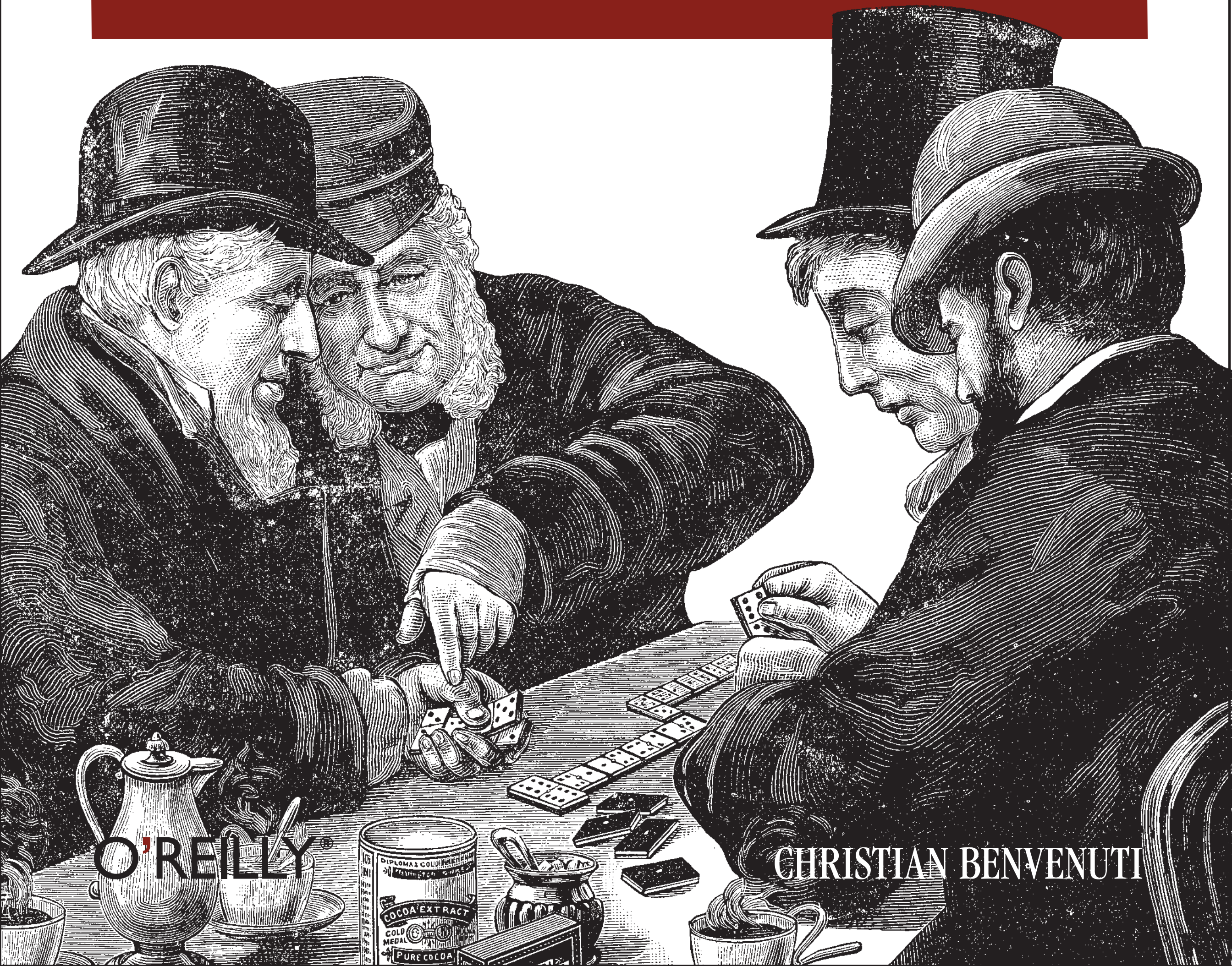
GUIDED TOUR TO NETWORKING ON LINUX

Covers TCP/IP, SLIP, NFS, Samba and More!

Understanding LINUX Network Internals



CHRISTIAN BENVENUTI

O'REILLY®

CHAPTER 10
Frame Reception

In the previous chapter, we saw that the functions that deal with frames at the L2 layer are driven by interrupts. In this chapter, we start our discussion about frame reception, where the hardware uses an interrupt to signal the CPU about the availability of the frame.

As shown in Figure 9-2 in Chapter 9, the CPU that receives an interrupt executes the `do_IRQ` function. The IRQ number causes the right handler to be invoked. The handler is typically a function within the device driver registered at device driver initialization time. IRQ function handlers are executed in interrupt mode, with further interrupts temporarily disabled.

As discussed in the section "Interrupt Handlers" in Chapter 9, the interrupt handler performs a few immediate tasks and schedules others in a bottom half to be executed later. Specifically, the interrupt handler:

1. Copies the frame into an `sk_buff` data structure.*
2. Initializes some of the `sk_buff` parameters for use later by upper network layers (notably `skb->protocol`, which identifies the higher-layer protocol handler and will play a major role in Chapter 13).
3. Updates some other parameters private to the device, which we do not consider in this chapter because they do not influence the frame's path inside the network stack.
4. Signals the kernel about the new frame by scheduling the `NET_RX_SOFTIRQ` softirq for execution.

Since a device can issue an interrupt for different reasons (new frame received, frame transmission successfully completed, etc.), the kernel is given a code along with the

* If DMA is used by the device, as is pretty common nowadays, the driver needs only to initialize a pointer (no copying is involved).

interrupt notification so that the device driver handler can process the interrupt based on the type.

Interactions with Other Features

While perusing the routines introduced in this chapter, you will often see pieces of code for interacting with optional kernel features. For features covered in this book, I will refer you to the chapter on that feature; for other features, I will not spend much time on the code. Most of the flowcharts in the chapter also show where those optional features are handled in the routines.

Here are the optional features we'll see, with the associated kernel symbols:

802.1d Ethernet Bridging (CONFIG_BRIDGE/CONFIG_BRIDGE_MODULE)
: Bridging is described in Part IV.

Netpoll (CONFIG_NETPOLL)
: Netpoll is a generic framework for sending and receiving frames by polling the network interface cards (NICs), eliminating the need for interrupts. Netpoll can be used by any kernel feature that benefits from its functionality; one prominent example is Netconsole, which logs kernel messages (i.e., strings printed with `printk`) to a remote host via UDP. Netconsole and its suboptions can be turned on from the *make xconfig* menu with the "Networking support → Network console logging support" option. To use Netpoll, devices must include support for it (which quite a few already do).

Packet Action (CONFIG_NET_CLS_ACT)
: With this feature, Traffic Control can classify and apply actions to ingress traffic. Possible actions include dropping the packet and consuming the packet. To see this option and all its suboptions from the *make xconfig* menu, you need first to select the "Networking support → Networking options → QoS and/or fair queueing → Packet classifier API" option.

Enabling and Disabling a Device

A device can be considered enabled when the `__LINK_STATE_START` flag is set in `net_device->state`. The section "Enabling and Disabling a Device" in Chapter 8 covers the details of this flag. The flag is normally set when the device is open (`dev_open`) and cleared when the device is closed (`dev_close`). While there is a flag that is used to explicitly enable and disable transmission for a device (`__LINK_STATE_XOFF`), there is none to enable and disable reception. That capability is achieved by other means—i.e., by disabling the device, as described in Chapter 8. The status of the `__LINK_STATE_START` flag can be checked with the `netif_running` function.

Several functions shown later in this chapter provide simple wrappers that check the correct status of flags such as `__LINK_STATE_START` to make sure the device is ready to do what is about to be asked of it.

Queues

When discussing L2 behavior, I often talk about queues for frames being received (ingress queues) and transmitted (egress queues). Each queue has a pointer to the devices associated with it, and to the `skb_buff` data structures that store the ingress/egress buffers. Only a few specialized devices work without queues; an example is the loopback device. The loopback device can dispense with queues because when you transmit a packet out of the loopback device, the packet is immediately delivered (to the local system) with no need for intermediate queuing. Moreover, since transmissions on the loopback device cannot fail, there is no need to requeue the packet for another transmission attempt.

Egress queues are associated directly to devices; Traffic Control (the Quality of Service, or QoS, layer) defines one queue for each device. As we will see in Chapter 11, the kernel keeps track of devices waiting to transmit frames, not the frames themselves. We will also see that not all devices actually use Traffic Control. The situation with ingress queues is a bit more complicated, as we'll see later.

Notifying the Kernel of Frame Reception: NAPI and netif_rx

In version 2.5 (then backported to a late revision of 2.4 as well), a new API for handling ingress frames was introduced into the Linux kernel, known (for lack of a better name) as NAPI. Since few devices have been upgraded to NAPI, there are two ways a Linux driver can notify the kernel about a new frame:

By means of the old function `netif_rx`
: This is the approach used by those devices that follow the technique described in the section "Processing Multiple Frames During an Interrupt" in Chapter 9. Most Linux device drivers still use this approach.

