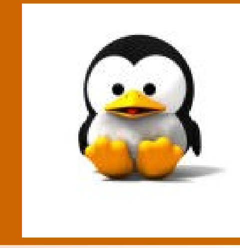
Linux Device Driver
(Network Drivers)

Amir Hossein Payberah

payberah@yahoo.com

Contents



Introduction

- Network interfaces are the third standard class of Linux devices.
- The role of a network interface within the system is similar to that of a mounted block device.
  - A block device registers its features in the blk_dev array and other kernel structures, and it then “transmits” and “receives” blocks on request, by means of its request function.
  - Similarly, a network interface must register itself in specific data structures in order to be invoked when packets are exchanged with the outside world.

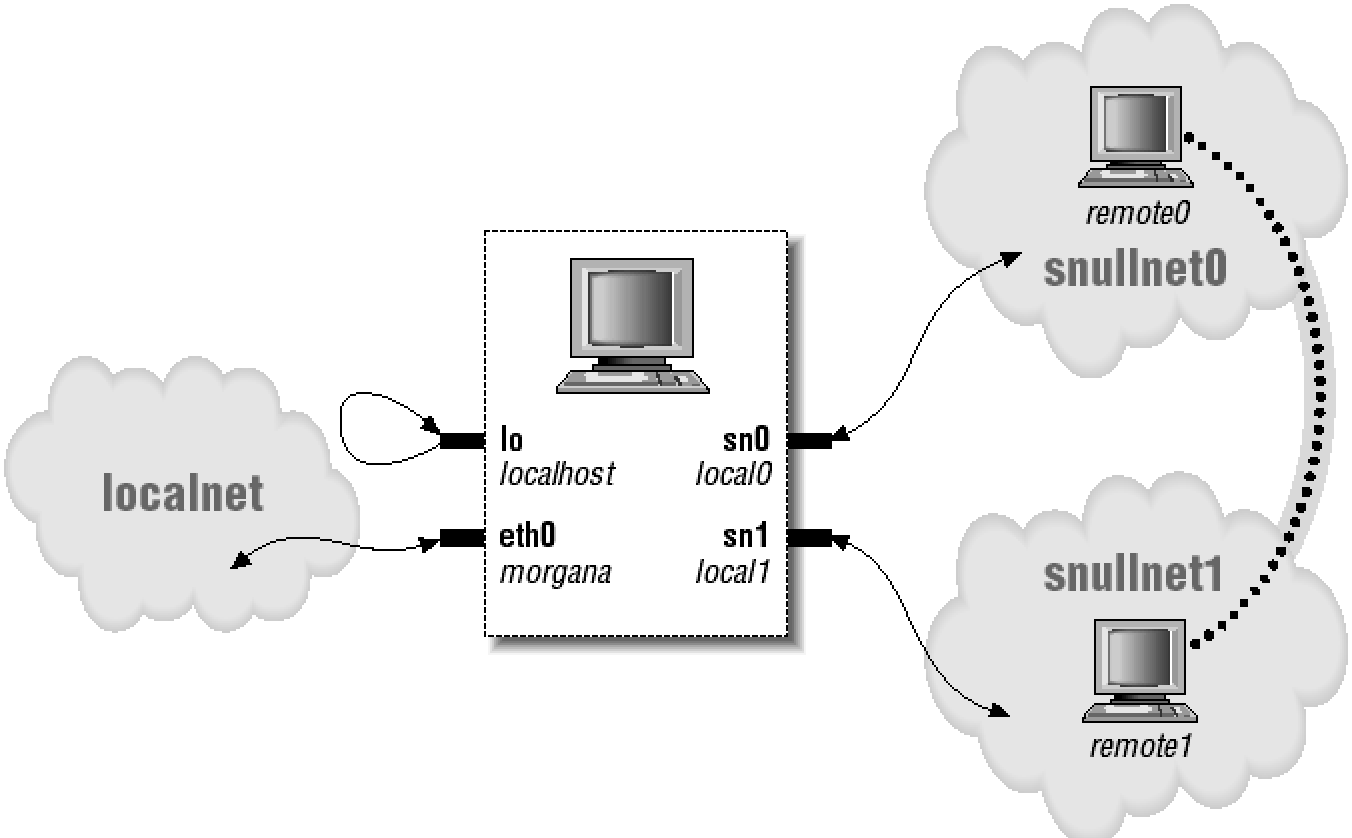
Introduction

- There are a few important differences between mounted disks and packet-delivery interfaces.
  - A disk exists as a special file in the /dev directory, whereas a network interface has no such entry point.
  - The read and write system calls when using sockets, act on a software object that is distinct from the interface.
  - The block drivers operate only in response to requests from the kernel, whereas network drivers receive packets asynchronously from the outside.

The snull design

- The snull module creates two interfaces.
  - These interfaces are different from a simple loopback.
  - Whatever you transmit through one of the interfaces loops back to the other one, not to itself.

The snull design



Contents

- Introduction
- The net_device structure
- Register the driver
- Opening and Closing
- Packet transmission
- Packet reception
- The interrupt handler
- Ioctl
- Statistical information
- The socket buffer
- MAC address resolution

The net_device structure

- The net_device structure is at the very core of the network driver layer.
- Struct net_device can be conceptually divided into two parts: visible and invisible.
  - The visible part of the structure is made up of the fields that can be explicitly assigned in static net_device structures.
  - The remaining fields are used internally by the network code and usually are not initialized at compilation time.
- It is defined in <linux/netdevice.h>.

The visible fields

- char name[IFNAMSIZ];
  - The name of the device.
- unsigned long rmem_end;
- unsigned long rmem_start;
- unsigned long mem_end;
- unsigned long mem_start;
  - These fields hold the beginning and ending addresses of the shared memory used by the device. The mem fields are used for transmit memory and the rmem fields for receive memory.

