MIT 6.828: Operating System Engineering

Lab Report
Lab 6: Network Driver

CSNLP
csnlp16@126.com
https://csnlp.github.io/

Contents

1 The Network Server

The network server is actually a combination of four environments:

- Core network server environment (includes socket call dispatcher and IwIP).
- Input environment.
- Output environment.
- Timer environment.

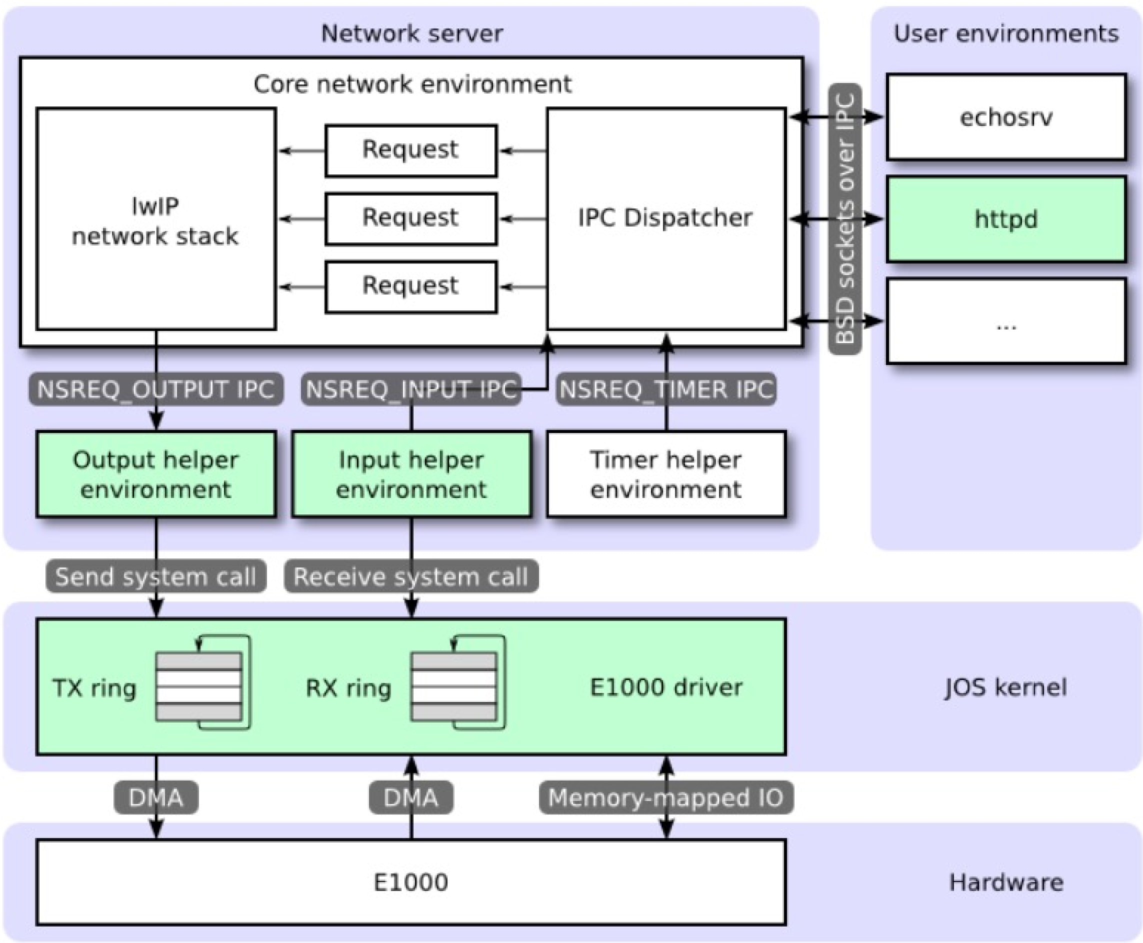

Figure 1: Different environments

1.1 The Core Network Server Environment

The core network server environment is composed of:

- The socket call dispatcher.
- IwIP.

