Linux Cross Reference

Free Electrons

Embedded Linux Experts

Linux/include/linux/netdevice.h

```
 1 /*
 2  * INET         An implementation of the TCP/IP protocol suite for the LINUX
 3  *              operating system.  INET is implemented using the  BSD Socket
 4  *              interface as the means of communication with the user level.
 5  *
 6  *              Definitions for the Interfaces handler.
 7  *
 8  * Version:     @(#)dev.h       1.0.10  08/12/93
 9  *
10  * Authors:     Ross Biro
11  *              Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
12  *              Corey Minyard <wf-rch!minyard@relay.EU.net>
13  *              Donald J. Becker, <becker@cesdis.gsfc.nasa.gov>
14  *              Alan Cox, <alan@lxorguk.ukuu.org.uk>
15  *              Bjorn Ekwall. <bj0rn@blox.se>
16  *              Pekka Riikonen <priikone@poseidon.pspt.fi>
17  *
18  *              This program is free software; you can redistribute it and/or
19  *              modify it under the terms of the GNU General Public License
20  *              as published by the Free Software Foundation; either version
21  *              2 of the License, or (at your option) any later version.
22  *
23  *              Moved to /usr/include/linux for NET3
24  */
25 #ifndef _LINUX_NETDEVICE_H
26 #define _LINUX_NETDEVICE_H
27
28 #include <linux/timer.h>
29 #include <linux/bug.h>
30 #include <linux/delay.h>
31 #include <linux/atomic.h>
32 #include <linux/prefetch.h>
33 #include <asm/cache.h>
34 #include <asm/byteorder.h>
35
36 #include <linux/percpu.h>
37 #include <linux/rculist.h>
38 #include <linux/dmaengine.h>
39 #include <linux/workqueue.h>
40 #include <linux/dynamic_queue_limits.h>
41
42 #include <linux/ethtool.h>
43 #include <net/net_namespace.h>
44 #include <net/dsa.h>
45 #ifdef CONFIG_DCB
46 #include <net/dcbnl.h>
47 #endif
48 #include <net/netprio_cgroup.h>
49
50 #include <linux/netdev_features.h>
51 #include <linux/neighbour.h>
52 #include <uapi/linux/netdevice.h>
53 #include <uapi/linux/if_bonding.h>
```

```
 54
 55 struct netpoll_info;
 56 struct device;
 57 struct phy_device;
 58 /* 802.11 specific */
 59 struct wireless_dev;
 60 /* 802.15.4 specific */
 61 struct wpan_dev;
 62 struct mpls_dev;
 63
 64 void netdev_set_default_ethtool_ops(struct net_device *dev,
 65                                     const struct ethtool_ops *ops);
 66
 67 /* Backlog congestion levels */
 68 #define NET_RX_SUCCESS          0       /* keep 'em coming, baby */
 69 #define NET_RX_DROP             1       /* packet dropped */
 70
 71 /*
 72  * Transmit return codes: transmit return codes originate from three different
 73  * namespaces:
 74  *
 75  * - qdisc return codes
 76  * - driver transmit return codes
 77  * - errno values
 78  *
 79  * Drivers are allowed to return any one of those in their hard_start_xmit()
 80  * function. Real network devices commonly used with qdiscs should only return
 81  * the driver transmit return codes though - when qdiscs are used, the actual
 82  * transmission happens asynchronously, so the value is not propagated to
 83  * higher layers. Virtual network devices transmit synchronously, in this case
 84  * the driver transmit return codes are consumed by dev_queue_xmit(), all
 85  * others are propagated to higher layers.
 86  */
 87
 88 /* qdisc ->enqueue() return codes. */
 89 #define NET_XMIT_SUCCESS        0x00
 90 #define NET_XMIT_DROP           0x01    /* skb dropped                  */
 91 #define NET_XMIT_CN             0x02    /* congestion notification      */
 92 #define NET_XMIT_POLICED        0x03    /* skb is shot by police        */
 93 #define NET_XMIT_MASK           0x0f    /* qdisc flags in net/sch_generic.h */
 94
 95 /* NET_XMIT_CN is special. It does not guarantee that this packet is lost. It
 96  * indicates that the device will soon be dropping packets, or already drops
 97  * some packets of the same priority; prompting us to send less aggressively. */
 98 #define net_xmit_eval(e)        ((e) == NET_XMIT_CN ? 0 : (e))
 99 #define net_xmit_errno(e)       ((e) != NET_XMIT_CN ? -ENOBUFS : 0)
100
101 /* Driver transmit return codes */
102 #define NETDEV_TX_MASK          0xf0
103
104 enum netdev_tx {
105         __NETDEV_TX_MIN  = INT_MIN,     /* make sure enum is signed */
106         NETDEV_TX_OK     = 0x00,        /* driver took care of packet */
107         NETDEV_TX_BUSY   = 0x10,        /* driver tx path was busy*/
108         NETDEV_TX_LOCKED = 0x20,        /* driver tx lock was already taken */
109 };
110 typedef enum netdev_tx netdev_tx_t;
111
112 /*
113  * Current order: NETDEV_TX_MASK > NET_XMIT_MASK >= 0 is significant;
114  * hard_start_xmit() return < NET_XMIT_MASK means skb was consumed.
115  */
116 static inline bool dev_xmit_complete(int rc)
117 {
118         /*
119          * Positive cases with an skb consumed by a driver:
120          * - successful transmission (rc == NETDEV_TX_OK)
121          * - error while transmitting (rc < 0)
122          * - error while queueing to a different device (rc & NET_XMIT_MASK)
123          */
124         if (likely(rc < NET_XMIT_MASK))
125                 return true;
126
```

```
127         return false;
128 }
129
130 /*
131  *      Compute the worst case header length according to the protocols
132  *      used.
133  */
134
135 #if defined(CONFIG_WLAN) || IS_ENABLED(CONFIG_AX25)
136 # if defined(CONFIG_MAC80211_MESH)
137 #  define LL_MAX_HEADER 128
138 # else
139 #  define LL_MAX_HEADER 96
140 # endif
141 #else
142 # define LL_MAX_HEADER 32
143 #endif
144
145 #if !IS_ENABLED(CONFIG_NET_IPIP) && !IS_ENABLED(CONFIG_NET_IPGRE) && \
146     !IS_ENABLED(CONFIG_IPV6_SIT) && !IS_ENABLED(CONFIG_IPV6_TUNNEL)
147 #define MAX_HEADER LL_MAX_HEADER
148 #else
149 #define MAX_HEADER (LL_MAX_HEADER + 48)
150 #endif
151
152 /*
153  *      Old network device statistics. Fields are native words
154  *      (unsigned long) so they can be read and written atomically.
155  */
156
157 struct net_device_stats {
158         unsigned long   rx_packets;
159         unsigned long   tx_packets;
160         unsigned long   rx_bytes;
161         unsigned long   tx_bytes;
162         unsigned long   rx_errors;
163         unsigned long   tx_errors;
164         unsigned long   rx_dropped;
165         unsigned long   tx_dropped;
166         unsigned long   multicast;
167         unsigned long   collisions;
168         unsigned long   rx_length_errors;
169         unsigned long   rx_over_errors;
170         unsigned long   rx_crc_errors;
171         unsigned long   rx_frame_errors;
172         unsigned long   rx_fifo_errors;
173         unsigned long   rx_missed_errors;
174         unsigned long   tx_aborted_errors;
175         unsigned long   tx_carrier_errors;
176         unsigned long   tx_fifo_errors;
177         unsigned long   tx_heartbeat_errors;
178         unsigned long   tx_window_errors;
179         unsigned long   rx_compressed;
180         unsigned long   tx_compressed;
181 };
182
183
184 #include <linux/cache.h>
185 #include <linux/skbuff.h>
186
187 #ifdef CONFIG_RPS
188 #include <linux/static_key.h>
189 extern struct static_key rps_needed;
190 #endif
191
192 struct neighbour;
193 struct neigh_parms;
194 struct sk_buff;
195
196 struct netdev_hw_addr {
197         struct list_head        list;
198         unsigned char           addr[MAX_ADDR_LEN];
199         unsigned char           type;
```

```
200 #define NETDEV_HW_ADDR_T_LAN            1
201 #define NETDEV_HW_ADDR_T_SAN            2
202 #define NETDEV_HW_ADDR_T_SLAVE          3
203 #define NETDEV_HW_ADDR_T_UNICAST        4
204 #define NETDEV_HW_ADDR_T_MULTICAST      5
205         bool                    global_use;
206         int                     sync_cnt;
207         int                     refcount;
208         int                     synced;
209         struct rcu_head         rcu_head;
210 };
211
212 struct netdev_hw_addr_list {
213         struct list_head        list;
214         int                     count;
215 };
216
217 #define netdev_hw_addr_list_count(l) ((l)->count)
218 #define netdev_hw_addr_list_empty(l) (netdev_hw_addr_list_count(l) == 0)
219 #define netdev_hw_addr_list_for_each(ha, l) \
220         list_for_each_entry(ha, &(l)->list, list)
221
222 #define netdev_uc_count(dev) netdev_hw_addr_list_count(&(dev)->uc)
223 #define netdev_uc_empty(dev) netdev_hw_addr_list_empty(&(dev)->uc)
224 #define netdev_for_each_uc_addr(ha, dev) \
225         netdev_hw_addr_list_for_each(ha, &(dev)->uc)
226
227 #define netdev_mc_count(dev) netdev_hw_addr_list_count(&(dev)->mc)
228 #define netdev_mc_empty(dev) netdev_hw_addr_list_empty(&(dev)->mc)
229 #define netdev_for_each_mc_addr(ha, dev) \
230         netdev_hw_addr_list_for_each(ha, &(dev)->mc)
231
232 struct hh_cache {
233         u16             hh_len;
234         u16             __pad;
235         seqlock_t       hh_lock;
236
237         /* cached hardware header; allow for machine alignment needs.        */
238 #define HH_DATA_MOD     16
239 #define HH_DATA_OFF(__len) \
240         (HH_DATA_MOD - (((__len - 1) & (HH_DATA_MOD - 1)) + 1))
241 #define HH_DATA_ALIGN(__len) \
242         (((__len)+(HH_DATA_MOD-1))&~(HH_DATA_MOD - 1))
243         unsigned long   hh_data[HH_DATA_ALIGN(LL_MAX_HEADER) / sizeof(long)];
244 };
245
246 /* Reserve HH_DATA_MOD byte aligned hard_header_len, but at least that much.
247  * Alternative is:
248  *   dev->hard_header_len ? (dev->hard_header_len +
249  *                           (HH_DATA_MOD - 1)) & ~(HH_DATA_MOD - 1) : 0
250  *
251  * We could use other alignment values, but we must maintain the
252  * relationship HH alignment <= LL alignment.
253  */
254 #define LL_RESERVED_SPACE(dev) \
255         ((((dev)->hard_header_len+(dev)->needed_headroom)&~(HH_DATA_MOD - 1)) + HH_DATA_MOD)
256 #define LL_RESERVED_SPACE_EXTRA(dev,extra) \
257         ((((dev)->hard_header_len+(dev)->needed_headroom+(extra))&~(HH_DATA_MOD - 1)) + HH_DATA_MOD)
258
259 struct header_ops {
260         int     (*create) (struct sk_buff *skb, struct net_device *dev,
261                            unsigned short type, const void *daddr,
262                            const void *saddr, unsigned int len);
263         int     (*parse)(const struct sk_buff *skb, unsigned char *haddr);
264         int     (*cache)(const struct neighbour *neigh, struct hh_cache *hh, __be16 type);
265         void    (*cache_update)(struct hh_cache *hh,
266                                 const struct net_device *dev,
267                                 const unsigned char *haddr);
268 };
269
270 /* These flag bits are private to the generic network queueing
271  * layer, they may not be explicitly referenced by any other
272  * code.
```

```
273  */
274
275 enum netdev_state_t {
276         __LINK_STATE_START,
277         __LINK_STATE_PRESENT,
278         __LINK_STATE_NOCARRIER,
279         __LINK_STATE_LINKWATCH_PENDING,
280         __LINK_STATE_DORMANT,
281 };
282
283
284 /*
285  * This structure holds at boot time configured netdevice settings. They
286  * are then used in the device probing.
287  */
288 struct netdev_boot_setup {
289         char name[IFNAMSIZ];
290         struct ifmap map;
291 };
292 #define NETDEV_BOOT_SETUP_MAX 8
293
294 int __init netdev_boot_setup(char *str);
295
296 /*
297  * Structure for NAPI scheduling similar to tasklet but with weighting
298  */
299 struct napi_struct {
300         /* The poll_list must only be managed by the entity which
301          * changes the state of the NAPI_STATE_SCHED bit.  This means
302          * whoever atomically sets that bit can add this napi_struct
303          * to the per-cpu poll_list, and whoever clears that bit
304          * can remove from the list right before clearing the bit.
305          */
306         struct list_head        poll_list;
307
308         unsigned long           state;
309         int                     weight;
310         unsigned int            gro_count;
311         int                     (*poll)(struct napi_struct *, int);
312 #ifdef CONFIG_NETPOLL
313         spinlock_t              poll_lock;
314         int                     poll_owner;
315 #endif
316         struct net_device       *dev;
317         struct sk_buff          *gro_list;
318         struct sk_buff          *skb;
319         struct hrtimer          timer;
320         struct list_head        dev_list;
321         struct hlist_node       napi_hash_node;
322         unsigned int            napi_id;
323 };
324
325 enum {
326         NAPI_STATE_SCHED,       /* Poll is scheduled */
327         NAPI_STATE_DISABLE,     /* Disable pending */
328         NAPI_STATE_NPSVC,       /* Netpoll - don't dequeue from poll_list */
329         NAPI_STATE_HASHED,      /* In NAPI hash */
330 };
331
332 enum gro_result {
333         GRO_MERGED,
334         GRO_MERGED_FREE,
335         GRO_HELD,
336         GRO_NORMAL,
337         GRO_DROP,
338 };
339 typedef enum gro_result gro_result_t;
340
341 /*
342  * enum rx_handler_result - Possible return values for rx_handlers.
