The socket buffer, or "SKB", is the most fundamental data structure in the Linux networking code. Every packet sent or received is handled using this data structure.

The most fundamental parts of the SKB structure are as follows:

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

        struct sk_buff_head     *list;
 ...
```

The first two members implement list handling. Packets can exist on several kinds of lists and queues. For example, a TCP socket send queue. The third member says which list the packet is on. Learn more about SKB list handling here.

```
        struct sock             *sk;
```

This is where we record the socket assosciated with this SKB. When a packet is sent or received for a socket, the memory assosciated with the packet must be charged to the socket for proper memory accounting. Read more about socket packet buffer memory accounting here.

```
        struct timeval          stamp;
```

Here we record the timestamp for the packet, either when it arrived or when it was sent. Calculating this is somewhat expensive, so this value is only recorded if necessary. When something happens that requires that we start recording timestamps, net_enable_timestamp() is called. If that need goes away, net_disable_timestamp() is called.

Timestamps are mostly used to packet sniffers. But they are also used to implement certain socket options, and also some netfilter modules make use of this value as well.

```
        struct net_device       *dev;
        struct net_device       *input_dev;
        struct net_device       *real_dev;
```

These three members help keep track of the devices assosciated with a packet. The reason we have three different device pointers is that the main 'skb->dev' member can change as we encapsulate and decapsulate via a virtual device.

So if we are receiving a packet from a device which is part of a bonding device instance, initially 'skb->dev' will be set to point the real underlying bonding slave. When the packet enters the networking (via 'netif_receive_skb()') we save 'skb->dev' away in 'skb->real_dev' and update 'skb->dev' to point to the bonding device.

Likewise, the physical device receiving a packet always records itself in 'skb->input_dev'. In this way, no matter how many layers of virtual devices end up being decapsulated, 'skb->input_dev' can always be used to find the top-level device that actually received this packet from the network.

```
union {
        struct tcphdr   *th;
        struct udphdr   *uh;
        struct icmphdr  *icmph;
        struct igmphdr  *igmph;
        struct iphdr    *ipiph;
        struct ipv6hdr  *ipv6h;
        unsigned char   *raw;
} h;

union {
        struct iphdr    *iph;
        struct ipv6hdr  *ipv6h;
        struct arphdr   *arph;
        unsigned char   *raw;
} nh;

union {
        unsigned char   *raw;
} mac;
```

Here we store the location of the various protocol layer headers as we build outgoing packets, and parse incoming ones. For example, 'skb->mac.raw' is set by 'eth_type_trans()', when an eternet packet is received. Later, we can use this to find the location of the MAC header.

These members are potentially redundant, and could be removed. Read a discussion about that here.

```
struct  dst_entry       *dst;
```

This member is the generic route for the packet. It tells us how to get the packet to it's destination. Note that routes are used for both input and output. DST entries are about as complex as SKBs are, and thus probably deserve their own tutorial.

```
        struct  sec_path        *sp;
```

Here we store the security path traversed by the packet, if any. For example, on input IPSEC records each transformation which has been applied to the packet by a decapsulator. The records are an array of 'struct sec_decap_state' which each record the security assosciation that matched and got applied. Later, when we are trying to validate the security policy against a packet, we make sure that the transformations applied match the ones allowed by the policy.

```
        char                    cb[40];
```

This is the SKB control block. It is an opaque storage area usable by protocols, and even some drivers, to store private per-packet information. TCP uses this, for example, to store sequence numbers and retransmission state for the frame.

```
        unsigned int            len,
                                data_len,
                                mac_len,
                                csum;
```

The three length members are pretty straight-forward. The total number of bytes in the packet is 'len'. SKBs are composed of a linear data buffer, and optionally a set of 1 or more page buffers. If there are page buffers, the total number of bytes in the page buffer area is 'data_len'. Therefore the number of bytes in the linear buffer is 'skb->len - skb->data_len'. There is a shorthand function for this in 'skb_headlen()'.

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

The 'mac_len' holds the length of the MAC header. Normally, this isn't really necessary to maintain, except to implement IPSEC decapsulation of IP tunnels properly. This field is initialized once inside of 'netif_receive_skb()' to the formula 'skb->nh.raw - skb->mac.raw'.

Since we only use this for one purpose, with some clever ideas we may be able to eliminate this member in the future. For example, perhaps we can store the value in the 'struct sec_path'.

Finally, 'csum' holds the checksum of the packet. When building send packets, we copy the data in from userspace and calculate the 16-bit two's complement sum in parallel for performance. This sum is accumulated in 'skb->csum'. This helps us compute the final checksum stored in the protocol packet header checksum field. This field can end up being ignored if, for example, the device will checksum the packet for us.

On input, the 'csum' field can be used to store a checksum calculated by the device. If the device indicates 'CHECKSUM_HW' in the SKB 'ip_summed' field, this means that 'csum' is the two's complement checksum of the entire packet data area starting at 'skb->data'. This is generic enough such that both IPV4 and IPV6 checksum offloading can be supported.