1.2 The Output Environment

1.3 The Input Environment

1.4 The Timer Environment

2 Part A: Initialization and Transmitting Packets

2.1 Exercise 1

There is currently a clock interrupt that is generated by the hardware every 10ms. On every clock interrupt we can increment a variable to indicate that time has advanced by 10ms. This is implemented in kern/time.c, but is not yet fully integrated into your kernel.

Exercise 1. Add a call to `time_tick` for every clock interrupt in `kern/trap.c`. Implement `sys_time_msec` and add it to `syscall` in `kern/syscall.c` so that user space has access to the time.

Figure 2: Exercise 1

2.1.1 The Old Clock Interrupt in kern/trap.c of LAB 4

```
		// Handle clock interrupts. Don't forget to acknowledge the
		// interrupt using lapic_eoi() before calling the scheduler!
		// LAB 4: Your code here.
		case IRQ_OFFSET + IRQ_TIMER:
			lapic_eoi();
			sched_yield();
			return;
```

2.1.2 The New Clock Interrupt in kern/trap.c of LAB 6

Just add one line

```
		// Handle clock interrupts. Don't forget to acknowledge the
		// interrupt using lapic_eoi() before calling the scheduler!
		// LAB 4: Your code here.
		case IRQ_OFFSET + IRQ_TIMER:
			lapic_eoi();
			time_tick();
			sched_yield();
			return;
```

See the time_tick() in kern/time.c

```
// This should be called once per timer interrupt.  A timer interrupt
// fires every 10 ms.
void
time_tick(void)
{
	ticks++;
	if (ticks * 10 < ticks)
		panic("time_tick: time overflowed");
}
```

2.1.3 The Implementation of sys_time_msec() in kern/syscall.c

```c
// Return the current time.
static int
sys_time_msec(void)
{
        // LAB 6: Your code here.
        //panic("sys_time_msec not implemented");
        return time_msec();
}
```

time_msec() in kern/time.c:

```c
unsigned int
time_msec(void)
{
        return ticks * 10;
}
```

2.1.4 Don't Forget syscall() in kern/syscall.c

```c
        // LAB 5: exercise 1: sys_time_msec();
        case SYS_time_msec:
                return sys_time_msec();
```

2.1.5 See The Output

```
cui@cui-VirtualBox:~/mit6828/lab$ make INIT_CFLAGS=-DTEST_NO_NS run-testtime
make[1]: Entering directory '/home/cui/mit6828/lab'
+ cc kern/trap.c
+ cc kern/syscall.c
+ ld obj/kern/kernel
+ mk obj/kern/kernel.img
make[1]: 'obj/fs/fs.img' is up to date.
make[1]: Leaving directory '/home/cui/mit6828/lab'
qemu-system-i386 -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -serial mon:stdio -gdb tcp::26000 -D qemu.log -smp 1 -drive file=obj/fs/fs.img,index=1,media=disk,format=raw -net user -net nic,model=e1000 -redir tcp:26001::7 -redir tcp:26002::80 -redir udp:26001::7 -net dump,file=qemu.pcap
6828 decimal is 15254 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2 4
PCI: 00:00.0: 8086:1237: class: 6.0 (Bridge device) irq: 0
PCI: 00:01.0: 8086:7000: class: 6.1 (Bridge device) irq: 0
PCI: 00:01.1: 8086:7010: class: 1.1 (Storage controller) irq: 0
PCI: 00:01.3: 8086:7113: class: 6.80 (Bridge device) irq: 9
PCI: 00:02.0: 1013:00b8: class: 3.0 (Display controller) irq: 0
PCI: 00:03.0: 8086:100e: class: 2.0 (Network controller) irq: 11
FS is running
FS can do I/O
Device 1 presence: 1
```

```
block cache is good
superblock is good
bitmap is good
starting count down: 5 4 3 2 1 0
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf034207c from CPU 0
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdfd0
  oesp 0xefffffdc
  ebx  0xffffffff
  edx  0xeebfde88
  ecx  0x00000001
  eax  0x00000001
  es   0x----0023
  ds   0x----0023
  trap 0x00000003 Breakpoint
  err  0x00000000
  eip  0x00800110
  cs   0x----001b
  flag 0x00000292
  esp  0xeebfdfb8
  ss   0x----0023
K>
```

2.2 The Network Interface Card

2.2.1 Exercise 2

2.2.2 PCI Interface

Exercise 2. Browse Intel's Software Developer's Manual for the E1000. This manual covers several closely related Ethernet controllers. QEMU emulates the 82540EM.

