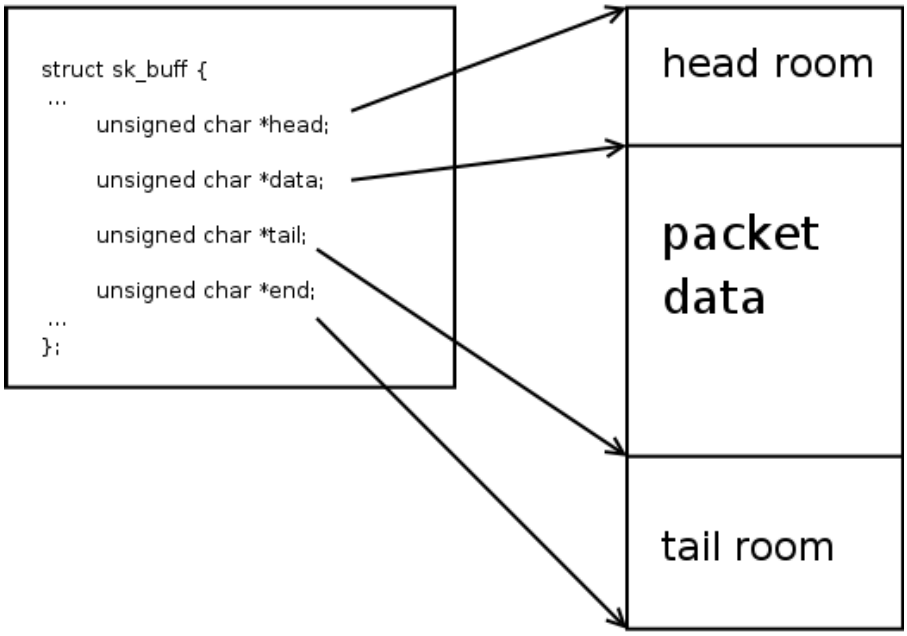
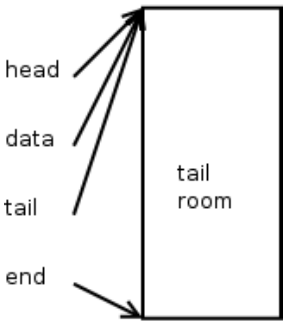


This first diagram illustrates the layout of the SKB data area and where in that area the various pointers in 'struct sk_buff' point.

The rest of this page will walk through what the SKB data area looks like in a newly allocated SKB. How to modify those pointers to add headers, add user data, and pop headers.

Also, we will discuss how page non-linear data areas are implemented. We will also discuss how to work with them.

```
skb = alloc_skb(len, GFP_KERNEL);
```

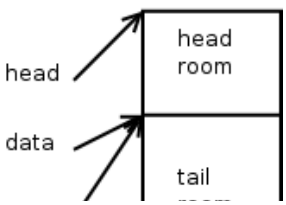


This is what a new SKB looks like right after you allocate it using `alloc_skb()`

As you can see, the head, data, and tail pointers all point to the beginning of the data buffer. And the end pointer points to the end of it. Note that all of the data area is considered tail room.

The length of this SKB is zero, it isn't very interesting since it doesn't contain any packet data at all. Let's reserve some space for protocol headers using `skb_reserve()`

```
skb_reserve(skb, header_len);
```



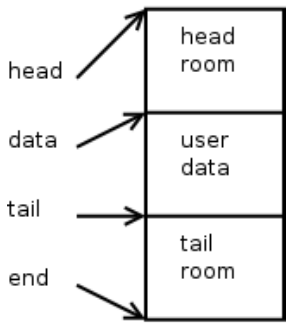
This is what a new SKB looks like right after the `skb_reserve()` call.

Typically, when building output packets, we reserve enough bytes for the maximum amount of header space we think we'll need. Most IPV4 protocols can do this by using the socket value `sk->sk_prot->max_header`.

