

The HyperNews [Linux KHG](#) Discussion Pages

Network Buffers And Memory Management

Reprinted with permission of [Linux Journal](#), from issue 29, September 1996. Some changes have been made to accomodate the web. This article was originally written for the Kernel Korner column. The Kernel Korner series has included many other articles of interest to Linux kernel hackers, as well.

by Alan Cox

The Linux operating system implements the industry-standard Berkeley socket API, which has its origins in the BSD unix developments (4.2/4.3/4.4 BSD). In this article, we will look at the way the memory management and buffering is implemented for network layers and network device drivers under the existing Linux kernel, as well as explain how and why some things have changed over time.

Core Concepts

The networking layer tries to be fairly object-oriented in its design, as indeed is much of the Linux kernel. The core structure of the networking code goes back to the initial networking and socket implementations by Ross Biro and Orest Zborowski respectively. The key objects are:

Device or Interface:

A network interface represents a thing which sends and receives packets. This is normally interface code for a physical device like an ethernet card. However some devices are software only such as the loopback device which is used for sending data to yourself.

Protocol:

Each protocol is effectively a different language of networking. Some protocols exist purely because vendors chose to use proprietary networking schemes, others are designed for special purposes. Within the Linux kernel each protocol is a seperate module of code which provides services to the socket layer.

Socket:

So called from the notion of plugs and sockets. A socket is a connection in the networking that provides unix file I/O and exists to the user program as a file descriptor. In the kernel each socket is a pair of structures that represent the high level socket interface and low level protocol interface.

sk_buff:

All the buffers used by the networking layers are `sk_buffs`. The control for these is provided by core low-level library routines available to the whole of the networking. `sk_buffs` provide the general buffering and flow control facilities needed by network protocols.

Implementation of sk_buffs

The primary goal of the `sk_buff` routines is to provide a consistent and efficient buffer handling method for all of the network layers, and by being consistent to make it possible to provide higher level `sk_buff` and socket handling facilities to all the protocols.

An `sk_buff` is a control structure with a block of memory attached. There are two primary sets of functions provided in the `sk_buff` library. Firstly routines to manipulate doubly linked lists of `sk_buffs`, secondly functions for controlling the attached memory. The buffers are held on linked lists optimised for the common network operations of append to end and remove from start. As so much of the networking functionality occurs during interrupts these routines are written to be atomic. The small extra overhead this causes is well worth the pain it saves in bug hunting.

We use the list operations to manage groups of packets as they arrive from the network, and as we send them to the physical interfaces. We use the memory manipulation routines for handling the contents of packets in a standardised and efficient manner.

At its most basic level, a list of buffers is managed using functions like this:

```
void append_frame(char *buf, int len)
{
    struct sk_buff *skb=alloc_skb(len, GFP_ATOMIC);
    if(skb==NULL)
        my_dropped++;
    else
    {
        skb_put(skb, len);
        memcpy(skb->data, data, len);
        skb_append(&my_list, skb);
    }
}

void process_queue(void)
{
    struct sk_buff *skb;
    while((skb=skb_dequeue(&my_list))!=NULL)
    {
        process_data(skb);
    }
}
```

```

    kfree_skb(skb, FREE_READ);
}
}

```

These two fairly simplistic pieces of code actually demonstrate the receive packet mechanism quite accurately. The `append_frame()` function is similar to the code called from an interrupt by a device driver receiving a packet, and `process_frame()` is similar to the code called to feed data into the protocols. If you go and look in `net/core/dev.c` at `netif_rx()` and `net_bh()`, you will see that they manage buffers similarly. They are far more complex, as they have to feed packets to the right protocol and manage flow control, but the basic operations are the same. This is just as true if you look at buffers going from the protocol code to a user application.

The example also shows the use of one of the data control functions, `skb_put()`. Here it is used to reserve space in the buffer for the data we wish to pass down.

Let's look at `append_frame()`. The `alloc_skb()` function obtains a buffer of `len` bytes ([Figure 1](#)), which consists of:

- 0 bytes of room at the head of the buffer
- 0 bytes of data, and
- `len` bytes of room at the end of the data.

The `skb_put()` function ([Figure 4](#)) grows the **data** area upwards in memory through the free space at the buffer end and thus reserves space for the `memcpy()`. Many network operations used in sending add to the start of the frame each time in order to add headers to packets, so the `skb_push()` function ([Figure 5](#)) is provided to allow you to move the start of the data frame down through memory, providing enough space has been reserved to leave room for doing this.

