Path of a packet in the Linux kernel

Vincent Guffens

April 22, 2003

Contents

1 Introduction

This section is intended to give an overview of how an ip packet will traverse the Linux kernel. The Linux networking code is complex and giving an exhaustive description of the different sub-paths an exceptions that can occur is outside the scope of this text.

The different layers refer to the well known Open Systems Interconnection (OSI) reference model that can be found in annex A.

A schema representing a graphical representation of this document can be found in annex B. This document is based on Linux 2.4.17.

2 From the physical medium to the network queue - Layer1

Linux supports a wide number of interface types. Beside the typical Ethernet adapter, 10/100baseT or gigabit Ethernet, one can find some atm (in pre-alpha stage at the time of writing), isdn, hssi, fddi, wireless lan adaptors and others.

Off course, each interface has its on way of receiving a frame. To fix the idea, let's consider what happens in the case of an Ethernet adapter : The on-board memory is typically split into two regions used for receiving and sending frames. The 3c509, for instance, has a 4kB packet buffer split into 2kB Rx, 2kB Tx. A newer model, the 3c509B as 8kB on-board that can be split into 4/4 5/3 or 6/2 for Rx/Tx.[5].

The frame is stored into that memory region in a FIFO structure called rx-ring. Upon reception of a frame, the adapter will then issue an interrupt (irq) to inform the cpu of the event. An interrupt handler, registered during the *open* method of the device will then be run (See Chap. 9 and 14 of [7]). A new Linux network buffer (*sk_buff*)[1] is then allocated and the frame is copied into the new buffer. The frame can be transfered using programmed I/O (NE200, 3com509), shared memory (WD90x03, 3c503) or Direct Memory Access (DMA) and bus mastering (Lance, DEC21040). If using DMA, the sk_buff can be pre-allocated an mapped to the DMA region. This optimization will reduce the cpu usage but will however not populate the cpu cache.

An interface in Linux is represented by a structure called *net_device*[2]. This structure holds a lot of information ranging from physical data to some layer 3 related pointers. Among other, you can find the name of the interface, the I/O memory, irq, information on the tx queue and some pointers to functions used to manipulate the device such as the initialization function, functions to manipulate the hardware header, to transmit a frame...

The time spend to process the interrupt is critical and must be kept at a minimum. Until the irq is acknowledged by the cpu, the interruption mechanism is disable. Packets can accumulate in the rx ring (which sometimes, can contain only 2 packets) and get dropped. User land processes won't have access to the cpu anymore and the system will finally seems to hang. Once the frame has been transfered into the *sk_buff*, the only action taken will be to queue the packet in the network buffer queue and return from the interrupt. This queue is refer to as *sofnet* an is unique for all the interfaces for a single cpu machine. More precisely, *sofnet_data[NR_CPUS]*[3] is an array of NR_CPUS sofnet_data structure, that is, one per cpu which

[1] A Linux network buffer is a structure called *sk_buff* declared in *include/linux/skbuff.h*. This structure is at the heart of the Linux networking code, see later in the text.

[2] include/linux/netdevice.h

[3] /include/linux/netdevice.h

permit to avoid serialization and locking issues (At the expense of out of order packets[4]). The packets enter an leave this queue in a FIFO manner.

Once the packet is queue, we can safely return from the interrupt after having warned the kernel that he will have to dequeue this packet sometimes later. The mechanism used for that purpose is called "software interrupt" or bottom half.[3]

Although we really want to return as fast as possible from the interrupt, we also want to try to control the queue in case of congestion. As of 2.4.17, the congestion control is quite simple. We queue the buffers until the queue length reach *netdev_max_backlog* (set to 300) at which point we throttle the queue until the queue length comes back to zero. We, off course, drop all the incoming packets during that time. A number of more elaborate control congestion methods are already used by some drivers and some enhancements have been suggested in [9]. We will discuss these methods in detail later in the text.

