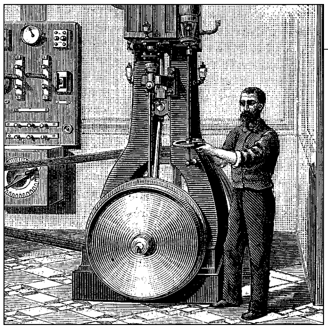

CHAPTER 17
Network Drivers

Having discussed char and block drivers, we are now ready to move on to the world of networking. Network interfaces are the third standard class of Linux devices, and this chapter describes how they interact with the rest of the kernel.

The role of a network interface within the system is similar to that of a mounted block device. A block device registers its disks and methods with the kernel, and then "transmits" and "receives" blocks on request, by means of its *request* function. Similarly, a network interface must register itself within specific kernel data structures in order to be invoked when packets are exchanged with the outside world.

There are a few important differences between mounted disks and packet-delivery interfaces. To begin with, a disk exists as a special file in the */dev* directory, whereas a network interface has no such entry point. The normal file operations (read, write, and so on) do not make sense when applied to network interfaces, so it is not possible to apply the Unix "everything is a file" approach to them. Thus, network interfaces exist in their own namespace and export a different set of operations.

Although you may object that applications use the *read* and *write* system calls when using sockets, those calls act on a software object that is distinct from the interface. Several hundred sockets can be multiplexed on the same physical interface.

But the most important difference between the two is that block drivers operate only in response to requests from the kernel, whereas network drivers receive packets asynchronously from the outside. Thus, while a block driver is *asked* to send a buffer toward the kernel, the network device *asks* to push incoming packets toward the kernel. The kernel interface for network drivers is designed for this different mode of operation.

Network drivers also have to be prepared to support a number of administrative tasks, such as setting addresses, modifying transmission parameters, and maintaining traffic and error statistics. The API for network drivers reflects this need and, therefore, looks somewhat different from the interfaces we have seen so far.

The network subsystem of the Linux kernel is designed to be completely protocol-independent. This applies to both networking protocols (Internet protocol [IP] versus IPX or other protocols) and hardware protocols (Ethernet versus token ring, etc.). Interaction between a network driver and the kernel properly deals with one network packet at a time; this allows protocol issues to be hidden neatly from the driver and the physical transmission to be hidden from the protocol.

This chapter describes how the network interfaces fit in with the rest of the Linux kernel and provides examples in the form of a memory-based modularized network interface, which is called (you guessed it) *snull*. To simplify the discussion, the interface uses the Ethernet hardware protocol and transmits IP packets. The knowledge you acquire from examining *snull* can be readily applied to protocols other than IP, and writing a non-Ethernet driver is different only in tiny details related to the actual network protocol.

This chapter doesn't talk about IP numbering schemes, network protocols, or other general networking concepts. Such topics are not (usually) of concern to the driver writer, and it's impossible to offer a satisfactory overview of networking technology in less than a few hundred pages. The interested reader is urged to refer to other books describing networking issues.

One note on terminology is called for before getting into network devices. The networking world uses the term *octet* to refer to a group of eight bits, which is generally the smallest unit understood by networking devices and protocols. The term byte is almost never encountered in this context. In keeping with standard usage, we will use octet when talking about networking devices.

8 bits构成的字节

The term "header" also merits a quick mention. A header is a set of bytes (err, octets) prepended to a packet as it is passed through the various layers of the networking subsystem. When an application sends a block of data through a TCP socket, the networking subsystem breaks that data up into packets and puts a TCP header, describing where each packet fits within the stream, at the beginning. The lower levels then put an IP header, used to route the packet to its destination, in front of the TCP header. If the packet moves over an Ethernet-like medium, an Ethernet header, interpreted by the hardware, goes in front of the rest. Network drivers need not concern themselves with higher-level headers (usually), but they often must be involved in the creation of the hardware-level header.

How snull Is Designed

This section discusses the design concepts that led to the *snull* network interface. Although this information might appear to be of marginal use, failing to understand it might lead to problems when you play with the sample code.

The first, and most important, design decision was that the sample interfaces should remain independent of real hardware, just like most of the sample code used in this book. This constraint led to something that resembles the loopback interface. *snull* is not a loopback interface; however, it simulates conversations with real remote hosts in order to better demonstrate the task of writing a network driver. The Linux loopback driver is actually quite simple; it can be found in *drivers/net/loopback.c*.

Another feature of *snull* is that it supports only IP traffic. This is a consequence of the internal workings of the interface—*snull* has to look inside and interpret the packets to properly emulate a pair of hardware interfaces. Real interfaces don't depend on the protocol being transmitted, and this limitation of *snull* doesn't affect the fragments of code shown in this chapter.

Assigning IP Numbers

The *snull* module creates two interfaces. These interfaces are different from a simple loopback, in that whatever you transmit through one of the interfaces loops back to the other one, not to itself. It looks like you have two external links, but actually your computer is replying to itself.

Unfortunately, this effect can't be accomplished through IP number assignments alone, because the kernel wouldn't send out a packet through interface A that was directed to its own interface B. Instead, it would use the loopback channel without passing through *snull*. To be able to establish a communication through the *snull* interfaces, the source and destination addresses need to be modified during data transmission. In other words, packets sent through one of the interfaces should be received by the other, but the receiver of the outgoing packet shouldn't be recognized as the local host. The same applies to the source address of received packets.

To achieve this kind of "hidden loopback," the *snull* interface toggles the least significant bit of the third octet of both the source and destination addresses; that is, it changes both the network number and the host number of class C IP numbers. The net effect is that packets sent to network A (connected to `sn0`, the first interface) appear on the `sn1` interface as packets belonging to network B.

To avoid dealing with too many numbers, let's assign symbolic names to the IP numbers involved:

- `snullnet0` is the network that is connected to the `sn0` interface. Similarly, `snullnet1` is the network connected to `sn1`. The addresses of these networks should differ only in the least significant bit of the third octet. These networks must have 24-bit netmasks.
- `local0` is the IP address assigned to the `sn0` interface; it belongs to `snullnet0`. The address associated with `sn1` is `local1`. `local0` and `local1` must differ in the least significant bit of their third octet and in the fourth octet.
- `remote0` is a host in `snullnet0`, and its fourth octet is the same as that of `local1`. Any packet sent to `remote0` reaches `local1` after its network address has been

modified by the interface code. The host remote1 belongs to snullnet1, and its fourth octet is the same as that of local0.

The operation of the *snull* interfaces is depicted in Figure 17-1, in which the hostname associated with each interface is printed near the interface name.

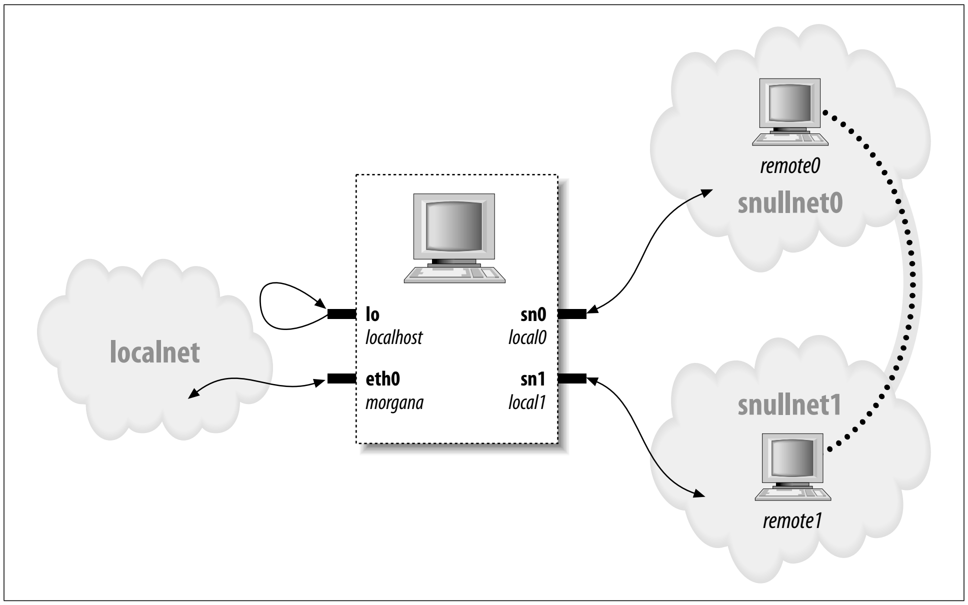

Figure 17-1. How a host sees its interfaces

Here are possible values for the network numbers. Once you put these lines in */etc/networks*, you can call your networks by name. The values were chosen from the range of numbers reserved for private use.

```
snullnet0       192.168.0.0
snullnet1       192.168.1.0
```

The following are possible host numbers to put into */etc/hosts*:

```
192.168.0.1   local0
192.168.0.2   remote0
192.168.1.2   local1
192.168.1.1   remote1
```

The important feature of these numbers is that the host portion of local0 is the same as that of remote1, and the host portion of local1 is the same as that of remote0. You can use completely different numbers as long as this relationship applies.

Be careful, however, if your computer is already connected to a network. The numbers you choose might be real Internet or intranet numbers, and assigning them to your interfaces prevents communication with the real hosts. For example, although the numbers just shown are not routable Internet numbers, they could already be used by your private network.

Whatever numbers you choose, you can correctly set up the interfaces for operation by issuing the following commands:

```
ifconfig sn0 local0
ifconfig sn1 local1
```

You may need to add the `netmask 255.255.255.0` parameter if the address range chosen is not a class C range.

At this point, the "remote" end of the interface can be reached. The following screendump shows how a host reaches `remote0` and `remote1` through the *snull* interface:

```
morgana% ping -c 2 remote0
64 bytes from 192.168.0.99: icmp_seq=0 ttl=64 time=1.6 ms
64 bytes from 192.168.0.99: icmp_seq=1 ttl=64 time=0.9 ms
2 packets transmitted, 2 packets received, 0% packet loss

morgana% ping -c 2 remote1
64 bytes from 192.168.1.88: icmp_seq=0 ttl=64 time=1.8 ms
64 bytes from 192.168.1.88: icmp_seq=1 ttl=64 time=0.9 ms
2 packets transmitted, 2 packets received, 0% packet loss
```

Note that you won't be able to reach any other "host" belonging to the two networks, because the packets are discarded by your computer after the address has been modified and the packet has been received. For example, a packet aimed at 192.168.0.32 will leave through `sn0` and reappear at `sn1` with a destination address of 192.168.1.32, which is not a local address for the host computer.

The Physical Transport of Packets

As far as data transport is concerned, the *snull* interfaces belong to the Ethernet class.

snull emulates Ethernet because the vast majority of existing networks—at least the segments that a workstation connects to—are based on Ethernet technology, be it 10base-T, 100base-T, or Gigabit. Additionally, the kernel offers some generalized support for Ethernet devices, and there's no reason not to use it. The advantage of being an Ethernet device is so strong that even the *plip* interface (the interface that uses the printer ports) declares itself as an Ethernet device.

The last advantage of using the Ethernet setup for *snull* is that you can run *tcpdump* on the interface to see the packets go by. Watching the interfaces with *tcpdump* can be a useful way to see how the two interfaces work.

As was mentioned previously, *snull* works only with IP packets. This limitation is a result of the fact that *snull* snoops in the packets and even modifies them, in order for the code to work. The code modifies the source, destination, and checksum in the IP header of each packet without checking whether it actually conveys IP information.

This quick-and-dirty data modification destroys non-IP packets. If you want to deliver other protocols through *snull*, you must modify the module's source code.

Connecting to the Kernel

We start looking at the structure of network drivers by dissecting the *snull* source. Keeping the source code for several drivers handy might help you follow the discussion and to see how real-world Linux network drivers operate. As a place to start, we suggest *loopback.c*, *plip.c*, and *e100.c*, in order of increasing complexity. All these files live in *drivers/net*, within the kernel source tree.

