操作系统JOS实习第六次报告

张弛 00848231,
zhangchitc@gmail.com

May 28, 2011

Contents

1 Introduction

此次Lab是所有JOS实验中最恶心的一次，需要阅读的东西超过了以前所有的总和。所以请做好准备。

在真正下手之前，最好的话请完整的将MIT的材料完整的读一遍，对各个名词和部分有个大致的印象。其他要读的材料还有很多，具体的部分我在报告中会着重提到。

纵观全局，这次Lab的最大难点就是在于你需要**从零开始**写出一个E100网卡的驱动程序。这个驱动程序从Web Server接收IPC调用向网卡发送数据，然后从网卡接收数据发回给Web Server。这和我们以前的实验都不一样，以前都是给出了结构的框架，我们只需要针对一个具体的功能函数进行细节的填补即可，相关的数据结构、接口设置都为我们设计好了。这次需要我们从头到尾完成整个网卡驱动，困难可想而知。

除开网卡之外的部分都相对简单。因此我们这篇报告重点介绍如何完成这个网卡驱动。操纵网卡我们需要了解的方面有：

1. 一个PCI设备在JOS中的设置和相关数据结构
2. 扫描和初始化网卡
3. 网卡的关键结构
4. 网卡如何和操作系统交互数据
5. 如何对网卡发送指令

报告在后面会一步一步从课程给出的资料中抽取出这些细节。同时我在写的过程中参考了http://code.google.com/p/os-xv6-network项目主页提供的一份源代码，说的比较极端的话我基本是完全照搬了它的代码，不过这并不影响，学习别人优秀的代码本来就是编程中获得提高最有效的方式。重要的仍是个人的理解。

2 Initialization and transmitting packets

对于Network Server的架构，我们只需要大致了解模块即可，这次的实验很少需要对Server进行大规模的修改。

2.0.1 The Timer Environment

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.

这个Exercise太简单了就不贴代码了，唯一需要注意的是在测试用户程序testtime之前由于我们还没有实现网络服务器的部分，所以需要注释掉JOS载入网络服务器的部分：

kern/init.c: i386_init()

```
        // Should always have an idle process as first one.
        ENV_CREATE(user_idle);

        // Start fs.
        ENV_CREATE(fs_fs);

#if !defined(TEST_NO_NS)
        // Start ns.
        //ENV_CREATE(net_ns);
#endif
```

这样运行客户程序才不会出错。

2.0.2 The Output Environment

2.0.3 The Input Environment

这两部分在材料中也提到了我们需要先实现驱动程序和系统调用部分才可以完成。所以我们先放下他们关注最重要的驱动部分。

2.1 The Network Interface Card

Exercise 2. Browse the Intel 82559 page and look at these two documents:

1. Intel 8255x 10/100 Mbps Ethernet Controller Family Open Source Software Developer Manual (local copy)
2. 82559ER Fast Ethernet PCI Controller Datasheet (local copy)

Do not worry about the details in your first pass. It is more important to read this assignment write-up first to get a high level pictures of how the Intel chip is organized and what is needed to create a device driver.

When you do read the open source developer manual in depth, glance over Section 4 to learn about the PCI interface but pay very close attention to Section 6 as it deals with the Software Interface. In fact, most everything you need is in Section 6. Use the datasheet solely as a reference if you find the developer manual vague.

A simple E100 driver needs only a fraction of the features and interfaces that the card provides. When you're reading through the developer manual, think carefully about the easiest way to interface with the card. You're of course welcome to use its more advanced, high-performance features (in fact, some of the challenge exercises ask you to do exactly this), but it's a good idea to get a basic driver working first.

The acronyms in both documents can get overwhelming. Consult the glossary at the end of this lab assignment for some help.

打开资料是不是已经晕了？这是我觉得这个Lab设计的非常不好的原因之一，一开始就甩出一大堆手册要我们看，而不是给出一个纵览性的结构和指南。先放下吧，后面我们再慢慢说具体要看这些资料的哪些部分。

2.1.1 PCI Interface

在使用网卡之前，我们先要通知硬件系统扫描出所有的PCI设备，并且找出其中的E100网卡，对其进行初始化。这是这部分任务的最终目的，为我们操作网卡做好准备工作。这里我们将大致介绍一下PCI设备的一些基础知识：

什么是PCI设备 ：

PCI是外围设备互连（Peripheral Component Interconnect）的简称，是在目前的计算机系统中得到了非常广泛应用的通用总线接口标准。

- 在一个PCI系统中，最多可以有256根PCI总线，一般的主机上只会用到其中很少的几条，比如Bus 0和Bus 1。
- 在一根PCI总线上可以连接多个物理设备，可以是一个网卡、显卡声卡等等。最多不超过32个。一般来说因为物理特性的限制，一条总线上不会有太多的设备。
- 一个PCI物理设备可以有多个功能，比如同时提供视频解析和声音解析。最多提供8个功能。
- 每个功能对应一个256 bytes的PCI Configuration Space，这个在我们接下来对网卡进行初始化的时候会特别提到这个东西的。

所以对于一个PCI设备的具体功能，我们可以使用**总线号：设备号：功能号**来对其进行定位，比如在Ubuntu下我们使用lspci命令，就可以得到这样的输出：

```
zhangchi@zhangchi-vostro1400:~$ lspci
00:00.0 Host bridge: Intel Corporation Mobile PM965/GM965/GL960 Memory Controller Hub (rev 0c)
00:02.0 VGA compatible controller: Intel Corporation Mobile GM965/GL960 Integrated Graphics Controller (rev 0c)
00:02.1 Display controller: Intel Corporation Mobile GM965/GL960 Integrated Graphics Controller (rev 0c)
00:1a.0 USB Controller: Intel Corporation 82801H (ICH8 Family) USB UHCI Controller #4 (rev 02)
00:1a.1 USB Controller: Intel Corporation 82801H (ICH8 Family) USB UHCI Controller #5 (rev 02)
00:1a.7 USB Controller: Intel Corporation 82801H (ICH8 Family) USB2 EHCI Controller #2 (rev 02)
00:1b.0 Audio device: Intel Corporation 82801H (ICH8 Family) HD Audio Controller (rev 02)
00:1c.0 PCI bridge: Intel Corporation 82801H (ICH8 Family) PCI Express Port 1 (rev 02)
00:1c.1 PCI bridge: Intel Corporation 82801H (ICH8 Family) PCI Express Port 2 (rev 02)
00:1c.3 PCI bridge: Intel Corporation 82801H (ICH8 Family) PCI Express Port 4 (rev 02)
00:1c.5 PCI bridge: Intel Corporation 82801H (ICH8 Family) PCI Express Port 6 (rev 02)
00:1d.0 USB Controller: Intel Corporation 82801H (ICH8 Family) USB UHCI Controller #1 (rev 02)
00:1d.1 USB Controller: Intel Corporation 82801H (ICH8 Family) USB UHCI Controller #2 (rev 02)
00:1d.2 USB Controller: Intel Corporation 82801H (ICH8 Family) USB UHCI Controller #3 (rev 02)
00:1d.7 USB Controller: Intel Corporation 82801H (ICH8 Family) USB2 EHCI Controller #1 (rev 02)
00:1e.0 PCI bridge: Intel Corporation 82801 Mobile PCI Bridge (rev f2)
00:1f.0 ISA bridge: Intel Corporation 82801HEM (ICH8M) LPC Interface Controller (rev 02)
00:1f.1 IDE interface: Intel Corporation 82801HBM/HEM (ICH8M/ICH8M-E) IDE Controller (rev 02)
00:1f.2 IDE interface: Intel Corporation 82801HBM/HEM (ICH8M/ICH8M-E) SATA IDE Controller (rev 02)
00:1f.3 SMBus: Intel Corporation 82801H (ICH8 Family) SMBus Controller (rev 02)
03:01.0 FireWire (IEEE 1394): Ricoh Co Ltd R5C832 IEEE 1394 Controller (rev 05)
03:01.1 SD Host controller: Ricoh Co Ltd R5C822 SD/SDIO/MMC/MS/MSPro Host Adapter (rev 22)
03:01.2 System peripheral: Ricoh Co Ltd R5C843 MMC Host Controller (rev 12)
03:01.3 System peripheral: Ricoh Co Ltd R5C592 Memory Stick Bus Host Adapter (rev 12)
03:01.4 System peripheral: Ricoh Co Ltd xD-Picture Card Controller (rev ff)
09:00.0 Ethernet controller: Broadcom Corporation NetLink BCM5906M Fast Ethernet PCI Express (rev 02)
0c:00.0 Network controller: Intel Corporation PRO/Wireless 3945ABG [Golan] Network Connection (rev 02)
```

