Received Packet Processing

In this section we consider the processing of a received packet as it moves from the device driver to the IP layer. The *device driver* relies principally upon two kernel functions

- *dev_alloc_skb()* — Allocates an *sk_buff* of the required size prior to transferring the packet to kernel memory. Two hardware strategies are commonly used. If packets are received directly into system memory owned by the kernel, the *sk_buff* must be allocated prior to initiating the receive operation. If packets are first received into NIC buffers and then transferred via DMA to system memory, an *sk_buff* of the exact size needed may be allocated after the packet has been received but before the DMA transfer is initiated.
- *netif_rx()* — Used to pass the *sk_buff* to the generic device layer when a receive operation completes.

The example below is taken from drivers/net/3c59x.c

```
skb = dev_alloc_skb(pkt_len + 5);
skb->protocol = eth_type_trans(skb, dev);
 :
netif_rx(skb);
```

Allocation and initialization of the *sk_buff*

dev_alloc_skb() is defined in include/linux/skbuff.h. It merely calls *__dev_alloc_skb* with the GFP_ATOMIC flag set. This flag forces *kmalloc()* to return an error code rather than sleeping if no memory is available. It is necessary because sleeping in an interrupt context is a fatal error.

```
static inline struct sk_buff *dev_alloc_skb(unsigned
                                             int length)
{
        return __dev_alloc_skb(length, GFP_ATOMIC);
}
```

__dev_alloc_skb() allocates memory for the sk_buff of specified size. A few additional bytes are always allocated for alignment optimization purposes.

```
static inline struct sk_buff *__dev_alloc_skb(unsigned
                                 int length, int gfp_mask)
{
        struct sk_buff *skb;

        skb = alloc_skb(length+16, gfp_mask);
```

The alloc_skb() function allocates the sk_buff. Recall that an sk_buff consists of a fixed size header of type struct sk_ buff which is allocated from a cache of such objects and a variable size data buffer allocated from one of the general caches and actually holds the packet's headers and user data.

```
struct sk_buff *alloc_skb(unsigned int size,int gfp_mask)
{
        struct sk_buff *skb;
        u8 *data;

        if (in_interrupt() && (gfp_mask & __GFP_WAIT)) {
             static int count = 0;
             if (++count < 5) {
                printk(KERN_ERR "alloc_skb called nonatomically "
                       "from interrupt %p\n", NET_CALLER(size));
                            BUG();
             }
             gfp_mask &= ~__GFP_WAIT;
        }

        /* Get the HEAD */
        skb = skb_head_from_pool();
        if (skb == NULL) {
             skb = kmem_cache_alloc(skbuff_head_cache, gfp_mask &
                       ~__GFP_DMA);
             if (skb == NULL)
                  goto nohead;
        }

        /* Get the DATA. Size must match skb_add_mtu(). */
        size = SKB_DATA_ALIGN(size);
        data = kmalloc(size + sizeof(struct skb_shared_info),
                                     gfp_mask);
        if (data == NULL)
             goto nodata;
```

When head and data have been successfully allocated, the head is initialized.

```

            /* XXX: does not include slab overhead */
        skb->truesize = size + sizeof(struct sk_buff);

            /* Load the data pointers. */
        skb->head = data;
        skb->data = data;
        skb->tail = data;
        skb->end = data + size;

            /* Set up other state */
        skb->len = 0;
        skb->cloned = 0;
        skb->data_len = 0;

        atomic_set(&skb->users, 1);
        atomic_set(&(skb_shinfo(skb)->dataref), 1);
        skb_shinfo(skb)->nr_frags = 0;
        skb_shinfo(skb)->frag_list = NULL;
        return skb;

    nodata:
        skb_head_to_pool(skb);
    nohead:
        return NULL;
    }


```

On return to __dev_alloc_skb() space is reserved for the MAC header.

```
        if (skb)
             skb_reserve(skb,16);
        return skb;
    }
```

skb_reserve moves the data and tail pointers to point to first byte after the 16 bytes of headroom.

```
    static inline void skb_reserve(struct sk_buff *skb,
                                  unsigned int len)
    {
        skb->data+=len;
        skb->tail+=len;
    }
```