343  * @RX_HANDLER_CONSUMED: skb was consumed by rx_handler, do not process it
344  * further.
345  * @RX_HANDLER_ANOTHER: Do another round in receive path. This is indicated in
```

```
346  * case skb->dev was changed by rx_handler.
347  * @RX_HANDLER_EXACT: Force exact delivery, no wildcard.
348  * @RX_HANDLER_PASS: Do nothing, passe the skb as if no rx_handler was called.
349  *
350  * rx_handlers are functions called from inside __netif_receive_skb(), to do
351  * special processing of the skb, prior to delivery to protocol handlers.
352  *
353  * Currently, a net_device can only have a single rx_handler registered. Trying
354  * to register a second rx_handler will return -EBUSY.
355  *
356  * To register a rx_handler on a net_device, use netdev_rx_handler_register().
357  * To unregister a rx_handler on a net_device, use
358  * netdev_rx_handler_unregister().
359  *
360  * Upon return, rx_handler is expected to tell __netif_receive_skb() what to
361  * do with the skb.
362  *
363  * If the rx_handler consumed to skb in some way, it should return
364  * RX_HANDLER_CONSUMED. This is appropriate when the rx_handler arranged for
365  * the skb to be delivered in some other ways.
366  *
367  * If the rx_handler changed skb->dev, to divert the skb to another
368  * net_device, it should return RX_HANDLER_ANOTHER. The rx_handler for the
369  * new device will be called if it exists.
370  *
371  * If the rx_handler consider the skb should be ignored, it should return
372  * RX_HANDLER_EXACT. The skb will only be delivered to protocol handlers that
373  * are registered on exact device (ptype->dev == skb->dev).
374  *
375  * If the rx_handler didn't changed skb->dev, but want the skb to be normally
376  * delivered, it should return RX_HANDLER_PASS.
377  *
378  * A device without a registered rx_handler will behave as if rx_handler
379  * returned RX_HANDLER_PASS.
380  */
381
382 enum rx_handler_result {
383         RX_HANDLER_CONSUMED,
384         RX_HANDLER_ANOTHER,
385         RX_HANDLER_EXACT,
386         RX_HANDLER_PASS,
387 };
388 typedef enum rx_handler_result rx_handler_result_t;
389 typedef rx_handler_result_t rx_handler_func_t(struct sk_buff **pskb);
390
391 void __napi_schedule(struct napi_struct *n);
392 void __napi_schedule_irqoff(struct napi_struct *n);
393
394 static inline bool napi_disable_pending(struct napi_struct *n)
395 {
396         return test_bit(NAPI_STATE_DISABLE, &n->state);
397 }
398
399 /**
400  *      napi_schedule_prep - check if napi can be scheduled
401  *      @n: napi context
402  *
403  * Test if NAPI routine is already running, and if not mark
404  * it as running.  This is used as a condition variable
405  * insure only one NAPI poll instance runs.  We also make
406  * sure there is no pending NAPI disable.
407  */
408 static inline bool napi_schedule_prep(struct napi_struct *n)
409 {
410         return !napi_disable_pending(n) &&
411                 !test_and_set_bit(NAPI_STATE_SCHED, &n->state);
412 }
413
414 /**
415  *      napi_schedule - schedule NAPI poll
416  *      @n: napi context
417  *
418  * Schedule NAPI poll routine to be called if it is not already
```

```
419  * running.
420  */
421 static inline void napi_schedule(struct napi_struct *n)
422 {
423         if (napi_schedule_prep(n))
424                 __napi_schedule(n);
425 }
426
427 /**
428  *      napi_schedule_irqoff - schedule NAPI poll
429  *      @n: napi context
430  *
431  * Variant of napi_schedule(), assuming hard irqs are masked.
432  */
433 static inline void napi_schedule_irqoff(struct napi_struct *n)
434 {
435         if (napi_schedule_prep(n))
436                 __napi_schedule_irqoff(n);
437 }
438
439 /* Try to reschedule poll. Called by dev->poll() after napi_complete().  */
440 static inline bool napi_reschedule(struct napi_struct *napi)
441 {
442         if (napi_schedule_prep(napi)) {
443                 __napi_schedule(napi);
444                 return true;
445         }
446         return false;
447 }
448
449 void __napi_complete(struct napi_struct *n);
450 void napi_complete_done(struct napi_struct *n, int work_done);
451 /**
452  *      napi_complete - NAPI processing complete
453  *      @n: napi context
454  *
455  * Mark NAPI processing as complete.
456  * Consider using napi_complete_done() instead.
457  */
458 static inline void napi_complete(struct napi_struct *n)
459 {
460         return napi_complete_done(n, 0);
461 }
462
463 /**
464  *      napi_by_id - lookup a NAPI by napi_id
465  *      @napi_id: hashed napi_id
466  *
467  * lookup @napi_id in napi_hash table
468  * must be called under rcu_read_lock()
469  */
470 struct napi_struct *napi_by_id(unsigned int napi_id);
471
472 /**
473  *      napi_hash_add - add a NAPI to global hashtable
474  *      @napi: napi context
475  *
476  * generate a new napi_id and store a @napi under it in napi_hash
477  */
478 void napi_hash_add(struct napi_struct *napi);
479
480 /**
481  *      napi_hash_del - remove a NAPI from global table
482  *      @napi: napi context
483  *
484  * Warning: caller must observe rcu grace period
485  * before freeing memory containing @napi
486  */
487 void napi_hash_del(struct napi_struct *napi);
488
489 /**
490  *      napi_disable - prevent NAPI from scheduling
491  *      @n: napi context
```

```
492  *
493  * Stop NAPI from being scheduled on this context.
494  * Waits till any outstanding processing completes.
495  */
496 void napi_disable(struct napi_struct *n);
497
498 /**
499  *      napi_enable - enable NAPI scheduling
500  *      @n: napi context
501  *
502  * Resume NAPI from being scheduled on this context.
503  * Must be paired with napi_disable.
504  */
505 static inline void napi_enable(struct napi_struct *n)
506 {
507         BUG_ON(!test_bit(NAPI_STATE_SCHED, &n->state));
508         smp_mb__before_atomic();
509         clear_bit(NAPI_STATE_SCHED, &n->state);
510 }
511
512 #ifdef CONFIG_SMP
513 /**
514  *      napi_synchronize - wait until NAPI is not running
515  *      @n: napi context
516  *
517  * Wait until NAPI is done being scheduled on this context.
518  * Waits till any outstanding processing completes but
519  * does not disable future activations.
520  */
521 static inline void napi_synchronize(const struct napi_struct *n)
522 {
523         while (test_bit(NAPI_STATE_SCHED, &n->state))
524                 msleep(1);
525 }
526 #else
527 # define napi_synchronize(n)    barrier()
528 #endif
529
530 enum netdev_queue_state_t {
531         __QUEUE_STATE_DRV_XOFF,
532         __QUEUE_STATE_STACK_XOFF,
533         __QUEUE_STATE_FROZEN,
534 };
535
536 #define QUEUE_STATE_DRV_XOFF    (1 << __QUEUE_STATE_DRV_XOFF)
537 #define QUEUE_STATE_STACK_XOFF  (1 << __QUEUE_STATE_STACK_XOFF)
538 #define QUEUE_STATE_FROZEN      (1 << __QUEUE_STATE_FROZEN)
539
540 #define QUEUE_STATE_ANY_XOFF    (QUEUE_STATE_DRV_XOFF | QUEUE_STATE_STACK_XOFF)
541 #define QUEUE_STATE_ANY_XOFF_OR_FROZEN (QUEUE_STATE_ANY_XOFF | \
542                                         QUEUE_STATE_FROZEN)
543 #define QUEUE_STATE_DRV_XOFF_OR_FROZEN (QUEUE_STATE_DRV_XOFF | \
544                                         QUEUE_STATE_FROZEN)
545
546 /*
547  * __QUEUE_STATE_DRV_XOFF is used by drivers to stop the transmit queue.  The
548  * netif_tx_* functions below are used to manipulate this flag.  The
549  * __QUEUE_STATE_STACK_XOFF flag is used by the stack to stop the transmit
550  * queue independently.  The netif_xmit_*stopped functions below are called
551  * to check if the queue has been stopped by the driver or stack (either
552  * of the XOFF bits are set in the state).  Drivers should not need to call
553  * netif_xmit*stopped functions, they should only be using netif_tx_*.
554  */
555
556 struct netdev_queue {
557 /*
558  * read mostly part
559  */
560         struct net_device       *dev;
561         struct Qdisc __rcu      *qdisc;
562         struct Qdisc            *qdisc_sleeping;
563 #ifdef CONFIG_SYSFS
564         struct kobject          kobj;
```

```
565 #endif
566 #if defined(CONFIG_XPS) && defined(CONFIG_NUMA)
567         int                     numa_node;
568 #endif
569 /*
570  * write mostly part
571  */
572         spinlock_t              _xmit_lock ____cacheline_aligned_in_smp;
573         int                     xmit_lock_owner;
574         /*
575          * please use this field instead of dev->trans_start
576          */
577         unsigned long           trans_start;
578
579         /*
580          * Number of TX timeouts for this queue
581          * (/sys/class/net/DEV/Q/trans_timeout)
582          */
583         unsigned long           trans_timeout;
584
585         unsigned long           state;
586
587 #ifdef CONFIG_BQL
588         struct dql              dql;
589 #endif
590         unsigned long           tx_maxrate;
591 } ____cacheline_aligned_in_smp;
592
593 static inline int netdev_queue_numa_node_read(const struct netdev_queue *q)
594 {
595 #if defined(CONFIG_XPS) && defined(CONFIG_NUMA)
596         return q->numa_node;
597 #else
598         return NUMA_NO_NODE;
599 #endif
600 }
601
602 static inline void netdev_queue_numa_node_write(struct netdev_queue *q, int node)
603 {
604 #if defined(CONFIG_XPS) && defined(CONFIG_NUMA)
605         q->numa_node = node;
606 #endif
607 }
608
609 #ifdef CONFIG_RPS
610 /*
611  * This structure holds an RPS map which can be of variable length.  The
612  * map is an array of CPUs.
613  */
614 struct rps_map {
615         unsigned int len;
616         struct rcu_head rcu;
617         u16 cpus[0];
618 };
619 #define RPS_MAP_SIZE(_num) (sizeof(struct rps_map) + ((_num) * sizeof(u16)))
620
621 /*
622  * The rps_dev_flow structure contains the mapping of a flow to a CPU, the
623  * tail pointer for that CPU's input queue at the time of last enqueue, and
624  * a hardware filter index.
625  */
626 struct rps_dev_flow {
627         u16 cpu;
628         u16 filter;
629         unsigned int last_qtail;
630 };
631 #define RPS_NO_FILTER 0xffff
632
633 /*
634  * The rps_dev_flow_table structure contains a table of flow mappings.
635  */
636 struct rps_dev_flow_table {
637         unsigned int mask;
```

```
638         struct rcu_head rcu;
639         struct rps_dev_flow flows[0];
640 };
641 #define RPS_DEV_FLOW_TABLE_SIZE(_num) (sizeof(struct rps_dev_flow_table) + \
642     ((_num) * sizeof(struct rps_dev_flow)))
643
644 /*
645  * The rps_sock_flow_table contains mappings of flows to the last CPU
646  * on which they were processed by the application (set in recvmsg).
647  * Each entry is a 32bit value. Upper part is the high order bits
648  * of flow hash, lower part is cpu number.
649  * rps_cpu_mask is used to partition the space, depending on number of
650  * possible cpus : rps_cpu_mask = roundup_pow_of_two(nr_cpu_ids) - 1
651  * For example, if 64 cpus are possible, rps_cpu_mask = 0x3f,
652  * meaning we use 32-6=26 bits for the hash.
653  */
654 struct rps_sock_flow_table {
655         u32     mask;
656
657         u32     ents[0] ____cacheline_aligned_in_smp;
658 };
659 #define RPS_SOCK_FLOW_TABLE_SIZE(_num) (offsetof(struct rps_sock_flow_table, ents[_num]))
660
661 #define RPS_NO_CPU 0xffff
662
663 extern u32 rps_cpu_mask;
664 extern struct rps_sock_flow_table __rcu *rps_sock_flow_table;
665
666 static inline void rps_record_sock_flow(struct rps_sock_flow_table *table,
667                                         u32 hash)
668 {
669         if (table && hash) {
670                 unsigned int index = hash & table->mask;
671                 u32 val = hash & ~rps_cpu_mask;
672
673                 /* We only give a hint, preemption can change cpu under us */
674                 val |= raw_smp_processor_id();
675
676                 if (table->ents[index] != val)
677                         table->ents[index] = val;
678         }
679 }
680
681 #ifdef CONFIG_RFS_ACCEL
682 bool rps_may_expire_flow(struct net_device *dev, u16 rxq_index, u32 flow_id,
683                          u16 filter_id);
684 #endif
685 #endif /* CONFIG_RPS */
686
687 /* This structure contains an instance of an RX queue. */
688 struct netdev_rx_queue {
689 #ifdef CONFIG_RPS
690         struct rps_map __rcu            *rps_map;
691         struct rps_dev_flow_table __rcu *rps_flow_table;
692 #endif
693         struct kobject                  kobj;
694         struct net_device               *dev;
695 } ____cacheline_aligned_in_smp;
696
697 /*
698  * RX queue sysfs structures and functions.