The visible fields

- unsigned long base_addr;
  - The I/O base address of the network interface.
- unsigned char irq;
  - The assigned interrupt number.
- unsigned char if_port;
  - Which port is in use on multiport devices.
- unsigned char dma;
  - The DMA channel allocated by the device.

The visible fields

- unsigned long state;
  - Device state. The field includes several flags.
- struct net_device *next;
  - Pointer to the next device in the global linked list.
- int (*init)(struct net_device *dev);
  - The initialization function.

The hidden fields

- These fields are usually assigned at device initialization.
- It has three separate groups.
  - Interface information
  - The device method
  - Utility fields

Interface information

- Most of the information about the interface is correctly set up by the function ether_setup.

Interface information

- unsigned short hard_header_len;
  - The hardware header length.
- unsigned mtu;
  - The maximum transfer unit (MTU).
- unsigned long tx_queue_len;
  - The maximum number of frames that can be queued on the device's transmission queue.
- unsigned short type;
  - The hardware type of the interface.
  - The type field is used by ARP to determine what kind of hardware address the interface supports.

Interface information

- unsigned char addr_len;
- unsigned char broadcast[MAX_ADDR_LEN];
- unsigned char dev_addr[MAX_ADDR_LEN];
  - Hardware (MAC) address length and device hardware addresses.
- unsigned short flags;
  - Interface flags.

The device method

- Device methods for a network interface can be divided into two groups: fundamental and optional.
- Fundamental methods include those that are needed to be able to use the interface.
- Optional methods implement more advanced functionalities that are not strictly required.

Fundamental methods

- int (*open)(struct net_device *dev);
- int (*stop)(struct net_device *dev);
- int (*hard_start_xmit) (struct sk_buff *skb, struct net_device *dev);
- int (*hard_header) (struct sk_buff *skb, struct net_device *dev, unsigned short type, void *daddr, void *saddr, unsigned len);

Fundamental methods

- int (*rebuild_header)(struct sk_buff *skb);
- void (*tx_timeout)(struct net_device *dev);
- struct net_device_stats *(*get_stats)(struct net_device *dev);
- int (*set_config)(struct net_device *dev, struct ifmap *map);

