Embedded Systems

Welcome- Anyone interested in things Embedded- such as PDAs, phones, MP3-players, DVD-players, digital cameras, routers, firewalls, servers, watches, computers, and of course Embedded Linux! We're here to promote the Embedded Linux Industry by providing a place to meet and discuss ideas about Embedded Linux...

Wednesday, March 3, 2010

How to write a Network Driver in Linux

Linux Network Interface Driver

This article explains how to write a network interface driver in Linux . There are many network

interface cards available in market. I have taken the Realtek chip driver 8139too.c to explain code snippets. This driver is implemented in linux/drivers/net/8139too.c file. I have chosen 8139too.c

driver because hardware specifications for Realtek chips are available free and you can download or read them online .

See References section for download links to Realtek Manuals RTL8139D_DataSheet.pdf , RTL8139_ProgrammersGuide.pdf. I suggest you, first read device manuals and then read this article for better understanding. RTL8139_ProgrammersGuide.pdf explains how the reception and transmission happen in Realtek 8139 chip where as RTL8139D_DataSheet.pdf explains register's details. This article explains the driver from linux 2..6.26 kernel.

I assume that reader is familiar with Linux kernel and PCI devices. Though this article explains

8139too.c driver , writing any other network interface driver is similar except for the hardware specific functionalities, which change.

This article is divided into 5 parts and I wish to present each part every week to facilitate short, quick and informative reading.

1) Overview

FEEDJIT Live Traffic Map

Live Traffic Map

2) Initialization

3) Packet handling (reception and transmission)

4) Status and Control

5) Uninitialization

Overview

In Linux, network drivers have different properties than other drivers like char drivers and block drivers. Char and block drivers have major and minor number concept . VFS identifies these drivers using their major and minor numbers and these drivers have files created in /dev directory with their major and minor numbers . But network drivers do not have any major or minor number concept and no files created in /dev directory. Network drivers are identified in the kernel with its interface descriptor block ,struct net_device (defined in linux/include/linux /netdevice.h). net_device objects of all network drivers are put into a global liked list and accessed by the kernel whenever it needs. An application cannot access the driver directly and it should go through the system calls like socket .

As you know, any device can be interfaced with CPU using any bus like PCI , USB, Firewire etc .

Network interface card also can be interfaced with CPU using any of the above said buses . Network interface card can be inbuilt into mother board or inserted into any bus slot. In this article we assume that network interface card is a PCI device. When a packet arrives, network interface card sees the destination MAC address of the packet and puts the packet in input data buffer(RxRing) if the destination MAC address matches with its MAC address , raises an interrupt and continue receiving the packets. If Driver wants to send a packet it puts the packet in output buffer(TxBuffer) . Network interface card takes the packet from TxBuffer and puts that in its controller's FIFO buffer and try to send the packet. Once the packet is sent network interface card raises an interrupt and writes status of the interrupt in its status registers. Any Network interface driver is responsible for activities in the following areas:

Initialization

Packet Reception

Packet Transmission

Status and Control

Uninitialization

Initialization

The initialization part of a network driver has the responsibility of initializing the driver and hardware. The following are the some of the responsibilities of initialization part of the driver.

Registering with the Linux low level bus interface subsystem

Allocating interface descriptor block (net_device) ,device specific structure and initializing media

specific fields

Getting device specific structure object pointer

Enabling Network interface card

Getting the Device resources (Memory mapped or port mapped I/O register map)

Getting device MAC address

Initialization of device methods in the net_device

Registering net_device object with the kernel

Registering the interrupt handler (ISR)

Allocating Rxring and Txring

Initializing the hardware (network interface card)

Start the network interface's transmit Queue

2.1) Registering with the Linux low level bus interface subsystem

Linux provides low level interface to access bus. For PCI devices PCI subsystem is provided. This

provides functions for accessing PCI devices. For USB devices USB subsystem is provided. The PCI subsystem in the kernel provides all the generic functions that are used in common by various PCI device drivers to access a PCI device. First step in initialization part of the driver should be registering with PCI subsystem. Once registration is over PCI subsystem will notify the driver whenever the device is found on the bus by calling probe function of the driver. The following code snippet shows registration with PCI subsystem. This code would be written in initialization routine of the driver module.

static struct pci_device_id rtl8139_pci_tbl[] = {

{0×10ec, 0×8139, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8139 },

{}, /*Null terminated entry*/

}

static struct pci_driver rtl8139_pci_driver = {

.name = DRV_NAME,

.id_table = rtl8139_pci_tbl,

Embedded Internet

How to write a Network Driver in Linux

Evolution of GNU/Linux system..must read for Linux...

► February (2)

► 2009 (2)

Categories

microcontroller (1)

Followers

Followers (1)

Follow

```
.probe = rtl8139_init_one,

.remove = __devexit_p(rtl8139_remove_one),

#ifdef CONFIG_PM

.suspend = rtl8139_suspend,

.resume = rtl8139_resume,

#endif /* CONFIG_PM */

};

static int __init rtl8139_init_module (void)

{

return pci_register_driver(&rtl8139_pci_driver);

}
```

To register with the PCI subsystem,the driver should create a struct pci_driver object , fill in the

object and call pci_register_driver function. pci_register_driver function takes one parameter , pointer to

an object of type struct pci_driver . The following are some of the fields of pci_driver:

‘name’ field is name of this driver,

‘probe’ field is a call back function which is called whenever your network controller is found on the PCI bus. It’s prototype is:

```
int (*probe) (struct pci_dev *dev, const struct pci_device_id *id);
```

‘id_table’ field is struct pci_device_id pointer which contains the device information

that you are writing driver for. struct pci_device_id has the following definition:

```
struct pci_device_id {

__u32 vendor, device; /* Vendor and device ID or PCI_ANY_ID*/

__u32 subvendor, subdevice; /* Subsystem ID’s or PCI_ANY_ID */

__u32 class, class_mask; /* (class,subclass,progif) triplet */

kernel_ulong_t driver_data; /* Data private to the driver */

};
```

‘vendor’ field is vendor of the any PCI device(in this case network interface card), ‘device ‘ field is product or device id of the device , subvendor is subvendor of the

device , ‘class’ is class of the device (see PCI specification for different types classes), driver_data is a private field that can be used by driver.

