...making Linux just a little more fun!

The Free International Online Linux Monthly ISSN: 1934-371X Main site: http://linuxgazette.net

Writing Network Device Drivers for Linux

By Mohan Lal Jangir

Introduction

This article has been written for kernel newcomers interested in learning about network device drivers. It assumes that reader has a significant exposure to C and the Linux environment.

This article is based on a network driver for the RealTek 8139 network card. I chose the RealTek chip for two reasons: First, RealTek provides technical specifications for its chips free of cost. (Thanks, RealTek!) Second; it's quite cheap. It is possible to get the chip under Rs 300 (approximately US$7) in Indian markets.

The driver presented in this article is minimal; it simply sends and receives packets and maintains some statistics. For a full-fledged and professional-grade driver, please refer to the Linux source.

Preparing for Driver Development

Before starting driver development, we need to set up our system for it. This article was written and tested on Linux 2.4.18, which contains the source code for the RealTek8139 chip driver. It's very likely that the kernel you are running has the driver compiled either within the kernel itself or as a module. It's advisable to build a kernel which does *not* have the RealTek8139 driver in any form, to avert unnecessary surprises. If you don't know how to recompile the Linux kernel, I recommend you take a look at http://www.linuxheadquarters.com/howto/tuning/kernelreasons.shtml.

From this point of discussion onwards, it is assumed that you have a working kernel, which does not have driver for RealTek8139. You'll also need the technical specifications for the chip, which you can download from http://www.realtek.com.tw/. The last activity in this series is to properly insert the NIC into the PCI slot, and we are ready to go ahead.

It is strongly recommended to have Rubini's *Linux Device Drivers* book with you for quick API reference. This is the best resource known to me for Linux device driver development, as of now.

Starting Driver Development

Driver development breaks down into the following steps:

1. Detecting the device
2. Enabling the device
3. Understanding the network device
4. Bus-independent device access
5. Understanding the PCI configuration space
6. Initializing net_device
7. Understanding RealTek8139's transmission mechanism
8. Understanding RealTek8139's receiving mechanism
9. Making the device ready to transmit packets
10. Making the device ready to receive packets

Detecting the Device

As a first step, we need to detect the device of our interest. The Linux kernel provides a rich set of APIs to detect a device over the PCI bus (Plug & Play), but we will go for the simplest one and the API is pci_find_device.

```
#define REALTEK_VENDER_ID  0x10EC
#define REALTEK_DEVICE_ID   0x8139

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/stddef.h>
#include <linux/pci.h>
int init_module(void)
{
    struct pci_dev *pdev;
    pdev = pci_find_device(REALTEK_VENDER_ID, REALTEK_DEVICE_ID, NULL);
    if(!pdev)
        printk("<1>Device not found\n");
    else
        printk("<1>Device found\n");
    return 0;
}
```

Table 1: Detecting the device

Each vendor has a unique ID assigned, and each vendor assigns a unique ID to a particular kind of device. The macros REALTEK_VENDER_ID and REALTEK_DEVICE_ID indicate those IDs. You can find these values from the "PCI Configuration Space Table" in the RealTek8139 specifications.

Enabling the Device

After detecting the device, we need to enable the device before starting any kind of interaction or communication with the device. The code snippet shown in Table 1 can be extended for device detection and enabling the device.

```
static struct pci_dev* probe_for_realtek8139(void)
{
        struct pci_dev *pdev = NULL;
        /* Ensure we are not working on a non-PCI system *
        if(!pci_present( )) {
               LOG_MSG("<1>pci not present\n");
               return pdev;
        }

#define REALTEK_VENDER_ID  0x10EC
#define REALTEK_DEVICE_ID  0X8139

        /* Look for RealTek 8139 NIC */
        pdev = pci_find_device(REALTEK_VENDER_ID, REALTEK_DEVICE_ID, NULL);
        if(pdev) {
               /* device found, enable it */
               if(pci_enable_device(pdev)) {
                       LOG_MSG("Could not enable the device\n");
                       return NULL;
               }
               else
                       LOG_MSG("Device enabled\n");
        }
        else {
               LOG_MSG("device not found\n");
               return pdev;
        }
        return pdev;
}

int init_module(void)
{
        struct pci_dev *pdev;
        pdev = probe_for_realtek8139();
        if(!pdev)
               return 0;

        return 0;
}
```

Table 2: Detecting and Enabling the Device

In Table 2, the function probe_for_realtek8139 performs the following tasks:

- Ensures that we are working on a PCI-capable system
    - pci_present returns NULL if system does not have PCI support
- Detects the RealTek 8139 device as explained in Table 1
- Enables the device (by calling pci_enable_device), if found

For time being, we temporarily suspend the thread of driver code study; instead, we look into some important topics in order to understand the Linux view of a network device. We will look at network devices, and the difference between memory-mapped I/O, port-mapped I/O, and PCI configuration space.