看到前面用冒号和点分割的数字了么，就是以总线号、设备号以及功能号进行标识的。可以看到我的机器上用到了0、3、9、12这四条PCI总线。

PCI配置寄存器 ：

每一个PCI设备都有它映射的内存地址空间和它的I/O区域。除此之外，PCI设备还有它的配置寄存器（即Configuration Space）。对于所有的PCI设备，配置地址空间一共256 bytes，其中前64 bytes是标准化的，它提供了厂商号，设备号，版本号等信息，唯一标识一个PCI设备。同时，它也提供了最多可多达6个的I/O地址区域，每个区域可以是内存也可以是I/O地址。这几个I/O地址区域是驱动程序找到设备映射到内存和I/O空间的具体位置的唯一途径。关于这64个字节的配置空间的详细情况，可以参考下图：

register	bits 31-24	bits 23-16	bits 15-8	bits 7-0
00	Device ID		Vendor ID	
04	Status		Command	
08	Class code	Subclass	Prog IF	Revision ID
0C	BIST	Header type	Latency Timer	Cache Line Size
10	Base address #0 (BAR0)			
14	Base address #1 (BAR1)			
18	Base address #2 (BAR2)			
1C	Base address #3 (BAR3)			
20	Base address #4 (BAR4)			
24	Base address #5 (BAR5)			
28	Cardbus CIS Pointer			
2C	Subsystem ID		Subsystem Vendor ID	
30	Expansion ROM base address			
34	Reserved			Capabilities Pointer
38	Reserved			
3C	Max latency	Min Grant	Interrupt PIN	Interrupt Line

PCI设备启动过程 ：

在系统引导阶段，PCI硬件设备保持未激活状态，每个设备都没有被分配内存空间和I/O端口。但每个PCI主板均配备有能够处理PCI的固件（比如BIOS），固件通过读写PCI控制器中的寄存器，提供了对设备配置地址空间的访问。系统启动以后，固件通过扫描每个PCI设备，通过读取他们的配置地址空间，为每个设备分配相应的内存和I/O端口，为后面硬件驱动程序做好准备。

如何使用PCI设备 ：

> 当我们想查询一个特定PCI设备的配置地址空间时，我们需要向I/O地址$[0cf8, 0cfb]$写入一个4 bytes查询码指定总线号：设备号：功能号以及其配置地址空间中的查询位置。那么PCI Host Bridge将监听对于这个I/O端口的写入并在接受到写入数据后将相应的查询结果写入到$[0cfc, 0cff]$，我们从这个地址读出一个32位整数表示查询到的相应信息。
>
> 查询配置地址空间时，我们一般会从其6个BARS中得到特定设备的控制端口和数据端口信息，那么只要在初始化时将这些端口地址保存下来，就可以在PCI硬件驱动程序中通过向这些端口输入输出数据来达到控制PCI设备的目的了

更多更详细的内容可以参考网站：http://xwindow.angelfire.com/page13_1.html，上面介绍的内容已经足够我们理解JOS中的相应代码了。

接下来我们看JOS中是如何对PCI设备进行编程的，这部分模块主要定义在kern/pci.c中，JOS在系统初始化时调用其中的`pci_init()`进行设备初始化，首先来看一些最基本的东西：

kern/pci.c

```
36  static void
37  pci_conf1_set_addr(uint32_t bus,
38                     uint32_t dev,
39                     uint32_t func,
40                     uint32_t offset)
41  {
42          assert(bus < 256);
43          assert(dev < 32);
44          assert(func < 8);
45          assert(offset < 256);
46          assert((offset & 0x3) == 0);
47
48          uint32_t v = (1 << 31) |                // config-space
49                  (bus << 16) | (dev << 11) | (func << 8) | (offset);
50          outl(pci_conf1_addr_ioport, v);
51  }
52
53  static uint32_t
54  pci_conf_read(struct pci_func *f, uint32_t off)
55  {
56          pci_conf1_set_addr(f->bus->busno, f->dev, f->func, off);
57          return inl(pci_conf1_data_ioport);
58  }
59
60  static void
61  pci_conf_write(struct pci_func *f, uintint
62  e100_sw_reset(struct dev_e100 *e100) {
63          outl(e100->reg_base[E100_IO] + CSR_PORT, PORT_SW_RESET);
64
65          // delay about 10us
66          int i = 0;
67          for (i = 0; i < 8; i++) {
68                  inb (0x84);
```

```
69          }
70
71          e100_init(e100);
72
73          return 1;
74  }
75  32_t off, uint32_t v)
76  {
77          pci_conf1_set_addr(f->bus->busno, f->dev, f->func, off);
78          outl(pci_conf1_data_ioport, v);
79  }
```

这三个函数是对PCI设备最基本的读状态和写状态的函数，其中 `pci_conf1_set_addr()` 负责设置需要读写的具体设备。这里涉及到的两个I/O端口定义在了文件最上方：

kern/pci.c

```
12  // PCI "configuration mechanism one"
13  static uint32_t pci_conf1_addr_ioport = 0x0cf8;
14  static uint32_t pci_conf1_data_ioport = 0x0cfc;
```

正是我们前面提到的两个端口。接下来我们看看它是怎么初始化PCI设备的，看到`pic_init()`：

kern/pci.c

```
36  static int
37  pci_scan_bus(struct pci_bus *bus)
38  {
39      int totaldev = 0;
40      struct pci_func df;
41      memset(&df, 0, sizeof(df));
42      df.bus = bus;
43
44      for (df.dev = 0; df.dev < 32; df.dev++) {
45          uint32_t bhlc = pci_conf_read(&df, PCI_BHLC_REG);
46          if (PCI_HDRTYPE_TYPE(bhlc) > 1)     // Unsupported or no device
47              continue;
48
49          totaldev++;
50
51          struct pci_func f = df;
52          for (f.func = 0; f.func < (PCI_HDRTYPE_MULTIFN(bhlc) ? 8 : 1); f.func++) {
53              struct pci_func af = f;
54
55              af.dev_id = pci_conf_read(&f, PCI_ID_REG);
56              if (PCI_VENDOR(af.dev_id) == 0xffff)
57                  continue;
58
59              uint32_t intr = pci_conf_read(&af, PCI_INTERRUPT_REG);
60              af.irq_line = PCI_INTERRUPT_LINE(intr);
61
62              af.dev_class = pci_conf_read(&af, PCI_CLASS_REG);
63              if (pci_show_devs)
64                  pci_print_func(&af);
65              pci_attach(&af);
66          }
67      }
68
69      return totaldev;
70  }
71
```

```
72 int
73 pci_init(void)
74 {
75         static struct pci_bus root_bus;
76         memset(&root_bus, 0, sizeof(root_bus));
77
78         return pci_scan_bus(&root_bus);
79 }
```

1. pci_init()中，root_bus被全部清0，然后交给pci_scan_bus()扫描这条总线上的所有设备，说明在JOS中E100是被放置在0号总线上的

2. pci_scan_bus() 中顺次查找0号总线上的32个设备，如果发现其存在，那么顺次扫描它们每个功能对应的配置地址空间，将一些关键的控制参数读入pci_func进行保存，其中pci_func的结构如下：

kern/pci.h

```
11 struct pci_func {
12     struct pci_bus *bus;        // Primary bus for bridges
13
14     uint32_t dev;
15     uint32_t func;
16
17     uint32_t dev_id;
18     uint32_t dev_class;
19
20     uint32_t reg_base[6];
21     uint32_t reg_size[6];
22     uint8_t irq_line;
23 };
```

对于网卡驱动来说，最重要的就是其reg_base数组，这是我们用于向E100发送命令的地址端口，在后面的初始化程序中我们需要将其记录下来。

3. 得到pci_func之后，它被传入pci_attach()去查找是否为已存在的硬件，如果匹配成功，则使用预设好的程序初始化该硬件

kern/pci.c

```
67 static int __attribute__((warn_unused_result))
68 pci_attach_match(uint32_t key1, uint32_t key2,
69                  struct pci_driver *list, struct pci_func *pcif)
70 {
71     uint32_t i;
72
73     for (i = 0; list[i].attachfn; i++) {
74         if (list[i].key1 == key1 && list[i].key2 == key2) {
75             int r = list[i].attachfn(pcif);
76             if (r > 0)
77                 return r;
78             if (r < 0)
79                 cprintf("pci_attach_match: attaching "
80                     "%x.%x (%p): e\n",
81                     key1, key2, list[i].attachfn, r);
82         }
83     }
84     return 0;
85 }
```

```
86
87  static int
88  pci_attach(struct pci_func *f)
89  {
90      return
91          pci_attach_match(PCI_CLASS(f->dev_class),
92                  PCI_SUBCLASS(f->dev_class),
93                  &pci_attach_class[0], f) ||
94          pci_attach_match(PCI_VENDOR(f->dev_id),
95                  PCI_PRODUCT(f->dev_id),
96                  &pci_attach_vendor[0], f);
97  }
```

到这里PCI设备的初始化就结束了，接下来我们要尝试写一个E100网卡的初始化过程。

> **Exercise 3.** Implement an attach function to initialize the 82559ER. Add an entry to the pci_attach_vendor array in kern/pci.c to trigger your function if a matching PCI device is found. The vendor ID and device ID for the 82559ER can be found in Section 4 of the developer manual. You should also see these listed when JOS scans the PCI bus while booting.
>
> After enabling the E100 device via pci_func_enable, your attach function should record the IRQ line and base I/O port assigned to the device so you'll be able to communicate with the E100.
>
> We have provided the kern/e100.c and kern/e100.h files for you so that you do not need to mess with the make system. You may still need to include the e100.h file in other places in the kernel.
>
> When you boot your kernel, you should see it print that the PCI function of the E100 card was enabled. Your code should now pass the pci attach test of make grade.