699  */
700 struct rx_queue_attribute {
701         struct attribute attr;
702         ssize_t (*show)(struct netdev_rx_queue *queue,
703             struct rx_queue_attribute *attr, char *buf);
704         ssize_t (*store)(struct netdev_rx_queue *queue,
705             struct rx_queue_attribute *attr, const char *buf, size_t len);
706 };
707
708 #ifdef CONFIG_XPS
709 /*
710  * This structure holds an XPS map which can be of variable length.  The
```

```
711  * map is an array of queues.
712  */
713 struct xps_map {
714         unsigned int len;
715         unsigned int alloc_len;
716         struct rcu_head rcu;
717         u16 queues[0];
718 };
719 #define XPS_MAP_SIZE(_num) (sizeof(struct xps_map) + ((_num) * sizeof(u16)))
720 #define XPS_MIN_MAP_ALLOC ((L1_CACHE_BYTES - sizeof(struct xps_map))    \
721     / sizeof(u16))
722
723 /*
724  * This structure holds all XPS maps for device.  Maps are indexed by CPU.
725  */
726 struct xps_dev_maps {
727         struct rcu_head rcu;
728         struct xps_map __rcu *cpu_map[0];
729 };
730 #define XPS_DEV_MAPS_SIZE (sizeof(struct xps_dev_maps) +                \
731     (nr_cpu_ids * sizeof(struct xps_map *)))
732 #endif /* CONFIG_XPS */
733
734 #define TC_MAX_QUEUE    16
735 #define TC_BITMASK      15
736 /* HW offloaded queuing disciplines txq count and offset maps */
737 struct netdev_tc_txq {
738         u16 count;
739         u16 offset;
740 };
741
742 #if defined(CONFIG_FCOE) || defined(CONFIG_FCOE_MODULE)
743 /*
744  * This structure is to hold information about the device
745  * configured to run FCoE protocol stack.
746  */
747 struct netdev_fcoe_hbainfo {
748         char    manufacturer[64];
749         char    serial_number[64];
750         char    hardware_version[64];
751         char    driver_version[64];
752         char    optionrom_version[64];
753         char    firmware_version[64];
754         char    model[256];
755         char    model_description[256];
756 };
757 #endif
758
759 #define MAX_PHYS_ITEM_ID_LEN 32
760
761 /* This structure holds a unique identifier to identify some
762  * physical item (port for example) used by a netdevice.
763  */
764 struct netdev_phys_item_id {
765         unsigned char id[MAX_PHYS_ITEM_ID_LEN];
766         unsigned char id_len;
767 };
768
769 typedef u16 (*select_queue_fallback_t)(struct net_device *dev,
770                                        struct sk_buff *skb);
771
772 /*
773  * This structure defines the management hooks for network devices.
774  * The following hooks can be defined; unless noted otherwise, they are
775  * optional and can be filled with a null pointer.
776  *
777  * int (*ndo_init)(struct net_device *dev);
778  *     This function is called once when network device is registered.
779  *     The network device can use this to any late stage initializaton
780  *     or semantic validattion. It can fail with an error code which will
781  *     be propogated back to register_netdev
782  *
783  * void (*ndo_uninit)(struct net_device *dev);
```

```
784  *     This function is called when device is unregistered or when registration
785  *     fails. It is not called if init fails.
786  *
787  * int (*ndo_open)(struct net_device *dev);
788  *     This function is called when network device transistions to the up
789  *     state.
790  *
791  * int (*ndo_stop)(struct net_device *dev);
792  *     This function is called when network device transistions to the down
793  *     state.
794  *
795  * netdev_tx_t (*ndo_start_xmit)(struct sk_buff *skb,
796  *                               struct net_device *dev);
797  *      Called when a packet needs to be transmitted.
798  *      Returns NETDEV_TX_OK.  Can return NETDEV_TX_BUSY, but you should stop
799  *      the queue before that can happen; it's for obsolete devices and weird
800  *      corner cases, but the stack really does a non-trivial amount
801  *      of useless work if you return NETDEV_TX_BUSY.
802  *        (can also return NETDEV_TX_LOCKED iff NETIF_F_LLTX)
803  *      Required can not be NULL.
804  *
805  * u16 (*ndo_select_queue)(struct net_device *dev, struct sk_buff *skb,
806  *                         void *accel_priv, select_queue_fallback_t fallback);
807  *      Called to decide which queue to when device supports multiple
808  *      transmit queues.
809  *
810  * void (*ndo_change_rx_flags)(struct net_device *dev, int flags);
811  *      This function is called to allow device receiver to make
812  *      changes to configuration when multicast or promiscious is enabled.
813  *
814  * void (*ndo_set_rx_mode)(struct net_device *dev);
815  *      This function is called device changes address list filtering.
816  *      If driver handles unicast address filtering, it should set
817  *      IFF_UNICAST_FLT to its priv_flags.
818  *
819  * int (*ndo_set_mac_address)(struct net_device *dev, void *addr);
820  *      This function  is called when the Media Access Control address
821  *      needs to be changed. If this interface is not defined, the
822  *      mac address can not be changed.
823  *
824  * int (*ndo_validate_addr)(struct net_device *dev);
825  *      Test if Media Access Control address is valid for the device.
826  *
827  * int (*ndo_do_ioctl)(struct net_device *dev, struct ifreq *ifr, int cmd);
828  *      Called when a user request an ioctl which can't be handled by
829  *      the generic interface code. If not defined ioctl's return
830  *      not supported error code.
831  *
832  * int (*ndo_set_config)(struct net_device *dev, struct ifmap *map);
833  *      Used to set network devices bus interface parameters. This interface
834  *      is retained for legacy reason, new devices should use the bus
835  *      interface (PCI) for low level management.
836  *
837  * int (*ndo_change_mtu)(struct net_device *dev, int new_mtu);
838  *      Called when a user wants to change the Maximum Transfer Unit
839  *      of a device. If not defined, any request to change MTU will
840  *      will return an error.
841  *
842  * void (*ndo_tx_timeout)(struct net_device *dev);
843  *      Callback uses when the transmitter has not made any progress
844  *      for dev->watchdog ticks.
845  *
846  * struct rtnl_link_stats64* (*ndo_get_stats64)(struct net_device *dev,
847  *                     struct rtnl_link_stats64 *storage);
848  * struct net_device_stats* (*ndo_get_stats)(struct net_device *dev);
849  *      Called when a user wants to get the network device usage
850  *      statistics. Drivers must do one of the following:
851  *      1. Define @ndo_get_stats64 to fill in a zero-initialised
852  *         rtnl_link_stats64 structure passed by the caller.
853  *      2. Define @ndo_get_stats to update a net_device_stats structure
854  *         (which should normally be dev->stats) and return a pointer to
855  *         it. The structure may be changed asynchronously only if each
856  *         field is written atomically.
```

```
857  *      3. Update dev->stats asynchronously and atomically, and define
858  *         neither operation.
859  *
860  * int (*ndo_vlan_rx_add_vid)(struct net_device *dev, __be16 proto, u16 vid);
861  *      If device support VLAN filtering this function is called when a
862  *      VLAN id is registered.
863  *
864  * int (*ndo_vlan_rx_kill_vid)(struct net_device *dev, __be16 proto, u16 vid);
865  *      If device support VLAN filtering this function is called when a
866  *      VLAN id is unregistered.
867  *
868  * void (*ndo_poll_controller)(struct net_device *dev);
869  *
870  *      SR-IOV management functions.
871  * int (*ndo_set_vf_mac)(struct net_device *dev, int vf, u8* mac);
872  * int (*ndo_set_vf_vlan)(struct net_device *dev, int vf, u16 vlan, u8 qos);
873  * int (*ndo_set_vf_rate)(struct net_device *dev, int vf, int min_tx_rate,
874  *                        int max_tx_rate);
875  * int (*ndo_set_vf_spoofchk)(struct net_device *dev, int vf, bool setting);
876  * int (*ndo_get_vf_config)(struct net_device *dev,
877  *                          int vf, struct ifla_vf_info *ivf);
878  * int (*ndo_set_vf_link_state)(struct net_device *dev, int vf, int link_state);
879  * int (*ndo_set_vf_port)(struct net_device *dev, int vf,
880  *                        struct nlattr *port[]);
881  *
882  *      Enable or disable the VF ability to query its RSS Redirection Table and
883  *      Hash Key. This is needed since on some devices VF share this information
884  *      with PF and querying it may adduce a theoretical security risk.
885  * int (*ndo_set_vf_rss_query_en)(struct net_device *dev, int vf, bool setting);
886  * int (*ndo_get_vf_port)(struct net_device *dev, int vf, struct sk_buff *skb);
887  * int (*ndo_setup_tc)(struct net_device *dev, u8 tc)
888  *      Called to setup 'tc' number of traffic classes in the net device. This
889  *      is always called from the stack with the rtnl lock held and netif tx
890  *      queues stopped. This allows the netdevice to perform queue management
891  *      safely.
892  *
893  *      Fiber Channel over Ethernet (FCoE) offload functions.
894  * int (*ndo_fcoe_enable)(struct net_device *dev);
895  *      Called when the FCoE protocol stack wants to start using LLD for FCoE
896  *      so the underlying device can perform whatever needed configuration or
897  *      initialization to support acceleration of FCoE traffic.
898  *
899  * int (*ndo_fcoe_disable)(struct net_device *dev);
900  *      Called when the FCoE protocol stack wants to stop using LLD for FCoE
901  *      so the underlying device can perform whatever needed clean-ups to
902  *      stop supporting acceleration of FCoE traffic.
903  *
904  * int (*ndo_fcoe_ddp_setup)(struct net_device *dev, u16 xid,
905  *                           struct scatterlist *sgl, unsigned int sgc);
906  *      Called when the FCoE Initiator wants to initialize an I/O that
907  *      is a possible candidate for Direct Data Placement (DDP). The LLD can
908  *      perform necessary setup and returns 1 to indicate the device is set up
909  *      successfully to perform DDP on this I/O, otherwise this returns 0.
910  *
911  * int (*ndo_fcoe_ddp_done)(struct net_device *dev,  u16 xid);
912  *      Called when the FCoE Initiator/Target is done with the DDPed I/O as
913  *      indicated by the FC exchange id 'xid', so the underlying device can
914  *      clean up and reuse resources for later DDP requests.
915  *
916  * int (*ndo_fcoe_ddp_target)(struct net_device *dev, u16 xid,
917  *                            struct scatterlist *sgl, unsigned int sgc);
918  *      Called when the FCoE Target wants to initialize an I/O that
919  *      is a possible candidate for Direct Data Placement (DDP). The LLD can
920  *      perform necessary setup and returns 1 to indicate the device is set up
921  *      successfully to perform DDP on this I/O, otherwise this returns 0.
922  *
923  * int (*ndo_fcoe_get_hbainfo)(struct net_device *dev,
924  *                             struct netdev_fcoe_hbainfo *hbainfo);
925  *      Called when the FCoE Protocol stack wants information on the underlying
926  *      device. This information is utilized by the FCoE protocol stack to
927  *      register attributes with Fiber Channel management service as per the
928  *      FC-GS Fabric Device Management Information(FDMI) specification.
929  *
```

```
930  * int (*ndo_fcoe_get_wwn)(struct net_device *dev, u64 *wwn, int type);
931  *      Called when the underlying device wants to override default World Wide
932  *      Name (WWN) generation mechanism in FCoE protocol stack to pass its own
933  *      World Wide Port Name (WWPN) or World Wide Node Name (WWNN) to the FCoE
934  *      protocol stack to use.
935  *
936  *      RFS acceleration.
937  * int (*ndo_rx_flow_steer)(struct net_device *dev, const struct sk_buff *skb,
938  *                          u16 rxq_index, u32 flow_id);
939  *      Set hardware filter for RFS.  rxq_index is the target queue index;
940  *      flow_id is a flow ID to be passed to rps_may_expire_flow() later.
941  *      Return the filter ID on success, or a negative error code.
942  *
943  *      Slave management functions (for bridge, bonding, etc).
944  * int (*ndo_add_slave)(struct net_device *dev, struct net_device *slave_dev);
945  *      Called to make another netdev an underling.
946  *
947  * int (*ndo_del_slave)(struct net_device *dev, struct net_device *slave_dev);
948  *      Called to release previously enslaved netdev.
949  *
950  *      Feature/offload setting functions.
951  * netdev_features_t (*ndo_fix_features)(struct net_device *dev,
952  *              netdev_features_t features);
953  *      Adjusts the requested feature flags according to device-specific
954  *      constraints, and returns the resulting flags. Must not modify
955  *      the device state.
956  *
957  * int (*ndo_set_features)(struct net_device *dev, netdev_features_t features);
958  *      Called to update device configuration to new features. Passed
959  *      feature set might be less than what was returned by ndo_fix_features()).
960  *      Must return >0 or -errno if it changed dev->features itself.
961  *
962  * int (*ndo_fdb_add)(struct ndmsg *ndm, struct nlattr *tb[],
963  *                    struct net_device *dev,
964  *                    const unsigned char *addr, u16 vid, u16 flags)
965  *      Adds an FDB entry to dev for addr.
