Linux Cross Reference

Free Electrons

Embedded Linux Experts

Linux/include/linux/skbuff.h

```
 1 /*
 2  *      Definitions for the 'struct sk_buff' memory handlers.
 3  *
 4  *      Authors:
 5  *              Alan Cox, <gw4pts@gw4pts.ampr.org>
 6  *              Florian La Roche, <rzsfl@rz.uni-sb.de>
 7  *
 8  *      This program is free software; you can redistribute it and/or
 9  *      modify it under the terms of the GNU General Public License
10  *      as published by the Free Software Foundation; either version
11  *      2 of the License, or (at your option) any later version.
12  */
13
14 #ifndef _LINUX_SKBUFF_H
15 #define _LINUX_SKBUFF_H
16
17 #include <linux/kernel.h>
18 #include <linux/kmemcheck.h>
19 #include <linux/compiler.h>
20 #include <linux/time.h>
21 #include <linux/bug.h>
22 #include <linux/cache.h>
23 #include <linux/rbtree.h>
24 #include <linux/socket.h>
25
26 #include <linux/atomic.h>
27 #include <asm/types.h>
28 #include <linux/spinlock.h>
29 #include <linux/net.h>
30 #include <linux/textsearch.h>
31 #include <net/checksum.h>
32 #include <linux/rcupdate.h>
33 #include <linux/hrtimer.h>
34 #include <linux/dma-mapping.h>
35 #include <linux/netdev_features.h>
36 #include <linux/sched.h>
37 #include <net/flow_dissector.h>
38 #include <linux/splice.h>
39 #include <linux/in6.h>
40
41 /* A. Checksumming of received packets by device.
42  *
43  * CHECKSUM_NONE:
44  *
```

```
 45  *   Device failed to checksum this packet e.g. due to lack of capabilities.
 46  *   The packet contains full (though not verified) checksum in packet but
 47  *   not in skb->csum. Thus, skb->csum is undefined in this case.
 48  *
 49  * CHECKSUM_UNNECESSARY:
 50  *
 51  *   The hardware you're dealing with doesn't calculate the full checksum
 52  *   (as in CHECKSUM_COMPLETE), but it does parse headers and verify checksums
 53  *   for specific protocols. For such packets it will set CHECKSUM_UNNECESSARY
 54  *   if their checksums are okay. skb->csum is still undefined in this case
 55  *   though. It is a bad option, but, unfortunately, nowadays most vendors do
 56  *   this. Apparently with the secret goal to sell you new devices, when you
 57  *   will add new protocol to your host, f.e. IPv6 8)
 58  *
 59  *   CHECKSUM_UNNECESSARY is applicable to following protocols:
 60  *     TCP: IPv6 and IPv4.
 61  *     UDP: IPv4 and IPv6. A device may apply CHECKSUM_UNNECESSARY to a
 62  *       zero UDP checksum for either IPv4 or IPv6, the networking stack
 63  *       may perform further validation in this case.
 64  *     GRE: only if the checksum is present in the header.
 65  *     SCTP: indicates the CRC in SCTP header has been validated.
 66  *
 67  *   skb->csum_level indicates the number of consecutive checksums found in
 68  *   the packet minus one that have been verified as CHECKSUM_UNNECESSARY.
 69  *   For instance if a device receives an IPv6->UDP->GRE->IPv4->TCP packet
 70  *   and a device is able to verify the checksums for UDP (possibly zero),
 71  *   GRE (checksum flag is set), and TCP-- skb->csum_level would be set to
 72  *   two. If the device were only able to verify the UDP checksum and not
 73  *   GRE, either because it doesn't support GRE checksum of because GRE
 74  *   checksum is bad, skb->csum_level would be set to zero (TCP checksum is
 75  *   not considered in this case).
 76  *
 77  * CHECKSUM_COMPLETE:
 78  *
 79  *   This is the most generic way. The device supplied checksum of the _whole_
 80  *   packet as seen by netif_rx() and fills out in skb->csum. Meaning, the
 81  *   hardware doesn't need to parse L3/L4 headers to implement this.
 82  *
 83  *   Note: Even if device supports only some protocols, but is able to produce
 84  *   skb->csum, it MUST use CHECKSUM_COMPLETE, not CHECKSUM_UNNECESSARY.
 85  *
 86  * CHECKSUM_PARTIAL:
 87  *
 88  *   A checksum is set up to be offloaded to a device as described in the
 89  *   output description for CHECKSUM_PARTIAL. This may occur on a packet
 90  *   received directly from another Linux OS, e.g., a virtualized Linux kernel
 91  *   on the same host, or it may be set in the input path in GRO or remote
 92  *   checksum offload. For the purposes of checksum verification, the checksum
 93  *   referred to by skb->csum_start + skb->csum_offset and any preceding
 94  *   checksums in the packet are considered verified. Any checksums in the
 95  *   packet that are after the checksum being offloaded are not considered to
 96  *   be verified.
 97  *
 98  * B. Checksumming on output.
 99  *
100  * CHECKSUM_NONE:
101  *
102  *   The skb was already checksummed by the protocol, or a checksum is not
103  *   required.
104  *
105  * CHECKSUM_PARTIAL:
106  *
107  *   The device is required to checksum the packet as seen by hard_start_xmit()
108  *   from skb->csum_start up to the end, and to record/write the checksum at
109  *   offset skb->csum_start + skb->csum_offset.
110  *
```

```
111  *   The device must show its capabilities in dev->features, set up at device
112  *   setup time, e.g. netdev_features.h:
113  *
114  *      NETIF_F_HW_CSUM - It's a clever device, it's able to checksum everything.
115  *      NETIF_F_IP_CSUM - Device is dumb, it's able to checksum only TCP/UDP over
116  *                        IPv4. Sigh. Vendors like this way for an unknown reason.
117  *                        Though, see comment above about CHECKSUM_UNNECESSARY. 8)
118  *      NETIF_F_IPV6_CSUM - About as dumb as the last one but does IPv6 instead.
119  *      NETIF_F_...     - Well, you get the picture.
120  *
121  * CHECKSUM_UNNECESSARY:
122  *
123  *   Normally, the device will do per protocol specific checksumming. Protocol
124  *   implementations that do not want the NIC to perform the checksum
125  *   calculation should use this flag in their outgoing skbs.
126  *
127  *      NETIF_F_FCOE_CRC - This indicates that the device can do FCoE FC CRC
128  *                         offload. Correspondingly, the FCoE protocol driver
129  *                         stack should use CHECKSUM_UNNECESSARY.
130  *
131  * Any questions? No questions, good.          --ANK
132  */
133
134 /* Don't change this without changing skb_csum_unnecessary! */
135 #define CHECKSUM_NONE           0
136 #define CHECKSUM_UNNECESSARY    1
137 #define CHECKSUM_COMPLETE       2
138 #define CHECKSUM_PARTIAL        3
139
140 /* Maximum value in skb->csum_level */
141 #define SKB_MAX_CSUM_LEVEL      3
142
143 #define SKB_DATA_ALIGN(X)       ALIGN(X, SMP_CACHE_BYTES)
144 #define SKB_WITH_OVERHEAD(X)    \
145         ((X) - SKB_DATA_ALIGN(sizeof(struct skb_shared_info)))
146 #define SKB_MAX_ORDER(X, ORDER) \
147         SKB_WITH_OVERHEAD((PAGE_SIZE << (ORDER)) - (X))
148 #define SKB_MAX_HEAD(X)         (SKB_MAX_ORDER((X), 0))
149 #define SKB_MAX_ALLOC           (SKB_MAX_ORDER(0, 2))
150
151 /* return minimum truesize of one skb containing X bytes of data */
152 #define SKB_TRUESIZE(X) ((X) +                                          \
153                          SKB_DATA_ALIGN(sizeof(struct sk_buff)) +       \
154                          SKB_DATA_ALIGN(sizeof(struct skb_shared_info)))
155
156 struct net_device;
157 struct scatterlist;
158 struct pipe_inode_info;
159 struct iov_iter;
160 struct napi_struct;
161
162 #if defined(CONFIG_NF_CONNTRACK) || defined(CONFIG_NF_CONNTRACK_MODULE)
163 struct nf_conntrack {
164         atomic_t use;
165 };
166 #endif
167
168 #if IS_ENABLED(CONFIG_BRIDGE_NETFILTER)
169 struct nf_bridge_info {
170         atomic_t                use;
171         enum {
172                 BRNF_PROTO_UNCHANGED,
173                 BRNF_PROTO_8021Q,
174                 BRNF_PROTO_PPPOE
175         } orig_proto:8;
176         bool                    pkt_otherhost;
```

```
177         __u16                   frag_max_size;
178         unsigned int            mask;
179         struct net_device       *physindev;
180         union {
181                 struct net_device *physoutdev;
182                 char neigh_header[8];
183         };
184         union {
185                 __be32          ipv4_daddr;
186                 struct in6_addr ipv6_daddr;
187         };
188 };
189 #endif
190
191 struct sk_buff_head {
192         /* These two members must be first. */
193         struct sk_buff  *next;
194         struct sk_buff  *prev;
195
196         __u32           qlen;
197         spinlock_t      lock;
198 };
199
200 struct sk_buff;
201
202 /* To allow 64K frame to be packed as single skb without frag_list we
203  * require 64K/PAGE_SIZE pages plus 1 additional page to allow for
204  * buffers which do not start on a page boundary.
205  *
206  * Since GRO uses frags we allocate at least 16 regardless of page
207  * size.
208  */
209 #if (65536/PAGE_SIZE + 1) < 16
210 #define MAX_SKB_FRAGS 16UL
211 #else
212 #define MAX_SKB_FRAGS (65536/PAGE_SIZE + 1)
213 #endif
214
215 typedef struct skb_frag_struct skb_frag_t;
216
217 struct skb_frag_struct {
218         struct {
219                 struct page *p;
220         } page;
221 #if (BITS_PER_LONG > 32) || (PAGE_SIZE >= 65536)
222         __u32 page_offset;
223         __u32 size;
224 #else
225         __u16 page_offset;
226         __u16 size;
227 #endif
228 };
229
230 static inline unsigned int skb_frag_size(const skb_frag_t *frag)
231 {
232         return frag->size;
233 }
234
235 static inline void skb_frag_size_set(skb_frag_t *frag, unsigned int size)
236 {
237         frag->size = size;
238 }
239
240 static inline void skb_frag_size_add(skb_frag_t *frag, int delta)
241 {
242         frag->size += delta;
```

```
243 }
244
245 static inline void skb_frag_size_sub(skb_frag_t *frag, int delta)
246 {
247         frag->size -= delta;
248 }
249
250 #define HAVE_HW_TIME_STAMP
251
252 /**
253  * struct skb_shared_hwtstamps - hardware time stamps
254  * @hwtstamp:   hardware time stamp transformed into duration
255  *              since arbitrary point in time
256  *
257  * Software time stamps generated by ktime_get_real() are stored in
258  * skb->tstamp.
259  *
260  * hwtstamps can only be compared against other hwtstamps from
261  * the same device.
262  *
263  * This structure is attached to packets as part of the
264  * &skb_shared_info. Use skb_hwtstamps() to get a pointer.
265  */
266 struct skb_shared_hwtstamps {
267         ktime_t hwtstamp;
268 };
269
270 /* Definitions for tx_flags in struct skb_shared_info */
271 enum {
272         /* generate hardware time stamp */
273         SKBTX_HW_TSTAMP = 1 << 0,
274
275         /* generate software time stamp when queueing packet to NIC */
276         SKBTX_SW_TSTAMP = 1 << 1,
277
278         /* device driver is going to provide hardware time stamp */
279         SKBTX_IN_PROGRESS = 1 << 2,
280
281         /* device driver supports TX zero-copy buffers */
282         SKBTX_DEV_ZEROCOPY = 1 << 3,
283
284         /* generate wifi status information (where possible) */
285         SKBTX_WIFI_STATUS = 1 << 4,
286
287         /* This indicates at least one fragment might be overwritten
288          * (as in vmsplice(), sendfile() ...)
289          * If we need to compute a TX checksum, we'll need to copy
290          * all frags to avoid possible bad checksum
291          */
292         SKBTX_SHARED_FRAG = 1 << 5,
293
294         /* generate software time stamp when entering packet scheduling */
295         SKBTX_SCHED_TSTAMP = 1 << 6,
296
297         /* generate software timestamp on peer data acknowledgment */
298         SKBTX_ACK_TSTAMP = 1 << 7,
299 };
300
301 #define SKBTX_ANY_SW_TSTAMP     (SKBTX_SW_TSTAMP    | \
302                                  SKBTX_SCHED_TSTAMP | \
303                                  SKBTX_ACK_TSTAMP)
304 #define SKBTX_ANY_TSTAMP        (SKBTX_HW_TSTAMP | SKBTX_ANY_SW_TSTAMP)
305
306 /*
307  * The callback notifies userspace to release buffers when skb DMA is done in
308  * lower device, the skb last reference should be 0 when calling this.
```

```
309  * The zerocopy_success argument is true if zero copy transmit occurred,
310  * false on data copy or out of memory error caused by data copy attempt.
311  * The ctx field is used to track device context.
312  * The desc field is used to track userspace buffer index.
313  */
314 struct ubuf_info {
315         void (*callback)(struct ubuf_info *, bool zerocopy_success);
316         void *ctx;
317         unsigned long desc;
318 };
319
320 /* This data is invariant across clones and lives at
321  * the end of the header data, ie. at skb->end.
322  */
323 struct skb_shared_info {
324         unsigned char   nr_frags;
325         __u8            tx_flags;
326         unsigned short  gso_size;
327         /* Warning: this field is not always filled in (UFO)! */
328         unsigned short  gso_segs;
329         unsigned short  gso_type;
330         struct sk_buff  *frag_list;
331         struct skb_shared_hwtstamps hwtstamps;
332         u32             tskey;
333         __be32          ip6_frag_id;
334
335         /*
336          * Warning : all fields before dataref are cleared in __alloc_skb()
337          */
338         atomic_t        dataref;
339
340         /* Intermediate layers must ensure that destructor_arg
341          * remains valid until skb destructor */
342         void *          destructor_arg;
343
344         /* must be last field, see pskb_expand_head() */
345         skb_frag_t      frags[MAX_SKB_FRAGS];
346 };
347
348 /* We divide dataref into two halves.  The higher 16 bits hold references
349  * to the payload part of skb->data.  The lower 16 bits hold references to
350  * the entire skb->data.  A clone of a headerless skb holds the length of
351  * the header in skb->hdr_len.
352  *
353  * All users must obey the rule that the skb->data reference count must be
354  * greater than or equal to the payload reference count.
355  *
356  * Holding a reference to the payload part means that the user does not
357  * care about modifications to the header part of skb->data.
358  */
359 #define SKB_DATAREF_SHIFT 16
360 #define SKB_DATAREF_MASK ((1 << SKB_DATAREF_SHIFT) - 1)
361
362
363 enum {
364         SKB_FCLONE_UNAVAILABLE, /* skb has no fclone (from head_cache) */
365         SKB_FCLONE_ORIG,        /* orig skb (from fclone_cache) */
366         SKB_FCLONE_CLONE,       /* companion fclone skb (from fclone_cache) */
367 };
368
369 enum {
370         SKB_GSO_TCPV4 = 1 << 0,
371         SKB_GSO_UDP = 1 << 1,
372
373         /* This indicates the skb is from an untrusted source. */
374         SKB_GSO_DODGY = 1 << 2,
```

```
375
376         /* This indicates the tcp segment has CWR set. */
377         SKB_GSO_TCP_ECN = 1 << 3,
378
379         SKB_GSO_TCPV6 = 1 << 4,
380
381         SKB_GSO_FCOE = 1 << 5,
382
383         SKB_GSO_GRE = 1 << 6,
384
385         SKB_GSO_GRE_CSUM = 1 << 7,
386
387         SKB_GSO_IPIP = 1 << 8,
388
389         SKB_GSO_SIT = 1 << 9,
390
391         SKB_GSO_UDP_TUNNEL = 1 << 10,
392
393         SKB_GSO_UDP_TUNNEL_CSUM = 1 << 11,
394
395         SKB_GSO_TUNNEL_REMCSUM = 1 << 12,
396 };
397
398 #if BITS_PER_LONG > 32
399 #define NET_SKBUFF_DATA_USES_OFFSET 1
400 #endif
401
402 #ifdef NET_SKBUFF_DATA_USES_OFFSET
403 typedef unsigned int sk_buff_data_t;
404 #else
405 typedef unsigned char *sk_buff_data_t;
406 #endif
407
408 /**
409  * struct skb_mstamp - multi resolution time stamps
410  * @stamp_us: timestamp in us resolution
411  * @stamp_jiffies: timestamp in jiffies
412  */
413 struct skb_mstamp {
414         union {
415                 u64             v64;
416                 struct {
417                         u32     stamp_us;
418                         u32     stamp_jiffies;
419                 };
420         };
421 };
422
423 /**
424  * skb_mstamp_get - get current timestamp
425  * @cl: place to store timestamps
426  */
427 static inline void skb_mstamp_get(struct skb_mstamp *cl)
428 {
429         u64 val = local_clock();
430
431         do_div(val, NSEC_PER_USEC);
432         cl->stamp_us = (u32)val;
433         cl->stamp_jiffies = (u32)jiffies;
434 }
435
436 /**
437  * skb_mstamp_delta - compute the difference in usec between two skb_mstamp
438  * @t1: pointer to newest sample
439  * @t0: pointer to oldest sample
440  */
```

```
441 static inline u32 skb_mstamp_us_delta(const struct skb_mstamp *t1,
442                                        const struct skb_mstamp *t0)
443 {
444         s32 delta_us = t1->stamp_us - t0->stamp_us;
445         u32 delta_jiffies = t1->stamp_jiffies - t0->stamp_jiffies;
446
447         /* If delta_us is negative, this might be because interval is too big,
448          * or local_clock() drift is too big : fallback using jiffies.