Understanding Network Devices

We have detected the PCI device and enabled it, but the networking stack in Linux sees interfaces as network devices. This is represented by the structure net_device. This means that the networking stack issues commands to the network device (represented by net_device), and the driver shall transfer those commands to the PCI device. Table 3 lists some important fields of the structure net_device, which will be used later in this article.

```
struct net_device
{
  char *name;
  unsigned long base_addr;
  unsigned char addr_len;
  unsigned char dev_addr[MAX_ADDR_LEN];
  unsigned char broadcast[MAX_ADDR_LEN];
  unsigned short hard_header_len;
  unsigned char irq;
  int (*open) (struct net_device *dev);
  int (*stop) (struct net_device *dev);
  int (*hard_start_xmit) (struct sk_buff *skb,  struct net_device *dev);
  struct net_device_stats* (*get_stats)(struct net_device *dev);
  void *priv;
};
```

Table 3: Structure net_device

Although this structure has many more members, for our minimal driver, these members are good enough. The following section describes the structure members:

- name - The name of the device. If the first character of the name is null, then register_netdev assigns it the name "ethn", where n is suitable numeric. For example, if your system already has eth0 and eth1, your device will be named eth2.
- base_addr - The I/O base address. We will discuss more about I/O addresses later in this article.
- addr_len - Hardware address (MAC address) length. It is 6 for Ethernet interfaces.
- dev_addr - Hardware address (Ethernet address or MAC address)
- broadcast - device broadcast address. It is FF:FF:FF:FF:FF:FF for Ethernet interfaces
- hard_header_len - The "hardware header length" is the number of octets that lead the transmitted packet before IP header, or other protocol information. The value of hard_header_len is 14 for Ethernet interfaces.
- irq - The assigned interrupt number.
- open - This is a pointer to a function that opens the device. This function is called whenever ifconfig activates the device (for example, "ifconfig eth0 up"). The open method should register any system resources it needs (I/O ports, IRQ, DMA, etc.), turn on the hardware and increment module usage count.
- stop - This is a pointer to a function that stops the interface. This function is called whenever ifconfig deactivates the device (for example, "ifconfig eth0 down"). The stop method releases all the resources acquired by open function.
- hard_start_xmit - This function transfers a given packet on the wire. The first argument of the function is a pointer to structure sk_buff. Structure sk_buff is used to hold packets in Linux networking stacks. Although this article does not require detailed knowledge about sk_buff's structure, its details can be found at http://www.tldp.org/LDP/khg/HyperNews/get/net/net-intro.html.
- get_stats - This function provides interfaces statistics. The output of the command "ifconfig eth0" has most of the fields from get_stats.
- priv - Private data to the driver. The driver owns this field and can use it at will. We will see later that our driver uses this field to keep data related to PCI devices.

Although we have not mentioned all members of the net_device structure, please note especially that there is no member function for receiving packets. This is done by the device interrupt handler, as we will see later in this article.

Bus-Independent Device Access

Note: This section has been taken from Alan Cox's book Bus-Independent Device Accesses available at http://tali.admingilde.org/linux-docbook/deviceiobook.pdf

Linux provides an API set that abstracts performing I/O operations across all buses and devices, allowing device drivers to be written independent of bus type.

Memory-Mapped I/O

The most widely supported form of I/O is memory-mapped I/O. That is, a part of the CPU's address space is interpreted not as accesses to memory, but as accesses to a device. Some architectures define devices to be at a fixed address, but most have some method of discovering devices. The PCI bus walk is a good example of such a scheme. This document does not cover how to receive such an address, but assumes you are starting with one.

Physical addresses are of type unsigned long. These addresses should not be used directly. Instead, to get an address suitable for passing to the functions described below, you should call ioremap. An address suitable for accessing the device will be returned to you.

After you've finished using the device (say, in your module's exit routine), call iounmap in order to return the address space to the kernel. Most architectures allocate new address space each time you call ioremap, and they can run out unless you call iounmap.

Accessing the device

The part of the interface most used by drivers is reading and writing memory-mapped registers on the device. Linux provides interfaces to read and write 8-bit, 16-bit, 32-bit and 64-bit quantities. Due to a historical accident, these are named byte, word, long, and quad accesses. Both read and write accesses are supported; there is no prefetch support at this time. The functions are named readb, readw, readl, readq, writeb, writew, writel, and writeq.

Some devices (such as framebuffers) would like to use larger transfers that are more than 8 bytes at a time. For these devices, the memcpy_toio, memcpy_fromio and memset_io functions are provided. Do not use memset or memcpy on I/O addresses; they are not guaranteed to copy data in order.

The read and write functions are defined to be ordered. That is, the compiler the the the is not permitted to reorder the I/O sequence. When the ordering can be compiler optimized, you can use __readb and friends to indicate the relaxed ordering. Use this with care. The rmb provides a read memory barrier. The wmb provides a write memory barrier.

While the basic functions are defined to be synchronous with respect to each other and ordered with respect to each other the buses the devices sit on may themselves have asynchronocity. In particular many authors are not comfortable by the fact that PCI bus writes are posted asynchronously. An author of the driver must issue a read from the same device to ensure that writes have occurred in the manner the author wanted it. This kind of property cannot be hidden from driver writers in the API.

Port Space Access

Another form of I/O commonly supported is Port Space. This is a range of addresses different from the normal memory address space. Access to these addresses is generally not as fast as accesses to the memory mapped addresses, and it also has a potentially smaller address space.

Unlike with memory mapped I/O, no preparation is required to access port space.