2.1 The sk_buff network buffer by Alan Cox

The full version of this section can be found online at :

http://www.linux.org.uk/Documents/buffers.html

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 optimized 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 standardized and efficient manner.

At its most basic 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
    {
```

[4]The problem of out of order packets was however solved using irq affinity, see [9]

```
                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 capture quite closely the receive packet mechanism. If you consider append_frame to be called from an interrupt by a device driver receiving a packet, and process_frame as the code called to feed data into the protocols you can go and look in dev.c at netif_rx() and net_bh(). 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 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. After alloc_skb() obtains a buffer of len bytes, it 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 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, thus skb_push() is provided to allow you to move the start of the frame down through memory, providing enough space has been reserved to leave room for this.

After a buffer has been allocated all the room is at the end. A further call skb_reserve() 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);
```

Figure 1: After alloc_skb()

Figure 2: After skb_reserve()

Figure 3: An sk_buff containing data

Figure 4: After skb_put()

Figure 5: After skb_push()

```
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 know in advance (often wasting a few bytes to allow for the worst case) because linear buffers make many other things much faster. One useful trick you will see in the Ethernet drivers is to reserve a couple of bytes before reading a buffer from the card to main memory. This lands the IP header on a four byte boundary for faster access. Better still it lands it on a sixteen byte boundary which leaves it cache aligned.

2.2 Congestion collapse and NAPI

The interrupt mechanism used for frame reception leads to a phenomena called "congestion collapse". In [9], the reasons for this collapse are analyzed and a solution, referred to as NAPI (New API) is proposed. Congestion collapse occurs when, although a very high number of packets per second enter a Linux router, not a single packet is going out. Measurements carried in [9] have shown that this point was situated at around 60 Kpps for a Pentium II based pc with Linux 2.3.99. This rather undesirable behavior is due to *interrupt live lock* : for each packet entering the system, an interruption is issued and a fraction of time is lost. For a very high number of interruptions, the processor does not have any time left for producing any useful

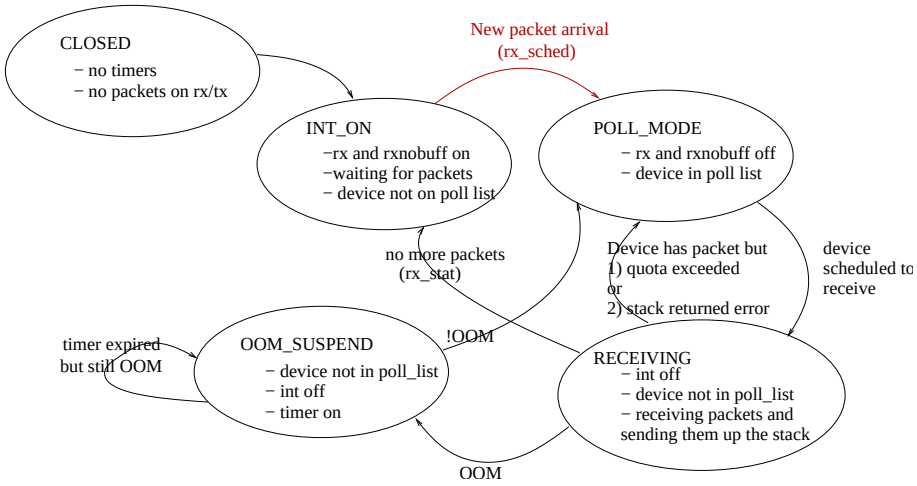

Figure 6: NAPI finite state machine (source : linux-net mailing list, Jamal)

work. The congestion collapse point is reached.

To alleviate this problem, a mechanism that prevent the network card from interrupting the processor must be found. This mechanism is shown on Fig. 6 which describes the NAPI strategy. Instead of having the card interrupt the processor for each packet it receives, the interface is registered on the poll list and its interruption is disabled. The interface is then scheduled to send a certain quota. If the interface has still some work to do after the quota is reached, it is put back in the poll list, otherwise, it return to its initial state with interruption on.