When setting up receive packets that an ethernet device will DMA into, we typically call `skb_reserve(skb, NET_IP_ALIGN)`. By default `NET_IP_ALIGN` is defined to '2'. This makes it so that, after the ethernet header, the protocol header will be aligned on at least a 4-byte boundary. Nearly all of the IPV4 and IPV6 protocol processing assumes that the headers are properly aligned.

Let's now add some user data to the packet.

```
unsigned char *data = skb_put(skb, user_data_len);
int err = 0;
skb->csum = csum_and_copy_from_user(user_pointer, data,
                                    user_data_len, 0, &err);
if (err)
        goto user_fault;
```



This is what a new SKB looks like right after the user data is added.

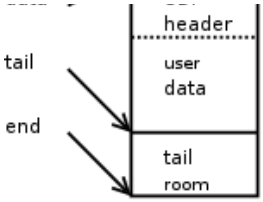
`skb_put()` advances 'skb->tail' by the specified number of bytes, it also increments 'skb->len' by that number of bytes as well. This routine must not be called on a SKB that has any paged data. You must also be sure that there is enough tail room in the SKB for the amount of bytes you are trying to put. Both of these conditions are checked for by `skb_put()` and an assertion failure will trigger if either rule is violated.

The computed checksum is remembered in 'skb->csum'. Now, it's time to build the protocol headers. We'll build a UDP header, then one for IPV4.

```
struct inet_sock *inet = inet_sk(sk);
struct flowi *fl = &inet->cork.fl;
struct udphdr *uh;

skb->h.raw = skb_push(skb, sizeof(struct udphdr));
uh = skb->h.uh
uh->source = fl->fl_ip_sport;
uh->dest = fl->fl_ip_dport;
uh->len = htons(user_data_len);
uh->check = 0;
skb->csum = csum_partial((char *)uh,
                         sizeof(struct udphdr), skb->csum);
uh->check = csum_tcpudp_magic(fl->fl4_src, fl->fl4_dst,
                              user_data_len, IPPROTO_UDP, skb->csum);
if (uh->check == 0)
        uh->check = -1;
```

This is what a new SKB looks like after we push the UDP header to the front of the SKB.



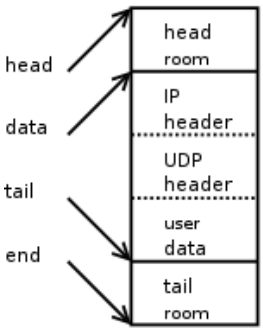
skb_push() will decrement the 'skb->data' pointer by the specified number of bytes. It will also increment 'skb->len' by that number of bytes as well. The caller must make sure there is enough head room for the push being performed. This condition is checked for by skb_push() and an assertion failure will trigger if this rule is violated.

Now, it's time to tack on an IPV4 header.

```
struct rtable *rt = inet->cork.rt;
struct iphdr *iph;

skb->nh.raw = skb_push(skb, sizeof(struct iphdr));
iph = skb->nh.iph;
iph->version = 4;
iph->ihl = 5;
iph->tos = inet->tos;
iph->tot_len = htons(skb->len);
iph->frag_off = 0;
iph->id = htons(inet->id++);
iph->ttl = ip_select_ttl(inet, &rt->u.dst);
iph->protocol = sk->sk_protocol; /* IPPROTO_UDP in this case */
iph->saddr = rt->rt_src;
iph->daddr = rt->rt_dst;
ip_send_check(iph);

skb->priority = sk->sk_priority;
skb->dst = dst_clone(&rt->u.dst);
```



This is what a new SKB looks like after we push the IPv4 header to the front of the SKB.

Just as above for UDP, skb_push() decrements 'skb->data' and increments 'skb->len'. We update the 'skb->nh.raw' pointer to the beginning of the new space, and build the IPv4 header.

This packet is basically ready to be pushed out to the device once we have the necessary information to build the ethernet header (from the generic neighbour layer and ARP).

Things start to get a little bit more complicated once paged data begins to be used. For the most part the ability to use [page, offset, len] tuples for SKB data came about so that file system file contents could be directly sent over a socket. But, as it turns out, it is sometimes beneficial to use this for nomal buffering of process sendmsg() data.