449          */
450         if (delta_us <= 0 ||
451             delta_jiffies >= (INT_MAX / (USEC_PER_SEC / HZ)))
452
453                 delta_us = jiffies_to_usecs(delta_jiffies);
454
455         return delta_us;
456 }
457
458
459 /**
460  *      struct sk_buff - socket buffer
461  *      @next: Next buffer in list
462  *      @prev: Previous buffer in list
463  *      @tstamp: Time we arrived/left
464  *      @rbnode: RB tree node, alternative to next/prev for netem/tcp
465  *      @sk: Socket we are owned by
466  *      @dev: Device we arrived on/are leaving by
467  *      @cb: Control buffer. Free for use by every layer. Put private vars here
468  *      @_skb_refdst: destination entry (with norefcount bit)
469  *      @sp: the security path, used for xfrm
470  *      @len: Length of actual data
471  *      @data_len: Data length
472  *      @mac_len: Length of link layer header
473  *      @hdr_len: writable header length of cloned skb
474  *      @csum: Checksum (must include start/offset pair)
475  *      @csum_start: Offset from skb->head where checksumming should start
476  *      @csum_offset: Offset from csum_start where checksum should be stored
477  *      @priority: Packet queueing priority
478  *      @ignore_df: allow local fragmentation
479  *      @cloned: Head may be cloned (check refcnt to be sure)
480  *      @ip_summed: Driver fed us an IP checksum
481  *      @nohdr: Payload reference only, must not modify header
482  *      @nfctinfo: Relationship of this skb to the connection
483  *      @pkt_type: Packet class
484  *      @fclone: skbuff clone status
485  *      @ipvs_property: skbuff is owned by ipvs
486  *      @peeked: this packet has been seen already, so stats have been
487  *              done for it, don't do them again
488  *      @nf_trace: netfilter packet trace flag
489  *      @protocol: Packet protocol from driver
490  *      @destructor: Destruct function
491  *      @nfct: Associated connection, if any
492  *      @nf_bridge: Saved data about a bridged frame - see br_netfilter.c
493  *      @skb_iif: ifindex of device we arrived on
494  *      @tc_index: Traffic control index
495  *      @tc_verd: traffic control verdict
496  *      @hash: the packet hash
497  *      @queue_mapping: Queue mapping for multiqueue devices
498  *      @xmit_more: More SKBs are pending for this queue
499  *      @ndisc_nodetype: router type (from link layer)
500  *      @ooo_okay: allow the mapping of a socket to a queue to be changed
501  *      @l4_hash: indicate hash is a canonical 4-tuple hash over transport
502  *              ports.
503  *      @sw_hash: indicates hash was computed in software stack
504  *      @wifi_acked_valid: wifi_acked was set
505  *      @wifi_acked: whether frame was acked on wifi or not
506  *      @no_fcs:  Request NIC to treat last 4 bytes as Ethernet FCS
```

```
507  *      @napi_id: id of the NAPI struct this skb came from
508  *      @secmark: security marking
509  *      @mark: Generic packet mark
510  *      @vlan_proto: vlan encapsulation protocol
511  *      @vlan_tci: vlan tag control information
512  *      @inner_protocol: Protocol (encapsulation)
513  *      @inner_transport_header: Inner transport layer header (encapsulation)
514  *      @inner_network_header: Network layer header (encapsulation)
515  *      @inner_mac_header: Link layer header (encapsulation)
516  *      @transport_header: Transport layer header
517  *      @network_header: Network layer header
518  *      @mac_header: Link layer header
519  *      @tail: Tail pointer
520  *      @end: End pointer
521  *      @head: Head of buffer
522  *      @data: Data head pointer
523  *      @truesize: Buffer size
524  *      @users: User count - see {datagram,tcp}.c
525  */
526
527 struct sk_buff {
528         union {
529                 struct {
530                         /* These two members must be first. */
531                         struct sk_buff          *next;
532                         struct sk_buff          *prev;
533
534                         union {
535                                 ktime_t         tstamp;
536                                 struct skb_mstamp skb_mstamp;
537                         };
538                 };
539                 struct rb_node  rbnode; /* used in netem & tcp stack */
540         };
541         struct sock             *sk;
542         struct net_device       *dev;
543
544         /*
545          * This is the control buffer. It is free to use for every
546          * layer. Please put your private variables there. If you
547          * want to keep them across layers you have to do a skb_clone()
548          * first. This is owned by whoever has the skb queued ATM.
549          */
550         char                    cb[48] __aligned(8);
551
552         unsigned long           _skb_refdst;
553         void                    (*destructor)(struct sk_buff *skb);
554 #ifdef CONFIG_XFRM
555         struct  sec_path        *sp;
556 #endif
557 #if defined(CONFIG_NF_CONNTRACK) || defined(CONFIG_NF_CONNTRACK_MODULE)
558         struct nf_conntrack     *nfct;
559 #endif
560 #if IS_ENABLED(CONFIG_BRIDGE_NETFILTER)
561         struct nf_bridge_info   *nf_bridge;
562 #endif
563         unsigned int            len,
564                                 data_len;
565         __u16                   mac_len,
566                                 hdr_len;
567
568         /* Following fields are _not_ copied in __copy_skb_header()
569          * Note that queue_mapping is here mostly to fill a hole.
570          */
571         kmemcheck_bitfield_begin(flags1);
572         __u16                   queue_mapping;
```

```
573         __u8                    cloned:1,
574                                 nohdr:1,
575                                 fclone:2,
576                                 peeked:1,
577                                 head_frag:1,
578                                 xmit_more:1;
579         /* one bit hole */
580         kmemcheck_bitfield_end(flags1);
581
582         /* fields enclosed in headers_start/headers_end are copied
583          * using a single memcpy() in __copy_skb_header()
584          */
585         /* private: */
586         __u32                   headers_start[0];
587         /* public: */
588
589 /* if you move pkt_type around you also must adapt those constants */
590 #ifdef __BIG_ENDIAN_BITFIELD
591 #define PKT_TYPE_MAX    (7 << 5)
592 #else
593 #define PKT_TYPE_MAX    7
594 #endif
595 #define PKT_TYPE_OFFSET()       offsetof(struct sk_buff, __pkt_type_offset)
596
597         __u8                    __pkt_type_offset[0];
598         __u8                    pkt_type:3;
599         __u8                    pfmemalloc:1;
600         __u8                    ignore_df:1;
601         __u8                    nfctinfo:3;
602
603         __u8                    nf_trace:1;
604         __u8                    ip_summed:2;
605         __u8                    ooo_okay:1;
606         __u8                    l4_hash:1;
607         __u8                    sw_hash:1;
608         __u8                    wifi_acked_valid:1;
609         __u8                    wifi_acked:1;
610
611         __u8                    no_fcs:1;
612         /* Indicates the inner headers are valid in the skbuff. */
613         __u8                    encapsulation:1;
614         __u8                    encap_hdr_csum:1;
615         __u8                    csum_valid:1;
616         __u8                    csum_complete_sw:1;
617         __u8                    csum_level:2;
618         __u8                    csum_bad:1;
619
620 #ifdef CONFIG_IPV6_NDISC_NODETYPE
621         __u8                    ndisc_nodetype:2;
622 #endif
623         __u8                    ipvs_property:1;
624         __u8                    inner_protocol_type:1;
625         __u8                    remcsum_offload:1;
626         /* 3 or 5 bit hole */
627
628 #ifdef CONFIG_NET_SCHED
629         __u16                   tc_index;       /* traffic control index */
630 #ifdef CONFIG_NET_CLS_ACT
631         __u16                   tc_verd;        /* traffic control verdict */
632 #endif
633 #endif
634
635         union {
636                 __wsum          csum;
637                 struct {
638                         __u16   csum_start;
```

```
639                         __u16   csum_offset;
640                 };
641         };
642         __u32                   priority;
643         int                     skb_iif;
644         __u32                   hash;
645         __be16                  vlan_proto;
646         __u16                   vlan_tci;
647 #if defined(CONFIG_NET_RX_BUSY_POLL) || defined(CONFIG_XPS)
648         union {
649                 unsigned int    napi_id;
650                 unsigned int    sender_cpu;
651         };
652 #endif
653 #ifdef CONFIG_NETWORK_SECMARK
654         __u32                   secmark;
655 #endif
656         union {
657                 __u32           mark;
658                 __u32           reserved_tailroom;
659         };
660
661         union {
662                 __be16          inner_protocol;
663                 __u8            inner_ipproto;
664         };
665
666         __u16                   inner_transport_header;
667         __u16                   inner_network_header;
668         __u16                   inner_mac_header;
669
670         __be16                  protocol;
671         __u16                   transport_header;
672         __u16                   network_header;
673         __u16                   mac_header;
674
675         /* private: */
676         __u32                   headers_end[0];
677         /* public: */
678
679         /* These elements must be at the end, see alloc_skb() for details.  */
680         sk_buff_data_t          tail;
681         sk_buff_data_t          end;
682         unsigned char           *head,
683                                 *data;
684         unsigned int            truesize;
685         atomic_t                users;
686 };
687
688 #ifdef __KERNEL__
689 /*
690  *      Handling routines are only of interest to the kernel
691  */
692 #include <linux/slab.h>
693
694
695 #define SKB_ALLOC_FCLONE        0x01
696 #define SKB_ALLOC_RX            0x02
697 #define SKB_ALLOC_NAPI          0x04
698
699 /* Returns true if the skb was allocated from PFMEMALLOC reserves */
700 static inline bool skb_pfmemalloc(const struct sk_buff *skb)
701 {
702         return unlikely(skb->pfmemalloc);
703 }
704
```

```
705 /*
706  * skb might have a dst pointer attached, refcounted or not.
707  * _skb_refdst low order bit is set if refcount was _not_ taken
708  */
709 #define SKB_DST_NOREF   1UL
710 #define SKB_DST_PTRMASK ~(SKB_DST_NOREF)
711
712 /**
713  * skb_dst - returns skb dst_entry
714  * @skb: buffer
715  *
716  * Returns skb dst_entry, regardless of reference taken or not.
717  */
718 static inline struct dst_entry *skb_dst(const struct sk_buff *skb)
719 {
720         /* If refdst was not refcounted, check we still are in a
721          * rcu_read_lock section
722          */
723         WARN_ON((skb->_skb_refdst & SKB_DST_NOREF) &&
724                 !rcu_read_lock_held() &&
725                 !rcu_read_lock_bh_held());
726         return (struct dst_entry *)(skb->_skb_refdst & SKB_DST_PTRMASK);
727 }
728
729 /**
730  * skb_dst_set - sets skb dst
731  * @skb: buffer
732  * @dst: dst entry
733  *
734  * Sets skb dst, assuming a reference was taken on dst and should
735  * be released by skb_dst_drop()
736  */
737 static inline void skb_dst_set(struct sk_buff *skb, struct dst_entry *dst)
738 {
739         skb->_skb_refdst = (unsigned long)dst;
740 }
741
742 /**
743  * skb_dst_set_noref - sets skb dst, hopefully, without taking reference
744  * @skb: buffer
745  * @dst: dst entry
746  *
747  * Sets skb dst, assuming a reference was not taken on dst.
748  * If dst entry is cached, we do not take reference and dst_release
749  * will be avoided by refdst_drop. If dst entry is not cached, we take
750  * reference, so that last dst_release can destroy the dst immediately.
751  */
752 static inline void skb_dst_set_noref(struct sk_buff *skb, struct dst_entry *dst)
753 {
754         WARN_ON(!rcu_read_lock_held() && !rcu_read_lock_bh_held());
755         skb->_skb_refdst = (unsigned long)dst | SKB_DST_NOREF;
756 }
757
758 /**
759  * skb_dst_is_noref - Test if skb dst isn't refcounted
760  * @skb: buffer
761  */
762 static inline bool skb_dst_is_noref(const struct sk_buff *skb)
763 {
764         return (skb->_skb_refdst & SKB_DST_NOREF) && skb_dst(skb);
765 }
766
767 static inline struct rtable *skb_rtable(const struct sk_buff *skb)
768 {
769         return (struct rtable *)skb_dst(skb);
770 }
```

```
771
772 void kfree_skb(struct sk_buff *skb);
773 void kfree_skb_list(struct sk_buff *segs);
774 void skb_tx_error(struct sk_buff *skb);
775 void consume_skb(struct sk_buff *skb);
776 void  __kfree_skb(struct sk_buff *skb);
777 extern struct kmem_cache *skbuff_head_cache;
778
779 void kfree_skb_partial(struct sk_buff *skb, bool head_stolen);
780 bool skb_try_coalesce(struct sk_buff *to, struct sk_buff *from,
781                       bool *fragstolen, int *delta_truesize);
782
783 struct sk_buff *__alloc_skb(unsigned int size, gfp_t priority, int flags,
784                             int node);
785 struct sk_buff *__build_skb(void *data, unsigned int frag_size);
786 struct sk_buff *build_skb(void *data, unsigned int frag_size);
787 static inline struct sk_buff *alloc_skb(unsigned int size,
788                                         gfp_t priority)
789 {
790         return __alloc_skb(size, priority, 0, NUMA_NO_NODE);
791 }
792
793 struct sk_buff *alloc_skb_with_frags(unsigned long header_len,
794                                      unsigned long data_len,
795                                      int max_page_order,
796                                      int *errcode,
797                                      gfp_t gfp_mask);
798
799 /* Layout of fast clones : [skb1][skb2][fclone_ref] */
800 struct sk_buff_fclones {
801         struct sk_buff  skb1;
802
803         struct sk_buff  skb2;
804
805         atomic_t        fclone_ref;
806 };
807
808 /**
809  *      skb_fclone_busy - check if fclone is busy
810  *      @skb: buffer
811  *
812  * Returns true is skb is a fast clone, and its clone is not freed.
813  * Some drivers call skb_orphan() in their ndo_start_xmit(),
814  * so we also check that this didnt happen.
815  */
816 static inline bool skb_fclone_busy(const struct sock *sk,
817                                    const struct sk_buff *skb)
818 {
819         const struct sk_buff_fclones *fclones;
820
821         fclones = container_of(skb, struct sk_buff_fclones, skb1);
822
823         return skb->fclone == SKB_FCLONE_ORIG &&
824                atomic_read(&fclones->fclone_ref) > 1 &&
825                fclones->skb2.sk == sk;
826 }
827
828 static inline struct sk_buff *alloc_skb_fclone(unsigned int size,
829                                                gfp_t priority)
830 {
831         return __alloc_skb(size, priority, SKB_ALLOC_FCLONE, NUMA_NO_NODE);
832 }
833
834 struct sk_buff *__alloc_skb_head(gfp_t priority, int node);
835 static inline struct sk_buff *alloc_skb_head(gfp_t priority)
836 {
```

```
837         return __alloc_skb_head(priority, -1);
838 }
839
840 struct sk_buff *skb_morph(struct sk_buff *dst, struct sk_buff *src);
841 int skb_copy_ubufs(struct sk_buff *skb, gfp_t gfp_mask);
842 struct sk_buff *skb_clone(struct sk_buff *skb, gfp_t priority);
843 struct sk_buff *skb_copy(const struct sk_buff *skb, gfp_t priority);
844 struct sk_buff *__pskb_copy_fclone(struct sk_buff *skb, int headroom,
845                                    gfp_t gfp_mask, bool fclone);
846 static inline struct sk_buff *__pskb_copy(struct sk_buff *skb, int headroom,
847                                           gfp_t gfp_mask)
848 {
849         return __pskb_copy_fclone(skb, headroom, gfp_mask, false);
850 }
851
852 int pskb_expand_head(struct sk_buff *skb, int nhead, int ntail, gfp_t gfp_mask);
853 struct sk_buff *skb_realloc_headroom(struct sk_buff *skb,
854                                      unsigned int headroom);
855 struct sk_buff *skb_copy_expand(const struct sk_buff *skb, int newheadroom,
856                                 int newtailroom, gfp_t priority);
857 int skb_to_sgvec_nomark(struct sk_buff *skb, struct scatterlist *sg,
858                         int offset, int len);
859 int skb_to_sgvec(struct sk_buff *skb, struct scatterlist *sg, int offset,
860                  int len);
861 int skb_cow_data(struct sk_buff *skb, int tailbits, struct sk_buff **trailer);
862 int skb_pad(struct sk_buff *skb, int pad);
863 #define dev_kfree_skb(a)        consume_skb(a)
864
865 int skb_append_datato_frags(struct sock *sk, struct sk_buff *skb,
866                             int getfrag(void *from, char *to, int offset,
867                                         int len, int odd, struct sk_buff *skb),
868                             void *from, int length);
869
870 int skb_append_pagefrags(struct sk_buff *skb, struct page *page,
871                          int offset, size_t size);
872
873 struct skb_seq_state {
874         __u32           lower_offset;
875         __u32           upper_offset;
876         __u32           frag_idx;
877         __u32           stepped_offset;
878         struct sk_buff  *root_skb;
879         struct sk_buff  *cur_skb;
880         __u8            *frag_data;
881 };
882
883 void skb_prepare_seq_read(struct sk_buff *skb, unsigned int from,
884                           unsigned int to, struct skb_seq_state *st);
885 unsigned int skb_seq_read(unsigned int consumed, const u8 **data,
886                           struct skb_seq_state *st);
887 void skb_abort_seq_read(struct skb_seq_state *st);
888
889 unsigned int skb_find_text(struct sk_buff *skb, unsigned int from,
890                            unsigned int to, struct ts_config *config);
891
892 /*
893  * Packet hash types specify the type of hash in skb_set_hash.