By means of the NAPI mechanism
: This is the approach used by those devices that follow the technique described in the variation introduced at the end of the section "Processing Multiple Frames During an Interrupt" in Chapter 9. This is new in the Linux kernel, and only a few drivers use it. *drivers/net/tg3.c* was the first one to be converted to NAPI.

A few device drivers allow you to choose between the two types of interfaces when you configure the kernel options with tools such as *make xconfig*.

The following piece of code comes from vortex_rx, which still uses the old function netif_rx, and you can expect most of the network device drivers not yet using NAPI to do something similar:

```
skb = dev_alloc_skb(pkt_len + 5);
        ... ... ...
if (skb != NULL) {
    skb->dev = dev;
    skb_reserve(skb, 2);    /* Align IP on 16 byte boundaries */
            ... ... ...
            /* copy the DATA into the sk_buff structure */
            ... ... ...
    skb->protocol = eth_type_trans(skb, dev);
    netif_rx(skb);
    dev->last_rx = jiffies;
        ... ... ...
}
```

First, the sk_buff data structure is allocated with dev_alloc_skb (see Chapter 2), and the frame is copied into it. Note that before copying, the code reserves two bytes to align the IP header to a 16-byte boundary. Each network device driver is associated with a given interface type; for instance, the Vortex device driver *driver/net/3c59x.c* is associated with a specific family of Ethernet cards. Therefore, the driver knows the length of the link layer's header and how to interpret it. Given a header length of $16*k+n$, the driver can force an alignment to a 16-byte boundary by simply calling skb_reserve with an offset of $16-n$. An Ethernet header is 14 bytes, so k=0, n=14, and the offset requested by the code is 2 (see the definition of NET_IP_ALIGN and the associated comment in *include/linux/sk_buff.h*).

Note also that at this stage, the driver does not make any distinction between different L3 protocols. It aligns the L3 header to a 16-byte boundary regardless of the type. The L3 protocol is probably IP because of IP's widespread usage, but that is not guaranteed at this point; it could be Netware's IPX or something else. The alignment is useful regardless of the L3 protocol to be used.

eth_type_trans, which is used to extract the protocol identifier skb->protocol, is described in Chapter 13.*

Depending on the complexity of the driver's design, the block shown may be followed by other housekeeping tasks, but we are not interested in those details in this book. The most important part of the function is the notification to the kernel about the frame's reception.

* Different device types use different functions; for instance, eth_type_trans is used by Ethernet devices and tr_type_trans by Token Ring interfaces.

Introduction to the New API (NAPI)

Even though some of the NIC device drivers have not been converted to NAPI yet, the new infrastructure has been integrated into the kernel, and even the interface between `netif_rx` and the rest of the kernel has to take NAPI into account. Instead of introducing the old approach (pure `netif_rx`) first and then talking about NAPI, we will first see NAPI and then show how the old drivers keep their old interface (`netif_rx`) while sharing some of the new infrastructure mechanisms.

NAPI mixes interrupts with polling and gives higher performance under high traffic load than the old approach, by reducing significantly the load on the CPU. The kernel developers backported that infrastructure to the 2.4 kernels.

In the old model, a device driver generates an interrupt for each frame it receives. Under a high traffic load, the time spent handling interrupts can lead to a considerable waste of resources.

The main idea behind NAPI is simple: instead of using a pure interrupt-driven model, it uses a mix of interrupts and polling. If new frames are received when the kernel has not finished handling the previous ones yet, there is no need for the driver to generate other interrupts: it is just easier to have the kernel keep processing whatever is in the device input queue (with interrupts disabled for the device), and re-enable interrupts once the queue is empty. This way, the driver reaps the advantages of both interrupts and polling:

- Asynchronous events, such as the reception of one or more frames, are indicated by interrupts so that the kernel does not have to check continuously if the device's ingress queue is empty.
- If the kernel knows there is something left in the device's ingress queue, there is no need to waste time handling interrupt notifications. A simple polling is enough.

From the kernel processing point of view, here are some of the advantages of the NAPI approach:

Reduced load on the CPU (because there are fewer interrupts)
: Given the same workload (i.e., number of frames per second), the load on the CPU is lower with NAPI. This is especially true at high workloads. At low workloads, you may actually have slightly higher CPU usage with NAPI, according to tests posted by the kernel developers on the kernel mailing list.

More fairness in the handling of devices
: We will see later how devices that have something in their ingress queues are accessed fairly in a round-robin fashion. This ensures that devices with low traffic can experience acceptable latencies even when other devices are much more loaded.

net_device Fields Used by NAPI

Before looking at NAPI's implementation and use, I need to describe a few fields of the net_device data structure, mentioned in the section "softnet_data Structure" in Chapter 9.

Four new fields have been added to this structure for use by the NET_RX_SOFTIRQ softirq when dealing with devices whose drivers use the NAPI interface. The other devices will not use them, but they will share the fields of the net_device structure embedded in the softnet_data structure as its backlog_dev field.

poll
: A virtual function used to dequeue buffers from the device's ingress queue. The queue is a private one for devices using NAPI, and softnet_data->input_pkt_queue for others. See the section "Backlog Processing: The process_backlog Poll Virtual Function."

poll_list
: List of devices that have new frames in the ingress queue waiting to be processed. These devices are known as being in *polling state*. The head of the list is softnet_data->poll_list. Devices in this list have interrupts disabled and the kernel is currently polling them.

quota
weight
: quota is an integer that represents the maximum number of buffers that can be dequeued by the poll virtual function in one shot. Its value is incremented in units of weight and it is used to enforce some sort of fairness among different devices. Lower quotas mean lower potential latencies and therefore a lower risk of starving other devices. On the other hand, a low quota increases the amount of switching among devices, and therefore overall overhead.

  For devices associated with non-NAPI drivers, the default value of weight is 64, stored in weight_p at the top of *net/core/dev.c*. The value of weight_p can be changed via */proc*.

  For devices associated with NAPI drivers, the default value is chosen by the drivers. The most common value is 64, but 16 and 32 are used, too. Its value can be tuned via *sysfs*.

  For both the */proc* and *sysfs* interfaces, see the section "Tuning via /proc and sysfs Filesystems" in Chapter 12.

The section "Old Versus New Driver Interfaces" describes how and when elements are added to poll_list, and the section "Backlog Processing: The process_backlog Poll Virtual Function" describes when the poll method extracts elements from the list and how quota is updated based on the value of weight.

Devices using NAPI initialize these four fields and other `net_device` fields according to the initialization model described in Chapter 8. For the fake `backlog_dev` devices, introduced in the section "Initialization of softnet_data" in Chapter 9 and described later in this chapter, the initialization is taken care of by `net_dev_init` (described in Chapter 5).

net_rx_action and NAPI

Figure 10-1 shows what happens each time the kernel polls for incoming network traffic. In the figure, you can see the relationships among the `poll_list` list of devices in polling state, the `poll` virtual function, and the software interrupt handler `net_rx_action`. The following sections will go into detail on each aspect of that diagram, but it is important to understand how the parts interact before moving to the source code.

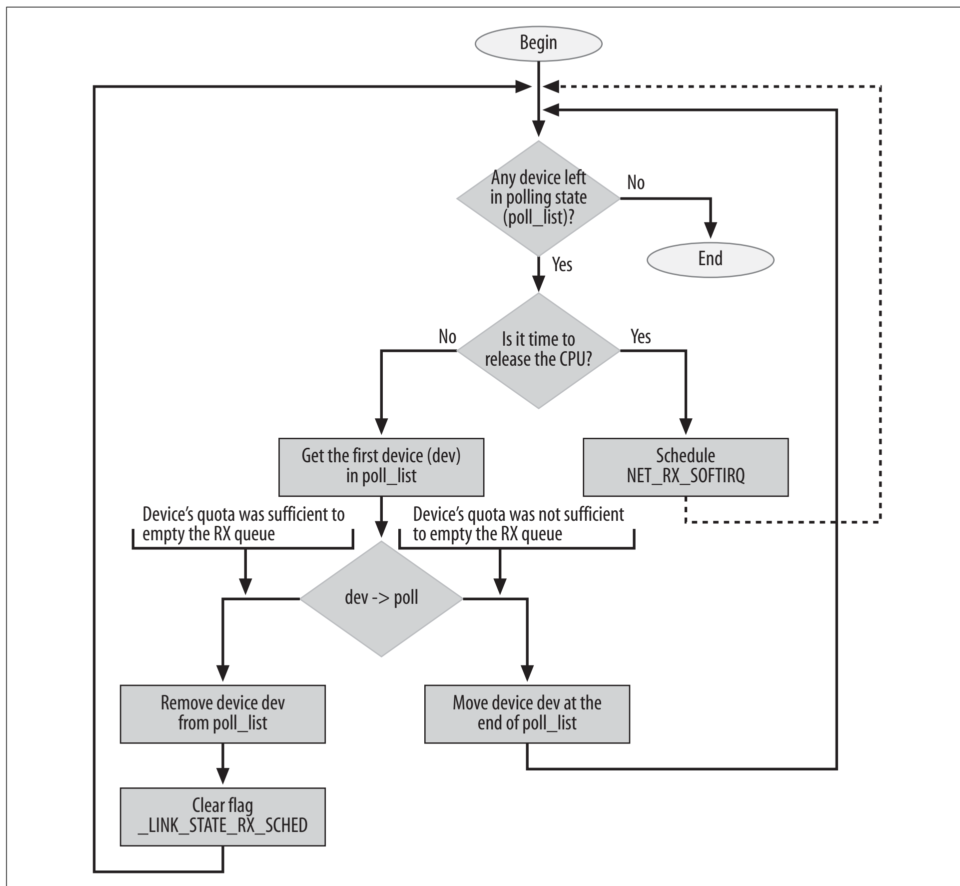


Figure 10-1. net_rx_action function and NAPI overview

We already know that net_rx_action is the function associated with the NET_RX_SOFTIRQ flag. For the sake of simplicity, let's suppose that after a period of very low activity, a few devices start receiving frames and that these somehow trigger the execution of net_rx_action—how they do so is not important for now.