It must be understood that once paged data starts to be used on an SKB, this puts a specific restriction on all future SKB data area operations. In particular, it is no longer possible to do skb_put() operations.

We will now mention that there are actually two length variables assosciated with an SKB, `len` and `data_len`. The latter only comes into play when there is paged data in the SKB. `skb->data_len` tells how many bytes of paged data there are in the SKB. From this we can derive a few more things:

- The existence of paged data in an SKB is indicated by `skb->data_len` being non-zero. This is codified in the helper routine `skb_is_nonlinear()` so that it the function you should use to test this.
- The amount of non-paged data at `skb->data` can be calculated as `skb->len - skb->data_len`. Again, there is a helper routine already defined for this called `skb_headlen()` so please use that.

The main abstraction is that, when there is paged data, the packet begins at `skb->data` for `skb_headlen(skb)` bytes, then continues on into the paged data area for `skb->data_len` bytes. That is why it is illogical to try and do an `skb_put(skb)` when there is paged data. You have to add data onto the end of the paged data area instead.

Each chunk of paged data in an SKB is described by the following structure:

```
struct skb_frag_struct {
        struct page *page;
        __u16 page_offset;
        __u16 size;
};
```

There is a pointer to the page (which you must hold a proper reference to), the offset within the page where this chunk of paged data starts, and how many bytes are there.

The paged frags are organized into an array in the shared SKB area, defined by this structure:

```
#define MAX_SKB_FRAGS (65536/PAGE_SIZE + 2)

struct skb_shared_info {
        atomic_t dataref;
        unsigned int    nr_frags;
        unsigned short  tso_size;
        unsigned short  tso_segs;
        struct sk_buff  *frag_list;
        skb_frag_t      frags[MAX_SKB_FRAGS];
};
```

The `nr_frags` member states how many frags there are active in the `frags[]` array. The `tso_size` and `tso_segs` is used to convey information to the device driver for TCP segmentation offload. The `frag_list` is used to maintain a chain of SKBs organized for fragmentation purposes, it is _not_ used for maintaining paged data. And finally the `frags[]` holds the frag descriptors themselves.

A helper routine is available to help you fill in page descriptors.

```
void skb_fill_page_desc(struct sk_buff *skb, int i,
                        struct page *page,
                        int off, int size)
```

This fills the `i'th` page vector to point to page at offset `off` of size `size`. It also updates the `nr_frags` member to be one past `i`.

If you wish to simply extend an existing frag entry by some number of bytes, increment the `size` member by that amount.

With all of the complications imposed by non-linear SKBs, it may seem difficult to inspect areas of a packet in a straightforward way, or to copy data out from a packet into another buffer. This is not the case. There are two helper routines available which make this pretty easy.

First, we have:

---

void *skb_header_pointer(const struct sk_buff *skb, int offset, int len, void *buffer)

You give it the SKB, the offset (in bytes) to the piece of data you are interested in, the number of bytes you want, and a local buffer which is to be used _only_ if the data you are interested in resides in the non-linear data area.

You are returned a pointer to the data item, or NULL if you asked for an invalid offset and len parameter. This pointer could be one of two things. First, if what you asked for is directly in the `skb->data` linear data area, you are given a direct pointer into there. Else, you are given the buffer pointer you passed in.

Code inspecting packet headers on the output path, especially, should use this routine to read and interpret protocol headers. The netfilter layer uses this function heavily.

For larger pieces of data other than protocol headers, it may be more appropriate to use the following helper routine instead.

```
int skb_copy_bits(const struct sk_buff *skb, int offset,
                  void *to, int len);
```

This will copy the specified number of bytes, and the specified offset, of the given SKB into the `'to'` buffer. This is used for copies of SKB data into kernel buffers, and therefore it is not to be used for copying SKB data into userspace. There is another helper routine for that:

```
int skb_copy_datagram_iovec(const struct sk_buff *from,
                            int offset, struct iovec *to,
                            int size);
```

Here, the user's data area is described by the given IOVEC. The other parameters are nearly identical to those passed in to `skb_copy_bits()` above.