Optional methods

- int (*do_ioctl)(struct net_device *dev, struct ifreq *ifr, int cmd);
- void (*set_multicast_list)(struct net_device *dev);
- int (*set_mac_address)(struct net_device *dev, void *addr);
- int (*change_mtu)(struct net_device *dev, int new_mtu);

Optional methods

- int (*header_cache) (struct neighbour *neigh, struct hh_cache *hh);
  - header_cache is called to fill in the hh_cache structure with the results of an ARP query.
- int (*header_cache_update) (struct hh_cache *hh, struct net_device *dev, unsigned char *haddr);
- int (*hard_header_parse) (struct sk_buff *skb, unsigned char *haddr);

Utility fields

- These fields are used by the interface to hold useful status information.

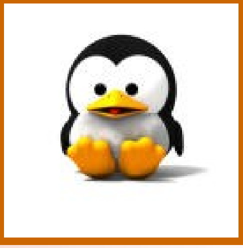
Utility fields

- unsigned long trans_start;
- unsigned long last_rx;
  - Both of these fields are meant to hold a jiffies value.
- int watchdog_timeo;
  - The minimum time (in jiffies) that should pass before the networking layer decides that a transmission timeout has occurred.
- void *priv;
  - The equivalent of filp->private_data.

Utility fields

- struct dev_mc_list *mc_list;
- int mc_count;
  - These two fields are used in handling multicast transmission.
- spinlock_t xmit_lock;
  - The xmit_lock is used to avoid multiple simultaneous calls to the driver’s hard_start_xmit function.
- int xmit_lock_owner;
  - The xmit_lock_owner is the number of the CPU that has obtained xmit_lock.
- struct module *owner;

Contents



- Introduction
- The net_device structure
- Register the driver
- Opening and Closing
- Packet transmission
- Packet reception
- The interrupt handler
- Ioctl
- Statistical information
- The socket buffer
- MAC address resolution

Driver initialization

- Each interface is described by a struct net_device item.
- Whenever you register a device, the kernel asks the driver to initialize itself.
- Initialization means probing for the physical interface and filling the net_device structure with the proper values.

Sample

- The structures for sn0 and sn1, the two snull interfaces, are declared like this:

```
struct net_device snull_devs[2] = {
  { init: snull_init, },
  { init: snull_init, }
};
```

Device name

- The driver can hardwire a name for the interface or it can allow dynamic assignment.

Sample

```
if (!snull_eth)
{
    strcpy(snull_devs[0].name, "sn0");
    strcpy(snull_devs[1].name, "sn1");
}
else
{
    strcpy(snull_devs[0].name, "eth%d");
    strcpy(snull_devs[1].name, "eth%d");
}
```

Register the driver

- int register_netdev(struct net_device *dev);
- void unregister_netdev(struct net_device *dev);

Sample

```
for (i=0; i<2; i++)
    register_netdev(snull_devs + i);
```

Initialize each device

- The main role of the initialization routine is to fill in the dev structure for this device.
  - The dev structure cannot be set up at compile time in the same manner as a file_operations or block_device_operations structure.
- Probing for the device should be performed in the init function for the interface.
  - No real probing is performed for the snull interface, because it is not bound to any hardware.

Sample

```
ether_setup(dev);
dev->open = snull_open;
dev->stop = snull_release;
dev->set_config = snull_config;
dev->hard_start_xmit = snull_tx;
dev->do_ioctl = snull_ioctl;
dev->get_stats = snull_stats;
dev->rebuild_header = snull_rebuild_header;
dev->hard_header = snull_header;
#ifdef HAVE_TX_TIMEOUT
dev->tx_timeout = snull_tx_timeout;
dev->watchdog_timeo = timeout;
#endif
dev->flags |= IFF_NOARP;
dev->hard_header_cache = NULL;
SET_MODULE_OWNER(dev);
```

Module unloading

- The module cleanup function simply unregisters the interfaces from the list after releasing memory associated with the private structure.