The skb_put() function can be used to update the len and tail values after data has been placed in the sk_buff(). The actual filling of the buffer is most commonly performed by a DMA transfer.

```
static inline unsigned char *skb_put(struct sk_buff
                       *skb, unsigned int len)
{
    unsigned char *tmp=skb->tail;
    SKB_LINEAR_ASSERT(skb);
    skb->tail += len;
    skb->len += len;
    if(skb->tail>skb->end) {
         skb_over_panic(skb,len,current_text_addr());
    }
    return tmp;
}
```

Non–linear sk_buffs are those consisting of unmapped page buffers and additional chained struct sk_buffs. A non–zero value of data_len is an indicator of non–linearity. For obvious reasons the simple skb_put() function neither supports nor tolerates non–linearity. SKB_LINEAR_ASSERT checks value of data_len through function skb_is_nonlinear. A non–zero value results in an error message to be logged by BUG.

```
#define SKB_LINEAR_ASSERT(skb)
    do { if (skb_is_nonlinear(skb)) BUG(); } while (0)
```

skb_is_nonlinear is defined as below.

```
static inline int skb_is_nonlinear(const struct
                            sk_buff *skb)
{
    return skb->data_len;
}
```

Queuing the packet with netif_rx()

The netif_rx() function is defined in net/core/dev.c. It typically runs in the context of the hardware interrupt that signalled the completion of the DMA transfer. Its function is to queue the sk_ buff for processing by network layer. The buffer may, however, be dropped during processing for congestion control. After queuing the packet, netif_rx() raises the NET_RX_SOFTIRQ. The bulk of the processing of an input packet is done in the context of this softirq by the net_rx_action() function. The netif_rx() function returns one of the following values which are defined in include/linux/netdevice.h.

```
/* Backlog congestion levels */
#define NET_RX_SUCCESS  0 /* keep 'em coming, baby */
#define NET_RX_DROP     1 /* packet dropped */
#define NET_RX_CN_LOW   2 /* storm alert, just in case */
#define NET_RX_CN_MOD   3 /* Storm on its way! */
#define NET_RX_CN_HIGH  4 /* The storm is here */
#define NET_RX_BAD      5  /* packet dropped due to
                             kernel error */
```

Incoming packets are placed on per-cpu queues so that no locking is needed. The softnet_data array, defined in include/linux/netdevice.h., consists of a struct softnet_data for each CPU.

```
struct softnet_data
{
    int                     throttle;
    int                     cng_level;
    int                     avg_blog;
    struct sk_buff_head     input_pkt_queue;
    struct net_device       *output_queue;
    struct sk_buff          *completion_queue;
} __attribute__((__aligned__(SMP_CACHE_BYTES)));

extern struct softnet_data softnet_data[NR_CPUS];
```

```
struct sk_buff_head {
    /* These two members must be first. */
    struct sk_buff   * next;
    struct sk_buff   * prev;

    __u32            qlen;
    spinlock_t       lock;
};
```

These are the congestion management parameters.

```
int netdev_max_backlog = 300;
/* These numbers are selected based on intuition and some
 * experimentatiom, if you have more scientific way
 * please go ahead and fix things.
 */
int no_cong_thresh = 10;
int no_cong = 20;
int lo_cong = 100;
int mod_cong = 290;

```

```
int netif_rx(struct sk_buff *skb)
{
        int this_cpu = smp_processor_id();
        struct softnet_data *queue;
        unsigned long flags;
```

If the device driver has not already time stamped the packet, it is done here.

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

The local variable queue is set to point to the struct sofnet_data for this cpu.

```
        /*   The code is rearranged so that the path is
             the most short when CPU is congested, but is
             still operating.
        */
        queue = &softnet_data[this_cpu];

```

Interrupts are disabled on this CPU while the packet is queued.

```
        local_irq_save(flags);

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

The length of the input packet queue is compared against its maximum backlog. If the queue is full, the sk_buff is discarded. The value of netdev_max_backlog is declared to be 300 packets in net/core/dev.c.

```
    if (queue->input_pkt_queue.qlen <= netdev_max_backlog) {
```

The following compound if first tests to see if the input queue is not empty. If the queue is not empty, then the throttle flag is tested to see if the packet should be dropped. The throttle flag indicates the presence (1) or absence (0) of congestion. If congestion is present, the sk_buff is discarded.

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

If the throttle flag is not set, the sk_buff is added to the input packet queue. Since the if statement above found qlen > 0, the queue is guaranteed to be non–empty here.