‘remove’ is a call back function which is called whenever network interface card is removed or

when the driver module is unloaded. Prototype of this function is:

void (*remove) (struct pci_dev *dev)

’suspend ‘ is a call back function which is called whenever device is to be suspended and should not process any more packets. Prototype of this function is:

int (*suspend) (struct pci_dev *dev, pm_message_t state);

‘resume’ is a call back function which is called by the PCI layer whenever suspended device wants to be woken up. Prototype of this function is:

int (*resume) (struct pci_dev *dev);

Once the driver creates pci_driver object it is registered with the PCI subsystem using

pci_register_driver function. Whenever our device (network interface card) is found on the bus, PCI subsystem calls driver’s ‘probe’ function. probe function receives two parameters. One is pointer to

struct pci_dev object and second is pointer to struct pci_device_id object . Each PCI device is an object of type struct pci_dev . The following is the 8139too.c probe function.

static int __devinit rtl8139_init_one (struct pci_dev *pdev, onst struct pci_device_id *ent)

c

{

struct net_device *dev = NULL;

struct rtl8139_private *tp;

}

2.2) Allocating interface descriptor block (net_device) ,device specific structure and

initializing media specific fields

Each interface is an object of type struct net_device , called interface descriptor block. Interface

can represent a specific media like Ethernet,Token ring and fddi etc. Interface descriptor block is a

combination of device specific fields and media specific fields . the following

function allocates

interface descriptor block struct net_device.

dev = alloc_netdev(sizeof(*tp),"eth%d",ether_setup);

This function creates struct net_device object ,allocates space for device specific structure

s name to this interface as 'eth%d' (%d

struct rtl8139_private (first argument) and assign

specifies eth0 for first interface ,eth1 for second interface etc)(second argument).

struct rtl8139_private is a device specific structure .There will be an object of this structure

for each interface found on the network interface card. Each driver will define its own device specific

structure. struct rtl8139_private definition can be found in 8139too.c file.

Realtek 8139 driver is a Ethernet driver. So we need to initialize Ethernet media specific fields of

n to

struct net_device object . alloc_netdev function calls ether_setup (third argument) functio

initialize the Ethernet media specific fields.

ether_setup is a kernel function which initialize some of the ethernet specific fields . tr_setup function

can be used to initialize token ring device fields. fddi_setup function can be used for fddi devices.

These all three tasks , allocating net_device , allocating space for device specific structure and initializing

media specific fields can also be performed by calling alloc_etherdev media specific function. The

following code shows that.

dev = alloc_etherdev(sizeof(*tp));

This allocates net_device object ,gives its name as eth%d and initialize ethernet media specific

fields.

For token ring devices you have alloc_trdev and for fddi devices you have

alloc_fddidev functions .

If you want more about struct net_device you can refer Understanding Linux network internals

book.

2.3 Getting device specific structure object pointer

Memory allocated for the device specific object will be pointed by the ‘priv’ field of the net_device

object. This can be stored into a local variable using netdev_priv function. Direct accessing of ‘priv’

field is discouraged.

tp = netdev_priv(dev);

This will store the pointer to device specific structure into ‘tp’ local pointer variable.

2.4 Enabling Network Interface Card

At the initialization time , network interface card will be in idle state . we need to enable the network

interface card by setting enable bit in the command register of the device configuration space. This can

be done by calling pci_enable_device function.

rc = pci_enable_device(pdev);

if(rc)

goto err_out;

2.5 Getting the Device resources (Memory mapped or Port mapped register map)

To talk to the network interface card , network controller provides some registers. Driver has to map

these registers into processor address space so that read/write operations by the driverwill be made on

system memory addresses directly. PCI devices can provide two types of memory mapping, one is

memory mapped I/O and second is port mapped I/ . O

Details of each register will be explained in the device specification. At the time of device enumeration,

base address of the registers, some flags and total size of these registers are stored in ‘resources’ field of

the struct pci_dev object. Network interface driver needs to get these details and store them in

device specific structure to be used later. The following code snippet shows this:

pio_start = pci_resource_start (pdev, 0);

pio_end = pci_resource_end (pdev, 0);

pio_flags = pci_resource_flags (pdev, 0);

pio_len = pci_resource_len (pdev, 0);

mmio_start = pci_resource_start (pdev, 1);

mmio_end = pci_resource_end (pdev, 1);

mmio_flags = pci_resource_flags (pdev, 1);

mmio_len = pci_resource_len (pdev, 1);

Each PCI device can support upto six base addresses like base address 0,base address 1 etc . One for

memory mapped ,one for port mapped and etc .For each base address an object of type struct

resource is created and filled with base address ,size and flags . Driver has to read each struct

resource object and get the base address and size and flags..

To get the base address of a device you can use pci_resource_start macro. This macro takes two

parameters , one is pci_dev object of the device that you want base address and another is base address

register number. Base address register number can be one of 05 numbers.

To get the end of base address you can use pci_resource_end macro,to get size of the registers you can

use pci_resource_len macro.

To get flags of the device you can use pci_resource_flags macro. Flags can be any of the following.

IORESOURCE_IO /* Resource type */

IORESOURCE_MEM

IORESOURCE_IRQ

IORESOURCE_DMA

IORESOURCE_PREFETCH

IORESOURCE_READONLY

IORESOURCE_CACHEABLE

IORESOURCE_RANGELENGTH

IORESOURCE_SHADOWABLE

IORESOURCE_SIZEALIGN /* size indicates alignment */

IORESOURCE_STARTAL IGN /* start field is alignment */

IORESOURCE_DISABLED

IORESOURCE_UNSET

IORESOURCE_AUTO

IORESOURCE_BUSY /* Driver has marked this resource busy */

As a driver developer you need to know only IORESOURCE_IO, IORESOURCE_MEM and

IORESOURCE_BUSY flags. IORESOURCE_IO flags tel s that this base address is port mapped ,

I

IORESOURCE_MEM tells that this base address is memory mapped and IORESOURCE_BUSY tells

that this resource is reserved by this driver. If resources are reserved, another driver cannot use these

resources.