Device Registration

When a driver module is loaded into a running kernel, it requests resources and offers facilities; there's nothing new in that. And there's also nothing new in the way resources are requested. The driver should probe for its device and its hardware location (I/O ports and IRQ line)—but not register them—as described in "Installing an Interrupt Handler" in Chapter 10. The way a network driver is registered by its module initialization function is different from char and block drivers. Since there is no equivalent of major and minor numbers for network interfaces, a network driver does not request such a number. Instead, the driver inserts a data structure for each newly detected interface into a global list of network devices.

Each interface is described by a `struct net_device` item, which is defined in *<linux/netdevice.h>*. The *snull* driver keeps pointers to two of these structures (for `sn0` and `sn1`) in a simple array:

```
struct net_device *snull_devs[2];
```

The `net_device` structure, like many other kernel structures, contains a kobject and is, therefore, reference-counted and exported via sysfs. As with other such structures, it must be allocated dynamically. The kernel function provided to perform this allocation is *alloc_netdev*, which has the following prototype:

```
struct net_device *alloc_netdev(int sizeof_priv,
                                const char *name,
                                void (*setup)(struct net_device *));
```

Here, `sizeof_priv` is the size of the driver's "private data" area; with network devices, that area is allocated along with the `net_device` structure. In fact, the two are allocated together in one large chunk of memory, but driver authors should pretend that they don't know that. `name` is the name of this interface, as is seen by user space; this name can have a *printf*-style `%d` in it. The kernel replaces the `%d` with the next available interface number. Finally, `setup` is a pointer to an initialization function that is called to set up the rest of the `net_device` structure. We get to the initialization function shortly, but, for now, suffice it to say that *snull* allocates its two device structures in this way:

```
snull_devs[0] = alloc_netdev(sizeof(struct snull_priv), "sn%d",
        snull_init);
snull_devs[1] = alloc_netdev(sizeof(struct snull_priv), "sn%d",
        snull_init);
if (snull_devs[0] == NULL || snull_devs[1] == NULL)
    goto out;
```

As always, we must check the return value to ensure that the allocation succeeded.

The networking subsystem provides a number of helper functions wrapped around *alloc_netdev* for various types of interfaces. The most common is *alloc_etherdev*, which is defined in <*linux/etherdevice.h*>:

```
struct net_device *alloc_etherdev(int sizeof_priv);
```

This function allocates a network device using `eth%d` for the name argument. It provides its own initialization function (*ether_setup*) that sets several `net_device` fields with appropriate values for Ethernet devices. Thus, there is no driver-supplied initialization function for *alloc_etherdev*; the driver should simply do its required initialization directly after a successful allocation. Writers of drivers for other types of devices may want to take advantage of one of the other helper functions, such as *alloc_fcdev* (defined in <*linux/fcdevice.h*>) for fiber-channel devices, *alloc_fddidev* (<*linux/fddidevice.h*>) for FDDI devices, or *alloc_trdev* (<*linux/trdevice.h*>) for token ring devices.

snull could use *alloc_etherdev* without trouble; we chose to use *alloc_netdev* instead, as a way of demonstrating the lower-level interface and to give us control over the name assigned to the interface.

Once the `net_device` structure has been initialized, completing the process is just a matter of passing the structure to *register_netdev*. In *snull*, the call looks as follows:

```
for (i = 0; i < 2;  i++)
    if ((result = register_netdev(snull_devs[i])))
        printk("snull: error %i registering device \"%s\"\n",
                result, snull_devs[i]->name);
```

The usual cautions apply here: as soon as you call *register_netdev*, your driver may be called to operate on the device. Thus, you should not register the device until everything has been completely initialized.

Initializing Each Device

We have looked at the allocation and registration of `net_device` structures, but we passed over the intermediate step of completely initializing that structure. Note that `struct net_device` is always put together at runtime; it cannot be set up at compile time in the same manner as a `file_operations` or `block_device_operations` structure. This initialization must be complete before calling *register_netdev*. The `net_device` structure is large and complicated; fortunately, the kernel takes care of some Ethernet-wide defaults through the *ether_setup* function (which is called by *alloc_etherdev*).

Since *snull* uses *alloc_netdev*, it has a separate initialization function. The core of this function (*snull_init*) is as follows:

```
ether_setup(dev); /* assign some of the fields */

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;
dev->tx_timeout      = snull_tx_timeout;
dev->watchdog_timeo = timeout;
/* keep the default flags, just add NOARP */
dev->flags           |= IFF_NOARP; //unable to handle ARP protocol
dev->features        |= NETIF_F_NO_CSUM;
dev->hard_header_cache = NULL;      /* Disable caching */
```

The above code is a fairly routine initialization of the `net_device` structure; it is mostly a matter of storing pointers to our various driver functions. The single unusual feature of the code is setting `IFF_NOARP` in the flags. This specifies that the interface cannot use the Address Resolution Protocol (ARP). ARP is a low-level Ethernet protocol; its job is to turn IP addresses into Ethernet medium access control (MAC) addresses. Since the "remote" systems simulated by *snull* do not really exist, there is nobody available to answer ARP requests for them. Rather than complicate *snull* with the addition of an ARP implementation, we chose to mark the interface as being unable to handle that protocol. The assignment to `hard_header_cache` is there for a similar reason: it disables the caching of the (nonexistent) ARP replies on this interface. This topic is discussed in detail in the section "MAC Address Resolution" later in this chapter.

ARP:地址解析协议

The initialization code also sets a couple of fields (`tx_timeout` and `watchdog_timeo`) that relate to the handling of transmission timeouts. We cover this topic thoroughly in the section "Transmission Timeouts."

We look now at one more `struct net_device` field, `priv`. Its role is similar to that of the `private_data` pointer that we used for char drivers. Unlike `fops->private_data`, this `priv` pointer is allocated along with the `net_device` structure. Direct access to the `priv` field is also discouraged, for performance and flexibility reasons. When a driver needs to get access to the private data pointer, it should use the *netdev_priv* function. Thus, the *snull* driver is full of declarations such as:

```
struct snull_priv *priv = netdev_priv(dev);
```

The *snull* module declares a `snull_priv` data structure to be used for `priv`:

```
struct snull_priv {
    struct net_device_stats stats;
    int status;
    struct snull_packet *ppool;
    struct snull_packet *rx_queue;  /* List of incoming packets */
    int rx_int_enabled;
    int tx_packetlen;
    u8 *tx_packetdata;
    struct sk_buff *skb;
    spinlock_t lock;
};
```

The structure includes, among other things, an instance of `struct net_device_stats`, which is the standard place to hold interface statistics. The following lines in *snull_init* allocate and initialize `dev->priv`:

```
priv = netdev_priv(dev);
memset(priv, 0, sizeof(struct snull_priv));
spin_lock_init(&priv->lock);
snull_rx_ints(dev, 1);      /* enable receive interrupts */
```

Module Unloading

Nothing special happens when the module is unloaded. The module cleanup function simply unregisters the interfaces, performs whatever internal cleanup is required, and releases the `net_device` structure back to the system:

```
void snull_cleanup(void)
{
    int i;

    for (i = 0; i < 2;  i++) {
        if (snull_devs[i]) {
            unregister_netdev(snull_devs[i]);
            snull_teardown_pool(snull_devs[i]);
            free_netdev(snull_devs[i]);
        }
    }
    return;
}
```

The call to *unregister_netdev* removes the interface from the system; *free_netdev* returns the `net_device` structure to the kernel. If a reference to that structure exists somewhere, it may continue to exist, but your driver need not care about that. Once you have unregistered the interface, the kernel no longer calls its methods.

Note that our internal cleanup (done in *snull_teardown_pool*) cannot happen until the device has been unregistered. It must, however, happen before we return the `net_device` structure to the system; once we have called *free_netdev*, we cannot make any further references to the device or our private area.

The net_device Structure in Detail

The `net_device` structure is at the very core of the network driver layer and deserves a complete description. This list describes all the fields, but more to provide a reference than to be memorized. The rest of this chapter briefly describes each field as soon as it is used in the sample code, so you don't need to keep referring back to this section.

Global Information

The first part of `struct net_device` is composed of the following fields:

`char name[IFNAMSIZ];`
: The name of the device. If the name set by the driver contains a `%d` format string, *register_netdev* replaces it with a number to make a unique name; assigned numbers start at `0`.

`unsigned long state;`
: Device state. The field includes several flags. Drivers do not normally manipulate these flags directly; instead, a set of utility functions has been provided. These functions are discussed shortly when we get into driver operations.

`struct net_device *next;`
: Pointer to the next device in the global linked list. This field shouldn't be touched by the driver.

`int (*init)(struct net_device *dev);`
: An initialization function. If this pointer is set, the function is called by *register_netdev* to complete the initialization of the `net_device` structure. Most modern network drivers do not use this function any longer; instead, initialization is performed before registering the interface.

Hardware Information

The following fields contain low-level hardware information for relatively simple devices. They are a holdover from the earlier days of Linux networking; most modern drivers do make use of them (with the possible exception of `if_port`). We list them here for completeness.

`unsigned long rmem_end;`
`unsigned long rmem_start;`
`unsigned long mem_end;`
`unsigned long mem_start;`
: Device memory information. These fields hold the beginning and ending addresses of the shared memory used by the device. If the device has different receive and transmit memories, the `mem` fields are used for transmit memory and the `rmem` fields for receive memory. The `rmem` fields are never referenced outside

of the driver itself. By convention, the `end` fields are set so that `end - start` is the amount of available onboard memory.

`unsigned long base_addr;`
: The I/O base address of the network interface. This field, like the previous ones, is assigned by the driver during the device probe. The *ifconfig* command can be used to display or modify the current value. The `base_addr` can be explicitly assigned on the kernel command line at system boot (via the `netdev=` parameter) or at module load time. The field, like the memory fields described above, is not used by the kernel.

`unsigned char irq;`
: The assigned interrupt number. The value of `dev->irq` is printed by *ifconfig* when interfaces are listed. This value can usually be set at boot or load time and modified later using *ifconfig*.

`unsigned char if_port;`
: The port in use on multiport devices. This field is used, for example, with devices that support both coaxial (`IF_PORT_10BASE2`) and twisted-pair (`IF_PORT_100BASET`) Ethernet connections. The full set of known port types is defined in <*linux/netdevice.h*>.

`unsigned char dma;`
: The DMA channel allocated by the device. The field makes sense only with some peripheral buses, such as ISA. It is not used outside of the device driver itself but for informational purposes (in *ifconfig*).

Interface Information

Most of the information about the interface is correctly set up by the *ether_setup* function (or whatever other setup function is appropriate for the given hardware type). Ethernet cards can rely on this general-purpose function for most of these fields, but the `flags` and `dev_addr` fields are device specific and must be explicitly assigned at initialization time.

Some non-Ethernet interfaces can use helper functions similar to *ether_setup*. *drivers/net/net_init.c* exports a number of such functions, including the following:

`void ltalk_setup(struct net_device *dev);`
: Sets up the fields for a LocalTalk device

`void fc_setup(struct net_device *dev);`
: Initializes fields for fiber-channel devices

`void fddi_setup(struct net_device *dev);`
: Configures an interface for a Fiber Distributed Data Interface (FDDI) network

`void hippi_setup(struct net_device *dev);`
: Prepares fields for a High-Performance Parallel Interface (HIPPI) high-speed interconnect driver

`void tr_setup(struct net_device *dev);`
: Handles setup for token ring network interfaces

Most devices are covered by one of these classes. If yours is something radically new and different, however, you need to assign the following fields by hand:

`unsigned short hard_header_len;`
: The hardware header length, that is, the number of octets that lead the transmitted packet before the IP header, or other protocol information. The value of `hard_header_len` is `14` (`ETH_HLEN`) for Ethernet interfaces.

`unsigned mtu;`
: The maximum transfer unit (MTU). This field is used by the network layer to drive packet transmission. Ethernet has an MTU of 1500 octets (`ETH_DATA_LEN`). This value can be changed with *ifconfig*.

`unsigned long tx_queue_len;`
: The maximum number of frames that can be queued on the device's transmission queue. This value is set to 1000 by *ether_setup*, but you can change it. For example, *plip* uses 10 to avoid wasting system memory (*plip* has a lower throughput than a real Ethernet interface).