Immediately after a buffer has been allocated, all the available room is at the end. A further function named `skb_reserve()` ([Figure 2](#)) can be called before data is added allows you to specify that some of the room should be at the beginning. Thus, many sending routines start with something like:

```

skb=alloc_skb(len+headspace, GFP_KERNEL);
skb_reserve(skb, headspace);
skb_put(skb, len);
memcpy_fromfs(skb->data, data, len);
pass_to_m_protocol(skb);

```

In systems such as BSD unix you don't need to know in advance how much space you will need as it uses chains of small buffers (mbufs) for its network buffers. Linux chooses to use linear buffers and save space in advance (often wasting a few bytes to allow for the worst case) because linear buffers make

many other things much faster.

Now to return to the list functions. Linux provides the following operations:

- `skb_dequeue()` takes the first buffer from a list. If the list is empty a `NULL` pointer is returned. This is used to pull buffers off queues. The buffers are added with the routines `skb_queue_head()` and `skb_queue_tail()`.
- `skb_queue_head()` places a buffer at the start of a list. As with all the list operations, it is atomic.
- `skb_queue_tail()` places a buffer at the end of a list, which is the most commonly used function. Almost all the queues are handled with one set of routines queueing data with this function and another set removing items from the same queues with `skb_dequeue()`.
- `skb_unlink()` removes a buffer from whatever list it was on. The buffer is not freed, merely removed from the list. To make some operations easier, you need not know what list the buffer is on, and you can always call `skb_unlink()` on a buffer which is not in a list. This enables network code to pull a buffer out of use even when the network protocol has no idea who is currently using it. A separate locking mechanism is provided so device drivers do not find someone removing a buffer they are using at that moment.
- Some more complex protocols like TCP keep frames in order and re-order their input as data is received. Two functions, `skb_insert()` and `skb_append()`, exist to allow users to place `sk_buffs` before or after a specific buffer in a list.
- `alloc_skb()` creates a new `sk_buff` and initialises it. The returned buffer is ready to use but does assume you will fill in a few fields to indicate how the buffer should be freed. Normally this is `skb->free=1`. A buffer can be told not to be freed when `kfree_skb()` (see below) is called.
- `kfree_skb()` releases a buffer, and if `skb->sk` is set it lowers the memory use counts of the socket (`sk`). It is up to the socket and protocol-level routines to have incremented these counts and to avoid freeing a socket with outstanding buffers. The memory counts are very important, as the kernel networking layers need to know how much memory is tied up by each connection in order to prevent remote machines or local processes from using too much memory.
- `skb_clone()` makes a copy of an `sk_buff` but does not copy the data area, which must be considered read only.
- For some things a copy of the data is needed for editing, and `skb_copy()` provides the same facilities but also copies the data (and thus has a much higher overhead).



Figure 1: After alloc_skb



Figure 2: After skb_reserve



Figure 3: An sk_buff containing data



Figure 4: After skb_put has been called on the buffer



Figure 5: After an skb_push has occurred on the previous buffer

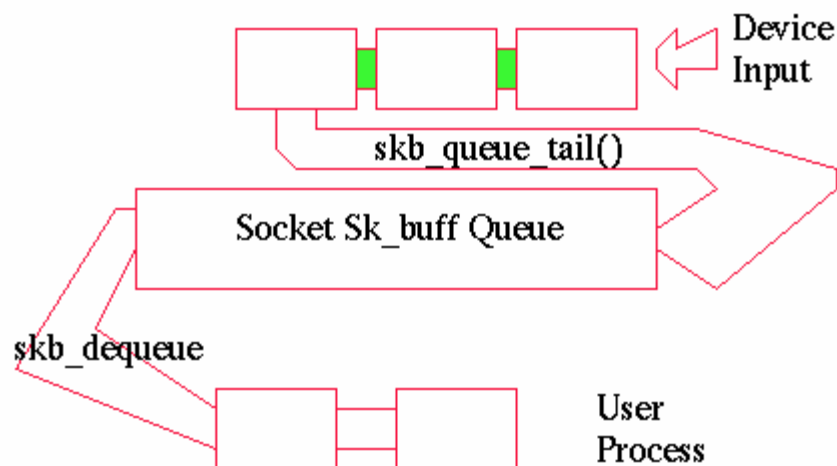


Figure 6: Network device data flow

Higher Level Support Routines

The semantics of allocating and queueing buffers for sockets also involve flow control rules and for sending a whole list of interactions with signals and optional settings such as non blocking. Two routines are designed to make this easy for most protocols.

The `sock_queue_rcv_skb()` function is used to handle incoming data flow control and is normally used in the form:

```
sk=my_find_socket(whatever);
if(sock_queue_rcv_skb(sk,skb)==-1)
{
    myproto_stats.dropped++;
    kfree_skb(skb,FREE_READ);
    return;
}
```

This function uses the socket read queue counters to prevent vast amounts of data being queued to a socket. After a limit is hit, data is discarded. It is up to the application to read fast enough, or as in TCP, for the protocol to do flow control over the network. TCP actually tells the sending machine to shut up when it can no longer queue data.