```
enqueue:
                dev_hold(skb->dev);
                __skb_queue_tail(&queue->input_pkt_queue,
                          skb);
```

The cpu_raise_softirq() function sets a flag to indicate that the NET_RX_SOFTIRQ software interrupt is now pending. Most of the actual work of handling the packet will take place in the context of the softirq.

```
            /*  Runs from irqs or BH's, no need to wake BH */
                cpu_raise_softirq(this_cpu, NET_RX_SOFTIRQ);
                local_irq_restore(flags);
```

After raising the softirq, netif_rx() returns the congestion level from softnet_data structure. The get_sample_stats() function in net/core/dev.c sets the congestion level.

```
#ifndef OFFLINE_SAMPLE
                get_sample_stats(this_cpu);
#endif
                return softnet_data[this_cpu].cng_level;
            }
```

If we reach this point in netif_rx(), the input packet queue is empty. The throttle flag, if set, is cleared.

```
1248            if (queue->throttle) {
1249                 queue->throttle = 0;
1250 #ifdef CONFIG_NET_HW_FLOWCONTROL
1251                 if (atomic_dec_and_test
                               (&netdev_dropping))
1252                      netdev_wakeup();
1253 #endif
1254            }
```

Here we jump back to the code that queue the sk_buff and raises the soft irq.

```
1255            goto enqueue;
1256       }
```

The if block that began at line 1231 ends here. If control reaches this point the input packet queue is full of sk_buffs. The throttle flag is set to indicate congestion. Note that once the queue becomes throttled it must drain completely to become ‘‘unthrottled’’.

```
1258       if (queue->throttle == 0) {
1259            queue->throttle = 1;
1260            netdev_rx_stat[this_cpu].throttled++;
1261 #ifdef CONFIG_NET_HW_FLOWCONTROL
1262            atomic_inc(&netdev_dropping);
1263 #endif
1264       }
```

The sk_buff is discarded here.

```
1266 drop:
1267       netdev_rx_stat[this_cpu].dropped++;
1268       local_irq_restore(flags);
1269
1270       kfree_skb(skb);
1271       return NET_RX_DROP;
1272 }
```

This is the end of netif_rx. We now turn our attention to the softirq.

Softirqs

In early versions of Linux processing of received packets took place in the context of what was called a bottom half. The softirq mechanism, which was designed to replace the bottom half was introduced in kernel 2.4. The primary advantage of the softirq mechanism is that multiple soft irqs may run concurrently on multiple processes. Bottom halves were permitted to run only on one CPU at a time.

Further handling of our packet is done in the network receive softirq (NET_RX_SOFTIRQ) which is called from kernel/softirq.c:do_softirq().

Recall that the function netdev_init() which runs at boot time registered two softirq handlers.

```
2865      open_softirq(NET_TX_SOFTIRQ, net_tx_action, NULL);
2866      open_softirq(NET_RX_SOFTIRQ, net_rx_action, NULL);
```

Therefore when netif_rx() executed the line of code shown below it indirectly causes the function net_rx_action() to be executed in a softirq context.

```
1240                cpu_raise_softirq(this_cpu, NET_RX_SOFTIRQ);
```

The do_softirq() function is defined in kernel/softirq.c. It invokes the appropriate action handler for each softirq raised.

```
asmlinkage void do_softirq()
{
    int cpu = smp_processor_id();
    __u32 pending;
    long flags;
    __u32 mask;

    if (in_interrupt())
         return;

    local_irq_save(flags);
```

The variable pending is a bit mask that indicates which of the possible 32 soft irqs are presently pending. It is set to irq_stat[cpu].__softirq_pending and mask is set to complement of this value.

```
    pending = softirq_pending(cpu);

    if (pending) {
         struct softirq_action *h;

         mask = ~pending;
         local_bh_disable();
```

local_bh_disable is defined in include/asm−i386/softirq.h. It is replaced by a call to cpu_bh_disable. local_bh_count is a macro which translates to irq_stat[cpu].__local_bh_count.

```
#define cpu_bh_disable(cpu) \
    do { local_bh_count(cpu)++; barrier(); } while (0)

#define local_bh_disable()
                   cpu_bh_disable(smp_processor_id())
```

Now in do_softirq, clear irq_stat[cpu].__softirq_pending.