`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. The proper value for Ethernet interfaces is `ARPHRD_ETHER`, and that is the value set by *ether_setup*. The recognized types are defined in <*linux/if_arp.h*>.

`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. The Ethernet address length is six octets (we are referring to the hardware ID of the interface board), and the broadcast address is made up of six `0xff` octets; *ether_setup* arranges for these values to be correct. The device address, on the other hand, must be read from the interface board in a device-specific way, and the driver should copy it to `dev_addr`. The hardware address is used to generate correct Ethernet headers before the packet is handed over to the driver for transmission. The *snull* device doesn't use a physical interface, and it invents its own hardware address.

`unsigned short flags;`
`int features;`
: Interface flags (detailed next).

The `flags` field is a bit mask including the following bit values. The `IFF_` prefix stands for "interface flags." Some flags are managed by the kernel, and some are set by the interface at initialization time to assert various capabilities and other features of the interface. The valid flags, which are defined in <*linux/if.h*>, are:

`IFF_UP`
: This flag is read-only for the driver. The kernel turns it on when the interface is active and ready to transfer packets.

`IFF_BROADCAST`
: This flag (maintained by the networking code) states that the interface allows broadcasting. Ethernet boards do.

`IFF_DEBUG`
: This marks debug mode. The flag can be used to control the verbosity of your *printk* calls or for other debugging purposes. Although no in-tree driver currently uses this flag, it can be set and reset by user programs via *ioctl*, and your driver can use it. The *misc-progs/netifdebug* program can be used to turn the flag on and off.

`IFF_LOOPBACK`
: This flag should be set only in the loopback interface. The kernel checks for `IFF_LOOPBACK` instead of hardwiring the `lo` name as a special interface.

`IFF_POINTOPOINT`
: This flag signals that the interface is connected to a point-to-point link. It is set by the driver or, sometimes, by *ifconfig*. For example, *plip* and the PPP driver have it set.

`IFF_NOARP`
: This means that the interface can't perform ARP. For example, point-to-point interfaces don't need to run ARP, which would only impose additional traffic without retrieving useful information. *snull* runs without ARP capabilities, so it sets the flag.

`IFF_PROMISC`
: This flag is set (by the networking code) to activate promiscuous operation. By default, Ethernet interfaces use a hardware filter to ensure that they receive broadcast packets and packets directed to that interface's hardware address only. Packet sniffers such as *tcpdump* set promiscuous mode on the interface in order to retrieve all packets that travel on the interface's transmission medium.

`IFF_MULTICAST`
: This flag is set by drivers to mark interfaces that are capable of multicast transmission. *ether_setup* sets `IFF_MULTICAST` by default, so if your driver does not support multicast, it must clear the flag at initialization time.

`IFF_ALLMULTI`
: This flag tells the interface to receive all multicast packets. The kernel sets it when the host performs multicast routing, only if `IFF_MULTICAST` is set. `IFF_ALLMULTI` is

read-only for the driver. Multicast flags are used in the section "Multicast," later in this chapter.

`IFF_MASTER`
`IFF_SLAVE`
: These flags are used by the load equalization code. The interface driver doesn't need to know about them.

`IFF_PORTSEL`
`IFF_AUTOMEDIA`
: These flags signal that the device is capable of switching between multiple media types; for example, unshielded twisted pair (UTP) versus coaxial Ethernet cables. If `IFF_AUTOMEDIA` is set, the device selects the proper medium automatically. In practice, the kernel makes no use of either flag.

`IFF_DYNAMIC`
: This flag, set by the driver, indicates that the address of this interface can change. It is not currently used by the kernel.

`IFF_RUNNING`
: This flag indicates that the interface is up and running. It is mostly present for BSD compatibility; the kernel makes little use of it. Most network drivers need not worry about `IFF_RUNNING`.

`IFF_NOTRAILERS`
: This flag is unused in Linux, but it exists for BSD compatibility.

When a program changes `IFF_UP`, the *open* or *stop* device method is called. Furthermore, when `IFF_UP` or any other flag is modified, the *set_multicast_list* method is invoked. If the driver needs to perform some action in response to a modification of the flags, it must take that action in *set_multicast_list*. For example, when `IFF_PROMISC` is set or reset, *set_multicast_list* must notify the onboard hardware filter. The responsibilities of this device method are outlined in the section "Multicast."

The `features` field of the `net_device` structure is set by the driver to tell the kernel about any special hardware capabilities that this interface has. We will discuss some of these features; others are beyond the scope of this book. The full set is:

`NETIF_F_SG`
`NETIF_F_FRAGLIST`
: Both of these flags control the use of scatter/gather I/O. If your interface can transmit a packet that has been split into several distinct memory segments, you should set `NETIF_F_SG`. Of course, you have to actually implement the scatter/gather I/O (we describe how that is done in the section "Scatter/Gather I/O"). `NETIF_F_FRAGLIST` states that your interface can cope with packets that have been fragmented; only the loopback driver does this in 2.6.

: Note that the kernel does not perform scatter/gather I/O to your device if it does not also provide some form of checksumming as well. The reason is that, if the kernel has to make a pass over a fragmented (“nonlinear”) packet to calculate the checksum, it might as well copy the data and coalesce the packet at the same time.

`NETIF_F_IP_CSUM`
`NETIF_F_NO_CSUM`
`NETIF_F_HW_CSUM`

These flags are all ways of telling the kernel that it need not apply checksums to some or all packets leaving the system by this interface. Set `NETIF_F_IP_CSUM` if your interface can checksum IP packets but not others. If no checksums are ever required for this interface, set `NETIF_F_NO_CSUM`. The loopback driver sets this flag, and *snull* does, too; since packets are only transferred through system memory, there is (one hopes!) no opportunity for them to be corrupted, and no need to check them. If your hardware does checksumming itself, set `NETIF_F_HW_CSUM`.

`NETIF_F_HIGHDMA`

Set this flag if your device can perform DMA to high memory. In the absence of this flag, all packet buffers provided to your driver are allocated in low memory.

`NETIF_F_HW_VLAN_TX`
`NETIF_F_HW_VLAN_RX`
`NETIF_F_HW_VLAN_FILTER`
`NETIF_F_VLAN_CHALLENGED`

These options describe your hardware’s support for 802.1q VLAN packets. VLAN support is beyond what we can cover in this chapter. If VLAN packets confuse your device (which they really shouldn’t), set the `NETIF_F_VLAN_CHALLENGED` flag.

`NETIF_F_TSO`

Set this flag if your device can perform TCP segmentation offloading. TSO is an advanced feature that we cannot cover here.

The Device Methods

As happens with the char and block drivers, each network device declares the functions that act on it. Operations that can be performed on network interfaces are listed in this section. Some of the operations can be left `NULL`, and others are usually untouched because *ether_setup* assigns suitable methods to them.

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. The following are the fundamental methods:

`int (*open)(struct net_device *dev);`

Opens the interface. The interface is opened whenever *ifconfig* activates it. The *open* method should register any system resource it needs (I/O ports, IRQ, DMA, etc.), turn on the hardware, and perform any other setup your device requires.

`int (*stop)(struct net_device *dev);`

Stops the interface. The interface is stopped when it is brought down. This function should reverse operations performed at open time.

`int (*hard_start_xmit) (struct sk_buff *skb, struct net_device *dev);`

Method that initiates the transmission of a packet. The full packet (protocol headers and all) is contained in a socket buffer (`sk_buff`) structure. Socket buffers are introduced later in this chapter.

`int (*hard_header) (struct sk_buff *skb, struct net_device *dev, unsigned short type, void *daddr, void *saddr, unsigned len);`

Function (called before *hard_start_xmit*) that builds the hardware header from the source and destination hardware addresses that were previously retrieved; its job is to organize the information passed to it as arguments into an appropriate, device-specific hardware header. *eth_header* is the default function for Ethernet-like interfaces, and *ether_setup* assigns this field accordingly.

`int (*rebuild_header)(struct sk_buff *skb);`

Function used to rebuild the hardware header after ARP resolution completes but before a packet is transmitted. The default function used by Ethernet devices uses the ARP support code to fill the packet with missing information.

`void (*tx_timeout)(struct net_device *dev);`

Method called by the networking code when a packet transmission fails to complete within a reasonable period, on the assumption that an interrupt has been missed or the interface has locked up. It should handle the problem and resume packet transmission.

`struct net_device_stats *(*get_stats)(struct net_device *dev);`

Whenever an application needs to get statistics for the interface, this method is called. This happens, for example, when *ifconfig* or *netstat -i* is run. A sample implementation for *snull* is introduced in the section “Statistical Information.”

`int (*set_config)(struct net_device *dev, struct ifmap *map);`

Changes the interface configuration. This method is the entry point for configuring the driver. The I/O address for the device and its interrupt number can be changed at runtime using *set_config*. This capability can be used by the system administrator if the interface cannot be probed for. Drivers for modern hardware normally do not need to implement this method.

The remaining device operations are optional:

`int weight;`
`int (*poll)(struct net_device *dev; int *quota);`

Method provided by NAPI-compliant drivers to operate the interface in a polled mode, with interrupts disabled. NAPI (and the `weight` field) are covered in the section “Receive Interrupt Mitigation.”

`void (*poll_controller)(struct net_device *dev);`

Function that asks the driver to check for events on the interface in situations where interrupts are disabled. It is used for specific in-kernel networking tasks, such as remote consoles and kernel debugging over the network.

`int (*do_ioctl)(struct net_device *dev, struct ifreq *ifr, int cmd);`

Performs interface-specific *ioctl* commands. (Implementation of those commands is described in the section “Custom ioctl Commands.”) The corresponding field in `struct net_device` can be left as `NULL` if the interface doesn’t need any interface-specific commands.

`void (*set_multicast_list)(struct net_device *dev);`

Method called when the multicast list for the device changes and when the flags change. See the section “Multicast” for further details and a sample implementation.

`int (*set_mac_address)(struct net_device *dev, void *addr);`

Function that can be implemented if the interface supports the ability to change its hardware address. Many interfaces don’t support this ability at all. Others use the default *eth_mac_addr* implementation (from *drivers/net/net_init.c*). *eth_mac_addr* only copies the new address into `dev->dev_addr`, and it does so only if the interface is not running. Drivers that use *eth_mac_addr* should set the hardware MAC address from `dev->dev_addr` in their *open* method.

`int (*change_mtu)(struct net_device *dev, int new_mtu);`

Function that takes action if there is a change in the maximum transfer unit (MTU) for the interface. If the driver needs to do anything particular when the MTU is changed by the user, it should declare its own function; otherwise, the default does the right thing. *snull* has a template for the function if you are interested.

`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. Almost all Ethernet-like drivers can use the default *eth_header_cache* implementation.

`int (*header_cache_update) (struct hh_cache *hh, struct net_device *dev, unsigned char *haddr);`

Method that updates the destination address in the `hh_cache` structure in response to a change. Ethernet devices use *eth_header_cache_update*.

`int (*hard_header_parse) (struct sk_buff *skb, unsigned char *haddr);`

The *hard_header_parse* method extracts the source address from the packet contained in `skb`, copying it into the buffer at `haddr`. The return value from the function is the length of that address. Ethernet devices normally use *eth_header_parse*.

Utility Fields

The remaining `struct net_device` data fields are used by the interface to hold useful status information. Some of the fields are used by *ifconfig* and *netstat* to provide the user with information about the current configuration. Therefore, an interface should assign values to these fields:

`unsigned long trans_start;`
`unsigned long last_rx;`

Fields that hold a jiffies value. The driver is responsible for updating these values when transmission begins and when a packet is received, respectively. The `trans_start` value is used by the networking subsystem to detect transmitter lockups. `last_rx` is currently unused, but the driver should maintain this field anyway to be prepared for future use.

`int watchdog_timeo;`

The minimum time (in jiffies) that should pass before the networking layer decides that a transmission timeout has occurred and calls the driver's *tx_timeout* function.