net_rx_action browses the list of devices in polling state and calls the associated poll virtual function for each device to process the frames in the ingress queue. I explained earlier that devices in that list are consulted in a round-robin fashion, and that there is a maximum number of frames they can process each time their poll method is invoked. If they cannot clear the queue during their slot, they have to wait for their next slot to continue. This means that net_rx_action keeps calling the poll method provided by the device driver for a device with something in its ingress queue until the latter empties out. At that point, there is no need anymore for polling, and the device driver can re-enable interrupt notifications for the device. It is important to underline that interrupts are disabled only for those devices in poll_list, which applies only to devices that use NAPI and do not share backlog_dev.

net_rx_action limits its execution time and reschedules itself for execution when it passes a given limit of execution time or processed frames; this is enforced to make net_rx_action behave fairly in relation to other kernel tasks. At the same time, each device limits the number of frames processed by each invocation of its poll method to be fair in relation to other devices. When a device cannot clear out its ingress queue, it has to wait until the next call of its poll method.

Old Versus New Driver Interfaces

Now that the meaning of the NAPI-related fields of the net_device structure, and the high-level idea behind NAPI, should be clear, we can get closer to the source code.

Figure 10-2 shows the difference between a NAPI-aware driver and the others with regard to how the driver tells the kernel about the reception of new frames.

From the device driver perspective, there are only two differences between NAPI and non-NAPI. The first is that NAPI drivers must provide a poll method, described in the section "net_device fields used by NAPI." The second difference is the function called to schedule a frame: non-NAPI drivers call netif_rx, whereas NAPI drivers call __netif_rx_schedule, defined in *include/linux/netdevice.h*. (The kernel provides a wrapper function named netif_rx_schedule, which checks to make sure that the device is running and that the softirq is not already scheduled, and then it calls __netif_rx_schedule. These checks are done with netif_rx_schedule_prep. Some drivers call netif_rx_schedule, and others call netif_rx_schedule_prep explicitly and then __netif_rx_schedule if needed).

As shown in Figure 10-2, both types of drivers queue the input device to a polling list (poll_list), schedule the NET_RX_SOFTIRQ software interrupt for execution, and therefore end up being handled by net_rx_action. Even though both types of drivers

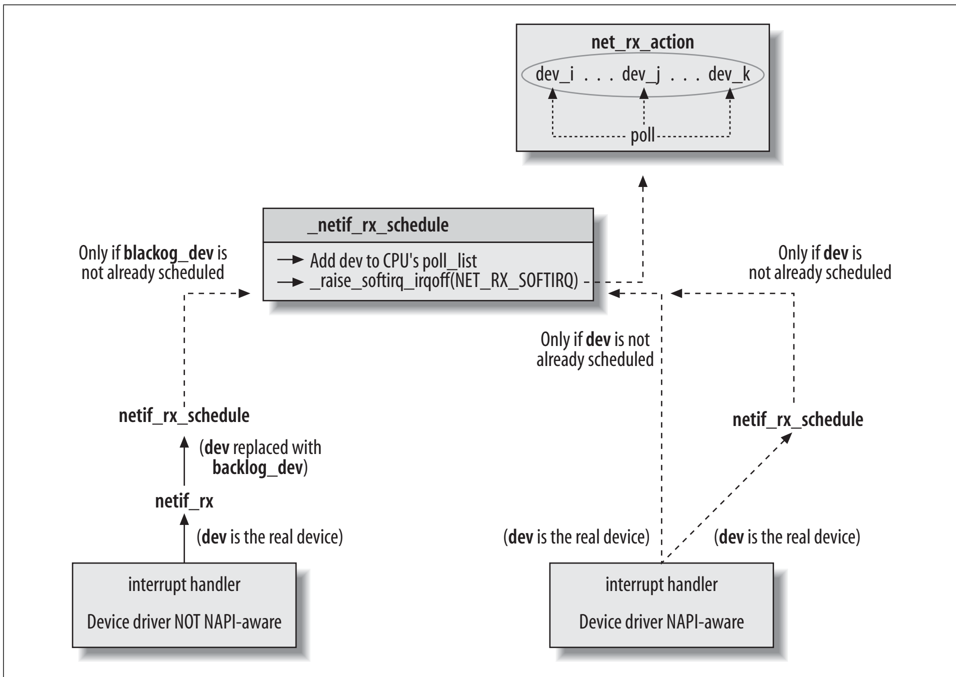


Figure 10-2. NAPI-aware drivers versus non-NAPI-aware devices

ultimately call `__netif_rx_schedule` (non-NAPI drivers do so within `netif_rx`), the NAPI devices offer potentially much better performance for the reasons we saw in the section "Notifying Drivers When Frames Are Received" in Chapter 9.

An important detail in Figure 10-2 is the `net_device` structure that is passed to `__netif_rx_schedule` in the two cases. Non-NAPI devices use the one that is built into the CPU's `softnet_data` structure, and NAPI devices use `net_device` structures that refer to themselves.

Manipulating poll_list

We saw in the previous section that any device (including the fake one, `backlog_dev`) is added to the `poll_list` list with a call to `netif_rx_schedule` or `__netif_rx_schedule`.

The reverse operation, removing a device from the list, is done with `netif_rx_complete` or `__netif_rx_complete` (the second one assumes interrupts are already disabled on the local CPU). We will see when these two routines are called in the section "Processing the NET_RX_SOFTIRQ: net_rx_action."

A device can also temporarily disable and re-enable polling with `netif_poll_disable` and `netif_poll_enable`, respectively. This does not mean that the device driver has decided to revert to an interrupt-based model. Polling might be disabled on a device,

for instance, when the device needs to be reset by the device driver to apply some kind of hardware configuration changes.

I already said that `netif_rx_schedule` filters requests for devices that are already in the `poll_list` (i.e., that have the `__LINK_STATE_RX_SCHED` flag set). For this reason, if a driver sets that flag but does not add the device to `poll_list`, it basically disables polling for the device: the device will never be added to `poll_list`. This is how `netif_poll_disable` works: if `__LINK_STATE_RX_SCHED` was not set, it simply sets it and returns. Otherwise, it waits for it to be cleared and then sets it.

```
static inline void netif_poll_disable(struct net_device *dev)
{
    while (test_and_set_bit(__LINK_STATE_RX_SCHED, &dev->state)) {
        /* No hurry. */
        current->state = TASK_INTERRUPTIBLE:
        schedule_timeout(1);
    }
}
```

Old Interface Between Device Drivers and Kernel: First Part of netif_rx

The `netif_rx` function, defined in *net/core/dev.c*, is normally called by device drivers when new input frames are waiting to be processed;* its job is to schedule the softirq that runs shortly to dequeue and handle the frames. Figure 10-3 shows what it checks for and the flow of its events. The figure is practically longer than the code, but it is useful to help understand how `netif_rx` reacts to its context.

`netif_rx` is usually called by a driver while in interrupt context, but there are exceptions, notably when the function is called by the loopback device. For this reason, `netif_rx` disables interrupts on the local CPU when it starts, and re-enables them when it finishes.†

When looking at the code, one should keep in mind that different CPUs can run `netif_rx` concurrently. This is not a problem, since each CPU is associated with a private `softnet_data` structure that maintains state information. Among other things, the CPU's `softnet_data` structure includes a private input queue (see the section "softnet_data Structure" in Chapter 9).

* There is an interesting exception: when a CPU of an SMP system dies, the `dev_cpu_callback` routine drains the `input_pkt_queue` queue of the associated `softnet_data` instance. `dev_cpu_callback` is the callback routine registered by `net_dev_init` in the `cpu_chain` introduced in Chapter 9.

† `netif_rx_ni` is a sister to `netif_rx` and is used in noninterrupt contexts. Among the systems using it is the TUN (Universal TUN/TAP) device driver in *drivers/net/tun.c*.