966  * int (*ndo_fdb_del)(struct ndmsg *ndm, struct nlattr *tb[],
967  *                    struct net_device *dev,
968  *                    const unsigned char *addr, u16 vid)
969  *      Deletes the FDB entry from dev coresponding to addr.
970  * int (*ndo_fdb_dump)(struct sk_buff *skb, struct netlink_callback *cb,
971  *                     struct net_device *dev, struct net_device *filter_dev,
972  *                     int idx)
973  *      Used to add FDB entries to dump requests. Implementers should add
974  *      entries to skb and update idx with the number of entries.
975  *
976  * int (*ndo_bridge_setlink)(struct net_device *dev, struct nlmsghdr *nlh,
977  *                           u16 flags)
978  * int (*ndo_bridge_getlink)(struct sk_buff *skb, u32 pid, u32 seq,
979  *                           struct net_device *dev, u32 filter_mask,
980  *                           int nlflags)
981  * int (*ndo_bridge_dellink)(struct net_device *dev, struct nlmsghdr *nlh,
982  *                           u16 flags);
983  *
984  * int (*ndo_change_carrier)(struct net_device *dev, bool new_carrier);
985  *      Called to change device carrier. Soft-devices (like dummy, team, etc)
986  *      which do not represent real hardware may define this to allow their
987  *      userspace components to manage their virtual carrier state. Devices
988  *      that determine carrier state from physical hardware properties (eg
989  *      network cables) or protocol-dependent mechanisms (eg
990  *      USB_CDC_NOTIFY_NETWORK_CONNECTION) should NOT implement this function.
991  *
992  * int (*ndo_get_phys_port_id)(struct net_device *dev,
993  *                             struct netdev_phys_item_id *ppid);
994  *      Called to get ID of physical port of this device. If driver does
995  *      not implement this, it is assumed that the hw is not able to have
996  *      multiple net devices on single physical port.
997  *
998  * void (*ndo_add_vxlan_port)(struct  net_device *dev,
999  *                            sa_family_t sa_family, __be16 port);
1000 *      Called by vxlan to notiy a driver about the UDP port and socket
1001 *      address family that vxlan is listnening to. It is called only when
1002 *      a new port starts listening. The operation is protected by the
```

```
1003  *      vxlan_net->sock_lock.
1004  *
1005  * void (*ndo_del_vxlan_port)(struct  net_device *dev,
1006  *                            sa_family_t sa_family, __be16 port);
1007  *      Called by vxlan to notify the driver about a UDP port and socket
1008  *      address family that vxlan is not listening to anymore. The operation
1009  *      is protected by the vxlan_net->sock_lock.
1010  *
1011  * void* (*ndo_dfwd_add_station)(struct net_device *pdev,
1012  *                               struct net_device *dev)
1013  *      Called by upper layer devices to accelerate switching or other
1014  *      station functionality into hardware. 'pdev is the lowerdev
1015  *      to use for the offload and 'dev' is the net device that will
1016  *      back the offload. Returns a pointer to the private structure
1017  *      the upper layer will maintain.
1018  * void (*ndo_dfwd_del_station)(struct net_device *pdev, void *priv)
1019  *      Called by upper layer device to delete the station created
1020  *      by 'ndo_dfwd_add_station'. 'pdev' is the net device backing
1021  *      the station and priv is the structure returned by the add
1022  *      operation.
1023  * netdev_tx_t (*ndo_dfwd_start_xmit)(struct sk_buff *skb,
1024  *                                    struct net_device *dev,
1025  *                                    void *priv);
1026  *      Callback to use for xmit over the accelerated station. This
1027  *      is used in place of ndo_start_xmit on accelerated net
1028  *      devices.
1029  * netdev_features_t (*ndo_features_check) (struct sk_buff *skb,
1030  *                                          struct net_device *dev
1031  *                                          netdev_features_t features);
1032  *      Called by core transmit path to determine if device is capable of
1033  *      performing offload operations on a given packet. This is to give
1034  *      the device an opportunity to implement any restrictions that cannot
1035  *      be otherwise expressed by feature flags. The check is called with
1036  *      the set of features that the stack has calculated and it returns
1037  *      those the driver believes to be appropriate.
1038  * int (*ndo_set_tx_maxrate)(struct net_device *dev,
1039  *                           int queue_index, u32 maxrate);
1040  *      Called when a user wants to set a max-rate limitation of specific
1041  *      TX queue.
1042  * int (*ndo_get_iflink)(const struct net_device *dev);
1043  *      Called to get the iflink value of this device.
1044  */
1045 struct net_device_ops {
1046         int                     (*ndo_init)(struct net_device *dev);
1047         void                    (*ndo_uninit)(struct net_device *dev);
1048         int                     (*ndo_open)(struct net_device *dev);
1049         int                     (*ndo_stop)(struct net_device *dev);
1050         netdev_tx_t             (*ndo_start_xmit) (struct sk_buff *skb,
1051                                                    struct net_device *dev);
1052         u16                     (*ndo_select_queue)(struct net_device *dev,
1053                                                     struct sk_buff *skb,
1054                                                     void *accel_priv,
1055                                                     select_queue_fallback_t fallback);
1056         void                    (*ndo_change_rx_flags)(struct net_device *dev,
1057                                                        int flags);
1058         void                    (*ndo_set_rx_mode)(struct net_device *dev);
1059         int                     (*ndo_set_mac_address)(struct net_device *dev,
1060                                                        void *addr);
1061         int                     (*ndo_validate_addr)(struct net_device *dev);
1062         int                     (*ndo_do_ioctl)(struct net_device *dev,
1063                                                 struct ifreq *ifr, int cmd);
1064         int                     (*ndo_set_config)(struct net_device *dev,
1065                                                   struct ifmap *map);
1066         int                     (*ndo_change_mtu)(struct net_device *dev,
1067                                                   int new_mtu);
1068         int                     (*ndo_neigh_setup)(struct net_device *dev,
1069                                                    struct neigh_parms *);
1070         void                    (*ndo_tx_timeout) (struct net_device *dev);
1071
1072         struct rtnl_link_stats64* (*ndo_get_stats64)(struct net_device *dev,
1073                                                      struct rtnl_link_stats64 *storage);
1074         struct net_device_stats* (*ndo_get_stats)(struct net_device *dev);
1075
```

```
1076         int                     (*ndo_vlan_rx_add_vid)(struct net_device *dev,
1077                                                        __be16 proto, u16 vid);
1078         int                     (*ndo_vlan_rx_kill_vid)(struct net_device *dev,
1079                                                         __be16 proto, u16 vid);
1080 #ifdef CONFIG_NET_POLL_CONTROLLER
1081         void                    (*ndo_poll_controller)(struct net_device *dev);
1082         int                     (*ndo_netpoll_setup)(struct net_device *dev,
1083                                                      struct netpoll_info *info);
1084         void                    (*ndo_netpoll_cleanup)(struct net_device *dev);
1085 #endif
1086 #ifdef CONFIG_NET_RX_BUSY_POLL
1087         int                     (*ndo_busy_poll)(struct napi_struct *dev);
1088 #endif
1089         int                     (*ndo_set_vf_mac)(struct net_device *dev,
1090                                                   int queue, u8 *mac);
1091         int                     (*ndo_set_vf_vlan)(struct net_device *dev,
1092                                                    int queue, u16 vlan, u8 qos);
1093         int                     (*ndo_set_vf_rate)(struct net_device *dev,
1094                                                    int vf, int min_tx_rate,
1095                                                    int max_tx_rate);
1096         int                     (*ndo_set_vf_spoofchk)(struct net_device *dev,
1097                                                        int vf, bool setting);
1098         int                     (*ndo_get_vf_config)(struct net_device *dev,
1099                                                      int vf,
1100                                                      struct ifla_vf_info *ivf);
1101         int                     (*ndo_set_vf_link_state)(struct net_device *dev,
1102                                                          int vf, int link_state);
1103         int                     (*ndo_get_vf_stats)(struct net_device *dev,
1104                                                     int vf,
1105                                                     struct ifla_vf_stats
1106                                                     *vf_stats);
1107         int                     (*ndo_set_vf_port)(struct net_device *dev,
1108                                                    int vf,
1109                                                    struct nlattr *port[]);
1110         int                     (*ndo_get_vf_port)(struct net_device *dev,
1111                                                    int vf, struct sk_buff *skb);
1112         int                     (*ndo_set_vf_rss_query_en)(
1113                                                    struct net_device *dev,
1114                                                    int vf, bool setting);
1115         int                     (*ndo_setup_tc)(struct net_device *dev, u8 tc);
1116 #if IS_ENABLED(CONFIG_FCOE)
1117         int                     (*ndo_fcoe_enable)(struct net_device *dev);
1118         int                     (*ndo_fcoe_disable)(struct net_device *dev);
1119         int                     (*ndo_fcoe_ddp_setup)(struct net_device *dev,
1120                                                       u16 xid,
1121                                                       struct scatterlist *sgl,
1122                                                       unsigned int sgc);
1123         int                     (*ndo_fcoe_ddp_done)(struct net_device *dev,
1124                                                      u16 xid);
1125         int                     (*ndo_fcoe_ddp_target)(struct net_device *dev,
1126                                                        u16 xid,
1127                                                        struct scatterlist *sgl,
1128                                                        unsigned int sgc);
1129         int                     (*ndo_fcoe_get_hbainfo)(struct net_device *dev,
1130                                                         struct netdev_fcoe_hbainfo *hbainfo);
1131 #endif
1132
1133 #if IS_ENABLED(CONFIG_LIBFCOE)
1134 #define NETDEV_FCOE_WWNN 0
1135 #define NETDEV_FCOE_WWPN 1
1136         int                     (*ndo_fcoe_get_wwn)(struct net_device *dev,
1137                                                     u64 *wwn, int type);
1138 #endif
1139
1140 #ifdef CONFIG_RFS_ACCEL
1141         int                     (*ndo_rx_flow_steer)(struct net_device *dev,
1142                                                      const struct sk_buff *skb,
1143                                                      u16 rxq_index,
1144                                                      u32 flow_id);
1145 #endif
1146         int                     (*ndo_add_slave)(struct net_device *dev,
1147                                                  struct net_device *slave_dev);
1148         int                     (*ndo_del_slave)(struct net_device *dev,
```

```
1149                                                struct net_device *slave_dev);
1150        netdev_features_t       (*ndo_fix_features)(struct net_device *dev,
1151                                                    netdev_features_t features);
1152        int                     (*ndo_set_features)(struct net_device *dev,
1153                                                    netdev_features_t features);
1154        int                     (*ndo_neigh_construct)(struct neighbour *n);
1155        void                    (*ndo_neigh_destroy)(struct neighbour *n);
1156
1157        int                     (*ndo_fdb_add)(struct ndmsg *ndm,
1158                                               struct nlattr *tb[],
1159                                               struct net_device *dev,
1160                                               const unsigned char *addr,
1161                                               u16 vid,
1162                                               u16 flags);
1163        int                     (*ndo_fdb_del)(struct ndmsg *ndm,
1164                                               struct nlattr *tb[],
1165                                               struct net_device *dev,
1166                                               const unsigned char *addr,
1167                                               u16 vid);
1168        int                     (*ndo_fdb_dump)(struct sk_buff *skb,
1169                                                struct netlink_callback *cb,
1170                                                struct net_device *dev,
1171                                                struct net_device *filter_dev,
1172                                                int idx);
1173
1174        int                     (*ndo_bridge_setlink)(struct net_device *dev,
1175                                                      struct nlmsghdr *nlh,
1176                                                      u16 flags);
1177        int                     (*ndo_bridge_getlink)(struct sk_buff *skb,
1178                                                      u32 pid, u32 seq,
1179                                                      struct net_device *dev,
1180                                                      u32 filter_mask,
1181                                                      int nlflags);
1182        int                     (*ndo_bridge_dellink)(struct net_device *dev,
1183                                                      struct nlmsghdr *nlh,
1184                                                      u16 flags);
1185        int                     (*ndo_change_carrier)(struct net_device *dev,
1186                                                      bool new_carrier);
1187        int                     (*ndo_get_phys_port_id)(struct net_device *dev,
1188                                                        struct netdev_phys_item_id *ppid);
1189        int                     (*ndo_get_phys_port_name)(struct net_device *dev,
1190                                                          char *name, size_t len);
1191        void                    (*ndo_add_vxlan_port)(struct  net_device *dev,
1192                                                      sa_family_t sa_family,
1193                                                      __be16 port);
1194        void                    (*ndo_del_vxlan_port)(struct  net_device *dev,
1195                                                      sa_family_t sa_family,
1196                                                      __be16 port);
1197
1198        void*                   (*ndo_dfwd_add_station)(struct net_device *pdev,
1199                                                        struct net_device *dev);
1200        void                    (*ndo_dfwd_del_station)(struct net_device *pdev,
1201                                                        void *priv);
1202
1203        netdev_tx_t             (*ndo_dfwd_start_xmit) (struct sk_buff *skb,
1204                                                        struct net_device *dev,
1205                                                        void *priv);
1206        int                     (*ndo_get_lock_subclass)(struct net_device *dev);
1207        netdev_features_t       (*ndo_features_check) (struct sk_buff *skb,
1208                                                       struct net_device *dev,
1209                                                       netdev_features_t features);
1210        int                     (*ndo_set_tx_maxrate)(struct net_device *dev,
1211                                                      int queue_index,
1212                                                      u32 maxrate);
1213        int                     (*ndo_get_iflink)(const struct net_device *dev);
1214 };
1215
1216 /**
1217  * enum net_device_priv_flags - &struct net_device priv_flags
1218  *
1219  * These are the &struct net_device, they are only set internally
1220  * by drivers and used in the kernel. These flags are invisible to
1221  * userspace, this means that the order of these flags can change
```

```
1222  * during any kernel release.