894  *
895  * Hash types refer to the protocol layer addresses which are used to
896  * construct a packet's hash. The hashes are used to differentiate or identify
897  * flows of the protocol layer for the hash type. Hash types are either
898  * layer-2 (L2), layer-3 (L3), or layer-4 (L4).
899  *
900  * Properties of hashes:
901  *
902  * 1) Two packets in different flows have different hash values
```

```
903  * 2) Two packets in the same flow should have the same hash value
904  *
905  * A hash at a higher layer is considered to be more specific. A driver should
906  * set the most specific hash possible.
907  *
908  * A driver cannot indicate a more specific hash than the layer at which a hash
909  * was computed. For instance an L3 hash cannot be set as an L4 hash.
910  *
911  * A driver may indicate a hash level which is less specific than the
912  * actual layer the hash was computed on. For instance, a hash computed
913  * at L4 may be considered an L3 hash. This should only be done if the
914  * driver can't unambiguously determine that the HW computed the hash at
915  * the higher layer. Note that the "should" in the second property above
916  * permits this.
917  */
918 enum pkt_hash_types {
919         PKT_HASH_TYPE_NONE,     /* Undefined type */
920         PKT_HASH_TYPE_L2,       /* Input: src_MAC, dest_MAC */
921         PKT_HASH_TYPE_L3,       /* Input: src_IP, dst_IP */
922         PKT_HASH_TYPE_L4,       /* Input: src_IP, dst_IP, src_port, dst_port */
923 };
924
925 static inline void
926 skb_set_hash(struct sk_buff *skb, __u32 hash, enum pkt_hash_types type)
927 {
928         skb->l4_hash = (type == PKT_HASH_TYPE_L4);
929         skb->sw_hash = 0;
930         skb->hash = hash;
931 }
932
933 static inline __u32 skb_get_hash(struct sk_buff *skb)
934 {
935         if (!skb->l4_hash && !skb->sw_hash)
936                 __skb_get_hash(skb);
937
938         return skb->hash;
939 }
940
941 __u32 skb_get_hash_perturb(const struct sk_buff *skb, u32 perturb);
942
943 static inline __u32 skb_get_hash_raw(const struct sk_buff *skb)
944 {
945         return skb->hash;
946 }
947
948 static inline void skb_clear_hash(struct sk_buff *skb)
949 {
950         skb->hash = 0;
951         skb->sw_hash = 0;
952         skb->l4_hash = 0;
953 }
954
955 static inline void skb_clear_hash_if_not_l4(struct sk_buff *skb)
956 {
957         if (!skb->l4_hash)
958                 skb_clear_hash(skb);
959 }
960
961 static inline void skb_copy_hash(struct sk_buff *to, const struct sk_buff *from)
962 {
963         to->hash = from->hash;
964         to->sw_hash = from->sw_hash;
965         to->l4_hash = from->l4_hash;
966 };
967
968 static inline void skb_sender_cpu_clear(struct sk_buff *skb)
```

```
969 {
970 #ifdef CONFIG_XPS
971         skb->sender_cpu = 0;
972 #endif
973 }
974
975 #ifdef NET_SKBUFF_DATA_USES_OFFSET
976 static inline unsigned char *skb_end_pointer(const struct sk_buff *skb)
977 {
978         return skb->head + skb->end;
979 }
980
981 static inline unsigned int skb_end_offset(const struct sk_buff *skb)
982 {
983         return skb->end;
984 }
985 #else
986 static inline unsigned char *skb_end_pointer(const struct sk_buff *skb)
987 {
988         return skb->end;
989 }
990
991 static inline unsigned int skb_end_offset(const struct sk_buff *skb)
992 {
993         return skb->end - skb->head;
994 }
995 #endif
996
997 /* Internal */
998 #define skb_shinfo(SKB) ((struct skb_shared_info *)(skb_end_pointer(SKB)))
999
1000 static inline struct skb_shared_hwtstamps *skb_hwtstamps(struct sk_buff *skb)
1001 {
1002         return &skb_shinfo(skb)->hwtstamps;
1003 }
1004
1005 /**
1006  *      skb_queue_empty - check if a queue is empty
1007  *      @list: queue head
1008  *
1009  *      Returns true if the queue is empty, false otherwise.
1010  */
1011 static inline int skb_queue_empty(const struct sk_buff_head *list)
1012 {
1013         return list->next == (const struct sk_buff *) list;
1014 }
1015
1016 /**
1017  *      skb_queue_is_last - check if skb is the last entry in the queue
1018  *      @list: queue head
1019  *      @skb: buffer
1020  *
1021  *      Returns true if @skb is the last buffer on the list.
1022  */
1023 static inline bool skb_queue_is_last(const struct sk_buff_head *list,
1024                                      const struct sk_buff *skb)
1025 {
1026         return skb->next == (const struct sk_buff *) list;
1027 }
1028
1029 /**
1030  *      skb_queue_is_first - check if skb is the first entry in the queue
1031  *      @list: queue head
1032  *      @skb: buffer
1033  *
1034  *      Returns true if @skb is the first buffer on the list.
```

```
1035  */
1036 static inline bool skb_queue_is_first(const struct sk_buff_head *list,
1037                                       const struct sk_buff *skb)
1038 {
1039         return skb->prev == (const struct sk_buff *) list;
1040 }
1041
1042 /**
1043  *      skb_queue_next - return the next packet in the queue
1044  *      @list: queue head
1045  *      @skb: current buffer
1046  *
1047  *      Return the next packet in @list after @skb.  It is only valid to
1048  *      call this if skb_queue_is_last() evaluates to false.
1049  */
1050 static inline struct sk_buff *skb_queue_next(const struct sk_buff_head *list,
1051                                              const struct sk_buff *skb)
1052 {
1053         /* This BUG_ON may seem severe, but if we just return then we
1054          * are going to dereference garbage.
1055          */
1056         BUG_ON(skb_queue_is_last(list, skb));
1057         return skb->next;
1058 }
1059
1060 /**
1061  *      skb_queue_prev - return the prev packet in the queue
1062  *      @list: queue head
1063  *      @skb: current buffer
1064  *
1065  *      Return the prev packet in @list before @skb.  It is only valid to
1066  *      call this if skb_queue_is_first() evaluates to false.
1067  */
1068 static inline struct sk_buff *skb_queue_prev(const struct sk_buff_head *list,
1069                                              const struct sk_buff *skb)
1070 {
1071         /* This BUG_ON may seem severe, but if we just return then we
1072          * are going to dereference garbage.
1073          */
1074         BUG_ON(skb_queue_is_first(list, skb));
1075         return skb->prev;
1076 }
1077
1078 /**
1079  *      skb_get - reference buffer
1080  *      @skb: buffer to reference
1081  *
1082  *      Makes another reference to a socket buffer and returns a pointer
1083  *      to the buffer.
1084  */
1085 static inline struct sk_buff *skb_get(struct sk_buff *skb)
1086 {
1087         atomic_inc(&skb->users);
1088         return skb;
1089 }
1090
1091 /*
1092  * If users == 1, we are the only owner and are can avoid redundant
1093  * atomic change.
1094  */
1095
1096 /**
1097  *      skb_cloned - is the buffer a clone
1098  *      @skb: buffer to check
1099  *
1100  *      Returns true if the buffer was generated with skb_clone() and is
```

```
1101  *      one of multiple shared copies of the buffer. Cloned buffers are
1102  *      shared data so must not be written to under normal circumstances.
1103  */
1104 static inline int skb_cloned(const struct sk_buff *skb)
1105 {
1106         return skb->cloned &&
1107                (atomic_read(&skb_shinfo(skb)->dataref) & SKB_DATAREF_MASK) != 1;
1108 }
1109
1110 static inline int skb_unclone(struct sk_buff *skb, gfp_t pri)
1111 {
1112         might_sleep_if(pri & __GFP_WAIT);
1113
1114         if (skb_cloned(skb))
1115                 return pskb_expand_head(skb, 0, 0, pri);
1116
1117         return 0;
1118 }
1119
1120 /**
1121  *      skb_header_cloned - is the header a clone
1122  *      @skb: buffer to check
1123  *
1124  *      Returns true if modifying the header part of the buffer requires
1125  *      the data to be copied.
1126  */
1127 static inline int skb_header_cloned(const struct sk_buff *skb)
1128 {
1129         int dataref;
1130
1131         if (!skb->cloned)
1132                 return 0;
1133
1134         dataref = atomic_read(&skb_shinfo(skb)->dataref);
1135         dataref = (dataref & SKB_DATAREF_MASK) - (dataref >> SKB_DATAREF_SHIFT);
1136         return dataref != 1;
1137 }
1138
1139 /**
1140  *      skb_header_release - release reference to header
1141  *      @skb: buffer to operate on
1142  *
1143  *      Drop a reference to the header part of the buffer.  This is done
1144  *      by acquiring a payload reference.  You must not read from the header
1145  *      part of skb->data after this.
1146  *      Note : Check if you can use __skb_header_release() instead.
1147  */
1148 static inline void skb_header_release(struct sk_buff *skb)
1149 {
1150         BUG_ON(skb->nohdr);
1151         skb->nohdr = 1;
1152         atomic_add(1 << SKB_DATAREF_SHIFT, &skb_shinfo(skb)->dataref);
1153 }
1154
1155 /**
1156  *      __skb_header_release - release reference to header
1157  *      @skb: buffer to operate on
1158  *
1159  *      Variant of skb_header_release() assuming skb is private to caller.
1160  *      We can avoid one atomic operation.
1161  */
1162 static inline void __skb_header_release(struct sk_buff *skb)
1163 {
1164         skb->nohdr = 1;
1165         atomic_set(&skb_shinfo(skb)->dataref, 1 + (1 << SKB_DATAREF_SHIFT));
1166 }
```

```
1167
1168
1169 /**
1170  *      skb_shared - is the buffer shared
1171  *      @skb: buffer to check
1172  *
1173  *      Returns true if more than one person has a reference to this
1174  *      buffer.
1175  */
1176 static inline int skb_shared(const struct sk_buff *skb)
1177 {
1178         return atomic_read(&skb->users) != 1;
1179 }
1180
1181 /**
1182  *      skb_share_check - check if buffer is shared and if so clone it
1183  *      @skb: buffer to check
1184  *      @pri: priority for memory allocation
1185  *
1186  *      If the buffer is shared the buffer is cloned and the old copy
1187  *      drops a reference. A new clone with a single reference is returned.
1188  *      If the buffer is not shared the original buffer is returned. When
1189  *      being called from interrupt status or with spinlocks held pri must
1190  *      be GFP_ATOMIC.
1191  *
1192  *      NULL is returned on a memory allocation failure.
1193  */
1194 static inline struct sk_buff *skb_share_check(struct sk_buff *skb, gfp_t pri)
1195 {
1196         might_sleep_if(pri & __GFP_WAIT);
1197         if (skb_shared(skb)) {
1198                 struct sk_buff *nskb = skb_clone(skb, pri);
1199
1200                 if (likely(nskb))
1201                         consume_skb(skb);
1202                 else
1203                         kfree_skb(skb);
1204                 skb = nskb;
1205         }
1206         return skb;
1207 }
1208
1209 /*
1210  *      Copy shared buffers into a new sk_buff. We effectively do COW on
1211  *      packets to handle cases where we have a local reader and forward
1212  *      and a couple of other messy ones. The normal one is tcpdumping
1213  *      a packet thats being forwarded.
1214  */
1215
1216 /**
1217  *      skb_unshare - make a copy of a shared buffer
1218  *      @skb: buffer to check
1219  *      @pri: priority for memory allocation
1220  *
1221  *      If the socket buffer is a clone then this function creates a new
1222  *      copy of the data, drops a reference count on the old copy and returns
1223  *      the new copy with the reference count at 1. If the buffer is not a clone
1224  *      the original buffer is returned. When called with a spinlock held or
1225  *      from interrupt state @pri must be %GFP_ATOMIC
1226  *
1227  *      %NULL is returned on a memory allocation failure.
1228  */
1229 static inline struct sk_buff *skb_unshare(struct sk_buff *skb,
1230                                           gfp_t pri)
1231 {
1232         might_sleep_if(pri & __GFP_WAIT);
```

```
1233         if (skb_cloned(skb)) {
1234                 struct sk_buff *nskb = skb_copy(skb, pri);
1235
1236                 /* Free our shared copy */
1237                 if (likely(nskb))
1238                         consume_skb(skb);
1239                 else
1240                         kfree_skb(skb);
1241                 skb = nskb;
1242         }
1243         return skb;
1244 }
1245
1246 /**
1247  *      skb_peek - peek at the head of an &sk_buff_head
1248  *      @list_: list to peek at
1249  *
1250  *      Peek an &sk_buff. Unlike most other operations you _MUST_
1251  *      be careful with this one. A peek leaves the buffer on the
1252  *      list and someone else may run off with it. You must hold
1253  *      the appropriate locks or have a private queue to do this.
1254  *
1255  *      Returns %NULL for an empty list or a pointer to the head element.
1256  *      The reference count is not incremented and the reference is therefore
1257  *      volatile. Use with caution.
1258  */
1259 static inline struct sk_buff *skb_peek(const struct sk_buff_head *list_)
1260 {
1261         struct sk_buff *skb = list_->next;
1262
1263         if (skb == (struct sk_buff *)list_)
1264                 skb = NULL;
1265         return skb;
1266 }
1267
1268 /**
1269  *      skb_peek_next - peek skb following the given one from a queue
1270  *      @skb: skb to start from
1271  *      @list_: list to peek at
1272  *
1273  *      Returns %NULL when the end of the list is met or a pointer to the
1274  *      next element. The reference count is not incremented and the
1275  *      reference is therefore volatile. Use with caution.
1276  */
1277 static inline struct sk_buff *skb_peek_next(struct sk_buff *skb,
1278                 const struct sk_buff_head *list_)
1279 {
1280         struct sk_buff *next = skb->next;
1281
1282         if (next == (struct sk_buff *)list_)
1283                 next = NULL;
1284         return next;
1285 }
1286
1287 /**
1288  *      skb_peek_tail - peek at the tail of an &sk_buff_head
1289  *      @list_: list to peek at
1290  *
1291  *      Peek an &sk_buff. Unlike most other operations you _MUST_
1292  *      be careful with this one. A peek leaves the buffer on the
1293  *      list and someone else may run off with it. You must hold
1294  *      the appropriate locks or have a private queue to do this.
1295  *
1296  *      Returns %NULL for an empty list or a pointer to the tail element.
1297  *      The reference count is not incremented and the reference is therefore
1298  *      volatile. Use with caution.
```

```
1299  */
1300 static inline struct sk_buff *skb_peek_tail(const struct sk_buff_head *list_)
1301 {
1302         struct sk_buff *skb = list_->prev;
1303
1304         if (skb == (struct sk_buff *)list_)
1305                 skb = NULL;
1306         return skb;
1307
1308 }
1309
1310 /**
1311  *      skb_queue_len   - get queue length
1312  *      @list_: list to measure
1313  *
1314  *      Return the length of an &sk_buff queue.
1315  */
1316 static inline __u32 skb_queue_len(const struct sk_buff_head *list_)
1317 {
1318         return list_->qlen;
1319 }
1320
1321 /**
1322  *      __skb_queue_head_init - initialize non-spinlock portions of sk_buff_head
1323  *      @list: queue to initialize
1324  *
1325  *      This initializes only the list and queue length aspects of
1326  *      an sk_buff_head object.  This allows to initialize the list
1327  *      aspects of an sk_buff_head without reinitializing things like
1328  *      the spinlock.  It can also be used for on-stack sk_buff_head
1329  *      objects where the spinlock is known to not be used.