Once you got the device resources you need to check them for what type of mapping they are. The

following code shows that :

/* make sure PCI base addr 0 is PIO */

if (!(pio_flags & IORESOURCE_IO)) {

dev_err(&pdev>dev, “region #0 not a PIO resource, aborting\n”);

rc = ENODEV;

goto err_out;

}

/* make sure PCI base addr 1 is MMIO */

if (!(mmio_flags & IORESOURCE_MEM)) {

dev_err(&pdev>dev, “region #1 not an MMIO resource, aborting\n”);

rc = ENODEV;

goto err_out;

}

After checking the resources you have to reserve the resources by calling pci_request_regions function.

This function reserves the resources and returns error if they are already reserved by another driver.

rc = pci_request_regions (pdev, DRV_NAME);

if (rc)

goto err_out;

Some PCI devices have the capability of bus mastering . We need to enable it by calling pci_set_master

function. This function sets the bus mastering bit of the command register in the device configuration

space.

/* enable PCI busmastering */

pci_set_master(pdev);

Device resources have to be remapped into the kernel address space so that page tables will be created

for the registers. For the Port mapped I/O this is done using ioport_map and for memory mapped I/O

this is done using pci_iomap function. After remapping you need to store base address in the

‘base_addr’ field of net_device object and also store base address and resource length in the device

specific object to be used later.

#ifdef USE_IO_OPS

ioaddr = ioport_map(pio_start, pio_len);

if (!ioaddr) {

dev_err(&pdev>dev, “cannot map PIO, aborting\n”);

```
rc = EIO;

goto err_out;

}

dev>base_addr = pio_start;

tp>mmio_addr = ioaddr;

tp>regs_len = pio_len;

#else

/* ioremap MMIO region */

ioaddr = pci_iomap(pdev, 1, 0);

if (ioaddr == NULL) {

dev_err(&pdev>dev, “cannot remap MMIO, aborting\n”);

rc = EIO;

goto err_out;

}

dev>base_addr = (long) ioaddr;

tp>mmio_addr = ioaddr;

tp>regs_len = mmio_len;

#endif /* USE_IO_OPS */
```

Once you got resources you need to reset the controller chip. This can be done by setting reset bit of the

command register of the controller.(see the device specification)

```
/* Soft reset the chip. */

RTL_W8 (ChipCmd, CmdReset);

/* Check that the chip has finished the reset. */

for (i = 1000; i > 0; i) {

barrier();

if ((RTL_R8 (ChipCmd) & CmdReset) == 0)

break;

udelay (10);
```

}

RTL_W8 macro definition is:

#define RTL_W8(reg, val8) iowrite8 ((val8), ioaddr + (reg));

2.6 Getting device MAC address

Driver has to read the MAC address stored in the ROM of the network interface card . To access ROM

of the network interface card , some of the registers of the memory mapped or port mapped I/O registers

are used. This details can be found in the device specification.8139too driver calls read_eeprom local

function to read the MAC address from the ROM of the controller. This MAC address is stored in the

‘dev_addr’ and ‘perm_addr’ fields of net_device object.

addr_len = read_eeprom (ioaddr, 0, == 0×8129 ? 8 : 6;

for (i = 0; i <>

((__le16 *) (dev>dev_addr))[i] = cpu_to_le16(read_eeprom (ioaddr, i + 7, addr_len));

memcpy(dev>perm_addr, dev>dev_addr, dev>addr_len);

2.7 Filling device methods in the net_device

Driver has to provide it’s functionalities to the kernel through the device methods of struct

net_device . These are the operations that can be performed on the network interface. Some of the

function pointers of net_device structure are left blank and some are filled by ‘ether_setup’ function

called at the time of net_device object allocation.

Driver should fill some basic fundamenta l operations and can leave optional operations. Fundamental

methods are those that are needed to be able to use the interface; optional methods implement more

advanced functionalities that are not strictly required.

/* The Rtl8139specific entries in the device structure. */

dev>open = rtl8139_open;

dev>hard_start_xmit = rtl8139_start_xmit;

netif_napi_add(dev, &tp>napi, rtl8139_poll, 64);

dev>stop = rtl8139_close;

dev>get_stats = rtl8139_get_stats;

dev>set_multicast_list = rtl8139_set_rx_mode;

dev>do_ioctl = netdev_ioctl;

dev>ethtool_ops = &rtl8139_ethtool_ops;

dev>tx_timeout = rtl8139_tx_timeout;

dev>watchdog_timeo = TX_TIMEOUT;

#ifdef CONFIG_NET_POLL_CONTROLLER

dev>poll_controller = rtl8139_poll_controller;

#endif

‘open ‘ function opens the interface and is called whenever user configures the interface using any

utilities like ifconfig or ip . This function explained later in detail.

‘hard_start_xmit’ function is called whenever kernel wants to send a packet . This function is

explained later in detail.

‘rtl8139_poll ‘ is called whenever there is an incoming packet to be processed. This function is

explained later in detail.

’stop’ function stops the interface and is called whenever interface is brought down. This function

is explained later in detail.

‘get_stats’ is called whenever an application needs to know statistics of the interface . This

function is explained later in detail.

’set_multicast_list’ is an optional method and is called when the multicast list for the device

changes and when the flags change . We are not going to explain this function.

‘do_ioctl’ is interface specific function . This is an optional routine and we are not going to

explain.

‘ethtool_ops’ is a pointer to a structure of type struct ethtool_ops. This structure contains some

function pointers that are used by the ethtool tool. This we are not going to explain.

‘poll_controller’ is called whenever kernel wants to poll for the device status. This function

simply calls drivers interrupt routine to check if the device has anything to say.

After this some of the net_device and device specific object fields are filled .

dev>features |= NETIF_F_SG | NETIF_F_HW_CSUM | NETIF_F_HIGHDMA;

dev>irq = pdev>irq;

/* tp zeroed and aligned in alloc_etherdev */

tp>mmio_addr = ioaddr;

‘features’ field of net_device object tells the capabilities of the driver like it support scatter gather

I/O ,check sum can be calculated in hardware and can DMA to high memory etc.’irq’ field of net_device

object contains irq number of this controller.