With this mechanism, congestion collapse is never reached. Instead, the number of outgoing packets reach a saturation point but does not decrease to zero anymore.

3 From the network queue to the protocol handler - Layer2

Once the software interrupt is scheduled for execution, the frame is dequeued and the protocol type can be used to send the frame to the right protocol handler. In fact the protocol type has already been identified by the drivers which knows what encapsulation type is used and therefore knows how to get the protocol information from the layer 2 header. For instance, the 3c509 driver use the function *eth_type_trans*[5] to set the value of $sk\_buff->protocol$. As the layer 2 has been further divided by the IEEE into a logical link control (LLC) and a Medium Access Control (MAC) layer, this func-

[5]net/ethernet/eth.c

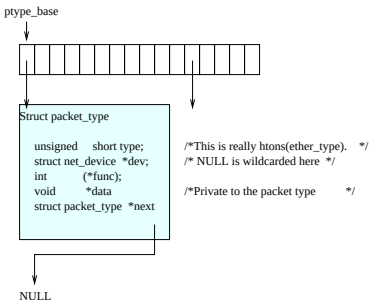

Figure 7: Protocols must be registered in a global structure

tion is a little bit more complex then one might expect. In effect the 802.2 LLC hold a "length" field after the destination and source address where the Ethernet II header has the "type" field. An 802.2 header is then used to carried protocol information[6]. (See for instance chap. 7 of [4] for more information) . Further complication arise from some old Novell implementation of the protocol which encapsulates IPX packets directly into an 802.3 header without 802.2 header.

The net result is that the layer 3 protocol that is carried into that layer 2 header can be unequivocally determined. Once this information is known, the right protocol handler must be determined. This is done by looking into the *ptype_base* array (see Fig. 7)that holds all the protocols that can be processed along with the corresponding entry function such as ip_rcv or arp_rcv. New protocols can be registered by calling the function *void dev_add_pack(struct packet_type *pt)*[7] that will link the *packet_type* structure to the global array *ptype_base[16]*.

4 Forwarding the packet - Layer3

In this section, we suppose that we are dealing with an IP packet that must be forwarded through the linux box. Therefore, the packet enter the layer 3 processing with the function *ip_rcv*. At this point, we find the first Netfilter hook. Netfilter is the packet filtering / packet mangling / NAT framework of the Linux 2.4 kernel series. Netfilter is a generalized framework of hooks in the network stack. Any kernel module can plug into one or more of these hooks an will receive each packet traversing this hook. The netfilter hooks are currently implemented for IPv4, IPv6 and DECnet. As explained in [6], there are five hooks in the Linux kernel, represented in Fig. 8. Three of

[6]In that case, the DSAP and SSAP field are set to 0xAA to indicate the presence of a SNAP header made of 3 bytes of organization code (assigned by IEEE) and finally the 2 bytes used for the protocol type

[7]net/core/dev.c

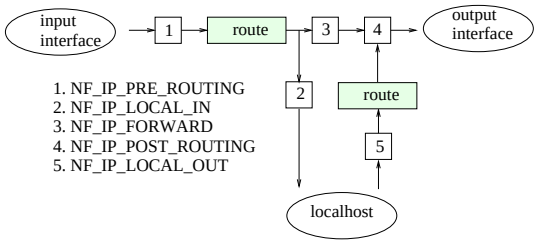

Figure 8: Netfilter hook along the packet path (source [6])

these hooks are also visible in Fig. 12.

Basically, these hooks can decide to drop the packet or to let it continue, sometimes after modification (mangling). Supposing that our packet survived the first hook, it is now time for it to be routed.

Routing involve a look up in a complex structure called Forwarding Information Base Table (FIB table). The goal of this lookup is to find a route entry corresponding to the destination ip address of the packet. A next hop is associated with each of these routes.