```
restart:
  /* Reset the pending bitmask before enabling irqs */
          softirq_pending(cpu) = 0;

          local_irq_enable();
```

"h" is set to point to first element in softirq_vec array.

```
          h = softirq_vec;

          do {
```

Softirqs are checked in order of their priority (HI_SOFTIRQ, NET_TX_SOFTIRQ ...) and the respective function handler is called. In the case of NET_RX_SOFTIRQ, it is net_rx_action.

```
               if (pending & 1)
                    h->action(h);
```

"h" is now set to point to next element in softirq_vec array.

```
               h++;
               pending >>= 1;
          } while (pending);

          local_irq_disable();
```

If new softirqs (other than those handled above) have been raised, they are handled as well. Recall that mask was originally set to the complement of pending. The masking here is presumably in place to prevent livelock type conditions.

```
          pending = softirq_pending(cpu);
          if (pending & mask) {
               mask &= ~pending;
               goto restart;
          }
```

__local_bh_enable is defined in include/asm–i386/softirq.h.

```
#define __cpu_bh_enable(cpu) \
    do { barrier(); local_bh_count(cpu)--; } while (0)
#define __local_bh_enable()
    __cpu_bh_enable(smp_processor_id())
```

```
            __local_bh_enable();

            if (pending)
                wakeup_softirqd(cpu);
        }

        local_irq_restore(flags);
}
```

Received packet handling in the softirq.

The net_rx_action() function resides in net/core/dev.c and was previously shown to have been installed as the handler for the NET_RX_SOFTIRQ. As might be expected, its mission is to consume packets from the queue that netif_rx() produces to and then to pass them on to the proper handler.

```
static void net_rx_action(struct softirq_action *h)
{
    int this_cpu = smp_processor_id();
```

A unique structure of type struct softnet_data is associated with each CPU for the purpose of managing input and output queues at the interface between the protocols and the device driver. Here queue is initalized to point to softnet_data structure for this CPU.

```
    struct softnet_data *queue =
                   &softnet_data[this_cpu];
    unsigned long start_time = jiffies;
    int bugdet = netdev_max_backlog;

    br_read_lock(BR_NETPROTO_LOCK);

```

This loop is executed until either the input queue has been emptied, at least one jiffy of CPU time has been consumed, or the value of bugdet becomes negative.

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

Attempt to dequeue an sk_buff from this CPU's input packet queue. Because the queue is local to the CPU local disabling of interrupts provides safe serialization.

```
         local_irq_disable();
         skb = __skb_dequeue(&queue->input_pkt_queue);
         local_irq_enable();
```

Return if the input packet queue is empty.

```
         if (skb == NULL)
              break;
```

The net_device pointer (which was set by the device driver) is potentially adjusted here.

```
         skb_bond(skb);
```

The skb_bond() function is defined in net/core/dev.c. It assigns the sk_buff to the master device for present device if such exists. We really don't understand device groups and master devices!

```
/*  Reparent skb to master device. This function is
    called only from net_rx_action under
    BR_NETPROTO_LOCK.
    It is misuse of BR_NETPROTO_LOCK, but it is OK for
    now.
 */
static __inline__ void skb_bond(struct sk_buff *skb)
{
    struct net_device *dev = skb->dev;

    if (dev->master) {
         dev_hold(dev->master);
         skb->dev = dev->master;
         dev_put(dev);
    }
}
```

Back in net_rx_action the net_device pointer is copied to a local variable.

```
        rx_dev = skb->dev;

```

CONFIG_NET_FASTROUTE is an option to allow direct NIC–to–NIC data transfer on a local network. We do will ignore it for now.

```
#ifdef CONFIG_NET_FASTROUTE
        if (skb->pkt_type == PACKET_FASTROUTE)  {
             netdev_rx_stat[this_cpu].
                  fastroute_deferred_out++;
             dev_queue_xmit(skb);
             dev_put(rx_dev);
             continue;
        }
#endif
```

Link level demultiplexing

Here skb−>data points to the start of the data area of the buffer. Therefore h.raw and nh.raw are both being set to point to the MAC header.

```
        skb->h.raw = skb->nh.raw = skb->data;
```

Here is where link layer demultiplexing takes place. For DIX framing the key to demultiplexing is the standard packet type field that is carried in the MAC header.

