```
/*
 * Aaron Chan
 * ECE 373 (Spring 2017)
 * Assignment #6
 *
 * In assignment #4, we initialized a PCI driver
 * to blink the LEDs on the Atom Box using a timer
 * and adding ways to customize a blink rate to change
 * the speed that it blinks.
 *
 * In this assignment, we will make the following changes:
 *  - small set of legacy descriptors for the receive queue (16 descriptors)
      with buffers allocated to 2048 bytes
    - control mechanism for keeping track of HEAD and TAIL of receive queue
    - interrupt handler, tied to legacy interrupt source
    - workqueue thread to handle deferred processing from interrupt
    - confgiure code to put chip into promiscuous mode, and force link up at 1 Gb
    - Update read() system call to return 16-bit unsigned int, lower 8 bits being
      value received from receive queue TAIL, upper 8 bits being value from the HEAD.

    As suggested, this assignment is done with MSI instead of Legacy descriptors.
 */

#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/timer.h>
#include <linux/delay.h>
#include <linux/workqueue.h>
#include <linux/interrupt.h>

// =========== MACROS ===================
#define DEVCNT 5
#define DEVNAME "hw6_pci_interrupts"
#define DEV_NODE_NAME "hw6_interrupts"

// Device Info for Intel 82583v
#define VENDOR_ID 0x8086
#define DEVICE_ID 0x150c

#define LED_REG 0x00E00     // LED control register offset
#define LED_ON  0x4E4E0F    //Green LEDs on value
#define LED_OFF 0x0F0F0F    //All LEDs off value

// Chicken Bit necessities
#define GCR  0x5B00 // 3GIO Control Reg
#define GCR2 0x5B64 // 3GPIO Control Reg 2
#define MDIC 0x0020 // MDI Control Reg
#define STATUS 0x08

// Necessary for IRQ
#define ICR 0xC0        // Cause Read
#define ICS 0xC8        // Cause Setup
#define IMS 0xD0        // Mask Set
#define IMC 0xD8        // Mask Clear
#define CTRL 0x04       // Device Control Registering
#define RCTL 0x00100    // Receive Control Reg
```

```c
#define IRQ_ENABLE 0x11000D4
#define PROMISCUOUS 0x801A
#define MAC 0x100A41

// Receive Descriptor
#define MAX 16          // Number of Descriptors
#define RDBAL 0x2800    // Base Address Low
#define RDBAH 0x2804    // Base Address High
#define RDLEN 0x2808    // Length
#define RDH 0x2810      // Head
#define RDT 0x2818      // Tail

// =========== Globals ===================

// Receive Descriptor
struct rx_desc
{
    __le64 buffer_addr; // Address of descriptor's data buffer
    union
    {
        __le32 data;
        struct
        {
            __le16 length;
            __le16 css;
        }flags;
    }lower;
    union
    {
        __le32 data;
        struct
        {
            __u8 status;
            __u8 error;
            __le16 vlan;
        }field;
    }upper;
};

struct buf_info
{
    void* mem;
    dma_addr_t physical;
}buffer_info[MAX];

// Receive Ring
struct rx_ring
{
    void* desc;         //Point to ring memory
    dma_addr_t dma;     // Physical addr of ring
    struct rx_desc * cpu_addr; // descriptor addresses
    int size;           // Length of ring in bytes
    int count;          // Number of desc. in ring

    u16 next_to_use;
    u16 next_to_clean;
};

// Data that is important will be kept here
static struct mydev_dev {
    // struct net_device netdev;
    struct cdev cdev;
    int input;       // store value for LED register
    bool status;     // flag for LED, (on or off)

    struct rx_ring* rx_ring;
    void* hw_addr;    // hold base addr of driver
    struct work_struct service_task;
} mydev;

// For making device node file
static dev_t mydev_node;
static struct class *cl;

//========= Functions ====================

// Workqueue service task
// sleep for 0.5seconds, then turn off LEDs
static void hw6_service_task(struct work_struct* work)
{
    u32 tail, head;

    head = readl(mydev.hw_addr + RDH);
    tail = readl(mydev.hw_addr + RDT);

    printk(KERN_INFO "Value of HEAD: %d\n",head);
    printk(KERN_INFO "Value of TAIL: %d\n",tail);

    printk(KERN_INFO "Service task: SLEEP!\n");
    msleep(500);

    printk(KERN_INFO "Service task: LEDs off!\n");
    writel((unsigned int)LED_OFF,mydev.hw_addr + LED_REG);

    // bump tail
    if(tail >=16)
        writel(0, mydev.hw_addr + RDT);
    else
        writel(tail+1,mydev.hw_addr + RDT);

    // re-enable interrupts
    writel(IRQ_ENABLE, mydev.hw_addr + IMS);
}

// Interrupt Handler
// Turn on both green LEDs, then schedule work.