`void *priv;`

The equivalent of `filp->private_data`. In modern drivers, this field is set by *alloc_netdev* and should not be accessed directly; use *netdev_priv* instead.

`struct dev_mc_list *mc_list;`
`int mc_count;`

Fields that handle multicast transmission. `mc_count` is the count of items in `mc_list`. See the section "Multicast" for further details.

`spinlock_t xmit_lock;`
`int xmit_lock_owner;`

The `xmit_lock` is used to avoid multiple simultaneous calls to the driver's *hard_start_xmit* function. `xmit_lock_owner` is the number of the CPU that has obtained `xmit_lock`. The driver should make no changes to these fields.

There are other fields in `struct net_device`, but they are not used by network drivers.

Opening and Closing

Our driver can probe for the interface at module load time or at kernel boot. Before the interface can carry packets, however, the kernel must open it and assign an address to it. The kernel opens or closes an interface in response to the *ifconfig* command.

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.

As far as the device is concerned, `ioctl(SIOCSIFADDR)` does nothing. No driver function is invoked—the task is device independent, and the kernel performs it. The latter command (`ioctl(SIOCSIFFLAGS)`), however, calls the *open* method for the device.

Similarly, when the interface is shut down, *ifconfig* uses `ioctl(SIOCSIFFLAGS)` to clear `IFF_UP`, and the *stop* method is called.

Both device methods return `0` in case of success and the usual negative value in case of error.

As far as the actual code is concerned, the driver has to perform many of the same tasks as the char and block drivers do. *open* requests any system resources it needs and tells the interface to come up; *stop* shuts down the interface and releases system resources. Network drivers must perform some additional steps at *open* time, however.

First, the hardware (MAC) address needs to be copied from the hardware device to `dev->dev_addr` before the interface can communicate with the outside world. The hardware address can then be copied to the device at open time. The *snull* software interface assigns it from within *open*; it just fakes a hardware number using an ASCII string of length `ETH_ALEN`, the length of Ethernet hardware addresses.

The *open* method should also start the interface's transmit queue (allowing it to accept packets for transmission) once it is ready to start sending data. The kernel provides a function to start the queue:

```
void netif_start_queue(struct net_device *dev);
```

The *open* code for *snull* looks like the following:

```
int snull_open(struct net_device *dev)
{
    /* request_region( ), request_irq( ), ....  (like fops->open) */

    /*
     * Assign the hardware address of the board: use "\0SNULx", where
     * x is 0 or 1. The first byte is '\0' to avoid being a multicast
     * address (the first byte of multicast addrs is odd).
     */
    memcpy(dev->dev_addr, "\0SNUL0", ETH_ALEN);
    if (dev == snull_devs[1])
```

```
        dev->dev_addr[ETH_ALEN-1]++; /* \0SNUL1 */
    netif_start_queue(dev);
    return 0;
}
```

As you can see, in the absence of real hardware, there is little to do in the *open* method. The same is true of the *stop* method; it just reverses the operations of *open*. For this reason, the function implementing *stop* is often called *close* or *release*.

```
int snull_release(struct net_device *dev)
{
    /* release ports, irq and such -- like fops->close */

    netif_stop_queue(dev); /* can't transmit any more */
    return 0;
}
```

The function:

```
void netif_stop_queue(struct net_device *dev);
```

is the opposite of *netif_start_queue*; it marks the device as being unable to transmit any more packets. The function must be called when the interface is closed (in the *stop* method) but can also be used to temporarily stop transmission, as explained in the next section.

Packet Transmission

The most important tasks performed by network interfaces are data transmission and reception. We start with transmission because it is slightly easier to understand.

Transmission refers to the act of sending a packet over a network link. Whenever the kernel needs to transmit a data packet, it calls the driver's *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`), whose definition is found in *<linux/skbuff.h>*. The structure gets its name from the Unix abstraction used to represent a network connection, the *socket*. Even if the interface has nothing to do with sockets, each network packet belongs to a socket in the higher network layers, and the input/output buffers of any socket are lists of `struct sk_buff` structures. The same `sk_buff` structure is used to host network data throughout all the Linux network subsystems, but a socket buffer is just a packet as far as the interface is concerned.

A pointer to `sk_buff` is usually called `skb`, and we follow this practice both in the sample code and in the text.

The socket buffer is a complex structure, and the kernel offers a number of functions to act on it. The functions are described later in the section "The Socket Buffers"; for now, a few basic facts about `sk_buff` are enough for us to write a working driver.

The socket buffer passed to *hard_start_xmit* contains the physical packet as it should appear on the media, complete with the transmission-level headers. The interface doesn't need to modify the data being transmitted. `skb->data` points to the packet being transmitted, and `skb->len` is its length in octets. This situation gets a little more complicated if your driver can handle scatter/gather I/O; we get to that in the section "Scatter/Gather I/O."

The *snull* packet transmission code follows; the physical transmission machinery has been isolated in another function, because every interface driver must implement it according to the specific hardware being driven:

```
int snull_tx(struct sk_buff *skb, struct net_device *dev)
{
    int len;
    char *data, shortpkt[ETH_ZLEN];
    struct snull_priv *priv = netdev_priv(dev);

    data = skb->data;
    len = skb->len;
    if (len < ETH_ZLEN) {
        memset(shortpkt, 0, ETH_ZLEN);
        memcpy(shortpkt, skb->data, skb->len);
        len = ETH_ZLEN;
        data = shortpkt;
    }
    dev->trans_start = jiffies; /* save the timestamp */

    /* Remember the skb, so we can free it at interrupt time */
    priv->skb = skb;

    /* actual deliver of data is device-specific, and not shown here */
    snull_hw_tx(data, len, dev);

    return 0; /* Our simple device can not fail */
}
```

The transmission function, thus, just performs some sanity checks on the packet and transmits the data through the hardware-related function. Do note, however, the care that is taken when the packet to be transmitted is shorter than the minimum length supported by the underlying media (which, for *snull*, is our virtual "Ethernet"). Many Linux network drivers (and those for other operating systems as well) have been found to leak data in such situations. Rather than create that sort of security vulnerability, we copy short packets into a separate array that we can explicitly zero-pad out to the full length required by the media. (We can safely put that data on the stack, since the minimum length—60 bytes—is quite small).

The return value from *hard_start_xmit* should be `0` on success; at that point, your driver has taken responsibility for the packet, should make its best effort to ensure that transmission succeeds, and must free the skb at the end. A nonzero return value indicates that the packet could not be transmitted at this time; the kernel will retry later. In this situation, your driver should stop the queue until whatever situation caused the failure has been resolved.

The "hardware-related" transmission function (*snull_hw_tx*) is omitted here since it is entirely occupied with implementing the trickery of the *snull* device, including manipulating the source and destination addresses, and has little of interest to authors of real network drivers. It is present, of course, in the sample source for those who want to go in and see how it works.

Controlling 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, however, 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. This is not an issue with *snull*, which does all of its work using the CPU, so packet transmission is complete before the transmission function returns.

Real hardware interfaces, on the other hand, transmit packets asynchronously and have a limited amount of memory available to store outgoing packets. When that memory is exhausted (which, for some hardware, happens with a single outstanding packet to transmit), the driver needs 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*, the function introduced earlier to stop the queue. 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. To do so, it should call:

```
void netif_wake_queue(struct net_device *dev);
```

This function is just like *netif_start_queue*, except that it also pokes the networking system to make it start transmitting packets again.

Most modern network hardware maintains an internal queue with multiple packets to transmit; in this way it can get the best performance from the network. Network drivers for these devices must support having multiple transmisions outstanding at any given time, but device memory can fill up whether or not the hardware supports multiple outstanding transmissions. Whenever device memory fills to the point that there is no room for the largest possible packet, the driver should stop the queue until space becomes available again.

If you must disable packet transmission from anywhere other than your *hard_start_xmit* function (in response to a reconfiguration request, perhaps), the function you want to use is:

```
void netif_tx_disable(struct net_device *dev);
```

This function behaves much like *netif_stop_queue*, but it also ensures that, when it returns, your *hard_start_xmit* method is not running on another CPU. The queue can be restarted with *netif_wake_queue*, as usual.

Transmission Timeouts

Most drivers that deal with real hardware have to be prepared for that hardware to fail to respond occasionally. Interfaces can forget what they are doing, or the system can lose an interrupt. This sort of problem is common with some devices designed to run on personal computers.

Many drivers handle this problem by setting timers; if the operation has not completed by the time the timer expires, something is wrong. The network system, as it happens, is essentially a complicated assembly of state machines controlled by a mass of timers. As such, the networking code is in a good position to detect transmission timeouts as part of its regular operation.

Thus, network drivers need not worry about detecting such problems themselves. Instead, they need only set a timeout period, which goes in the `watchdog_timeo` field of the `net_device` structure. This period, which is in jiffies, should be long enough to account for normal transmission delays (such as collisions caused by congestion on the network media).

If the current system time exceeds the device's `trans_start` time by at least the timeout period, the networking layer eventually calls the driver's *tx_timeout* method. That method's job is to do whatever is needed to clear up the problem and to ensure the proper completion of any transmissions that were already in progress. It is important, in particular, that the driver not lose track of any socket buffers that have been entrusted to it by the networking code.

snull has the ability to simulate transmitter lockups, which is controlled by two load-time parameters:

```
static int lockup = 0;
module_param(lockup, int, 0);

static int timeout = SNULL_TIMEOUT;
module_param(timeout, int, 0);
```

If the driver is loaded with the parameter `lockup=n`, a lockup is simulated once every `n` packets transmitted, and the `watchdog_timeo` field is set to the given `timeout` value. When simulating lockups, *snull* also calls *netif_stop_queue* to prevent other transmission attempts from occurring.

The *snull* transmission timeout handler looks like this:

```
void snull_tx_timeout (struct net_device *dev)
{
    struct snull_priv *priv = netdev_priv(dev);
```

```
        PDEBUG("Transmit timeout at %ld, latency %ld\n", jiffies,
                jiffies - dev->trans_start);
            /* Simulate a transmission interrupt to get things moving */
        priv->status = SNULL_TX_INTR;
        snull_interrupt(0, dev, NULL);
        priv->stats.tx_errors++;
        netif_wake_queue(dev);
        return;
    }
```

When a transmission timeout happens, the driver must mark the error in the interface statistics and arrange for the device to be reset to a sane state so that new packets can be transmitted. When a timeout happens in *snull*, the driver calls *snull_interrupt* to fill in the "missing" interrupt and restarts the transmit queue with *netif_wake_queue*.

Scatter/Gather I/O

The process of creating a packet for transmission on the network involves assembling multiple pieces. Packet data must often be copied in from user space, and the headers used by various levels of the network stack must be added as well. This assembly can require a fair amount of data copying. If, however, the network interface that is destined to transmit the packet can perform scatter/gather I/O, the packet need not be assembled into a single chunk, and much of that copying can be avoided. Scatter/gather I/O also enables "zero-copy" transmission of network data directly from user-space buffers.

The kernel does not pass scattered packets to your *hard_start_xmit* method unless the `NETIF_F_SG` bit has been set in the `features` field of your device structure. If you have set that flag, you need to look at a special "shared info" field within the skb to see whether the packet is made up of a single fragment or many and to find the scattered fragments if need be. A special macro exists to access this information; it is called *skb_shinfo*. The first step when transmitting potentially fragmented packets usually looks something like this:

```
    if (skb_shinfo(skb)->nr_frags == 0) {
        /* Just use skb->data and skb->len as usual */
    }
```

The `nr_frags` field tells how many fragments have been used to build the packet. If it is `0`, the packet exists in a single piece and can be accessed via the `data` field as usual. If, however, it is nonzero, your driver must pass through and arrange to transfer each individual fragment. The `data` field of the skb structure points conveniently to the first fragment (as compared to the full packet, as in the unfragmented case). The length of the fragment must be calculated by subtracting `skb->data_len` from `skb->len` (which still contains the length of the full packet). The remaining fragments are to be found in an array called `frags` in the shared information structure; each entry in `frags` is an `skb_frag_struct` structure:

```
    struct skb_frag_struct {
        struct page *page;
        __u16 page_offset;
        __u16 size;
    };
```

As you can see, we are once again dealing with page structures, rather than kernel virtual addresses. Your driver should loop through the fragments, mapping each for a DMA transfer and not forgetting the first fragment, which is pointed to by the skb directly. Your hardware, of course, must assemble the fragments and transmit them as a single packet. Note that, if you have set the NETIF_F_HIGHDMA feature flag, some or all of the fragments may be located in high memory.