第一步我们查阅手册得到E100的Vender ID为8086h，Device ID为1229h，然后将初始化程序作为驱动程序的一部分定义在kern/e100.c中，先撰写头文件kern/e100.h：

kern/e100.h

```
1  #ifndef JOS_KERN_E100_H
2  #define JOS_KERN_E100_H
3
4  #include <kern/pci.h>
5
6  #define E100_VENDOR             0x8086
7  #define E100_DEVICE             0x1209
8
9  int e100_attach(struct pci_func *pcif);
10
11 #endif  // JOS_KERN_E100_H
```

然后是主过程，定义了一个e100记录其相应的设备信息：

kern/e100.c

```
1  // LAB 6: Your driver code here
2
```

```
3  #include <inc/x86.h>
4  #include <inc/stdio.h>
5
6  #include <kern/e100.h>
7
8  struct pci_func e100;
9
10 int
11 e100_attach(struct pci_func *pcif)
12 {
13     pci_func_enable(pcif);
14     e100.bus = pcif->bus;
15     e100.dev_id = pcif->dev_id;
16     e100.dev_class = pcif->dev_class;
17     int i;
18     for (i = 0; i < 6; i++) {
19         e100.reg_base[i] = pcif->reg_base[i];
20         e100.reg_size[i] = pcif->reg_size[i];
21         cprintf ("zhangchi: The %dth Bar: base = %x, size = %x\n",
22             i, e100.reg_base[i], e100.reg_size[i]);
23     }
24     e100.irq_line = pcif->irq_line;
25
26     return 0;
27 }
```

接下来修改kern/pci.c中的pci_attach_vendor数组，把我们的E100初始化程序添加进去：

kern/pci.c

```
31 // pci_attach_vendor matches the vendor ID and device ID of a PCI device
32 struct pci_driver pci_attach_vendor[] = {
33         { E100_VENDOR, E100_DEVICE, &e100_attach },
34         { 0, 0, 0 },
35 };
```

然后make qemu启动JOS，应该能看到E100网卡被顺利激活了：

```
enabled interrupts: 1 2
            Setup timer interrupts via 8259A
enabled interrupts: 0 1 2
            unmasked timer interrupt
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: 8086:1209: class: 2.0 (Network controller) irq: 11
PCI function 00:03.0 (8086:1209) enabled
zhangchi: The 0th Bar: base = f2020000, size = 1000
zhangchi: The 1th Bar: base = c040, size = 40
zhangchi: The 2th Bar: base = f2040000, size = 20000
zhangchi: The 3th Bar: base = 0, size = 0
zhangchi: The 4th Bar: base = 0, size = 0
zhangchi: The 5th Bar: base = 0, size = 0
FS is running
FS can do I/O
```

可以看到第二个端口应该就是我们以后进行操作的I/O端口c040，空间大小为40h = 64 bytes

2.1.2 E100 Reset

> **Exercise 4.** Add code to your attach function to reset the 82559ER. If you set the -debug-e100 flag, QEMU should tell you if the reset was successfully. It will print something like this after JOS starts scanning the PCI bus:
>
> ```
> EE100   nic_reset              0xacea498
> ```
>
> There will also be a few nic_reset's before JOS starts; those are the BIOS itself resetting the device.

这一段的MIT提供的资料是相对来说比较详尽的，还记得我们前面打印出来的c040地址么？这个地方就是我们要写入CSR的端口。CSR(Control/Status Registers)是我们对于E100网卡的控制字，如前面所说，它是一个64 bytes的地址空间，其中我们最需要关注的是它的前12个bytes，称为SCB(System Control Block)，我们对网卡的主要控制主要是对于SCB相应参数的进行设置。其布局如下：

Upper Word (31–16)	Lower Word (15–0)	Offset
SCB Command Word	SCB Status Word	0h
SCB General Pointer		4h
PORT		8h
EEPROM Control Register	Reserved	Ch
MDI Control Register		10h
RX DMA Byte Count		14h
PMDR / Flow Control Register / Reserved		18h
Reserved / General Status / General Control		1Ch
Reserved		20h-2Ch
Function Event Register		30h
Function Event Mask Register		34h
Function Present State Register		38h
Force Event Register		3Ch

对于重设网卡，是采用的PORT Interface的形式进行控制，详见Manual的6.3.3 PORT Interface。这里E100允许我们只对SCB中的PORT设置特定值以后就执行相应的功能，比如说有：

Function	Pointer Field (Bits 31:4)	Opcode (Bits 3:0)
Software Reset	Don't care	0000
Self-test	Self-test results pointer (16 byte alignment)	0001
Selective Reset	Don't care	0010
Dump	Dump area pointer (16 byte alignment)	0011
Dump Wake-up	Dump area pointer (16 byte alignment)	0111

从这个表格看出，我们只需要对PORT字段写入全0就可以达到重启的目的了。

但是在进行编码之前，我们需要对头文件进行一些修改，方便后续的程序编写。

kern/e1000.h

```
#ifndef JOS_KERN_E100_H
#define JOS_KERN_E100_H

#include <kern/pci.h>

#define E100_VENDOR             0x8086
#define E100_DEVICE             0x1209

#define E100_MEMORY     0
#define E100_IO         1
#define E100_FLASH      2

#define CSR_SCB         0x0
#define CSR_STATUS      0x0
#define CSR_US          0x0
#define CSR_STATACK     0x1
#define CSR_COMMAND     0x2
#define CSR_UC          0x2
#define CSR_INT         0x3
#define CSR_GP          0x4
#define CSR_PORT        0x8


#define PORT_SW_RESET   0x0
#define PORT_SELF_TEST  0x1
#define PORT_SEL_RESET  0x2

int e100_attach(struct pci_func *pcif);

#endif  // JOS_KERN_E100_H
```

1. 在手册中的4.1 PCI Configuration Space，对于E100而言，PCI配置中提供的6个地址中的前三个分别为

   (a) CSR Memory Mapped Base Address Register

   (b) CSR I/O Mapped Base Address Register

   (c) Flash Memory Mapped Base Address Register

   因为我们只使用I/O端口对CSR进行控制，不使用内存地址的原因资料中也提到了，有可能因为编译器的原因使得地址端口失效，所以最稳固的方法还是使用I/O的方式。这三个地址在初始化时已经被载入到e100.reg_base[0-2]中了。在使用他们的基址的时候，我们为他们定义了相应的数组索引位置

2. 定义了一系列SCB字段在CSR中的位移，以便于后面我们使用in和out指令对他们进行读写操作

3. 预定义了三条PORT Interface指令

作这项工作中我大量参考了http://code.google.com/p/os-xv6-network/source/browse/trunk/dev/e100.h提供的参数，节省了我大量的时间，对作者表示感谢。

然后就可以真正开始对网卡进行重启了：

kern/e100.c

```
// LAB 6: Your driver code here

#include <inc/x86.h>
#include <inc/stdio.h>

#include <kern/e100.h>

struct pci_func e100;

static void e100_sw_reset(struct pci_func e100);

int
e100_attach(struct pci_func *pcif)
{
    pci_func_enable(pcif);
    e100.bus = pcif->bus;
    e100.dev_id = pcif->dev_id;
    e100.dev_class = pcif->dev_class;
    int i;
    for (i = 0; i < 6; i++) {
        e100.reg_base[i] = pcif->reg_base[i];
        e100.reg_size[i] = pcif->reg_size[i];
    }
    e100.irq_line = pcif->irq_line;

    e100_sw_reset(e100);
}

static void
e100_sw_reset(struct pci_func e100) {
    outl(e100.reg_base[E100_IO] + CSR_PORT, PORT_SW_RESET);

    // delay about 10us
    int i = 0;
    for (i = 0; i < 8; i++) {
        inb (0x84);
    }
}
```

注意不要忘了按照MIT材料的提示重启后delay 一段时间再返回。使用make qemu QEMUEXTRA="-debug-e100"启动JOS应该可以看到网卡的重启消息:

```
EE100   nic_init
EE100   pci_reset               0x9566008
EE100   nic_init                macaddr:  52 54 00 12 34 56
EE100   nic_reset               0x9566008
EE100   nic_selective_reset     checksum=0xbe34
EE100   nic_init                model=i82559er,macaddr=52:54:00:12:34:56
EE100   nic_reset               0x9566008
EE100   nic_selective_reset     checksum=0xbe34
EE100   pci_mmio_map            region 0, addr=0xf2020000, size=0x00001000, type=8
EE100   pci_map                 region 1, addr=0x0000c040, size=0x00000040, type=1
EE100   pci_mmio_map            region 2, addr=0xf2040000, size=0x00020000, type=0
6828 decimal is 15254 octal!
Hooray! Passed all test cases for stdlib!!
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
```

```
page_check() succeeded!
check_boot_pgdir() succeeded!
enabled interrupts: 1 2
            Setup timer interrupts via 8259A
enabled interrupts: 0 1 2
            unmasked timer interrupt
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: 8086:1209: class: 2.0 (Network controller) irq: 11
EE100   pci_mmio_map            region 0, addr=0xf2020000, size=0x00001000, type=8
EE100   pci_map                 region 1, addr=0x0000c040, size=0x00000040, type=1
EE100   pci_mmio_map            region 2, addr=0xf2040000, size=0x00020000, type=0
PCI function 00:03.0 (8086:1209) enabled
EE100   eepro100_write4         addr=Port+0 val=0x00000000
EE100   nic_reset               0x9566008
EE100   nic_selective_reset     checksum=0xbe34
FS is running
```

