Unlike BSD 'mbufs', we store a lot of protocol information in our packet structure, 'struct sk_buff'. The most clear case of this are the protocol header pointers we store in 'skb->h', 'skb->nh', and 'skb->mac'. It is instructive to first look at how we make use of these members, and then how BSD makes up for the lack of them in it's 'mbuf' structure.

Let us watch an IPV4 UDP packet being received by the Linux networking. First, the driver determines the header type, records 'skb->mac.raw', and pops the link level header such that 'skb->data' points just past the MAC header. For ethernet devices, this is performed by eth_type_trans().

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

        skb->mac.raw=skb->data;
        skb_pull(skb,ETH_HLEN);
        eth = eth_hdr(skb);
        skb->input_dev = dev;
        ...
```

Next, netif_receive_skb() gets the packet and sets both 'skb->h.raw' and 'skb->nh.raw' to 'skb->data'. Note that since 'skb->h' and 'skb->nh' are unions, these values show up in 'skb->h.th', 'skb->nh.iph' and friends as well.

```
int netif_receive_skb(struct sk_buff *skb)
{
        struct packet_type *ptype, *pt_prev;
        int ret = NET_RX_DROP;
        unsigned short type;

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

The packet is now passed to the input handler for the protocol indicated by 'skb->protocol'.

```
        ...
        type = skb->protocol;
        list_for_each_entry_rcu(ptype, &ptype_base[ntohs(type)&15], list) {
                if (ptype->type == type &&
                    (!ptype->dev || ptype->dev == skb->dev)) {
                        if (pt_prev)
                                ret = deliver_skb(skb, pt_prev);
                        pt_prev = ptype;
                }
        }

        if (pt_prev) {
                ret = pt_prev->func(skb, skb->dev, pt_prev);
        } else {
                kfree_skb(skb);
                /* Jamal, now you will not able to escape explaining
                 * me how you were going to use this. :-)
                 */
                ret = NET_RX_DROP;
        }
```

```
        ...
```

After it enters the stack it will be given to 'ip_rcv()' in 'net/ipv4/ip_input.c'. We verify the packet is for us, unshare it, and make sure we have at least the size of a minimal IPV4 header in the packet.

```
int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt)
{
        struct iphdr *iph;

        /* When the interface is in promisc. mode, drop all the crap
         * that it receives, do not try to analyse it.
         */
        if (skb->pkt_type == PACKET_OTHERHOST)
                goto drop;

        IP_INC_STATS_BH(IPSTATS_MIB_INRECEIVES);

        if ((skb = skb_share_check(skb, GFP_ATOMIC)) == NULL) {
                IP_INC_STATS_BH(IPSTATS_MIB_INDISCARDS);
                goto out;
        }

        if (!pskb_may_pull(skb, sizeof(struct iphdr)))
                goto inhdr_error;

        iph = skb->nh.iph;
        ...
```

We end up in ip_rcv_finish(), which looks up a route for the packet and sends it off to the route's destination. For packets destined for this host, that would be ip_local_deliver().

```
static inline int ip_rcv_finish(struct sk_buff *skb)
{
        struct net_device *dev = skb->dev;
        struct iphdr *iph = skb->nh.iph;

        /*
         *      Initialise the virtual path cache for the packet. It describes
         *      how the packet travels inside Linux networking.
         */
        if (skb->dst == NULL) {
                if (ip_route_input(skb, iph->daddr, iph->saddr, iph->tos, dev))
                        goto drop;
        }
        ...
        return dst_input(skb);
        ...
```

Now we call ip_local_deliver_finish() to pass the packet to the appropriate upper layer protocol input handler. In our case, this would be 'udp_rcv()' in 'net/ipv4/udp.c'

One of the first things ip_local_delivery_finish() does is pop the IPV4 header (which advances 'skb->data' 'ihl' bytes forward, where 'ihl' is the length of the IPV4 header, including options). Then, it assigns 'skb->h.raw' to 'skb->data'.

```
static inline int ip_local_deliver_finish(struct sk_buff *skb)
{
```

```
        int ihl = skb->nh.iph->ihl*4;

        __skb_pull(skb, ihl);

        ...

        /* Point into the IP datagram, just past the header. */
        skb->h.raw = skb->data;
        ...
```

Now, 'udp_rcv()' can just look at 'skb->h.uh' to find the UDP protocol header in the packet.

```
int udp_rcv(struct sk_buff *skb)
{
        struct sock *sk;
        struct udphdr *uh;
        unsigned short ulen;
        struct rtable *rt = (struct rtable*)skb->dst;
        u32 saddr = skb->nh.iph->saddr;
        u32 daddr = skb->nh.iph->daddr;
        int len = skb->len;

        /*
         *      Validate the packet and the UDP length.
         */
        if (!pskb_may_pull(skb, sizeof(struct udphdr)))
                goto no_header;

        uh = skb->h.uh;
        ...
```

What you should gather from that walk-through is that these pointers merely help one layer tell the next layer where their headers are. In the BSD networking stack, this information is passed down from one function to another. For example, when IP input processing passes the packet down to UDP, the data area points to the beginning of the IPV4 header, and UDP is told "and this is how many bytes the IPV4 header is". So the UDP input code expects to find the UDP header at that many bytes past the beginning of the MBUF.

Likewise, both networking implementations might need to respond to a UDP packet with an ICMP error message. In which case, it would need to get at the IPV4 header in order to generate the ICMP response packet properly.

In the Linux case, 'skb->nh.iph' says where that is. Whereas in the BSD case, the head of the MBUF points there. But unfortunately, that by itself cannot be used because the BSD stack modifies the packet header contents on input. Thus, BSD's UDP input saves a copy of the IPV4 header on the local stack before making the modifications, the passes a pointer to that on-stack copy to the ICMP packet generation.

So, in theory, we could eliminate these helper points to make the SKB structure smaller. The only area I have not explored is netfilter which may complicate things. Another potentially problematic area is the IPSEC layer which parses the upper level protocol headers in order to determine the UDP or TCP ports for the IPSEC rule lookup.

If anything, such a simplification would need to be performed over a long period of time. It would need to be done in small steps in order to safely eliminate all 'skb->h' and 'skb->nh' references in the code, for example.