Packet Reception

Receiving data from the network is trickier than transmitting it, because an sk_buff must be allocated and handed off to the upper layers from within an atomic context. There are two modes of packet reception that may be implemented by network drivers: interrupt driven and polled. Most drivers implement the interrupt-driven technique, and that is the one we cover first. Some drivers for high-bandwidth adapters may also implement the polled technique; we look at this approach in the section "Receive Interrupt Mitigation."

The implementation of *snull* separates the "hardware" details from the device-independent housekeeping. Therefore, the function *snull_rx* is called from the *snull* "interrupt" handler after the hardware has received the packet, and it is already in the computer's memory. *snull_rx* receives a pointer to the data and the length of the packet; its sole responsibility is to send the packet and some additional information to the upper layers of networking code. This code is independent of the way the data pointer and length are obtained.

```
void snull_rx(struct net_device *dev, struct snull_packet *pkt)
{
    struct sk_buff *skb;
    struct snull_priv *priv = netdev_priv(dev);

    /*
     * The packet has been retrieved from the transmission
     * medium. Build an skb around it, so upper layers can handle it
     */
    skb = dev_alloc_skb(pkt->datalen + 2);
    if (!skb) {
        if (printk_ratelimit( ))
            printk(KERN_NOTICE "snull rx: low on mem - packet dropped\n");
        priv->stats.rx_dropped++;
        goto out;
    }
    memcpy(skb_put(skb, pkt->datalen), pkt->data, pkt->datalen);

    /* Write metadata, and then pass to the receive level */
    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 += pkt->datalen;
    netif_rx(skb);
  out:
    return;
}
```

The function is sufficiently general to act as a template for any network driver, but some explanation is necessary before you can reuse this code fragment with confidence.

The first step is to allocate a buffer to hold the packet. Note that the buffer allocation function (*dev_alloc_skb*) needs to know the data length. The information is used by the function to allocate space for the buffer. *dev_alloc_skb* calls *kmalloc* with atomic priority, so it can be used safely at interrupt time. The kernel offers other interfaces to socket-buffer allocation, but they are not worth introducing here; socket buffers are explained in detail in the section “The Socket Buffers.”

Of course, the return value from *dev_alloc_skb* must be checked, and *snull* does so. We call *printk_ratelimit* before complaining about failures, however. Generating hundreds or thousands of console messages per second is a good way to bog down the system entirely and hide the real source of problems; *printk_ratelimit* helps prevent that problem by returning `0` when too much output has gone to the console, and things need to be slowed down a bit.

Once there is a valid `skb` pointer, the packet data is copied into the buffer by calling *memcpy*; the *skb_put* function updates the end-of-data pointer in the buffer and returns a pointer to the newly created space.

If you are writing a high-performance driver for an interface that can do full bus-mastering I/O, there is a possible optimization that is worth considering here. Some drivers allocate socket buffers for incoming packets prior to their reception, then instruct the interface to place the packet data directly into the socket buffer’s space. The networking layer cooperates with this strategy by allocating all socket buffers in DMA-capable space (which may be in high memory if your device has the `NETIF_F_HIGHDMA` feature flag set). Doing things this way avoids the need for a separate copy operation to fill the socket buffer, but requires being careful with buffer sizes because you won’t know in advance how big the incoming packet is. The implementation of a *change_mtu* method is also important in this situation, since it allows the driver to respond to a change in the maximum packet size.

The network layer needs to have some information spelled out before it can make sense of the packet. To this end, the `dev` and `protocol` fields must be assigned before the buffer is passed upstairs. The Ethernet support code exports a helper function (*eth_type_trans*), which finds an appropriate value to put into `protocol`. Then we need to specify how checksumming is to be performed or has been performed on the packet (*snull* does not need to perform any checksums). The possible policies for `skb->ip_summed` are:

`CHECKSUM_HW`
: The device has already performed checksums in hardware. An example of a hardware checksum is the SPARC HME interface.

`CHECKSUM_NONE`
: Checksums have not yet been verified, and the task must be accomplished by system software. This is the default in newly allocated buffers.

`CHECKSUM_UNNECESSARY`
: Don't do any checksums. This is the policy in *snull* and in the loopback interface.

You may be wondering why the checksum status must be specified here when we have already set a flag in the `features` field of our `net_device` structure. The answer is that the `features` flag tells the kernel about how our device treats outgoing packets. It is not used for incoming packets, which must, instead, be marked individually.

Finally, the driver updates its statistics counter to record that a packet has been received. The statistics structure is made up of several fields; the most important are `rx_packets`, `rx_bytes`, `tx_packets`, and `tx_bytes`, which contain the number of packets received and transmitted and the total number of octets transferred. All the fields are thoroughly described in the section "Statistical Information."

The last step in packet reception is performed by *netif_rx*, which hands off the socket buffer to the upper layers. *netif_rx* actually returns an integer value; `NET_RX_SUCCESS` (`0`) means that the packet was successfully received; any other value indicates trouble. There are three return values (`NET_RX_CN_LOW`, `NET_RX_CN_MOD`, and `NET_RX_CN_HIGH`) that indicate increasing levels of congestion in the networking subsystem; `NET_RX_DROP` means the packet was dropped. A driver could use these values to stop feeding packets into the kernel when congestion gets high, but, in practice, most drivers ignore the return value from *netif_rx*. If you are writing a driver for a high-bandwidth device and wish to do the right thing in response to congestion, the best approach is to implement NAPI, which we get to after a quick discussion of interrupt handlers.

The Interrupt Handler

Most hardware interfaces are controlled by means of an interrupt handler. The hardware interrupts the processor to signal one of two possible events: a new packet has arrived or transmission of an outgoing packet is complete. Network interfaces can also generate interrupts to signal errors, link status changes, and so on.

The usual interrupt routine can tell the difference between a new-packet-arrived interrupt and a done-transmitting notification by checking a status register found on the physical device. The *snull* interface works similarly, but its status word is implemented in software and lives in dev->priv. The interrupt handler for a network interface looks like this:

```
static void snull_regular_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    int statusword;
    struct snull_priv *priv;
    struct snull_packet *pkt = NULL;
    /*
     * As usual, check the "device" pointer to be sure it is
     * really interrupting.
     * Then assign "struct device *dev"
     */
    struct net_device *dev = (struct net_device *)dev_id;
    /* ... and check with hw if it's really ours */

    /* paranoid */
    if (!dev)
        return;

    /* Lock the device */
    priv = netdev_priv(dev);
    spin_lock(&priv->lock);

    /* retrieve statusword: real netdevices use I/O instructions */
    statusword = priv->status;
    priv->status = 0;
    if (statusword & SNULL_RX_INTR) {
        /* send it to snull_rx for handling */
        pkt = priv->rx_queue;
        if (pkt) {
            priv->rx_queue = pkt->next;
            snull_rx(dev, pkt);
        }
    }
    if (statusword & SNULL_TX_INTR) {
        /* a transmission is over: free the skb */
        priv->stats.tx_packets++;
        priv->stats.tx_bytes += priv->tx_packetlen;
        dev_kfree_skb(priv->skb);
    }

    /* Unlock the device and we are done */
    spin_unlock(&priv->lock);
    if (pkt) snull_release_buffer(pkt); /* Do this outside the lock! */
    return;
}
```

The handler's first task is to retrieve a pointer to the correct struct net_device. This pointer usually comes from the dev_id pointer received as an argument.

The interesting part of this handler deals with the "transmission done" situation. In this case, the statistics are updated, and *dev_kfree_skb* is called to return the (no longer needed) socket buffer to the system. There are, actually, three variants of this function that may be called:

`dev_kfree_skb(struct sk_buff *skb);`
: This version should be called when you know that your code will not be running in interrupt context. Since *snull* has no actual hardware interrupts, this is the version we use.

`dev_kfree_skb_irq(struct sk_buff *skb);`
: If you know that you will be freeing the buffer in an interrupt handler, use this version, which is optimized for that case.

`dev_kfree_skb_any(struct sk_buff *skb);`
: This is the version to use if the relevant code could be running in either interrupt or noninterrupt context.

Finally, if your driver has temporarily stopped the transmission queue, this is usually the place to restart it with *netif_wake_queue*.

Packet reception, in contrast to transmission, doesn't need any special interrupt handling. Calling *snull_rx* (which we have already seen) is all that's required.

Receive Interrupt Mitigation

When a network driver is written as we have described above, the processor is interrupted for every packet received by your interface. In many cases, that is the desired mode of operation, and it is not a problem. High-bandwidth interfaces, however, can receive thousands of packets per second. With that sort of interrupt load, the overall performance of the system can suffer.

As a way of improving the performance of Linux on high-end systems, the networking subsystem developers have created an alternative interface (called NAPI)[*] based on polling. "Polling" can be a dirty word among driver developers, who often see polling techniques as inelegant and inefficient. Polling is inefficient, however, only if the interface is polled when there is no work to do. When the system has a high-speed interface handling heavy traffic, there is *always* more packets to process. There is no need to interrupt the processor in such situations; it is enough that the new packets be collected from the interface every so often.

Stopping receive interrupts can take a substantial amount of load off the processor. NAPI-compliant drivers can also be told not to feed packets into the kernel if those packets are just dropped in the networking code due to congestion, which can also help performance when that help is needed most. For various reasons, NAPI drivers are also less likely to reorder packets.

[*] NAPI stands for "new API"; the networking hackers are better at creating interfaces than naming them.

Not all devices can operate in the NAPI mode, however. A NAPI-capable interface must be able to store several packets (either on the card itself, or in an in-memory DMA ring). The interface should be capable of disabling interrupts for received packets, while continuing to interrupt for successful transmissions and other events. There are other subtle issues that can make writing a NAPI-compliant driver harder; see *Documentation/networking/NAPI_HOWTO.txt* in the kernel source tree for the details.

Relatively few drivers implement the NAPI interface. If you are writing a driver for an interface that may generate a huge number of interrupts, however, taking the time to implement NAPI may well prove worthwhile.

The *snull* driver, when loaded with the `use_napi` parameter set to a nonzero value, operates in the NAPI mode. At initialization time, we have to set up a couple of extra `struct net_device` fields:

```
if (use_napi) {
    dev->poll        = snull_poll;
    dev->weight      = 2;
}
```

The `poll` field must be set to your driver's polling function; we look at *snull_poll* shortly. The `weight` field describes the relative importance of the interface: how much traffic should be accepted from the interface when resources are tight. There are no strict rules for how the `weight` parameter should be set; by convention, 10 MBps Ethernet interfaces set `weight` to `16`, while faster interfaces use `64`. You should not set `weight` to a value greater than the number of packets your interface can store. In *snull*, we set the `weight` to two as a way of demonstrating deferred packet reception.

The next step in the creation of a NAPI-compliant driver is to change the interrupt handler. When your interface (which should start with receive interrupts enabled) signals that a packet has arrived, the interrupt handler should *not* process that packet. Instead, it should disable further receive interrupts and tell the kernel that it is time to start polling the interface. In the *snull* "interrupt" handler, the code that responds to packet reception interrupts has been changed to the following:

```
if (statusword & SNULL_RX_INTR) {
    snull_rx_ints(dev, 0);  /* Disable further interrupts */
    netif_rx_schedule(dev);
}
```

When the interface tells us that a packet is available, the interrupt handler leaves it in the interface; all that needs to happen at this point is a call to *netif_rx_schedule*, which causes our *poll* method to be called at some future point.