2.1.3 E100 Structure

其实这里Intel的手册挺让人费解的，我读了以后发现CU其实就是负责发送数据的模块，RU就是接收数据的模块，按道理两个是相对的，那么CU应该称为Transmit Unit才对，Intel却命名为Control Unit，让人感觉这两个模块是分立的，并且CU有控制RU一样。

对于CU和RU的讨论我们要等到将DMA Ring看完以后才能完整的描述，我们先来看DMA Rings

2.1.4 DMA Rings

根据资料的描述，DMA Rings就是系统为E100在内存中开辟的一片区域，用于网卡使用DMA缓存当前的收发数据的。一个DMA Rings在申请好以后，就可以将其所在的物理地址通知给E100的DMA控制器，那么E100就可以在不占用CPU的情况下自己根据内存中的数据开始进行收发操作了。**实际情况中内存里应该有两条DMA Rings**，一个专门用于放置待发送的数据（Control Block list, CBL），一个专门放置接受到的数据（Receive Frame Area, RFA）。

因为采取了这样的操作模式，所以提示我们E100是根据DMA Rings中的内容进行工作的。接下来我们就会详细讲述利用CU发送数据时数据包格式是如何用Control Block(CB)进行描述的。E100读取CB中的设置，然后进行相应的发送操作，当工作完成后，**通过引发中断或者改变DMA Rings相应的状态位来提示系统工作已完成**。我们可以修改CB中相应场位的设置，来改变E100的工作模式。

上面是E100自动进行工作的一种方式，如果我们需要人为的干预E100的运行的话，**还可以向CSR中的SCB寄存器写入相应的控制指令**，比如终止运行等，来改变其运行状态。

所以，在完成这个部分的工作前，我们需要了解两方面重要的内容：

1. CB (Control Block)的控制设置
2. SCB (System Control Block)的控制设置

在了解DMA Rings 的相关结构以后，我们会结合DMA Rings来进一步阐述这些控制设置在DMA Rings上运行的效果。

2.2 Device Driver Organization

2.3 Transmitting Packets

前面提到DMA Rings主要分为两种用途：发送和接受。

- 发送数据的DMA Rings是由若干个CB (Control Block)组成的，这些CB通过指针连接成一个环状的结构CBL（Control Block List）。
- 接受数据的DMA Rings由若干个RFD (Receive Frame Description)组成，也是连成环状，称为RFA (Receive Frame Area)

这个阶段我们主要关注发送，发送是由CU模块来完成的。

CB是一个通用的概念，即使是只在发送时用到，但是因为CU执行的不止一种命令，所以根据不同命令的需要对CB进行更加细致的规定。只有CB的前三个成员Control, Status和Link在各个应用场景中都是相同的，接下来的Command Specific Data才是根据不同的命令发生变化的。一个CB的结构在Intel的开源手册中的6.4.1.1 General Action Command Format中可以找到具体格式，如下图所示：

Figure 14. General Action Command Format

Offset	Command Word Bits 31:16					Status Word Bits 15:0			
00h	EL	S	I	0000000000	CMD	C	X	OK	XXXXXXXXXXXX
04h	Link Offset								
08h	Optional Address and Data Fields								

其中有三个位置是特别值得我们关注的：

- S：如果该位被设置为1，那么当E100的**CU**执行完此CB的命令之后，将停止运行进入挂起状态(Suspend)，只有使用SCB对E100下达恢复运行(Resume)指令后，CU模块才会重新运行，重新运行开始时执行的地址是该CB的Link Address指向的下一CB。注意，这里要注意**发送和接受是互相独立不干扰的**，所以CU停止工作的时候有可能RU还在正常运行，注意明确他们两个结构的概念。
- Status Word Bits中的C位：在操作CU执行任意CB上的命令时，**程序员首先应该负责手动清除该位上的值**，将其置为0。那么当CU模块执行完任务后，这个位置被设置成1。就可以检测某些指令的执行情况了。比如发送数据时由于数据包比较大，从CU获取到CB开始执行到发送完毕需要耗费一定的时间，

- CMD：CB支持的不同的操作类型，具体操作类型可以在手册的6.4.2 Specific Action Commands中找到，大致有以下几种操作:
  - NOP (000b)
  - Individual Address Setup (001b)
  - Configure (010b)
  - Multicast Setup (011b)
  - Transmit (100b)
  - Load Microcode (101b)
  - Dump (110b)
  - Diagnose (111b)

  在这次实验中我们在CU模块的操作只会遇到两个：NOP和Transmit，严格的说只要使用Transmit就好了，但是在写的时候我们可以用NOP来进行一些测试，来检测程序的正确性。NOP的作用就是使CU在处理到该CB时什么事情也不作，**但是相应的状态位S、C等等都会对CU的执行有作用**。所以我们可以利用NOP指令设置S位**使其停止在某个CB上**，然后利用debug的输出信息进行调试，而不用去理会Transimit指令中其他那些乱七八糟的参数的设置。

看完CB的相关字段说明，我们可以将这些状态定义到头文件里方便后续的设置和读取判断了：

kern/e100.h

```
1  // Control Block Command
2  #define CBF_EL          0x8000
3  #define CBF_S           0x4000
4  #define CBF_I           0x2000
5
6  #define CBC_NOP         0x0
7  #define CBC_IAS         0x1
8  #define CBC_CONFIG      0x2
9  #define CBC_MAS         0x3
10 #define CBC_TRANSMIT    0x4
11 #define CBC_LOADMC      0x5
12 #define CBC_DUMP        0x6
13 #define CBC_DIAGNOSE    0x7
14
15 // Control Block Status
16 #define CBS_F           0x0800
17 #define CBS_OK          0x2000
18 #define CBS_C           0x8000
```

上面描述的是CB的一些通用概念，具体到发送包指令Transmit的时候，我们将这样一连串的CB称为TCBs (Transmit Command Blocks)。在MIT资料中已

Figure 19. Transmit Command Format

Offset	Command Word Bits 31:16								Status Word Bits 15:0				
00h	EL	S	I	CID	000	NC	SF	100	C	X	OK	U	XXXXXXXXXXXX
04h	Link Address (A31:A0)												
08h	Transmit Buffer Descriptor Array Address												
	TBD Number				Transmit Threshold				EOF	0	Transmit Command Block Byte Count		

经有一个形象的图为我们展示出了TCB的串联结构，更详细的信息可以在手册中的6.4.2.5 Transmit找到。这个是TCB的一个布局图。

可以看到，虽然TCB的结构比CB更细化了，但是我们现在还不需要关注它的设置的细节，第一步我们先考虑**在内存中分配空间和建立起相应的TCB结构**。这就需要结合CU的工作方式来阐述它是如何读取和使用TCB的了。以下列出了使用TCBs发送数据包的全过程：

1. 首先在内存中建立起TCBs的环状结构
2. 将其中第一个TCB的物理地址写入CU的General Pointer，通知其要操作的TCB所在的内存位置
3. 给CU一个Start指令，CU开始工作
4. 如果环状的TCB中所有的S位都是0，那么CU将根据Link Address无限循环的处理这些TCB，每次执行完成以后，以中断和修改TCB中C状态位来表明任务已完成(后面我们会说明处理这些消息的方式，现在先不用管)
5. 如果CU碰到了某个TCB的S被设置为1，那么处理完该TCB之后，CU进入Suspend状态，等待用户发送Resume命令恢复运行，那么CU将从当前TCB的下个TCB开始执行

这只是硬件的执行机制，我们现在要考虑在TCBs上实现一个**支持多数据包发送的排队系统**，应该注意些什么？

1. 首先要记录当前TCB中哪些块是等待被发送的，哪些是可以被重新利用的
2. 其次要做好相应S位设置，让CU在发送完需要发送的包之后就停下来

考虑到上述两点，我们设计出的系统是这样描述TCB的：

kern/e100.h

```
1  #define TCB_MAXSIZE     1518
2  #define CB_MAX_NUM      10
3
4  // Transmit Command Blocks
5  struct tcb {
6      uint32_t tcb_tbd_array_addr;
7      uint16_t tcb_byte_count;
8      uint8_t tcb_thrs;
9      uint8_t tcb_tbd_count;
10     char tcb_data[TCB_MAXSIZE];
```

```
};

// Control Blocks
struct cb {
    volatile uint16_t cb_status;
    uint16_t cb_control;
    uint32_t cb_link;

    union cb_cmd_spec_data {
        struct tcb tcb;
    } cb_cmd_spec;

    struct cb *prev, *next;
    physaddr_t phy_addr;
};

// Control Block List
struct cbl {
    int cb_avail;
    int cb_wait;

    struct cb *start;
    struct cb *front, *rear;
};
```