You should skim over chapter 2 now to get a feel for the device. To write your driver, you'll need to be familiar with chapters 3 and 14, as well as 4.1 (though not 4.1's subsections). You'll also need to use chapter 13 as reference. The other chapters mostly cover components of the E1000 that your driver won't have to interact with. Don't worry about the details right now; just get a feel for how the document is structured so you can find things later.

While reading the manual, keep in mind that the E1000 is a sophisticated device with many advanced features. A working E1000 driver only needs a fraction of the features and interfaces that the NIC provides. Think carefully about the easiest way to interface with the card. We strongly recommend that you get a basic driver working before taking advantage of the advanced features.

Figure 3: Exercise 2

A PCI device needs to be discovered and initialized before it can be used:

- **Discovery** is the process of walking the PCI bus looking for attached devices.
- **Initialization** is the process of allocating I/O and memory space as well as negotiating the IRQ line for the device to use.

2.2.2.1 Discovery

The PCi code walks the PCI bus looking for devices. When it finds a device, it reads its vendor ID and device ID and uses these two values as the key to search the pci_attack_vendor array. The array is composed of struct pci_driver entries like this:

```
struct pci_driver {
    uint32_t key1, key2;
    int (*attachfn) (struct pci|_func *pcif);
}
```

2.2.2.2 Initialization

If the discovered device's vendor ID and device ID match one entry in the array, the PCI code calls that entry's attachfn to perform device initialization.

2.2.3 Exercise 3

Exercise 3. Implement an attach function to initialize the E1000. Add an entry to the `pci_attach_vendor` array in `kern/pci.c` to trigger your function if a matching PCI device is found (be sure to put it before the `{0, 0, 0}` entry that mark the end of the table). You can find the vendor ID and device ID of the 82540EM that QEMU emulates in section 5.2. You should also see these listed when JOS scans the PCI bus while booting.

For now, just enable the E1000 device via `pci_func_enable`. We'll add more initialization throughout the lab.

We have provided the `kern/e1000.c` and `kern/e1000.h` files for you so that you do not need to mess with the build system. They are currently blank; you need to fill them in for this exercise. You may also need to include the `e1000.h` file in other places in the kernel.

When you boot your kernel, you should see it print that the PCI function of the E1000 card was enabled. Your code should now pass the `pci attach` test of **`make grade`**.

Figure 4: Exercise 3

2.2.3.1 The code in kern/pci.h

```c
#ifndef JOS_KERN_PCI_H
#define JOS_KERN_PCI_H

#include <inc/types.h>

// PCI subsystem interface
enum { pci_res_bus, pci_res_mem, pci_res_io, pci_res_max };

struct pci_bus;

struct pci_func {
    struct pci_bus *bus;        // Primary bus for bridges

    uint32_t dev;
    uint32_t func;

    uint32_t dev_id;
    uint32_t dev_class;

    uint32_t reg_base[6];
    uint32_t reg_size[6];
    uint8_t irq_line;
};

struct pci_bus {
    struct pci_func *parent_bridge;
    uint32_t busno;
};

int  pci_init(void);
void pci_func_enable(struct pci_func *f);

#endif
```