The next hop is the router we should forward the packet to. Because looking up in this complex structure is quite expensive, a route cache is also used to store the routes that are being used. Look up in the cache is done with a hash function that uses a combination of source and destination address, TOS and incoming device. Therefore, two packets having these fields in common will be routed toward the same next hop, rendering multipath routing impossible without modification to the caching system.

Traditionally, the FIB has been populated by way of IOCTL call. Today, netlink offer a reacher set of possibilities and is used by programs such has *ip* and *tc*. A library *libnetlink* is available to use the functions offered by netlink.

Now that a next hop has been found, the packet is ready to be forwarded. During the routing phase, the $skb->dst$ field was set. The next step is to call the *input* method associated with this destination. At this stage, the TTL field in the ip header is decremented and the mtu of the outgoing interface is check. If this mtu is smaller than the packet size, the ip packet has to be fragmented, otherwise, it can directly be transmitted to the outgoing interface. ICMP messages may also be generated at this stage (see [2] for details on requirement for IP routers)

If the information needed to successfully send the packet to the next router is not known, an arp packet might be sent to obtain the hardware address of the next interface. When this information is known, the mac rewrite can take place and the packet is ready to be sent toward the next hop.

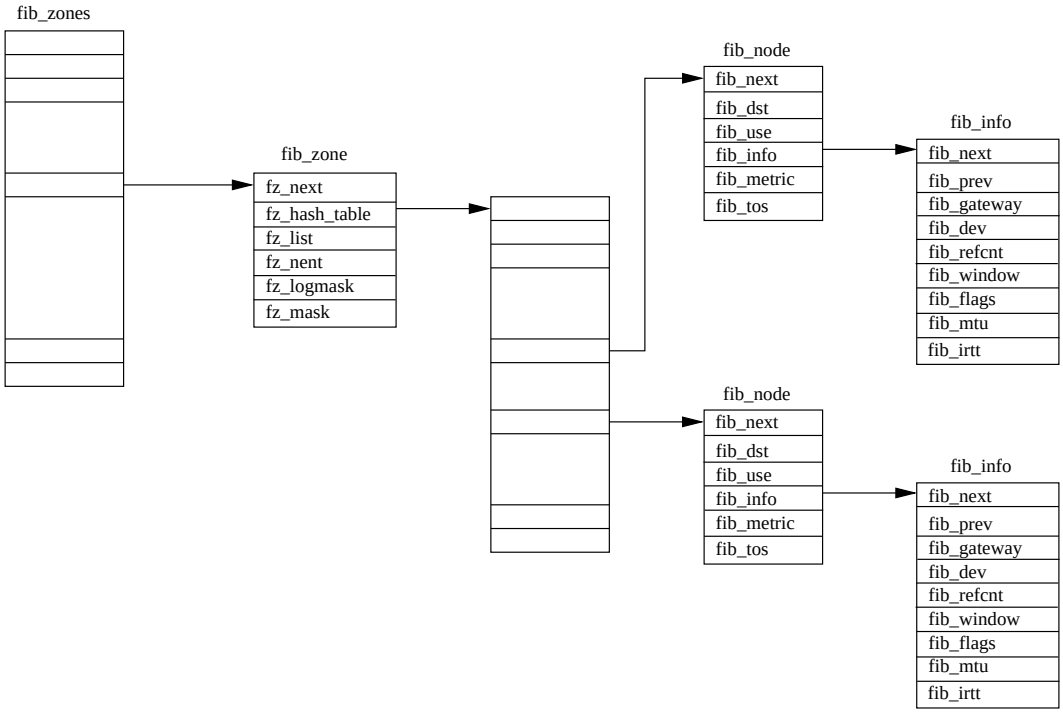

Figure 9: The forwarding information base (source [8]).