1330  */
1331 static inline void __skb_queue_head_init(struct sk_buff_head *list)
1332 {
1333         list->prev = list->next = (struct sk_buff *)list;
1334         list->qlen = 0;
1335 }
1336
1337 /*
1338  * This function creates a split out lock class for each invocation;
1339  * this is needed for now since a whole lot of users of the skb-queue
1340  * infrastructure in drivers have different locking usage (in hardirq)
1341  * than the networking core (in softirq only). In the long run either the
1342  * network layer or drivers should need annotation to consolidate the
1343  * main types of usage into 3 classes.
1344  */
1345 static inline void skb_queue_head_init(struct sk_buff_head *list)
1346 {
1347         spin_lock_init(&list->lock);
1348         __skb_queue_head_init(list);
1349 }
1350
1351 static inline void skb_queue_head_init_class(struct sk_buff_head *list,
1352                 struct lock_class_key *class)
1353 {
1354         skb_queue_head_init(list);
1355         lockdep_set_class(&list->lock, class);
1356 }
1357
1358 /*
1359  *      Insert an sk_buff on a list.
1360  *
1361  *      The "__skb_xxxx()" functions are the non-atomic ones that
1362  *      can only be called with interrupts disabled.
1363  */
1364 void skb_insert(struct sk_buff *old, struct sk_buff *newsk,
```

```
1365                 struct sk_buff_head *list);
1366 static inline void __skb_insert(struct sk_buff *newsk,
1367                                 struct sk_buff *prev, struct sk_buff *next,
1368                                 struct sk_buff_head *list)
1369 {
1370         newsk->next = next;
1371         newsk->prev = prev;
1372         next->prev  = prev->next = newsk;
1373         list->qlen++;
1374 }
1375
1376 static inline void __skb_queue_splice(const struct sk_buff_head *list,
1377                                       struct sk_buff *prev,
1378                                       struct sk_buff *next)
1379 {
1380         struct sk_buff *first = list->next;
1381         struct sk_buff *last = list->prev;
1382
1383         first->prev = prev;
1384         prev->next = first;
1385
1386         last->next = next;
1387         next->prev = last;
1388 }
1389
1390 /**
1391  *      skb_queue_splice - join two skb lists, this is designed for stacks
1392  *      @list: the new list to add
1393  *      @head: the place to add it in the first list
1394  */
1395 static inline void skb_queue_splice(const struct sk_buff_head *list,
1396                                     struct sk_buff_head *head)
1397 {
1398         if (!skb_queue_empty(list)) {
1399                 __skb_queue_splice(list, (struct sk_buff *) head, head->next);
1400                 head->qlen += list->qlen;
1401         }
1402 }
1403
1404 /**
1405  *      skb_queue_splice_init - join two skb lists and reinitialise the emptied list
1406  *      @list: the new list to add
1407  *      @head: the place to add it in the first list
1408  *
1409  *      The list at @list is reinitialised
1410  */
1411 static inline void skb_queue_splice_init(struct sk_buff_head *list,
1412                                          struct sk_buff_head *head)
1413 {
1414         if (!skb_queue_empty(list)) {
1415                 __skb_queue_splice(list, (struct sk_buff *) head, head->next);
1416                 head->qlen += list->qlen;
1417                 __skb_queue_head_init(list);
1418         }
1419 }
1420
1421 /**
1422  *      skb_queue_splice_tail - join two skb lists, each list being a queue
1423  *      @list: the new list to add
1424  *      @head: the place to add it in the first list
1425  */
1426 static inline void skb_queue_splice_tail(const struct sk_buff_head *list,
1427                                          struct sk_buff_head *head)
1428 {
1429         if (!skb_queue_empty(list)) {
1430                 __skb_queue_splice(list, head->prev, (struct sk_buff *) head);
```

```
1431                 head->qlen += list->qlen;
1432         }
1433 }
1434
1435 /**
1436  *      skb_queue_splice_tail_init - join two skb lists and reinitialise the emptied list
1437  *      @list: the new list to add
1438  *      @head: the place to add it in the first list
1439  *
1440  *      Each of the lists is a queue.
1441  *      The list at @list is reinitialised
1442  */
1443 static inline void skb_queue_splice_tail_init(struct sk_buff_head *list,
1444                                               struct sk_buff_head *head)
1445 {
1446         if (!skb_queue_empty(list)) {
1447                 __skb_queue_splice(list, head->prev, (struct sk_buff *) head);
1448                 head->qlen += list->qlen;
1449                 __skb_queue_head_init(list);
1450         }
1451 }
1452
1453 /**
1454  *      __skb_queue_after - queue a buffer at the list head
1455  *      @list: list to use
1456  *      @prev: place after this buffer
1457  *      @newsk: buffer to queue
1458  *
1459  *      Queue a buffer int the middle of a list. This function takes no locks
1460  *      and you must therefore hold required locks before calling it.
1461  *
1462  *      A buffer cannot be placed on two lists at the same time.
1463  */
1464 static inline void __skb_queue_after(struct sk_buff_head *list,
1465                                      struct sk_buff *prev,
1466                                      struct sk_buff *newsk)
1467 {
1468         __skb_insert(newsk, prev, prev->next, list);
1469 }
1470
1471 void skb_append(struct sk_buff *old, struct sk_buff *newsk,
1472                 struct sk_buff_head *list);
1473
1474 static inline void __skb_queue_before(struct sk_buff_head *list,
1475                                       struct sk_buff *next,
1476                                       struct sk_buff *newsk)
1477 {
1478         __skb_insert(newsk, next->prev, next, list);
1479 }
1480
1481 /**
1482  *      __skb_queue_head - queue a buffer at the list head
1483  *      @list: list to use
1484  *      @newsk: buffer to queue
1485  *
1486  *      Queue a buffer at the start of a list. This function takes no locks
1487  *      and you must therefore hold required locks before calling it.
1488  *
1489  *      A buffer cannot be placed on two lists at the same time.
1490  */
1491 void skb_queue_head(struct sk_buff_head *list, struct sk_buff *newsk);
1492 static inline void __skb_queue_head(struct sk_buff_head *list,
1493                                     struct sk_buff *newsk)
1494 {
1495         __skb_queue_after(list, (struct sk_buff *)list, newsk);
1496 }
```

```
1497
1498 /**
1499  *      __skb_queue_tail - queue a buffer at the list tail
1500  *      @list: list to use
1501  *      @newsk: buffer to queue
1502  *
1503  *      Queue a buffer at the end of a list. This function takes no locks
1504  *      and you must therefore hold required locks before calling it.
1505  *
1506  *      A buffer cannot be placed on two lists at the same time.
1507  */
1508 void skb_queue_tail(struct sk_buff_head *list, struct sk_buff *newsk);
1509 static inline void __skb_queue_tail(struct sk_buff_head *list,
1510                                     struct sk_buff *newsk)
1511 {
1512         __skb_queue_before(list, (struct sk_buff *)list, newsk);
1513 }
1514
1515 /*
1516  * remove sk_buff from list. _Must_ be called atomically, and with
1517  * the list known..
1518  */
1519 void skb_unlink(struct sk_buff *skb, struct sk_buff_head *list);
1520 static inline void __skb_unlink(struct sk_buff *skb, struct sk_buff_head *list)
1521 {
1522         struct sk_buff *next, *prev;
1523
1524         list->qlen--;
1525         next       = skb->next;
1526         prev       = skb->prev;
1527         skb->next  = skb->prev = NULL;
1528         next->prev = prev;
1529         prev->next = next;
1530 }
1531
1532 /**
1533  *      __skb_dequeue - remove from the head of the queue
1534  *      @list: list to dequeue from
1535  *
1536  *      Remove the head of the list. This function does not take any locks
1537  *      so must be used with appropriate locks held only. The head item is
1538  *      returned or %NULL if the list is empty.
1539  */
1540 struct sk_buff *skb_dequeue(struct sk_buff_head *list);
1541 static inline struct sk_buff *__skb_dequeue(struct sk_buff_head *list)
1542 {
1543         struct sk_buff *skb = skb_peek(list);
1544         if (skb)
1545                 __skb_unlink(skb, list);
1546         return skb;
1547 }
1548
1549 /**
1550  *      __skb_dequeue_tail - remove from the tail of the queue
1551  *      @list: list to dequeue from
1552  *
1553  *      Remove the tail of the list. This function does not take any locks
1554  *      so must be used with appropriate locks held only. The tail item is
1555  *      returned or %NULL if the list is empty.
1556  */
1557 struct sk_buff *skb_dequeue_tail(struct sk_buff_head *list);
1558 static inline struct sk_buff *__skb_dequeue_tail(struct sk_buff_head *list)
1559 {
1560         struct sk_buff *skb = skb_peek_tail(list);
1561         if (skb)
1562                 __skb_unlink(skb, list);
```

```
1563         return skb;
1564 }
1565
1566
1567 static inline bool skb_is_nonlinear(const struct sk_buff *skb)
1568 {
1569         return skb->data_len;
1570 }
1571
1572 static inline unsigned int skb_headlen(const struct sk_buff *skb)
1573 {
1574         return skb->len - skb->data_len;
1575 }
1576
1577 static inline int skb_pagelen(const struct sk_buff *skb)
1578 {
1579         int i, len = 0;
1580
1581         for (i = (int)skb_shinfo(skb)->nr_frags - 1; i >= 0; i--)
1582                 len += skb_frag_size(&skb_shinfo(skb)->frags[i]);
1583         return len + skb_headlen(skb);
1584 }
1585
1586 /**
1587  * __skb_fill_page_desc - initialise a paged fragment in an skb
1588  * @skb: buffer containing fragment to be initialised
1589  * @i: paged fragment index to initialise
1590  * @page: the page to use for this fragment
1591  * @off: the offset to the data with @page
1592  * @size: the length of the data
1593  *
1594  * Initialises the @i'th fragment of @skb to point to &size bytes at
1595  * offset @off within @page.
1596  *
1597  * Does not take any additional reference on the fragment.
1598  */
1599 static inline void __skb_fill_page_desc(struct sk_buff *skb, int i,
1600                                         struct page *page, int off, int size)
1601 {
1602         skb_frag_t *frag = &skb_shinfo(skb)->frags[i];
1603
1604         /*
1605          * Propagate page pfmemalloc to the skb if we can. The problem is
1606          * that not all callers have unique ownership of the page but rely
1607          * on page_is_pfmemalloc doing the right thing(tm).
1608          */
1609         frag->page.p              = page;
1610         frag->page_offset         = off;
1611         skb_frag_size_set(frag, size);
1612
1613         page = compound_head(page);
1614         if (page_is_pfmemalloc(page))
1615                 skb->pfmemalloc = true;
1616 }
1617
1618 /**
1619  * skb_fill_page_desc - initialise a paged fragment in an skb
1620  * @skb: buffer containing fragment to be initialised
1621  * @i: paged fragment index to initialise
1622  * @page: the page to use for this fragment
1623  * @off: the offset to the data with @page
1624  * @size: the length of the data
1625  *
1626  * As per __skb_fill_page_desc() -- initialises the @i'th fragment of
1627  * @skb to point to @size bytes at offset @off within @page. In
1628  * addition updates @skb such that @i is the last fragment.
```

```
1629  *
1630  * Does not take any additional reference on the fragment.
1631  */
1632 static inline void skb_fill_page_desc(struct sk_buff *skb, int i,
1633                                       struct page *page, int off, int size)
1634 {
1635         __skb_fill_page_desc(skb, i, page, off, size);
1636         skb_shinfo(skb)->nr_frags = i + 1;
1637 }
1638
1639 void skb_add_rx_frag(struct sk_buff *skb, int i, struct page *page, int off,
1640                      int size, unsigned int truesize);
1641
1642 void skb_coalesce_rx_frag(struct sk_buff *skb, int i, int size,
1643                           unsigned int truesize);
1644
1645 #define SKB_PAGE_ASSERT(skb)    BUG_ON(skb_shinfo(skb)->nr_frags)
1646 #define SKB_FRAG_ASSERT(skb)    BUG_ON(skb_has_frag_list(skb))
1647 #define SKB_LINEAR_ASSERT(skb)  BUG_ON(skb_is_nonlinear(skb))
1648
1649 #ifdef NET_SKBUFF_DATA_USES_OFFSET
1650 static inline unsigned char *skb_tail_pointer(const struct sk_buff *skb)
1651 {
1652         return skb->head + skb->tail;
1653 }
1654
1655 static inline void skb_reset_tail_pointer(struct sk_buff *skb)
1656 {
1657         skb->tail = skb->data - skb->head;
1658 }
1659
1660 static inline void skb_set_tail_pointer(struct sk_buff *skb, const int offset)
1661 {
1662         skb_reset_tail_pointer(skb);
1663         skb->tail += offset;
1664 }
1665
1666 #else /* NET_SKBUFF_DATA_USES_OFFSET */
1667 static inline unsigned char *skb_tail_pointer(const struct sk_buff *skb)
1668 {
1669         return skb->tail;
1670 }
1671
1672 static inline void skb_reset_tail_pointer(struct sk_buff *skb)
1673 {
1674         skb->tail = skb->data;
1675 }
1676
1677 static inline void skb_set_tail_pointer(struct sk_buff *skb, const int offset)
1678 {
1679         skb->tail = skb->data + offset;
1680 }
1681
1682 #endif /* NET_SKBUFF_DATA_USES_OFFSET */
1683
1684 /*
1685  *      Add data to an sk_buff
1686  */
1687 unsigned char *pskb_put(struct sk_buff *skb, struct sk_buff *tail, int len);
1688 unsigned char *skb_put(struct sk_buff *skb, unsigned int len);
1689 static inline unsigned char *__skb_put(struct sk_buff *skb, unsigned int len)
1690 {
1691         unsigned char *tmp = skb_tail_pointer(skb);
1692         SKB_LINEAR_ASSERT(skb);
1693         skb->tail += len;
1694         skb->len  += len;
```

```
1695         return tmp;
1696 }
1697
1698 unsigned char *skb_push(struct sk_buff *skb, unsigned int len);
1699 static inline unsigned char *__skb_push(struct sk_buff *skb, unsigned int len)
1700 {
1701         skb->data -= len;
1702         skb->len  += len;
1703         return skb->data;
1704 }
1705
1706 unsigned char *skb_pull(struct sk_buff *skb, unsigned int len);
1707 static inline unsigned char *__skb_pull(struct sk_buff *skb, unsigned int len)
1708 {
1709         skb->len -= len;
1710         BUG_ON(skb->len < skb->data_len);
1711         return skb->data += len;
1712 }
1713
1714 static inline unsigned char *skb_pull_inline(struct sk_buff *skb, unsigned int len)
1715 {
1716         return unlikely(len > skb->len) ? NULL : __skb_pull(skb, len);
1717 }
1718
1719 unsigned char *__pskb_pull_tail(struct sk_buff *skb, int delta);
1720
1721 static inline unsigned char *__pskb_pull(struct sk_buff *skb, unsigned int len)
1722 {
1723         if (len > skb_headlen(skb) &&
1724             !__pskb_pull_tail(skb, len - skb_headlen(skb)))
1725                 return NULL;
1726         skb->len -= len;
1727         return skb->data += len;
1728 }
1729
1730 static inline unsigned char *pskb_pull(struct sk_buff *skb, unsigned int len)
1731 {
1732         return unlikely(len > skb->len) ? NULL : __pskb_pull(skb, len);
1733 }
1734
1735 static inline int pskb_may_pull(struct sk_buff *skb, unsigned int len)
1736 {
1737         if (likely(len <= skb_headlen(skb)))
1738                 return 1;
1739         if (unlikely(len > skb->len))
1740                 return 0;
1741         return __pskb_pull_tail(skb, len - skb_headlen(skb)) != NULL;
1742 }
1743
1744 /**
1745  *      skb_headroom - bytes at buffer head
1746  *      @skb: buffer to check
1747  *
1748  *      Return the number of bytes of free space at the head of an &sk_buff.
1749  */
1750 static inline unsigned int skb_headroom(const struct sk_buff *skb)
1751 {
1752         return skb->data - skb->head;
1753 }
1754
1755 /**
1756  *      skb_tailroom - bytes at buffer end
1757  *      @skb: buffer to check
1758  *
1759  *      Return the number of bytes of free space at the tail of an sk_buff
1760  */
```

```
1761 static inline int skb_tailroom(const struct sk_buff *skb)
1762 {
1763         return skb_is_nonlinear(skb) ? 0 : skb->end - skb->tail;
1764 }
1765
1766 /**
1767  *      skb_availroom - bytes at buffer end
1768  *      @skb: buffer to check
1769  *
1770  *      Return the number of bytes of free space at the tail of an sk_buff
1771  *      allocated by sk_stream_alloc()
1772  */
1773 static inline int skb_availroom(const struct sk_buff *skb)
1774 {
1775         if (skb_is_nonlinear(skb))
1776                 return 0;
1777
1778         return skb->end - skb->tail - skb->reserved_tailroom;
1779 }
1780
1781 /**
1782  *      skb_reserve - adjust headroom
1783  *      @skb: buffer to alter
1784  *      @len: bytes to move
1785  *
1786  *      Increase the headroom of an empty &sk_buff by reducing the tail
1787  *      room. This is only allowed for an empty buffer.