```
#define ETH_P_LOOP      0x0060   /* Ethernet Loopback packet     */
#define ETH_P_PUP       0x0200   /* Xerox PUP packet             */
#define ETH_P_PUPAT     0x0201   /* Xerox PUP Addr Trans packet  */
#define ETH_P_IP        0x0800   /* Internet Protocol packet     */
#define ETH_P_X25       0x0805   /* CCITT X.25                   */
#define ETH_P_ARP       0x0806   /* Address Resolution packet    */
#define ETH_P_BPQ       0x08FF   /* G8BPQ AX.25 Ethernet Packet */
#define ETH_P_IEEEPUP   0x0a00   /* Xerox IEEE802.3 PUP packet */
#define ETH_P_IEEEPUPAT 0x0a01   /* Xerox IEEE802.3 PUP Addr Trans pkt*/
```

For 802.3 life is a bit more complicated:

```
#define ETH_P_802_3     0x0001   /* Dummy type for 802.3 frames  */
#define ETH_P_AX25      0x0002   /* Dummy protocol id for AX.25  */
#define ETH_P_ALL       0x0003   /* Every packet (be careful!!!) */
#define ETH_P_802_2     0x0004   /* 802.2 frames                 */
#define ETH_P_SNAP      0x0005   /* Internal only                */
#define ETH_P_DDCMP     0x0006   /* DEC DDCMP: Internal only     */
```

For all types this routine depends upon the device driver to have extracted the appropriate type from the MAC header and stored in in skb->protocol in host byte order. A convenience function is provided to the device driver as shown in the following extract from 3c59x.c

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

```
/*
* Determine packet's protocol ID. The rule here is that we
* assume 802.3 if type field is short enough to be a length.
* This is normal and works for any 'now in use' protocol.
*/

unsigned short eth_type_trans(struct sk_buff *skb,
    struct net_device *dev)
{
    struct ethhdr *eth;
    unsigned char *rawp;

```

The call to skb_pull() advances skb->data so that it points to the network layer header (or the IEEE 802.2 LLC header for 802.2/3 framing, and decrements skb->len by the length of the MAC header (hard_header_len);

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

```

If the low order bit of the high order byte of the MAC address is 1, then this packet is a broadcast or a multicast.

```
    if (*eth->h_dest&1)
    {
         if(memcmp(eth->h_dest,dev->broadcast, ETH_ALEN)==0)
              skb->pkt_type=PACKET_BROADCAST;
         else
              skb->pkt_type=PACKET_MULTICAST;
    }

/*
*      This ALLMULTI check should be redundant by 1.4
*      so don't forget to remove it.
*
*      Seems, you forgot to remove it. All silly devices
*      seems to set IFF_PROMISC.
*/

    else if(1 /*dev->flags&IFF_PROMISC*/)
    {
         if(memcmp(eth->h_dest,dev->dev_addr, ETH_ALEN))
              skb->pkt_type = PACKET_OTHERHOST;
    }

```

The two byte field immediately following the destination MAC address is the packet type for DIX framing but it is the packet length for IEEE 802.2/3 framing. For IP, ARP, RARP, and IPX the packet type is at least 0x800 which is 2048 and thus larger than the maximum frame size. It does look like something really ugly could ensue here if jumbo frames were used in conjunction with 802.2/3 framing.

```
    if (ntohs(eth->h_proto) >= 1536)
         return eth->h_proto;

    rawp = skb->data;

/*
*This is a hack to spot IPX packets. Older Novell breaks
*the proto and runs IPX over 802.3 without an 802.2 LLC
*layer. We look for FFFF which isn't a used 802.2 SSAP/DSAP.
*This won't work for fault tol netware but does for the rest.
*/
    if (*(unsigned short *)rawp == 0xFFFF)
         return htons(ETH_P_802_3);

```

For "real" 802.2/3 framing, the length field is followed by the 802.2 LLC header containing the DSAP,SSAP, and cntl fields which are normally set to 0xaa, 0xaa, 0x03. This is followed by the 802.2 SNAP header which contains a 3 byte originator code and finally the 2 byte type field. This module just returns the code for 802_2 in that case and leaves it to the 802.2 module to eventually perform the demultiplexing.

```
/*
*      Real 802.2 LLC
*/
    return htons(ETH_P_802_2);
}
```