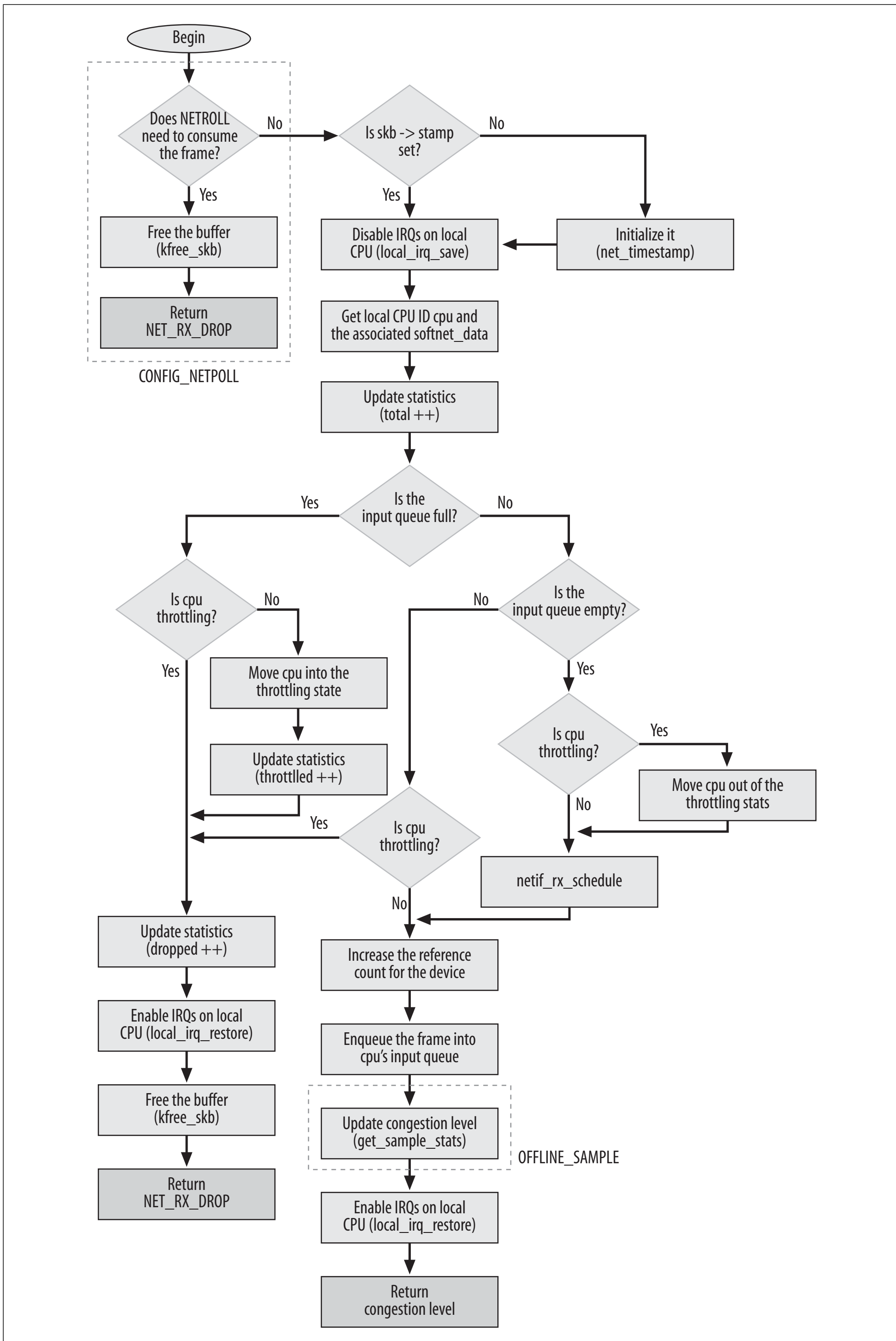


Figure 10-3. netif_rx function

This is the function's prototype:

```
int netif_rx(struct sk_buff *skb)
```

Its only input parameter is the buffer received by the device, and the output value is an indication of the congestion level (you can find details in the section "Congestion Management").

The main tasks of netif_rx, whose detailed flowchart is depicted in Figure 10-3, include:

- Initializing some of the sk_buff data structure fields (such as the time the frame was received).
- Storing the received frame onto the CPU's private input queue and notifying the kernel about the frame by triggering the associated softirq NET_RX_SOFTIRQ. This step takes place only if certain conditions are met, the most important of which is whether there is space in the queue.
- Updating the statistics about the congestion level.

Figure 10-4 shows an example of a system with a bunch of CPUs and devices. Each CPU has its own instance of softnet_data, which includes the private input queue where netif_rx will store ingress frames, and the completion_queue where buffers are sent when they are not needed anymore (see the section "Processing the NET_TX_SOFTIRQ: net_tx_action" in Chapter 11). The figure shows an example where CPU 1 receives an RxComplete interrupt from *eth0*. The associated driver stores the ingress frame into CPU 1's queue. CPU m receives a DMADone interrupt from *ethn* saying that the transmitted buffer is not needed anymore and can therefore be moved to the completion_queue queue.*

Initial Tasks of netif_rx

netif_rx starts by saving the time the function was invoked (which also represents the time the frame was received) into the stamp field of the buffer structure:

```
if (skb->stamp.tv_sec == 0)
        net_timestamp(&skb->stamp);
```

Saving the timestamp has a CPU cost—therefore, net_timestamp initializes skb->stamp only if there is at least one interested user for that field. Interest in the field can be advertised by calling net_enable_timestamp.

Do not confuse this assignment with the one done by the device driver right before or after it calls netif_rx:

```
netif_rx(skb);
dev->last_rx = jiffies;
```

* Both input_pkt_queue and completion_queue keep only the pointers to the buffers, even if the figure makes it look as if they actually store the complete buffers.

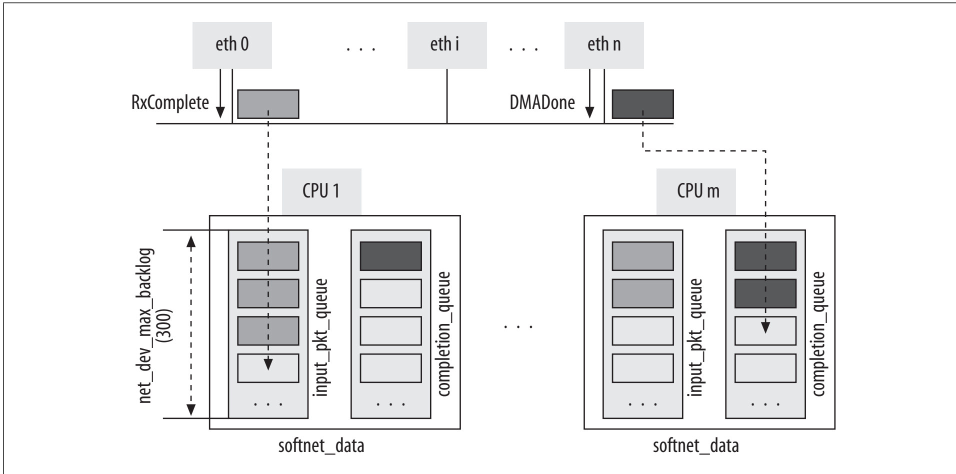


Figure 10-4. CPU's ingress queues

The device driver stores in the `net_device` structure the time its *most recent* frame was received, and `netif_rx` stores the time the frame was received in the buffer itself. Thus, one timestamp is associated with a device and the other one is associated with a frame. Note, moreover, that the two timestamps use two different precisions. The device driver stores the timestamp of the most recent frame in `jiffies`, which in kernel 2.6 comes with a precision of 10 or 1 ms, depending on the architecture (for instance, before 2.6, the i386 used the value 10, but starting with 2.6 the value is 1). `netif_rx`, however, gets its timestamp by calling `get_fast_time`, which returns a far more precise value.

The ID of the local CPU is retrieved with `smp_processor_id( )` and is stored in the local variable `this_cpu`:

```
this_cpu = smp_processor_id( );
```

The local CPU ID is needed to retrieve the data structure associated with that CPU in a per-CPU vector, such as the following code in `netif_rx`:

```
queue = &__get_cpu_var(softnet_data);
```

The preceding line stores in `queue` a pointer to the `softnet_data` structure associated with the local CPU that is serving the interrupt triggered by the device driver that called `netif_rx`.

Now `netif_rx` updates the total number of frames received by the CPU, including both the ones accepted and the ones discarded (because there was no space in the queue, for instance):

```
netdev_rx_stat[this_cpu].total++
```

Each device driver also keeps statistics, storing them in the private data structure that `dev->priv` points to. These statistics, which include the number of received frames,

the number of dropped frames, etc., are kept on a per-device basis (see Chapter 2), and the ones updated by `netif_rx` are on a per-CPU basis.

Managing Queues and Scheduling the Bottom Half

The input queue is managed by `softnet_data->input_pkt_queue`. Each input queue has a maximum length given by the global variable `netdev_max_backlog`, whose value is 300. This means that each CPU can have up to 300 frames in its input queue waiting to be processed, regardless of the number of devices in the system.*

Common sense would say that the value of `netdev_max_backlog` should depend on the number of devices and their speeds. However, this is hard to keep track of in an SMP system where the interrupts are distributed dynamically among the CPUs. It is not obvious which device will talk to which CPU. Thus, the value of `netdev_max_backlog` is chosen through trial and error. In the future, we could imagine it being set dynamically in a manner reflecting the types and number of interfaces. Its value is already configurable by the system administrator, as described in the section "Tuning via /proc and sysfs Filesystems" in Chapter 12. The performance issues are as follows: an unnecessarily large value is a waste of memory, and a slow system may simply never be able to catch up. A value that is too small, on the other hand, could reduce the performance of the device because a burst of traffic could lead to many dropped frames. The optimal value depends a lot on the system's role (host, server, router, etc.).