Sample

```
void snull_cleanup(void)
{
    int i;
    for (i=0; i<2; i++)
    {
        kfree(snull_devs[i].priv);
        unregister_netdev(snull_devs + i);
    }
    return;
}
```

Contents

Opening and Closing

- Before the interface can carry packets, however, the kernel must open it and assign an address to it.
- The kernel will open or close an interface in response to the ifconfig command.

Open and stop

- open requests any system resources it needs and tells the interface to come up.
  - The open method should also start the interface's transmit queue.
  - void netif_start_queue(struct net_device *dev);
- stop shuts down the interface and releases system resources.
  - void netif_stop_queue(struct net_device *dev);

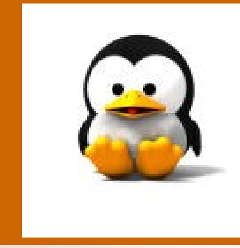
Sample

```
int snull_open(struct net_device *dev)
{
    MOD_INC_USE_COUNT;
    memcpy(dev->dev_addr, "\0SNUL0", ETH_ALEN);
    dev->dev_addr[ETH_ALEN-1] += (dev -
    snull_devs);
    netif_start_queue(dev);
    return 0;
}
```

Sample

```
int snull_release(struct net_device
  *dev)
{
  netif_stop_queue(dev);
  MOD_DEC_USE_COUNT;
  return 0;
}
```

Contents



Packet transmission

- The most important tasks performed by network interfaces are data transmission and reception.
- Whenever the kernel needs to transmit a data packet, it calls the hard_start_transmit method.
  - To put the data on an outgoing queue.
- Each packet handled by the kernel is contained in a socket buffer structure (struct sk_buff).
  - It is defined in <linux/skbuff.h>.

Sample

```
int snull_tx(struct sk_buff *skb, struct net_device *dev)
{
    int len;
    char *data;
    struct snull_priv *priv = (struct snull_priv *) dev->priv;
    len = skb->len < ETH_ZLEN ? ETH_ZLEN : skb->len;
    data = skb->data;
    dev->trans_start = jiffies;
    priv->skb = skb;
    snull_hw_tx(data, len, dev);
    return 0;
}
```

Transmission concurrency

- The hard_start_xmit function is protected from concurrent calls by a spinlock (xmit_lock) in the net_device structure.
  - As soon as the function returns, it may be called again.
  - The function returns when the software is done instructing the hardware about packet transmission, but hardware transmission will likely not have been completed.

Transmission concurrency

- Real hardware interfaces, transmit packets asynchronously and have a limited amount of memory available to store outgoing packets.
- When that memory is exhausted, the driver will need to tell the networking system not to start any more transmissions until the hardware is ready to accept new data.
- This notification is accomplished by calling netif_stop_queue.

Transmission concurrency

- Once your driver has stopped its queue, it must arrange to restart the queue at some point in the future, when it is again able to accept packets for transmission.
- void netif_wake_queue(struct net_device *dev);

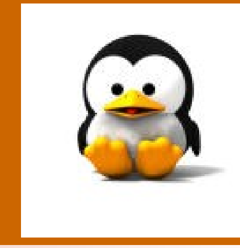
Transmission timeouts

- Interfaces can forget what they are doing, or the system can lose an interrupt.
- Many drivers handle this problem by setting timers.
- Network drivers need only set a timeout period, which goes in the watchdog_timeo field of the net_device structure.
- If the current system time exceeds the device’s trans_start time by at least the timeout period, the networking layer will eventually call the driver’s tx_timeout method.

Sample

```
void snull_tx_timeout (struct net_device *dev)
{
    struct snull_priv *priv = (struct snull_priv *) dev->priv;
    priv->status = SNULL_TX_INTR;
    snull_interrupt(0, dev, NULL);
    priv->stats.tx_errors++;
    netif_wake_queue(dev);
    return;
}
```

Contents



Packet reception

- In receiving data from the network an sk_buff must be allocated and handed off to the upper layers from within an interrupt handler.
- The function snull_rx is thus called after the hardware has received the packet and it is already in the computer’s memory.