1788  */
1789 static inline void skb_reserve(struct sk_buff *skb, int len)
1790 {
1791         skb->data += len;
1792         skb->tail += len;
1793 }
1794
1795 #define ENCAP_TYPE_ETHER        0
1796 #define ENCAP_TYPE_IPPROTO      1
1797
1798 static inline void skb_set_inner_protocol(struct sk_buff *skb,
1799                                           __be16 protocol)
1800 {
1801         skb->inner_protocol = protocol;
1802         skb->inner_protocol_type = ENCAP_TYPE_ETHER;
1803 }
1804
1805 static inline void skb_set_inner_ipproto(struct sk_buff *skb,
1806                                          __u8 ipproto)
1807 {
1808         skb->inner_ipproto = ipproto;
1809         skb->inner_protocol_type = ENCAP_TYPE_IPPROTO;
1810 }
1811
1812 static inline void skb_reset_inner_headers(struct sk_buff *skb)
1813 {
1814         skb->inner_mac_header = skb->mac_header;
1815         skb->inner_network_header = skb->network_header;
1816         skb->inner_transport_header = skb->transport_header;
1817 }
1818
1819 static inline void skb_reset_mac_len(struct sk_buff *skb)
1820 {
1821         skb->mac_len = skb->network_header - skb->mac_header;
1822 }
1823
1824 static inline unsigned char *skb_inner_transport_header(const struct sk_buff
1825                                                         *skb)
1826 {
```

```
1827         return skb->head + skb->inner_transport_header;
1828 }
1829
1830 static inline void skb_reset_inner_transport_header(struct sk_buff *skb)
1831 {
1832         skb->inner_transport_header = skb->data - skb->head;
1833 }
1834
1835 static inline void skb_set_inner_transport_header(struct sk_buff *skb,
1836                                                   const int offset)
1837 {
1838         skb_reset_inner_transport_header(skb);
1839         skb->inner_transport_header += offset;
1840 }
1841
1842 static inline unsigned char *skb_inner_network_header(const struct sk_buff *skb)
1843 {
1844         return skb->head + skb->inner_network_header;
1845 }
1846
1847 static inline void skb_reset_inner_network_header(struct sk_buff *skb)
1848 {
1849         skb->inner_network_header = skb->data - skb->head;
1850 }
1851
1852 static inline void skb_set_inner_network_header(struct sk_buff *skb,
1853                                                 const int offset)
1854 {
1855         skb_reset_inner_network_header(skb);
1856         skb->inner_network_header += offset;
1857 }
1858
1859 static inline unsigned char *skb_inner_mac_header(const struct sk_buff *skb)
1860 {
1861         return skb->head + skb->inner_mac_header;
1862 }
1863
1864 static inline void skb_reset_inner_mac_header(struct sk_buff *skb)
1865 {
1866         skb->inner_mac_header = skb->data - skb->head;
1867 }
1868
1869 static inline void skb_set_inner_mac_header(struct sk_buff *skb,
1870                                             const int offset)
1871 {
1872         skb_reset_inner_mac_header(skb);
1873         skb->inner_mac_header += offset;
1874 }
1875 static inline bool skb_transport_header_was_set(const struct sk_buff *skb)
1876 {
1877         return skb->transport_header != (typeof(skb->transport_header))~0U;
1878 }
1879
1880 static inline unsigned char *skb_transport_header(const struct sk_buff *skb)
1881 {
1882         return skb->head + skb->transport_header;
1883 }
1884
1885 static inline void skb_reset_transport_header(struct sk_buff *skb)
1886 {
1887         skb->transport_header = skb->data - skb->head;
1888 }
1889
1890 static inline void skb_set_transport_header(struct sk_buff *skb,
1891                                             const int offset)
1892 {
```

```
1893         skb_reset_transport_header(skb);
1894         skb->transport_header += offset;
1895 }
1896
1897 static inline unsigned char *skb_network_header(const struct sk_buff *skb)
1898 {
1899         return skb->head + skb->network_header;
1900 }
1901
1902 static inline void skb_reset_network_header(struct sk_buff *skb)
1903 {
1904         skb->network_header = skb->data - skb->head;
1905 }
1906
1907 static inline void skb_set_network_header(struct sk_buff *skb, const int offset)
1908 {
1909         skb_reset_network_header(skb);
1910         skb->network_header += offset;
1911 }
1912
1913 static inline unsigned char *skb_mac_header(const struct sk_buff *skb)
1914 {
1915         return skb->head + skb->mac_header;
1916 }
1917
1918 static inline int skb_mac_header_was_set(const struct sk_buff *skb)
1919 {
1920         return skb->mac_header != (typeof(skb->mac_header))~0U;
1921 }
1922
1923 static inline void skb_reset_mac_header(struct sk_buff *skb)
1924 {
1925         skb->mac_header = skb->data - skb->head;
1926 }
1927
1928 static inline void skb_set_mac_header(struct sk_buff *skb, const int offset)
1929 {
1930         skb_reset_mac_header(skb);
1931         skb->mac_header += offset;
1932 }
1933
1934 static inline void skb_pop_mac_header(struct sk_buff *skb)
1935 {
1936         skb->mac_header = skb->network_header;
1937 }
1938
1939 static inline void skb_probe_transport_header(struct sk_buff *skb,
1940                                               const int offset_hint)
1941 {
1942         struct flow_keys keys;
1943
1944         if (skb_transport_header_was_set(skb))
1945                 return;
1946         else if (skb_flow_dissect_flow_keys(skb, &keys))
1947                 skb_set_transport_header(skb, keys.control.thoff);
1948         else
1949                 skb_set_transport_header(skb, offset_hint);
1950 }
1951
1952 static inline void skb_mac_header_rebuild(struct sk_buff *skb)
1953 {
1954         if (skb_mac_header_was_set(skb)) {
1955                 const unsigned char *old_mac = skb_mac_header(skb);
1956
1957                 skb_set_mac_header(skb, -skb->mac_len);
1958                 memmove(skb_mac_header(skb), old_mac, skb->mac_len);
```

```
1959         }
1960 }
1961
1962 static inline int skb_checksum_start_offset(const struct sk_buff *skb)
1963 {
1964         return skb->csum_start - skb_headroom(skb);
1965 }
1966
1967 static inline int skb_transport_offset(const struct sk_buff *skb)
1968 {
1969         return skb_transport_header(skb) - skb->data;
1970 }
1971
1972 static inline u32 skb_network_header_len(const struct sk_buff *skb)
1973 {
1974         return skb->transport_header - skb->network_header;
1975 }
1976
1977 static inline u32 skb_inner_network_header_len(const struct sk_buff *skb)
1978 {
1979         return skb->inner_transport_header - skb->inner_network_header;
1980 }
1981
1982 static inline int skb_network_offset(const struct sk_buff *skb)
1983 {
1984         return skb_network_header(skb) - skb->data;
1985 }
1986
1987 static inline int skb_inner_network_offset(const struct sk_buff *skb)
1988 {
1989         return skb_inner_network_header(skb) - skb->data;
1990 }
1991
1992 static inline int pskb_network_may_pull(struct sk_buff *skb, unsigned int len)
1993 {
1994         return pskb_may_pull(skb, skb_network_offset(skb) + len);
1995 }
1996
1997 /*
1998  * CPUs often take a performance hit when accessing unaligned memory
1999  * locations. The actual performance hit varies, it can be small if the
2000  * hardware handles it or large if we have to take an exception and fix it
2001  * in software.
2002  *
2003  * Since an ethernet header is 14 bytes network drivers often end up with
2004  * the IP header at an unaligned offset. The IP header can be aligned by
2005  * shifting the start of the packet by 2 bytes. Drivers should do this
2006  * with:
2007  *
2008  * skb_reserve(skb, NET_IP_ALIGN);
2009  *
2010  * The downside to this alignment of the IP header is that the DMA is now
2011  * unaligned. On some architectures the cost of an unaligned DMA is high
2012  * and this cost outweighs the gains made by aligning the IP header.
2013  *
2014  * Since this trade off varies between architectures, we allow NET_IP_ALIGN
2015  * to be overridden.
2016  */
2017 #ifndef NET_IP_ALIGN
2018 #define NET_IP_ALIGN    2
2019 #endif
2020
2021 /*
2022  * The networking layer reserves some headroom in skb data (via
2023  * dev_alloc_skb). This is used to avoid having to reallocate skb data when
2024  * the header has to grow. In the default case, if the header has to grow
```

```
2025  * 32 bytes or less we avoid the reallocation.
2026  *
2027  * Unfortunately this headroom changes the DMA alignment of the resulting
2028  * network packet. As for NET_IP_ALIGN, this unaligned DMA is expensive
2029  * on some architectures. An architecture can override this value,
2030  * perhaps setting it to a cacheline in size (since that will maintain
2031  * cacheline alignment of the DMA). It must be a power of 2.
2032  *
2033  * Various parts of the networking layer expect at least 32 bytes of
2034  * headroom, you should not reduce this.
2035  *
2036  * Using max(32, L1_CACHE_BYTES) makes sense (especially with RPS)
2037  * to reduce average number of cache lines per packet.
2038  * get_rps_cpus() for example only access one 64 bytes aligned block :
2039  * NET_IP_ALIGN(2) + ethernet_header(14) + IP_header(20/40) + ports(8)
2040  */
2041 #ifndef NET_SKB_PAD
2042 #define NET_SKB_PAD     max(32, L1_CACHE_BYTES)
2043 #endif
2044
2045 int ___pskb_trim(struct sk_buff *skb, unsigned int len);
2046
2047 static inline void __skb_trim(struct sk_buff *skb, unsigned int len)
2048 {
2049         if (unlikely(skb_is_nonlinear(skb))) {
2050                 WARN_ON(1);
2051                 return;
2052         }
2053         skb->len = len;
2054         skb_set_tail_pointer(skb, len);
2055 }
2056
2057 void skb_trim(struct sk_buff *skb, unsigned int len);
2058
2059 static inline int __pskb_trim(struct sk_buff *skb, unsigned int len)
2060 {
2061         if (skb->data_len)
2062                 return ___pskb_trim(skb, len);
2063         __skb_trim(skb, len);
2064         return 0;
2065 }
2066
2067 static inline int pskb_trim(struct sk_buff *skb, unsigned int len)
2068 {
2069         return (len < skb->len) ? __pskb_trim(skb, len) : 0;
2070 }
2071
2072 /**
2073  *      pskb_trim_unique - remove end from a paged unique (not cloned) buffer
2074  *      @skb: buffer to alter
2075  *      @len: new length
2076  *
2077  *      This is identical to pskb_trim except that the caller knows that
2078  *      the skb is not cloned so we should never get an error due to out-
2079  *      of-memory.
2080  */
2081 static inline void pskb_trim_unique(struct sk_buff *skb, unsigned int len)
2082 {
2083         int err = pskb_trim(skb, len);
2084         BUG_ON(err);
2085 }
2086
2087 /**
2088  *      skb_orphan - orphan a buffer
2089  *      @skb: buffer to orphan
2090  *
```

```
2091  *      If a buffer currently has an owner then we call the owner's
2092  *      destructor function and make the @skb unowned. The buffer continues
2093  *      to exist but is no longer charged to its former owner.
2094  */
2095 static inline void skb_orphan(struct sk_buff *skb)
2096 {
2097         if (skb->destructor) {
2098                 skb->destructor(skb);
2099                 skb->destructor = NULL;
2100                 skb->sk         = NULL;
2101         } else {
2102                 BUG_ON(skb->sk);
2103         }
2104 }
2105
2106 /**
2107  *      skb_orphan_frags - orphan the frags contained in a buffer
2108  *      @skb: buffer to orphan frags from
2109  *      @gfp_mask: allocation mask for replacement pages
2110  *
2111  *      For each frag in the SKB which needs a destructor (i.e. has an
2112  *      owner) create a copy of that frag and release the original
2113  *      page by calling the destructor.
2114  */
2115 static inline int skb_orphan_frags(struct sk_buff *skb, gfp_t gfp_mask)
2116 {
2117         if (likely(!(skb_shinfo(skb)->tx_flags & SKBTX_DEV_ZEROCOPY)))
2118                 return 0;
2119         return skb_copy_ubufs(skb, gfp_mask);
2120 }
2121
2122 /**
2123  *      __skb_queue_purge - empty a list
2124  *      @list: list to empty
2125  *
2126  *      Delete all buffers on an &sk_buff list. Each buffer is removed from
2127  *      the list and one reference dropped. This function does not take the
2128  *      list lock and the caller must hold the relevant locks to use it.
2129  */
2130 void skb_queue_purge(struct sk_buff_head *list);
2131 static inline void __skb_queue_purge(struct sk_buff_head *list)
2132 {
2133         struct sk_buff *skb;
2134         while ((skb = __skb_dequeue(list)) != NULL)
2135                 kfree_skb(skb);
2136 }
2137
2138 void *netdev_alloc_frag(unsigned int fragsz);
2139
2140 struct sk_buff *__netdev_alloc_skb(struct net_device *dev, unsigned int length,
2141                                    gfp_t gfp_mask);
2142
2143 /**
2144  *      netdev_alloc_skb - allocate an skbuff for rx on a specific device
2145  *      @dev: network device to receive on
2146  *      @length: length to allocate
2147  *
2148  *      Allocate a new &sk_buff and assign it a usage count of one. The
2149  *      buffer has unspecified headroom built in. Users should allocate
2150  *      the headroom they think they need without accounting for the
2151  *      built in space. The built in space is used for optimisations.
2152  *
2153  *      %NULL is returned if there is no free memory. Although this function
2154  *      allocates memory it can be called from an interrupt.
2155  */
2156 static inline struct sk_buff *netdev_alloc_skb(struct net_device *dev,
```

```
2157                                                unsigned int length)
2158 {
2159         return __netdev_alloc_skb(dev, length, GFP_ATOMIC);
2160 }
2161
2162 /* legacy helper around __netdev_alloc_skb() */
2163 static inline struct sk_buff *__dev_alloc_skb(unsigned int length,
2164                                               gfp_t gfp_mask)
2165 {
2166         return __netdev_alloc_skb(NULL, length, gfp_mask);
2167 }
2168
2169 /* legacy helper around netdev_alloc_skb() */
2170 static inline struct sk_buff *dev_alloc_skb(unsigned int length)
2171 {
2172         return netdev_alloc_skb(NULL, length);
2173 }
2174
2175
2176 static inline struct sk_buff *__netdev_alloc_skb_ip_align(struct net_device *dev,
2177                 unsigned int length, gfp_t gfp)
2178 {
2179         struct sk_buff *skb = __netdev_alloc_skb(dev, length + NET_IP_ALIGN, gfp);
2180
2181         if (NET_IP_ALIGN && skb)
2182                 skb_reserve(skb, NET_IP_ALIGN);
2183         return skb;
2184 }
2185
2186 static inline struct sk_buff *netdev_alloc_skb_ip_align(struct net_device *dev,
2187                 unsigned int length)
2188 {
2189         return __netdev_alloc_skb_ip_align(dev, length, GFP_ATOMIC);
2190 }
2191
2192 static inline void skb_free_frag(void *addr)
2193 {
2194         __free_page_frag(addr);
2195 }
2196
2197 void *napi_alloc_frag(unsigned int fragsz);
2198 struct sk_buff *__napi_alloc_skb(struct napi_struct *napi,
2199                                  unsigned int length, gfp_t gfp_mask);
2200 static inline struct sk_buff *napi_alloc_skb(struct napi_struct *napi,
2201                                              unsigned int length)
2202 {
2203         return __napi_alloc_skb(napi, length, GFP_ATOMIC);
2204 }
2205
2206 /**
2207  * __dev_alloc_pages - allocate page for network Rx
2208  * @gfp_mask: allocation priority. Set __GFP_NOMEMALLOC if not for network Rx
2209  * @order: size of the allocation
2210  *
2211  * Allocate a new page.
2212  *
2213  * %NULL is returned if there is no free memory.
2214 */
2215 static inline struct page *__dev_alloc_pages(gfp_t gfp_mask,
2216                                              unsigned int order)
2217 {
2218         /* This piece of code contains several assumptions.
2219          * 1.  This is for device Rx, therefor a cold page is preferred.
2220          * 2.  The expectation is the user wants a compound page.
2221          * 3.  If requesting a order 0 page it will not be compound
2222          *     due to the check to see if order has a value in prep_new_page
```

```
2223          * 4.  __GFP_MEMALLOC is ignored if __GFP_NOMEMALLOC is set due to
2224          *      code in gfp_to_alloc_flags that should be enforcing this.
