look at lspci to find the device ID for the device in question

```
nathanm@Ubuntu:~$ lspci | grep Ethernet
00:03.0 Ethernet controller: Intel Corporation 82540EM Gigabit Ethernet Controller (rev 02)
nathanm@Ubuntu:~$ cd /sys/bus/pci
nathanm@Ubuntu:/sys/bus/pci$ ls
devices  drivers  drivers_autoprobe  drivers_probe  rescan  resource_alignment  slots  uevent
nathanm@Ubuntu:/sys/bus/pci$ ls -l
total 0
drwxr-xr-x  2 root root    0 May 23 14:22 devices
drwxr-xr-x 31 root root    0 May 23 14:22 drivers
-rw-r--r--  1 root root 4096 May 23 14:23 drivers_autoprobe
--w-------  1 root root 4096 May 23 14:23 drivers_probe
--w-------  1 root root 4096 May 23 14:23 rescan
-rw-r--r--  1 root root 4096 May 23 14:23 resource_alignment
drwxr-xr-x  2 root root    0 May 23 14:22 slots
--w-------  1 root root 4096 May 23 14:22 uevent
nathanm@Ubuntu:/sys/bus/pci$ cd devices
nathanm@Ubuntu:/sys/bus/pci/devices$ ls -l
total 0
lrwxrwxrwx 1 root root 0 May 23 14:22 0000:00:00.0 -> ../../../devices/pci0000:00/0000:00:00.0
lrwxrwxrwx 1 root root 0 May 23 14:22 0000:00:01.0 -> ../../../devices/pci0000:00/0000:00:01.0
lrwxrwxrwx 1 root root 0 May 23 14:22 0000:00:01.1 -> ../../../devices/pci0000:00/0000:00:01.1
lrwxrwxrwx 1 root root 0 May 23 14:22 0000:00:02.0 -> ../../../devices/pci0000:00/0000:00:02.0
lrwxrwxrwx 1 root root 0 May 23 14:22 0000:00:03.0 -> ../../../devices/pci0000:00/0000:00:03.0
lrwxrwxrwx 1 root root 0 May 23 14:22 0000:00:04.0 -> ../../../devices/pci0000:00/0000:00:04.0
lrwxrwxrwx 1 root root 0 May 23 14:22 0000:00:05.0 -> ../../../devices/pci0000:00/0000:00:05.0
lrwxrwxrwx 1 root root 0 May 23 14:22 0000:00:06.0 -> ../../../devices/pci0000:00/0000:00:06.0
lrwxrwxrwx 1 root root 0 May 23 14:22 0000:00:07.0 -> ../../../devices/pci0000:00/0000:00:07.0
lrwxrwxrwx 1 root root 0 May 23 14:22 0000:00:0b.0 -> ../../../devices/pci0000:00/0000:00:0b.0
lrwxrwxrwx 1 root root 0 May 23 14:22 0000:00:0d.0 -> ../../../devices/pci0000:00/0000:00:0d.0
nathanm@Ubuntu:/sys/bus/pci/devices$ cd 0000\:00\:03.0
nathanm@Ubuntu:/sys/bus/pci/devices/0000:00:03.0$ ls -l
total 0
-r--r--r-- 1 root root   4096 May 23 14:22 ari_enabled
-rw-r--r-- 1 root root   4096 May 23 14:22 broken_parity_status
-r--r--r-- 1 root root   4096 May 23 14:22 class
-rw-r--r-- 1 root root    256 May 23 14:22 config
-r--r--r-- 1 root root   4096 May 23 14:22 consistent_dma_mask_bits
-rw-r--r-- 1 root root   4096 May 23 14:22 d3cold_allowed
-r--r--r-- 1 root root   4096 May 23 14:22 device
-r--r--r-- 1 root root   4096 May 23 14:22 dma_mask_bits
lrwxrwxrwx 1 root root      0 May 23 14:22 driver -> ../../../bus/pci/drivers/e1000
-rw-r--r-- 1 root root   4096 May 23 14:22 driver_override
-rw-r--r-- 1 root root   4096 May 23 14:22 enable
-r--r--r-- 1 root root   4096 May 23 14:22 irq
drwxr-xr-x 2 root root      0 May 23 14:22 link
-r--r--r-- 1 root root   4096 May 23 14:22 local_cpulist
-r--r--r-- 1 root root   4096 May 23 14:22 local_cpus
-r--r--r-- 1 root root   4096 May 23 14:22 modalias
-rw-r--r-- 1 root root   4096 May 23 14:22 msi_bus
drwxr-xr-x 3 root root      0 May 23 14:22 net
-rw-r--r-- 1 root root   4096 May 23 14:22 numa_node
drwxr-xr-x 2 root root      0 May 23 14:22 power
-r--r--r-- 1 root root   4096 May 23 14:22 power_state
--w--w---- 1 root root   4096 May 23 14:22 remove
--w------- 1 root root   4096 May 23 14:22 rescan
--w------- 1 root root   4096 May 23 14:22 reset
-rw-r--r-- 1 root root   4096 May 23 14:22 reset_method
-r--r--r-- 1 root root   4096 May 23 14:22 resource
-rw------- 1 root root 131072 May 23 14:22 resource0
-rw------- 1 root root      8 May 23 14:22 resource2
-r--r--r-- 1 root root   4096 May 23 14:22 revision
lrwxrwxrwx 1 root root      0 May 23 14:22 subsystem -> ../../../bus/pci
-r--r--r-- 1 root root   4096 May 23 14:22 subsystem_device
-r--r--r-- 1 root root   4096 May 23 14:22 subsystem_vendor
-rw-r--r-- 1 root root   4096 May 23 14:22 uevent
-r--r--r-- 1 root root   4096 May 23 14:22 vendor
```