Sample

```
void snull_rx(struct net_device *dev, int len, unsigned char *buf)
{
    struct sk_buff *skb;
    struct snull_priv *priv = (struct snull_priv *) dev->priv;
    skb = dev_alloc_skb(len+2);
    memcpy(skb_put(skb, len), buf, len);
    skb->dev = dev;
    skb->protocol = eth_type_trans(skb, dev);
    skb->ip_summed = CHECKSUM_UNNECESSARY; /* don't check it */
    priv->stats.rx_packets++;
    priv->stats.rx_bytes += len;
    netif_rx(skb);
    return;
}
```

Contents

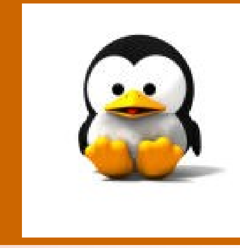
The interrupt handler

- Most hardware interfaces are controlled by means of an interrupt handler.
- The interface interrupts the processor to signal one of two possible events:
  - A new packet has arrived.
  - A transmission of an outgoing packet is complete.

Sample

```
void snull_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    int statusword;
    struct snull_priv *priv;
    struct net_device *dev = (struct net_device *)dev_id;
    priv = (struct snull_priv *) dev->priv;
    spin_lock(&priv->lock);
    statusword = priv->status;
    if (statusword & SNULL_RX_INTR)
        snull_rx(dev, priv->rx_packetlen, priv->rx_packetdata);
    if (statusword & SNULL_TX_INTR)
    {
        priv->stats.tx_packets++;
        priv->stats.tx_bytes += priv->tx_packetlen;
        dev_kfree_skb(priv->skb);
    }
    spin_unlock(&priv->lock);
    return;
}
```

Contents



Custom ioctl commands

- When the ioctl system call is invoked on a socket, the command number is one of the symbols defined in <linux/sockios.h>, and the function sock_ioctl directly invokes a protocol-specific function.
- Any ioctl command that is not recognized by the protocol layer is passed to the device layer.

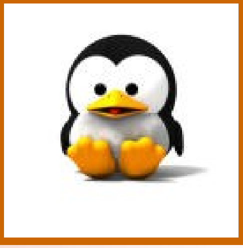
Custom ioctl commands

- These device-related ioctl commands accept a third argument from user space, a struct ifreq *.
- This structure is defined in <linux/if.h>.
- The SIOCSIFADDR and SIOCSIFMAP commands actually work on the ifreq structure.
- The ioctl implementation for sockets recognizes 16 commands as private to the interface:
  - SIOCDEVPRIVATE through SIOCDEVPRIVATE+15.

Custom ioctl commands

- When one of these commands is recognized, dev->do_ioctl is called in the relevant interface driver.
- int (*do_ioctl)(struct net_device *dev, struct ifreq *ifr, int cmd);

Contents



Statistical information

- A method a driver needs is get_stats.
- This method returns a pointer to the statistics for the device.

net_device_stats

- unsigned long rx_packets;
- unsigned long tx_packets;
  - These fields hold the total number of incoming and outgoing packets successfullytransferred by the interface.
- unsigned long rx_bytes;
- unsigned long tx_bytes;
  - The number of bytes received and transmitted by the interface.

net_device_stats

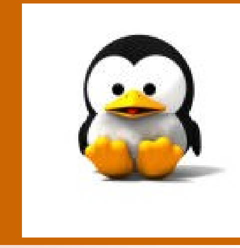
- unsigned long rx_errors;
- unsigned long tx_errors;
  - The number of erroneous receptions and transmissions.
- unsigned long rx_dropped;
- unsigned long tx_dropped;
  - The number of packets dropped during reception and transmission.
- unsigned long collisions;
  - The number of collisions due to congestion on the medium.
- unsigned long multicast;
  - The number of multicast packets received.

Sample

```
struct snull_priv
{
    struct net_device_stats stats;
    ...
};

struct net_device_stats *snull_stats(struct net_device
    *dev)
{
    struct snull_priv *priv = (struct snull_priv *) dev-
    >priv;
    return &priv->stats;
}
```

Contents



The socket buffer

- This structure is at the core of the network subsystem of the Linux kernel.
- It is defined in <linux/skbuff.h>.