Recall that protocol packet handlers register themselves by filling in the packet_type structure and passing it to dev_add_pack() where the structure is placed on an appropriate chain.

```
struct packet_type
{
    unsigned short  type;   /* really htons(ether_type).*/
    struct net_device *dev; /* NULL is wildcarded here  */
    int    (*func) (struct sk_buff *, struct net_device *,
                    struct packet_type *);
    void   *data;       /* Private to the packet type */
    struct packet_type *next;
};
```

This is the block in net_rx_action in which the packet processing occurs. Protocols which wish to receive all incoming packets are linked into a list pointed to by ptype_all. These protocols register the have type ETH_P_ALL and are processed before considering the protocols that consume only a specific packet type.

```
        {
                struct packet_type *ptype, *pt_prev;
                unsigned short type =  skb->protocol;
```

pt_prev is initialized to NULL. We try to match sk buff against each protocol, registered with ptype_all.

```
                pt_prev = NULL;
                for (ptype = ptype_all; ptype; ptype =
                     ptype->next) {
```

Even though every packet handler in this chain says it wants to see all packets, it can also say that it wants to limit the packets to those received on a specific device. If ptype−>dev is null, then any device is acceptable. A value of 0 in the data field of the packet_type structure indicates that this is an old protocol that does not understand shared sk_buffs. We don't really grasp what the oddball use of pt_prev is all about, but possibly it is trying to deal with the necessity of cloning an skb that is to be consumed by an old style protocol. It should be necessary to clone if and only if there is more than one protocol interested.

```
                     if (!ptype->dev || ptype->dev==skb->dev){
                          if (pt_prev) {
                               if  (!pt_prev->data) {
                                    deliver_to_old_ones
                                         (pt_prev, skb,0);
                               } else {
                                    atomic_inc(&skb->users);
                                    pt_prev->func(skb,
                                         skb->dev,
                                         pt_prev);
                               }
                          }
```

pt_prev is set to matched protocol. The protocol specific function is called when next match is found.

```
                          pt_prev = ptype;
                     }
                }
```

We are also not interested in diverters at the moment.

```
#ifdef CONFIG_NET_DIVERT
               if (skb->dev->divert &&
                         skb->dev->divert->divert)
                    handle_diverter(skb);
#endif /* CONFIG_NET_DIVERT */
```

CONFIG_BRIDGE is an option to enable ethernet bridging that we are also not considering.

```
#if defined(CONFIG_BRIDGE) ||
          defined(CONFIG_BRIDGE_MODULE)
               if (skb->dev->br_port != NULL &&
                         br_handle_frame_hook != NULL){
                    handle_bridge(skb, pt_prev);
                    dev_put(rx_dev);
                    continue;
               }
#endif
```

This is the point at which protocols wanting only specific packet types are processed. Recall that the packet_type structures of these protocols are placed in a hash table with 16 hash lists. The low order 4 bits of the protocol type is the hash key.   The loop processes the hash list corresponding to the protocol type of the current packet.

```
               for (ptype=ptype_base[ntohs(type)&15];
                         ptype;ptype=ptype->next) {
```

Test to see if the type of the packet matches the type registered in the packet_type structure. If so, and the protocol also registered a specific struct netdevice then it is necessary to see if the input device matches as well.

```
                    if (ptype->type == type &&
                              (!ptype->dev ||
                              ptype->dev == skb->dev))
                         {
                         if (pt_prev) {
                              if (!pt_prev->data)
                                   deliver_to_old_ones
                                        (pt_prev,
                                         skb, 0);
                              else {
                                   atomic_inc
                                        (&skb->users);
```

Depending on the protocol type, the appropriate handler function is called. It is ip_rcv for ETH_P_IP and arp_rcv for ETH_P_ARP.

```
                                        pt_prev->func(skb,
                                             skb->dev,
                                             pt_prev);
                              }
                         }
                         pt_prev = ptype;
                    }
               }
```

On exit from the loop invoke the function handler for the last matched protocol.

```
               if (pt_prev) {
                    if (!pt_prev->data)
                         deliver_to_old_ones(pt_prev,
                                             skb, 1);
                    else
                         pt_prev->func(skb, skb->dev,
                              pt_prev);
```

If no protocol was matched , pt_prev is NULL and the sk_buff is discarded.

```
                } else
                    kfree_skb(skb);
            }