2225          */
2226         gfp_mask |= __GFP_COLD | __GFP_COMP | __GFP_MEMALLOC;
2227
2228         return alloc_pages_node(NUMA_NO_NODE, gfp_mask, order);
2229 }
2230
2231 static inline struct page *dev_alloc_pages(unsigned int order)
2232 {
2233         return __dev_alloc_pages(GFP_ATOMIC, order);
2234 }
2235
2236 /**
2237  * __dev_alloc_page - allocate a page for network Rx
2238  * @gfp_mask: allocation priority. Set __GFP_NOMEMALLOC if not for network Rx
2239  *
2240  * Allocate a new page.
2241  *
2242  * %NULL is returned if there is no free memory.
2243  */
2244 static inline struct page *__dev_alloc_page(gfp_t gfp_mask)
2245 {
2246         return __dev_alloc_pages(gfp_mask, 0);
2247 }
2248
2249 static inline struct page *dev_alloc_page(void)
2250 {
2251         return __dev_alloc_page(GFP_ATOMIC);
2252 }
2253
2254 /**
2255  *      skb_propagate_pfmemalloc - Propagate pfmemalloc if skb is allocated after RX page
2256  *      @page: The page that was allocated from skb_alloc_page
2257  *      @skb: The skb that may need pfmemalloc set
2258  */
2259 static inline void skb_propagate_pfmemalloc(struct page *page,
2260                                             struct sk_buff *skb)
2261 {
2262         if (page_is_pfmemalloc(page))
2263                 skb->pfmemalloc = true;
2264 }
2265
2266 /**
2267  * skb_frag_page - retrieve the page referred to by a paged fragment
2268  * @frag: the paged fragment
2269  *
2270  * Returns the &struct page associated with @frag.
2271  */
2272 static inline struct page *skb_frag_page(const skb_frag_t *frag)
2273 {
2274         return frag->page.p;
2275 }
2276
2277 /**
2278  * __skb_frag_ref - take an addition reference on a paged fragment.
2279  * @frag: the paged fragment
2280  *
2281  * Takes an additional reference on the paged fragment @frag.
2282  */
2283 static inline void __skb_frag_ref(skb_frag_t *frag)
2284 {
2285         get_page(skb_frag_page(frag));
2286 }
2287
2288 /**
```

```
2289  * skb_frag_ref - take an addition reference on a paged fragment of an skb.
2290  * @skb: the buffer
2291  * @f: the fragment offset.
2292  *
2293  * Takes an additional reference on the @f'th paged fragment of @skb.
2294  */
2295 static inline void skb_frag_ref(struct sk_buff *skb, int f)
2296 {
2297         __skb_frag_ref(&skb_shinfo(skb)->frags[f]);
2298 }
2299
2300 /**
2301  * __skb_frag_unref - release a reference on a paged fragment.
2302  * @frag: the paged fragment
2303  *
2304  * Releases a reference on the paged fragment @frag.
2305  */
2306 static inline void __skb_frag_unref(skb_frag_t *frag)
2307 {
2308         put_page(skb_frag_page(frag));
2309 }
2310
2311 /**
2312  * skb_frag_unref - release a reference on a paged fragment of an skb.
2313  * @skb: the buffer
2314  * @f: the fragment offset
2315  *
2316  * Releases a reference on the @f'th paged fragment of @skb.
2317  */
2318 static inline void skb_frag_unref(struct sk_buff *skb, int f)
2319 {
2320         __skb_frag_unref(&skb_shinfo(skb)->frags[f]);
2321 }
2322
2323 /**
2324  * skb_frag_address - gets the address of the data contained in a paged fragment
2325  * @frag: the paged fragment buffer
2326  *
2327  * Returns the address of the data within @frag. The page must already
2328  * be mapped.
2329  */
2330 static inline void *skb_frag_address(const skb_frag_t *frag)
2331 {
2332         return page_address(skb_frag_page(frag)) + frag->page_offset;
2333 }
2334
2335 /**
2336  * skb_frag_address_safe - gets the address of the data contained in a paged fragment
2337  * @frag: the paged fragment buffer
2338  *
2339  * Returns the address of the data within @frag. Checks that the page
2340  * is mapped and returns %NULL otherwise.
2341  */
2342 static inline void *skb_frag_address_safe(const skb_frag_t *frag)
2343 {
2344         void *ptr = page_address(skb_frag_page(frag));
2345         if (unlikely(!ptr))
2346                 return NULL;
2347
2348         return ptr + frag->page_offset;
2349 }
2350
2351 /**
2352  * __skb_frag_set_page - sets the page contained in a paged fragment
2353  * @frag: the paged fragment
2354  * @page: the page to set
```

```
2355  *
2356  * Sets the fragment @frag to contain @page.
2357  */
2358 static inline void __skb_frag_set_page(skb_frag_t *frag, struct page *page)
2359 {
2360         frag->page.p = page;
2361 }
2362
2363 /**
2364  * skb_frag_set_page - sets the page contained in a paged fragment of an skb
2365  * @skb: the buffer
2366  * @f: the fragment offset
2367  * @page: the page to set
2368  *
2369  * Sets the @f'th fragment of @skb to contain @page.
2370  */
2371 static inline void skb_frag_set_page(struct sk_buff *skb, int f,
2372                                      struct page *page)
2373 {
2374         __skb_frag_set_page(&skb_shinfo(skb)->frags[f], page);
2375 }
2376
2377 bool skb_page_frag_refill(unsigned int sz, struct page_frag *pfrag, gfp_t prio);
2378
2379 /**
2380  * skb_frag_dma_map - maps a paged fragment via the DMA API
2381  * @dev: the device to map the fragment to
2382  * @frag: the paged fragment to map
2383  * @offset: the offset within the fragment (starting at the
2384  *          fragment's own offset)
2385  * @size: the number of bytes to map
2386  * @dir: the direction of the mapping (%PCI_DMA_*)
2387  *
2388  * Maps the page associated with @frag to @device.
2389  */
2390 static inline dma_addr_t skb_frag_dma_map(struct device *dev,
2391                                           const skb_frag_t *frag,
2392                                           size_t offset, size_t size,
2393                                           enum dma_data_direction dir)
2394 {
2395         return dma_map_page(dev, skb_frag_page(frag),
2396                             frag->page_offset + offset, size, dir);
2397 }
2398
2399 static inline struct sk_buff *pskb_copy(struct sk_buff *skb,
2400                                         gfp_t gfp_mask)
2401 {
2402         return __pskb_copy(skb, skb_headroom(skb), gfp_mask);
2403 }
2404
2405
2406 static inline struct sk_buff *pskb_copy_for_clone(struct sk_buff *skb,
2407                                                   gfp_t gfp_mask)
2408 {
2409         return __pskb_copy_fclone(skb, skb_headroom(skb), gfp_mask, true);
2410 }
2411
2412
2413 /**
2414  *      skb_clone_writable - is the header of a clone writable
2415  *      @skb: buffer to check
2416  *      @len: length up to which to write
2417  *
2418  *      Returns true if modifying the header part of the cloned buffer
2419  *      does not requires the data to be copied.
2420  */
```

```
2421 static inline int skb_clone_writable(const struct sk_buff *skb, unsigned int len)
2422 {
2423         return !skb_header_cloned(skb) &&
2424                skb_headroom(skb) + len <= skb->hdr_len;
2425 }
2426
2427 static inline int __skb_cow(struct sk_buff *skb, unsigned int headroom,
2428                             int cloned)
2429 {
2430         int delta = 0;
2431
2432         if (headroom > skb_headroom(skb))
2433                 delta = headroom - skb_headroom(skb);
2434
2435         if (delta || cloned)
2436                 return pskb_expand_head(skb, ALIGN(delta, NET_SKB_PAD), 0,
2437                                         GFP_ATOMIC);
2438         return 0;
2439 }
2440
2441 /**
2442  *      skb_cow - copy header of skb when it is required
2443  *      @skb: buffer to cow
2444  *      @headroom: needed headroom
2445  *
2446  *      If the skb passed lacks sufficient headroom or its data part
2447  *      is shared, data is reallocated. If reallocation fails, an error
2448  *      is returned and original skb is not changed.
2449  *
2450  *      The result is skb with writable area skb->head...skb->tail
2451  *      and at least @headroom of space at head.
2452  */
2453 static inline int skb_cow(struct sk_buff *skb, unsigned int headroom)
2454 {
2455         return __skb_cow(skb, headroom, skb_cloned(skb));
2456 }
2457
2458 /**
2459  *      skb_cow_head - skb_cow but only making the head writable
2460  *      @skb: buffer to cow
2461  *      @headroom: needed headroom
2462  *
2463  *      This function is identical to skb_cow except that we replace the
2464  *      skb_cloned check by skb_header_cloned.  It should be used when
2465  *      you only need to push on some header and do not need to modify
2466  *      the data.
2467  */
2468 static inline int skb_cow_head(struct sk_buff *skb, unsigned int headroom)
2469 {
2470         return __skb_cow(skb, headroom, skb_header_cloned(skb));
2471 }
2472
2473 /**
2474  *      skb_padto       - pad an skbuff up to a minimal size
2475  *      @skb: buffer to pad
2476  *      @len: minimal length
2477  *
2478  *      Pads up a buffer to ensure the trailing bytes exist and are
2479  *      blanked. If the buffer already contains sufficient data it
2480  *      is untouched. Otherwise it is extended. Returns zero on
2481  *      success. The skb is freed on error.
2482  */
2483 static inline int skb_padto(struct sk_buff *skb, unsigned int len)
2484 {
2485         unsigned int size = skb->len;
2486         if (likely(size >= len))
```

```
2487                return 0;
2488        return skb_pad(skb, len - size);
2489 }
2490
2491 /**
2492  *      skb_put_padto - increase size and pad an skbuff up to a minimal size
2493  *      @skb: buffer to pad
2494  *      @len: minimal length
2495  *
2496  *      Pads up a buffer to ensure the trailing bytes exist and are
2497  *      blanked. If the buffer already contains sufficient data it
2498  *      is untouched. Otherwise it is extended. Returns zero on
2499  *      success. The skb is freed on error.
2500  */
2501 static inline int skb_put_padto(struct sk_buff *skb, unsigned int len)
2502 {
2503         unsigned int size = skb->len;
2504
2505         if (unlikely(size < len)) {
2506                 len -= size;
2507                 if (skb_pad(skb, len))
2508                         return -ENOMEM;
2509                 __skb_put(skb, len);
2510         }
2511         return 0;
2512 }
2513
2514 static inline int skb_add_data(struct sk_buff *skb,
2515                                struct iov_iter *from, int copy)
2516 {
2517         const int off = skb->len;
2518
2519         if (skb->ip_summed == CHECKSUM_NONE) {
2520                 __wsum csum = 0;
2521                 if (csum_and_copy_from_iter(skb_put(skb, copy), copy,
2522                                             &csum, from) == copy) {
2523                         skb->csum = csum_block_add(skb->csum, csum, off);
2524                         return 0;
2525                 }
2526         } else if (copy_from_iter(skb_put(skb, copy), copy, from) == copy)
2527                 return 0;
2528
2529         __skb_trim(skb, off);
2530         return -EFAULT;
2531 }
2532
2533 static inline bool skb_can_coalesce(struct sk_buff *skb, int i,
2534                                     const struct page *page, int off)
2535 {
2536         if (i) {
2537                 const struct skb_frag_struct *frag = &skb_shinfo(skb)->frags[i - 1];
2538
2539                 return page == skb_frag_page(frag) &&
2540                        off == frag->page_offset + skb_frag_size(frag);
2541         }
2542         return false;
2543 }
2544
2545 static inline int __skb_linearize(struct sk_buff *skb)
2546 {
2547         return __pskb_pull_tail(skb, skb->data_len) ? 0 : -ENOMEM;
2548 }
2549
2550 /**
2551  *      skb_linearize - convert paged skb to linear one
2552  *      @skb: buffer to linarize
```

```
2553  *
2554  *      If there is no free memory -ENOMEM is returned, otherwise zero
2555  *      is returned and the old skb data released.
2556  */
2557 static inline int skb_linearize(struct sk_buff *skb)
2558 {
2559         return skb_is_nonlinear(skb) ? __skb_linearize(skb) : 0;
2560 }
2561
2562 /**
2563  * skb_has_shared_frag - can any frag be overwritten
2564  * @skb: buffer to test
2565  *
2566  * Return true if the skb has at least one frag that might be modified
2567  * by an external entity (as in vmsplice()/sendfile())
2568  */
2569 static inline bool skb_has_shared_frag(const struct sk_buff *skb)
2570 {
2571         return skb_is_nonlinear(skb) &&
2572                skb_shinfo(skb)->tx_flags & SKBTX_SHARED_FRAG;
2573 }
2574
2575 /**
2576  *      skb_linearize_cow - make sure skb is linear and writable
2577  *      @skb: buffer to process
2578  *
2579  *      If there is no free memory -ENOMEM is returned, otherwise zero
2580  *      is returned and the old skb data released.
2581  */
2582 static inline int skb_linearize_cow(struct sk_buff *skb)
2583 {
2584         return skb_is_nonlinear(skb) || skb_cloned(skb) ?
2585                __skb_linearize(skb) : 0;
2586 }
2587
2588 /**
2589  *      skb_postpull_rcsum - update checksum for received skb after pull
2590  *      @skb: buffer to update
2591  *      @start: start of data before pull
2592  *      @len: length of data pulled
2593  *
2594  *      After doing a pull on a received packet, you need to call this to
2595  *      update the CHECKSUM_COMPLETE checksum, or set ip_summed to
2596  *      CHECKSUM_NONE so that it can be recomputed from scratch.
2597  */
2598
2599 static inline void skb_postpull_rcsum(struct sk_buff *skb,
2600                                       const void *start, unsigned int len)
2601 {
2602         if (skb->ip_summed == CHECKSUM_COMPLETE)
2603                 skb->csum = csum_sub(skb->csum, csum_partial(start, len, 0));
2604 }
2605
2606 unsigned char *skb_pull_rcsum(struct sk_buff *skb, unsigned int len);
2607
2608 /**
2609  *      pskb_trim_rcsum - trim received skb and update checksum
2610  *      @skb: buffer to trim
2611  *      @len: new length
2612  *
2613  *      This is exactly the same as pskb_trim except that it ensures the
2614  *      checksum of received packets are still valid after the operation.