The important fields

- struct net_device *rx_dev;
- struct net_device *dev;
  - The devices receiving and sending this buffer.
- union { /* . . . */ } h;
- union { /* . . . */ } nh;
- union { /* . . . */} mac;
  - Pointers to the various levels of headers contained within the packet.
  - h hosts pointers to transport layer headers.
  - nh includes network layer headers.
  - mac collects pointers to link layer headers.

The socket buffer

- This structure is at the core of the network subsystem of the Linux kernel.
- It is defined in <linux/skbuff.h>.

The important fields

- struct net_device *rx_dev;
- struct net_device *dev;
  - The devices receiving and sending this buffer.
- union { /* . . . */ } h;
- union { /* . . . */ } nh;
- union { /* . . . */} mac;
  - Pointers to the various levels of headers contained within the packet.
  - h hosts pointers to transport layer headers.
  - nh includes network layer headers.
  - mac collects pointers to link layer headers.

The important fields

- unsigned char *head;
- unsigned char *data;
- unsigned char *tail;
- unsigned char *end;
  - Pointers used to address the data in the packet.
- unsigned long len;
  - The length of the data itself (skb->tail - skb->data).
- unsigned char ip_summed;
  - The checksum policy for this packet.
- unsigned char pkt_type;
  - Packet classification used in delivering it.

Socket buffer functions

- struct sk_buff *alloc_skb(unsigned int len, int priority);
- struct sk_buff *dev_alloc_skb(unsigned int len);
  - Allocate a buffer.
- void kfree_skb(struct sk_buff *skb);
- void dev_kfree_skb(struct sk_buff *skb);
  - Free a buffer.

Socket buffer functions

- unsigned char *skb_put(struct sk_buff *skb, int len);
- unsigned char *_ _skb_put(struct sk_buff *skb, int len);
  - They are used to add data to the end of the buffer.
- unsigned char *skb_push(struct sk_buff *skb, int len);
- unsigned char *_ _skb_push(struct sk_buff *skb, int len);
  - They are similar to skb_put, except that data is added to the beginning of the packet instead of the end.

Socket buffer functions

- int skb_tailroom(struct sk_buff *skb);
  - This function returns the amount of space available for putting data in the buffer.
- int skb_headroom(struct sk_buff *skb);
  - Returns the amount of space available in front of data.
- void skb_reserve(struct sk_buff *skb, int len);
  - This function increments both data and tail.
- unsigned char *skb_pull(struct sk_buff *skb, int len);
  - Removes data from the head of the packet.

Contents

- Introduction
- The net_device structure
- Register the driver
- Opening and Closing
- Packet transmission
- Packet reception
- The interrupt handler
- Ioctl
- Statistical information
- The socket buffer
- MAC address resolution

MAC address resolution

- An interesting issue with Ethernet communication is how to associate the MAC addresses with the IP number.
- The usual way to deal with address resolution is by using ARP.

MAC address resolution

- Fortunately, ARP is managed by the kernel, and an Ethernet interface doesn’t need to do anything special to support ARP.
- As long as dev->addr and dev->addr_len are correctly assigned at open time, the driver doesn’t need to worry about resolving IP numbers to physical addresses;
  - ether_setup assigns the correct device methods to dev->hard_header and dev->rebuild_header.

Overriding ARP

- If your device wants to use the usual hardware header without running ARP, you need to override the default dev->hard_header method.

Sample

```
int snull_header(struct sk_buff *skb, struct net_device *dev,
unsigned short type, void *daddr, void *saddr,
unsigned int len)
{
    struct ethhdr *eth = (struct ethhdr
    *)skb_push(skb,ETH_HLEN);
    eth->h_proto = htons(type);
    memcpy(eth->h_source, saddr ? saddr : dev->dev_addr,
    dev->addr_len);
    memcpy(eth->h_dest, daddr ? daddr : dev->dev_addr, dev-
    >addr_len);
    eth->h_dest[ETH_ALEN-1] ^= 0x01;
    return (dev->hard_header_len);
}
```

Question?