2.2.3.2 The code in kern/pci.c

Let's first check kern/pci.c

```c
#include <inc/x86.h>
#include <inc/assert.h>
#include <inc/string.h>
#include <kern/pci.h>
#include <kern/pcireg.h>
#include <kern/e1000.h>

// Flag to do "lspci" at bootup
static int pci_show_devs = 1;
static int pci_show_addrs = 0;

// PCI "configuration mechanism one"
static uint32_t pci_conf1_addr_ioport = 0x0cf8;
static uint32_t pci_conf1_data_ioport = 0x0cfc;

// Forward declarations
static int pci_bridge_attach(struct pci_func *pcif);

// PCI driver table
struct pci_driver {
        uint32_t key1, key2;
```

```c
	int (*attachfn) (struct pci_func *pcif);
};
// pci_attach_class matches the class and subclass of a PCI device
struct pci_driver pci_attach_class[] = {
	{ PCI_CLASS_BRIDGE, PCI_SUBCLASS_BRIDGE_PCI, &pci_bridge_attach },
	{ 0, 0, 0 },
};

// pci_attach_vendor matches the vendor ID and device ID of a PCI device. key1
// and key2 should be the vendor ID and device ID respectively
struct pci_driver pci_attach_vendor[] = {
	{ 0, 0, 0 },
};

static void
pci_conf1_set_addr(uint32_t bus,
		   uint32_t dev,
		   uint32_t func,
		   uint32_t offset)
{
	assert(bus < 256);
	assert(dev < 32);
	assert(func < 8);
	assert(offset < 256);
	assert((offset & 0x3) == 0);

	uint32_t v = (1 << 31) |		// config-space
		(bus << 16) | (dev << 11) | (func << 8) | (offset);
	outl(pci_conf1_addr_ioport, v);
}

static uint32_t
pci_conf_read(struct pci_func *f, uint32_t off)
{
	pci_conf1_set_addr(f->bus->busno, f->dev, f->func, off);
	return inl(pci_conf1_data_ioport);
}

static void
pci_conf_write(struct pci_func *f, uint32_t off, uint32_t v)
{
	pci_conf1_set_addr(f->bus->busno, f->dev, f->func, off);
	outl(pci_conf1_data_ioport, v);
}

static int __attribute__((warn_unused_result))
pci_attach_match(uint32_t key1, uint32_t key2,
		 struct pci_driver *list, struct pci_func *pcif)
{
	uint32_t i;

	for (i = 0; list[i].attachfn; i++) {
		if (list[i].key1 == key1 && list[i].key2 == key2) {
			int r = list[i].attachfn(pcif);
			if (r > 0)
				return r;
			if (r < 0)
				cprintf("pci_attach_match: attaching "
```

```
                                        "%x.%x (%p): e\n",
                                        key1, key2, list[i].attachfn, r);
                }
        }
        return 0;
}

static int
pci_attach(struct pci_func *f)
{
        return
                pci_attach_match(PCI_CLASS(f->dev_class),
                                 PCI_SUBCLASS(f->dev_class),
                                 &pci_attach_class[0], f) ||
                pci_attach_match(PCI_VENDOR(f->dev_id),
                                 PCI_PRODUCT(f->dev_id),
                                 &pci_attach_vendor[0], f);
}

static const char *pci_class[] =
{
        [0x0] = "Unknown",
        [0x1] = "Storage controller",
        [0x2] = "Network controller",
        [0x3] = "Display controller",
        [0x4] = "Multimedia device",
        [0x5] = "Memory controller",
        [0x6] = "Bridge device",
};

static void
pci_print_func(struct pci_func *f)
{
        const char *class = pci_class[0];
        if (PCI_CLASS(f->dev_class) < ARRAY_SIZE(pci_class))
                class = pci_class[PCI_CLASS(f->dev_class)];

        cprintf("PCI: %02x:%02x.%d: %04x:%04x: class: %x.%x (%s) irq: %d\n",
                f->bus->busno, f->dev, f->func,
                PCI_VENDOR(f->dev_id), PCI_PRODUCT(f->dev_id),
                PCI_CLASS(f->dev_class), PCI_SUBCLASS(f->dev_class), class,
                f->irq_line);
}

static int
pci_scan_bus(struct pci_bus *bus)
{
        int totaldev = 0;
        struct pci_func df;
        memset(&df, 0, sizeof(df));
        df.bus = bus;

        for (df.dev = 0; df.dev < 32; df.dev++) {
                uint32_t bhlc = pci_conf_read(&df, PCI_BHLC_REG);
                if (PCI_HDRTYPE_TYPE(bhlc) > 1)     // Unsupported or no device
                        continue;

                totaldev++;
```

```

		struct pci_func f = df;
		for (f.func = 0; f.func < (PCI_HDRTYPE_MULTIFN(bhlc) ? 8 : 1);
		     f.func++) {
			struct pci_func af = f;

			af.dev_id = pci_conf_read(&f, PCI_ID_REG);
			if (PCI_VENDOR(af.dev_id) == 0xffff)
				continue;

			uint32_t intr = pci_conf_read(&af, PCI_INTERRUPT_REG);
			af.irq_line = PCI_INTERRUPT_LINE(intr);

			af.dev_class = pci_conf_read(&af, PCI_CLASS_REG);
			if (pci_show_devs)
				pci_print_func(&af);
			pci_attach(&af);
		}
	}

	return totaldev;
}

static int
pci_bridge_attach(struct pci_func *pcif)
{
	uint32_t ioreg  = pci_conf_read(pcif, PCI_BRIDGE_STATIO_REG);
	uint32_t busreg = pci_conf_read(pcif, PCI_BRIDGE_BUS_REG);

	if (PCI_BRIDGE_IO_32BITS(ioreg)) {
		cprintf("PCI: %02x:%02x.%d: 32-bit bridge IO not supported.\n",
			pcif->bus->busno, pcif->dev, pcif->func);
		return 0;
	}

	struct pci_bus nbus;
	memset(&nbus, 0, sizeof(nbus));
	nbus.parent_bridge = pcif;
	nbus.busno = (busreg >> PCI_BRIDGE_BUS_SECONDARY_SHIFT) & 0xff;

	if (pci_show_devs)
		cprintf("PCI: %02x:%02x.%d: bridge to PCI bus %d--%d\n",
			pcif->bus->busno, pcif->dev, pcif->func,
			nbus.busno,
			(busreg >> PCI_BRIDGE_BUS_SUBORDINATE_SHIFT) & 0xff);

	pci_scan_bus(&nbus);
	return 1;
}

void
pci_func_enable(struct pci_func *f)
{
	pci_conf_write(f, PCI_COMMAND_STATUS_REG,
		       PCI_COMMAND_IO_ENABLE |
		       PCI_COMMAND_MEM_ENABLE |
		       PCI_COMMAND_MASTER_ENABLE);