Accessing Port Space or I/O mapped devices

Accesses to this space are provided through a set of functions which allow 8-bit, 16-bit and 32-bit accesses; also known as byte, word and long. These functions are inb, inw, inl, outb, outw and outl.

Some variants are provided for these functions. Some devices require that accesses to their ports are slowed down. This functionality is provided by appending a _p to the end of the function. There are also equivalents to memcpy. The ins and outs functions copy bytes, words or longs to/from the given port.

Understanding PCI Configuration Space

In this section, we will look at PCI configuration space. PCI devices feature a 256-byte address space. The first 64 bytes are standardized while the rest of the bytes are device dependent. Figure 1 shows the standard PCI configuration space.

PCI Configuration Space

Figure 1: PCI Configuration Space

The fields "Vendor ID" and "Device ID" are unique identifiers assigned to the vendor and the device, respectively. (We have seen them in the section "Device Detection".) Another field to note is "Base Address Registers", popularly known as BAR. We will see how BARs are used shortly.

Initializing net_device

Now it's time to revert back to driver development. Before that I remind you about the priv field of the structure net_device. We will declare a structure which holds data private to our device and that structure shall be pointed to by member priv. The structure has the following members (We will update structure members as we progress).

```
struct rtl8139_private
{
        struct pci_dev *pci_dev;  /* PCI device */
        void *mmio_addr; /* memory mapped I/O addr */
        unsigned long regs_len; /* length of I/O or MMI/O region */
};
```

Table 4: rtl8139_private structure

Now we define a net_device pointer and initialize it, in the rest of the init_module function.

```
#define DRIVER "rtl8139"
static struct net_device *rtl8139_dev;

static int rtl8139_init(struct pci_dev *pdev, struct net_device **dev_out)
{
        struct net_device *dev;
        struct rtl8139_private *tp;

        /*
         * alloc_etherdev allocates memory for dev and dev->priv.
         * dev->priv shall have sizeof(struct rtl8139_private) memory
         * allocated.
         */
        dev = alloc_etherdev(sizeof(struct rtl8139_private));
        if(!dev) {
               LOG_MSG("Could not allocate etherdev\n");
               return -1;
        }

        tp = dev->priv;
        tp->pci_dev = pdev;
        *dev_out = dev;

        return 0;
}

int init_module(void)
{
        struct pci_dev *pdev;
        unsigned long mmio_start, mmio_end, mmio_len, mmio_flags;
        void *ioaddr;
        struct rtl8139_private *tp;
        int i;

        pdev = probe_for_realtek8139( );
        if(!pdev)
               return 0;

        if(rtl8139_init(pdev, &rtl8139_dev)) {
               LOG_MSG("Could not initialize device\n");
               return 0;
        }

        tp = rtl8139_dev->priv; /* rtl8139 private information */

        /* get PCI memory mapped I/O space base address from BAR1 */
        mmio_start = pci_resource_start(pdev, 1);
        mmio_end = pci_resource_end(pdev, 1);
        mmio_len = pci_resource_len(pdev, 1);
        mmio_flags = pci_resource_flags(pdev, 1);

        /* make sure above region is MMI/O */
        if(!(mmio_flags & I/ORESOURCE_MEM)) {
                LOG_MSG("region not MMI/O region\n");
                goto cleanup1;
        }

        /* get PCI memory space */
        if(pci_request_regions(pdev, DRIVER)) {
                LOG_MSG("Could not get PCI region\n");
                goto cleanup1;
        }

        pci_set_master(pdev);

        /* ioremap MMI/O region */
        ioaddr = ioremap(mmio_start, mmio_len);
        if(!ioaddr) {
                LOG_MSG("Could not ioremap\n");
                goto cleanup2;
        }

        rtl8139_dev->base_addr = (long)ioaddr;
        tp->mmio_addr = ioaddr;
        tp->regs_len = mmio_len;

        /* UPDATE NET_DEVICE */

        for(i = 0; i < 6; i++) {  /* Hardware Address */
                rtl8139_dev->dev_addr[i] = readb(rtl8139_dev->base_addr+i);
                rtl8139_dev->broadcast[i] = 0xff;
        }
        rtl8139_dev->hard_header_len = 14;

        memcpy(rtl8139_dev->name, DRIVER, sizeof(DRIVER)); /* Device Name */
        rtl8139_dev->irq = pdev->irq;  /* Interrupt Number */
        rtl8139_dev->open = rtl8139_open;
        rtl8139_dev->stop = rtl8139_stop;
        rtl8139_dev->hard_start_xmit = rtl8139_start_xmit;
        rtl8139_dev->get_stats = rtl8139_get_stats;

        /* register the device */
        if(register_netdev(rtl8139_dev)) {
                LOG_MSG("Could not register netdevice\n");
                goto cleanup0;
        }

        return 0;
}
```

Table 5: net_device initialization

It's time to explain what we have done in Table 5. Function probe_for_realtek8139, we have already seen. Function rtl8139_init allocates memory for global pointer rtl8139_dev, which we shall be using as net_device. Additionally, this function sets the member pci_dev of rtl8139_private to the detected device.