In the previous kernels, when the `softnet_data` per-CPU data structure was not present, a single input queue, called `backlog`, was shared by all devices with the same size of 300 frames. The main gain with `softnet_data` is not that *n* CPUs leave room on the queues for *n**300 frames, but rather, that there is no need for locking among CPUs because each has its own queue.

The following code controls the conditions under which `netif_rx` inserts its new frame on a queue, and the conditions under which it schedules the queue to be run:

```
        if (queue->input_pkt_queue.qlen <= netdev_max_backlog) {
            if (queue->input_pkt_queue.qlen) {
                if (queue->throttle)
                    goto drop;

    enqueue:
                dev_hold(skb->dev);
                __skb_queue_tail(&queue->input_pkt_queue,skb);
    #ifndef OFFLINE_SAMPLE
                get_sample_stats(this_cpu);
    #endif
```

* This applies to non-NAPI devices. Because NAPI devices use private queues, the devices can select the maximum length they prefer. Common values are 16, 32, and 64. The 10-Gigabit Ethernet driver *drivers/net/s2io.c* uses a larger value (90).

```
                local_irq_restore(flags);
            return queue->cng_level;
        }

        if (queue->throttle)
            queue->throttle = 0;

        netif_rx_schedule(&queue->backlog_dev);
        goto enqueue;
    }

    ... ... ...

drop:
    __get_cpu_var(netdev_rx_stat).dropped++;
    local_irq_restore(flags);

    kfree_skb(skb);
    return NET_RX_DROP;
}
```

The first `if` statement determines whether there is space. If the queue is full and the statement returns a false result, the CPU is put into a throttle state, which means that it is overloaded by input traffic and therefore is dropping all further frames. The code instituting the throttle is not shown here, but appears in the following section on congestion management.

If there is space on the queue, however, that is not sufficient to ensure that the frame is accepted. The CPU could already be in the "throttle" state (as determined by the third `if` statement), in which case, the frame is dropped.

The throttle state can be lifted when the queue is empty. This is what the second `if` statement tests for. When there is data on the queue and the CPU is in the throttle state, the frame is dropped. But when the queue is empty and the CPU is in the throttle state (which an `if` statement tests for in the second half of the code shown here), the throttle state is lifted.*

The `dev_hold(skb->dev)` call increases the reference count for the device so that the device cannot be removed until this buffer has been completely processed. The corresponding decrement, done by `dev_put`, takes place inside `net_rx_action`, which we will analyze later in this chapter.

If all tests are satisfactory, the buffer is queued into the input queue with `__skb_queue_tail(&queue->input_pkt_queue,skb)`, the IRQ's status is restored for the CPU, and the function returns.

* This case is actually rare because `net_rx_action` probably lifts the throttle state (indirectly via `process_backlog`) earlier. We will see this later in this chapter.

Queuing the frame is extremely fast because it does not involve any memory copying, just pointer manipulation. `input_pkt_queue` is a list of pointers. `__skb_queue_tail` adds the pointer to the new buffer to the list, without copying the buffer.

The `NET_RX_SOFTIRQ` software interrupt is scheduled for execution with `netif_rx_schedule`. Note that `netif_rx_schedule` is called only when the new buffer is added to an empty queue. The reason is that if the queue is not empty, `NET_RX_SOFTIRQ` has already been scheduled and there is no need to do it again.

In the section "Pending softirq Handling" in Chapter 9, we saw how the kernel takes care of scheduled software interrupts. In the upcoming section "Processing the NET_RX_SOFTIRQ: net_rx_action," we will see the internals of the `NET_RX_SOFTIRQ` softirq's handler.

Congestion Management

Congestion management is an important component of the input frame-processing task. An overloaded CPU can become unstable and introduce a big latency into the system. The section "Interrupts" in Chapter 9 explained why the interrupts generated by a high load can cripple the system. For this reason, congestion management mechanisms are needed to make sure the system's stability is not compromised under high network load. Common ways to reduce the CPU load under high traffic loads include:

Reducing the number of interrupts if possible
: This is accomplished by coding drivers either to process several frames with a single interrupt (see the section "Processing Multiple Frames During an Interrupt" in Chapter 9), or to use NAPI.

Discarding frames as early as possible in the ingress path
: If code knows that a frame is going to be dropped by higher layers, it can save CPU time by dropping the frame quickly. For instance, if a device driver knew that the ingress queue was full, it could drop a frame right away instead of relaying it to the kernel and having the latter drop it.

The second point is what we cover in this section.

A similar optimization applies to the egress path: if a device driver does not have resources to accept new frames for transmission (that is, if the device is out of memory), it would be a waste of CPU time to have the kernel pushing new frames down to the driver for transmission. This point is discussed in Chapter 11 in the section "Enabling and Disabling Transmissions."

In both cases, reception and transmission, the kernel provides a set of functions to set, clear, and retrieve the status of the receive and transmit queues, which allows device drivers (on reception) and the core kernel (on transmission) to perform the optimizations just mentioned.

A good indication of the congestion level is the number of frames that have been received and are waiting to be processed. When a device driver uses NAPI, it is up to the driver to implement any congestion control mechanism. This is because ingress frames are kept in the NIC's memory or in the receive ring managed by the driver, and the kernel cannot keep track of traffic congestion. In contrast, when a device driver does not use NAPI, frames are added to per-CPU queues (`softnet_data->input_pkt_queue`) and the kernel keeps track of the congestion level of the queues. In this section, we cover this latter case.

Queue theory is a complex topic, and this book is not the place for the mathematical details. I will content myself with one simple point: the current number of frames in the queue does not necessarily represent the real congestion level. An average queue length is a better guide to the queue's status. Keeping track of the average keeps the system from wrongly classifying a burst of traffic as congestion. In the Linux network stack, average queue length is reported by two fields of the `softnet_data` structure, `cng_level` and `avg_blog`, that were introduced in "softnet_data Structure" in Chapter 9.

Being an average, `avg_blog` could be both bigger and smaller than the length of `input_pkt_queue` at any time. The former represents recent history and the latter represents the present situation. Because of that, they are used for two different purposes:

- By default, every time a frame is queued into `input_pkt_queue`, `avg_blog` is updated and an associated congestion level is computed and saved into `cng_level`. The latter is used as the return value by `netif_rx` so that the device driver that called this function is given a feedback about the queue status and can change its behavior accordingly.
- The number of frames in `input_pkt_queue` cannot exceed a maximum size. When that size is reached, following frames are dropped because the CPU is clearly overwhelmed.

Let's go back to the computation and use of the congestion level. `avg_blog` and `cng_level` are updated inside `get_sample_stats`, which is called by `netif_rx`.

At the moment, few device drivers use the feedback from `netif_rx`. The most common use of this feedback is to update statistics local to the device drivers. For a more interesting use of the feedback, see *drivers/net/tulip/de2104x.c*: when `netif_rx` returns `NET_RX_DROP`, a local variable drop is set to 1, which causes the main loop to start dropping the frames in the receive ring instead of processing them.

So long as the ingress queue `input_pkt_queue` is not full, it is the job of the device driver to use the feedback from `netif_rx` to handle congestion. When the situation gets worse and the input queue fills in, the kernel comes into play and uses the `softnet_data->throttle` flag to disable frame reception for the CPU. (Remember that there is a `softnet_data` structure for each CPU.)

Congestion Management in netif_rx

Let's go back to netif_rx and look at some of the code that was omitted from the previous section of this chapter. The following two excerpts include some of the code shown previously, along with new code that shows when a CPU is placed in the throttle state.

```
if (queue->input_pkt_queue.qlen <= netdev_max_backlog) {
    if (queue->input_pkt_queue.qlen) {
        if (queue->throttle)
            goto drop;
        ... ... ...
        return queue->cng_level;
    }
    ... ...    ...
}

if (!queue->throttle) {
    queue->throttle = 1;
    __get_cpu_var(netdev_rx_stat).throttled++;
}
```

softnet_data->throttle is cleared when the queue gets empty. To be exact, it is cleared by netif_rx when the first frame is queued into an empty queue. It could also happen in process_backlog, as we will see in the section "Backlog Processing: The process_backlog Poll Virtual Function."

Average Queue Length and Congestion-Level Computation

The value of avg_blog and cng_level is always updated within get_sample_stats. The latter can be invoked in two different ways:

- Every time a new frame is received (netif_rx). This is the default.
- With a periodic timer. To use this technique, one has to define the OFFLINE_SAMPLE symbol. That's the reason why in netif_rx, the execution of get_sample_stats depends on the definition of the OFFLINE_SAMPLE symbol. It is disabled by default.

The first approach ends up running get_sample_stats more often than the second approach under medium and high traffic load.