2.8 Registering net_device object with kernel

Driver has to register with the kernel by giving net_device object . All net_device objects of all

interfaces are put into linked lists and accessed by the kernel whenever it needs. Network driver can

register with the kernel using register_netdev function.

i = register_netdev(dev);

register_netdev function takes a completed net_device object and adds it to the kernel interfaces.

0 is returned on success and a negative error no code is returned on a failure to set up the device, or if

the name is a duplicate.

struct net_device object is stored in struct pci_dev object so that it can be accessed later.

This is done with pci_set_drvdata function.

pci_set_drvdata(pdev , dev);

} /*end of rtl8139_init_one function */

/*start of open function */

static int rtl8139_open (struct net_device *dev)

{

open function is called by the kernel whenever this network interface is configured by admin using any

user space utilities like ifconfig or ip .When ifconfig is used to assign an address to the interface, it

performs two tasks. First, it assigns the address by means of ioctl(SIOCSIFADDR) (Socket I/O Control

Set Interface Address). Then it sets the IFF_UP bit in dev>flag by means of

ioctl(SIOCSIFFLAGS) (Socket I/O Control Set Interface Flags) to turn the interface on.

open function receives net_device object as its parameter. Driver should get the device specific

object which is stored in the ‘priv’ field of net_device object at the time of net_device object

allocation. This can be done by calling netdev_priv inline function.

struct rtl8139_private *tp = netdev_priv(dev);

/* get registers base address in a local variable */

void __iomem *ioaddr = tp>mmio_addr;

2.9 Registering Interrupt handler (ISR)

Whenever a packet is received or a packet is sent an interrupt is raised by the network controller. Driver

needs to register an interrupt handler(ISR) and this handler is called whenever controller raises an

interrupt. Driver can register the interrupt handler either in driver’s init routine or in the open function .

Driver registers with interrupt handler using request_irq routine.

retval = request_irq (dev>irq, rtl8139_interrupt, IRQF_SHARED, dev>name, dev);

if (retval)

return retval;

request_irq routine takes five parameters. First parameter is irq number of the device , second parameter

is interrupt handler (ISR) ,Third parameter is irqflags,fourth parameter is device name and last parameter

is dev_id. ‘dev_id’ feild can be of any object pointer ,is necessary if irqflags is IRQF_SHARED and

used at the time of freeing the ISR using free_irq routine. Generally this field will be a pointer to

net_device instance and is used at the time of interrupt handler execution. Interrupt handler has the

following prototype:

irqreturn_t (*irq_handler_t)(in t, void *)

and irqflags can be any of the following.

* IRQF_DISABLED keep irqs disabled when calling the action handler

* IRQF_SAMPLE_RANDOM irq is used to feed the random generator

* IRQF_SHARED allow sharing the irq among several devices

* IRQF_PROBE_SHARED set by callers when they expect sharing mismatches to

occur

* IRQF_TIMER Flag to mark this interrupt as timer interrupt

* IRQF_PERCPU Interrupt is per cpu

* IRQF_NOBALANCING lag to exclude this interrupt from irq balancing

F

* IRQF_IRQPOLL Interrupt is used for polling (only the interrupt that is registered first

in an shared interrupt is considered for performance reasons)

request_irq routine allocates interrupt resources and enables the interrupt line and IRQ handling. From

the point this call is made driver handler function may be invoked. Since driver handler function must

clear any interrupt the board raises, driver must take care both to initialize the hardware and to set up

the interrupt handler in the right order.

We will describe functionalities of interrupt handler(ISR) later in this article.

2.10 Allocating Rxring and Txbuffer

Whenever network controller receives a packet it puts that in a receive buffer called RxRing and raises an

interrupt and continue to receive next packet. Driver puts outgoing packets in Txbuffer and network

controller takes the packets from the Txbuffer , sends them out of the wire and raises interrupt. We will

describe this process later in detail. Driver allocates Rxring and Txbuffer using dma_alloc_coherent

routine. If Memory allocated for the buffers ,the by dma_free_coherent routine returns virtual address

of the buffers .

```
tp>tx_bufs = dma_alloc_coherent(&tp>pci_dev>dev,TX_BUF_TOT_LEN,

&tp>tx_bufs_dma, GFP_KERNEL);

tp>rx_ring = dma_alloc_coherent(&tp>pci_dev>dev, X_BUF_TOT_LEN, R

&tp>rx_ring_dma, GFP_KERNEL);

if (tp>tx_bufs == NULL || tp>rx_ring == NULL) {

free_irq(dev>irq, dev);

if (tp>tx_bufs)

dma_free_coherent(&tp>pci_dev>dev,TX_BUF_TOT_LEN,

tp>tx_bufs, tp>tx_bufs_dma);

if (tp>rx_ring)

dma_free_coherent(&tp>pci_dev>dev,RX_BUF_TOT_LEN,

tp>rx_ring, tp>rx_ring_dma);

return ENOMEM;

}
```

dma_alloc_coherent routine takes four parameters. First is generic device struct device object ,

second is length of the buffer ,third is an output param of type dma_addr_t which is filled with physical

address (bus address) of the allocated memory and fifth is GFP flag which tells how the memory should

be allocated..This function returns virtual address of the memory allocated.

2.11 Initialize the hardware (network interface card)

Network driver has to initialize network controller at the time of open function is called. Controller

initialization includes, resetting the chip , restoring MAC address in the chip register enabling ,

reception(rx) and transmission by setting rxenable and txenable bits in command register of the

controller,setting transfer thresholds , initializing transmission descriptors and reception descriptor of the

controller with the physical addresses of the Txbuffer and RxRing,setting what type of packets to be

received by setting in RxConfig and TxConfig registers, and enabling interrupts by setting Interrupt

Mask Register(IMR) .