2615  */
2616
2617 static inline int pskb_trim_rcsum(struct sk_buff *skb, unsigned int len)
2618 {
```

```
2619         if (likely(len >= skb->len))
2620                 return 0;
2621         if (skb->ip_summed == CHECKSUM_COMPLETE)
2622                 skb->ip_summed = CHECKSUM_NONE;
2623         return __pskb_trim(skb, len);
2624 }
2625
2626 #define skb_queue_walk(queue, skb) \
2627                 for (skb = (queue)->next;                                       \
2628                      skb != (struct sk_buff *)(queue);                          \
2629                      skb = skb->next)
2630
2631 #define skb_queue_walk_safe(queue, skb, tmp)                                    \
2632                 for (skb = (queue)->next, tmp = skb->next;                      \
2633                      skb != (struct sk_buff *)(queue);                          \
2634                      skb = tmp, tmp = skb->next)
2635
2636 #define skb_queue_walk_from(queue, skb)                                         \
2637                 for (; skb != (struct sk_buff *)(queue);                        \
2638                      skb = skb->next)
2639
2640 #define skb_queue_walk_from_safe(queue, skb, tmp)                               \
2641                 for (tmp = skb->next;                                           \
2642                      skb != (struct sk_buff *)(queue);                          \
2643                      skb = tmp, tmp = skb->next)
2644
2645 #define skb_queue_reverse_walk(queue, skb) \
2646                 for (skb = (queue)->prev;                                       \
2647                      skb != (struct sk_buff *)(queue);                          \
2648                      skb = skb->prev)
2649
2650 #define skb_queue_reverse_walk_safe(queue, skb, tmp)                            \
2651                 for (skb = (queue)->prev, tmp = skb->prev;                      \
2652                      skb != (struct sk_buff *)(queue);                          \
2653                      skb = tmp, tmp = skb->prev)
2654
2655 #define skb_queue_reverse_walk_from_safe(queue, skb, tmp)                       \
2656                 for (tmp = skb->prev;                                           \
2657                      skb != (struct sk_buff *)(queue);                          \
2658                      skb = tmp, tmp = skb->prev)
2659
2660 static inline bool skb_has_frag_list(const struct sk_buff *skb)
2661 {
2662         return skb_shinfo(skb)->frag_list != NULL;
2663 }
2664
2665 static inline void skb_frag_list_init(struct sk_buff *skb)
2666 {
2667         skb_shinfo(skb)->frag_list = NULL;
2668 }
2669
2670 static inline void skb_frag_add_head(struct sk_buff *skb, struct sk_buff *frag)
2671 {
2672         frag->next = skb_shinfo(skb)->frag_list;
2673         skb_shinfo(skb)->frag_list = frag;
2674 }
2675
2676 #define skb_walk_frags(skb, iter)       \
2677         for (iter = skb_shinfo(skb)->frag_list; iter; iter = iter->next)
2678
2679 struct sk_buff *__skb_recv_datagram(struct sock *sk, unsigned flags,
2680                                     int *peeked, int *off, int *err);
2681 struct sk_buff *skb_recv_datagram(struct sock *sk, unsigned flags, int noblock,
2682                                   int *err);
2683 unsigned int datagram_poll(struct file *file, struct socket *sock,
2684                            struct poll_table_struct *wait);
```

```
2685 int skb_copy_datagram_iter(const struct sk_buff *from, int offset,
2686                            struct iov_iter *to, int size);
2687 static inline int skb_copy_datagram_msg(const struct sk_buff *from, int offset,
2688                                         struct msghdr *msg, int size)
2689 {
2690         return skb_copy_datagram_iter(from, offset, &msg->msg_iter, size);
2691 }
2692 int skb_copy_and_csum_datagram_msg(struct sk_buff *skb, int hlen,
2693                                    struct msghdr *msg);
2694 int skb_copy_datagram_from_iter(struct sk_buff *skb, int offset,
2695                                 struct iov_iter *from, int len);
2696 int zerocopy_sg_from_iter(struct sk_buff *skb, struct iov_iter *frm);
2697 void skb_free_datagram(struct sock *sk, struct sk_buff *skb);
2698 void skb_free_datagram_locked(struct sock *sk, struct sk_buff *skb);
2699 int skb_kill_datagram(struct sock *sk, struct sk_buff *skb, unsigned int flags);
2700 int skb_copy_bits(const struct sk_buff *skb, int offset, void *to, int len);
2701 int skb_store_bits(struct sk_buff *skb, int offset, const void *from, int len);
2702 __wsum skb_copy_and_csum_bits(const struct sk_buff *skb, int offset, u8 *to,
2703                               int len, __wsum csum);
2704 ssize_t skb_socket_splice(struct sock *sk,
2705                           struct pipe_inode_info *pipe,
2706                           struct splice_pipe_desc *spd);
2707 int skb_splice_bits(struct sk_buff *skb, struct sock *sk, unsigned int offset,
2708                     struct pipe_inode_info *pipe, unsigned int len,
2709                     unsigned int flags,
2710                     ssize_t (*splice_cb)(struct sock *,
2711                                          struct pipe_inode_info *,
2712                                          struct splice_pipe_desc *));
2713 void skb_copy_and_csum_dev(const struct sk_buff *skb, u8 *to);
2714 unsigned int skb_zerocopy_headlen(const struct sk_buff *from);
2715 int skb_zerocopy(struct sk_buff *to, struct sk_buff *from,
2716                  int len, int hlen);
2717 void skb_split(struct sk_buff *skb, struct sk_buff *skb1, const u32 len);
2718 int skb_shift(struct sk_buff *tgt, struct sk_buff *skb, int shiftlen);
2719 void skb_scrub_packet(struct sk_buff *skb, bool xnet);
2720 unsigned int skb_gso_transport_seglen(const struct sk_buff *skb);
2721 struct sk_buff *skb_segment(struct sk_buff *skb, netdev_features_t features);
2722 struct sk_buff *skb_vlan_untag(struct sk_buff *skb);
2723 int skb_ensure_writable(struct sk_buff *skb, int write_len);
2724 int skb_vlan_pop(struct sk_buff *skb);
2725 int skb_vlan_push(struct sk_buff *skb, __be16 vlan_proto, u16 vlan_tci);
2726
2727 static inline int memcpy_from_msg(void *data, struct msghdr *msg, int len)
2728 {
2729         return copy_from_iter(data, len, &msg->msg_iter) == len ? 0 : -EFAULT;
2730 }
2731
2732 static inline int memcpy_to_msg(struct msghdr *msg, void *data, int len)
2733 {
2734         return copy_to_iter(data, len, &msg->msg_iter) == len ? 0 : -EFAULT;
2735 }
2736
2737 struct skb_checksum_ops {
2738         __wsum (*update)(const void *mem, int len, __wsum wsum);
2739         __wsum (*combine)(__wsum csum, __wsum csum2, int offset, int len);
2740 };
2741
2742 __wsum __skb_checksum(const struct sk_buff *skb, int offset, int len,
2743                       __wsum csum, const struct skb_checksum_ops *ops);
2744 __wsum skb_checksum(const struct sk_buff *skb, int offset, int len,
2745                     __wsum csum);
2746
2747 static inline void * __must_check
2748 __skb_header_pointer(const struct sk_buff *skb, int offset,
2749                      int len, void *data, int hlen, void *buffer)
2750 {
```

```
2751         if (hlen - offset >= len)
2752                 return data + offset;
2753
2754         if (!skb ||
2755             skb_copy_bits(skb, offset, buffer, len) < 0)
2756                 return NULL;
2757
2758         return buffer;
2759 }
2760
2761 static inline void * __must_check
2762 skb_header_pointer(const struct sk_buff *skb, int offset, int len, void *buffer)
2763 {
2764         return __skb_header_pointer(skb, offset, len, skb->data,
2765                                     skb_headlen(skb), buffer);
2766 }
2767
2768 /**
2769  *      skb_needs_linearize - check if we need to linearize a given skb
2770  *                            depending on the given device features.
2771  *      @skb: socket buffer to check
2772  *      @features: net device features
2773  *
2774  *      Returns true if either:
2775  *      1. skb has frag_list and the device doesn't support FRAGLIST, or
2776  *      2. skb is fragmented and the device does not support SG.
2777  */
2778 static inline bool skb_needs_linearize(struct sk_buff *skb,
2779                                        netdev_features_t features)
2780 {
2781         return skb_is_nonlinear(skb) &&
2782                ((skb_has_frag_list(skb) && !(features & NETIF_F_FRAGLIST)) ||
2783                 (skb_shinfo(skb)->nr_frags && !(features & NETIF_F_SG)));
2784 }
2785
2786 static inline void skb_copy_from_linear_data(const struct sk_buff *skb,
2787                                              void *to,
2788                                              const unsigned int len)
2789 {
2790         memcpy(to, skb->data, len);
2791 }
2792
2793 static inline void skb_copy_from_linear_data_offset(const struct sk_buff *skb,
2794                                                     const int offset, void *to,
2795                                                     const unsigned int len)
2796 {
2797         memcpy(to, skb->data + offset, len);
2798 }
2799
2800 static inline void skb_copy_to_linear_data(struct sk_buff *skb,
2801                                            const void *from,
2802                                            const unsigned int len)
2803 {
2804         memcpy(skb->data, from, len);
2805 }
2806
2807 static inline void skb_copy_to_linear_data_offset(struct sk_buff *skb,
2808                                                   const int offset,
2809                                                   const void *from,
2810                                                   const unsigned int len)
2811 {
2812         memcpy(skb->data + offset, from, len);
2813 }
2814
2815 void skb_init(void);
2816
```

```
2817 static inline ktime_t skb_get_ktime(const struct sk_buff *skb)
2818 {
2819         return skb->tstamp;
2820 }
2821
2822 /**
2823  *      skb_get_timestamp - get timestamp from a skb
2824  *      @skb: skb to get stamp from
2825  *      @stamp: pointer to struct timeval to store stamp in
2826  *
2827  *      Timestamps are stored in the skb as offsets to a base timestamp.
2828  *      This function converts the offset back to a struct timeval and stores
2829  *      it in stamp.
2830  */
2831 static inline void skb_get_timestamp(const struct sk_buff *skb,
2832                                      struct timeval *stamp)
2833 {
2834         *stamp = ktime_to_timeval(skb->tstamp);
2835 }
2836
2837 static inline void skb_get_timestampns(const struct sk_buff *skb,
2838                                        struct timespec *stamp)
2839 {
2840         *stamp = ktime_to_timespec(skb->tstamp);
2841 }
2842
2843 static inline void __net_timestamp(struct sk_buff *skb)
2844 {
2845         skb->tstamp = ktime_get_real();
2846 }
2847
2848 static inline ktime_t net_timedelta(ktime_t t)
2849 {
2850         return ktime_sub(ktime_get_real(), t);
2851 }
2852
2853 static inline ktime_t net_invalid_timestamp(void)
2854 {
2855         return ktime_set(0, 0);
2856 }
2857
2858 struct sk_buff *skb_clone_sk(struct sk_buff *skb);
2859
2860 #ifdef CONFIG_NETWORK_PHY_TIMESTAMPING
2861
2862 void skb_clone_tx_timestamp(struct sk_buff *skb);
2863 bool skb_defer_rx_timestamp(struct sk_buff *skb);
2864
2865 #else /* CONFIG_NETWORK_PHY_TIMESTAMPING */
2866
2867 static inline void skb_clone_tx_timestamp(struct sk_buff *skb)
2868 {
2869 }
2870
2871 static inline bool skb_defer_rx_timestamp(struct sk_buff *skb)
2872 {
2873         return false;
2874 }
2875
2876 #endif /* !CONFIG_NETWORK_PHY_TIMESTAMPING */
2877
2878 /**
2879  * skb_complete_tx_timestamp() - deliver cloned skb with tx timestamps
2880  *
2881  * PHY drivers may accept clones of transmitted packets for
2882  * timestamping via their phy_driver.txtstamp method. These drivers
```

```
2883  * must call this function to return the skb back to the stack with a
2884  * timestamp.
2885  *
2886  * @skb: clone of the the original outgoing packet
2887  * @hwtstamps: hardware time stamps
2888  *
2889  */
2890 void skb_complete_tx_timestamp(struct sk_buff *skb,
2891                                struct skb_shared_hwtstamps *hwtstamps);
2892
2893 void __skb_tstamp_tx(struct sk_buff *orig_skb,
2894                      struct skb_shared_hwtstamps *hwtstamps,
2895                      struct sock *sk, int tstype);
2896
2897 /**
2898  * skb_tstamp_tx - queue clone of skb with send time stamps
2899  * @orig_skb:   the original outgoing packet
2900  * @hwtstamps:  hardware time stamps, may be NULL if not available
2901  *
2902  * If the skb has a socket associated, then this function clones the
2903  * skb (thus sharing the actual data and optional structures), stores
2904  * the optional hardware time stamping information (if non NULL) or
2905  * generates a software time stamp (otherwise), then queues the clone
2906  * to the error queue of the socket.  Errors are silently ignored.
2907  */
2908 void skb_tstamp_tx(struct sk_buff *orig_skb,
2909                    struct skb_shared_hwtstamps *hwtstamps);
2910
2911 static inline void sw_tx_timestamp(struct sk_buff *skb)
2912 {
2913         if (skb_shinfo(skb)->tx_flags & SKBTX_SW_TSTAMP &&
2914             !(skb_shinfo(skb)->tx_flags & SKBTX_IN_PROGRESS))
2915                 skb_tstamp_tx(skb, NULL);
2916 }
2917
2918 /**
2919  * skb_tx_timestamp() - Driver hook for transmit timestamping
2920  *
2921  * Ethernet MAC Drivers should call this function in their hard_xmit()
2922  * function immediately before giving the sk_buff to the MAC hardware.
2923  *
2924  * Specifically, one should make absolutely sure that this function is
2925  * called before TX completion of this packet can trigger.  Otherwise
2926  * the packet could potentially already be freed.
2927  *
2928  * @skb: A socket buffer.
2929  */
2930 static inline void skb_tx_timestamp(struct sk_buff *skb)
2931 {
2932         skb_clone_tx_timestamp(skb);
2933         sw_tx_timestamp(skb);
2934 }
2935
2936 /**
2937  * skb_complete_wifi_ack - deliver skb with wifi status
2938  *
2939  * @skb: the original outgoing packet
2940  * @acked: ack status
2941  *
2942  */
2943 void skb_complete_wifi_ack(struct sk_buff *skb, bool acked);
2944
2945 __sum16 __skb_checksum_complete_head(struct sk_buff *skb, int len);
2946 __sum16 __skb_checksum_complete(struct sk_buff *skb);
2947
2948 static inline int skb_csum_unnecessary(const struct sk_buff *skb)
```

```
2949 {
2950         return ((skb->ip_summed == CHECKSUM_UNNECESSARY) ||
2951                 skb->csum_valid ||
2952                 (skb->ip_summed == CHECKSUM_PARTIAL &&
2953                  skb_checksum_start_offset(skb) >= 0));
2954 }
2955
2956 /**
2957  *      skb_checksum_complete - Calculate checksum of an entire packet
2958  *      @skb: packet to process
2959  *
2960  *      This function calculates the checksum over the entire packet plus
2961  *      the value of skb->csum.  The latter can be used to supply the
2962  *      checksum of a pseudo header as used by TCP/UDP.  It returns the
2963  *      checksum.
2964  *
2965  *      For protocols that contain complete checksums such as ICMP/TCP/UDP,
2966  *      this function can be used to verify that checksum on received
2967  *      packets.  In that case the function should return zero if the
2968  *      checksum is correct.  In particular, this function will return zero
2969  *      if skb->ip_summed is CHECKSUM_UNNECESSARY which indicates that the
2970  *      hardware has already verified the correctness of the checksum.
2971  */
2972 static inline __sum16 skb_checksum_complete(struct sk_buff *skb)
2973 {
2974         return skb_csum_unnecessary(skb) ?
2975                0 : __skb_checksum_complete(skb);
2976 }
2977
2978 static inline void __skb_decr_checksum_unnecessary(struct sk_buff *skb)
2979 {
2980         if (skb->ip_summed == CHECKSUM_UNNECESSARY) {
2981                 if (skb->csum_level == 0)
2982                         skb->ip_summed = CHECKSUM_NONE;
2983                 else
2984                         skb->csum_level--;
2985         }
2986 }
2987
2988 static inline void __skb_incr_checksum_unnecessary(struct sk_buff *skb)
2989 {
2990         if (skb->ip_summed == CHECKSUM_UNNECESSARY) {
2991                 if (skb->csum_level < SKB_MAX_CSUM_LEVEL)
2992                         skb->csum_level++;
2993         } else if (skb->ip_summed == CHECKSUM_NONE) {
2994                 skb->ip_summed = CHECKSUM_UNNECESSARY;
2995                 skb->csum_level = 0;
2996         }
2997 }
2998
2999 static inline void __skb_mark_checksum_bad(struct sk_buff *skb)
3000 {
3001         /* Mark current checksum as bad (typically called from GRO
3002          * path). In the case that ip_summed is CHECKSUM_NONE
3003          * this must be the first checksum encountered in the packet.
3004          * When ip_summed is CHECKSUM_UNNECESSARY, this is the first
3005          * checksum after the last one validated. For UDP, a zero
3006          * checksum can not be marked as bad.
3007          */
3008
3009         if (skb->ip_summed == CHECKSUM_NONE ||
3010             skb->ip_summed == CHECKSUM_UNNECESSARY)
3011                 skb->csum_bad = 1;
3012 }
3013
3014 /* Check if we need to perform checksum complete validation.
```

```
3015  *
3016  * Returns true if checksum complete is needed, false otherwise
3017  * (either checksum is unnecessary or zero checksum is allowed).
3018  */
3019 static inline bool __skb_checksum_validate_needed(struct sk_buff *skb,
3020                                                   bool zero_okay,
3021                                                   __sum16 check)
3022 {
3023         if (skb_csum_unnecessary(skb) || (zero_okay && !check)) {
3024                 skb->csum_valid = 1;
3025                 __skb_decr_checksum_unnecessary(skb);
3026                 return false;
3027         }
3028
3029         return true;
3030 }
3031
3032 /* For small packets <= CHECKSUM_BREAK peform checksum complete directly
3033  * in checksum_init.
3034  */
3035 #define CHECKSUM_BREAK 76
3036
3037 /* Unset checksum-complete
3038  *
3039  * Unset checksum complete can be done when packet is being modified
3040  * (uncompressed for instance) and checksum-complete value is
3041  * invalidated.
3042  */
3043 static inline void skb_checksum_complete_unset(struct sk_buff *skb)
3044 {
3045         if (skb->ip_summed == CHECKSUM_COMPLETE)
3046                 skb->ip_summed = CHECKSUM_NONE;
3047 }
3048
3049 /* Validate (init) checksum based on checksum complete.