Our next objective is to get the base_addr field of the net_device. This is the starting memory location of device registers. This driver has been written for memory-mapped I/O only. To get the memory-mapped I/O base address, we use PCI APIs like pci_resource_start, pci_resource_end, pci_resource_len, pci_resource_flags etc. These APIs let us read the PCI configuration space without knowing internal details. The second argument to these APIs is the BAR number. If you see, RealTek8139 specifications, you will find that the first BAR (numbered as 0) is I/OAR, while second BAR (numbered as 1) is MEMAR. Since this driver is using memory-mapped I/O, we pass the second argument as 1. Before accessing the addresses returned by the above APIs, we have to do two things. First is to reserve the above resources (memory space) by driver; this is done by calling the function pci_request_regions. The second thing is to remap I/O addresses as explained in section above on Memory-Mapped I/O. The remapped io_addr is assigned to the base_addr member of the net_device, and this is the point where we can start to read/write the device registers.

The rest of the code in Table 5 does straightforward initialization of net_device. Note that now we are reading the hardware address from the device and assigning it to dev_addr. If you see "Register Descriptions" in RealTek8139 specification, the first 6 bytes are the hardware address of the device. Also we have initialized function pointer members but haven't defined any corresponding function. For time being, we define dummy functions to compile the module.

```
static int rtl8139_open(struct net_device *dev) { LOG_MSG("rtl8139_open is
called\n"); return 0; }

static int rtl8139_stop(struct net_device *dev)
{
        LOG_MSG("rtl8139_open is called\n");
        return 0;
}

static int rtl8139_start_xmit(struct sk_buff *skb, struct net_device *dev)
{
        LOG_MSG("rtl8139_start_xmit is called\n");
        return 0;
}

static struct net_device_stats* rtl8139_get_stats(struct net_device *dev)
{
        LOG_MSG("rtl8139_get_stats is called\n");
        return 0;
}
```

Table 6: Dummy functions

Note that the error-handling part has been skipped in init_module. You can write it by looking into the cleanup_module function defined below:

```
void cleanup_module(void)
{
        struct rtl8139_private *tp;
        tp = rtl8139_dev->priv;

        iounmap(tp->mmio_addr);
        pci_release_regions(tp->pci_dev);

        unregister_netdev(rtl8139_dev);
        pci_disable_device(tp->pci_dev);
        return;
}
```

Table 7: Function cleanup_module

Now we have a dummy or template driver ready. Compile the module and insert it as explained in Table 8 (assuming the kernel source is in /usr/src/linux-2.4.18).

```
 gcc   - c  rtl8139.c   - D__KERNEL__  -DMODULE   - I /usr/src/linux-2.4.18/include
 insmod  rtl8139.o
```

Table 8: Compiling the driver

Now execute a series of commands; "*ifconfig*", "*ifconfig - a*", "*ifconfig rtl8139 up*", "*ifconfig*" and "*ifconfig rtl8139 down*", and observe their output. These calls show you when each function is called. If everything goes fine, you should see device rtl8139 when you issue "ifconfig - a" and should get message "function rtl8139_get_stat" called. You should get message "function rtl8139_open called" when you issue command "ifconfig rtl8139 up". Similarly you should get "function rtl8139_stop called" when you issue command "ifconfig rtl8139 down".

Now again, we stop driver development in order to better understand the device transmission and receiving mechanism.

Understanding the RealTek 8139 Transmission Mechanism

In this section, I describe RealTek8139 transmission mechanism; however I recommend to download "RTL8139 (A/B) Programming Guide", which provides exact details. RealTek8139 has 4 Transmission Descriptors; each descriptor has a fixed I/O address offset. The 4 descriptors are used round-robin. This means that for transmitting four packets, the driver will use descriptor 0, descriptor 1, descriptor 2 and descriptor 3 in round-robin order. For transmitting next packet, driver will use descriptor 0 again (provided that is available). If you read the RealTek8139 specification, the section "Register Description" has TSAD0, TSAD1, TSAD2 and TSAD3 registers at offset 0x20, 0x24, 0x28, 0x2C, respectively. These registers store "Transmit Start Address of Descriptors" i.e., they store starting address (in memory) of packets to be transmitted. Later device reads packet contents from these addresses, DMA to its own FIFO, and transmits on wire.

We will shortly see that this driver allocates DMAable memory for packet contents, and stores the address of that memory in TSAD registers.

Understanding the RealTek 8139 Receiving Mechanism

The receive path of RTL8139 is designed as a ring buffer (A liner memory, managed as ring memory). Whenever the device receives a packet, packet contents are stored in ring buffer memory, and the location of the next packet to store is updated (to first packet starting address + first packet length). The device keeps on storing packets in this fashion until linear memory is exhausted. In that case, the device starts again writing at the starting address of linear memory, thus making it a ring buffer.