static irqreturn_t hw6_irq_handler(int irq, void* data)
{
        u32 cause;
    // disable interrupts
    writel(0xFFFFFFFF, mydev.hw_addr + IMC);

    printk(KERN_INFO "Interrupt: LEDs on!\n");
    writel((unsigned int)LED_ON, mydev.hw_addr + LED_REG);
    schedule_work(&mydev.service_task);

    // Read to clear interrupt bit
        cause = readl(mydev.hw_addr + ICR);
    printk(KERN_INFO "Cause from ICR: %x\n",cause);

    return IRQ_HANDLED;
}

// Setup resources for ring
```

```c
static void set_ring(struct pci_dev* pdev)
{
    int i; // for looping
    unsigned int reg;
    mydev.rx_ring = kzalloc(sizeof(struct rx_ring),GFP_KERNEL);
    mydev.rx_ring->count = MAX;
    mydev.rx_ring->size = sizeof(struct rx_desc) * MAX; //total size of all 16 descriptors
    mydev.rx_ring->size = ALIGN(mydev.rx_ring->size,2048);
    printk(KERN_INFO "ring size set and aligned\n");

    // Allocate and get addresses for ring
    mydev.rx_ring->desc = dma_alloc_coherent(&pdev->dev, mydev.rx_ring->size, &mydev.rx_ring->dma, GFP_KERNEL);
    printk(KERN_INFO "dma_alloc_coherent done!\n");

    reg = (mydev.rx_ring->dma >> 32) & 0xffffffff; // Higher
    printk(KERN_INFO "Higher: 0x%x \n", reg);
    writel(reg, mydev.hw_addr+RDBAH);

    reg = (mydev.rx_ring->dma) & 0xffffffff; // Lower
    printk(KERN_INFO "Lower: 0x%x \n", reg);
    writel(reg, mydev.hw_addr+RDBAL);

    writel(15, mydev.hw_addr+RDT);

    mydev.rx_ring->next_to_use = 0;
    mydev.rx_ring->next_to_clean = 0;
    mydev.rx_ring->cpu_addr = kzalloc(mydev.rx_ring->size, GFP_KERNEL);
    printk(KERN_INFO "Ring resources set~! Start filling descriptor buffer\n");

    // Set length for Receive Descriptors. Write to RDLEN
    writel(mydev.rx_ring->size, mydev.hw_addr + RDLEN);
    for(i=0;i<MAX;i++)
    {
        buffer_info[i].mem = kmalloc(2048,GFP_KERNEL);
        buffer_info[i].physical = dma_map_single(&pdev->dev, buffer_info[i].mem,2048, DMA_FROM_DEVICE);
        mydev.rx_ring->cpu_addr[i].buffer_addr = buffer_info[i].physical;
    }

}

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

// Enable PCI device and map to memory
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,DEVNAME);

    pci_set_master(pdev);

    // map memory, get base addr of desired device
    mmio_start = pci_resource_start(pdev, 0);
    mmio_len = pci_resource_len(pdev,0);
    mydev.hw_addr = ioremap(mmio_start,mmio_len);

    // Follow steps in 82583v Controller Datasheet
    // Steps for Software Initialization under 4.6
    // disable interrupts
    writel(0xffffffff, mydev.hw_addr+IMC);

    // Do device reset
    writel((1 << 26),mydev.hw_addr+CTRL);

    // Modify HEAD to point at zero
    writel(0,mydev.hw_addr+RDH);

    // Disable Interrupts again
    writel(0xFFFFFFFF, mydev.hw_addr+IMC);
    readl(mydev.hw_addr+STATUS); // read to flush register

    INIT_WORK(&mydev.service_task, hw6_service_task);

    // Needed for PCIe workarounds - reserved chicken bits
    // Write GCR bit 22, GCR bit 1
    writel((readl(mydev.hw_addr+GCR)) | (1 << 22), mydev.hw_addr+GCR);
    writel((readl(mydev.hw_addr+GCR2)) | 1, mydev.hw_addr+GCR2);

    // Needed for a forced PHY setup
    // PHY setup
    writel(0x1831af08, mydev.hw_addr+MDIC);

    // MAC setup
    // Set link up while preserving defaults.