In both cases, the formula used to compute avg_blog should be simple and quick, because it could be invoked frequently. The formula used takes into account the recent history and the present:

```
new_value_for_avg_blog = (old_value_of_avg_blog + current_value_of_queue_len) / 2
```

How much to weight the present and the past is not a simple problem. The preceding formula can adapt quickly to changes in the congestion level, since the past (the old value) is given only 50% of the weight and the present the other 50%.

get_sample_stats also updates cng_level, basing it on avg_blog through the mapping shown earlier in Figure 9-4 in Chapter 9. If the RAND_LIE symbol is defined, the function performs an extra operation in which it can randomly decide to set cng_level one level higher. This random adjustment requires more time to calculate but, oddly enough, can cause the kernel to perform better under one specific scenario.

Let's spend a few more words on the benefits of random lies. Do not confuse this behavior with Random Early Detection (RED).

In a system with only one interface, it does not really make sense to drop random frames here and there if there is no congestion; it would simply lower the throughput. But let's suppose we have multiple interfaces sharing an input queue and one device with a traffic load much higher than the others. Since the greedy device fills the shared ingress queue faster than the other devices, the latter will often find no space in the ingress queue and therefore their frames will be dropped.* The greedy device will also see some of its frames dropped, but not proportionally to its load. When a system with multiple interfaces experiences congestion, it should drop ingress frames across all the devices proportionally to their loads. The RAND_LIE code adds some fairness when used in this context: dropping extra frames randomly should end up dropping them proportionally to the load.

Processing the NET_RX_SOFTIRQ: net_rx_action

net_rx_action is the bottom-half function used to process incoming frames. Its execution is triggered whenever a driver notifies the kernel about the presence of input frames. Figure 10-5 shows the flow of control through the function.

Frames can wait in two places for net_rx_action to process them:

A shared CPU-specific queue
: Non-NAPI devices' interrupt handlers, which call netif_rx, place frames into the softnet_data->input_pkt_queue of the CPU on which the interrupt handlers run.

Device memory
: The poll method used by NAPI drivers extracts frames directly from the device (or the device driver receive rings).

The section "Old Versus New Driver Interfaces" showed how the kernel is notified about the need to run net_rx_action in both cases.

* When sharing a queue, it is up to the users to behave fairly with others, but that's not always possible. NAPI does not encounter this problem because each device using NAPI has its own queue. However, non-NAPI drivers still using the shared input queue input_pkt_queue have to live with the possibility of overloading by other devices.

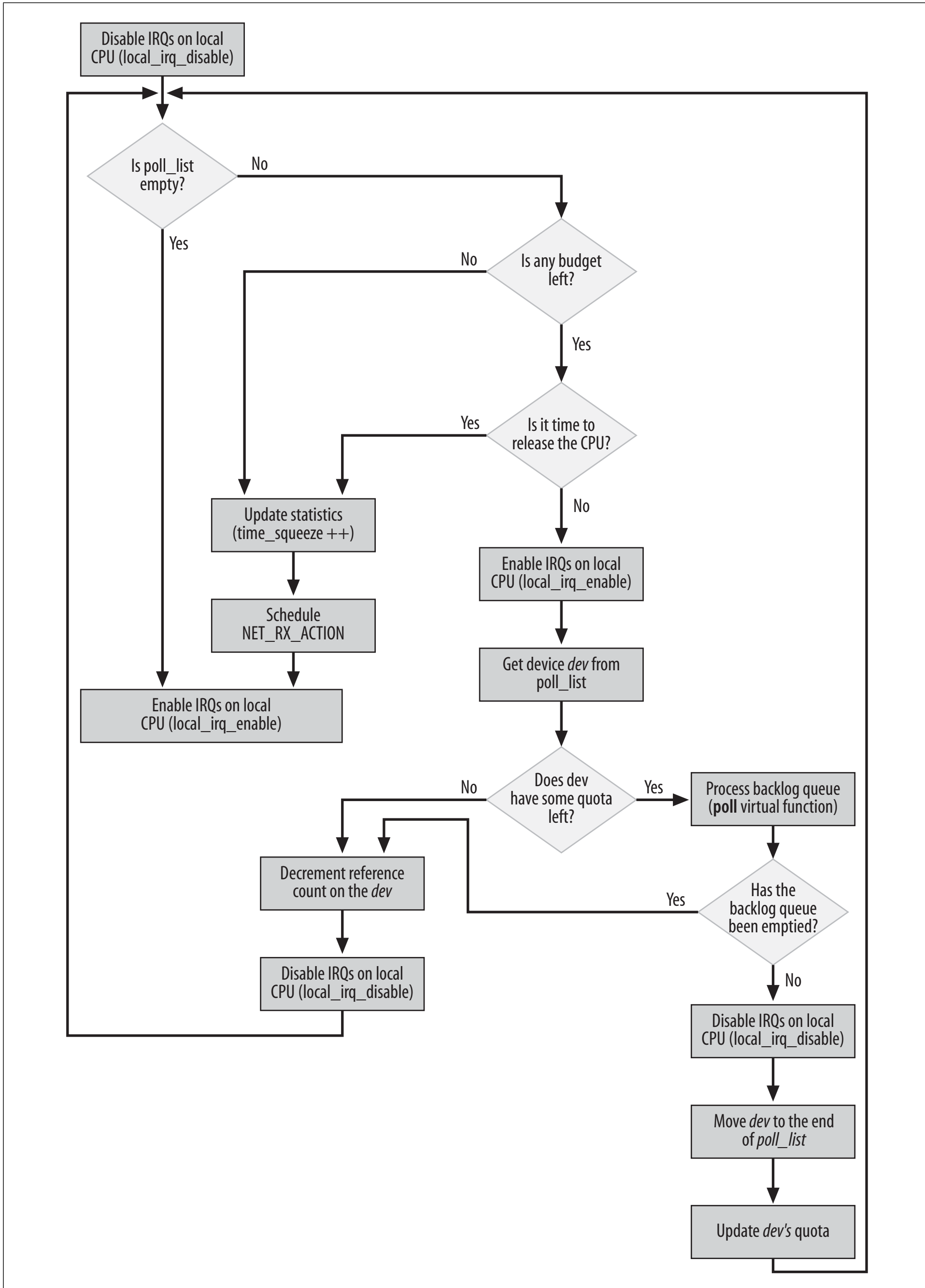


Figure 10-5. net_rx_action function

The job of net_rx_action is pretty simple: to browse the poll_list list of devices that have something in their ingress queue and invoke for each one the associated poll virtual function until one of the following conditions is met:

- There are no more devices in the list.
- net_rx_action has run for too long and therefore it is supposed to release the CPU so that it does not become a CPU hog.
- The number of frames already dequeued and processed has reached a given upper bound limit (budget). budget is initialized at the beginning of the function to netdev_max_backlog, which is defined in *net/core/dev.c* as 300.

As we will see in the next section, net_rx_action calls the driver's poll virtual function and depends partly on this function to obey these constraints.

The size of the queue, as we saw in the section "Managing Queues and Scheduling the Bottom Half," is restricted to the value of netdev_max_backlog. This value is considered the *budget* for net_rx_action. However, because net_rx_action runs with interrupts enabled, new frames could be added to a device's input queue while net_rx_action is running. Thus, the number of available frames could become greater than budget, and net_rx_action has to take action to make sure it does not run too long in such cases.

Now we will see in detail what net_rx_action does inside:

```
static void net_rx_action(struct softirq_action *h)
{
    struct softnet_data *queue = &__get_cpu_var(softnet_data);
    unsigned long start_time = jiffies;
    int budget = netdev_max_backlog;

    local_irq_disable( );
```

If the current device has not yet used its entire quota, it is given a chance to dequeue buffers from its queue with the poll virtual function:

```
    while (!list_empty(&queue->poll_list)) {
        struct net_device *dev;

        if (budget <= 0 || jiffies - start_time > 1)
            goto softnet_break;

        local_irq_enable( );

        dev = list_entry(queue->poll_list.next, struct net_device, poll_list);
```

If dev->poll returns because the device quota was not large enough to dequeue all the buffers in the ingress queue (in which case, the return value is nonzero), the device is moved to the end of poll_list:

```
        if (dev->quota <= 0 || dev->poll(dev, &budget)) {
            local_irq_disable( );
```

```
                list_del(&dev->poll_list);
                list_add_tail(&dev->poll_list, &queue->poll_list);
                if (dev->quota < 0)
                    dev->quota += dev->weight;
                else
                    dev->quota = dev->weight;
            } else {
```

When instead poll manages to empty the device ingress queue, net_rx_action does not remove the device from poll_list: poll is supposed to take care of it with a call to netif_rx_complete (__netif_rx_complete can also be called if IRQs are disabled on the local CPU). This will be illustrated in the process_backlog function in the next section.

