x0E00 // LED control register offset

// =========== Global ==================
static struct mydev_dev {
    struct cdev cdev;
    int input;    // store value for LED register
    bool status;     // flag for LED, (on or off)
    void *hw_addr;  // base address of driver
} mydev;

static dev_t mydev_node;
static struct class *cl;    // device class
struct timer_list blinkLED;
//static char *device_name = "my_pci_dev";

static int blink_rate = 2;
module_param(blink_rate, int, S_IRUSR | S_IWUSR);

//========= Functions ====================
// Called by timer. Turn LED0 on and off by timer intervals
static void timer_blink(unsigned long data)
{
    // Turn LED0 on or off for blink
    if(mydev.status)
    {
        mydev.status = false;
        writel((unsigned int)LED_OFF,mydev.hw_addr + LED_REG);
    }
    else
    {
        mydev.status = true;
        writel((unsigned int)LED_ON,mydev.hw_addr + LED_REG);
```

```
64      }
65
66      // restart timer
67      mod_timer(&blinkLED, (HZ/blink_rate)+jiffies);
68
69  }
70
71  // Devices supported by this driver
72  static DEFINE_PCI_DEVICE_TABLE(pci_test_tbl) = {
73      { PCI_DEVICE(VENDOR_ID, DEVICE_ID) },
74      { }, /* must have an empty at the end! */
75  };
76
77  // Enable PCI device and map to memory
78  static int my_pci_probe(struct pci_dev *pdev, const struct pci_device_id *ent)
79  {
80      resource_size_t mmio_start, mmio_len;
81      int bars, err;
82
83      /* this is where I'd map BAR's for access, save stuff off, etc. */
84      printk(KERN_INFO "It's dangerous to go alone, take this with you.\n");
85
86      err = pci_enable_device_mem(pdev);
87
88      // set up pci pci connections
89      bars = pci_select_bars(pdev,IORESOURCE_MEM);
90      err = pci_request_selected_regions(pdev,bars,DEVNAME);
91
92      pci_set_master(pdev);
93
94      // map memory
95      mmio_start = pci_resource_start(pdev, 0);
96      mmio_len = pci_resource_len(pdev,0);
97      mydev.hw_addr = ioremap(mmio_start,mmio_len);
98
99      /* 0 means success */
100     return 0;
101
102 }
103
104 // Clean up PCI allocations, disable device
105 static void my_pci_remove(struct pci_dev *pdev)
106 {
107     iounmap(mydev.hw_addr);
108     pci_release_selected_regions(pdev, pci_select_bars(pdev, IORESOURCE_MEM));
109     pci_disable_device(pdev);
110
111     printk(KERN_INFO "So long!!\n");
112 }
113
114 // Information about my PCI driver
115 static struct pci_driver my_pci_driver = {
116     .name = DEVNAME,
117     .id_table = pci_test_tbl,
118     .probe = my_pci_probe,
119     .remove = my_pci_remove,
120 };
121
122 // Open function
123 static int pci_hw4_open(struct inode *inode, struct file *file)
124 {
125     printk(KERN_INFO "(my_pci_driver)successfully opened!\n");
126     // Initialize and turn on LED when userspace opens file
```