1223  *
1224  * You should have a pretty good reason to be extending these flags.
1225  *
1226  * @IFF_802_1Q_VLAN: 802.1Q VLAN device
1227  * @IFF_EBRIDGE: Ethernet bridging device
1228  * @IFF_SLAVE_INACTIVE: bonding slave not the curr. active
1229  * @IFF_MASTER_8023AD: bonding master, 802.3ad
1230  * @IFF_MASTER_ALB: bonding master, balance-alb
1231  * @IFF_BONDING: bonding master or slave
1232  * @IFF_SLAVE_NEEDARP: need ARPs for validation
1233  * @IFF_ISATAP: ISATAP interface (RFC4214)
1234  * @IFF_MASTER_ARPMON: bonding master, ARP mon in use
1235  * @IFF_WAN_HDLC: WAN HDLC device
1236  * @IFF_XMIT_DST_RELEASE: dev_hard_start_xmit() is allowed to
1237  *      release skb->dst
1238  * @IFF_DONT_BRIDGE: disallow bridging this ether dev
1239  * @IFF_DISABLE_NETPOLL: disable netpoll at run-time
1240  * @IFF_MACVLAN_PORT: device used as macvlan port
1241  * @IFF_BRIDGE_PORT: device used as bridge port
1242  * @IFF_OVS_DATAPATH: device used as Open vSwitch datapath port
1243  * @IFF_TX_SKB_SHARING: The interface supports sharing skbs on transmit
1244  * @IFF_UNICAST_FLT: Supports unicast filtering
1245  * @IFF_TEAM_PORT: device used as team port
1246  * @IFF_SUPP_NOFCS: device supports sending custom FCS
1247  * @IFF_LIVE_ADDR_CHANGE: device supports hardware address
1248  *      change when it's running
1249  * @IFF_MACVLAN: Macvlan device
1250  */
1251 enum netdev_priv_flags {
1252         IFF_802_1Q_VLAN                 = 1<<0,
1253         IFF_EBRIDGE                     = 1<<1,
1254         IFF_SLAVE_INACTIVE              = 1<<2,
1255         IFF_MASTER_8023AD               = 1<<3,
1256         IFF_MASTER_ALB                  = 1<<4,
1257         IFF_BONDING                     = 1<<5,
1258         IFF_SLAVE_NEEDARP               = 1<<6,
1259         IFF_ISATAP                      = 1<<7,
1260         IFF_MASTER_ARPMON               = 1<<8,
1261         IFF_WAN_HDLC                    = 1<<9,
1262         IFF_XMIT_DST_RELEASE            = 1<<10,
1263         IFF_DONT_BRIDGE                 = 1<<11,
1264         IFF_DISABLE_NETPOLL             = 1<<12,
1265         IFF_MACVLAN_PORT                = 1<<13,
1266         IFF_BRIDGE_PORT                 = 1<<14,
1267         IFF_OVS_DATAPATH                = 1<<15,
1268         IFF_TX_SKB_SHARING              = 1<<16,
1269         IFF_UNICAST_FLT                 = 1<<17,
1270         IFF_TEAM_PORT                   = 1<<18,
1271         IFF_SUPP_NOFCS                  = 1<<19,
1272         IFF_LIVE_ADDR_CHANGE            = 1<<20,
1273         IFF_MACVLAN                     = 1<<21,
1274         IFF_XMIT_DST_RELEASE_PERM       = 1<<22,
1275         IFF_IPVLAN_MASTER               = 1<<23,
1276         IFF_IPVLAN_SLAVE                = 1<<24,
1277 };
1278
1279 #define IFF_802_1Q_VLAN                 IFF_802_1Q_VLAN
1280 #define IFF_EBRIDGE                     IFF_EBRIDGE
1281 #define IFF_SLAVE_INACTIVE              IFF_SLAVE_INACTIVE
1282 #define IFF_MASTER_8023AD               IFF_MASTER_8023AD
1283 #define IFF_MASTER_ALB                  IFF_MASTER_ALB
1284 #define IFF_BONDING                     IFF_BONDING
1285 #define IFF_SLAVE_NEEDARP               IFF_SLAVE_NEEDARP
1286 #define IFF_ISATAP                      IFF_ISATAP
1287 #define IFF_MASTER_ARPMON               IFF_MASTER_ARPMON
1288 #define IFF_WAN_HDLC                    IFF_WAN_HDLC
1289 #define IFF_XMIT_DST_RELEASE            IFF_XMIT_DST_RELEASE
1290 #define IFF_DONT_BRIDGE                 IFF_DONT_BRIDGE
1291 #define IFF_DISABLE_NETPOLL             IFF_DISABLE_NETPOLL
1292 #define IFF_MACVLAN_PORT                IFF_MACVLAN_PORT
1293 #define IFF_BRIDGE_PORT                 IFF_BRIDGE_PORT
1294 #define IFF_OVS_DATAPATH                IFF_OVS_DATAPATH
```

```
1295 #define IFF_TX_SKB_SHARING              IFF_TX_SKB_SHARING
1296 #define IFF_UNICAST_FLT                 IFF_UNICAST_FLT
1297 #define IFF_TEAM_PORT                   IFF_TEAM_PORT
1298 #define IFF_SUPP_NOFCS                  IFF_SUPP_NOFCS
1299 #define IFF_LIVE_ADDR_CHANGE            IFF_LIVE_ADDR_CHANGE
1300 #define IFF_MACVLAN                     IFF_MACVLAN
1301 #define IFF_XMIT_DST_RELEASE_PERM       IFF_XMIT_DST_RELEASE_PERM
1302 #define IFF_IPVLAN_MASTER               IFF_IPVLAN_MASTER
1303 #define IFF_IPVLAN_SLAVE                IFF_IPVLAN_SLAVE
1304
1305 /**
1306  *      struct net_device - The DEVICE structure.
1307  *              Actually, this whole structure is a big mistake.  It mixes I/O
1308  *              data with strictly "high-level" data, and it has to know about
1309  *              almost every data structure used in the INET module.
1310  *
1311  *      @name:  This is the first field of the "visible" part of this structure
1312  *              (i.e. as seen by users in the "Space.c" file).  It is the name
1313  *              of the interface.
1314  *
1315  *      @name_hlist:    Device name hash chain, please keep it close to name[]
1316  *      @ifalias:       SNMP alias
1317  *      @mem_end:       Shared memory end
1318  *      @mem_start:     Shared memory start
1319  *      @base_addr:     Device I/O address
1320  *      @irq:           Device IRQ number
1321  *
1322  *      @carrier_changes:       Stats to monitor carrier on<->off transitions
1323  *
1324  *      @state:         Generic network queuing layer state, see netdev_state_t
1325  *      @dev_list:      The global list of network devices
1326  *      @napi_list:     List entry, that is used for polling napi devices
1327  *      @unreg_list:    List entry, that is used, when we are unregistering the
1328  *                      device, see the function unregister_netdev
1329  *      @close_list:    List entry, that is used, when we are closing the device
1330  *
1331  *      @adj_list:      Directly linked devices, like slaves for bonding
1332  *      @all_adj_list:  All linked devices, *including* neighbours
1333  *      @features:      Currently active device features
1334  *      @hw_features:   User-changeable features
1335  *
1336  *      @wanted_features:       User-requested features
1337  *      @vlan_features:         Mask of features inheritable by VLAN devices
1338  *
1339  *      @hw_enc_features:       Mask of features inherited by encapsulating devices
1340  *                              This field indicates what encapsulation
1341  *                              offloads the hardware is capable of doing,
1342  *                              and drivers will need to set them appropriately.
1343  *
1344  *      @mpls_features: Mask of features inheritable by MPLS
1345  *
1346  *      @ifindex:       interface index
1347  *      @group:         The group, that the device belongs to
1348  *
1349  *      @stats:         Statistics struct, which was left as a legacy, use
1350  *                      rtnl_link_stats64 instead
1351  *
1352  *      @rx_dropped:    Dropped packets by core network,
1353  *                      do not use this in drivers
1354  *      @tx_dropped:    Dropped packets by core network,
1355  *                      do not use this in drivers
1356  *
1357  *      @wireless_handlers:     List of functions to handle Wireless Extensions,
1358  *                              instead of ioctl,
1359  *                              see <net/iw_handler.h> for details.
1360  *      @wireless_data: Instance data managed by the core of wireless extensions
1361  *
1362  *      @netdev_ops:    Includes several pointers to callbacks,
1363  *                      if one wants to override the ndo_*() functions
1364  *      @ethtool_ops:   Management operations
1365  *      @header_ops:    Includes callbacks for creating,parsing,caching,etc
1366  *                      of Layer 2 headers.
1367  *
```

```
1368  *      @flags:         Interface flags (a la BSD)
1369  *      @priv_flags:    Like 'flags' but invisible to userspace,
1370  *                      see if.h for the definitions
1371  *      @gflags:        Global flags ( kept as legacy )
1372  *      @padded:        How much padding added by alloc_netdev()
1373  *      @operstate:     RFC2863 operstate
1374  *      @link_mode:     Mapping policy to operstate
1375  *      @if_port:       Selectable AUI, TP, ...
1376  *      @dma:           DMA channel
1377  *      @mtu:           Interface MTU value
1378  *      @type:          Interface hardware type
1379  *      @hard_header_len: Hardware header length
1380  *
1381  *      @needed_headroom: Extra headroom the hardware may need, but not in all
1382  *                        cases can this be guaranteed
1383  *      @needed_tailroom: Extra tailroom the hardware may need, but not in all
1384  *                        cases can this be guaranteed. Some cases also use
1385  *                        LL_MAX_HEADER instead to allocate the skb
1386  *
1387  *      interface address info:
1388  *
1389  *      @perm_addr:             Permanent hw address
1390  *      @addr_assign_type:      Hw address assignment type
1391  *      @addr_len:              Hardware address length
1392  *      @neigh_priv_len;        Used in neigh_alloc(),
1393  *                              initialized only in atm/clip.c
1394  *      @dev_id:                Used to differentiate devices that share
1395  *                              the same link layer address
1396  *      @dev_port:              Used to differentiate devices that share
1397  *                              the same function
1398  *      @addr_list_lock:        XXX: need comments on this one
1399  *      @uc_promisc:            Counter, that indicates, that promiscuous mode
1400  *                              has been enabled due to the need to listen to
1401  *                              additional unicast addresses in a device that
1402  *                              does not implement ndo_set_rx_mode()
1403  *      @uc:                    unicast mac addresses
1404  *      @mc:                    multicast mac addresses
1405  *      @dev_addrs:             list of device hw addresses
1406  *      @queues_kset:           Group of all Kobjects in the Tx and RX queues
1407  *      @promiscuity:           Number of times, the NIC is told to work in
1408  *                              Promiscuous mode, if it becomes 0 the NIC will
1409  *                              exit from working in Promiscuous mode
1410  *      @allmulti:              Counter, enables or disables allmulticast mode
1411  *
1412  *      @vlan_info:     VLAN info
1413  *      @dsa_ptr:       dsa specific data
1414  *      @tipc_ptr:      TIPC specific data
1415  *      @atalk_ptr:     AppleTalk link
1416  *      @ip_ptr:        IPv4 specific data
1417  *      @dn_ptr:        DECnet specific data
1418  *      @ip6_ptr:       IPv6 specific data
1419  *      @ax25_ptr:      AX.25 specific data
1420  *      @ieee80211_ptr: IEEE 802.11 specific data, assign before registering
1421  *
1422  *      @last_rx:       Time of last Rx
1423  *      @dev_addr:      Hw address (before bcast,
1424  *                      because most packets are unicast)
1425  *
1426  *      @_rx:                   Array of RX queues
1427  *      @num_rx_queues:         Number of RX queues
1428  *                              allocated at register_netdev() time
1429  *      @real_num_rx_queues:    Number of RX queues currently active in device
1430  *
1431  *      @rx_handler:            handler for received packets
1432  *      @rx_handler_data:       XXX: need comments on this one
1433  *      @ingress_queue:         XXX: need comments on this one
1434  *      @broadcast:             hw bcast address
1435  *
1436  *      @rx_cpu_rmap:   CPU reverse-mapping for RX completion interrupts,
1437  *                      indexed by RX queue number. Assigned by driver.