On the sending side, `sock_alloc_send_skb()` handles signal handling, the non blocking flag, and all the semantics of blocking until there is space in the send queue so you cannot tie up all of memory with data queued for a slow interface. Many protocol send routines have this function doing almost all the work:

```
skb=sock_alloc_send_skb(sk,...)
if(skb==NULL)
    return -err;
skb->sk=sk;
skb_reserve(skb, headroom);
skb_put(skb,len);
memcpy(skb->data, data, len);
protocol_do_something(skb);
```

Most of this we have met before. The very important line is `skb->sk=sk`. The `sock_alloc_send_skb()` has charged the memory for the buffer to the socket. By setting `skb->sk` we tell the kernel that whoever does a `kfree_skb()` on the buffer should cause the socket to be credited the memory for the buffer. Thus when a device has sent a buffer and frees it the user will be able to send more.

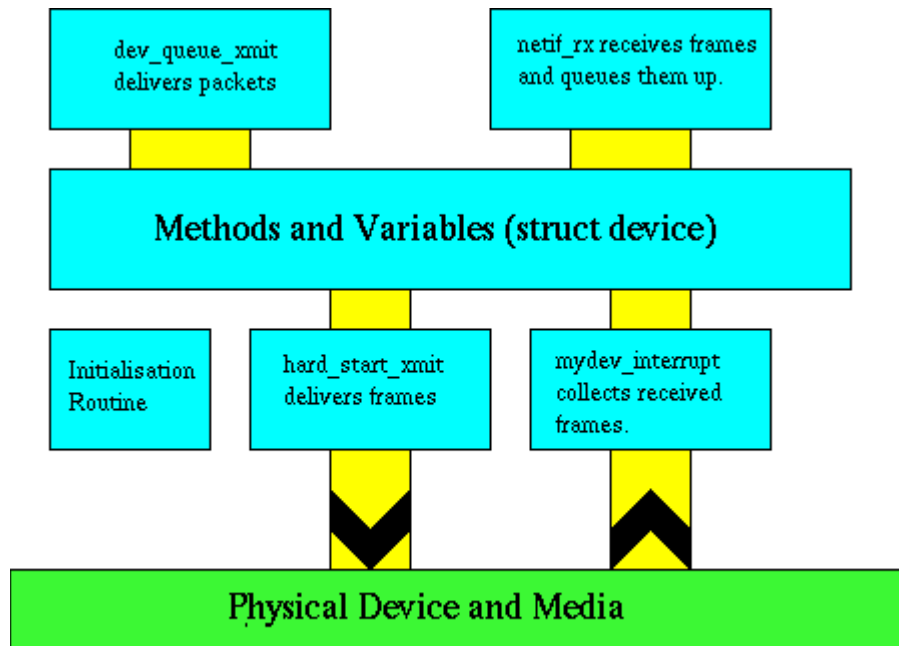
Network Devices

All Linux network devices follow the same interface although many functions available in that interface will not be needed for all devices. An object oriented

mentality is used and each device is an object with a series of methods that are filled into a structure. Each method is called with the device itself as the first argument. This is done to get around the lack of the C++ concept of this within the C language.

The file `drivers/net/skeleton.c` contains the skeleton of a network device driver. View or print a copy from a recent kernel and follow along throughout the rest of the article.

Basic Structure



Each network device deals entirely in the transmission of network buffers from the protocols to the physical media, and in receiving and decoding the responses the hardware generates. Incoming frames are turned into network buffers, identified by protocol and delivered to `netif_rx()`. This function then passes the frames off to the protocol layer for further processing.

Each device provides a set of additional methods for the handling of stopping, starting, control and physical encapsulation of packets. These and all the other control information are collected together in the device structures that are used to manage each device.

Naming

All Linux network devices have a unique name. This is not in any way related to the file system names devices may have, and indeed network devices do not

normally have a filesystem representation, although you may create a device which is tied to device drivers. Traditionally the name indicates only the type of a device rather than its maker. Multiple devices of the same type are numbered upwards from 0. Thus ethernet devices are known as ``eth0'', ``eth1'', ``eth2'' etc. The naming scheme is important as it allows users to write programs or system configuration in terms of ``an ethernet card'' rather than worrying about the manufacturer of the board and forcing reconfiguration if a board is changed.

The following names are currently used for generic devices:

`ethn`

Ethernet controllers, both 10 and 100Mb/second

`trn`

Token ring devices.

`sln`

SLIP devices. Also used in AX.25 KISS mode.

`pppn`

PPP devices both asynchronous and synchronous.

`plipn`

PLIP units. The number matches the printer port.