```

```
        uint32_t bar_width;
        uint32_t bar;
        for (bar = PCI_MAPREG_START; bar < PCI_MAPREG_END;
             bar += bar_width)
        {
                uint32_t oldv = pci_conf_read(f, bar);

                bar_width = 4;
                pci_conf_write(f, bar, 0xffffffff);
                uint32_t rv = pci_conf_read(f, bar);

                if (rv == 0)
                        continue;

                int regnum = PCI_MAPREG_NUM(bar);
                uint32_t base, size;
                if (PCI_MAPREG_TYPE(rv) == PCI_MAPREG_TYPE_MEM) {
                        if (PCI_MAPREG_MEM_TYPE(rv) == PCI_MAPREG_MEM_TYPE_64BIT)
                                bar_width = 8;

                        size = PCI_MAPREG_MEM_SIZE(rv);
                        base = PCI_MAPREG_MEM_ADDR(oldv);
                        if (pci_show_addrs)
                                cprintf("  mem region %d: %d bytes at 0x%x\n",
                                        regnum, size, base);
                } else {
                        size = PCI_MAPREG_IO_SIZE(rv);
                        base = PCI_MAPREG_IO_ADDR(oldv);
                        if (pci_show_addrs)
                                cprintf("  mem region %d: %d bytes at 0x%x\n",
                                        regnum, size, base);
                } else {
                        size = PCI_MAPREG_IO_SIZE(rv);
                        base = PCI_MAPREG_IO_ADDR(oldv);
                        if (pci_show_addrs)
                                cprintf("  io region %d: %d bytes at 0x%x\n",
                                        regnum, size, base);
                }

                pci_conf_write(f, bar, oldv);
                f->reg_base[regnum] = base;
                f->reg_size[regnum] = size;

                if (size && !base)
                        cprintf("PCI device %02x:%02x.%d (%04x:%04x) "
                                "may be misconfigured: "
                                "region %d: base 0x%x, size %d\n",
                                f->bus->busno, f->dev, f->func,
                                PCI_VENDOR(f->dev_id), PCI_PRODUCT(f->dev_id),
                                regnum, base, size);
        }

        cprintf("PCI function %02x:%02x.%d (%04x:%04x) enabled\n",
                f->bus->busno, f->dev, f->func,
                PCI_VENDOR(f->dev_id), PCI_PRODUCT(f->dev_id));
}

int
```

```c
pci_init(void)
{
        static struct pci_bus root_bus;
        memset(&root_bus, 0, sizeof(root_bus));

        return pci_scan_bus(&root_bus);
}
```