The *poll* method has this prototype:

```
int (*poll)(struct net_device *dev, int *budget);
```

The *snull* implementation of the *poll* method looks like this:

```
static int snull_poll(struct net_device *dev, int *budget)
{
    int npackets = 0, quota = min(dev->quota, *budget);
    struct sk_buff *skb;
    struct snull_priv *priv = netdev_priv(dev);
    struct snull_packet *pkt;

    while (npackets < quota && priv->rx_queue) {
        pkt = snull_dequeue_buf(dev);
        skb = dev_alloc_skb(pkt->datalen + 2);
        if (! skb) {
            if (printk_ratelimit( ))
                printk(KERN_NOTICE "snull: packet dropped\n");
            priv->stats.rx_dropped++;
            snull_release_buffer(pkt);
            continue;
        }
        memcpy(skb_put(skb, pkt->datalen), pkt->data, pkt->datalen);
        skb->dev = dev;
        skb->protocol = eth_type_trans(skb, dev);
        skb->ip_summed = CHECKSUM_UNNECESSARY; /* don't check it */
        netif_receive_skb(skb);

            /* Maintain stats */
        npackets++;
        priv->stats.rx_packets++;
        priv->stats.rx_bytes += pkt->datalen;
        snull_release_buffer(pkt);
    }
    /* If we processed all packets, we're done; tell the kernel and reenable ints */
    *budget -= npackets;
    dev->quota -= npackets;
    if (! priv->rx_queue) {
        netif_rx_complete(dev);
        snull_rx_ints(dev, 1);
        return 0;
    }
    /* We couldn't process everything. */
    return 1;
}
```

The central part of the function is concerned with the creation of an skb holding the packet; this code is the same as what we saw in *snull_rx* before. A number of things are different, however:

- The budget parameter provides a maximum number of packets that we are allowed to pass into the kernel. Within the device structure, the quota field gives another maximum; the *poll* method must respect the lower of the two limits. It should also decrement both dev->quota and *budget by the number of packets actually received. The budget value is a maximum number of packets that the current CPU can receive from all interfaces, while quota is a per-interface value that usually starts out as the weight assigned to the interface at initialization time.

- Packets should be fed to the kernel with *netif_receive_skb*, rather than *netif_rx*.
- If the *poll* method is able to process all of the available packets within the limits given to it, it should re-enable receive interrupts, call *netif_rx_complete* to turn off polling, and return 0. A return value of 1 indicates that there are packets remaining to be processed.

The networking subsystem guarantees that any given device's *poll* method will not be called concurrently on more than one processor. Calls to *poll* can still happen concurrently with calls to your other device methods, however.

Changes in Link State

Network connections, by definition, deal with the world outside the local system. Therefore, they are often affected by outside events, and they can be transient things. The networking subsystem needs to know when network links go up or down, and it provides a few functions that the driver may use to convey that information.

Most networking technologies involving an actual, physical connection provide a *carrier* state; the presence of the carrier means that the hardware is present and ready to function. Ethernet adapters, for example, sense the carrier signal on the wire; when a user trips over the cable, that carrier vanishes, and the link goes down. By default, network devices are assumed to have a carrier signal present. The driver can change that state explicitly, however, with these functions:

```
void netif_carrier_off(struct net_device *dev);
void netif_carrier_on(struct net_device *dev);
```

If your driver detects a lack of carrier on one of its devices, it should call *netif_carrier_off* to inform the kernel of this change. When the carrier returns, *netif_carrier_on* should be called. Some drivers also call *netif_carrier_off* when making major configuration changes (such as media type); once the adapter has finished resetting itself, the new carrier is detected and traffic can resume.

An integer function also exists:

```
int netif_carrier_ok(struct net_device *dev);
```

This can be used to test the current carrier state (as reflected in the device structure).

The Socket Buffers

We've now covered most of the issues related to network interfaces. What's still missing is some more detailed discussion of the sk_buff structure. The structure is at the core of the network subsystem of the Linux kernel, and we now introduce both the main fields of the structure and the functions used to act on it.

Although there is no strict need to understand the internals of `sk_buff`, the ability to look at its contents can be helpful when you are tracking down problems and when you are trying to optimize your code. For example, if you look in *loopback.c*, you'll find an optimization based on knowledge of the `sk_buff` internals. The usual warning applies here: if you write code that takes advantage of knowledge of the `sk_buff` structure, you should be prepared to see it break with future kernel releases. Still, sometimes the performance advantages justify the additional maintenance cost.

We are not going to describe the whole structure here, just the fields that might be used from within a driver. If you want to see more, you can look at *<linux/skbuff.h>*, where the structure is defined and the functions are prototyped. Additional details about how the fields and functions are used can be easily retrieved by grepping in the kernel sources.

The Important Fields

The fields introduced here are the ones a driver might need to access. They are listed in no particular order.

`struct net_device *dev;`

The device receiving or sending this buffer.

`union { /* ... */ } h;`
`union { /* ... */ } nh;`
`union { /*... */} mac;`

Pointers to the various levels of headers contained within the packet. Each field of the union is a pointer to a different type of data structure. `h` hosts pointers to transport layer headers (for example, `struct tcphdr *th`); `nh` includes network layer headers (such as `struct iphdr *iph`); and `mac` collects pointers to link-layer headers (such as `struct ethdr *ethernet`).

If your driver needs to look at the source and destination addresses of a TCP packet, it can find them in `skb->h.th`. See the header file for the full set of header types that can be accessed in this way.

Note that network drivers are responsible for setting the `mac` pointer for incoming packets. This task is normally handled by *eth_type_trans*, but non-Ethernet drivers have to set `skb->mac.raw` directly, as shown in the section "Non-Ethernet Headers."

`unsigned char *head;`
`unsigned char *data;`
`unsigned char *tail;`
`unsigned char *end;`

Pointers used to address the data in the packet. `head` points to the beginning of the allocated space, `data` is the beginning of the valid octets (and is usually slightly greater than `head`), `tail` is the end of the valid octets, and `end` points to the maximum address tail can reach. Another way to look at it is that the *available* buffer space is skb->end - skb->head, and the *currently used* data space is skb->tail - skb->data.

unsigned int len;
unsigned int data_len;
: len is the full length of the data in the packet, while data_len is the length of the portion of the packet stored in separate fragments. The data_len field is 0 unless scatter/gather I/O is being used.

unsigned char ip_summed;
: The checksum policy for this packet. The field is set by the driver on incoming packets, as described in the section "Packet Reception."

unsigned char pkt_type;
: Packet classification used in its delivery. The driver is responsible for setting it to PACKET_HOST (this packet is for me), PACKET_OTHERHOST (no, this packet is not for me), PACKET_BROADCAST, or PACKET_MULTICAST. Ethernet drivers don't modify pkt_type explicitly because *eth_type_trans* does it for them.

shinfo(struct sk_buff *skb);
unsigned int shinfo(skb)->nr_frags;
skb_frag_t shinfo(skb)->frags;
: For performance reasons, some skb information is stored in a separate structure that appears immediately after the skb in memory. This "shared info" (so called because it can be shared among copies of the skb within the networking code) must be accessed via the *shinfo* macro. There are several fields in this structure, but most of them are beyond the scope of this book. We saw nr_frags and frags in the section "Scatter/Gather I/O."

The remaining fields in the structure are not particularly interesting. They are used to maintain lists of buffers, to account for memory belonging to the socket that owns the buffer, and so on.

Functions Acting on Socket Buffers

Network devices that use an sk_buff structure act on it by means of the official interface functions. Many functions operate on socket buffers; here are the most interesting ones:

struct sk_buff *alloc_skb(unsigned int len, int priority);
struct sk_buff *dev_alloc_skb(unsigned int len);
: Allocate a buffer. The *alloc_skb* function allocates a buffer and initializes both skb->data and skb->tail to skb->head. The *dev_alloc_skb* function is a shortcut that calls *alloc_skb* with GFP_ATOMIC priority and reserves some space between skb->head and skb->data. This data space is used for optimizations within the network layer and should not be touched by the driver.

```
void kfree_skb(struct sk_buff *skb);
void dev_kfree_skb(struct sk_buff *skb);
void dev_kfree_skb_irq(struct sk_buff *skb);
void dev_kfree_skb_any(struct sk_buff *skb);
```

Free a buffer. The *kfree_skb* call is used internally by the kernel. A driver should use one of the forms of *dev_kfree_skb* instead: *dev_kfree_skb* for noninterrupt context, *dev_kfree_skb_irq* for interrupt context, or *dev_kfree_skb_any* for code that can run in either context.

```
unsigned char *skb_put(struct sk_buff *skb, int len);
unsigned char *__skb_put(struct sk_buff *skb, int len);
```

Update the `tail` and `len` fields of the `sk_buff` structure; they are used to add data to the end of the buffer. Each function's return value is the previous value of `skb->tail` (in other words, it points to the data space just created). Drivers can use the return value to copy data by invoking `memcpy(skb_put(...), data, len)` or an equivalent. The difference between the two functions is that *skb_put* checks to be sure that the data fits in the buffer, whereas *__skb_put* omits the check.

```
unsigned char *skb_push(struct sk_buff *skb, int len);
unsigned char *__skb_push(struct sk_buff *skb, int len);
```

Functions to decrement `skb->data` and increment `skb->len`. They are similar to *skb_put*, except that data is added to the beginning of the packet instead of the end. The return value points to the data space just created. The functions are used to add a hardware header before transmitting a packet. Once again, *__skb_push* differs in that it does not check for adequate available space.

```
int skb_tailroom(struct sk_buff *skb);
```

Returns the amount of space available for putting data in the buffer. If a driver puts more data into the buffer than it can hold, the system panics. Although you might object that a *printk* would be sufficient to tag the error, memory corruption is so harmful to the system that the developers decided to take definitive action. In practice, you shouldn't need to check the available space if the buffer has been correctly allocated. Since drivers usually get the packet size before allocating a buffer, only a severely broken driver puts too much data in the buffer, and a panic might be seen as due punishment.

```
int skb_headroom(struct sk_buff *skb);
```

Returns the amount of space available in front of `data`, that is, how many octets one can "push" to the buffer.

```
void skb_reserve(struct sk_buff *skb, int len);
```

Increments both `data` and `tail`. The function can be used to reserve headroom before filling the buffer. Most Ethernet interfaces reserve two bytes in front of the packet; thus, the IP header is aligned on a 16-byte boundary, after a 14-byte Ethernet header. *snull* does this as well, although the instruction was not shown in "Packet Reception" to avoid introducing extra concepts at that point.

`unsigned char *skb_pull(struct sk_buff *skb, int len);`

Removes data from the head of the packet. The driver won't need to use this function, but it is included here for completeness. It decrements `skb->len` and increments `skb->data`; this is how the hardware header (Ethernet or equivalent) is stripped from the beginning of incoming packets.

`int skb_is_nonlinear(struct sk_buff *skb);`

Returns a true value if this skb is separated into multiple fragments for scatter/gather I/O.

`int skb_headlen(struct sk_buff *skb);`

Returns the length of the first segment of the skb (that part pointed to by `skb->data`).

`void *kmap_skb_frag(skb_frag_t *frag);`
`void kunmap_skb_frag(void *vaddr);`

If you must directly access fragments in a nonlinear skb from within the kernel, these functions map and unmap them for you. An atomic kmap is used, so you cannot have more than one fragment mapped at a time.

The kernel defines several other functions that act on socket buffers, but they are meant to be used in higher layers of networking code, and the driver doesn't need them.

MAC Address Resolution

An interesting issue with Ethernet communication is how to associate the MAC addresses (the interface's unique hardware ID) with the IP number. Most protocols have a similar problem, but we concentrate on the Ethernet-like case here. We try to offer a complete description of the issue, so we show three situations: ARP, Ethernet headers without ARP (such as *plip*), and non-Ethernet headers.

Using ARP with Ethernet

The usual way to deal with address resolution is by using the Address Resolution Protocol (ARP). 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 MAC addresses; *ether_setup* assigns the correct device methods to `dev->hard_header` and `dev->rebuild_header`.