解释一下这些结构：

- tcb：直接按照MIT给出的结构定义的，没什么好说
- cb: 这里有两点值得注意：
  1. union结构cb_cmd_spec是根据不同指令的需要说明的，其实这里我们只用到了Transmit指令，所以union中只有一个成员看着比较别扭，但是这样写更具有**扩展性和维护性**，当需要使用到其他指令时，直接在cb_cmd_spec里添加其他的数据结构即可
  2. 在cb_cmd_spec的后面我们增加了三个成员prev, next和phy_addr，这个和手册上关于CB的定义是不符的，但是并不影响CU的执行，主要是为了我们自己在后续操作中的方便，
- cbl: 这里维护了几个值，分别说明一下：
  - cb_avail: 表示当前有多少个闲置的TCB可以用来放置数据以发送
  - cb_wait: 表示当前有多少个TCB正在处于等待发送状态
  - 很明显上面两者相加应该等于所有TCB的总数，在这里应该是我们定义的CB_MAX_NUM = 10
  - start: 所有TCB中的第一个TCB，用于开始的时候初始化CU用
  - front和rear: 表示当前正在等待发送的TCB的起始和结尾

那么这个系统是如何根据cbl中定义的结构工作的呢?

1. 初始的时候front = start, rear = start→prev，表示当前等待发送的数据包为空，并且cb_avail = CB_MAX_NUM, cb_wait = 0

2. 当需要发送一个数据包时，将其添加到rear后面，并且移动rear指针，同时增加cb_wait, 减少cb_avail

3. 当确认等待数据包发送时，检查front指向TCB的C状态是否为1，可以的话则将front向后移动，同时减少cb_wait, 增加cb_avail

那么相应的边界状态比如队列空或者队列满就可以很容易的通过cb_wait和cb_avail 检测出来了。

好了到这里我们已经清楚了一个TCB结构是如何被建立起来并且在后续过程中维护的详细过程，现在我们可以考虑建立起这样的结构了：

> **Exercise 5.** Construct a control DMA ring for the CU to use. You do not need to worry about configuring the device because the default setting are fine. You also do not need to worry about setting up the device MAC address because the emulated E100 has one already configured.

首先为了程序的易于管理，我们新定义了一个结构nic:

kern/e100.h

```
// Network Interface Card
struct nic {
    uint32_t io_base;
    uint32_t io_size;

    struct cbl cbl;
};
```

nic是用于编写E100网卡驱动中所有过程中一个记录需要使用到的资源的工具，管理我们用到的I/O端口和CBL、RFA等等。

注意在前面的初始化硬件过程中添加上nic的初始化过程：

kern/e100.c

```
struct nic nic;

int
e100_attach(struct pci_func *pcif)
{
    pci_func_enable(pcif);
    e100.bus = pcif->bus;
    e100.dev_id = pcif->dev_id;
    e100.dev_class = pcif->dev_class;
    int i;
    for (i = 0; i < 6; i++) {
        e100.reg_base[i] = pcif->reg_base[i];
        e100.reg_size[i] = pcif->reg_size[i];
    }
    e100.irq_line = pcif->irq_line;

    // Initialize NIC
    nic.io_base = pcif->reg_base[E100_IO];
```

```
19      nic.io_size = pcif->reg_size[E100_IO];
20
21      e100_init ();
22
23      return 0;
24  }
```

在初始化的第一步我们先需要为TCB分配物理空间，在内核空间地址里我们注意到KERNBASE以上应该是没有被映射的（或者说映射了但是没有实际意义，在内存管理的Lab中我们就知道，KERNBASE以上的空间全部被**静态映射**成一一对应的物理内存，注意回顾静态映射的概念，就是**被映射到的物理页没有改变其引用数**，这个特性导致了我后面的程序的一个错误，请多注意！）

我们可以考虑将TCB放置到KERNBASE 以上的区域中去，我的做法是将KERNBASE 上CBLBASE = (KERNBASE + PGSIZE)作为TCB的内存空间，连续分配CB_MAX_NUM个物理页，每页对应一个TCB。（实际上一个TCB大概只占用半页的样子，因为tcb_data最大才1518个bytes而已，但是这样比较方便操作），于是初始化的程序如下：

kern/e100.c: cbl_alloc()

```
1   static void
2   cbl_alloc () {
3       int i, r;
4       void *va;
5       struct Page *p;
6       struct cb *prevcb = NULL;
7       struct cb *currcb = NULL;
8
9       // Allocate physical page for Control block
10      for (i = 0; i < CB_MAX_NUM; i++) {
11
12          va = (void *)CBLBASE + i * PGSIZE;
13
14          if ((r = page_alloc (&p)) != 0)
15              panic ("cbl_init: run out of physical memory! %e\n", r);
16
17          if ((r = page_insert (boot_pgdir, p, va, PTE_W|PTE_P)) != 0)
18              panic ("cbl_init: cannot insert page into pgdir %e\n", r);
19
20          memset (va, 0, PGSIZE);
21
22          currcb = (struct cb *)va;
23          currcb->phy_addr = page2pa (p);
24
25          if (i == 0)
26              nic.cbl.start = currcb;
27          else {
28              prevcb->cb_link = currcb->phy_addr;
29              prevcb->next = currcb;
30              currcb->prev = prevcb;
31          }
32
33          prevcb = currcb;
34      }
35
36      prevcb->cb_link = nic.cbl.start->phy_addr;
37      nic.cbl.start->prev = prevcb;
38      prevcb->next = nic.cbl.start;
39
40      nic.cbl.cb_avail = CB_MAX_NUM;
41      nic.cbl.cb_wait = 0;
```

```
42
43      nic.cbl.front = nic.cbl.start;
44      nic.cbl.rear = nic.cbl.start->prev;
45  }
```

看起来是不是好像没有问题？在初始化中调用`cbl_alloc()`确实也不会有错，但是当我调试的时候打印的时候发现了不对的地方，注意其中第17行：

```
17          if ((r = page_insert (boot_pgdir, p, va, PTE_W|PTE_P)) != 0)
18              panic ("cbl_init: cannot insert page into pgdir %e\n", r);
```

如果我们在其前后添加上打印相应物理页的信息：

kern/e100.c: cbl_alloc()

```
1           pte_t *pte = pgdir_walk (boot_pgdir, va, 1);
2           struct Page *pp = pa2page (PTE_ADDR (*pte));
3
4           cprintf ("zhangchi: current mapping va %x -> pa %x, ppref = %d\n", va,
                PTE_ADDR (*pte), pp->pp_ref);
5
6           if ((r = page_insert (boot_pgdir, p, va, PTE_W|PTE_P)) != 0)
7               panic ("cbl_init: cannot insert page into pgdir %e\n", r);
8
9           cprintf ("zhangchi: changed mapping va %x -> pa %x, ppref = %d\n", va,
                PTE_ADDR (*pte), pp->pp_ref);
```

解释一下，这里的pp指向的是va地址以前映射到的物理页，当我们使用`page_insert()`将当前申请到的TCB所在物理页插入页表以后，pp就会被卸载。启动以后我们发现JOS打印出了下面的信息：

```
zhangchi: current mapping va f0001000 -> pa 1000, ppref = 0
zhangchi: changed mapping va f0001000 -> pa 40bb000, ppref = 65535
zhangchi: current mapping va f0002000 -> pa 2000, ppref = 0
zhangchi: changed mapping va f0002000 -> pa 40ba000, ppref = 65535
zhangchi: current mapping va f0003000 -> pa 3000, ppref = 0
zhangchi: changed mapping va f0003000 -> pa 40b9000, ppref = 65535
zhangchi: current mapping va f0004000 -> pa 4000, ppref = 0
zhangchi: changed mapping va f0004000 -> pa 40b8000, ppref = 65535
zhangchi: current mapping va f0005000 -> pa 5000, ppref = 0
zhangchi: changed mapping va f0005000 -> pa 40b7000, ppref = 65535
zhangchi: current mapping va f0006000 -> pa 6000, ppref = 0
zhangchi: changed mapping va f0006000 -> pa 40b6000, ppref = 65535
zhangchi: current mapping va f0007000 -> pa 7000, ppref = 0
zhangchi: changed mapping va f0007000 -> pa 40b5000, ppref = 65535
zhangchi: current mapping va f0008000 -> pa 8000, ppref = 0
zhangchi: changed mapping va f0008000 -> pa 40b4000, ppref = 65535
zhangchi: current mapping va f0009000 -> pa 9000, ppref = 0
zhangchi: changed mapping va f0009000 -> pa 40b3000, ppref = 65535
zhangchi: current mapping va f000a000 -> pa a000, ppref = 0
zhangchi: changed mapping va f000a000 -> pa 40b2000, ppref = 65535
```