```
127     mydev.status = true;
128     writel((unsigned int)LED_ON,mydev.hw_addr + LED_REG);
129
130     // Start timer for blinking
131     mod_timer(&blinkLED, (HZ/blink_rate)+jiffies);
132     return 0;
133 }
134
135 // Release function
136 static int pci_hw4_release(struct inode *inode, struct file *file)
137 {
138     printk(KERN_INFO "(my_pci_driver)successfully closed!\n");
139
140     // Turn off LED when userspace closes file
141     mydev.status = false;
142     writel((unsigned int)LED_OFF,mydev.hw_addr + LED_REG);
143
144     // Stop and remove timer
145     del_timer_sync(&blinkLED);
146     return 0;
147 }
148 // Read function
149 static ssize_t pci_hw4_read(struct file *file, char __user *buf,
150                             size_t len, loff_t *offset)
151 {
152     /* Get a local kernel buffer set aside */
153     int ret;
154
155     if (*offset >= sizeof(int))
156         return 0;
157
158     /* Make sure our user wasn't bad... */
159     if (!buf) {
160         ret = -EINVAL;
161         goto out;
162     }
163
164     // Pass blink rate value to userspace
165     if (copy_to_user(buf, &blink_rate, sizeof(unsigned int))) {
166         ret = -EFAULT;
167         goto out;
168     }
169     ret = sizeof(unsigned int);
170     *offset += len;
171
172     /* Good to go, so printk the thingy */
173     printk(KERN_INFO "(my_pci_driver:read)User got from us %d\n",blink_rate);
174
175 out:
176     return ret;
177 }
178
179 // Write function
180 static ssize_t pci_hw4_write(struct file *file, const char __user *buf,
181                              size_t len, loff_t *offset)
182 {
183     int ret;
184     /* Make sure our user isn't bad... */
185     if (!buf) {
186         ret = -EINVAL;
187         goto out;
188     }
189
```

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

    if(mydev.input < 0)
    {
       printk("(my_pci_driver:write)User wrote negative value. Return error\n");
       ret = EINVAL;
        goto out;
    }
    else if(mydev.input == 0)
        printk("(my_pci_driver:write)User wrote 0. Do nothing\n");
    else
    {
        printk("(my_pci_driver:write)User wrote %d\n",mydev.input);
        blink_rate = mydev.input;
    }

    ret = len;

out:
    return ret;
}

/* File operations for our device */
static struct file_operations mydev_fops = {
    .owner = THIS_MODULE,
    .open = pci_hw4_open,
    .read = pci_hw4_read,
    .write = pci_hw4_write,
    .release = pci_hw4_release,
};

// Initialization
static int __init pci_hw4_init(void)
{
    int ret;
    mydev.status = false;
    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;
    }

    // Get major number for device
    printk(KERN_INFO "Allocated %d devices at major: %d\n", DEVCNT,
           MAJOR(mydev_node));

    // Create node file. No need for mknod
    if((cl = class_create( THIS_MODULE, DEVNAME)) == NULL)
    {
        printk(KERN_ALERT "Class creation failed\n");
        unregister_chrdev_region(mydev_node,DEVCNT);
        return -1;
    }
    if(device_create(cl, NULL, mydev_node, NULL, DEV_NODE_NAME) == NULL)
```

```
253      {
254          printk(KERN_ALERT "Device creation failed\n");
255          class_destroy(cl);
256          unregister_chrdev_region(mydev_node,DEVCNT);
257      }
258
259      /* Initialize the character device and add it to the kernel */
260      cdev_init(&mydev.cdev, &mydev_fops);
261      mydev.cdev.owner = THIS_MODULE;
262
263      if (cdev_add(&mydev.cdev, mydev_node, DEVCNT)) {
264          printk(KERN_ERR "cdev_add() failed!\n");
265          /* clean up chrdev allocation */
266          unregister_chrdev_region(mydev_node, DEVCNT);
267
268          return -1;
269      }
270
271      // Setup timer and start timer.
272      setup_timer(&blinkLED, timer_blink,0);
273      return ret;
274  }
275
276  // Clean up when removing driver
277  static void __exit pci_hw4_exit(void)
278  {
279      /* destroy the cdev */
280      cdev_del(&mydev.cdev);
281      device_destroy(cl,mydev_node);
282      class_destroy(cl);
283
284      /* Unregister PCI Driver*/
285      pci_unregister_driver(&my_pci_driver);
286
287      // Get rid of timer
288      del_timer_sync(&blinkLED);
289
290      /* clean up the devices */
291      unregister_chrdev_region(mydev_node, DEVCNT);
292      printk(KERN_INFO "(my_pci_driver) module unloaded!\n");
293  }
294
295  MODULE_AUTHOR("Aaron Chan");
296  MODULE_LICENSE("GPL");
297  MODULE_VERSION("0.2");
298  module_init(pci_hw4_init);
299  module_exit(pci_hw4_exit);
300
```