1438  *                      This must only be set if the ndo_rx_flow_steer
1439  *                      operation is defined
1440  *      @index_hlist:           Device index hash chain
```

```
1441  *
1442  *      @_tx:                   Array of TX queues
1443  *      @num_tx_queues:         Number of TX queues allocated at alloc_netdev_mq() time
1444  *      @real_num_tx_queues:    Number of TX queues currently active in device
1445  *      @qdisc:                 Root qdisc from userspace point of view
1446  *      @tx_queue_len:          Max frames per queue allowed
1447  *      @tx_global_lock:        XXX: need comments on this one
1448  *
1449  *      @xps_maps:      XXX: need comments on this one
1450  *
1451  *      @trans_start:           Time (in jiffies) of last Tx
1452  *      @watchdog_timeo:        Represents the timeout that is used by
1453  *                              the watchdog ( see dev_watchdog() )
1454  *      @watchdog_timer:        List of timers
1455  *
1456  *      @pcpu_refcnt:           Number of references to this device
1457  *      @todo_list:             Delayed register/unregister
1458  *      @link_watch_list:       XXX: need comments on this one
1459  *
1460  *      @reg_state:             Register/unregister state machine
1461  *      @dismantle:             Device is going to be freed
1462  *      @rtnl_link_state:       This enum represents the phases of creating
1463  *                              a new link
1464  *
1465  *      @destructor:            Called from unregister,
1466  *                              can be used to call free_netdev
1467  *      @npinfo:                XXX: need comments on this one
1468  *      @nd_net:                Network namespace this network device is inside
1469  *
1470  *      @ml_priv:       Mid-layer private
1471  *      @lstats:        Loopback statistics
1472  *      @tstats:        Tunnel statistics
1473  *      @dstats:        Dummy statistics
1474  *      @vstats:        Virtual ethernet statistics
1475  *
1476  *      @garp_port:     GARP
1477  *      @mrp_port:      MRP
1478  *
1479  *      @dev:           Class/net/name entry
1480  *      @sysfs_groups:  Space for optional device, statistics and wireless
1481  *                      sysfs groups
1482  *
1483  *      @sysfs_rx_queue_group:  Space for optional per-rx queue attributes
1484  *      @rtnl_link_ops: Rtnl_link_ops
1485  *
1486  *      @gso_max_size:  Maximum size of generic segmentation offload
1487  *      @gso_max_segs:  Maximum number of segments that can be passed to the
1488  *                      NIC for GSO
1489  *      @gso_min_segs:  Minimum number of segments that can be passed to the
1490  *                      NIC for GSO
1491  *
1492  *      @dcbnl_ops:     Data Center Bridging netlink ops
1493  *      @num_tc:        Number of traffic classes in the net device
1494  *      @tc_to_txq:     XXX: need comments on this one
1495  *      @prio_tc_map    XXX: need comments on this one
1496  *
1497  *      @fcoe_ddp_xid:  Max exchange id for FCoE LRO by ddp
1498  *
1499  *      @priomap:       XXX: need comments on this one
1500  *      @phydev:        Physical device may attach itself
1501  *                      for hardware timestamping
1502  *
1503  *      @qdisc_tx_busylock:     XXX: need comments on this one
1504  *
1505  *      FIXME: cleanup struct net_device such that network protocol info
1506  *      moves out.
1507  */
1508
1509 struct net_device {
1510         char                    name[IFNAMSIZ];
1511         struct hlist_node       name_hlist;
1512         char                    *ifalias;
1513         /*
```

```
1514          *      I/O specific fields
1515          *      FIXME: Merge these and struct ifmap into one
1516          */
1517         unsigned long           mem_end;
1518         unsigned long           mem_start;
1519         unsigned long           base_addr;
1520         int                     irq;
1521
1522         atomic_t                carrier_changes;
1523
1524         /*
1525          *      Some hardware also needs these fields (state,dev_list,
1526          *      napi_list,unreg_list,close_list) but they are not
1527          *      part of the usual set specified in Space.c.
1528          */
1529
1530         unsigned long           state;
1531
1532         struct list_head        dev_list;
1533         struct list_head        napi_list;
1534         struct list_head        unreg_list;
1535         struct list_head        close_list;
1536         struct list_head        ptype_all;
1537         struct list_head        ptype_specific;
1538
1539         struct {
1540                 struct list_head upper;
1541                 struct list_head lower;
1542         } adj_list;
1543
1544         struct {
1545                 struct list_head upper;
1546                 struct list_head lower;
1547         } all_adj_list;
1548
1549         netdev_features_t       features;
1550         netdev_features_t       hw_features;
1551         netdev_features_t       wanted_features;
1552         netdev_features_t       vlan_features;
1553         netdev_features_t       hw_enc_features;
1554         netdev_features_t       mpls_features;
1555
1556         int                     ifindex;
1557         int                     group;
1558
1559         struct net_device_stats stats;
1560
1561         atomic_long_t           rx_dropped;
1562         atomic_long_t           tx_dropped;
1563
1564 #ifdef CONFIG_WIRELESS_EXT
1565         const struct iw_handler_def *   wireless_handlers;
1566         struct iw_public_data * wireless_data;
1567 #endif
1568         const struct net_device_ops *netdev_ops;
1569         const struct ethtool_ops *ethtool_ops;
1570 #ifdef CONFIG_NET_SWITCHDEV
1571         const struct switchdev_ops *switchdev_ops;
1572 #endif
1573
1574         const struct header_ops *header_ops;
1575
1576         unsigned int            flags;
1577         unsigned int            priv_flags;
1578
1579         unsigned short          gflags;
1580         unsigned short          padded;
1581
1582         unsigned char           operstate;
1583         unsigned char           link_mode;
1584
1585         unsigned char           if_port;
1586         unsigned char           dma;
```

```
1587
1588        unsigned int            mtu;
1589        unsigned short          type;
1590        unsigned short          hard_header_len;
1591
1592        unsigned short          needed_headroom;
1593        unsigned short          needed_tailroom;
1594
1595        /* Interface address info. */
1596        unsigned char           perm_addr[MAX_ADDR_LEN];
1597        unsigned char           addr_assign_type;
1598        unsigned char           addr_len;
1599        unsigned short          neigh_priv_len;
1600        unsigned short          dev_id;
1601        unsigned short          dev_port;
1602        spinlock_t              addr_list_lock;
1603        unsigned char           name_assign_type;
1604        bool                    uc_promisc;
1605        struct netdev_hw_addr_list      uc;
1606        struct netdev_hw_addr_list      mc;
1607        struct netdev_hw_addr_list      dev_addrs;
1608
1609 #ifdef CONFIG_SYSFS
1610        struct kset             *queues_kset;
1611 #endif
1612        unsigned int            promiscuity;
1613        unsigned int            allmulti;
1614
1615
1616        /* Protocol specific pointers */
1617
1618 #if IS_ENABLED(CONFIG_VLAN_8021Q)
1619        struct vlan_info __rcu  *vlan_info;
1620 #endif
1621 #if IS_ENABLED(CONFIG_NET_DSA)
1622        struct dsa_switch_tree  *dsa_ptr;
1623 #endif
1624 #if IS_ENABLED(CONFIG_TIPC)
1625        struct tipc_bearer __rcu *tipc_ptr;
1626 #endif
1627        void                    *atalk_ptr;
1628        struct in_device __rcu  *ip_ptr;
1629        struct dn_dev __rcu     *dn_ptr;
1630        struct inet6_dev __rcu  *ip6_ptr;
1631        void                    *ax25_ptr;
1632        struct wireless_dev     *ieee80211_ptr;
1633        struct wpan_dev         *ieee802154_ptr;
1634 #if IS_ENABLED(CONFIG_MPLS_ROUTING)
1635        struct mpls_dev __rcu   *mpls_ptr;
1636 #endif
1637
1638 /*
1639  * Cache lines mostly used on receive path (including eth_type_trans())
1640  */
1641        unsigned long           last_rx;
1642
1643        /* Interface address info used in eth_type_trans() */
1644        unsigned char           *dev_addr;
1645
1646
1647 #ifdef CONFIG_SYSFS
1648        struct netdev_rx_queue  *_rx;
1649
1650        unsigned int            num_rx_queues;
1651        unsigned int            real_num_rx_queues;
1652
1653 #endif
1654
1655        unsigned long           gro_flush_timeout;
1656        rx_handler_func_t __rcu *rx_handler;
1657        void __rcu              *rx_handler_data;
1658
1659 #ifdef CONFIG_NET_CLS_ACT
```

```
1660         struct tcf_proto __rcu  *ingress_cl_list;
1661 #endif
1662         struct netdev_queue __rcu *ingress_queue;
1663 #ifdef CONFIG_NETFILTER_INGRESS
1664         struct list_head        nf_hooks_ingress;
1665 #endif
1666
1667         unsigned char           broadcast[MAX_ADDR_LEN];
1668 #ifdef CONFIG_RFS_ACCEL
1669         struct cpu_rmap         *rx_cpu_rmap;
1670 #endif
1671         struct hlist_node       index_hlist;
1672
1673 /*
1674  * Cache lines mostly used on transmit path
1675  */
1676         struct netdev_queue     *_tx ____cacheline_aligned_in_smp;
1677         unsigned int            num_tx_queues;
1678         unsigned int            real_num_tx_queues;
1679         struct Qdisc            *qdisc;
1680         unsigned long           tx_queue_len;
1681         spinlock_t              tx_global_lock;
1682         int                     watchdog_timeo;
1683
1684 #ifdef CONFIG_XPS
1685         struct xps_dev_maps __rcu *xps_maps;
1686 #endif
1687
1688         /* These may be needed for future network-power-down code. */
1689
1690         /*
1691          * trans_start here is expensive for high speed devices on SMP,
1692          * please use netdev_queue->trans_start instead.
1693          */
1694         unsigned long           trans_start;
1695
1696         struct timer_list       watchdog_timer;
1697
1698         int __percpu            *pcpu_refcnt;
1699         struct list_head        todo_list;
1700
1701         struct list_head        link_watch_list;
1702
1703         enum { NETREG_UNINITIALIZED=0,
1704                NETREG_REGISTERED,       /* completed register_netdevice */
1705                NETREG_UNREGISTERING,    /* called unregister_netdevice */
1706                NETREG_UNREGISTERED,     /* completed unregister todo */
1707                NETREG_RELEASED,         /* called free_netdev */
1708                NETREG_DUMMY,            /* dummy device for NAPI poll */
1709         } reg_state:8;
1710
1711         bool dismantle;
1712
1713         enum {
1714                 RTNL_LINK_INITIALIZED,
1715                 RTNL_LINK_INITIALIZING,
1716         } rtnl_link_state:16;
1717
1718         void (*destructor)(struct net_device *dev);
1719
1720 #ifdef CONFIG_NETPOLL
1721         struct netpoll_info __rcu       *npinfo;
1722 #endif
1723
1724         possible_net_t                  nd_net;
1725
1726         /* mid-layer private */
1727         union {
1728                 void                                    *ml_priv;
1729                 struct pcpu_lstats __percpu             *lstats;
1730                 struct pcpu_sw_netstats __percpu        *tstats;
1731                 struct pcpu_dstats __percpu             *dstats;
1732                 struct pcpu_vstats __percpu             *vstats;
```

```
1733         };
1734
1735         struct garp_port __rcu  *garp_port;
1736         struct mrp_port __rcu   *mrp_port;
1737
1738         struct device   dev;
1739         const struct attribute_group *sysfs_groups[4];
1740         const struct attribute_group *sysfs_rx_queue_group;
1741
1742         const struct rtnl_link_ops *rtnl_link_ops;
1743
1744         /* for setting kernel sock attribute on TCP connection setup */
1745 #define GSO_MAX_SIZE            65536
1746         unsigned int            gso_max_size;
1747 #define GSO_MAX_SEGS            65535
1748         u16                     gso_max_segs;
1749         u16                     gso_min_segs;
1750 #ifdef CONFIG_DCB
1751         const struct dcbnl_rtnl_ops *dcbnl_ops;
1752 #endif
1753         u8 num_tc;
1754         struct netdev_tc_txq tc_to_txq[TC_MAX_QUEUE];
1755         u8 prio_tc_map[TC_BITMASK + 1];
1756
1757 #if IS_ENABLED(CONFIG_FCOE)
1758         unsigned int            fcoe_ddp_xid;
1759 #endif
1760 #if IS_ENABLED(CONFIG_CGROUP_NET_PRIO)
1761         struct netprio_map __rcu *priomap;
1762 #endif
1763         struct phy_device *phydev;
1764         struct lock_class_key *qdisc_tx_busylock;
1765 };
1766 #define to_net_dev(d) container_of(d, struct net_device, dev)
1767
1768 #define NETDEV_ALIGN            32
1769
1770 static inline
1771 int netdev_get_prio_tc_map(const struct net_device *dev, u32 prio)
1772 {
1773         return dev->prio_tc_map[prio & TC_BITMASK];
1774 }
1775
1776 static inline
1777 int netdev_set_prio_tc_map(struct net_device *dev, u8 prio, u8 tc)
1778 {
1779         if (tc >= dev->num_tc)
1780                 return -EINVAL;
1781
1782         dev->prio_tc_map[prio & TC_BITMASK] = tc & TC_BITMASK;
1783         return 0;
1784 }
1785
1786 static inline
1787 void netdev_reset_tc(struct net_device *dev)
1788 {
1789         dev->num_tc = 0;
1790         memset(dev->tc_to_txq, 0, sizeof(dev->tc_to_txq));
1791         memset(dev->prio_tc_map, 0, sizeof(dev->prio_tc_map));
1792 }
1793
1794 static inline
1795 int netdev_set_tc_queue(struct net_device *dev, u8 tc, u16 count, u16 offset)
1796 {
1797         if (tc >= dev->num_tc)
1798                 return -EINVAL;
1799
1800         dev->tc_to_txq[tc].count = count;
1801         dev->tc_to_txq[tc].offset = offset;
1802         return 0;
1803 }
1804
1805 static inline
```

```
1806 int netdev_set_num_tc(struct net_device *dev, u8 num_tc)
1807 {
1808         if (num_tc > TC_MAX_QUEUE)
1809                 return -EINVAL;
1810
1811         dev->num_tc = num_tc;
1812         return 0;
1813 }
1814
1815 static inline
1816 int netdev_get_num_tc(struct net_device *dev)
1817 {
1818         return dev->num_tc;
1819 }
1820
1821 static inline
1822 struct netdev_queue *netdev_get_tx_queue(const struct net_device *dev,
1823                                          unsigned int index)
1824 {
1825         return &dev->_tx[index];
1826 }
1827
1828 static inline struct netdev_queue *skb_get_tx_queue(const struct net_device *dev,
1829                                                     const struct sk_buff *skb)
1830 {
1831         return netdev_get_tx_queue(dev, skb_get_queue_mapping(skb));
1832 }
1833
1834 static inline void netdev_for_each_tx_queue(struct net_device *dev,
1835                                             void (*f)(struct net_device *,
1836                                                       struct netdev_queue *,
1837                                                       void *),
1838                                             void *arg)
1839 {
1840         unsigned int i;
1841
1842         for (i = 0; i < dev->num_tx_queues; i++)
1843                 f(dev, &dev->_tx[i], arg);
1844 }
1845
1846 struct netdev_queue *netdev_pick_tx(struct net_device *dev,
1847                                     struct sk_buff *skb,
1848                                     void *accel_priv);
1849
1850 /*
1851  * Net namespace inlines
1852  */
1853 static inline
1854 struct net *dev_net(const struct net_device *dev)
1855 {
1856         return read_pnet(&dev->nd_net);
1857 }
1858
1859 static inline
1860 void dev_net_set(struct net_device *dev, struct net *net)
1861 {
1862         write_pnet(&dev->nd_net, net);
1863 }
1864
1865 static inline bool netdev_uses_dsa(struct net_device *dev)
1866 {
1867 #if IS_ENABLED(CONFIG_NET_DSA)
1868         if (dev->dsa_ptr != NULL)
1869                 return dsa_uses_tagged_protocol(dev->dsa_ptr);
1870 #endif
1871         return false;
1872 }
1873
1874 /**
1875  *      netdev_priv - access network device private data
1876  *      @dev: network device
1877  *
1878  * Get network device private data
```

```
1879  */
1880 static inline void *netdev_priv(const struct net_device *dev)
1881 {
1882         return (char *)dev + ALIGN(sizeof(struct net_device), NETDEV_ALIGN);
1883 }
1884
1885 /* Set the sysfs physical device reference for the network logical device
1886  * if set prior to registration will cause a symlink during initialization.