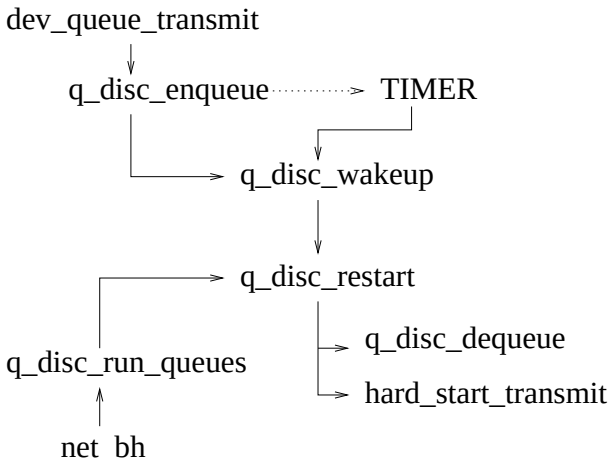

Figure 10: Functions called when queueing and sending packets (source [1])

5 Queueing the packet to the outgoing interface - Layer2

At this stage, the packet enters the outgoing queue where traffic control, bandwidth shaping and complex queueing can take place. The way the packet enter this queue is illustrated in Fig. 10.

The queueing policy in use may not allow for directly sending a packet. Therefore, the packet has to be left in the queue and a timer has to be configure to schedule the departure of this packet. Because of the very low timing resolution of Linux (1 tick = 10 [ms]) on intel based architecture, this timer may lead to restriction on configurable parameters. Let's say that a packet is left in a TBF queue. This packet may wait $1/HZ$ (in Linux 1 tick = $1/HZ$ seconds and is the resolution of the kernel, HZ=100 for intel architecture, 1000 for alpha) seconds. Because a maximum of B bytes can be send by this buckets in any time interval, the maximum achievable bandwidth is therefore $B * HZ$. Of course, the queue can be woken up asynchronously by others packets entering the system which has a positive effect on the effective resolution.

6 Appendix

A The OSI model

B The path of a packet figure

See fig. 12.

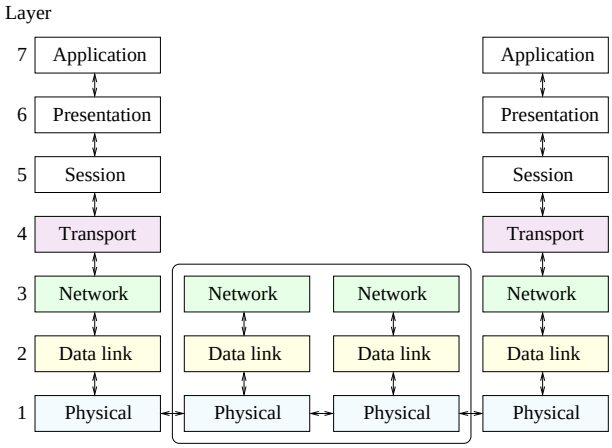

Figure 11: The Open System Interconnection reference model

References

[1] Werner Almesberger. Linux traffic control - implementation overview, November 1998.

[2] F. Baker. *RFC1812 - Requirements for IP Version 4 Routers*, June 1995.

[3] D. P. Bovet and M. Cesati. *Linux Kernel*. O'Reilly, 2001.

[4] K. Downes, M. Ford, H. K. Lew, S. Spanier, and T. Stevenson. *Internetworking technologies handbook*. Macmillan Technical Publishing, 1998.

[5] P. Gortmaker. Linux ethernet-howto, 2000.

[6] LaForge. Irc talk on netfilter - http://www.gnumonks.org/papers/netfilter-lk2000, 2000.

[7] A. Rubini and J. Corbet. *Linux Device Drivers*. O'Reilly, 2001.

[8] David A. Rusling. *The Linux Kernel*. 1996.

[9] J. H. Salim, R. Olsson, and A. Kuznetsov. Beyond sofnet, proceedings of the 5th annual linux showcase & conference, oakland, california, November 2001.