After starting the OS, we can have:

```
enabled interrupts: 1 2 4
PCI: 00:00.0: 8086:1237: class: 6.0 (Bridge device) irq: 0
PCI: 00:01.0: 8086:7000: class: 6.1 (Bridge device) irq: 0
PCI: 00:01.1: 8086:7010: class: 1.1 (Storage controller) irq: 0
PCI: 00:01.3: 8086:7113: class: 6.80 (Bridge device) irq: 9
PCI: 00:02.0: 1013:00b8: class: 3.0 (Display controller) irq: 0
PCI: 00:03.0: 8086:100e: class: 2.0 (Network controller) irq: 11
```

2.2.3.3 Get the vendor ID and device ID of E1000

2.2.3.4 Step 1: Add an entry to the pci_attach_vendor array in kern/pci.c to trigger your function if a matching PCI device is found

```c
// pci_attach_vendor matches the vendor ID and device ID of a PCI device. key1
// and key2 should be the vendor ID and device ID respectively
struct pci_driver pci_attach_vendor[] = {
        { 0, 0, 0 },
};
```

```c
struct pci_driver pci_attach_vendor[] = {
        {PCI_E1000_VENDOR, PCI_E1000_DEVICE, &pci_e1000_attach},
        { 0, 0, 0 },
};
```

Just add macro the e1000 vendor ID and e1000 device ID in kern/e1000.h

```c
// LAB 6. Exercise 3
#define PCI_E1000_VENDOR 0x8086
#define PCI_E1000_DEVICE 0x100E
```

2.2.3.5 Step 2: pci_e1000_attach() function

Firstly, declare it in kern/pci.c. Then implement it in kern/e1000.c

```c
// LAB 6: exercise 3: declare pci_e1000_attach function
int pci_e1000_attach(struct pci_func *pcif);
```

```c
// LAB 6: exercise 3
int
pci_e1000_attach(struct pci_func *pcif)
{
        pci_func_enable(pcif);
        return 0;
}
```

2.2.4 Exercise 3: Again

2.2.4.1 kern/e1000.c

Define the e1000_attach_func and further include < *kern/pmap.h* >

```c
int
e1000_attach_func(struct pci_func *pcif)
{
        pci_func_enable(pcif);
        cprintf("reg_base: %x, reg_size: %x\n",
                pcif->reg_base[0], pcif->reg_size[0]);
        return 0;
}
```

2.2.4.2 kern/e1000.h

Define the vendor ID and device ID of 82540EM e1000.

```c
#ifndef JOS_KERN_E1000_H
#define JOS_KERN_E1000_H

// LAB 6. Exercise 3
#define E1000_VENDOR_ID 0x8086
#define E1000_DEVICE_ID 0x100E

#include <kern/pci.h>
int e100_attach_func(struct pci_func *pcif);


#endif  // SOL >= 6
```

2.2.4.3 kern/pci.c

```c
// pci_attach_vendor matches the vendor ID and device ID of a PCI device. key1
// and key2 should be the vendor ID and device ID respectively
struct pci_driver pci_attach_vendor[] = {
        {E1000_VENDOR_ID, E1000_DEVICE_ID, &e1000_attach_func},
        { 0, 0, 0 },
};
```