Furthermore, note that budget was passed by reference to the poll virtual function; this is because that function will return a new budget that reflects the frames it processed. The main loop in net_rx_action checks budget at each pass so that the overall limit is not exceeded. In other words, budget allows net_rx_action and the poll function to cooperate to stay within their limit.

```
                dev_put(dev);
                local_irq_disable( );
            }
        }
    out:
        local_irq_enable( );
        return;
```

This last piece of code is executed when net_rx_action is forced to return while buffers are still left in the ingress queue. In this case, the NET_RX_SOFTIRQ softirq is scheduled again for execution so that net_rx_action will be invoked later and will take care of the remaining buffers:

```
    softnet_break:
        __get_cpu_var(netdev_rx_stat).time_squeeze++;
        __raise_softirq_irqoff(NET_RX_SOFTIRQ);
        goto out;
    }
```

Note that net_rx_action disables interrupts with local_irq_disable only while manipulating the poll_list list of devices to poll (i.e., when accessing its softnet_data structure instance). The netpoll_poll_lock and netpoll_poll_unlock calls, used by the NETPOLL feature, have been omitted. If you can access the kernel source code, see *net_rx_action* in *net/core/dev.c* for details.

Backlog Processing: The process_backlog Poll Virtual Function

The poll virtual function of the net_device data structure, which is executed by net_rx_action to process the backlog queue of a device, is initialized by default to process_backlog in net_dev_init for those devices not using NAPI.

As of kernel 2.6.12, only a few device drivers use NAPI, and initialize `dev->poll` with a pointer to a function of its own: the Broadcom Tigon3 Ethernet driver in *drivers/net/tg3.c* was the first one to adopt NAPI and is a good example to look at. In this section, we will analyze the default handler `process_backlog` defined in *net/core/dev.c*. Its implementation is very similar to that of a `poll` method of a device driver using NAPI (you can, for instance, compare `process_backlog` to `tg3_poll`).

However, since `process_backlog` can take care of a bunch of devices sharing the same ingress queue, there is one important difference to take into account. When `process_backlog` runs, hardware interrupts are enabled, so the function could be preempted. For this reason, accesses to the `softnet_data` structure are always protected by disabling interrupts on the local CPU with `local_irq_disable`, especially the calls to `__skb_dequeue`. This lock is not needed by a device driver using NAPI:* when its `poll` method is invoked, hardware interrupts are disabled for the device. Moreover, each device has its own queue.

Let's see the main parts of `process_backlog`. Figure 10-6 shows its flowchart.

The function starts with a few initializations:

```
static int process_backlog(struct net_device *backlog_dev, int *budget)
{
    int work = 0;
    int quota = min(backlog_dev->quota, *budget);
    struct softnet_data *queue = &__get_cpu_var(softnet_data);
    unsigned long start_time = jiffies;
```

Then begins the main loop, which tries to dequeue all the buffers in the input queue and is interrupted only if one of the following conditions is met:

- The queue becomes empty.
- The device's quota has been used up.
- The function has been running for too long.

The last two conditions are similar to the ones that constrain `net_rx_action`. Because `process_backlog` is called within a loop in `net_rx_action`, the latter can respect its constraints only if `process_backlog` cooperates. For this reason, `net_rx_action` passes its leftover budget to `process_backlog`, and the latter sets its quota to the minimum of that input parameter (`budget`) and its own quota.

`budget` is initialized by `net_rx_action` to 300 when it starts. The default value for `dev->quota` is 64 (and most devices stick with the default). Let's examine a case where several devices have full queues. The first four devices to run within this function receive a value of `budget` greater than their internal quota of 64, and can empty their queues.

* Because each CPU has its own instance of `softnet_data`, there is no need for extra locking to take care of SMP.

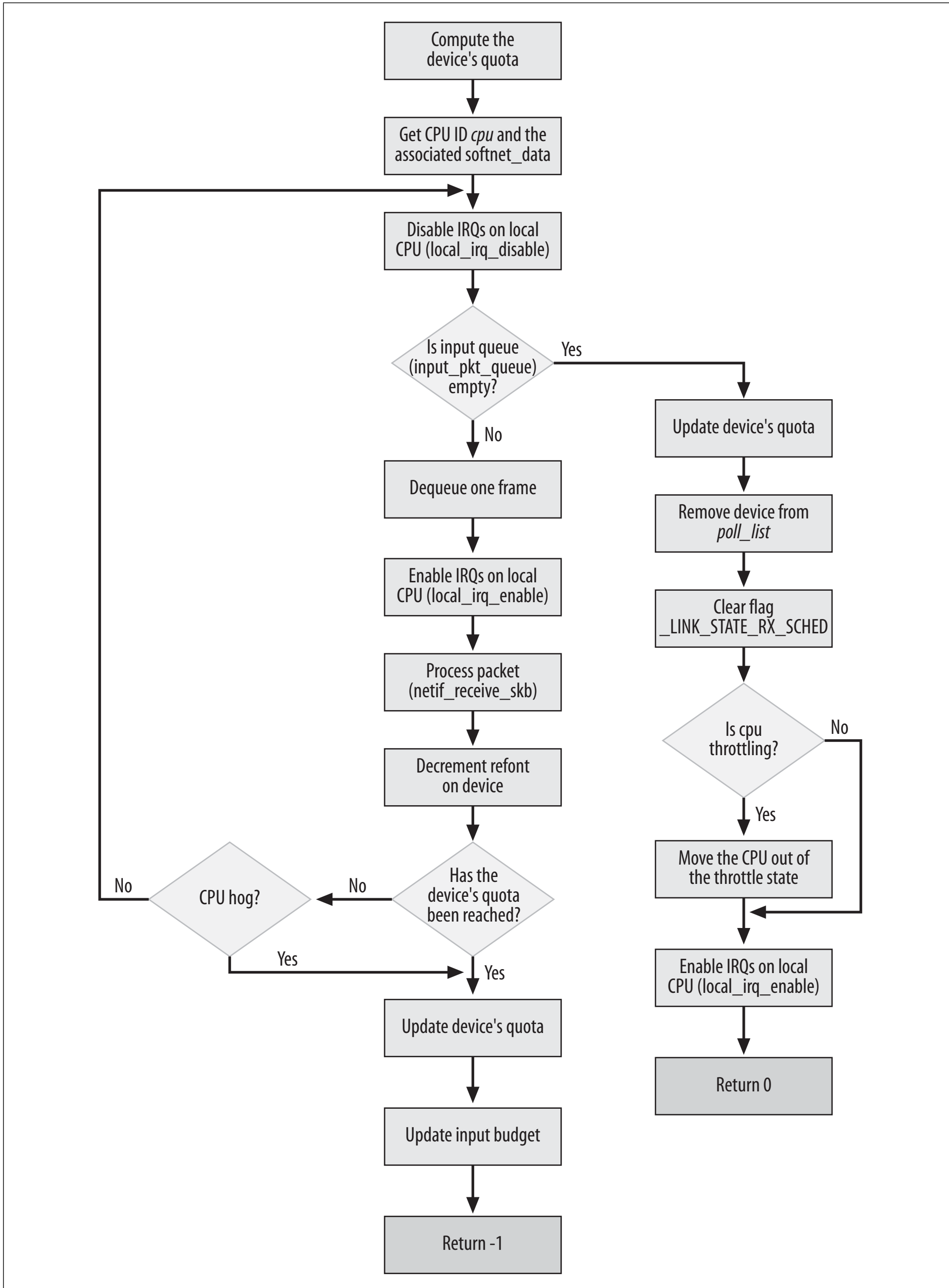


Figure 10-6. process_backlog function

The next device may have to stop after sending a part of its queue. That is, the number of buffers dequeued by `process_backlog` depends both on the device

configuration (dev->quota), and on the traffic load on the other devices (budget). This ensures some more fairness among the devices.

```
        for (;;) {
            struct sk_buff *skb;
            struct net_device *dev;

            local_irq_disable( );
            skb = __skb_dequeue(&queue->input_pkt_queue);
            if (!skb)
                goto job_done;
            local_irq_enable( );

            dev = skb->dev;

            netif_receive_skb(skb);

            dev_put(dev);

            work++;
            if (work >= quota || jiffies - start_time > 1)
                break;
```

netif_receive_skb is the function that processes the frame; it is described in the next section. It is used by all poll virtual functions, both NAPI and non-NAPI.

The device's quota is updated based on the number of buffers successfully dequeued. As explained earlier, the input parameter budget is also updated because it is needed by net_rx_action to keep track of how much work it can continue to do:

```
        backlog_dev->quota -= work;
        *budget -= work;
        return -1;
```

The main loop shown earlier jumps to the label job_done if the input queue is emptied. If the function reaches this point, the throttle state can be cleared (if it was set) and the device can be removed from poll_list. The __LINK_STATE_RX_SCHED flag is also cleared since the device does not have anything in the input queue and therefore it does not need to be scheduled for backlog processing.

```
    job_done:
        backlog_dev->quota -= work;
        *budget -= work;

        list_del(&backlog_dev->poll_list);
        smp_mb__before_clear_bit( );
        netif_poll_enable(backlog_dev);

        if (queue->throttle)
            queue->throttle = 0;
        local_irq_enable( );
        return 0;
    }
```

Actually, there is another difference between process_backlog and a NAPI driver's poll method. Let's return to *drivers/net/tg3.c* as an example:

```
if (done) {
        spin_lock_irqsave(&tp->lock, flags);
    __netif_rx_complete(netdev);
    tg3_restart_ints(tp);
        spin_unlock_irqrestore(&tp->lock, flags);
}
```

done here is the counterpart of job_done in process_backlog, with the same meaning that the queue is empty. At this point, in the NAPI driver, the __netif_rx_complete function (defined in the same file) removes the device from the poll_list list, a task that process_backlog does directly. Finally, the NAPI driver re-enables interrupts for the device. As we anticipated at the beginning of the section, process_backlog runs with interrupts enabled.