Although the kernel normally handles the details of address resolution (and caching of the results), it calls upon the interface driver to help in the building of the packet. After all, the driver knows about the details of the physical layer header, while the authors of the networking code have tried to insulate the rest of the kernel from that knowledge. To this end, the kernel calls the driver's *hard_header* method to lay out the packet with the results of the ARP query. Normally, Ethernet driver writers need not know about this process—the common Ethernet code takes care of everything.

Overriding ARP

Simple point-to-point network interfaces, such as *plip*, might benefit from using Ethernet headers, while avoiding the overhead of sending ARP packets back and forth. The sample code in *snull* also falls into this class of network devices. *snull* cannot use ARP because the driver changes IP addresses in packets being transmitted, and ARP packets exchange IP addresses as well. Although we could have implemented a simple ARP reply generator with little trouble, it is more illustrative to show how to handle physical-layer headers directly.

If your device wants to use the usual hardware header without running ARP, you need to override the default *dev->hard_header* method. This is how *snull* implements it, as a very short function:

```
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;   /* dest is us xor 1 */
    return (dev->hard_header_len);
}
```

The function simply takes the information provided by the kernel and formats it into a standard Ethernet header. It also toggles a bit in the destination Ethernet address, for reasons described later.

When a packet is received by the interface, the hardware header is used in a couple of ways by *eth_type_trans*. We have already seen this call in *snull_rx*:

```
skb->protocol = eth_type_trans(skb, dev);
```

The function extracts the protocol identifier (`ETH_P_IP`, in this case) from the Ethernet header; it also assigns `skb->mac.raw`, removes the hardware header from packet data (with *skb_pull*), and sets `skb->pkt_type`. This last item defaults to `PACKET_HOST` at `skb` allocation (which indicates that the packet is directed to this host), and *eth_type_trans* changes it to reflect the Ethernet destination address: if that address does not match the address of the interface that received it, the `pkt_type` field is set to `PACKET_OTHERHOST`. Subsequently, unless the interface is in promiscuous mode or packet forwarding is enabled in the kernel, *netif_rx* drops any packet of type `PACKET_OTHERHOST`. For this reason, *snull_header* is careful to make the destination hardware address match that of the “receiving” interface.

If your interface is a point-to-point link, you won't want to receive unexpected multicast packets. To avoid this problem, remember that a destination address whose first octet has `0` as the least significant bit (LSB) is directed to a single host (i.e., it is either `PACKET_HOST` or `PACKET_OTHERHOST`). The *plip* driver uses `0xfc` as the first octet of its hardware address, while *snull* uses `0x00`. Both addresses result in a working Ethernet-like point-to-point link.

Non-Ethernet Headers

We have just seen that the hardware header contains some information in addition to the destination address, the most important being the communication protocol. We now describe how hardware headers can be used to encapsulate relevant information. If you need to know the details, you can extract them from the kernel sources or the technical documentation for the particular transmission medium. Most driver writers are able to ignore this discussion and just use the Ethernet implementation.

It's worth noting that not all information has to be provided by every protocol. A point-to-point link such as *plip* or *snull* could avoid transferring the whole Ethernet header without losing generality. The *hard_header* device method, shown earlier as implemented by *snull_header*, receives the delivery information—both protocol-level and hardware addresses—from the kernel. It also receives the 16-bit protocol number in the `type` argument; IP, for example, is identified by `ETH_P_IP`. The driver is expected to correctly deliver both the packet data and the protocol number to the receiving host. A point-to-point link could omit addresses from its hardware header, transferring only the protocol number, because delivery is guaranteed independent of the source and destination addresses. An IP-only link could even avoid transmitting any hardware header whatsoever.

When the packet is picked up at the other end of the link, the receiving function in the driver should correctly set the fields `skb->protocol`, `skb->pkt_type`, and `skb->mac.raw`.

`skb->mac.raw` is a char pointer used by the address-resolution mechanism implemented in higher layers of the networking code (for instance, *net/ipv4/arp.c*). It must point to a machine address that matches `dev->type`. The possible values for the device type are defined in *<linux/if_arp.h>*; Ethernet interfaces use `ARPHRD_ETHER`. For example, here is how *eth_type_trans* deals with the Ethernet header for received packets:

```
skb->mac.raw = skb->data;
skb_pull(skb, dev->hard_header_len);
```

In the simplest case (a point-to-point link with no headers), `skb->mac.raw` can point to a static buffer containing the hardware address of this interface, `protocol` can be set to `ETH_P_IP`, and `packet_type` can be left with its default value of `PACKET_HOST`.

Because every hardware type is unique, it is hard to give more specific advice than already discussed. The kernel is full of examples, however. See, for example, the

AppleTalk driver (*drivers/net/appletalk/cops.c*), the infrared drivers (such as *drivers/net/irda/smc_ircc.c*), or the PPP driver (*drivers/net/ppp_generic.c*).

Custom ioctl Commands

We have seen that the *ioctl* system call is implemented for sockets; `SIOCSIFADDR` and `SIOCSIFMAP` are examples of "socket *ioctls*." Now let's see how the third argument of the system call is used by networking code.

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 *sock_ioctl* function directly invokes a protocol-specific function (where "protocol" refers to the main network protocol being used, for example, IP or AppleTalk).

Any *ioctl* command that is not recognized by the protocol layer is passed to the device layer. 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 extra argument to `SIOCSIFMAP`, although defined as `ifmap`, is just a field of `ifreq`.

In addition to using the standardized calls, each interface can define its own *ioctl* commands. The *plip* interface, for example, allows the interface to modify its internal timeout values via *ioctl*. The *ioctl* implementation for sockets recognizes 16 commands as private to the interface: `SIOCDEVPRIVATE` through `SIOCDEVPRIVATE+15`.[*]

When one of these commands is recognized, `dev->do_ioctl` is called in the relevant interface driver. The function receives the same `struct ifreq *` pointer that the general-purpose *ioctl* function uses:

```
int (*do_ioctl)(struct net_device *dev, struct ifreq *ifr, int cmd);
```

The `ifr` pointer points to a kernel-space address that holds a copy of the structure passed by the user. After *do_ioctl* returns, the structure is copied back to user space; Therefore, the driver can use the private commands to both receive and return data.

The device-specific commands can choose to use the fields in `struct ifreq`, but they already convey a standardized meaning, and it's unlikely that the driver can adapt the structure to its needs. The field `ifr_data` is a `caddr_t` item (a pointer) that is meant to be used for device-specific needs. The driver and the program used to invoke its *ioctl* commands should agree about the use of `ifr_data`. For example, *pppstats* uses device-specific commands to retrieve information from the *ppp* interface driver.

[*] Note that, according to <*linux/sockios.h*>, the `SIOCDEVPRIVATE` commands are deprecated. What should replace them is not clear, however, and numerous in-tree drivers still use them.

It's not worth showing an implementation of *do_ioctl* here, but with the information in this chapter and the kernel examples, you should be able to write one when you need it. Note, however, that the *plip* implementation uses `ifr_data` incorrectly and should not be used as an example for an *ioctl* implementation.

Statistical Information

The last method a driver needs is *get_stats*. This method returns a pointer to the statistics for the device. Its implementation is pretty easy; the one shown works even when several interfaces are managed by the same driver, because the statistics are hosted within the device data structure.

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

The real work needed to return meaningful statistics is distributed throughout the driver, where the various fields are updated. The following list shows the most interesting fields in `struct net_device_stats`:

`unsigned long rx_packets;`
`unsigned long tx_packets;`
: The total number of incoming and outgoing packets successfully transferred by the interface.

`unsigned long rx_bytes;`
`unsigned long tx_bytes;`
: The number of bytes received and transmitted by the interface.

`unsigned long rx_errors;`
`unsigned long tx_errors;`
: The number of erroneous receptions and transmissions. There's no end of things that can go wrong with packet transmission, and the `net_device_stats` structure includes six counters for specific receive errors and five for transmit errors. See *<linux/netdevice.h>* for the full list. If possible, your driver should maintain detailed error statistics, because they can be most helpful to system administrators trying to track down a problem.

`unsigned long rx_dropped;`
`unsigned long tx_dropped;`
: The number of packets dropped during reception and transmission. Packets are dropped when there's no memory available for packet data. `tx_dropped` is rarely used.

`unsigned long collisions;`
The number of collisions due to congestion on the medium.

`unsigned long multicast;`
The number of multicast packets received.

It is worth repeating that the *get_stats* method can be called at any time—even when the interface is down—so the driver must retain statistical information for as long as the `net_device` structure exists.

Multicast

A *multicast* packet is a network packet meant to be received by more than one host, but not by all hosts. This functionality is obtained by assigning special hardware addresses to groups of hosts. Packets directed to one of the special addresses should be received by all the hosts in that group. In the case of Ethernet, a multicast address has the least significant bit of the first address octet set in the destination address, while every device board has that bit clear in its own hardware address.

The tricky part of dealing with host groups and hardware addresses is performed by applications and the kernel, and the interface driver doesn't need to deal with these problems.

Transmission of multicast packets is a simple problem because they look exactly like any other packets. The interface transmits them over the communication medium without looking at the destination address. It's the kernel that has to assign a correct hardware destination address; the *hard_header* device method, if defined, doesn't need to look in the data it arranges.

The kernel handles the job of tracking which multicast addresses are of interest at any given time. The list can change frequently, since it is a function of the applications that are running at any given time and the users' interest. It is the driver's job to accept the list of interesting multicast addresses and deliver to the kernel any packets sent to those addresses. How the driver implements the multicast list is somewhat dependent on how the underlying hardware works. Typically, hardware belongs to one of three classes, as far as multicast is concerned:

- Interfaces that cannot deal with multicast. These interfaces either receive packets directed specifically to their hardware address (plus broadcast packets) or receive every packet. They can receive multicast packets only by receiving every packet, thus, potentially overwhelming the operating system with a huge number of "uninteresting" packets. You don't usually count these interfaces as multicast capable, and the driver won't set `IFF_MULTICAST` in `dev->flags`.

  Point-to-point interfaces are a special case because they always receive every packet without performing any hardware filtering.

- Interfaces that can tell multicast packets from other packets (host-to-host or broadcast). These interfaces can be instructed to receive every multicast packet and let the software determine if the address is interesting for this host. The overhead introduced in this case is acceptable, because the number of multicast packets on a typical network is low.
- Interfaces that can perform hardware detection of multicast addresses. These interfaces can be passed a list of multicast addresses for which packets are to be received, and ignore other multicast packets. This is the optimal case for the kernel, because it doesn't waste processor time dropping "uninteresting" packets received by the interface.

The kernel tries to exploit the capabilities of high-level interfaces by supporting the third device class, which is the most versatile, at its best. Therefore, the kernel notifies the driver whenever the list of valid multicast addresses is changed, and it passes the new list to the driver so it can update the hardware filter according to the new information.

Kernel Support for Multicasting

Support for multicast packets is made up of several items: a device method, a data structure, and device flags:

`void (*dev->set_multicast_list)(struct net_device *dev);`
: Device method called whenever the list of machine addresses associated with the device changes. It is also called when `dev->flags` is modified, because some flags (e.g., `IFF_PROMISC`) may also require you to reprogram the hardware filter. The method receives a pointer to `struct net_device` as an argument and returns `void`. A driver not interested in implementing this method can leave the field set to `NULL`.

`struct dev_mc_list *dev->mc_list;`
: A linked list of all the multicast addresses associated with the device. The actual definition of the structure is introduced at the end of this section.

`int dev->mc_count;`
: The number of items in the linked list. This information is somewhat redundant, but checking `mc_count` against `0` is a useful shortcut for checking the list.

`IFF_MULTICAST`
: Unless the driver sets this flag in `dev->flags`, the interface won't be asked to handle multicast packets. Nonetheless, the kernel calls the driver's *set_multicast_list* method when `dev->flags` changes, because the multicast list may have changed while the interface was not active.

IFF_ALLMULTI
: Flag set in dev->flags by the networking software to tell the driver to retrieve all multicast packets from the network. This happens when multicast routing is enabled. If the flag is set, dev->mc_list shouldn't be used to filter multicast packets.

