<< Prev | TOC | Front Page | Talkback | FAQ | Next >>

LINUX GAZETTE
...making Linux just a little more fun!

Writing a Network device driver - Part 1

By Bhaskaran

Introduction

This article will help the reader to understand and develop a network driver for an ethernet card in Linux. As a note, the driver development was done in C and as a module, so I assume its readers to be significantly exposed to C and l inux environment. The document intends only to show some essential points in building a driver for a network card. (For better and professional ones please refer to linux source listing).

Linux Networking and PCI cards

It is apparent that support for networking is inherent to the Linux kernel. One could also see Linux as one of the most 'safest and secure' Networking Operating system presently available in the market. Internally Linux kernel implements the **TCP/IP** protocol stack . It is possible to divide the networking code into parts - one which implements the actual protocols (the /usr/linux /net/ipv4 directory) and the other which implements device driver various network hardware.(/usr/src/linux/drivers/net).

The kernel code for **TCP/IP** is written in such a way that it is very simple to "slide in" drivers for many kind of real (or virtual) communication channels without bothering too much about the functioning of the network and transport layer code. It just requires a module in a standard manner, connecting the card hardware to actual software interface. The hardware part consists of an Ethernet card in case of LAN or a modem in internet.

Now a days a lot of Networking cards are available in the market, one of them is RTL8139 PCI ethernet card. RTL8139 cards are plug and play kind of devices, connected to the cpu through PCI bus scheme. PCI stands for Peripheral Component Interconnect, it's a complete set of specifications defining how different parts of computer interact with others. PCI architecture was designed as a replacement to earlier ISA standards because of its promising features like speed of data transfer, independent nature, simplification in adding and removing a device etc.

Networking Basics

One could set his/her PC for networking through **netconfig** command. It

configures the communication address (IP address given as four octets), netmask, gateway, primary nameserver etc through a self automated process. Once succeeded, the Linux box listens to messages to the assigned IP address.

Another important way is by manually detecting and configuring a network card, for which **ifconfig** command is used. A typical output of ifconfig command without any arguments is shown below (it could vary system to system depending upon the configuration).

```
eth0      Link encap:Ethernet  HWaddr 00:80:48:12:FE:B2
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:10 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:0 (0.0 b)  TX bytes:600 (600.0 b)
          Interrupt:11 Base address:0x7000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:4 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:336 (336.0 b)  TX bytes:336 (336.0 b)
```

It shows that I have a running interface for eth0 and lo, which corresponds to ethernet card and loopback interface respectively. The loopback is completely software based and used as an dummy interface to the network. The eth0 is the default name given to real hardware interface for realtek 8139 network card. The listing also tells about its hardware (HWaddr), internet (inet addr), Broadcast(Bcast), Mask (Mask) addresses with some other statistical information on data transfer that include Maximum data unit that can be transferred (MTU), no. of received (RX) packets, no. of transmitted packets (TX), collisions etc. The ifconfig command can also be used to bring up the interface if it is not detected at boot time. This could also be associated with an IP address as given below.

```
ifconfig eht0 192.9.200.1 up
```

This brings up the ethernet card to listen to an IP address 192.9.200.1, a class-C client. At the same time **ifoconfig** can also be used to bring down an activated interface. This is as given below.

```
ifconfig eth0 down
```

The same is applicable to loopback interface. That is these are quite possible.

```
ifconfig lo 192.9.200.1 up
ifconfig lo down
```

'ifconfig' supports plenty of options that may be discovered through reference to man pages.

Another command that needs reference is ***netstat***, It prints out network connections, routing tables, interface statistics, masquerade connections, and multicast memberships. An exhaustive list of options may be found in man pages.

Kernel Interface

Kernel as usual provides concise but efficient data structures and functions to perform elegant programming, even understandable to a moderate programmer, and the interface provided is completely independent of higher protocol suit. For an quick overview of the kernel data structures, functions, the interactions between driver and upper layer of protocol stack, we first attempt to develop a hardware independent driver. Once we get a big picture we can dig into the real platform.

Whenever a module is loaded into kernel memory, it requests the resources needed for its functioning like I/O ports, IRQ etc. Similarly when a network driver registers itself; it inserts a data structure for each newly detected interface into a global list of network devices.