First unbind the device from the e1000e driver (remember to this in ROOT)

```
root@Ubuntu:/sys/bus/pci/devices/0000:00:03.0/driver# ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT group default qlen 1000
    link/ether 08:00:27:df:ca:0f brd ff:ff:ff:ff:ff:ff
root@Ubuntu:/sys/bus/pci/devices/0000:00:03.0/driver# ethtool -i enp0s3
driver: e1000
version: 5.19.0-41-generic
firmware-version:
expansion-rom-version:
bus-info: 0000:00:03.0
supports-statistics: yes
supports-test: yes
supports-eeprom-access: yes
supports-register-dump: yes
supports-priv-flags: no
root@Ubuntu:/sys/bus/pci/devices/0000:00:03.0/driver#
```

If you do “ip link”, you'll notice that enp0s3 is the device that is attached to this VM

remember ethtool? Use ethtool -i enp0s3

you can see that the driver is: e1000

UNBIND:

```
root@Ubuntu:/sys/bus/pci/devices/0000:00:03.0/driver# echo "0000:00:03.0" > unbind
root@Ubuntu:/sys/bus/pci/devices/0000:00:03.0/driver# ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
root@Ubuntu:/sys/bus/pci/devices/0000:00:03.0/driver#
```

Notice that after the unbind command, ip link does not show the 2nd device anymore, and only the 1st loopback device

BIND (again)

When you do bind, this will envoke the probe() function in the e1000 driver, and it will load and setup the device driver

```
root@Ubuntu:/sys/bus/pci/devices/0000:00:03.0/driver# echo "0000:00:03.0" > unbind
root@Ubuntu:/sys/bus/pci/devices/0000:00:03.0/driver# ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
root@Ubuntu:/sys/bus/pci/devices/0000:00:03.0/driver# echo "0000:00:03.0" > bind
root@Ubuntu:/sys/bus/pci/devices/0000:00:03.0/driver# ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
3: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT group default qlen 1000
    link/ether 08:00:27:df:ca:0f brd ff:ff:ff:ff:ff:ff
root@Ubuntu:/sys/bus/pci/devices/0000:00:03.0/driver#
```

Again, ip link, will show you that the device (enp0s3) is binded to the e1000 driver

Bind the device to YOUR driver (Need ROOT)

**** Note, this is only after you have load your own .ko module into the kernel****

echo "0000:00:03.0" > /sys/module/pci_led/drivers/pci\:HW3_pci_intel/bind

the directory here is strictly different and is base on when create your dev directory. Refer to previous assignment on how to do this. You have done this in the past previous assignments.

Attach the device by implementing the .probe() function callback for PCI devices.

```
/* device table */

static const struct pci_device_id pci_tbl[] = {

        { PCI_DEVICE(PCI_DEVICE_ID_INTEL, PCI_DEVICE_82545EM), 0, 0, 0 },
        /* Required null terminator */
        {0, }
};


/* PCI device object */

struct pci_device {
        struct pci_dev *pdev;
        void *hw_addr;
};
struct pci_device *intel;


#define PCI_DEVICE_ID_INTEL 0x8086
#define PCI_DEVICE_82545EM 0x100F
#define INTEL_LED_CONTROL 0x00E00
#define INTEL_LED_MASK 0x0F
#define INTEL_REG_CONFIG1 0x00
```