```
        unsigned char           local_df,
                                cloned:1,
                                nohdr:1,
                                pkt_type,
                                ip_summed;
```

The 'local_df' field is used by the IPV4 protocol, and when set allows us to locally fragment frames which have already been fragmented. This situation can arise, for example, with IPSEC.

In order to make quick references to SKB data, Linux has the concept of SKB clones. When a clone of an SKB is made, all of the 'struct sk_buff' structure members of the clone are private to the clone. The data, however, is shared between the primary SKB and it's clone. When an SKB is cloned, the 'cloned' field will be set in both the primary and clone SKB. Otherwise is will be zero.

The 'nohdr' field is used in the support of TCP Segmentation Offload ('TSO' for short). Most devices supporting this feature need to make some minor modifications to the TCP and IP headers of an outgoing packet to get it in the right form for the hardware to process. We do not want these modifications to be seen by packet sniffers and the like. So we use this 'nohdr' field and a special bit in the data area reference count to keep track of whether the device needs to replace the data area before making the packet header modifications.

The type of the packet (basically, who is it for), is stored in the 'pkt_type' field. It takes on one of the 'PACKET_*' values defined in the 'linux/if_packet.h' header file. For example, when an incoming ethernet frame is to a destination MAC address matching the MAC address of the ethernet device it arrived on, this

field will be set to 'PACKET_HOST'. When a broadcast frame is received, it will be set to 'PACKET_BROADCAST'. And likewise when a multicast packet is received it will be set to 'PACKET_MULTICAST'.

The 'ip_summed' field describes what kind of checksumming assistence the card has provided for a receive packet. It takes on one of three values: 'CHECKSUM_NONE' if the card provided no checksum assistence, 'CHECKSUM_HW' if the two's complement checksum over the entire packet has been provides in 'skb->csum', and 'CHECKSUM_UNNECESSARY' if it is not necessary to verify the checksum of this packet. The latter usually occurs when the packet is received over the loopback device. 'CHECKSUM_UNNECESSARY' can also be used when the device only provides a 'checksum OK' indication for receive packet checksum offload.

```
__u32                   priority;
```

The 'priority' field is used in the implement of QoS. The packet's value of this field can be determined by, for example, the TOS field setting in the IPV4 header. Then, the packet scheduler and classifier layer can key off of this SKB priority value to schedule or classify the packet, as configured by the administrator.

```
unsigned short          protocol,
                        security;
```

The 'protocol' field is initialized by routines such as 'eth_type_trans()'. It takes on one of the 'ETH_P_*' values defined in the 'linux/if_ether.h' header file. Even non-ethernet devices use these ethernet protocol type values to indicate what protocol should receive the packet. As long as we always have some ethernet protocol value for each and every protocol, this should not be a problem.

The 'security' field was meant to be used in the implementation of IP Security, but that never materialized. It can probably be safely removed. Since the next field is a pointer, and thus needs to be aligned properly, eliminating the 'security' field would unfortunately not buy us any space savings.

```
void                    (*destructor)(struct sk_buff *skb);
...
unsigned int            truesize;
```

The SKB 'destructor' and 'truesize' fields are used for socket buffer accounting. See the SKB socket accounting page for details.

```
	atomic_t		users;
```

We reference count SKB objects using the 'users' field. Extra references can be obtained by invoking 'skb_get()'. An implicit single reference is present in the SKB (that is, 'users' has a value of '1') when it is first allocated. References are dropped by invoking 'kfree_skb()'.

```
	unsigned char		*head,
				*data,
				*tail,
				*end;
```

These four pointers provide the core management of the linear packet data area of an SKB. SKB data area handling is involved enough to deserve it's very own tutorial. Check it out here.