1 Soft reset the chip

```
RTL_W8 (ChipCmd, CmdReset);

/* Check that the chip has finished the reset. */

for (i = 1000; i > 0; i) {

barrier();

if ((RTL_R8 (ChipCmd) & CmdReset) == 0)

break;

udelay (10);

}
```

2 Restore the MAC address

```
RTL_W32_F (MAC0 + 0, le32_to_cpu (*(__le32 *) (dev>dev_addr + 0)));

RTL_W32_F (MAC0 + 4, le16_to_cpu (*(__le16 *) (dev>dev_addr + 4)));
```

3 Enable transmission and reception

```
RTL_W8 (ChipCmd, CmdRxEnb | CmdTxEnb);
```

4 Set type of packets to be received

RTL_W32 (RxConfig, tp>rx_config);

RTL_W32 (TxConfig, rtl8139_tx_config);

5 initialize reception buffer descriptor(RxBuf) with RxRing DMA address(bus address).This s

i

where physical address used.

RTL_W32_F (RxBuf, tp>rx_ring_dma);

6 Initialize transmission descriptors with Txbuffer DMA address (bus address)

for (i = 0; i <>

RTL_W32_F (TxAddr0 + (i * 4), tp>tx_bufs_dma + (tp>tx_buf[i] tp>tx_bufs));

7 Enable all known interrupts by setting the interrupt mask register

RTL_W16 (IntrMask, rtl8139_intr_mask);

2.12 Start the network interface's transmit Queue

The open function should also start the interface's transmit queue (allowing it to accept packets for

transmission) once it is ready to start sending data. Driver should call kernel function netif_start_queue

to start queue.

netif_start_queue (dev);

netif_start_queue takes net_device object as its parameter and returns nothing. This function simply

sets a bit in 'state' field of net_device object that allows upper layers to call the device

hard_start_xmit function .

}/ * end of open function */

3 Packet handling

Packet handling is a task of performing transmission and reception of packets. Packet handling is most

important task of any network interface driver. in kernel discussions, transmission refers only to sending

frames outward, whereas reception refers to frames coming in.

Before going to see how transmission and reception happen we will see the role of

interrupts in network

drivers. Kernel can use two main techniques for exchanging data: pol ing and interrupts. There is also an

I

option of combination of these two techniques.

Polling is a technique where the kernel constantly keeps checking whether the device has anything to

say. It can do that by continually reading a memory register on the device, for instance, or returning to

check it when a timer expires.

Interrupts is another technique technique of exchanging data. Here the device driver, on behalf of the

kernel, instructs the device to generate a hardware interrupt when specific events occur. The kernel,

interrupted from its other activities, will then invoke a handler registered by the driver to take care of the

device's needs. Interrupts can be raised by the device when frame is received and when a frame is

transmitted .. If this is the reception of a frame, the handler queues the frame somewhere and notifies the

kernel about it and if it is transmission the handler updates its status.

The code that takes care of an input frame is split into two parts: first the driver

copies the frame into an

input queue accessible by the kernel, and then the kernel processes it (usually passing it to a handler

dedicated to the associated protocol such as IP). The first part is executed in interrupt context and second

part is executed in the bottom half ..Second part may interrupted by the first because interrupt context

has the higher priority than bottom half . More about bottom halves can be read in Understanding Linux

Kernel and Understanding Linux network internals books.

Multiple Frames also can be processed During an Interrupt. This approach is used by quite a few Linux

device drivers. When an interrupt is notified and the driver handler is executed, the latter keeps

downloading frames and queuing them to the kernel input queue, up to a maximum number of frames

TimerDriven Interrupts technique is an enhancement to the previous ones. Instead of having the device

asynchronously notify the driver about frame receptions, the driver instructs the device to generate an

interrupt at regular intervals. The handler will then check if any frames have arrived since the previous

interrupt, and handles all of them in one shot.

Combination of all above said techniques is also possible . A good combination would use the interrupt

technique under low load and switch to the timerdriven interrupt under high load.

Pros and cons of all above said and more detail description of above said techniques can be found in

Understanding Linux Network Internals book by Christian Benvenuti.

3.1 Packet Reception

When a packet arrives into the network interface card , network controller checks the destination MAC

address of the packet. if it matches with its MAC address or if it is broadcast address network interface

card copies the packet into receive buffer(Rxring) and raises an interrupt.

Receive buffer(Rxrinf) is a block of I/O memory allocated by the river . Network interface card will

d

have a receive buffer descriptor register and a receive buffer status register. Driver has to write Physical

address of the receive buffer allocated into the receive buffer descriptor register. This process is specific

to hardware and is provided in the device manual. Some devices provide some receive block

descriptors,a structure,that contains status of the each packet , buffer pointer and size of packets etc.

Driver has to allocate a buffer and physical address of this is put in the buffer pointer .

When a packet is received by the network interface card, it adds a packet header before the packet , puts

the packet in receive buffer(Rxring) and raises interrupt. This is specific to Realtek 8139 chips and see

device specification.

When the interrupt is raised , driver interrupt handler(ISR) gets called. Driver interrupt handler has to

check the interrupt status register(ISR) of the device what the interrupt is raised for and has to take

appropriate action. In the case of reception ,reception bit of interrupt status register is set. Now we will

see step by step what interrupt handler does:

3.1.1 Interrupt handler of the driver:

Interrupt handler is called when a packet is received or transmitted. Kernel will send two arguments to

this function . One is irq number and other is dev_id which is sent as last parameter of request_irq

function at the time of interrupt handler registration. Generally this is a pointer to net_device object.

As i said, driver has to check status register of the network interface card and should take action

accordingly. In the case of packet reception , driver should schedule a bottom half and return from

interrupt handler. The next step of processing of packet is done by the kernel in the bottom half. In the

bottom half, kernel calls driver’s poll function to do the later processing of packet.

static irqreturn_t rtl8139_interrupt (int irq, void *dev_id)

{

1 Get net_device object from the dev_id

struct net_device *dev = (struct net_device *) dev_id;

2 Get device specific structure object from the net_device. This is stored in the ‘priv’ field of

net_device. Use netdev_priv inline function to get pointer to that and store registers base

address into a local variable ‘ioaddr’ .

struct rtl8139_private *tp = netdev_priv(dev);

void __iomem *ioaddr = tp>mmio_addr;

3 Get interrupt status register (ISR) of the controller into a local variable

status = RTL_R16 (IntrStatus);

4 Irq numbers in PCI devices can be shared by many devices. So many devices might have been

registered interrupt handler on the same irq number . Driver has to confirm that interrupt has

been raised by its interface card . This can be done by checking interrupt status register for

any pending interrupts by the interface card.