2.2.5 Present Score

```
testtime: OK (7.4s)
pci attach: OK (1.1s)
testoutput [5 packets]: FAIL (1.7s)
```

Part A score: 10/35

2.2.6 Memory-mapped I/O

2.2.7 Exercise 4

> **Exercise 4.** In your attach function, create a virtual memory mapping for the E1000's BAR 0 by calling `mmio_map_region` (which you wrote in lab 4 to support memory-mapping the LAPIC).
>
> You'll want to record the location of this mapping in a variable so you can later access the registers you just mapped. Take a look at the `lapic` variable in `kern/lapic.c` for an example of one way to do this. If you do use a pointer to the device register mapping, be sure to declare it `volatile`; otherwise, the compiler is allowed to cache values and reorder accesses to this memory.
>
> To test your mapping, try printing out the device status register (section 13.4.2). This is a 4 byte register that starts at byte 8 of the register space. You should get `0x80080783`, which indicates a full duplex link is up at 1000 MB/s, among other things.

Figure 5: Exercise 4

2.2.7.1 Declare the variable

As the suggestion, record the location of this mapping in a variable. Declare it as bar_va.

```
volatile void *bar_va;
#define E1000REG(offset) (void *) (bar_va + offset)
```

Listing 1: volatile variable in kern/e1000.c

2.2.7.2 mmio_map_region in e1000_attach_func in kern/e1000.c

```
int
e1000_attach_func(struct pci_func *pcif)
{
        pci_func_enable(pcif);
        cprintf("reg_base: %x, reg_size: %x\n",
                pcif->reg_base[0], pcif->reg_size[0]);
        bar_va = mmio_map_region(pcif->reg_base[0], pcif->reg_size[0]);
        uint32_t *status_reg = (uint32_t *)E1000REG(E1000_STATUS);
        assert(*status_reg == 0x80080783);
        return 0;
}
```

2.2.7.3 Define E1000_STATUS in kern/e1000.h

```c
#define E1000_STATUS    0x00008 // LAB 6: Exercise 4
```

2.2.8 Exercise 4: Solution Again

2.2.8.1 kern/e1000.c

```c
volatile void *bar_va;
#define E1000REG(offset) (void *) (bar_va + offset)
```

2.2.9 DMA

Transmitting and receiving packets by writing and reading from the E1000's registers is quite slow. So, E1000 uses Direct Memory Access (DMA) to read and write packet data directly from memory without involving the CPU. The driver is responsible for:

- Allocating memory for the transmit and received queues.
- Setting up DMA descriptors.
- Configuring the E1000 with the location of these queues.

2.3 Transmitting Packets

2.3.1 C Structure

Consider the legacy transmit descriptor given in Table 3-8 of the manual.

```
63            48 47   40 39   32 31   24 23   16 15             0
+---------------------------------------------------------------+
|                         Buffer address                        |
+---------------+-------+-------+-------+-------+---------------+
|    Special    |  CSS  | Status|  Cmd  |  CSO  |    Length     |
+---------------+-------+-------+-------+-------+---------------+
```

Figure 6: Legacy transmit descriptor

The structure of C structure:

```c
// C structure
#include <inc/types.h>
struct tx_desc
{
        uint64_t addr;
        uint16_t length;
        uint8_t cso;
        uint8_t cmd;
        uint8_t status;
        uint8_t css;
        uint16_t special;
} __attribute__((packed));

struct rx_desc
{
        uint64_t addr;
```

```c
        uint16_t length;
        uint8_t cso;
        uint8_t cmd;
        uint8_t status;
        uint8_t css;
        uint16_t special;
} __attribute__((packed));

// main body of packet
struct packet
{
        char body[2048];
};
```

Listing 2: C struct in kern/e1000.h

tx_desc, rx_desc is about the send and receive descriptor.

2.3.2 Exercise 5

Exercise 5. Perform the initialization steps described in section 14.5 (but not its subsections). Use section 13 as a reference for the registers the initialization process refers to and sections 3.3.3 and 3.4 for reference to the transmit descriptors and transmit descriptor array.