`tunln`

IPIP encapsulated tunnels

`nrn`

NetROM virtual devices

`isdnn`

ISDN interfaces handled by isdn4linux. (*)

`dummysn`

Null devices

`lo`

The loopback device

(*) At least one ISDN interface is an ethernet impersonator, that is the Sonix PC/Volante driver. Therefore, it uses an ``eth'' device name as it behaves in all aspects as if it was ethernet rather than ISDN.

If possible, a new device should pick a name that reflects existing practice. When you are adding a whole new physical layer type you should look for other people working on such a project and use a common naming scheme.

Certain physical layers present multiple logical interfaces over one media. Both ATM and Frame Relay have this property, as does multi-drop KISS in the amateur radio environment. Under such circumstances a driver needs to exist for each active channel. The Linux networking code is structured in such a way as to make this manageable without excessive additional code, and the name

registration scheme allows you to create and remove interfaces almost at will as channels come into and out of existence. The proposed convention for such names is still under some discussion, as the simple scheme of ``sl0a'', ``sl0b'', "sl0c" works for basic devices like multidrop KISS, but does not cope with multiple frame relay connections where a virtual channel may be moved across physical boards.

Registering A Device

Each device is created by filling in a `struct device` object and passing it to the `register_netdev(struct device *)` call. This links your device structure into the kernel network device tables. As the structure you pass in is used by the kernel, you must not free this until you have unloaded the device with `void unregister_netdev(struct device *)` calls. These calls are normally done at boot time, or module load and unload.

The kernel will not object if you create multiple devices with the same name, it will break. Therefore, if your driver is a loadable module you should use the `struct device *dev_get(const char *name)` call to ensure the name is not already in use. If it is in use, you should fail or pick another name. You may not use `unregister_netdev()` to unregister the other device with the name if you discover a clash!

A typical code sequence for registration is:

```
int register_my_device(void)
{
    int i=0;
    for(i=0;i<100;i++)
    {
        sprintf(mydevice.name,"mydev%d",i);
        if(dev_get(mydevice.name)==NULL)
        {
            if(register_netdev(&mydevice)!=0)
                return -EIO;
            return 0;
        }
    }
    printk("100 mydevs loaded. Unable to load more.\n");
    return -ENFILE;
}
```

The Device Structure

All the generic information and methods for each network device are kept in the device structure. To create a device you need to fill most of these in. This section covers how they should be set up.

Naming

First, the name field holds the device name. This is a string pointer to a name in the formats discussed previously. It may also be " " (four spaces), in which case the kernel will automatically assign an *ethn* name to it. This is a special feature that is best not used. After Linux 2.0, we intend to change to a simple support function of the form `dev_make_name("eth")`.

Bus Interface Parameters

The next block of parameters are used to maintain the location of a device within the device address spaces of the architecture. The `irq` field holds the interrupt (IRQ) the device is using. This is normally set at boot, or by the initialization function. If an interrupt is not used, not currently known, or not assigned, the value zero should be used. The interrupt can be set in a variety of fashions. The auto-irq facilities of the kernel may be used to probe for the device interrupt, or the interrupt may be set when loading the network module. Network drivers normally use a global int called `irq` for this so that users can load the module with `insmod mydevice irq=5` style commands. Finally, the IRQ may be set dynamically from the `ifconfig` command. This causes a call to your device that will be discussed later on.

The `base_addr` field is the base I/O space address the device resides at. If the device uses no I/O locations or is running on a system with no I/O space concept this field should be zero. When this is user settable, it is normally set by a global variable called `io`. The interface I/O address may also be set with `ifconfig`.

Two hardware shared memory ranges are defined for things like ISA bus shared memory ethernet cards. For current purposes, the `rmem_start` and `rmem_end` fields are obsolete and should be loaded with 0. The `mem_start` and `mem_end` addresses should be loaded with the start and end of the shared memory block used by this device. If no shared memory block is used, then the value 0 should be stored. Those devices that allow the user to specify this parameter use a global variable called `mem` to set the memory base, and set the `mem_end` appropriately themselves.

The `dma` variable holds the DMA channel in use by the device. Linux allows DMA (like interrupts) to be automatically probed. If no DMA channel is used, or the DMA channel is not yet set, the value 0 is used. This may have to change, since the latest PC boards allow ISA bus DMA channel 0 to be used by hardware boards and do not just tie it to memory refresh. If the user can set the DMA channel the global variable `dma` is used.

It is important to realise that the physical information is provided for control

and user viewing (as well as the driver's internal functions), and does not register these areas to prevent them being reused. Thus the device driver must also allocate and register the I/O, DMA and interrupt lines it wishes to use, using the same kernel functions as any other device driver. [See the recent Kernel Korner articles on writing a character device driver in issues 23, 24, 25, 26, and 28 of [Linux Journal](#).]

