Device Driver and *eth* Level Processing of Received Packets

In this section we consider the processing of a received packet as it moves from the device driver to the *dev* 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.

*eth_type_trans()* Determine the packet type which will be used to identify the network layer handler.

*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*

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

```
 1053  static inline struct sk_buff *dev_alloc_skb(unsigned int length)
 1054  { return __dev_alloc_skb(length, GFP_ATOMIC);
 1056  }
```
Link level demultiplexing

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

\begin{verbatim}
421 struct packet_type
422 {
423   unsigned short  type;   /* really htons(ether_type).*/
424   struct net_device *dev; /* NULL is wildcarded here */
425   int    (*func) (struct sk_buff *, struct net_device *,
426                     struct packet_type *);
427   void   *data;            /* Private to the packet type */
428   struct packet_type *next;
429 }

The objective here is to determine the numeric \textit{packet type} associated with the received packet and to save it in the \textit{protocol} field of the \textit{struct sk_buff}. This packet type will be used by the generic \textit{dev} layer to identify the network layer handler to which to pass the packet. For DIX framing the key to demultiplexing is the standard packet type field that is carried in the MAC header.

\begin{verbatim}
39 #define ETH_P_LOOP 0x0060 /* Ethernet Loopback packet */
40 #define ETH_P_PUP 0x0200 /* Xerox PUP packet */
41 #define ETH_P_PUPAT 0x0201 /* Xerox PUP Addr Trans packet */
42 #define ETH_P_IP 0x0800 /* Internet Protocol packet */
43 #define ETH_P_X25 0x0805 /* CCITT X.25 */
44 #define ETH_P_ARP 0x0806 /* Address Resolution packet */
45 #define ETH_P_BPQ 0x08FF /* GBBPQ AX.25 Ethernet Packet */
46 #define ETH_P_IEEUPUP 0x0a00 /* Xerox IEEE802.3 PUP packet */
47 #define ETH_P_IEEE802PUPAT 0x0a01 /* Xerox IEEE802.3 PUP Addr Trans pkt*/

For 802.3 life is a bit more complicated:

\begin{verbatim}
73 #define ETH_P_802_3 0x0001 /* Dummy type for 802.3 frames */
74 #define ETH_P_AX25 0x0002 /* Dummy protocol id for AX.25 */
75 #define ETH_P_ALL 0x0003 /* Every packet (be careful!!!) */
76 #define ETH_P_802_2 0x0004 /* 802.2 frames */
77 #define ETH_P_SNAP 0x0005 /* Internal only */
78 #define ETH_P_DDCMP 0x0006 /* DEC DDCMP: Internal only */
\end{verbatim}
\end{verbatim}

2
Extracting the packet type from the MAC header

The `eth_type_trans()` function is a convenience function provided by the `eth` layer to the ethernet device drivers. It is invoked as shown in the following extract from 3c59x.c

```c
2419     skb->protocol = eth_type_trans(skb, dev);
```

152 /*
153 * Determine packet's protocol ID. The rule here is that we
154 * assume 802.3 if type field is short enough to be a length.
155 * This is normal and works for any 'now in use' protocol.
156 */
157
158 unsigned short eth_type_trans(struct sk_buff *skb,
159     struct net_device *dev)
160 {
161     struct ethhdr *eth;
162     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`);

163     skb->mac.raw = skb->data;
164     skb_pull(skb, dev->hard_header_len);
165     eth = skb->mac.ethernet;
166 ```
Distinguishing broadcast and multicast

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. The MAC LAYER broadcast address is held in the net_device structure. So if the destination doesn't match the broadcast address it must be a multicast.

```c
167   if (*eth->h_dest&1)
168   {
169       if(memcmp(eth->h_dest,dev->broadcast, ETH_ALEN)==0)
170           skb->pkt_type=PACKET_BROADCAST;
171       else
172           skb->pkt_type=PACKET_MULTICAST;
173   }
174
175  /*
176   *      This ALLMULTI check should be redundant by 1.4
177   *      so don't forget to remove it.
178   *      Seems, you forgot to remove it. All silly devices
179   *      seems to set IFF_PROMISC.
180   */
181
The interface might be in a promiscuous mode in which all packets appearing on the wire are received. This facility has little impact in switched networks because absent intervention at the switch a host will never see UNICASTs directed to other hosts.

```c
183   else if(1 /*dev->flags&IFF_PROMISC*/)
184   {
185       if(memcmp(eth->h_dest,dev->dev_addr, ETH_ALEN))
186           skb->pkt_type = PACKET_OTHERHOST;
187   }
188```
Determining the packet type

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. In any case this mechanism will work correctly with DIX framing regardless of max frame size.

```
189 if (ntohs(eth->h_proto) >= 1536)
190   return eth->h_proto;
191
192 rawp = skb->data;
193
194 /*
195 *This is a hack to spot IPX packets. Older Novell breaks
196 *the proto and runs IPX over 802.3 without an 802.2 LLC
197 *layer. We look for FFFF which isn't a used 802.2 SSAP/DSAP.
198 *This won't work for fault tol netware but does for the rest.
199 */
200 if (*((unsigned short *)rawp == 0xFFFF)
201   return htons(ETH_P_802_3);
202
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.

```
203 /*
204 * Real 802.2 LLC
205 */
206 return htons(ETH_P_802_2);
207 ```

```