/* shared irq? */

if (unlikely((status & rtl8139_intr_mask) == 0))

goto out;

5 Check if the device is present or not (hot pluggable) or if there is major problem.

if (unlikely(status == 0xFFFF))

goto out;

6 Driver has to acknowledge the interrupts by clearing appropriate bits in the interrupt status

register (ISR)

ackstat = status & ~(RxAckBits | TxErr);

if (ackstat)

RTL_W16 (IntrStatus, ackstat);

7 As i said , receive packets are processed by poll function vector in the bottom handler.

Network uses softirq NET_ RX_SOFTIRQ bottom half for input packets. Driver should

schedule the bottom half and finish the interrupt handler. netif_rx_schedule is used to enable

NET_RX_SOFTIRQ bottom half. This functions takes two parameters, one is net_device

object and other is an object of type struct napi_struct . This structure has the

following definition.

```
struct napi_struct {
/* The poll_list must only be managed by the entity which
* changes the state of the NAPI_STATE_SCHED bit. This means
* whoever atomically sets that bit can add this napi_struct
* to the percpu poll_list, and whoever clears that bit
* can remove from the list right before clearing the bit.
*/
struct list_head poll_list;
unsigned long state;
int weight;
int (*poll)(struct napi_struct *, int);
#ifdef CONFIG_NETPOLL
spinlock_t poll_lock;
int poll_owner;
struct net_device *dev;
struct list_head dev_list;
#endif
};
```

This object is filled at the time of function pointers assignment, using netif_napi_add

function.

The following code checks status of the interrupt and calls netif_rx_schedule to

enable the bottom half. netif_rx_schdule function first tests if poll needs to be scheduled

using netif_rx_schedule_prep, is scheduled only if network interface up and next calls

__netif_rx_schedule to schedule poll .

if (status & RxAckBits)

netif_rx_schedule(dev, &tp>napi);

8 return from the interrupt handler. (we will see transmission part of interrupt handler later in

Packet transmission section)

return IRQ_RETVAL(handled);

} /* end of interrupt handler routine */

3.1.2 NET_RX_SOFTIRQ softirq calls driver's poll method

Kernel calls poll method of the driver in the NET_RX_SOFTIRQ softirq bottom half to process the

input packet . This softirq was scheduled in interrupt handler routine.

poll method receives two parameters, one is pointer to struct napi_struct object and second is

'budget'. napi_struct object is the object which was created at the time of initialization of methods in

the net_device object .'budget' field is the maximum number of packets the kernel can accept at this

time or we can say that the 'budget' value is a maximum number of packets that the current CPU can

receive from all interfaces.

The following is the 8139too.c driver poll method. It checks for the interrupt status register and calls

rtl8139_rx local function to do the rest of the process.

static int rtl8139_poll(struct napi_struct *napi, int budget)

{

1 Get device specific structure object. 'napi' is a field of type stuct napi_struct in device

specific structure struct rtl8139_private. If we know the address of 'napi' field of

struct rtl8139_private we can find the starting address of the struct

rtl8139_private object using container_of function.

struct rtl8139_private *tp = container_of(napi, struct rtl8139_private, napi);

2 Get net_device object from the struct rtl8139_private object and store register

base address into a local variable.

struct net_device *dev = tp>dev;

void __iomem *ioaddr = tp>mmio_addr;

3 Check if it was the receive interrupt and call rtl8139_rx local function. rtl8139_rx returns

number of packets received. This function is explained below.

if (likely(RTL_R16(IntrStatus) & RxAckB its))

work_done += rtl8139_rx(dev, tp, budget);

4 If number of packets received is less than budget , reenable receive interrupts by setting the

interrupts mask regsiter(IMR) and all __netif_rx_complete function to turn of polling.

c

RTL_W16_F(IntrMask, rtl8139 _intr_mask);

__netif_rx_complete(dev, napi);

__netif_rx_complete function removes this interface from the polling list.

5 Return number of packets received

return work_done;

} /*end of rtl8139_poll method*/

A received packet is put into a structure struct sk_buff called socket buffer. This structure is the

main encapsulation of a packet and contains pointers to point different layer headers in the packet and

pointers to input ,output interfaces . More about this structure can be read in Understanding Linux

network internals book.

Driver has to take the packet from the RxRing , copy that into sk_buff object and give it to the kernel.

rtl8139_rx local function will do that. This function receives three parameters , one is pointer to

net_device object ,second is pointer to device specific object struct rtl8139_private

and last is

‘budget’.

static int rtl8139_rx(struct net_device *dev, struct rtl8139_private *tp, int budget)

{

void __iomem *ioaddr = tp>mmio_addr;

int received = 0;

1 Get pointer to RxRing

unsigned char *rx_ring = tp>rx_ring;

2 Get current packet offeset (see device specification)

unsigned int cur_rx = tp>cur_rx;

3 Now the driver should run in a loop and take the each packet from the RxRing . The loop

should run until three conditions are satisfied . Three conditions are network interface should be

up, received packets should be less than the ‘budget’ and the RxRing is not empty.

while (netif_running(dev) && received <>

&& (RTL_R8 (ChipCmd) & RxBufEmpty) == 0) {

struct sk_buf *skb;

4 Get the packet offset into the RxRing

u32 ring_offset = cur_rx % RX_BUF_LEN;

5 Get the packet header (See device Manual) and find packet size. Note that packet size is

receive packet size minus 4(4 is CRC)