Making Device Ready to Transmit Packets

In this section, we discuss driver source used to make the device ready for transmission. We defer discussion of the receiving source to further sections. We will discuss functions rtl8139_open and rtl8139_stop, in this section. Before that, we enhance our rtl8139_private structure, to accommodate members to hold data related to packet transmission.

```
#define NUM_TX_DESC 4
struct rtl8139_private
{
        struct pci_dev *pci_dev;  /* PCI device */
        void *mmio_addr; /* memory mapped I/O addr */
        unsigned long regs_len; /* length of I/O or MMI/O region */
        unsigned int tx_flag;
        unsigned int cur_tx;
        unsigned int dirty_tx;
        unsigned char *tx_buf[NUM_TX_DESC];   /* Tx bounce buffers */
        unsigned char *tx_bufs;        /* Tx bounce buffer region. */
        dma_addr_t tx_bufs_dma;
};
```

Table 9: rtl8139_private structure

Member tx_flag shall contain transmission flags to notify the device about some parameters described shortly. Field cur_tx shall hold current transmission descriptor, while dirty_tx denotes the first of transmission descriptors, which have not completed transmission. (This also means that, we can't use dirty descriptor for further packet transmission until previous packet is transmitted completely.) Array tx_buf holds addresses of 4 "transmission descriptors". Field tx_bufs is also used in same context, as we will see shortly. Both tx_buf and tx_bufs do hold kernel virtual address, which can be used by the driver, but the device cannot use these addresses. The device need to access physical addresses, which are stored in field tx_bufs_dma. Here is a list of register offsets, used in code. You can get more details about these values from the RealTek8139 specifications.

```
#define TX_BUF_SIZE  1536  /* should be at least MTU + 14 + 4 */
#define TOTAL_TX_BUF_SIZE  (TX_BUF_SIZE * NUM_TX_SIZE)

/* 8139 register offsets */
#define TSD0          0x10
#define TSAD0       0x20
#define RBSTART  0x30
#define CR                0x37
#define CAPR         0x38
#define IMR            0x3c
#define ISR             0x3e
#define TCR           0x40
#define RCR           0x44
#define MPC           0x4c
#define MULINT    0x5c

/* TSD register commands */
#define TxHostOwns    0x2000
#define TxUnderrun    0x4000
#define TxStatOK      0x8000
#define TxOutOfWindow 0x20000000
#define TxAborted     0x40000000
#define TxCarrierLost 0x80000000

/* CR register commands */
#define RxBufEmpty 0x01
#define CmdTxEnb   0x04
#define CmdRxEnb   0x08
#define CmdReset   0x10

/* ISR Bits */
#define RxOK       0x01
#define RxErr      0x02
#define TxOK       0x04
#define TxErr      0x08
#define RxOverFlow 0x10
#define RxUnderrun 0x20
```

```
#define RxFIFOOver 0x40
#define CableLen   0x2000
#define TimeOut    0x4000
#define SysErr     0x8000

#define INT_MASK (RxOK | RxErr | TxOK | TxErr | \
               RxOverFlow | RxUnderrun | RxFIFOOver | \
               CableLen | TimeOut | SysErr)
```

Table 10: RTL 8139 Register Definitions

With above definition, we look into function rtl8139_open:

```
static int rtl8139_open(struct net_device *dev)
{
        int retval;
        struct rtl8139_private *tp = dev->priv;

        /* get the IRQ
         * second arg is interrupt handler
         * third is flags, 0 means no IRQ sharing
         */
        retval = request_irq(dev->irq, rtl8139_interrupt, 0, dev->name, dev);
        if(retval)
               return retval;

        /* get memory for Tx buffers
         * memory must be DMAable
         */
        tp->tx_bufs = pci_alloc_consistent(
                       tp->pci_dev, TOTAL_TX_BUF_SIZE, &tp->tx_bufs_dma);

        if(!tp->tx_bufs) {
               free_irq(dev->irq, dev);
               return -ENOMEM;
        }

        tp->tx_flag = 0;
        rtl8139_init_ring(dev);
        rtl8139_hw_start(dev);

        return 0;
}

static void rtl8139_init_ring (struct net_device *dev)
{
        struct rtl8139_private *tp = dev->priv;
        int i;

        tp->cur_tx = 0;
        tp->dirty_tx = 0;

        for (i = 0; i < NUM_TX_DESC; i++)
               tp->tx_buf[i] = &tp->tx_bufs[i * TX_BUF_SIZE];

        return;
}

static void rtl8139_hw_start (struct net_device *dev)
{
        struct rtl8139_private *tp = dev->priv;
        void *ioaddr = tp->mmio_addr;
        u32 i;

        rtl8139_chip_reset(ioaddr);

        /* Must enable Tx before setting transfer thresholds! */
        writeb(CmdTxEnb, ioaddr + CR);

        /* tx config */
        writel(0x00000600, ioaddr + TCR); /* DMA burst size 1024 */

        /* init Tx buffer DMA addresses */
        for (i = 0; i < NUM_TX_DESC; i++) {
               writel(tp->tx_bufs_dma + (tp->tx_buf[i] - tp->tx_bufs),
                                      ioaddr + TSAD0 + (i * 4));
        }
        /* Enable all known interrupts by setting the interrupt mask. */
        writew(INT_MASK, ioaddr + IMR);

        netif_start_queue (dev);
        return;
}

static void rtl8139_chip_reset (void *ioaddr)
{
        int i;

        /* Soft reset the chip. */
        writeb(CmdReset, ioaddr + CR);

        /* Check that the chip has finished the reset. */
        for (i = 1000; i > 0; i--) {
               barrier();
               if ((readb(ioaddr + CR) & CmdReset) == 0)
                       break;
               udelay (10);
        }
        return;
}
```