这里有两个东西需要我们关注：

1. 在插入之前映射到的物理页引用数ppref为0，还记得前面我说过的静态映射么？因为在映射KERNBASE以上的地址空间时，我们没有增加映射到的物理页的引用数，为的就是使得这些物理页能够被重用，而不是因为KERNBASE的映射使其引用数大于0导致不能回收。

2. 被卸载以后，ppref变成了65535, 很明显这个值不正常，如此大的引用数会造成这些页面几乎永久不能回收

为什么？原因出在page_insert()过程中，我们来看看具体代码：

kern/pmap.c: page_insert()

```
1  int
2  page_insert(pde_t *pgdir, struct Page *pp, void *va, int perm)
3  {
4      pte_t *pte = pgdir_walk (pgdir, va, 1);
5
6      if (pte == NULL) {
7          return -E_NO_MEM;
8      }
9
10     if (*pte & PTE_P) {
11
12         if (PTE_ADDR(*pte) == page2pa (pp)) {
13             tlb_invalidate (pgdir, va);
14             pp -> pp_ref --;
15         } else {
16             // inside the page_remove, it will invoke tlb_invalidate and
                   page_decref
17             page_remove (pgdir, va);
18         }
19     }
20
21     *pte = page2pa (pp)|perm|PTE_P;
22     pp -> pp_ref ++;
23
24     return 0;
25 }
```

注意第17行，其中卸载原有物理页时使用了page_remove()，我们知道原有页面**不是经过正常步骤加入页表的**（即通过page_insert()插入的），而是通过静态映射，那么在这里用page_remove()卸载就肯定会出问题，我们看到page_remove()的代码：

kern/pmap.c: page_remove()

```
1  void
2  page_remove(pde_t *pgdir, void *va)
3  {
4      pte_t *pte;
5      struct Page *physpage = page_lookup (pgdir, va, &pte);
6
7      if (physpage != NULL) {
8          page_decref (physpage);
9          *pte = 0;
10         tlb_invalidate (pgdir, va);
11     }
12 }
```

可以看到这里**将页面的引用数减少了1**，于是前面的现象就可以得到解释了，原本的引用数为0，减一以后因为无符号所以变成了65535。

这样我们就知道了在改变KERNBASE以上空间的映射时，不能使用page_insert()，只能自己手动将页面插入页表了，修改后的程序如下，在插入物理页面的时候将那句page_insert()调用换成：

```
1        pte_t *pte = pgdir_walk (boot_pgdir, va, 1);
2
3        *pte = page2pa (p)|PTE_W|PTE_P;
4        p -> pp_ref ++;
```

这样修改后内存的申请就正式完成了。调试的时候我想看看系统页表到底对不对，是否真的把我申请的页面加入了页表，于是启动JOS以后用QEMU自带的命令info pg打印出页表，发现怎么搞都不对，打印结果如下：

```
K> QEMU 0.12.5 monitor - type 'help' for more information
(qemu) info pg
PDE(001) 00000000-00400000 00400000 urw
 |-- PTE(000030) 00200000-00230000 00030000 urw
PDE(001) 00400000-00800000 00400000 urw
 |-- PTE(000001) 007ff000-00800000 00001000 urw
PDE(001) 00800000-00c00000 00400000 urw
 |-- PTE(000018) 00800000-00818000 00018000 ur-
 |-- PTE(000001) 00818000-00819000 00001000 urw
 |-- PTE(00021f) 00819000-00a38000 0021f000 ur-
 |-- PTE(000107) 00a38000-00b3f000 00107000 urw
PDE(001) 08000000-08400000 00400000 urw
 |-- PTE(000007) 08000000-08007000 00007000 urw
PDE(001) 10000000-10400000 00400000 urw
 |-- PTE(000001) 10000000-10001000 00001000 urw
PDE(001) ee800000-eec00000 00400000 urw
 |-- PTE(000001) eebfd000-eebfe000 00001000 urw
 |-- PTE(000001) eebff000-eec00000 00001000 urw
PDE(001) eec00000-ef000000 00400000 urw
 |-- PTE(00001f) eec00000-eec1f000 0001f000 ur-
PDE(001) ef000000-ef400000 00400000 urw
 |-- PTE(000031) ef000000-ef031000 00031000 ur-
PDE(001) ef400000-ef800000 00400000 ur-
 |-- PTE(000003) ef400000-ef403000 00003000 urw
 |-- PTE(000001) ef420000-ef421000 00001000 urw
 |-- PTE(000001) ef440000-ef441000 00001000 urw
 |-- PTE(000003) ef7ba000-ef7bd000 00003000 urw
 |-- PTE(000001) ef7bd000-ef7be000 00001000 ur-
 |-- PTE(000001) ef7be000-ef7bf000 00001000 urw
 |-- PTE(000001) ef7bf000-ef7c0000 00001000 -rw
 |-- PTE(000040) ef7c0000-ef800000 00040000 urw
PDE(001) ef800000-efc00000 00400000 -rw
 |-- PTE(000008) efbf8000-efc00000 00008000 -rw
PDE(001) efc00000-f0000000 00400000 -rw
 |-- PTE(000003) efc00000-efc03000 00003000 urw
 |-- PTE(000001) efc20000-efc21000 00001000 urw
 |-- PTE(000001) efc40000-efc41000 00001000 urw
 |-- PTE(000003) effba000-effbd000 00003000 urw
 |-- PTE(000001) effbd000-effbe000 00001000 ur-
 |-- PTE(000001) effbe000-effbf000 00001000 urw
 |-- PTE(000001) effbf000-effc0000 00001000 -rw
 |-- PTE(000040) effc0000-f0000000 00040000 urw
PDE(040) f0000000-00000000 10000000 urw
 |-- PTE(010000) f0000000-00000000 10000000 -rw
(qemu)
```

只要关心最后一项关于f0000000-00000000 的映射即可，这个现象让我百思不得其解，因为在程序中所有打印的测试都是没问题的，后来我想到在以前的Lab中我们曾经实现过一个打印地址映射的命令showmappings，我尝试着用自己写的命令打印了一下：

```
K> showmappings 0xf0000000 0xf000c000
0xf0000000 - 0xf0001000     0x0    kernel: read/write
0xf0001000 - 0xf0002000     0x40bb000    kernel: read/write
0xf0002000 - 0xf0003000     0x40ba000    kernel: read/write
0xf0003000 - 0xf0004000     0x40b9000    kernel: read/write
0xf0004000 - 0xf0005000     0x40b8000    kernel: read/write
0xf0005000 - 0xf0006000     0x40b7000    kernel: read/write
0xf0006000 - 0xf0007000     0x40b6000    kernel: read/write
0xf0007000 - 0xf0008000     0x40b5000    kernel: read/write
0xf0008000 - 0xf0009000     0x40b4000    kernel: read/write
0xf0009000 - 0xf000a000     0x40b3000    kernel: read/write
0xf000a000 - 0xf000b000     0x40b2000    kernel: read/write
0xf000b000 - 0xf000c000     0xb000    kernel: read/write
K>
```

发现在这里又是正确的。可能是QEMU对于某些打印区间作了简化把，这个情况我确实不清楚为什么QEMU的打印信息不对了。不过至少写到现在代码在后面没有出现过问题。

TCB结构建立完毕以后，我们可以考虑使用CU在TCB上跑一跑了。从这里开始就涉及到对于CU的操作字的问题。首先回顾一下我们在建立TCB以后可能需要进行的操作：

1. 将其中第一个TCB的物理地址写入CU的General Pointer，通知其要操作的TCB所在的内存位置
2. 给CU一个Start指令，CU开始工作
3. 如果环状的TCB中所有的S位都是0，那么CU将根据Link Address无限循环的处理这些TCB，每次执行完成以后，以中断和修改TCB中C状态位来表明任务已完成(后面我们会说明处理这些消息的方式，现在先不用管)
4. 如果CU碰到了某个TCB的S被设置为1，那么处理完该TCB之后，CU进入Suspend状态，等待用户发送Resume命令恢复运行，那么CU将从当前TCB的下个TCB开始执行

在这里首先回答一下关于CU**执行完成后的响应问题**。如果让CU使用中断的方式提醒我们，效率会比较高，但是比较麻烦的是要去处理中断响应。而如果使用轮询的方式的话，根据我们CBL的结构，只需要每次测试front头的C状态位是否完成即可知道任务的完成状态，非常方便。

我们带着这些需求来看看具体操作方法。控制CU是通过向SCB中写入相应的控制字决定的，具体规定在手册的6.3.2 System Control Block (SCB)中规定：

Table 12. System Control Block

31 16	15 0	
Upper Word	Lower Word	Offset
SCB Command Word	SCB Status Word	Base + 00h
SCB General Pointer		Base + 04h

最主要的两个部分是控制字和状态字:

控制字 ：

Figure 10. SCB Command Word

31 26	25	24	23 20	19	18 16
Specific Interrupt Mask Bits	SI	M	CU Command	0	RU Command