IFF_PROMISC
: Flag set in dev->flags when the interface is put into promiscuous mode. Every packet should be received by the interface, independent of dev->mc_list.

The last bit of information needed by the driver developer is the definition of struct dev_mc_list, which lives in <*linux/netdevice.h*>:

```
struct dev_mc_list {
    struct dev_mc_list   *next;          /* Next address in list */
    __u8                 dmi_addr[MAX_ADDR_LEN]; /* Hardware address */
    unsigned char        dmi_addrlen;    /* Address length */
    int                  dmi_users;      /* Number of users */
    int                  dmi_gusers;     /* Number of groups */
};
```

Because multicasting and hardware addresses are independent of the actual transmission of packets, this structure is portable across network implementations, and each address is identified by a string of octets and a length, just like dev->dev_addr.

A Typical Implementation

The best way to describe the design of *set_multicast_list* is to show you some pseudocode.

The following function is a typical implementation of the function in a full-featured (ff) driver. The driver is full featured in that the interface it controls has a complex hardware packet filter, which can hold a table of multicast addresses to be received by this host. The maximum size of the table is FF_TABLE_SIZE.

All the functions prefixed with ff_ are placeholders for hardware-specific operations:

```
void ff_set_multicast_list(struct net_device *dev)
{
    struct dev_mc_list *mcptr;

    if (dev->flags & IFF_PROMISC) {
        ff_get_all_packets( );
        return;
    }
    /* If there's more addresses than we handle, get all multicast
    packets and sort them out in software. */
    if (dev->flags & IFF_ALLMULTI || dev->mc_count > FF_TABLE_SIZE) {
        ff_get_all_multicast_packets( );
        return;
    }
    /* No multicast?  Just get our own stuff */
    if (dev->mc_count == 0) {
        ff_get_only_own_packets();
        return;
    }
    /* Store all of the multicast addresses in the hardware filter */
    ff_clear_mc_list();
    for (mc_ptr = dev->mc_list; mc_ptr; mc_ptr = mc_ptr->next)
        ff_store_mc_address(mc_ptr->dmi_addr);
    ff_get_packets_in_multicast_list();
}
```

This implementation can be simplified if the interface cannot store a multicast table in the hardware filter for incoming packets. In that case, `FF_TABLE_SIZE` reduces to `0`, and the last four lines of code are not needed.

As was mentioned earlier, even interfaces that can't deal with multicast packets need to implement the *set_multicast_list* method to be notified about changes in `dev->flags`. This approach could be called a "nonfeatured" (`nf`) implementation. The implementation is very simple, as shown by the following code:

```
void nf_set_multicast_list(struct net_device *dev)
{
    if (dev->flags & IFF_PROMISC)
        nf_get_all_packets();
    else
        nf_get_only_own_packets();
}
```

Implementing `IFF_PROMISC` is important, because otherwise the user won't be able to run *tcpdump* or any other network analyzers. If the interface runs a point-to-point link, on the other hand, there's no need to implement *set_multicast_list* at all, because users receive every packet anyway.

A Few Other Details

This section covers a few other topics that may be of interest to network driver authors. In each case, we simply try to point you in the right direction. Obtaining a complete picture of the subject probably requires spending some time digging through the kernel source as well.

Media Independent Interface Support

Media Independent Interface (or MII) is an IEEE 802.3 standard describing how Ethernet transceivers can interface with network controllers; many products on the market conform with this interface. If you are writing a driver for an MII-compliant controller, the kernel exports a generic MII support layer that may make your life easier.

To use the generic MII layer, you should include <*linux/mii.h*>. You need to fill out an `mii_if_info` structure with information on the physical ID of the transceiver, whether full duplex is in effect, etc. Also required are two methods for the `mii_if_info` structure:

```
    int (*mdio_read) (struct net_device *dev, int phy_id, int location);
    void (*mdio_write) (struct net_device *dev, int phy_id, int location, int val);
```

As you might expect, these methods should implement communications with your specific MII interface.

The generic MII code provides a set of functions for querying and changing the operating mode of the transceiver; many of these are designed to work with the *ethtool* utility (described in the next section). Look in <*linux/mii.h*> and *drivers/net/mii.c* for the details.

Ethtool Support

Ethtool is a utility designed to give system administrators a great deal of control over the operation of network interfaces. With *ethtool*, it is possible to control various interface parameters including speed, media type, duplex operation, DMA ring setup, hardware checksumming, wake-on-LAN operation, etc., but only if *ethtool* is supported by the driver. *Ethtool* may be downloaded from *http://sf.net/projects/gkernel/*.

The relevant declarations for *ethtool* support may be found in <*linux/ethtool.h*>. At the core of it is a structure of type `ethtool_ops`, which contains a full 24 different methods for *ethtool* support. Most of these methods are relatively straightforward; see <*linux/ethtool.h*> for the details. If your driver is using the MII layer, you can use *mii_ethtool_gset* and *mii_ethtool_sset* to implement the *get_settings* and *set_settings* methods, respectively.

For *ethtool* to work with your device, you must place a pointer to your `ethtool_ops` structure in the `net_device` structure. The macro `SET_ETHTOOL_OPS` (defined in <*linux/netdevice.h*>) should be used for this purpose. Do note that your *ethtool* methods can be called even when the interface is down.

Netpoll

"Netpoll" is a relatively late (2.6.5) addition to the network stack; its purpose is to enable the kernel to send and receive packets in situations where the full network and I/O subsystems may not be available. It is used for features like remote network consoles and remote kernel debugging. Supporting netpoll in your driver is not, by any means, necessary, but it may make your device more useful in some situations. Supporting netpoll is also relatively easy in most cases.

Drivers implementing netpoll should implement the *poll_controller* method. Its job is to keep up with anything that may be happening on the controller in the absence of device interrupts. Almost all *poll_controller* methods take the following form:

```
void my_poll_controller(struct net_device *dev)
{
    disable_device_interrupts(dev);
    call_interrupt_handler(dev->irq, dev, NULL);
    reenable_device_interrupts(dev);
}
```

The *poll_controller* method, in essence, is simply simulating interrupts from the given device.

Quick Reference

This section provides a reference for the concepts introduced in this chapter. It also explains the role of each header file that a driver needs to include. The lists of fields in the `net_device` and `sk_buff` structures, however, are not repeated here.

`#include <linux/netdevice.h>`
: Header that hosts the definitions of `struct net_device` and `struct net_device_stats`, and includes a few other headers that are needed by network drivers.

`struct net_device *alloc_netdev(int sizeof_priv, char *name, void (*setup)(struct net_device *);`
`struct net_device *alloc_etherdev(int sizeof_priv);`
`void free_netdev(struct net_device *dev);`
: Functions for allocating and freeing `net_device` structures.

`int register_netdev(struct net_device *dev);`
`void unregister_netdev(struct net_device *dev);`
: Registers and unregisters a network device.

`void *netdev_priv(struct net_device *dev);`
: A function that retrieves the pointer to the driver-private area of a network device structure.

`struct net_device_stats;`
: A structure that holds device statistics.

`netif_start_queue(struct net_device *dev);`
`netif_stop_queue(struct net_device *dev);`
`netif_wake_queue(struct net_device *dev);`
: Functions that control the passing of packets to the driver for transmission. No packets are transmitted until *netif_start_queue* has been called. *netif_stop_queue* suspends transmission, and *netif_wake_queue* restarts the queue and pokes the network layer to restart transmitting packets.

```
skb_shinfo(struct sk_buff *skb);
```

A macro that provides access to the “shared info” portion of a packet buffer.

```
void netif_rx(struct sk_buff *skb);
```

Function that can be called (including at interrupt time) to notify the kernel that a packet has been received and encapsulated into a socket buffer.

```
void netif_rx_schedule(dev);
```

Function that informs the kernel that packets are available and that polling should be started on the interface; it is used only by NAPI-compliant drivers.

```
int netif_receive_skb(struct sk_buff *skb);
void netif_rx_complete(struct net_device *dev);
```

Functions that should be used only by NAPI-compliant drivers. *netif_receive_skb* is the NAPI equivalent to *netif_rx*; it feeds a packet into the kernel. When a NAPI-compliant driver has exhausted the supply of received packets, it should reenable interrupts, and call *netif_rx_complete* to stop polling.

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

Included by *netdevice.h*, this file declares the interface flags (`IFF_` macros) and `struct ifmap`, which has a major role in the *ioctl* implementation for network drivers.

```
void netif_carrier_off(struct net_device *dev);
void netif_carrier_on(struct net_device *dev);
int netif_carrier_ok(struct net_device *dev);
```

The first two functions may be used to tell the kernel whether a carrier signal is currently present on the given interface. *netif_carrier_ok* tests the carrier state as reflected in the device structure.

```
#include <linux/if_ether.h>
ETH_ALEN
ETH_P_IP
struct ethhdr;
```

Included by *netdevice.h*, *if_ether.h* defines all the `ETH_` macros used to represent octet lengths (such as the address length) and network protocols (such as IP). It also defines the `ethhdr` structure.

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

The definition of `struct sk_buff` and related structures, as well as several inline functions to act on the buffers. This header is included by *netdevice.h*.

```
struct sk_buff *alloc_skb(unsigned int len, int priority);
struct sk_buff *dev_alloc_skb(unsigned int len);
void kfree_skb(struct sk_buff *skb);
void dev_kfree_skb(struct sk_buff *skb);
void dev_kfree_skb_irq(struct sk_buff *skb);
void dev_kfree_skb_any(struct sk_buff *skb);
```

Functions that handle the allocation and freeing of socket buffers. Drivers should normally use the `dev_` variants, which are intended for that purpose.

```
unsigned char *skb_put(struct sk_buff *skb, int len);
unsigned char *__skb_put(struct sk_buff *skb, int len);
unsigned char *skb_push(struct sk_buff *skb, int len);
unsigned char *__skb_push(struct sk_buff *skb, int len);
```

Functions that add data to an `skb`; *skb_put* puts the data at the end of the `skb`, while *skb_push* puts it at the beginning. The regular versions perform checking to ensure that adequate space is available; double-underscore versions leave those tests out.

```
int skb_headroom(struct sk_buff *skb);
int skb_tailroom(struct sk_buff *skb);
void skb_reserve(struct sk_buff *skb, int len);
```

Functions that perform management of space within an `skb`. *skb_headroom* and *skb_tailroom* tell how much space is available at the beginning and end, respectively, of an `skb`. *skb_reserve* may be used to reserve space at the beginning of an `skb`, which must be empty.

```
unsigned char *skb_pull(struct sk_buff *skb, int len);
```

skb_pull "removes" data from an `skb` by adjusting the internal pointers.

```
int skb_is_nonlinear(struct sk_buff *skb);
```

Function that returns a true value if this skb is separated into multiple fragments for scatter/gather I/O.

```
int skb_headlen(struct sk_buff *skb);
```

Returns the length of the first segment of the skb—that part pointed to by `skb->data`.

```
void *kmap_skb_frag(skb_frag_t *frag);
void kunmap_skb_frag(void *vaddr);
```

Functions that provide direct access to fragments within a nonlinear skb.

```
#include <linux/etherdevice.h>
void ether_setup(struct net_device *dev);
```

Function that sets most device methods to the general-purpose implementation for Ethernet drivers. It also sets `dev->flags` and assigns the next available `eth`*x* name to `dev->name` if the first character in the name is a blank space or the `NULL` character.

`unsigned short eth_type_trans(struct sk_buff *skb, struct net_device *dev);`

When an Ethernet interface receives a packet, this function can be called to set `skb->pkt_type`. The return value is a protocol number that is usually stored in `skb->protocol`.

`#include <linux/sockios.h>`
`SIOCDEVPRIVATE`

The first of 16 *ioctl* commands that can be implemented by each driver for its own private use. All the network *ioctl* commands are defined in *sockios.h*.

`#include <linux/mii.h>`
`struct mii_if_info;`

Declarations and a structure supporting drivers of devices that implement the MII standard.

`#include <linux/ethtool.h>`
`struct ethtool_ops;`

Declarations and structures that let devices work with the *ethtool* utility.