Table 11: Writing the open function

Now, we explain the code in Table 11. The function rtl8139_open starts with requesting the IRQ by calling API request_irq. In this function, we register the interrupt handler rtl8139_interrupt. This function shall be called by kernel, whenever the device generates an interrupt. Now, we allocate memory, where outgoing packets reside before being sent on wire. Note that API pci_allocate_consistant returns kernel virtual address. The physical address is returned in third argument, which is later used by driver. Also observe that we have allocated memory needed for all four descriptors. Function rtl8139_init_ring distributes this memory to four descriptors. Here, we call function rtl8139_hw_start to make the device ready for transmitting packets. At first, we reset the device, so that device shall be in a predictable and known state. This is done by writing reset value (described in specification) in CR (Command Register). We wait until the written value is read back, which means device has reset. The next function, barrier ( ), is called to force the kernel to do required memory I/O immediately without doing any optimization. Once the device is reset, we enable transmission mode of the device by writing transmission enable value in CR. Next, we configure TCR (Transmission Configuration Register). The only thing we are specifying to TCR register is "Max DMA Burst Size per Tx DMA Burst". The rest we leave at default values. (See specification for more details.) Now we write the DMAable address of all four descriptors to TSAD (Transmission Start Address Descriptor) registers. Next, we enable the interrupt, by writing in IMR (Interrupt Mask Register). This register lets us configure the interrupts; the device will be generating. Last, we call netif_start_queue to tell the kernel that device is ready. The only thing remaining is writing the rtl8139_interrupt function. For the time being, let's skip this. At this time, the device is ready to send packets, but the function to send packets out is missing. (Remember hard_start_xmit.) So, let's do it.

```
static int rtl8139_start_xmit(struct sk_buff *skb, struct net_device *dev)
{
        struct rtl8139_private *tp = dev->priv;
        void *ioaddr = tp->mmio_addr;
        unsigned int entry = tp->cur_tx;
        unsigned int len = skb->len;
#define ETH_MIN_LEN 60  /* minimum Ethernet frame size */
        if (len < TX_BUF_SIZE) {
               if(len < ETH_MIN_LEN)
                       memset(tp->tx_buf[entry], 0, ETH_MIN_LEN);
               skb_copy_and_csum_dev(skb, tp->tx_buf[entry]);
               dev_kfree_skb(skb);
        } else {
               dev_kfree_skb(skb);
               return 0;
        }
        writel(tp->tx_flag | max(len, (unsigned int)ETH_MIN_LEN),
                       ioaddr + TSD0 + (entry * sizeof (u32)));
        entry++;
        tp->cur_tx = entry % NUM_TX_DESC;

        if(tp->cur_tx == tp->dirty_tx) {
               netif_stop_queue(dev);
        }
        return 0;
}
```

Table 12: Writing start_xmit function

The function rtl8139_start_xmit, explained in Table 12, is very trivial. First, it finds the available transmission descriptor and then checks that the packet size is at least 60 bytes (as Ethernet packet size can't be less than 60 bytes). Once this is ensured, the function skb_copy_and_csum_dev is called, which copies the packet contents to the DMA capable memory. In the next writel, we inform the device about the packet length. At this time, the packet is transmitted on the wire. Next, we determine the next available transmission descriptors, and, if this happens to be equal to a dirty descriptor, we stop the device; otherwise we simply return.

Our device is now ready to send packets out. (Remember, we can't receive packets, yet.) Compile the driver, and try sending ping packets out of the host. At other end, you should see some ARP packets. Even remote hosts reply to ARP packets; they are useless for us, as we are not ready to receive packets.

Making Device Ready to Receive Packets

Now, we will make the device ready to receive packets. For this, we will look into some of already discussed functions, and then the interrupt handler. First, we extend the structure rtl8139_private to accommodate variables needed to receive packets.

```
struct rtl8139_private
{
        struct pci_dev *pci_dev;  /* PCI device */
        void *mmio_addr; /* memory mapped I/O addr */
        unsigned long regs_len; /* length of I/O or MMI/O region */
        unsigned int tx_flag;
        unsigned int cur_tx;
        unsigned int dirty_tx;
        unsigned char *tx_buf[NUM_TX_DESC];   /* Tx bounce buffers */
        unsigned char *tx_bufs;        /* Tx bounce buffer region. */
        dma_addr_t tx_bufs_dma;

        struct net_device_stats stats;
        unsigned char *rx_ring;
        dma_addr_t rx_ring_dma;
        unsigned int cur_rx;
};
```

Table 13: Extending rtl8139_private structure

The member stats shall keep device statistics (most of the ifconfig statistics is from this structure). The next member, rx_ring, is the kernel address of memory where received packets are stored, while rx_ring_dma is the physical address of the same memory. Member cur_rx is used to keep track of next packet writing, as we will see shortly.