static int devprobe(struct pci_dev *pdev, const struct pci_device_id *ent)

```
{

        /* Enable the device memory */

        err = pci_enable_device_mem(pdev);


        /* Set up for high or low dma */

        err = dma_set_mask(&pdev->dev, DMA_BIT_MASK(64));
        If (err) {
                dev_err(&pdev->dev, "DMA configuration failed: 0x%x\n", err);
                goto err_dma;
```

```
        }

/* Set up pci connections */

err = pci_request_selected_regions(pdev, pci_select_bars(pdev,
        IORESOURCE_MEM), driver_name);
        if (err) {
                        dev_info(&pdev->dev, "pci_request_selected_regions failed %d\n", err);
        goto err_pci_reg;
}


/* Set the device as a master */

pci_set_master(pdev);


/* Create an instance of the pci device */
intel = kzalloc(sizeof(*intel), GFP_KERNEL);
        if (!intel) {
        err = -ENOMEM;
        goto err_intel_alloc;
}
intel->pdev = pdev;
pci_set_drvdata(pdev, intel);
```

Make sure to map the BAR for register access, and store the physical address of the BAR for use later.

```
/* Map device memory */

ioremap_len = min_t(int, pci_resource_len(pdev, 0), 1024);
intel->hw_addr = ioremap(pci_resource_start(pdev, 0), ioremap_len);

if (!intel->hw_addr) {
        err = -EIO;
        dev_info(&pdev->dev, "ioremap(0x%04x, 0x%04x) failed: 0x%x\n",
        (unsigned int)pci_resource_start(pdev, 0),
        (unsigned int)pci_resource_len(pdev, 0), err);
        goto err_ioremap;
}

/* Readbase register cofiguration */
config1 = readb(intel->hw_addr + INTEL_REG_CONFIG1);
dev_info(&pdev->dev, "config1 = 0x%02x\n", config1);
```

```
/* Update dev_info with new led value if required */
if (new_leds) {
        config1 = (config1 & ~INTEL_REG_CONFIG1) |
        (new_leds & INTEL_LED_MASK);

        writeb(config1, (intel->hw_addr + INTEL_REG_CONFIG1));
        dev_info(&pdev->dev, "new config1 = 0x%02x\n", config1);
}
```

Don't forget to properly clean up the device with a .remove() routine.

```
static void devremove(struct pci_dev *pdev)

{
        struct pci_device *intel = pci_get_drvdata(pdev);

        printk(KERN_INFO "devremove enter\n");

        /* Unmap device from memory */
        iounmap(intel->hw_addr);

        /* Free any allocated memory */
        kfree(intel);
        pci_release_selected_regions(pdev,
        pci_select_bars(pdev, IORESOURCE_MEM));

        /* Disable the device */
        pci_disable_device(pdev);
        printk(KERN_INFO "devremove exit\n");
}
```

You'll also need to figure out which header file to include.

```
#include <linux/init.h>

#include <linux/module.h>

#include <linux/types.h>

#include <linux/kdev_t.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/usb.h>
#include <linux/slab.h>
#include <linux/uaccess.h>
#include <linux/kernel.h>
#include <linux/errno.h>
#include <linux/pci.h>
```

hook up the PCI driver into your init_module() routine

```
/* Allocate memory for device */

/* Create a device class to be used for the char device file */

/* Initialize char device and add it to the kernel */

/* Create a device file in /dev */


/* Register pci driver */

        ret = pci_register_driver(&hw3_pci_driver);
```

subsequent unregister in your exit_module()

```
/* Destroy the cdev */

cdev_del(&mydev.cdev);

/* Clean up the devices */
unregister_chrdev_region(mydev_node, DEVCNT);
pci_unregister_driver(&hw3_pci_driver);
```

Part 7 - Read/Write implementation should be straight forward.

Modify your read() implementation so it will read the LED register, using the register information you found above in problem 3.

Modify your write() implementation so it will write the LED register with the value supplied in the write() system call.

/* File operations for the device */

```
static struct file_operations mydev_fops =
{
        .owner = THIS_MODULE,
        .open = devopen,
        .release = devrelease,
        .read = devread,
        .write = devwrite,
};
```

Make sure to at least verify the data is somewhat valid...(make sure it's a 32-bit number).

Sleep for 2 seconds,

```
/* Sleep for 2 seconds */
        printf("\n\n\t\t*** sleeping ***\n\n");
        usleep(2000000);
```

- Remember usleep? We went over this in-class

-

Reference: https://elixir.bootlin.com/linux/latest/source/drivers/net/ethernet/intel/e1000/e1000_main.c