我们主要需要关注三个字段：

- M: 当控制字的这个位被设置成1，那么CU将不会发出任何中断，这个位正是当我们屏蔽中断时第一个需要给出的命令
- CUC (CU Command): 这里对CU的控制命令有以下几种：
  1. 0000 NOP
  2. 0001 CU Start
  3. 0010 CU Resume
  4. 0100 Load Dump Counters Address
  5. 0101 Dump Statistical Counters
  6. 0110 Load CU Base
  7. 0111 Dump and Reset Statistical Counters
  8. 1010 CU Static Resume

  我们要用到的主要是**CU Start**和**CU Resume**，他们对应的需求为：
  - 在建立好TCB并且给CU设置好TCB的地址以后，发出一个CU Start指令，CU开始工作
  - 当CU在某个TCB因为S位被挂起以后，发出一个CU Resume指令可以让其恢复工作

  手册读到这里我们可以把相应的设置场位定义到kern/e100.h中了：

kern/e100.h

```
1 // CU Command Word
2 #define CUC_NOP          0x00
3 #define CUC_START        0x10
4 #define CUC_RESUME       0x20
5 #define CUC_LD_COUNTER   0x40
6 #define CUC_DUMP_SCNT    0x50
7 #define CUC_LOAD_BASE    0x60
8 #define CUC_DUMP_RSCNT   0x70
9 #define CUC_SRESUME      0xa0
```

- RUC (RU Command): 这个我们在后面会详细讲述

看到这里，我们已经知道**如何关闭CU的中断**、使其**开始**和**恢复**执行，但是在开始前需要将TCB的地址告诉CU使其能够开始运行，这个如何设置？这里就涉及到SCB中General Pointer的设置了：

Table 15. SCB General Pointer for the CU Command

RUC Field	RU Command	SCB General Pointer	Added to
0	NOP	Don't care	
1	CU Start	Pointer to first command block in the command block list	CU Base
2	CU Resume	Don't care	
3	CU HPQ Start	Pointer to first command block in the HPQ command block list	CU Base
4	Load Dump Counters Address	Absolute address written to by Dump Counters and Dump & Reset Counters commands	
5	Dump Counters	Don't care	
6	Load CU Base	32-bit Base Register for CU data structures	
7	Dump & Reset Counters	Don't care	
10	CU Static Resume	Don't care	
11	CU HPQ Resume	Don't care	

General Pointer是SCB中根据不同的命令设置的一个场位，**提供某些命令执行时需要的数据**，比如这里我们只需要关注CU Start命令，在执行该命令前，General Pointer里必须写入开始执行的TCB的物理地址，那么CU Start时就可以从该TCB开始执行。而CU Resume就不需要，因为其下一次运行的TCB地址已经在被Suspend的时候被写入了内部寄存器。

设置General Pointer只需要像写入SCB的状态字和控制字一样直接像相应的I/O端口写入值即可。

状态字 ：

Figure 9. SCB Status Word

15 8	7 6	5 2	1	0
STAT / ACK	CUS	RUS	0	0

这里我们只需要关注CU的状态字即可，它主要可能有以下状态：

1. 00: Idle
2. 01: Suspend
3. 10: LPQ Active
4. 11: HQP Active

后面两个我都不知道是干吗的，这次实验里只需要知道前两个即可，因为如果不是前面两个的停止状态，那么当前CU肯定是在工作状态，然后将该状态值也定义到头文件里：

kern/e100.h

```
1 // CU Status Word
2 #define CUS_MASK        0xc0
3 #define CUS_IDLE        0x0
4 #define CUS_SUSPENDED   0x1
5 #define CUS_LPQ_ACTIVE  0x2
6 #define CUS_HQP_ACTIVE  0x3
```

CUS_MASK是由于要从SCB中读取其状态但CUS又是在中间的位置，所以先要用CUS_MASK取出CUS相应场位出来

到这里我们已经将所有的控制指令都了解完毕了，在开始编写真正的驱动之前，还有一个东西我们需要明确，就是发出指令的状态控制，在手册的6.3.2.2 SCB Command Word中提到了这么一段话：

> When software wants to issue an action command, it should write to the Command byte. The CUC and RUC fields of the Command byte specify the actions to be performed by the 8255x. The command is ready for acceptance by the device as soon as it is written into the CUC or RUC field. The actual command execution may not start instantaneously and will depend on current receive and transmit DMA activity. The Command byte is set by the CPU and cleared by the 8255x indicating command acceptance.

因为硬件控制的原因，所以在我们对于SCB写入相应的控制指令时，**并不会马上开始执行**，硬件要过一段时间以后才会接受，那么如何知道是否接受呢？上面的材料提到当CU接受命令以后，**SCB中的命令字会被硬件清除**。进一步的，在手册中的6.5 Starting and Completing Control Commands中也提到：

> * Software must wait for this byte to be cleared before the next control command can be issued.
> * CU and RU control commands must never be issued together in the same SCB write cycle.

通过修改SCB对CU发出一条指令之后，**我们必须等待其命令字被清空以后**，才能继续下面的指令。

好了我们可以正式开始编程了，首先是对CU发出命令的基本模块：

kern/e100.c: e100_exec_cmd()

```
1  static void
2  e100_exec_cmd (int csr_comp, uint8_t cmd)
3  {
4      int scb_command;
5
6      outb(nic.io_base + csr_comp, cmd);
7      do {
8          scb_command = inb(nic.io_base + CSR_COMMAND);
9      } while (scb_command != 0);
10 }
```

csr_comp是SCB命令字的一个字段，cmd是需要执行的命令。发出命令后，我们通过轮询等待命令字被清空确认指令被接受。

初始化的第一步，屏蔽所有的中断：

kern/e100.c: e100_init()

```
1  static void
2  e100_init ()
3  {
4      // Software Reset E100
5      e100_sw_reset(e100);
6
7      // disable all interrupts
8      e100_exec_cmd (CSR_INT, 1);
9
10     cbl_init ();
11 }
```

我们把刚才在e100_attch()中调用的软重启放到了e100_init() 中，并将e100_attch() 换成了调用e100_init() 完成E100的所有初始化。

然后我们看看对于CBL进行初始化的cbl_init():

kern/e100.c: cbl_init()

```
1  static int
2  cbl_append_nop (uint16_t flag)
3  {
4      if (nic.cbl.cb_avail == 0)
5          return -E_CBL_FULL;
6
7      nic.cbl.rear = nic.cbl.rear->next;
8      nic.cbl.rear->cb_control = CBC_NOP | flag;
9
10     return 0;
11 }
12
13 static void
14 cbl_init ()
15 {
16     cbl_alloc ();
17
18     cbl_append_nop (0);
19     cbl_append_nop (0);
20     cbl_append_nop (0);
21     cbl_append_nop (CBF_S);
22
23     outl(nic.io_base + CSR_GP, nic.cbl.front->phy_addr);
24     e100_exec_cmd (CSR_COMMAND, CUC_START);
25 }
```

cbl_append_nop()是在CBL待发送队列中添加一个NOP指令，用于我们待会查看网卡输出判断我们在cbl_alloc() 建立的结构是否被CU正确找到。

具体的话它的工作就是在待发送队列的末尾添加了一个NOP指令TCB。然后设置其flag为我们需要的状态，一般来说就是S=0或者S=1的区别。同样这里使用到了出错状态，我一共定义了四种边界的错误：

kern/e100.h

```
1  // Error CODE
2  #define E_CBL_FULL    1
3  #define E_CBL_EMPTY   2
4  #define E_RFA_FULL    3
5  #define E_RFA_EMPTY   4
```

在cbl_init()第23行，添加完以后我们往SCB的General Pointer里写入了当前CBL里的第一个TCB的物理地址，然后发送了一条CU Start指令。这个逻辑产生的效果应该是，网卡启动后执行了4条NOP指令，然后在最后一条执行完后被挂起。

我们通过make qemu QEMUEXTRA="-debug-e100"启动JOS，其打印出的网卡记录为：

```
1  PCI: 00:00.0: 8086:1237: class: 6.0 (Bridge device) irq: 0
2  PCI: 00:01.0: 8086:7000: class: 6.1 (Bridge device) irq: 0
3  PCI: 00:01.1: 8086:7010: class: 1.1 (Storage controller) irq: 0
4  PCI: 00:01.3: 8086:7113: class: 6.80 (Bridge device) irq: 9
5  PCI: 00:02.0: 1013:00b8: class: 3.0 (Display controller) irq: 0
6  PCI: 00:03.0: 8086:1209: class: 2.0 (Network controller) irq: 11
7  EE100   pci_mmio_map            region 0, addr=0xf2020000, size=0x00001000, type=8
8  EE100   pci_map                 region 1, addr=0x0000c040, size=0x00000040, type=1
9  EE100   pci_mmio_map            region 2, addr=0xf2040000, size=0x00020000, type=0
10 PCI function 00:03.0 (8086:1209) enabled
11 EE100   eepro100_write4         addr=Port+0 val=0x00000000
12 EE100   nic_reset               0x92d8008
13 EE100   nic_selective_reset     checksum=0xbe34
14 EE100   eepro100_write1         addr=Command/Status+3 val=0x01
15 EE100   eepro100_read1          addr=Command/Status+2 val=0x00
16 EE100   eepro100_write_pointer  val=0x040bb000
17 EE100   eepro100_write1         addr=Command/Status+2 val=0x10
18 EE100   action_command          val=0x10 (cu start), status=0x0000, command=0x0000, link=0x040ba000
19 EE100   action_command          CU list with at least one more entry
20 EE100   action_command          val=0x10 (cu start), status=0x0000, command=0x0000, link=0x040b9000
21 EE100   action_command          CU list with at least one more entry
22 EE100   action_command          val=0x10 (cu start), status=0x0000, command=0x0000, link=0x040b8000
23 EE100   action_command          CU list with at least one more entry
24 EE100   action_command          val=0x10 (cu start), status=0x0000, command=0x4000, link=0x040b7000
25 EE100   action_command          CU list empty
26 EE100   eepro100_read1          addr=Command/Status+2 val=0x00
27 FS is running
28 FS can do I/O
```