Now we re-look into rtl8139_open function, where we allocated memory for transmission side only. Now, we allocate memory for packet receiving also.

```
/* Size of the in-memory receive ring. */
#define RX_BUF_LEN_IDX 2         /* 0==8K, 1==16K, 2==32K, 3==64K */
#define RX_BUF_LEN     (8192 << RX_BUF_LEN_IDX)
#define RX_BUF_PAD     16           /* see 11th and 12th bit of RCR: 0x44 */
#define RX_BUF_WRAP_PAD 2048   /* spare padding to handle pkt wrap */
#define RX_BUF_TOT_LEN  (RX_BUF_LEN + RX_BUF_PAD + RX_BUF_WRAP_PAD)

/* this we have already done */
tp->tx_bufs = pci_alloc_consistent(tp->pci_dev, TOTAL_TX_BUF_SIZE, &tp->tx_bufs_dma);

/* add this code to rtl8139_function */
tp->rx_ring = pci_alloc_consistent(tp->pci_dev, RX_BUF_TOT_LEN, &tp->rx_ring_dma);

if((!tp->tx_bufs)  || (!tp->rx_ring)) {
        free_irq(dev->irq, dev);

        if(tp->tx_bufs) {
                        pci_free_consistent(tp->pci_dev, TOTAL_TX_BUF_SIZE, tp->tx_bufs, tp->tx_bufs_dma);
                        tp->tx_bufs = NULL;
                }
        if(tp->rx_ring) {
                        pci_free_consistent(tp->pci_dev, RX_BUF_TOT_LEN, tp->rx_ring, tp->rx_ring_dma);
                        tp->rx_ring = NULL;
                }
        return -ENOMEM;
}
```

Table 14: Extending rtl8139_open function

The code in Table 14 calculates the memory required for ring buffer. The calculation of RX_BUF_TOT_LEN depends upon some device configuration parameters. As we see shortly in rtl8139_hw_start, we configure Bits 12-11 of RCR register as 10, which configures a 32K+16 receiver buffer length. Therefore, we allocate that much memory for the receiver buffer. Also, we configure bits 7 to 1, which means RTL8139 will keep moving the rest of the packet data into the memory, immediately after the end of Rx buffer. Therefore, we allocate 2048 bytes of buffer extra to cope up with such situations.

Now that we've looked into function rtl8139_open, we look into rtl8139_hw_start, where we configure the device for receiving packets.

```
static void rtl8139_hw_start (struct net_device *dev)
{
        struct rtl8139_private *tp = dev->priv;
        void *ioaddr = tp->mmio_addr;
        u32 i;

        rtl8139_chip_reset(ioaddr);

        /* Must enable Tx/Rx before setting transfer thresholds! */
        writeb(CmdTxEnb | CmdRxEnb, ioaddr + CR);

        /* tx config */
        writel(0x00000600, ioaddr + TCR); /* DMA burst size 1024 */

        /* rx config */
        writel(((1 << 12) | (7 << 8) | (1 << 7) |
                                (1 << 3) | (1 << 2) | (1 << 1)), ioaddr + RCR);

        /* init Tx buffer DMA addresses */
        for (i = 0; i < NUM_TX_DESC; i++) {
               writel(tp->tx_bufs_dma + (tp->tx_buf[i] - tp->tx_bufs),
                                      ioaddr + TSAD0 + (i * 4));
        }

        /* init RBSTART */
        writel(tp->rx_ring_dma, ioaddr + RBSTART);

        /* initialize missed packet counter */
        writel(0, ioaddr + MPC);

        /* no early-rx interrupts */
        writew((readw(ioaddr + MULINT) & 0xF000), ioaddr + MULINT);

        /* Enable all known interrupts by setting the interrupt mask. */
        writew(INT_MASK, ioaddr + IMR);

        netif_start_queue (dev);
        return;
}
```

Table 15: Extending rtl8139_hw_start function

As shown in Table 15, the first change in rtl8139_hw_start function is that we are writing CmdTxEnb | CmdRxEnb to CR register, which means the device will be transmitting as well as receiving packets. The next change is device receive configuration. I have not used macros in code, but they are quite obvious, if you see the rtl8139 specification. The bits used in this statement are as follows:

- Bit 1 - Accept physical match packets
- Bit 2 - Accept multicast packets
- Bit 3 - Accept broadcast packets
- Bit 7 - WRAP. When set to 1, RTL8139 will keep moving the rest of packet data into the memory immediately after the end of Rx buffer.
- Bit 8-10 - Max DMA burst size per Rx DMA burst. We are configuring this to 111, which means unlimited.
- Bit 11-12 - Rx buffer length. We are configuring to 10 which means 32K+16 bytes.

The next major change is configuring RBSTART register. This register contains starting address of receive buffer. Later, we initialize MPC (Missed Packet Counter) register to zero and configure the device for not generating early interrupts.