/* read size+status of next frame from DMA ring buffer */

rx_status = le32_to_cpu (*(__le32 *) (rx_ring + ring_offset));

rx_size = rx_status >> 16; /*receive packet size from packet header*/

pkt_size = rx_size – 4; /* packet size */

6 Allocate struct sk_buff object.

skb = dev_alloc_skb (pkt_size + 2);

7 Copy packet into sk_buff . This will copy packet into ‘data’ field of sk_buff

skb_copy_to_linear_data (skb, &rx_ring[ring_offset + 4], pkt_size);

8 Set ‘protocol’ field of sk_buff to appropriate packet type. This is ‘type’ of ethernet

frame. Call eth_type_trans to get the packet type. For token ting devices you can use

tr_type_trans function.

skb>protocol = eth_type_trans (skb, dev);

9 Update statistics

tp>stats.rx_bytes += pkt_size;

tp>stats.rx_packets++;

10 Give the skb to kernel

netif_receive_skb (skb);

received ++;

} /* end of while*/

tp>cu_rx = cur_rx;

return received;

}

netif_running inline function tests if the interface is up and running. It checks if

__LINK_STATE_START bit of ’status’ field of et_device object is set. dev_alloc_skb function

n

allocates memory for sk_buff from the cache and fills some the fields. netif_receive_skb is

main receive data processing function.

3.2 Packet Transmission

When the kernel has packets to send out of the interface ,it calls driver’s hard_start_xmit method.

hard_start_xmit function receives two parameters ,one is sk_buff of the packet to be transmitted and

another is net_device object .

sk_buff of the trasmitted packet is filled by the upper layers. ‘data’ field of sk_buff contains

packet to be sent. Driver should extract packet from the sk_buff and put that into TxBuffers. Then

driver should write length of packet and threshold in the Transmission descriptor status register of the

device . Then the device takes the packet from the Txbuffers and sends it.

Now we will describe rtl8139_start_xmit(hard_start_xmit) function of 8139too driver.

static int rtl8139_start_xmit (struct sk_buff *skb, struct net_device *dev)

{

struct rtl8139_private *tp = netdev_priv(dev);

void __iomem ioaddr = tp>mmio_addr; /*register base address */

unsigned int len = skb>len; /* length of the packet */

1 Calculate the next Tx descriptor entry

entry = tp>cur_tx % NUM_TX_DESC;

2 Copy the packet into TxBuffer

skb_copy_and_csum_dev(skb, tp>tx_buf[entry]);

3 Free the socket buffer sk_buff

dev_kfree_skb(skb);

4 Writer length of packet and threshold in Transmission descripor status register

RTL_W32_F (TxStatus0 + entry * sizeof (u32)),

(

tp>tx_flag | max(len, (unsigned int)ETH_ZLEN));

} /* end of tl8139_start_xmit */

Network interface card will take the packet from the TxBuffer and puts that in it's FIFO. Once the FIFO

reached threshold value set by the driver it sends the packet. After sending the packet network interface

card will raise an interrupt. Driver's interrupt handler will be called.

Driver's interrupt handler should check why the interrupt has occurred and if it is transmission interrupt it

updates its statistics. The following code shows how rtl8139too driver will handle the transmission

interrupt.

static irqreturn_t rtl8139_interrupt (int irq, void *dev_instance)

{

struct net_device *dev = (struct net_device *) dev_instance;

struct rtl8139_private *tp = netdev_priv(dev);

void __iomem *ioaddr = tp>mmio_addr;

status = status = RTL_R16 (IntrStatus);

1 Check if the interrupt is transmission interrupt and call rtl8139_tx_interrupt local

function.

if (status & (TxOK | TxErr)) {

rtl8139_tx_interrupt (dev, tp, ioaddr);

if (status & TxErr)

RTL_W16 (IntrStatus, TxErr);

} /* end of rtl8139_interrupt */

static void rtl8139_tx_interrupt (struct net_device *dev, struct rtl8139_private *tp,

void __iomem *ioaddr)

{

1 Read transmission descriptor status register and see what the status of the
packet.

txstatus = RTL_R32 (TxStatus0 + (entry * sizeof (u32)));

2 Increment error statistics if there are any problems in the transmission

if (txstatus & (TxOutOfWindow | TxAborted)) {

tp>stats.tx_errors++;

if (txstatus & TxAborted) {

tp>stats.tx_aborted_errors++;

RTL_W32 (TxConfig, TxClearAbt);

RTL_W16 (IntrStatus, TxErr);

}

if (txstatus & TxCarrierLost)

tp>stats.tx_carrier_errors++;

if (txstatus & TxOutOfWindow)

tp>stats.tx_window_errors++;

}

3 Increment statistics of successful transmitted packets

else {

tp>stats.collisions += (txstatus >> 24) & 15;

tp>stats.tx_bytes += txstatus & 0×7ff;

tp>stats.tx_packets++;

}

4 start the transmission queue allowing kernel to call driver’s hard_start_xmit method

again.

netif_wake_queue (dev);

}/* End of rtl8139_tx_interrupt */

4 Status and Control

4.1 When kernel wants to stop interface it calls stop method of driver

When kernel wants to stop interface it calls stop function of driver. This is called ,for example, when the

interface is brought down by using any utilities like ifconfig. Responsibility of this function is would be

exactly opposite to what we have done in open method. Some of the responsibilities include freeing

receive and transmission buffers , freeing irq and stopping transmission queue etc.

stop takes struct net_device object as its parameter. The following code shows how 8139too.c

implements stop method.

static int rtl8139_close (struct net_device *dev)

{

struct rtl8139_private *tp = netdev_priv(dev);

void __iomem *ioaddr = tp>mmio_addr;

unsigned long flags;

1 stop transmission queue. Once transmission queue is stopped kernel cannot send any more

packets to the driver

netif_stop_queue(dev);

2 Prevent poll function to be scheduled .

napi_disable(&tp>napi);

3 Stop the chip's Transmission and reception DMA processes . This can be done by writing 0 into

command register of device.