Be mindful of the alignment requirements on the transmit descriptor array and the restrictions on length of this array. Since TDLEN must be 128–byte aligned and each transmit descriptor is 16 bytes, your transmit descriptor array will need some multiple of 8 transmit descriptors. However, don't use more than 64 descriptors or our tests won't be able to test transmit ring overflow.

For the TCTL.COLD, you can assume full–duplex operation. For TIPG, refer to the default values described in table 13–77 of section 13.4.34 for the IEEE 802.3 standard IPG (don't use the values in the table in section 14.5).

Figure 7: Exercise 5

Solution

- Define certain macros about size in **kern/e1000.h**

```
// LAB 6. Exercise 5. Define certain length
#define TXRING_LEN      64
#define RXRING_LEN      128
#define TBUFFSIZE       2048
#define RBUFFSIZE       2048
```

- Initialize the packet and descriptor array in kern/e1000.c

```
// declare arraies of desc and packet: exercise 5
struct tx_desc tx_d[TXRING_LEN] __attribute__((aligned(PGSIZE)))
                = {{0, 0, 0, 0, 0, 0, 0}};
struct rx_desc rx_d[RXRING_LEN] __attribute__((aligned(PGSIZE)))
                = {{0, 0, 0, 0, 0, 0, 0}};

struct packet ptxbuf[TXRING_LEN] __attribute__((aligned(PGSIZE))) = {{{0}}};

struct packet prxbuf[RXRING_LEN] __attribute__((aligned(PGSIZE))) = {{{0}}};
```

2.3.2.1 Descriptor Initialization

```
// Descriptor initialization

static void
init_desc() {
        int i;
        for (i = 0; i < TXRING_LEN; i++) {
                memeset(&tx_d[i], 0, sizeof(tx_d[i]));
                tx_d[i].addr = PADDR(&ptxbuf[i]);
                tx_d[i].status = TXD_STAT_DD;
                tx_d[i].cmd = TXD_CMD_RS | TXD_CMD_EOP;
        }
        for (i = 0; i < RXRING_LEN; i++) {
                memset(&rx_d[i], 0, sizeof(rx_d[i]));
                rx_d[i].addr = PADDR(&prxbuf[i]);
                rx_d[i].status = 0;
        }
}
```

2.3.3 Exercise 5: Solution Again

2.3.3.1 e1000_attach_func() from kern/e1000.c

```
int
e1000_attach_func(struct pci_func *pcif)
{
        pci_func_enable(pcif);
        cprintf("reg_base: %x, reg_size: %x\n",
                pcif->reg_base[0], pcif->reg_size[0]);
        bar_va = mmio_map_region(pcif->reg_base[0], pcif->reg_size[0]);
        uint32_t *status_reg = (uint32_t *)E1000REG(E1000_STATUS);
        assert(*status_reg == 0x80080783);

        e1000_transmit_init();
        return 0;
}
```

2.3.4 Exercise 6

Exercise 6. Write a function to transmit a packet by checking that the next descriptor is free, copying the packet data into the next descriptor, and updating TDT. Make sure you handle the transmit queue being full.

Figure 8: Exercise 6

After the exercise 5, you'll have to transmit a packet and make it accessible to user space.

To transmit a packet, we should add it (the packet) to tail of the transmit queue, which means:

- Copying the packet data into the next packet buffer.
- Updating the TDT (transmit descriptor tail) to inform the card that there's another packet in the transmit queue. TDT is an index into the transmit descriptor array, not a byte offset

3 Part B: Receiving Packets and the Web Server

4 Reference

1. bysui's github and blog
   - github: https://github.com/bysui/mit6.828
   - blog: https://blog.csdn.net/bysui
2. SmallPond's github and blog
   - github: https://github.com/SmallPond/MIT6.828_OS
   - blog: https://me.csdn.net/Small_Pond
3. SimpCosm's github
   - github: https://github.com/SimpCosm/6.828
4. fatsheep9146's blog
   - blog: https://www.cnblogs.com/fatsheep9146/category/769143.html