从13行nic_selective_reset记录后开始，产生的log分别为：

1. 14-15行，向Command/Status+3写入0x01，即写入SCB Command字的M位，用于屏蔽中断，然后读取Command/Status+2即SCB Command字，查看是否被清空，这两句是我们在e100_exec_cmd()中一起执行的两条命令
2. 16行，向General Pointer写入TCB的物理地址
3. 17行，向Command/Status+2写入0x10，即发出CU Start指令
4. 18-25行，CU开始执行TCB中指明的指令了，到了第四个以后执行完毕停止
5. 26行，这句是对应17行中CU Start指令的验证，也是同在e100_exec_cmd()的

现在看起来还挺不错的，但是我们还没有验证以下几个方面：

- TCB的环状结构是否正常
- TCB的发送指令是否正常
- CU的Resume命令是否正常

所以我们将初始化过程修改成这样：

kern/e100.c

```
1  static int
2  cbl_append_transmit (char *data, uint16_t l, uint16_t flag)
3  {
4      if (nic.cbl.cb_avail == 0)
5          return -E_CBL_FULL;
6
7      nic.cbl.rear = nic.cbl.rear->next;
8      nic.cbl.rear->cb_control = CBC_TRANSMIT | flag;
9
10     nic.cbl.rear->cb_cmd_spec.tcb.tcb_tbd_array_addr    = 0xFFFFFFFF;
11     nic.cbl.rear->cb_cmd_spec.tcb.tcb_byte_count        = 1;
12     nic.cbl.rear->cb_cmd_spec.tcb.tcb_thrs              = 0xE0;
13     nic.cbl.rear->cb_cmd_spec.tcb.tcb_tbd_count         = 0;
14
15     memmove (nic.cbl.rear->cb_cmd_spec.tcb.tcb_data, (void *)data, l);
16
17     return 0;
18 }
19
20 static void
21 cbl_init ()
22 {
23     cbl_alloc ();
24
25     cbl_append_nop (0);
26     cbl_append_nop (0);
27     cbl_append_nop (0);
28     cbl_append_nop (CBF_S);
29     cbl_append_nop (0);
30     cbl_append_nop (0);
31     cbl_append_nop (0);
32     cbl_append_nop (0);
33     cbl_append_nop (0);
34
35     cbl_append_transmit ("aaaax", 5, 0);
36
37
38     outl(nic.io_base + CSR_GP, nic.cbl.front->phy_addr);
39     e100_exec_cmd (CSR_COMMAND, CUC_START);
40
41     e100_exec_cmd (CSR_COMMAND, CUC_RESUME);
42 }
```

这里添加了一个函数`cbl_append_transmit()`添加一个发送指令的TCB到队尾，不再赘述。

然后初始化过程中我们添加发送了CBL中所有的10个TCB，并且最后一次发送transmit不是suspend，那么在我们发出的第一次Start指令后，应该会在第四个NOP指令停下来，接下来发出Resume指令后，应该会循环执行CBL中的所有TCB后在同样在第四个NOP停下来，启动JOS后打印的信息如下：

```
PCI function 00:03.0 (8086:1209) enabled
EE100   eepro100_write4         addr=Port+0 val=0x00000000
EE100   nic_reset               0x8a8e008
EE100   nic_selective_reset     checksum=0xbe34
EE100   eepro100_write1         addr=Command/Status+3 val=0x01
EE100   eepro100_read1          addr=Command/Status+2 val=0x00
EE100   eepro100_write_pointer  val=0x040bb000
EE100   eepro100_write1         addr=Command/Status+2 val=0x10
EE100   action_command          val=0x10 (cu start), status=0x0000, command=0x0000, link=0x040ba000
EE100   action_command          CU list with at least one more entry
EE100   action_command          val=0x10 (cu start), status=0x0000, command=0x0000, link=0x040b9000
EE100   action_command          CU list with at least one more entry
EE100   action_command          val=0x10 (cu start), status=0x0000, command=0x0000, link=0x040b8000
EE100   action_command          CU list with at least one more entry
EE100   action_command          val=0x10 (cu start), status=0x0000, command=0x4000, link=0x040b7000
EE100   action_command          CU list empty
```

```
EE100   eepro100_read1          addr=Command/Status+2 val=0x00
EE100   eepro100_write1         addr=Command/Status+2 val=0x20
EE100   eepro100_cu_command     CU resuming
EE100   action_command          val=0x10 (cu start), status=0x0000, command=0x0000, link=0x040b6000
EE100   action_command          CU list with at least one more entry
EE100   action_command          val=0x10 (cu start), status=0x0000, command=0x0000, link=0x040b5000
EE100   action_command          CU list with at least one more entry
EE100   action_command          val=0x10 (cu start), status=0x0000, command=0x0000, link=0x040b4000
EE100   action_command          CU list with at least one more entry
EE100   action_command          val=0x10 (cu start), status=0x0000, command=0x0000, link=0x040b3000
EE100   action_command          CU list with at least one more entry
EE100   action_command          val=0x10 (cu start), status=0x0000, command=0x0000, link=0x040b2000
EE100   action_command          CU list with at least one more entry
EE100   action_command          val=0x10 (cu start), status=0x0000, command=0x0004, link=0x040bb000
EE100   action_command          transmit, TBD array address 0xffffffff, TCB byte count 0x0005, TBD count 0
EE100   action_command          TBD (simplified mode): buffer address 0x040b2010, size 0x0005
EE100   action_command          0x8a8e008 sending frame, len=5, 61 61 61 61 78
EE100   action_command          CU list with at least one more entry
EE100   action_command          val=0x10 (cu start), status=0xa000, command=0x0000, link=0x040ba000
EE100   action_command          CU list with at least one more entry
EE100   action_command          val=0x10 (cu start), status=0xa000, command=0x0000, link=0x040b9000
EE100   action_command          CU list with at least one more entry
EE100   action_command          val=0x10 (cu start), status=0xa000, command=0x0000, link=0x040b8000
EE100   action_command          CU list with at least one more entry
EE100   action_command          val=0x10 (cu start), status=0xa000, command=0x4000, link=0x040b7000
EE100   action_command          CU list empty
EE100   eepro100_read1          addr=Command/Status+2 val=0x00
FS is running
FS can do I/O
```

可以看到发送的流程和我们的预期是正常的，但是这里看到发送的时候打印的信息不是很给力：

```
EE100   action_command          transmit, TBD array address 0xffffffff, TCB byte count 0x0005, TBD count 0
EE100   action_command          TBD (simplified mode): buffer address 0x040b2010, size 0x0005
EE100   action_command          0x8a8e008 sending frame, len=5, 61 61 61 61 78
EE100   action_command          CU list with at least one more entry
```

尤其是在发送大量数据的话在大量包记录里很难找到有效信息，我们可以使用提供的包拦截参数将其记录到文件，使用make qemu QEMUEXTRA="-debug-e100 -pcap slirp.cap"打印调试信息的同时将网卡的包抓取后放入slirp.cap文件中，这个文件是一个二进制文件，人无法直接读取，需要tcpdump来为我们解析，使用tcpdump -XXr slirp.cap命令打印其中的内容，得到的输出如下：

```
zhangchi@zhangchi-vostro1400:~/lab$ tcpdump -XXr slirp.cap
reading from file slirp.cap, link-type EN10MB (Ethernet)
15:05:39.002011 [|ether]
        0x0000:  6161 6161 78                                  aaaax
zhangchi@zhangchi-vostro1400:~/lab$
```

可以看到我们发送的aaaax消息，这样看着就直观多了。

kern/pci.c

kern/pci.c

2.3.1 C Structures

SCB COMMAND Word：当CPU接受命令，即将COMMAND清空：

kern/pci.c

kern/pci.c

kern/pci.c

2.4 Transmitting Packets: Network Server

3 Receiving packets and the web server

3.1 Receiving Packets

3.2 Receiving Packets: Network Server

3.3 The Web Server

: ()

: ()

: ()