RTL_W8 (ChipCmd, 0);

4 Disable interrupts by clearing the interrupt mask register(IMR).

RTL_W16 (IntrMask, 0);

5 wait for pending IRQ handlers (on other CPUs) to be completed. This can be done using

synchronize_irq function.

synchronize_irq (dev>irq);

6 Unregister the interrupt handler (ISR)

free_irq(dev>irq,dev);

free_irq function removes an interrupt handler. The handler is removed and if the interrupt line

is no longer in use by any driver it is disabled. On a shared IRQ the driver must ensure the

interrupts are disabled by clearing interrupt mask register on the card it drives before calling

this function. This function does not return until any executing interrupts for this IRQ have

completed. This function must not be called from interrupt context. This function takes two

parameters one is irq line that is to be freed and second is 'dev_id' which is sent as last argument

to request_irq function.

7 Free receive(RxRing) and transmission(TxBuffer) buffers .

dma_free_coherent(&tp>pci_dev>dev,RX_BUF_TOT_LEN,

tp>rx_ring, tp>rx_ring_dma);

dma_free_coherent(&tp>pci_dev>dev,TX_BUF_TOT_LEN,

tp>tx_bufs, tp>tx_bufs_dma);

tp>rx_ring = NULL;

tp>tx_bufs = NULL;

return 0;

}

When Application wants statistics of the interface, drivers's get_stats

4.

method is called

Whenever an application needs to get statistics for the interface, get_stats method of driver is called. This

happens, for example, when ifconfig or netstat i is used by the user.

This function receives struct net_device object as its parameter and returns struct

net_device_stats object. Driver has to fill in the struct net_device_stats object with

interface statistics stored in device specific structure rtl8139_private and return it. The

following is the get_stats implementation of 8139too.c driver.

static struct net_device_stats *rtl8139_get_stats (struct net_device *dev)

{

struct rtl8139_private *tp = netdev_priv(dev);

void __iomem *ioaddr = tp>mmio_addr;

/*return statistics stored in 'stats' field of device specific object*/

return &tp>stats;

}

5 Uninitialization

5.1 remove function of pci_driver (pci remove function)

pci_driver 's remove method is called whenever network interface card is removed

or when the

driver module is unloaded. Functionalities of this driver includes unregistering the net_device with

the kernel and disabling the network interface card,freeing resources etc . remove function takes

pci_dev object as its parameter and returns nothing.

static void __devexit rtl8139_remove_one (struct pci_dev *pdev)

{

1 Get net_device object from pci_dev object. We have stored net_device object in

pci_dev object by calling pci_set_drvdata in probe method of the driver.

struct net_device *dev = pci_get_drvdata (pdev);

2 flush if there are any packets to be transmitted yet. This can be done using

flush_scheduled_work function. flush_scheduled_work function starts the work queue

rtl8139_thread ,that is created at the time of probe function.

flush_scheduled_work();

3 Unregister the net_device object with kernel

unregister_netdev (dev);

4 Free IO Resources and net_device object

__rtl8139_cleanup_dev (dev);

5 Disable the network interface card .

pci_disable_device (pdev);

}

static void __rtl8139_cleanup_dev (struct net_device *dev)

{

struct rtl8139_private *tp = netdev_priv(dev);

struct pci_dev *pdev;

pdev = tp>pci_dev;

1 Remove kernel page tables for IO resources

#ifdef USE_IO_OPS

```
if (tp>mmio_addr)

ioport_unmap (tp>mmio_addr);

#else

if (tp>mmio_addr)

pci_iounmap (pdev, tp>mmio_addr);

#endif /* USE_IO_OPS */
```

2 Release reserved PCI I/O and memory resources. These resources were previously reserved by

pci_request_regions function. This function takes pci_dev object as its parameter.

```
pci_release_regions (pdev);
```

3 Free net_device object using free_netdev function. free_netdev function does the last stage of

destroying an allocated device interface. The reference to the device object is released.

```
free_netdev(dev);

}
```

5.2 Unregistering driver with the low level bus interface (PCI Subsystem)

Last step in the network driver development will be unregistering with low level bus interface ,in this

case PCI subsystem. Driver can unregister with PCI subsystem using pci_unregister_driver function.

This function takes pci_driver object as its parameter.

Unregistering will be done in driver module's cleanup routine.

```
static void __exit rtl8139_cleanup_module (void)

{

pci_unregister_driver (&rtl8139_pci_driver);

}
```

References:

1) PCI Local Bus specification

2) Device specifications (RTL8139D_DataSheet.pdf , RTL8139_ProgrammersGuide.pdf)

3) Understanding Linux Network Internals by By Christian Benvenu ti

4) Linux source code

comments and suggestions can be sent to mohan@techveda.org

Posted by Unknown at 6:44 PM

2 comments:

SamRaj said...

Very good article. Thanks. Please write this kind of article which will be helpful to many. Thanks once again.

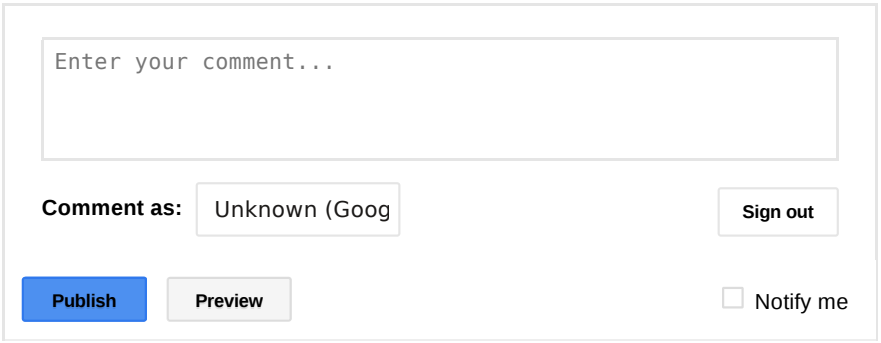
August 28, 2010 at 5:56 AM

iam aditya said...

I have been recently into this fiels ..basically Im a fresher with no experience in this field. I have to work on Spear board with ARM processor. Could you please give me an overview (like a road map),regarding development of linux applications on spearboard. I have no idea from where to start ....kindly help.

June 14, 2011 at 5:23 AM

Post a Comment



Newer Post Home Older Post