Each interface is defined by a ***struct net_device*** item. The declaration of device **rtl8139** could done as follows

```
struct net_device rtl8139 = {init: rtl8139_init};
```

The ***struct net_device*** structure is defined in include file **linux/net_device.h** .The code above initializes only a single field 'init' that carries the initialization functions. Whenever we register a device the kernel calls this init function, which initializes the hardware and fills up ***struct net_device*** item. The ***struct net_device*** is huge and handles all the functions related to operations of the hardware. Let us look upon some revelent ones.

name : The first field that needs explanation is the 'name' field, which holds the name of the interface (the string identifying the interface). Obviously it is the string "rtl8139" in our case.

int (*open) (struct net_device *dev) : This method opens the interface whenever ifconfig activates it. The open method should register any system resource it needs.

int (*stop) (struct net_device *dev) : This method closes or stops the interface (like when brought down by ifconfig).

int (*hard_start_xmit) (struct sk_buff *skb, struct net_device *dev) : This

method initiates the transmission through the device 'dev'. The data is contained in the socket buffer structure skb. The structure skb is defined later.

struct net_device * (*get_status) (struct net_device *dev): Whenever a application needs to get statistics for the interface, this method is called. This happens, for example, when ***ifconfig*** or ***netstat -i*** is run.

void *priv :The driver writer owns this pointer and can use it at will. The utility of this member will be persuaded at a later stage. There exist a lot more methods to be explained but before that let us look at some working code demonstration of a dummy driver built upon the discussion above. This code would make the interactions between these elements crystal clear.

Code Listing 1

```
#define MODULE
#define __KERNEL__

#include < linux/module.h >
#include < linux/config.h >

#include < linux/netdevice.h >

int rtl8139_open (struct net_device *dev)
{
        printk("rtl8139_open called\n");
        netif_start_queue (dev);
        return 0;
}

int rtl8139_release (struct net_device *dev)
{
        printk ("rtl8139_release called\n");
        netif_stop_queue(dev);
        return 0;
}

static int rtl8139_xmit (struct sk_buff *skb,
                                 struct net_device *dev)
{
        printk ("dummy xmit function called....\n");
        dev_kfree_skb(skb);
        return 0;
}

int rtl8139_init (struct net_device *dev)
{
        dev->open = rtl8139_open;
        dev->stop = rtl8139_release;
        dev->hard_start_xmit = rtl8139_xmit;
        printk ("8139 device initialized\n");
        return 0;
}

struct net_device rtl8139 = {init: rtl8139_init};
```

```
int rtl8139_init_module (void)
{
        int result;

        strcpy (rtl8139.name, "rtl8139");
        if ((result = register_netdev (&rtl8139))) {
                printk ("rtl8139: Error %d  initializing card rtl8139 card",result);
                return result;
        }
return 0;
}

void rtl8139_cleanup (void)
{
        printk ("<0> Cleaning Up the Module\n");
        unregister_netdev (&rtl8139);
        return;
}

module_init (rtl8139_init_module);
module_exit (rtl8139_cleanup);
```

This typical module defines its entry point at ***rtl8139_init_module*** function. The method defines a net_device, names it to be "rtl8139" and register this device into kernel. Another important function ***rtl8139_init*** inserts the dummy functions ***rtl8139_open, rtl8139_stop, rtl8139_xmit*** to net_device structure. Although dummy functions, they perform a little task, whenever the rtl8139 interface is activated. When the ***rtl8139_open*** is called - then this routine announces the readiness of the driver to accept data by calling ***netif_start_queue***. Similarly it gets stopped by calling ***netif_stop_queue.***

Let us compile the above program and play with it. A command line invocation of 'cc' like below is sufficient to compile our file rtl8139.c

```
[root@localhost modules]# cc -I/usr/src/linux-2.4/include/ -Wall -c rtl8139.c
```

Let us check our dummy network driver. The following output was obtained on my system. We can use **lsmod** for checking the existing loaded modules. A output of lsmod is also shown.

(NB: You should be a super user in order to insert or delete a module.)

```
[root@localhost modules]# insmod rtl8139.o
Warning: loading test.o will taint the kernel: no license
  See http://www.tux.org/lkml/#export-tainted for information about tainted modules
Module test loaded, with warnings

[root@localhost modules]# lsmod
Module                  Size  Used by    Tainted: P
rtl8139                 2336   0  (unused)
mousedev                5492   1  (autoclean)
input                   5856   0  (autoclean) [mousedev]
i810                   67300   6
agpgart                47776   7  (autoclean)
```

```
autofs                  13268   0  (autoclean) (unused)

[root@localhost modules]# ifconfig rtl8139 192.9.200.1 up
[root@localhost modules]# ifconfig

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:4 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:336 (336.0 b)  TX bytes:336 (336.0 b)

rtl8139   Link encap:AMPR NET/ROM  HWaddr
          inet addr:192.9.200.1  Mask:255.255.255.0
          UP RUNNING  MTU:0  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:10 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 b)  TX bytes:600 (600.0 b)
```

Now You have been acquainted with writing a dummy driver, Let us move on to a real driver interface for rtl8139.