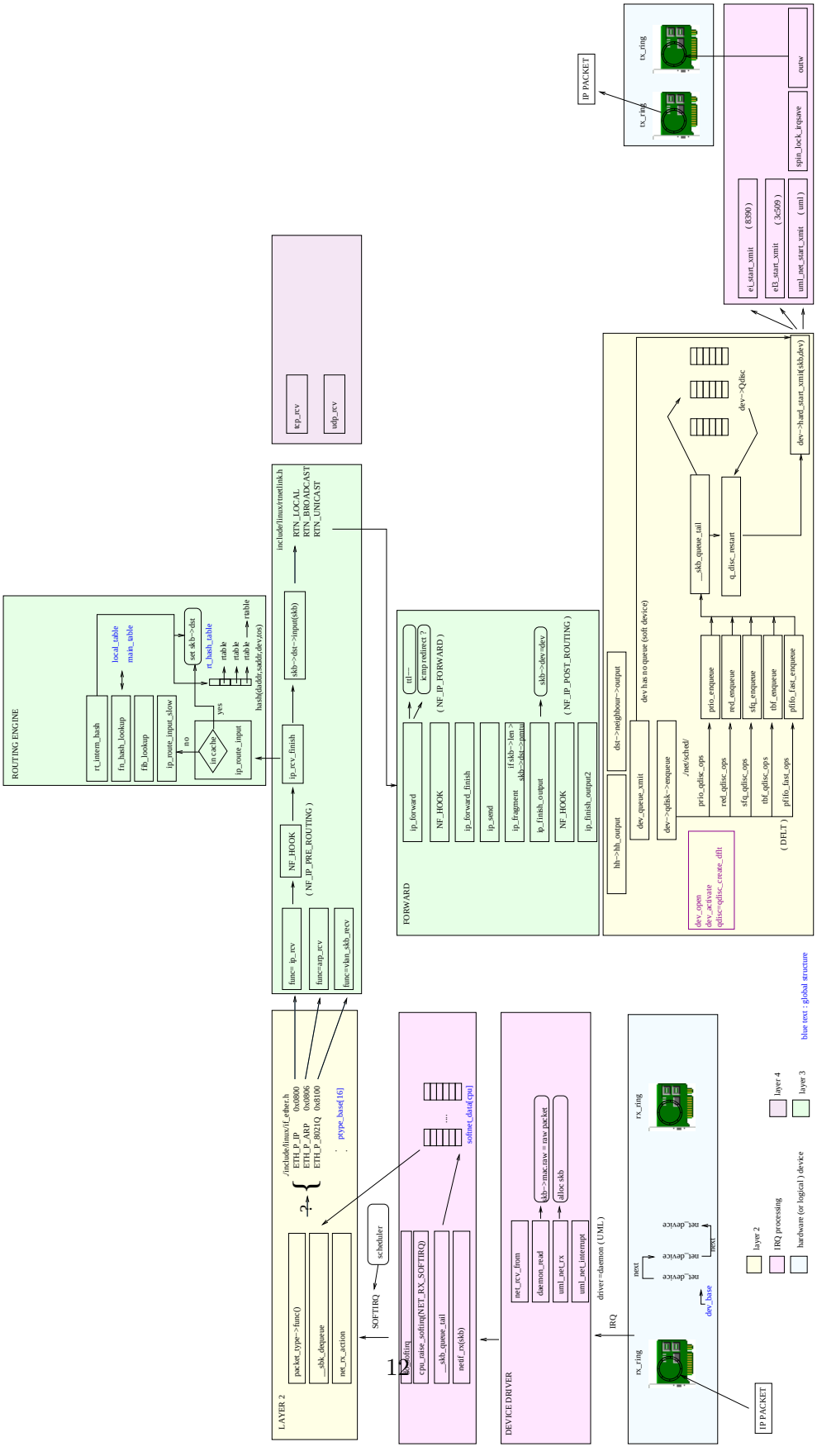

Figure 12: The path of a packet