    writel((readl(mydev.hw_addr + CTRL) | 0x40), mydev.hw_addr + CTRL);

    // Clear Status register by reading
    readl(mydev.hw_addr+STATUS);
    // Work Queue
    printk(KERN_INFO "Probe: Work initialized\n");

    // Setup Receive Ring
    printk(KERN_INFO "Probe: Setup ring resources!\n");
    set_ring(pdev);
    printk(KERN_INFO "Probe: Ring Resources set and descriptors filled!\n”);

    // Set interrupts. Enable MSI and request IRQ
    pci_enable_msi(pdev);
    err = request_irq(pdev->irq, hw6_irq_handler, 0, "Aaron's_IRQ",&mydev);

    // Enable interrupts
    writel(0x00000000, mydev.hw_addr + IMC); // Not sure this is needed
    writel(IRQ_ENABLE, mydev.hw_addr + IMS);
    printk(KERN_INFO "Probe:Interrupts Enabled\n");

    // Enable receiver and setup promiscuous
    writel(PROMISCUOUS, mydev.hw_addr + RCTL);
    printk(KERN_INFO "Probe: Receiver enabled!\n");

    /* 0 means success */
    return 0;

}

// Clean up PCI allocations, disable device
static void my_pci_remove(struct pci_dev *pdev)
{
    int i;

    // Cleanup Work Queue
    cancel_work_sync(&mydev.service_task);

    // Disable interrupts
    free_irq(pdev->irq,&mydev);
    pci_disable_msi(pdev);

    // Free and unpin memory for buffer info
    for(i=0;i<MAX;i++)
    {
        kfree(buffer_info[i].mem);
        dma_unmap_single(&pdev->dev,buffer_info->physical,2048,DMA_TO_DEVICE);
    }

    // Free ring
    dma_free_coherent(&pdev->dev, mydev.rx_ring->size, mydev.rx_ring->desc, mydev.rx_ring->dma);
    kfree(mydev.rx_ring->cpu_addr);
    kfree(mydev.rx_ring);

    // unmap pci device
    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");
}

// Name of my driver and associated functions
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_hw6_open(struct inode *inode, struct file *file)
{
    printk(KERN_INFO "(my_pci_driver)successfully opened!\n");
    return 0;
}

// Release function
static int pci_hw6_release(struct inode *inode, struct file *file)
{
    printk(KERN_INFO "(my_pci_driver)successfully closed!\n");
    return 0;
}
// Read function
static ssize_t pci_hw6_read(struct file *file, char __user *buf,
                             size_t len, loff_t *offset)
{
    /* Get a local kernel buffer set aside */
    int ret;
    u32 head_tail;
    head_tail= (readl(mydev.hw_addr + RDH) << 16) | readl(mydev.hw_addr + RDT);

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

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

    // Pass blink rate value to userspace
    if (copy_to_user(buf, &head_tail, 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",head_tail);

out:
    return ret;
}

// Write function
static ssize_t pci_hw6_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.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;

```

```c
out:
    return ret;
}

/* File operations for our device */
static struct file_operations mydev_fops = {
    .owner = THIS_MODULE,
    .open = pci_hw6_open,
    .read = pci_hw6_read,
    .write = pci_hw6_write,
    .release = pci_hw6_release,
};

// Initialization
static int __init pci_hw6_init(void)
{
    mydev.status = false;

    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)
    {
        printk(KERN_ALERT "Device creation failed\n");
        class_destroy(cl);
        unregister_chrdev_region(mydev_node,DEVCNT);
    }

    /* 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 "Node created\n");

    printk(KERN_INFO "(my_pci_driver) Registering PCI Driver...\n");
    return pci_register_driver(&my_pci_driver);
}

// Clean up when removing driver
static void __exit pci_hw6_exit(void)
```

```
{
    // Disable interrupts
    writel(0xffffffff, mydev.hw_addr+IMC);

    /* destroy the cdev */
    cdev_del(&mydev.cdev);
    device_destroy(cl,mydev_node);
    class_destroy(cl);

    /* 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_hw6_init);
module_exit(pci_hw6_exit);

```