PCI card and their initialization

Though Network interface has been built up, but still it is not possible for us to probe and initialize the card. This is only possible until we check for a PCI interface and a PCI device available. Thus it becomes necessary that we have a close look upon the PCI and PCI functions available.

As I have described earlier that the PCI hardware is a complete protocol that determines the way each components interaction with the other. Each PCI device is identified by a *bus* number, a *device* number and a *function* number. The PCI specification permits a system to hold upon 256 buses, with each buses having a capacity to hold 32 multiboard devices.

The PC firmware initializes PCI hardware at system boot, mapping each devices I/O region to a different address, which is accessible from PCI configuration space, which consist of 256 bytes for each device. Three of the PCI registers identify a device: **vendorID, deviceID, class**. Sometimes **Subsystem vendorID and Subsystem deviceID** are also used. Let us see them in detail.

- The vendorID is 16 bit register that identifies a hardware manufacture. For example every Intel device has a vendor ID 0x8086.
- The deviceID is another 16-bit register, selected by the manufacturer. This ID is paired with the vendor ID to uniquely identify the device.
- Every peripheral device belongs to a class. The class register is 16-bit value whose most significant byte defines the group (of devices). e.g. ethernet belongs to network class.
- Subsystem vendorID and Subsystem deviceID are fields that can be used for

further identification of a device.

A complete list of PCI devices on ones linux box could be seen through command ***lspci***.

Based on the above information we can perform the detection of the rtl8139 could done in the rtl8139_init function itself, a modified version will look like

Code Listing 2

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

static int rtl8139_probe (struct net_device *dev, struct pci_dev *pdev)
{
        int ret;
        unsigned char pci_rev;

        if (! pci_present ()) {
                printk ("No pci device present\n");
                return -ENODEV;
        }
        else  printk ("<0> pci device were found\n");

        pdev = pci_find_device (PCI_VENDOR_ID_REALTEK,
                        PCI_DEVICE_ID_REALTEK_8139, pdev);

        if (pdev)  printk ("probed for rtl 8139 \n");
        else       printk ("Rtl8193 card not present\n");

        pci_read_config_byte (pdev, PCI_REVISION_ID, &pci_rev);

        if (ret = pci_enable_device (pdev)) {
                printk ("Error enabling the device\n");
                return ret;
        }

        if (pdev->irq < 2) {
                printk ("Invalid irq number\n");
                ret = -EIO;
        }
        else {
                printk ("Irq Obtained is %d",pdev->irq);
                dev->irq = pdev->irq;
        }
        return 0;
}

int rtl8139_init (struct net_device *dev)
{
        int ret;
        struct pci_dev *pdev = NULL;

        if ((ret = rtl8139_probe (dev, pdev)) != 0)
                return ret;

        dev->open = rtl8139_open;
```

```
        dev->stop = rtl8139_release;
        dev->hard_start_xmit = rtl8139_xmit;
        printk ("My device initialized\n");
        return 0;
}
```

As you can see a probe funtion is called through ***rtl8139_init*** function. A detailed analysis of the probe functions shows that it has been passed pointers of kind ***struct net_device*** and ***struct pci_dev***. The ***struct pci_dev*** holds the pci interface and other holds the network interface respectively, which has been mentioned earlier.

The function ***pci_present*** checks for a valid pci support available. It returns a value '0' on Success. Thereafter a probe of RTL8139 is initiated through the ***pci_find_device*** function. It accepts the vendor_ID, device_ID and the 'pdev' structure as argument. On an error-free return i.e. when RTL8139 is present, it sends the pdev structure filled. The constants PCI_VENDOR_ID_REALTEK, PCI_DEVICE_ID_REALTEK_8139 defines the vendorID and device_ID of the realtek card. These are defined in linux/pci.h.

pci_read_config_byte/word/dword are functions read byte/word/dword memory locations from the configuration space respectively. A call to ***pci_enable*** function to enable pci device for rtl8139, which also helps in registering its interrupt number to the interface. Hence if everything goes safe and error-free, your rtl_8139 has been detected and assigned an interrupt number.

In the next section we would see how to detect the hardware address of rtl8139 and start communication.

Author has just completed B.Tech from Govt. Engg. College Thrissur.

Published in Issue 93 of *Linux Gazette*, August 2003