```

The dev_put() function decrements the value of rx_dev->refcount. This was incremented back in netif_rx() and preserved across the scheduling of the softirq. But what if the skb_bond() call changed it ... All is well skb_bond also released and reallocated.

```
            dev_put(rx_dev);
```

If more than one jiffy has elapsed while consuming sk_buffs or netdev_max_backlog buffers have been consumed exit the loop. Otherwise, continue dequeueing sk_buffs.

```
            if (bugdet-- < 0 || jiffies - start_time > 1)
                goto softnet_break;
```

This option enables NIC (Network Interface Card) hardware throttling during periods of extremal congestion. At the moment only a couple of device drivers support it.

```
#ifdef CONFIG_NET_HW_FLOWCONTROL
            if (queue->throttle && queue->input_pkt_queue.qlen
                    < no_cong_thresh ) {
                if (atomic_dec_and_test
                        (&netdev_dropping)) {
                    queue->throttle = 0;
                    netdev_wakeup();
                    goto softnet_break;
                }
            }
#endif

        }
        br_read_unlock(BR_NETPROTO_LOCK);
```

Reaching this point means that the `for` loop was exited via the `break` at line 1437 and the input packet queue has been completely emptied. The `throttle` flag is cleared, if set.

```
1532     local_irq_disable();
1533     if (queue->throttle) {
1534          queue->throttle = 0;
1535 #ifdef CONFIG_NET_HW_FLOWCONTROL
1536          if (atomic_dec_and_test(&netdev_dropping))
1537               netdev_wakeup();
1538 #endif
1539     }
1540     local_irq_enable();
1541
1542     NET_PROFILE_LEAVE(softnet_process);
1543     return;
```

Reaching this point means that the loop was exited via the `goto` at line 1517 which is triggered by the jiffy count. The interrupt is raised again since there are `sk_buffs` remaining to be processed.

```
1545 softnet_break:
1546     br_read_unlock(BR_NETPROTO_LOCK);
1547
1548     local_irq_disable();
1549     netdev_rx_stat[this_cpu].time_squeeze++;
1550     /*   This already runs in BH context, no need to
          wake up BH's */
1551     cpu_raise_softirq(this_cpu, NET_RX_SOFTIRQ);
1552     local_irq_enable();
```

NET_PROFILE_LEAVE has effect only when network code profiler is configured.

```
1554     NET_PROFILE_LEAVE(softnet_process);
1555     return;
1556 }
```

The deliver_to_old_ones() function is defined in net/core/dev.c. It invokes the handler function for old protocols that do not understand shared sk_buffs.

```
/*   Deliver skb to an old protocol, which is not
     threaded well or which do not understand shared
     skbs.
 */
static int deliver_to_old_ones(struct packet_type *pt,
                   struct sk_buff *skb, int last)
{
     static spinlock_t net_bh_lock = SPIN_LOCK_UNLOCKED;
     int ret = NET_RX_DROP;

```

The value of last is one if and only if this is the last protocol to which the packet must be delivered. In this case it is not necessary to create a clone of the sk_buff.

```
     if (!last) {
          skb = skb_clone(skb, GFP_ATOMIC);
          if (skb == NULL)
               return ret;
     }
```

An sk_buff which contains data in unmapped page sections is made linear by skb_linearize. A linear sb_buff is one which consists of a fixed length header of type struct sk_buff and a single kmalloc'ed data buffer.

```
     if (skb_is_nonlinear(skb) &&
               skb_linearize(skb, GFP_ATOMIC) != 0) {
          kfree_skb(skb);
          return ret;
     }

```

Here several hacks are inserted to provide the expected environment to the old protocols. We do not consider these in detail.

```
  /* The assumption (correct one) is that old
     protocols did not depened on BHs different of
     NET_BH and TIMER_BH. */

   /* Emulate NET_BH with special spinlock */
     spin_lock(&net_bh_lock);
```

All timers are disabled due to above assumption.

```
  /* Disable timers and wait for all timers completion */
     tasklet_disable(bh_task_vec+TIMER_BH);
```

The protocol specific function is invoked here. Timers are enabled thereafter.

```
        ret = pt->func(skb, skb->dev, pt);
        tasklet_hi_enable(bh_task_vec+TIMER_BH);
        spin_unlock(&net_bh_lock);
        return ret;
}
```