The `if_port` field holds the physical media type for multi-media devices such as combo ethernet boards.

Protocol Layer Variables

In order for the network protocol layers to perform in a sensible manner, the device has to provide a set of capability flags and variables. These are also maintained in the device structure.

The `mtu` is the largest payload that can be sent over this interface (that is, the largest packet size not including any bottom layer headers that the device itself will provide). This is used by the protocol layers such as IP to select suitable packet sizes to send. There are minimums imposed by each protocol. A device is not usable for IPX without a 576 byte frame size or higher. IP needs at least 72 bytes, and does not perform sensibly below about 200 bytes. It is up to the protocol layers to decide whether to co-operate with your device.

The `family` is always set to `AF_INET` and indicates the protocol family the device is using. Linux allows a device to be using multiple protocol families at once, and maintains this information solely to look more like the standard BSD networking API.

The interface hardware type (`type`) field is taken from a table of physical media types. The values used by the ARP protocol (see RFC1700) are used for those media supporting ARP and additional values are assigned for other physical layers. New values are added when necessary both to the kernel and to **net-tools** which is the package containing programs like **ifconfig** that need to be able to decode this field. The fields defined as of Linux pre2.0.5 are:

From RFC1700:

`ARPHRD_NETROM`

NET/ROM(tm) devices.

`ARPHRD_ETHER`

10 and 100Mbit/second ethernet.

`ARPHRD_EETHER`

Experimental Ethernet (not used).

`ARPHRD_AX25`

AX.25 level 2 interfaces.

ARPHRD_PRNET

PRNet token ring (not used).

ARPHRD_CHAOS

ChaosNET (not used).

ARPHRD_IEE802

802.2 networks notably token ring.

ARPHRD_ARCNET

ARCnet interfaces.

ARPHRD_DLCI

Frame Relay DLCI.

Defined by Linux:

ARPHRD_SLIP

Serial Line IP protocol

ARPHRD_CSLIP

SLIP with VJ header compression

ARPHRD_SLIP6

6bit encoded SLIP

ARPHRD_CSLIP6

6bit encoded header compressed SLIP

ARPHRD_ADAPT

SLIP interface in adaptive mode

ARPHRD_PPP

PPP interfaces (async and sync)

ARPHRD_TUNNEL

IPIP tunnels

ARPHRD_TUNNEL6

IPv6 over IP tunnels

ARPHRD_FRAD

Frame Relay Access Device.

ARPHRD_SKIP

SKIP encryption tunnel.

ARPHRD_LOOPBACK

Loopback device.

ARPHRD_LOCALTLK

Localtalk apple networking device.

ARPHRD_METRICOM

Metricom Radio Network.

Those interfaces marked unused are defined types but without any current support on the existing net-tools. The Linux kernel provides additional generic support routines for devices using ethernet and token ring.

The `pa_addr` field is used to hold the IP address when the interface is up. Interfaces should start down with this variable clear. `pa_brddaddr` is used to hold the configured broadcast address, `pa_dstaddr` the target of a point to point link and `pa_mask` the IP netmask of the interface. All of these can be initialised to zero. The `pa_alen` field holds the length of an address (in our case an IP address), this should be initialised to 4.

Link Layer Variables

The `hard_header_len` is the number of bytes the device desires at the start of a network buffer it is passed. It does not have to be the number of bytes of physical header that will be added, although this is normal. A device can use this to provide itself a scratchpad at the start of each buffer.

In the 1.2.x series kernels, the `skb->data` pointer will point to the buffer start and you must avoid sending your scratchpad yourself. This also means for devices with variable length headers you will need to allocate `max_size+1` bytes and keep a length byte at the start so you know where the header really begins (the header should be contiguous with the data). Linux 1.3.x makes life much simpler and ensures you will have at least as much room as you asked free at the start of the buffer. It is up to you to use `skb_push()` appropriately as was discussed in the section on networking buffers.

The physical media addresses (if any) are maintained in `dev_addr` and `broadcast` respectively. These are byte arrays and addresses smaller than the size of the array are stored starting from the left. The `addr_len` field is used to hold the length of a hardware address. With many media there is no hardware address, and this should be set to zero. For some other interfaces the address must be set by a user program. The `ifconfig` tool permits the setting of an interface hardware address. In this case it need not be set initially, but the open code should take care not to allow a device to start transmitting without an address being set.

Flags

A set of flags are used to maintain the interface properties. Some of these are ``compatibility'' items and as such not directly useful. The flags are:

IFF_UP

The interface is currently active. In Linux, the `IFF_RUNNING` and `IFF_UP` flags are basically handled as a pair. They exist as two items for compatibility reasons. When an interface is not marked as `IFF_UP` it may be removed. Unlike BSD, an interface that does not have `IFF_UP` set will never receive packets.