1887  */
1888 #define SET_NETDEV_DEV(net, pdev)       ((net)->dev.parent = (pdev))
1889
1890 /* Set the sysfs device type for the network logical device to allow
1891  * fine-grained identification of different network device types. For
1892  * example Ethernet, Wirelss LAN, Bluetooth, WiMAX etc.
1893  */
1894 #define SET_NETDEV_DEVTYPE(net, devtype)        ((net)->dev.type = (devtype))
1895
1896 /* Default NAPI poll() weight
1897  * Device drivers are strongly advised to not use bigger value
1898  */
1899 #define NAPI_POLL_WEIGHT 64
1900
1901 /**
1902  *      netif_napi_add - initialize a napi context
1903  *      @dev:  network device
1904  *      @napi: napi context
1905  *      @poll: polling function
1906  *      @weight: default weight
1907  *
1908  * netif_napi_add() must be used to initialize a napi context prior to calling
1909  * *any* of the other napi related functions.
1910  */
1911 void netif_napi_add(struct net_device *dev, struct napi_struct *napi,
1912                     int (*poll)(struct napi_struct *, int), int weight);
1913
1914 /**
1915  *  netif_napi_del - remove a napi context
1916  *  @napi: napi context
1917  *
1918  *  netif_napi_del() removes a napi context from the network device napi list
1919  */
1920 void netif_napi_del(struct napi_struct *napi);
1921
1922 struct napi_gro_cb {
1923         /* Virtual address of skb_shinfo(skb)->frags[0].page + offset. */
1924         void *frag0;
1925
1926         /* Length of frag0. */
1927         unsigned int frag0_len;
1928
1929         /* This indicates where we are processing relative to skb->data. */
1930         int data_offset;
1931
1932         /* This is non-zero if the packet cannot be merged with the new skb. */
1933         u16     flush;
1934
1935         /* Save the IP ID here and check when we get to the transport layer */
1936         u16     flush_id;
1937
1938         /* Number of segments aggregated. */
1939         u16     count;
1940
1941         /* Start offset for remote checksum offload */
1942         u16     gro_remcsum_start;
1943
1944         /* jiffies when first packet was created/queued */
1945         unsigned long age;
1946
1947         /* Used in ipv6_gro_receive() and foo-over-udp */
1948         u16     proto;
1949
1950         /* This is non-zero if the packet may be of the same flow. */
1951         u8      same_flow:1;
```

```
1952
1953         /* Used in udp_gro_receive */
1954         u8      udp_mark:1;
1955
1956         /* GRO checksum is valid */
1957         u8      csum_valid:1;
1958
1959         /* Number of checksums via CHECKSUM_UNNECESSARY */
1960         u8      csum_cnt:3;
1961
1962         /* Free the skb? */
1963         u8      free:2;
1964 #define NAPI_GRO_FREE             1
1965 #define NAPI_GRO_FREE_STOLEN_HEAD 2
1966
1967         /* Used in foo-over-udp, set in udp[46]_gro_receive */
1968         u8      is_ipv6:1;
1969
1970         /* 7 bit hole */
1971
1972         /* used to support CHECKSUM_COMPLETE for tunneling protocols */
1973         __wsum  csum;
1974
1975         /* used in skb_gro_receive() slow path */
1976         struct sk_buff *last;
1977 };
1978
1979 #define NAPI_GRO_CB(skb) ((struct napi_gro_cb *)(skb)->cb)
1980
1981 struct packet_type {
1982         __be16                  type;   /* This is really htons(ether_type). */
1983         struct net_device       *dev;   /* NULL is wildcarded here           */
1984         int                     (*func) (struct sk_buff *,
1985                                          struct net_device *,
1986                                          struct packet_type *,
1987                                          struct net_device *);
1988         bool                    (*id_match)(struct packet_type *ptype,
1989                                             struct sock *sk);
1990         void                    *af_packet_priv;
1991         struct list_head        list;
1992 };
1993
1994 struct offload_callbacks {
1995         struct sk_buff          *(*gso_segment)(struct sk_buff *skb,
1996                                                 netdev_features_t features);
1997         struct sk_buff          **(*gro_receive)(struct sk_buff **head,
1998                                                  struct sk_buff *skb);
1999         int                     (*gro_complete)(struct sk_buff *skb, int nhoff);
2000 };
2001
2002 struct packet_offload {
2003         __be16                  type;  /* This is really htons(ether_type). */
2004         u16                     priority;
2005         struct offload_callbacks callbacks;
2006         struct list_head        list;
2007 };
2008
2009 struct udp_offload;
2010
2011 struct udp_offload_callbacks {
2012         struct sk_buff          **(*gro_receive)(struct sk_buff **head,
2013                                                  struct sk_buff *skb,
2014                                                  struct udp_offload *uoff);
2015         int                     (*gro_complete)(struct sk_buff *skb,
2016                                                 int nhoff,
2017                                                 struct udp_offload *uoff);
2018 };
2019
2020 struct udp_offload {
2021         __be16                  port;
2022         u8                      ipproto;
2023         struct udp_offload_callbacks callbacks;
2024 };
```

```
2025
2026 /* often modified stats are per cpu, other are shared (netdev->stats) */
2027 struct pcpu_sw_netstats {
2028         u64     rx_packets;
2029         u64     rx_bytes;
2030         u64     tx_packets;
2031         u64     tx_bytes;
2032         struct u64_stats_sync   syncp;
2033 };
2034
2035 #define netdev_alloc_pcpu_stats(type)                           \
2036 ({                                                              \
2037         typeof(type) __percpu *pcpu_stats = alloc_percpu(type); \
2038         if (pcpu_stats) {                                       \
2039                 int __cpu;                                      \
2040                 for_each_possible_cpu(__cpu) {                  \
2041                         typeof(type) *stat;                     \
2042                         stat = per_cpu_ptr(pcpu_stats, __cpu);  \
2043                         u64_stats_init(&stat->syncp);           \
2044                 }                                               \
2045         }                                                       \
2046         pcpu_stats;                                             \
2047 })
2048
2049 #include <linux/notifier.h>
2050
2051 /* netdevice notifier chain. Please remember to update the rtnetlink
2052  * notification exclusion list in rtnetlink_event() when adding new
2053  * types.
2054  */
2055 #define NETDEV_UP       0x0001  /* For now you can't veto a device up/down */
2056 #define NETDEV_DOWN     0x0002
2057 #define NETDEV_REBOOT   0x0003  /* Tell a protocol stack a network interface
2058                                    detected a hardware crash and restarted
2059                                    - we can use this eg to kick tcp sessions
2060                                    once done */
2061 #define NETDEV_CHANGE   0x0004  /* Notify device state change */
2062 #define NETDEV_REGISTER 0x0005
2063 #define NETDEV_UNREGISTER       0x0006
2064 #define NETDEV_CHANGEMTU        0x0007 /* notify after mtu change happened */
2065 #define NETDEV_CHANGEADDR       0x0008
2066 #define NETDEV_GOING_DOWN       0x0009
2067 #define NETDEV_CHANGENAME       0x000A
2068 #define NETDEV_FEAT_CHANGE      0x000B
2069 #define NETDEV_BONDING_FAILOVER 0x000C
2070 #define NETDEV_PRE_UP           0x000D
2071 #define NETDEV_PRE_TYPE_CHANGE  0x000E
2072 #define NETDEV_POST_TYPE_CHANGE 0x000F
2073 #define NETDEV_POST_INIT        0x0010
2074 #define NETDEV_UNREGISTER_FINAL 0x0011
2075 #define NETDEV_RELEASE          0x0012
2076 #define NETDEV_NOTIFY_PEERS     0x0013
2077 #define NETDEV_JOIN             0x0014
2078 #define NETDEV_CHANGEUPPER      0x0015
2079 #define NETDEV_RESEND_IGMP      0x0016
2080 #define NETDEV_PRECHANGEMTU     0x0017 /* notify before mtu change happened */
2081 #define NETDEV_CHANGEINFODATA   0x0018
2082 #define NETDEV_BONDING_INFO     0x0019
2083
2084 int register_netdevice_notifier(struct notifier_block *nb);
2085 int unregister_netdevice_notifier(struct notifier_block *nb);
2086
2087 struct netdev_notifier_info {
2088         struct net_device *dev;
2089 };
2090
2091 struct netdev_notifier_change_info {
2092         struct netdev_notifier_info info; /* must be first */
2093         unsigned int flags_changed;
2094 };
2095
2096 static inline void netdev_notifier_info_init(struct netdev_notifier_info *info,
2097                                              struct net_device *dev)
```

```
2098 {
2099         info->dev = dev;
2100 }
2101
2102 static inline struct net_device *
2103 netdev_notifier_info_to_dev(const struct netdev_notifier_info *info)
2104 {
2105         return info->dev;
2106 }
2107
2108 int call_netdevice_notifiers(unsigned long val, struct net_device *dev);
2109
2110
2111 extern rwlock_t                         dev_base_lock;          /* Device list lock */
2112
2113 #define for_each_netdev(net, d)         \
2114                 list_for_each_entry(d, &(net)->dev_base_head, dev_list)
2115 #define for_each_netdev_reverse(net, d) \
2116                 list_for_each_entry_reverse(d, &(net)->dev_base_head, dev_list)
2117 #define for_each_netdev_rcu(net, d)             \
2118                 list_for_each_entry_rcu(d, &(net)->dev_base_head, dev_list)
2119 #define for_each_netdev_safe(net, d, n) \
2120                 list_for_each_entry_safe(d, n, &(net)->dev_base_head, dev_list)
2121 #define for_each_netdev_continue(net, d)                \
2122                 list_for_each_entry_continue(d, &(net)->dev_base_head, dev_list)
2123 #define for_each_netdev_continue_rcu(net, d)            \
2124         list_for_each_entry_continue_rcu(d, &(net)->dev_base_head, dev_list)
2125 #define for_each_netdev_in_bond_rcu(bond, slave)        \
2126                 for_each_netdev_rcu(&init_net, slave)   \
2127                         if (netdev_master_upper_dev_get_rcu(slave) == (bond))
2128 #define net_device_entry(lh)    list_entry(lh, struct net_device, dev_list)
2129
2130 static inline struct net_device *next_net_device(struct net_device *dev)
2131 {
2132         struct list_head *lh;
2133         struct net *net;
2134
2135         net = dev_net(dev);
2136         lh = dev->dev_list.next;
2137         return lh == &net->dev_base_head ? NULL : net_device_entry(lh);
2138 }
2139
2140 static inline struct net_device *next_net_device_rcu(struct net_device *dev)
2141 {
2142         struct list_head *lh;
2143         struct net *net;
2144
2145         net = dev_net(dev);
2146         lh = rcu_dereference(list_next_rcu(&dev->dev_list));
2147         return lh == &net->dev_base_head ? NULL : net_device_entry(lh);
2148 }
2149
2150 static inline struct net_device *first_net_device(struct net *net)
2151 {
2152         return list_empty(&net->dev_base_head) ? NULL :
2153                 net_device_entry(net->dev_base_head.next);
2154 }
2155
2156 static inline struct net_device *first_net_device_rcu(struct net *net)
2157 {
2158         struct list_head *lh = rcu_dereference(list_next_rcu(&net->dev_base_head));
2159
2160         return lh == &net->dev_base_head ? NULL : net_device_entry(lh);
2161 }
2162
2163 int netdev_boot_setup_check(struct net_device *dev);
2164 unsigned long netdev_boot_base(const char *prefix, int unit);
2165 struct net_device *dev_getbyhwaddr_rcu(struct net *net, unsigned short type,
2166                                        const char *hwaddr);
2167 struct net_device *dev_getfirstbyhwtype(struct net *net, unsigned short type);
2168 struct net_device *__dev_getfirstbyhwtype(struct net *net, unsigned short type);
2169 void dev_add_pack(struct packet_type *pt);
2170 void dev_remove_pack(struct packet_type *pt);
```

```
2171 void __dev_remove_pack(struct packet_type *pt);
2172 void dev_add_offload(struct packet_offload *po);
2173 void dev_remove_offload(struct packet_offload *po);
2174