3050  *
3051  * Return values:
3052  *   0: checksum is validated or try to in skb_checksum_complete. In the latter
3053  *      case the ip_summed will not be CHECKSUM_UNNECESSARY and the pseudo
3054  *      checksum is stored in skb->csum for use in __skb_checksum_complete
3055  *   non-zero: value of invalid checksum
3056  *
3057  */
3058 static inline __sum16 __skb_checksum_validate_complete(struct sk_buff *skb,
3059                                                        bool complete,
3060                                                        __wsum psum)
3061 {
3062         if (skb->ip_summed == CHECKSUM_COMPLETE) {
3063                 if (!csum_fold(csum_add(psum, skb->csum))) {
3064                         skb->csum_valid = 1;
3065                         return 0;
3066                 }
3067         } else if (skb->csum_bad) {
3068                 /* ip_summed == CHECKSUM_NONE in this case */
3069                 return (__force __sum16)1;
3070         }
3071
3072         skb->csum = psum;
3073
3074         if (complete || skb->len <= CHECKSUM_BREAK) {
3075                 __sum16 csum;
3076
3077                 csum = __skb_checksum_complete(skb);
3078                 skb->csum_valid = !csum;
3079                 return csum;
3080         }
```

```
3081
3082         return 0;
3083 }
3084
3085 static inline __wsum null_compute_pseudo(struct sk_buff *skb, int proto)
3086 {
3087         return 0;
3088 }
3089
3090 /* Perform checksum validate (init). Note that this is a macro since we only
3091  * want to calculate the pseudo header which is an input function if necessary.
3092  * First we try to validate without any computation (checksum unnecessary) and
3093  * then calculate based on checksum complete calling the function to compute
3094  * pseudo header.
3095  *
3096  * Return values:
3097  *   0: checksum is validated or try to in skb_checksum_complete
3098  *   non-zero: value of invalid checksum
3099  */
3100 #define __skb_checksum_validate(skb, proto, complete,                   \
3101                                 zero_okay, check, compute_pseudo)       \
3102 ({                                                                      \
3103         __sum16 __ret = 0;                                              \
3104         skb->csum_valid = 0;                                            \
3105         if (__skb_checksum_validate_needed(skb, zero_okay, check))      \
3106                 __ret = __skb_checksum_validate_complete(skb,           \
3107                                 complete, compute_pseudo(skb, proto));  \
3108         __ret;                                                          \
3109 })
3110
3111 #define skb_checksum_init(skb, proto, compute_pseudo)                   \
3112         __skb_checksum_validate(skb, proto, false, false, 0, compute_pseudo)
3113
3114 #define skb_checksum_init_zero_check(skb, proto, check, compute_pseudo) \
3115         __skb_checksum_validate(skb, proto, false, true, check, compute_pseudo)
3116
3117 #define skb_checksum_validate(skb, proto, compute_pseudo)               \
3118         __skb_checksum_validate(skb, proto, true, false, 0, compute_pseudo)
3119
3120 #define skb_checksum_validate_zero_check(skb, proto, check,             \
3121                                          compute_pseudo)                \
3122         __skb_checksum_validate(skb, proto, true, true, check, compute_pseudo)
3123
3124 #define skb_checksum_simple_validate(skb)                               \
3125         __skb_checksum_validate(skb, 0, true, false, 0, null_compute_pseudo)
3126
3127 static inline bool __skb_checksum_convert_check(struct sk_buff *skb)
3128 {
3129         return (skb->ip_summed == CHECKSUM_NONE &&
3130                 skb->csum_valid && !skb->csum_bad);
3131 }
3132
3133 static inline void __skb_checksum_convert(struct sk_buff *skb,
3134                                           __sum16 check, __wsum pseudo)
3135 {
3136         skb->csum = ~pseudo;
3137         skb->ip_summed = CHECKSUM_COMPLETE;
3138 }
3139
3140 #define skb_checksum_try_convert(skb, proto, check, compute_pseudo)     \
3141 do {                                                                    \
3142         if (__skb_checksum_convert_check(skb))                          \
3143                 __skb_checksum_convert(skb, check,                      \
3144                                        compute_pseudo(skb, proto));     \
3145 } while (0)
3146
```

```
3147 static inline void skb_remcsum_adjust_partial(struct sk_buff *skb, void *ptr,
3148                                                 u16 start, u16 offset)
3149 {
3150         skb->ip_summed = CHECKSUM_PARTIAL;
3151         skb->csum_start = ((unsigned char *)ptr + start) - skb->head;
3152         skb->csum_offset = offset - start;
3153 }
3154
3155 /* Update skbuf and packet to reflect the remote checksum offload operation.
3156  * When called, ptr indicates the starting point for skb->csum when
3157  * ip_summed is CHECKSUM_COMPLETE. If we need create checksum complete
3158  * here, skb_postpull_rcsum is done so skb->csum start is ptr.
3159  */
3160 static inline void skb_remcsum_process(struct sk_buff *skb, void *ptr,
3161                                        int start, int offset, bool nopartial)
3162 {
3163         __wsum delta;
3164
3165         if (!nopartial) {
3166                 skb_remcsum_adjust_partial(skb, ptr, start, offset);
3167                 return;
3168         }
3169
3170          if (unlikely(skb->ip_summed != CHECKSUM_COMPLETE)) {
3171                 __skb_checksum_complete(skb);
3172                 skb_postpull_rcsum(skb, skb->data, ptr - (void *)skb->data);
3173         }
3174
3175         delta = remcsum_adjust(ptr, skb->csum, start, offset);
3176
3177         /* Adjust skb->csum since we changed the packet */
3178         skb->csum = csum_add(skb->csum, delta);
3179 }
3180
3181 #if defined(CONFIG_NF_CONNTRACK) || defined(CONFIG_NF_CONNTRACK_MODULE)
3182 void nf_conntrack_destroy(struct nf_conntrack *nfct);
3183 static inline void nf_conntrack_put(struct nf_conntrack *nfct)
3184 {
3185         if (nfct && atomic_dec_and_test(&nfct->use))
3186                 nf_conntrack_destroy(nfct);
3187 }
3188 static inline void nf_conntrack_get(struct nf_conntrack *nfct)
3189 {
3190         if (nfct)
3191                 atomic_inc(&nfct->use);
3192 }
3193 #endif
3194 #if IS_ENABLED(CONFIG_BRIDGE_NETFILTER)
3195 static inline void nf_bridge_put(struct nf_bridge_info *nf_bridge)
3196 {
3197         if (nf_bridge && atomic_dec_and_test(&nf_bridge->use))
3198                 kfree(nf_bridge);
3199 }
3200 static inline void nf_bridge_get(struct nf_bridge_info *nf_bridge)
3201 {
3202         if (nf_bridge)
3203                 atomic_inc(&nf_bridge->use);
3204 }
3205 #endif /* CONFIG_BRIDGE_NETFILTER */
3206 static inline void nf_reset(struct sk_buff *skb)
3207 {
3208 #if defined(CONFIG_NF_CONNTRACK) || defined(CONFIG_NF_CONNTRACK_MODULE)
3209         nf_conntrack_put(skb->nfct);
3210         skb->nfct = NULL;
3211 #endif
3212 #if IS_ENABLED(CONFIG_BRIDGE_NETFILTER)
```

```
3213         nf_bridge_put(skb->nf_bridge);
3214         skb->nf_bridge = NULL;
3215 #endif
3216 }
3217
3218 static inline void nf_reset_trace(struct sk_buff *skb)
3219 {
3220 #if IS_ENABLED(CONFIG_NETFILTER_XT_TARGET_TRACE) || defined(CONFIG_NF_TABLES)
3221         skb->nf_trace = 0;
3222 #endif
3223 }
3224
3225 /* Note: This doesn't put any conntrack and bridge info in dst. */
3226 static inline void __nf_copy(struct sk_buff *dst, const struct sk_buff *src,
3227                              bool copy)
3228 {
3229 #if defined(CONFIG_NF_CONNTRACK) || defined(CONFIG_NF_CONNTRACK_MODULE)
3230         dst->nfct = src->nfct;
3231         nf_conntrack_get(src->nfct);
3232         if (copy)
3233                 dst->nfctinfo = src->nfctinfo;
3234 #endif
3235 #if IS_ENABLED(CONFIG_BRIDGE_NETFILTER)
3236         dst->nf_bridge  = src->nf_bridge;
3237         nf_bridge_get(src->nf_bridge);
3238 #endif
3239 #if IS_ENABLED(CONFIG_NETFILTER_XT_TARGET_TRACE) || defined(CONFIG_NF_TABLES)
3240         if (copy)
3241                 dst->nf_trace = src->nf_trace;
3242 #endif
3243 }
3244
3245 static inline void nf_copy(struct sk_buff *dst, const struct sk_buff *src)
3246 {
3247 #if defined(CONFIG_NF_CONNTRACK) || defined(CONFIG_NF_CONNTRACK_MODULE)
3248         nf_conntrack_put(dst->nfct);
3249 #endif
3250 #if IS_ENABLED(CONFIG_BRIDGE_NETFILTER)
3251         nf_bridge_put(dst->nf_bridge);
3252 #endif
3253         __nf_copy(dst, src, true);
3254 }
3255
3256 #ifdef CONFIG_NETWORK_SECMARK
3257 static inline void skb_copy_secmark(struct sk_buff *to, const struct sk_buff *from)
3258 {
3259         to->secmark = from->secmark;
3260 }
3261
3262 static inline void skb_init_secmark(struct sk_buff *skb)
3263 {
3264         skb->secmark = 0;
3265 }
3266 #else
3267 static inline void skb_copy_secmark(struct sk_buff *to, const struct sk_buff *from)
3268 { }
3269
3270 static inline void skb_init_secmark(struct sk_buff *skb)
3271 { }
3272 #endif
3273
3274 static inline bool skb_irq_freeable(const struct sk_buff *skb)
3275 {
3276         return !skb->destructor &&
3277 #if IS_ENABLED(CONFIG_XFRM)
3278                 !skb->sp &&
```

```
3279 #endif
3280 #if IS_ENABLED(CONFIG_NF_CONNTRACK)
3281                 !skb->nfct &&
3282 #endif
3283                 !skb->_skb_refdst &&
3284                 !skb_has_frag_list(skb);
3285 }
3286
3287 static inline void skb_set_queue_mapping(struct sk_buff *skb, u16 queue_mapping)
3288 {
3289         skb->queue_mapping = queue_mapping;
3290 }
3291
3292 static inline u16 skb_get_queue_mapping(const struct sk_buff *skb)
3293 {
3294         return skb->queue_mapping;
3295 }
3296
3297 static inline void skb_copy_queue_mapping(struct sk_buff *to, const struct sk_buff *from)
3298 {
3299         to->queue_mapping = from->queue_mapping;
3300 }
3301
3302 static inline void skb_record_rx_queue(struct sk_buff *skb, u16 rx_queue)
3303 {
3304         skb->queue_mapping = rx_queue + 1;
3305 }
3306
3307 static inline u16 skb_get_rx_queue(const struct sk_buff *skb)
3308 {
3309         return skb->queue_mapping - 1;
3310 }
3311
3312 static inline bool skb_rx_queue_recorded(const struct sk_buff *skb)
3313 {
3314         return skb->queue_mapping != 0;
3315 }
3316
3317 static inline struct sec_path *skb_sec_path(struct sk_buff *skb)
3318 {
3319 #ifdef CONFIG_XFRM
3320         return skb->sp;
3321 #else
3322         return NULL;
3323 #endif
3324 }
3325
3326 /* Keeps track of mac header offset relative to skb->head.
3327  * It is useful for TSO of Tunneling protocol. e.g. GRE.
3328  * For non-tunnel skb it points to skb_mac_header() and for
3329  * tunnel skb it points to outer mac header.
3330  * Keeps track of level of encapsulation of network headers.
3331  */
3332 struct skb_gso_cb {
3333         int     mac_offset;
3334         int     encap_level;
3335         __u16   csum_start;
3336 };
3337 #define SKB_GSO_CB(skb) ((struct skb_gso_cb *)(skb)->cb)
3338
3339 static inline int skb_tnl_header_len(const struct sk_buff *inner_skb)
3340 {
3341         return (skb_mac_header(inner_skb) - inner_skb->head) -
3342                 SKB_GSO_CB(inner_skb)->mac_offset;
3343 }
3344
```

```
3345 static inline int gso_pskb_expand_head(struct sk_buff *skb, int extra)
3346 {
3347         int new_headroom, headroom;
3348         int ret;
3349
3350         headroom = skb_headroom(skb);
3351         ret = pskb_expand_head(skb, extra, 0, GFP_ATOMIC);
3352         if (ret)
3353                 return ret;
3354
3355         new_headroom = skb_headroom(skb);
3356         SKB_GSO_CB(skb)->mac_offset += (new_headroom - headroom);
3357         return 0;
3358 }
3359
3360 /* Compute the checksum for a gso segment. First compute the checksum value
3361  * from the start of transport header to SKB_GSO_CB(skb)->csum_start, and
3362  * then add in skb->csum (checksum from csum_start to end of packet).
3363  * skb->csum and csum_start are then updated to reflect the checksum of the
3364  * resultant packet starting from the transport header-- the resultant checksum
3365  * is in the res argument (i.e. normally zero or ~ of checksum of a pseudo
3366  * header.
3367  */
3368 static inline __sum16 gso_make_checksum(struct sk_buff *skb, __wsum res)
3369 {
3370         int plen = SKB_GSO_CB(skb)->csum_start - skb_headroom(skb) -
3371                    skb_transport_offset(skb);
3372         __wsum partial;
3373
3374         partial = csum_partial(skb_transport_header(skb), plen, skb->csum);
3375         skb->csum = res;
3376         SKB_GSO_CB(skb)->csum_start -= plen;
3377
3378         return csum_fold(partial);
3379 }
3380
3381 static inline bool skb_is_gso(const struct sk_buff *skb)
3382 {
3383         return skb_shinfo(skb)->gso_size;
3384 }
3385
3386 /* Note: Should be called only if skb_is_gso(skb) is true */
3387 static inline bool skb_is_gso_v6(const struct sk_buff *skb)
3388 {
3389         return skb_shinfo(skb)->gso_type & SKB_GSO_TCPV6;
3390 }
3391
3392 void __skb_warn_lro_forwarding(const struct sk_buff *skb);
3393
3394 static inline bool skb_warn_if_lro(const struct sk_buff *skb)
3395 {
3396         /* LRO sets gso_size but not gso_type, whereas if GSO is really
3397          * wanted then gso_type will be set. */
3398         const struct skb_shared_info *shinfo = skb_shinfo(skb);
3399
3400         if (skb_is_nonlinear(skb) && shinfo->gso_size != 0 &&
3401             unlikely(shinfo->gso_type == 0)) {
3402                 __skb_warn_lro_forwarding(skb);
3403                 return true;
3404         }
3405         return false;
3406 }
3407
3408 static inline void skb_forward_csum(struct sk_buff *skb)
3409 {
3410         /* Unfortunately we don't support this one.  Any brave souls? */
```

```
3411         if (skb->ip_summed == CHECKSUM_COMPLETE)
3412                 skb->ip_summed = CHECKSUM_NONE;
3413 }
3414
3415 /**
3416  * skb_checksum_none_assert - make sure skb ip_summed is CHECKSUM_NONE
3417  * @skb: skb to check
3418  *
3419  * fresh skbs have their ip_summed set to CHECKSUM_NONE.
3420  * Instead of forcing ip_summed to CHECKSUM_NONE, we can
3421  * use this helper, to document places where we make this assertion.
3422  */
3423 static inline void skb_checksum_none_assert(const struct sk_buff *skb)
3424 {
3425 #ifdef DEBUG
3426         BUG_ON(skb->ip_summed != CHECKSUM_NONE);
3427 #endif
3428 }
3429
3430 bool skb_partial_csum_set(struct sk_buff *skb, u16 start, u16 off);
3431
3432 int skb_checksum_setup(struct sk_buff *skb, bool recalculate);
3433 struct sk_buff *skb_checksum_trimmed(struct sk_buff *skb,
3434                                      unsigned int transport_len,
3435                                      __sum16(*skb_chkf)(struct sk_buff *skb));
3436
3437 /**
3438  * skb_head_is_locked - Determine if the skb->head is locked down
3439  * @skb: skb to check
3440  *
3441  * The head on skbs build around a head frag can be removed if they are
3442  * not cloned.  This function returns true if the skb head is locked down
3443  * due to either being allocated via kmalloc, or by being a clone with
3444  * multiple references to the head.
3445  */
3446 static inline bool skb_head_is_locked(const struct sk_buff *skb)
3447 {
3448         return !skb->head_frag || skb_cloned(skb);
3449 }
3450
3451 /**
3452  * skb_gso_network_seglen - Return length of individual segments of a gso packet
3453  *
3454  * @skb: GSO skb
3455  *
3456  * skb_gso_network_seglen is used to determine the real size of the
3457  * individual segments, including Layer3 (IP, IPv6) and L4 headers (TCP/UDP).
3458  *
3459  * The MAC/L2 header is not accounted for.
3460  */
3461 static inline unsigned int skb_gso_network_seglen(const struct sk_buff *skb)
3462 {
3463         unsigned int hdr_len = skb_transport_header(skb) -
3464                                skb_network_header(skb);
3465         return hdr_len + skb_gso_transport_seglen(skb);
3466 }
3467 #endif  /* __KERNEL__ */
3468 #endif  /* _LINUX_SKBUFF_H */
3469
```

This page was automatically generated by LXR 0.3.1 (source). • Linux is a registered trademark of Linus Torvalds • Contact us

- Home
- Development

- Services
- Training
- Docs
- Community
- Company
- Blog