IFF_BROADCAST

The interface has broadcast capability. There will be a valid IP address stored in the device addresses.

IFF_DEBUG

Available to indicate debugging is desired. Not currently used.

IFF_LOOPBACK

The loopback interface (lo) is the only interface that has this flag set. Setting it on other interfaces is neither defined nor a very good idea.

IFF_POINTOPOINT

The interface is a point to point link (such as SLIP or PPP). There is no broadcast capability as such. The remote point to point address in the device structure is valid. A point to point link has no netmask or broadcast normally, but this can be enabled if needed.

IFF_NOTRAILERS

More of a prehistoric than a historic compatibility flag. Not used.

IFF_RUNNING

See IFF_UP

IFF_NOARP

The interface does not perform ARP queries. Such an interface must have either a static table of address conversions or no need to perform mappings. The NetROM interface is a good example of this. Here all entries are hand configured as the NetROM protocol cannot do ARP queries.

IFF_PROMISC

The interface if it is possible will hear all packets on the network. This is typically used for network monitoring although it may also be used for bridging. One or two interfaces like the AX.25 interfaces are always in promiscuous mode.

IFF_ALLMULTI

Receive all multicast packets. An interface that cannot perform this operation but can receive all packets will go into promiscuous mode when asked to perform this task.

IFF_MULTICAST

Indicate that the interface supports multicast IP traffic. This is not the same as supporting a physical multicast. AX.25 for example supports IP multicast using physical broadcast. Point to point protocols such as SLIP generally support IP multicast.

The Packet Queue

Packets are queued for an interface by the kernel protocol code. Within each device, `buffs[]` is an array of packet queues for each kernel priority level. These are maintained entirely by the kernel code, but must be initialised by the device itself on boot up. The intialisation code used is:

```
int ct=0;
while(ct<DEV_NUMBUFFS)
{
    skb_queue_head_init(&dev->buffs[ct]);
    ct++;
}
```

All other fields should be initialised to 0.

The device gets to select the queue length it wants by setting the field `dev->tx_queue_len` to the maximum number of frames the kernel should queue for the device. Typically this is around 100 for ethernet and 10 for serial lines. A device can modify this dynamically, although its effect will lag the change slightly.

Network Device Methods

Each network device has to provide a set of actual functions (methods) for the basic low level operations. It should also provide a set of support functions that interface the protocol layer to the protocol requirements of the link layer it is providing.

Setup

The `init` method is called when the device is initialised and registered with the system. It should perform any low level verification and checking needed, and return an error code if the device is not present, areas cannot be registered or it is otherwise unable to proceed. If the `init` method returns an error the `register_netdev()` call returns the error code and the device is not created.

Frame Transmission

All devices must provide a `transmit` function. It is possible for a device to exist that cannot transmit. In this case the device needs a `transmit` function that simply frees the buffer it is passed. The dummy device has exactly this functionality on `transmit`.

The `dev->hard_start_xmit()` function is called and provides the driver with its own device pointer and network buffer (an `sk_buff`) to transmit. If your device is unable to accept the buffer, it should return 1 and set `dev->tbusy` to a non-zero value. This will queue the buffer and it may be retried again later, although there is no guarantee that the buffer will be retried. If the protocol layer decides to free the buffer the driver has rejected, then it will not be offered back to the device. If the device knows the buffer cannot be transmitted in the near future, for example due to bad congestion, it can call `dev_kfree_skb()` to

dump the buffer and return 0 indicating the buffer is processed.

If there is room the buffer should be processed. The buffer handed down already contains all the headers, including link layer headers, neccessary and need only be actually loaded into the hardware for transmission. In addition, the buffer is locked. This means that the device driver has absolute ownership of the buffer until it chooses to relinquish it. The contents of an `sk_buff` remain read-only, except that you are guaranteed that the next/previous pointers are free so you can use the `sk_buff` list primitives to build internal chains of buffers.

When the buffer has been loaded into the hardware, or in the case of some DMA driven devices, when the hardware has indicated transmission complete, the driver must release the buffer. This is done by calling `dev_kfree_skb(skb, FREE_WRITE)`. As soon as this call is made, the `sk_buff` in question may spontaneously disappear and the device driver thus should not reference it again.

Frame Headers

It is neccessary for the high level protocols to append low level headers to each frame before queueing it for transmission. It is also clearly undesirable that the protocol know in advance how to append low level headers for all possible frame types. Thus the protocol layer calls down to the device with a buffer that has at least `dev->hard_header_len` bytes free at the start of the buffer. It is then up to the network device to correctly call `skb_push()` and to put the header on the packet in its `dev->hard_header()` method. Devices with no link layer header, such as SLIP, may have this method specified as NULL.