The last major function we want to discuss is the device interrupt handler. This interrupt handler is responsible for receiving packets, as well as for updating necessary statistics. Here is the source code for an interrupt handler.

```
static void rtl8139_interrupt (int irq, void *dev_instance, struct pt_regs *regs)
{
        struct net_device *dev = (struct net_device*)dev_instance;
        struct rtl8139_private *tp = dev->priv;
        void *ioaddr = tp->mmio_addr;
        unsigned short isr = readw(ioaddr + ISR);

        /* clear all interrupt.
         * Specs says reading ISR clears all interrupts and writing
         * has no effect. But this does not seem to be case. I keep on
         * getting interrupt unless I forcibly clears all interrupt :-(
         */
        writew(0xffff, ioaddr + ISR);

        if((isr & TxOK) || (isr & TxErr))
        {
               while((tp->dirty_tx != tp->cur_tx) || netif_queue_stopped(dev))
               {
                       unsigned int txstatus =
                               readl(ioaddr + TSD0 + tp->dirty_tx * sizeof(int));

                       if(!(txstatus & (TxStatOK | TxAborted | TxUnderrun)))
                               break; /* yet not transmitted */

                       if(txstatus & TxStatOK) {
                               LOG_MSG("Transmit OK interrupt\n");
                               tp->stats.tx_bytes += (txstatus & 0x1fff);
                               tp->stats.tx_packets++;
                       }
                       else {
                               LOG_MSG("Transmit Error interrupt\n");
                               tp->stats.tx_errors++;
                       }

                       tp->dirty_tx++;
                       tp->dirty_tx = tp->dirty_tx % NUM_TX_DESC;

                       if((tp->dirty_tx == tp->cur_tx) & netif_queue_stopped(dev))
                       {
                               LOG_MSG("waking up queue\n");
                               netif_wake_queue(dev);
                       }
               }
        }

        if(isr & RxErr) {
               /* TODO: Need detailed analysis of error status */
               LOG_MSG("receive err interrupt\n");
               tp->stats.rx_errors++;
        }

        if(isr & RxOK) {
               LOG_MSG("receive interrupt received\n");
               while((readb(ioaddr + CR) & RxBufEmpty) == 0)
               {
                       unsigned int rx_status;
                       unsigned short rx_size;
                       unsigned short pkt_size;
                       struct sk_buff *skb;

                       if(tp->cur_rx > RX_BUF_LEN)
                               tp->cur_rx = tp->cur_rx % RX_BUF_LEN;

                       /* TODO: need to convert rx_status from little to host endian
                        * XXX: My CPU is little endian only :-)
                        */
                       rx_status = *(unsigned int*)(tp->rx_ring + tp->cur_rx);
                       rx_size = rx_status >> 16;

                       /* first two bytes are receive status register
                        * and next two bytes are frame length
                        */
                       pkt_size = rx_size - 4;

                       /* hand over packet to system */
                       skb = dev_alloc_skb (pkt_size + 2);
```

```
                        if (skb) {
                                skb->dev = dev;
                                skb_reserve (skb, 2); /* 16 byte align the IP fields */

                                eth_copy_and_sum(
                                       skb, tp->rx_ring + tp->cur_rx + 4, pkt_size, 0);

                                skb_put (skb, pkt_size);
                                skb->protocol = eth_type_trans (skb, dev);
                                netif_rx (skb);

                                dev->last_rx = jiffies;
                                tp->stats.rx_bytes += pkt_size;
                                tp->stats.rx_packets++;
                        }
                        else {
                                LOG_MSG("Memory squeeze, dropping packet.\n");
                                tp->stats.rx_dropped++;
                        }

                        /* update tp->cur_rx to next writing location  * /
                        tp->cur_rx = (tp->cur_rx + rx_size + 4 + 3) & ~3;

                        /* update CAPR */
                        writew(tp->cur_rx, ioaddr + CAPR);
                }
        }

        if(isr & CableLen)
                LOG_MSG("cable length change interrupt\n");
        if(isr & TimeOut)
                LOG_MSG("time interrupt\n");
        if(isr & SysErr)
                LOG_MSG("system err interrupt\n");
        return;
}
```

Table 16: Interrupt Handler

As shown in Table 16, the ISR register is read in variable isr. Any further demultiplexing of interrupts is the interrupt handler's job. If we receive TxOK, TxErr, or RxErr, we update necessary statistics. Receiving an RxOK interrupt means we have received a frame successfully, and the driver has to process it. We read from the receiver buffer until we have read all data. (loop while ((readb (ioaddr + CR) & RxBufEmpty) == 0) does this job.) First, we check if tp->cur_rx has gone beyond RX_BUF_LEN. If that is case, we wrap it. The received frame contains 4 extra bytes at the start of frame (appended by RTL8139), apart from packet contents and other headers. The first two bytes indicate frame status and next two bytes indicate frame length. (The length includes first 4 bytes, also.) These values are always in little-endian order, and must be converted to host order. Then, we allocate a skb for received packet, copy the frame contents into skb, and queue the skb for later processing. Then, we update CAPR (Current Address of Packet Read), to let the RTL8139 know about the next write location. Note that we have already registered this interrupt handler in function rtl8139_open. So far, we had a dummy definition; now, we can replace that with this definition.

The last function we want to add is rtl8139_get_stats, which simply returns tp->stats.

```
static struct net_device_stats* rtl8139_get_stats(struct net_device *dev)
{
  struct rtl8139_private *tp = dev->priv;
  return &(tp->stats);
}
```

Table 17: rtl8139_get_stats function

This ends our driver development. Compile and insert this again (you must unload earlier module using rmmod), and ping to another host. You should be able to receive ping replies.

Although a professional-grade driver includes many more features than described in this driver, the latter gives you a good insight into network drivers and will help you understanding production drivers.

Talkback: Discuss this article with The Answer Gang

Mohan Lal Jangir is working as Development Lead at Samsung India Software Operations, Bangalore, INDIA. He has a Masters in Computer Technology from IIT DELHI, and is keen interested in Linux, Networking and Network Security.

Published in Issue 156 of Linux Gazette, November 2008

<-- prev | next -->