Ingress Frame Processing

As mentioned in the previous section, netif_receive_skb is the helper function used by the poll virtual function to process ingress frames. It is illustrated in Figure 10-7.

Multiple protocols are allowed by both L2 and L3. Each device driver is associated with a specific hardware type (e.g., Ethernet), so it is easy for it to interpret the L2 header and extract the information that tells it which L3 protocol is being used, if any (see Chapter 13). When net_rx_action is invoked, the L3 protocol identifier has already been extracted from the L2 header and stored into skb->protocol by the device driver.

The three main tasks of netif_receive_skb are:

- Passing a copy of the frame to each protocol tap, if any are running
- Passing a copy of the frame to the L3 protocol handler associated with skb->protocol*
- Taking care of those features that need to be handled at this layer, notably bridging (which is described in Part IV)

If no protocol handler is associated with skb->protocol and none of the features handled in netif_receive_skb (such as bridging) consumes the frame, it is dropped because the kernel doesn't know how to process it.

Before delivering an input frame to these protocol handlers, netif_receive_skb must handle a few features that can change the destiny of the frame.

* See Chapter 13 for more details on protocol handlers.

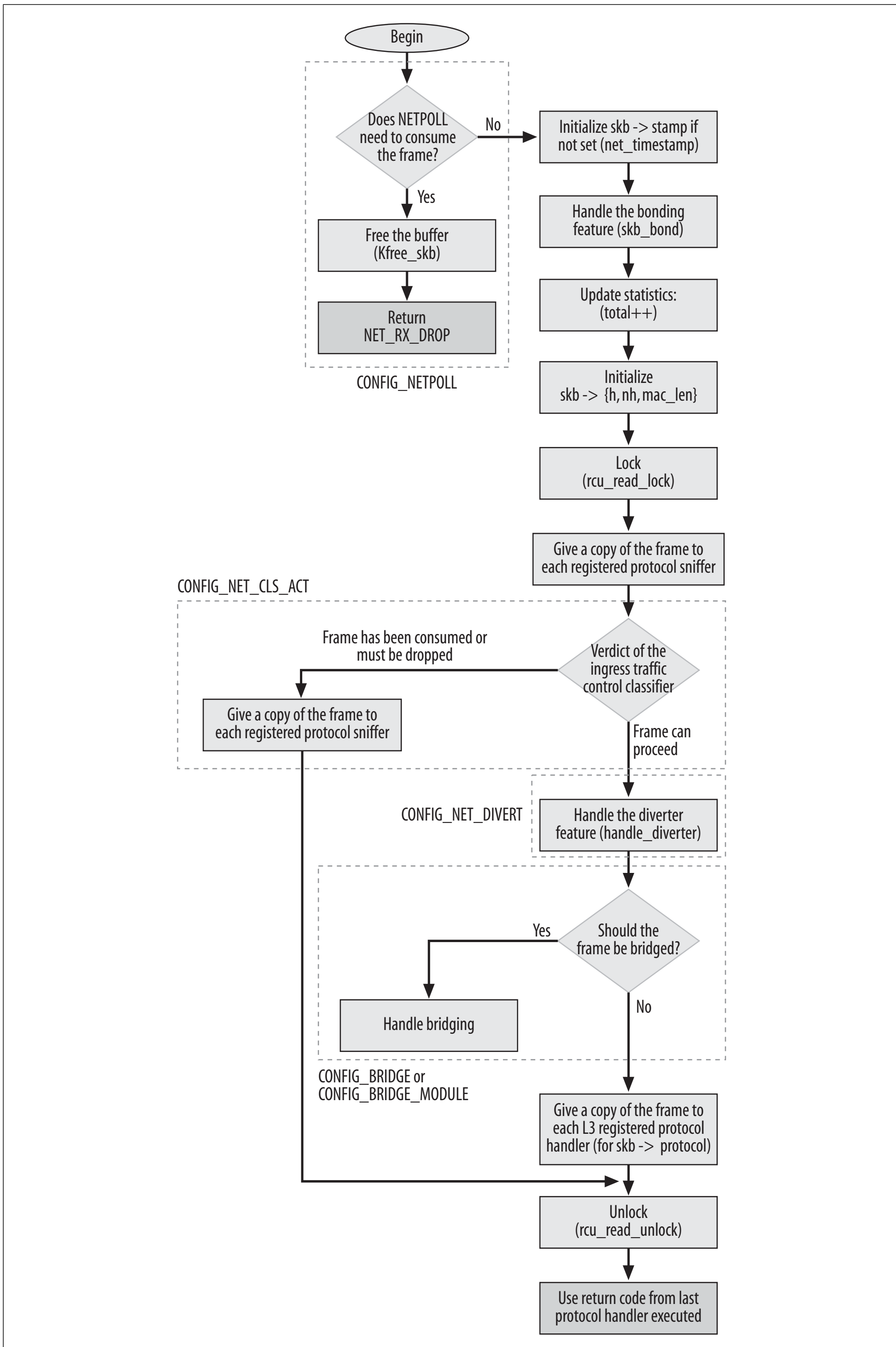


Figure 10-7. The netif_receive_skb function

Bonding allows a group of interfaces to be grouped together and be treated as a single interface. If the interface from which the frame was received belonged to one such group, the reference to the receiving interface in the `sk_buff` data structure must be changed to the device in the group with the role of master before `netif_receive_skb` delivers the packet to the L3 handler. This is the purpose of `skb_bond`.

```
skb_bond(skb);
```

The delivery of the frame to the sniffers and protocol handlers is covered in detail in Chapter 13.

Once all of the protocol sniffers have received their copy of the packet, and before the real protocol handler is given its copy, Diverter, ingress Traffic Control, and bridging features must be handled (see the next section).

When neither the bridging code nor the ingress Traffic Control code consumes the frame, the latter is passed to the L3 protocol handlers (usually there is only one handler per protocol, but multiple ones can be registered). In older kernel versions, this was the only processing needed. The more the kernel network stack was enhanced and the more features that were added (in this layer and in others), the more complex the path of a packet through the network stack became.

At this point, the reception part is complete and it will be up to the L3 protocol handlers to decide what to do with the packets:

- Deliver them to a recipient (application) running in the receiving workstation.
- Drop them (for instance, during a failed sanity check).
- Forward them.

The last choice is common for routers, but not for single-interface workstations. Parts V and VI cover L3 behavior in detail.

The kernel determines from the destination L3 address whether the packet is addressed to its local system. I will postpone a discussion of this process until Part VII; let's take it for granted for the moment that somehow the packet will be delivered to the above layers (i.e., TCP, UDP, ICMP, etc.) if it is addressed to the local system, and to `ip_forward` otherwise (see Figure 9-2 in Chapter 9).

This finishes our long discussion of how frame reception works. The next chapter describes how frames are transmitted. This second path includes both frames generated locally and received frames that need to be forwarded.

Handling special features

`netif_receive_skb` checks whether any Netpoll client would like to consume the frame.

Traffic Control has always been used to implement QoS on the egress path. However, with recent releases of the kernel, you can configure filters and actions on

ingress traffic, too. Based on such a configuration, ing_filter may decide that the input buffer is to be dropped or that it will be processed further somewhere else (i.e., the frame is consumed).

Diverter allows the kernel to change the L2 destination address of frames originally addressed to other hosts so that the frames can be diverted to the local host. There are many possible uses for this feature, as discussed at *http://diverter.sourceforge.net*. The kernel can be configured to determine the criteria used by Diverter to decide whether to divert a frame. Common criteria used for Diverter include:

- All IP packets (regardless of L4 protocol)
- All TCP packets
- TCP packets with specific port numbers
- All UDP packets
- UDP packets with specific port numbers

The call to handle_diverter decides whether to change the destination MAC address. In addition to the change to the destination MAC address, skb->pkt_type must be changed to PACKET_HOST.

Yet another L2 feature could influence the destiny of the frame: Bridging. Bridging, the L2 counterpart of L3 routing, is addressed in Part IV. Each net_device data structure has a pointer to a data structure of type net_bridge_port that is used to store the extra information needed to represent a bridge port. Its value is NULL when the interface has not enabled bridging. When a port is configured as a bridge port, the kernel looks only at L2 headers. The only L3 information the kernel uses in this situation is information pertaining to firewalling.

Since net_rx_action represents the boundary between device drivers and the L3 protocol handlers, it is right in this function that the Bridging feature must be handled. When the kernel has support for bridging, handle_bridge is initialized to a function that checks whether the frame is to be handed to the bridging code. When the frame is handed to the bridging code and the latter consumes it, handle_bridge returns 1. In all other cases, handle_bridge returns 0 and netif_receive_skb will continue processing the frame skb.

```
if (handle_bridge(skb, &pt_prev, &ret));
    goto out;
```