The method is invoked giving the buffer concerned, the device's own pointers, its protocol identity, pointers to the source and destination hardware addresses, and the length of the packet to be sent. As the routine may be called before the protocol layers are fully assembled, it is vital that the method use the length parameter, **not** the buffer length.

The source address may be NULL to mean ``use the default address of this device'', and the destination may be NULL to mean ``unknown''. If as a result of an unknown destination the header may not be completed, the space should be allocated and any bytes that can be filled in should be filled in. This facility is currently only used by IP when ARP processing must take place. The function must then return the negative of the bytes of header added. If the header is completely built it must return the number of bytes of header added.

When a header cannot be completed the protocol layers will attempt to resolve the address neccessary. When this occurs, the `dev->rebuild_header()` method is called with the address at which the header is located, the device in question,

the destination IP address, and the network buffer pointer. If the device is able to resolve the address by whatever means available (normally ARP), then it fills in the physical address and returns 1. If the header cannot be resolved, it returns 0 and the buffer will be retried the next time the protocol layer has reason to believe resolution will be possible.

Reception

There is no receive method in a network device, because it is the device that invokes processing of such events. With a typical device, an interrupt notifies the handler that a completed packet is ready for reception. The device allocates a buffer of suitable size with `dev_alloc_skb()` and places the bytes from the hardware into the buffer. Next, the device driver analyses the frame to decide the packet type. The driver sets `skb->dev` to the device that received the frame. It sets `skb->protocol` to the protocol the frame represents so that the frame can be given to the correct protocol layer. The link layer header pointer is stored in `skb->mac.raw` and the link layer header removed with `skb_pull()` so that the protocols need not be aware of it. Finally, to keep the link and protocol isolated, the device driver must set `skb->pkt_type` to one of the following:

```
PACKET_BROADCAST
    Link layer broadcast.
PACKET_MULTICAST
    Link layer multicast.
PACKET_SELF
    Frame to us.
PACKET_OTHERHOST
    Frame to another single host.
```

This last type is normally reported as a result of an interface running in promiscuous mode.

Finally, the device driver invokes `netif_rx()` to pass the buffer up to the protocol layer. The buffer is queued for processing by the networking protocols after the interrupt handler returns. Deferring the processing in this fashion dramatically reduces the time interrupts are disabled and improves overall responsiveness. Once `netif_rx()` is called, the buffer ceases to be property of the device driver and may not be altered or referred to again.

Flow control on received packets is applied at two levels by the protocols. Firstly a maximum amount of data may be outstanding for `netif_rx()` to process. Secondly each socket on the system has a queue which limits the amount of pending data. Thus all flow control is applied by the protocol layers. On the transmit side a per device variable `dev->tx_queue_len` is used as a queue length

limiter. The size of the queue is normally 100 frames, which is enough that the queue will be kept well filled when sending a lot of data over fast links. On a slow link such as slip link, the queue is normally set to about 10 frames, as sending even 10 frames is several seconds of queued data.

One piece of magic that is done for reception with most existing device, and one you should implement if possible, is to reserve the necessary bytes at the head of the buffer to land the IP header on a long word boundary. The existing ethernet drivers thus do:

```
skb=dev_alloc_skb(length+2);
if(skb==NULL)
    return;
skb_reserve(skb,2);
/* then 14 bytes of ethernet hardware header */
```

to align IP headers on a 16 byte boundary, which is also the start of a cache line and helps give performance improvements. On the Sparc or DEC Alpha these improvements are very noticable.

Optional Functionality

Each device has the option of providing additional functions and facilities to the protocol layers. Not implementing these functions will cause a degradation in service available via the interface but not prevent operation. These operations split into two categories--configuration and activation/shutdown.

Activation And Shutdown

When a device is activated (that is, the flag `IFF_UP` is set) the `dev->open()` method is invoked if the device has provided one. This permits the device to take any action such as enabling the interface that are needed when the interface is to be used. An error return from this function causes the device to stay down and causes the user request to activate the device to fail with the error returned by `dev->open()`

The second use of this function is with any device loaded as a module. Here it is necessary to prevent a device being unloaded while it is open. Thus the `MOD_INC_USE_COUNT` macro must be used within the open method.

The `dev->close()` method is invoked when the device is configured down and should shut off the hardware in such a way as to minimise machine load (for example by disabling the interface or its ability to generate interrupts). It can also be used to allow a module device to be unloaded now that it is down. The rest of the kernel is structured in such a way that when a device is closed, all references to it by pointer are removed. This ensures that the device may safely

be unloaded from a running system. The close method is not permitted to fail.

Configuration And Statistics

A set of functions provide the ability to query and to set operating parameters. The first and most basic of these is a `get_stats` routine which when called returns a struct `enet_statistics` block for the interface. This allows user programs such as `ifconfig` to see the loading on the interface and any problem frames logged. Not providing this will lead to no statistics being available.

The `dev->set_mac_address()` function is called whenever a superuser process issues an `ioctl` of type `SIOCSIFHWADDR` to change the physical address of a device. For many devices this is not meaningful and for others not supported. If so leave this function pointer as `NULL`. Some devices can only perform a physical address change if the interface is taken down. For these check `IFF_UP` and if set then return `-EBUSY`.

The `dev->set_config()` function is called by the `SIOCSIFMAP` function when a user enters a command like `ifconfig eth0 irq 11`. It passes an `ifmap` structure containing the desired I/O and other interface parameters. For most interfaces this is not useful and you can return `NULL`.

Finally, the `dev->do_ioctl()` call is invoked whenever an `ioctl` in the range `SIOCDEVPRIVATE` to `SIOCDEVPRIVATE+15` is used on your interface. All these `ioctl` calls take a struct `ifreq`. This is copied into kernel space before your handler is called and copied back at the end. For maximum flexibility any user may make these calls and it is up to your code to check for superuser status when appropriate. For example the PLIP driver uses these to set parallel port time out speeds to allow a user to tune the plip device for their machine.

Multicasting

Certain physical media types such as ethernet support multicast frames at the physical layer. A multicast frame is heard by a group, but not all, hosts on the network, rather than going from one host to another.

The capabilities of ethernet cards are fairly variable. Most fall into one of three categories:

1. No multicast filters. The card either receives all multicasts or none of them. Such cards can be a nuisance on a network with a lot of multicast traffic such as group video conferences.
2. Hash filters. A table is loaded onto the card giving a mask of entries that we wish to hear multicast for. This filters out some of the unwanted

multicasts but not all.

3. Perfect filters. Most cards that support perfect filters combine this option with 1 or 2 above. This is done because the perfect filter often has a length limit of 8 or 16 entries.

It is especially important that ethernet interfaces are programmed to support multicasting. Several ethernet protocols (notably Appletalk and IP multicast) rely on ethernet multicasting. Fortunately, most of the work is done by the kernel for you (see `net/core/dev_mcast.c`).

The kernel support code maintains lists of physical addresses your interface should be allowing for multicast. The device driver may return frames matching more than the requested list of multicasts if it is not able to do perfect filtering.

Whenever the list of multicast addresses changes the device drivers `dev->set_multicast_list()` function is invoked. The driver can then reload its physical tables. Typically this looks something like:

```
if(dev->flags&IFF_PROMISC)
    SetToHearAllPackets();
else if(dev->flags&IFF_ALLMULTI)
    SetToHearAllMulticasts();
else
{
    if(dev->mc_count<16)
    {
        LoadAddressList(dev->mc_list);
        SetToHearList();
    }
    else
        SetToHearAllMulticasts();
}
```

There are a small number of cards that can only do unicast or promiscuous mode. In this case the driver, when presented with a request for multicasts has to go promiscuous. If this is done, the driver must itself also set the `IFF_PROMISC` flag in `dev->flags`.

In order to aid driver writer the multicast list is kept valid at all times. This simplifies many drivers, as a reset from error condition in a driver often has to reload the multicast address lists.

Ethernet Support Routines

Ethernet is probably the most common physical interface type that is handled. The kernel provides a set of general purpose ethernet support routines that such drivers can use.







`eth_header()` is the standard ethernet handler for the `dev->hard_header` routine, and can be used in any ethernet driver. Combined with `eth_rebuild_header()` for the rebuild routine it provides all the ARP lookup required to put ethernet headers on IP packets.

The `eth_type_trans()` routine expects to be fed a raw ethernet packet. It analyses the headers and sets `skb->pkt_type` and `skb->mac` itself as well as returning the suggested value for `skb->protocol`. This routine is normally called from the ethernet driver receive interrupt handler to classify packets.

`eth_copy_and_sum()`, the final ethernet support routine, is quite internally complex but offers significant performance improvements for memory mapped cards. It provides the support to copy and checksum data from the card into an `sk_buff` in a single pass. This single pass through memory almost eliminates the cost of checksum computation when used and can really help IP throughput.

Alan Cox has been working on Linux since version 0.95, when he installed it in order to do further work on the AberMUD game. He now manages the Linux Networking, SMP, and Linux/8086 projects and hasn't done any work on AberMUD since November 1993.

Messages

- 2.  [Question on alloc_skb\(\)](#) by Joern Wohlrab
- 1.  [Re: Question on alloc_skb\(\)](#) by Erik Petersen
- 1.  [Question on network interfaces](#) by Vijay Gupta
- 2.  [Re: Question on network interfaces](#) by Pedro Roque
- 1.  [Untitled](#)
 - 1.  [Finding net info](#) by Alan Cox