2175 int dev_get_iflink(const struct net_device *dev);
2176 struct net_device *__dev_get_by_flags(struct net *net, unsigned short flags,
2177                                       unsigned short mask);
2178 struct net_device *dev_get_by_name(struct net *net, const char *name);
2179 struct net_device *dev_get_by_name_rcu(struct net *net, const char *name);
2180 struct net_device *__dev_get_by_name(struct net *net, const char *name);
2181 int dev_alloc_name(struct net_device *dev, const char *name);
2182 int dev_open(struct net_device *dev);
2183 int dev_close(struct net_device *dev);
2184 int dev_close_many(struct list_head *head, bool unlink);
2185 void dev_disable_lro(struct net_device *dev);
2186 int dev_loopback_xmit(struct sock *sk, struct sk_buff *newskb);
2187 int dev_queue_xmit_sk(struct sock *sk, struct sk_buff *skb);
2188 static inline int dev_queue_xmit(struct sk_buff *skb)
2189 {
2190         return dev_queue_xmit_sk(skb->sk, skb);
2191 }
2192 int dev_queue_xmit_accel(struct sk_buff *skb, void *accel_priv);
2193 int register_netdevice(struct net_device *dev);
2194 void unregister_netdevice_queue(struct net_device *dev, struct list_head *head);
2195 void unregister_netdevice_many(struct list_head *head);
2196 static inline void unregister_netdevice(struct net_device *dev)
2197 {
2198         unregister_netdevice_queue(dev, NULL);
2199 }
2200
2201 int netdev_refcnt_read(const struct net_device *dev);
2202 void free_netdev(struct net_device *dev);
2203 void netdev_freemem(struct net_device *dev);
2204 void synchronize_net(void);
2205 int init_dummy_netdev(struct net_device *dev);
2206
2207 DECLARE_PER_CPU(int, xmit_recursion);
2208 static inline int dev_recursion_level(void)
2209 {
2210         return this_cpu_read(xmit_recursion);
2211 }
2212
2213 struct net_device *dev_get_by_index(struct net *net, int ifindex);
2214 struct net_device *__dev_get_by_index(struct net *net, int ifindex);
2215 struct net_device *dev_get_by_index_rcu(struct net *net, int ifindex);
2216 int netdev_get_name(struct net *net, char *name, int ifindex);
2217 int dev_restart(struct net_device *dev);
2218 int skb_gro_receive(struct sk_buff **head, struct sk_buff *skb);
2219
2220 static inline unsigned int skb_gro_offset(const struct sk_buff *skb)
2221 {
2222         return NAPI_GRO_CB(skb)->data_offset;
2223 }
2224
2225 static inline unsigned int skb_gro_len(const struct sk_buff *skb)
2226 {
2227         return skb->len - NAPI_GRO_CB(skb)->data_offset;
2228 }
2229
2230 static inline void skb_gro_pull(struct sk_buff *skb, unsigned int len)
2231 {
2232         NAPI_GRO_CB(skb)->data_offset += len;
2233 }
2234
2235 static inline void *skb_gro_header_fast(struct sk_buff *skb,
2236                                         unsigned int offset)
2237 {
2238         return NAPI_GRO_CB(skb)->frag0 + offset;
2239 }
2240
2241 static inline int skb_gro_header_hard(struct sk_buff *skb, unsigned int hlen)
2242 {
2243         return NAPI_GRO_CB(skb)->frag0_len < hlen;
```

```
2244 }
2245
2246 static inline void *skb_gro_header_slow(struct sk_buff *skb, unsigned int hlen,
2247                                         unsigned int offset)
2248 {
2249         if (!pskb_may_pull(skb, hlen))
2250                 return NULL;
2251
2252         NAPI_GRO_CB(skb)->frag0 = NULL;
2253         NAPI_GRO_CB(skb)->frag0_len = 0;
2254         return skb->data + offset;
2255 }
2256
2257 static inline void *skb_gro_network_header(struct sk_buff *skb)
2258 {
2259         return (NAPI_GRO_CB(skb)->frag0 ?: skb->data) +
2260                skb_network_offset(skb);
2261 }
2262
2263 static inline void skb_gro_postpull_rcsum(struct sk_buff *skb,
2264                                         const void *start, unsigned int len)
2265 {
2266         if (NAPI_GRO_CB(skb)->csum_valid)
2267                 NAPI_GRO_CB(skb)->csum = csum_sub(NAPI_GRO_CB(skb)->csum,
2268                                                   csum_partial(start, len, 0));
2269 }
2270
2271 /* GRO checksum functions. These are logical equivalents of the normal
2272  * checksum functions (in skbuff.h) except that they operate on the GRO
2273  * offsets and fields in sk_buff.
2274  */
2275
2276 __sum16 __skb_gro_checksum_complete(struct sk_buff *skb);
2277
2278 static inline bool skb_at_gro_remcsum_start(struct sk_buff *skb)
2279 {
2280         return (NAPI_GRO_CB(skb)->gro_remcsum_start - skb_headroom(skb) ==
2281                 skb_gro_offset(skb));
2282 }
2283
2284 static inline bool __skb_gro_checksum_validate_needed(struct sk_buff *skb,
2285                                                       bool zero_okay,
2286                                                       __sum16 check)
2287 {
2288         return ((skb->ip_summed != CHECKSUM_PARTIAL ||
2289                 skb_checksum_start_offset(skb) <
2290                  skb_gro_offset(skb)) &&
2291                 !skb_at_gro_remcsum_start(skb) &&
2292                 NAPI_GRO_CB(skb)->csum_cnt == 0 &&
2293                 (!zero_okay || check));
2294 }
2295
2296 static inline __sum16 __skb_gro_checksum_validate_complete(struct sk_buff *skb,
2297                                                            __wsum psum)
2298 {
2299         if (NAPI_GRO_CB(skb)->csum_valid &&
2300             !csum_fold(csum_add(psum, NAPI_GRO_CB(skb)->csum)))
2301                 return 0;
2302
2303         NAPI_GRO_CB(skb)->csum = psum;
2304
2305         return __skb_gro_checksum_complete(skb);
2306 }
2307
2308 static inline void skb_gro_incr_csum_unnecessary(struct sk_buff *skb)
2309 {
2310         if (NAPI_GRO_CB(skb)->csum_cnt > 0) {
2311                 /* Consume a checksum from CHECKSUM_UNNECESSARY */
2312                 NAPI_GRO_CB(skb)->csum_cnt--;
2313         } else {
2314                 /* Update skb for CHECKSUM_UNNECESSARY and csum_level when we
2315                  * verified a new top level checksum or an encapsulated one
2316                  * during GRO. This saves work if we fallback to normal path.
```

```
2317                 */
2318                __skb_incr_checksum_unnecessary(skb);
2319        }
2320 }
2321
2322 #define __skb_gro_checksum_validate(skb, proto, zero_okay, check,       \
2323                                     compute_pseudo)                     \
2324 ({                                                                      \
2325        __sum16 __ret = 0;                                               \
2326        if (__skb_gro_checksum_validate_needed(skb, zero_okay, check))   \
2327                __ret = __skb_gro_checksum_validate_complete(skb,        \
2328                                compute_pseudo(skb, proto));             \
2329        if (__ret)                                                       \
2330                __skb_mark_checksum_bad(skb);                            \
2331        else                                                             \
2332                skb_gro_incr_csum_unnecessary(skb);                      \
2333        __ret;                                                           \
2334 })
2335
2336 #define skb_gro_checksum_validate(skb, proto, compute_pseudo)           \
2337        __skb_gro_checksum_validate(skb, proto, false, 0, compute_pseudo)
2338
2339 #define skb_gro_checksum_validate_zero_check(skb, proto, check,         \
2340                                              compute_pseudo)            \
2341        __skb_gro_checksum_validate(skb, proto, true, check, compute_pseudo)
2342
2343 #define skb_gro_checksum_simple_validate(skb)                           \
2344        __skb_gro_checksum_validate(skb, 0, false, 0, null_compute_pseudo)
2345
2346 static inline bool __skb_gro_checksum_convert_check(struct sk_buff *skb)
2347 {
2348        return (NAPI_GRO_CB(skb)->csum_cnt == 0 &&
2349                !NAPI_GRO_CB(skb)->csum_valid);
2350 }
2351
2352 static inline void __skb_gro_checksum_convert(struct sk_buff *skb,
2353                                               __sum16 check, __wsum pseudo)
2354 {
2355        NAPI_GRO_CB(skb)->csum = ~pseudo;
2356        NAPI_GRO_CB(skb)->csum_valid = 1;
2357 }
2358
2359 #define skb_gro_checksum_try_convert(skb, proto, check, compute_pseudo) \
2360 do {                                                                    \
2361        if (__skb_gro_checksum_convert_check(skb))                       \
2362                __skb_gro_checksum_convert(skb, check,                   \
2363                                           compute_pseudo(skb, proto)); \
2364 } while (0)
2365
2366 struct gro_remcsum {
2367        int offset;
2368        __wsum delta;
2369 };
2370
2371 static inline void skb_gro_remcsum_init(struct gro_remcsum *grc)
2372 {
2373        grc->offset = 0;
2374        grc->delta = 0;
2375 }
2376
2377 static inline void skb_gro_remcsum_process(struct sk_buff *skb, void *ptr,
2378                                            int start, int offset,
2379                                            struct gro_remcsum *grc,
2380                                            bool nopartial)
2381 {
2382        __wsum delta;
2383
2384        BUG_ON(!NAPI_GRO_CB(skb)->csum_valid);
2385
2386        if (!nopartial) {
2387                NAPI_GRO_CB(skb)->gro_remcsum_start =
2388                    ((unsigned char *)ptr + start) - skb->head;
2389                return;
```

```
2390         }
2391
2392         delta = remcsum_adjust(ptr, NAPI_GRO_CB(skb)->csum, start, offset);
2393
2394         /* Adjust skb->csum since we changed the packet */
2395         NAPI_GRO_CB(skb)->csum = csum_add(NAPI_GRO_CB(skb)->csum, delta);
2396
2397         grc->offset = (ptr + offset) - (void *)skb->head;
2398         grc->delta = delta;
2399 }
2400
2401 static inline void skb_gro_remcsum_cleanup(struct sk_buff *skb,
2402                                            struct gro_remcsum *grc)
2403 {
2404         if (!grc->delta)
2405                 return;
2406
2407         remcsum_unadjust((__sum16 *)(skb->head + grc->offset), grc->delta);
2408 }
2409
2410 static inline int dev_hard_header(struct sk_buff *skb, struct net_device *dev,
2411                                   unsigned short type,
2412                                   const void *daddr, const void *saddr,
2413                                   unsigned int len)
2414 {
2415         if (!dev->header_ops || !dev->header_ops->create)
2416                 return 0;
2417
2418         return dev->header_ops->create(skb, dev, type, daddr, saddr, len);
2419 }
2420
2421 static inline int dev_parse_header(const struct sk_buff *skb,
2422                                    unsigned char *haddr)
2423 {
2424         const struct net_device *dev = skb->dev;
2425
2426         if (!dev->header_ops || !dev->header_ops->parse)
2427                 return 0;
2428         return dev->header_ops->parse(skb, haddr);
2429 }
2430
2431 typedef int gifconf_func_t(struct net_device * dev, char __user * bufptr, int len);
2432 int register_gifconf(unsigned int family, gifconf_func_t *gifconf);
2433 static inline int unregister_gifconf(unsigned int family)
2434 {
2435         return register_gifconf(family, NULL);
2436 }
2437
2438 #ifdef CONFIG_NET_FLOW_LIMIT
2439 #define FLOW_LIMIT_HISTORY      (1 << 7)  /* must be ^2 and !overflow buckets */
2440 struct sd_flow_limit {
2441         u64                     count;
2442         unsigned int            num_buckets;
2443         unsigned int            history_head;
2444         u16                     history[FLOW_LIMIT_HISTORY];
2445         u8                      buckets[];
2446 };
2447
2448 extern int netdev_flow_limit_table_len;
2449 #endif /* CONFIG_NET_FLOW_LIMIT */
2450
2451 /*
2452  * Incoming packets are placed on per-cpu queues
2453  */
2454 struct softnet_data {
2455         struct list_head        poll_list;
2456         struct sk_buff_head     process_queue;
2457
2458         /* stats */
2459         unsigned int            processed;
2460         unsigned int            time_squeeze;
2461         unsigned int            cpu_collision;
2462         unsigned int            received_rps;
```

```
2463 #ifdef CONFIG_RPS
2464         struct softnet_data     *rps_ipi_list;
2465 #endif
2466 #ifdef CONFIG_NET_FLOW_LIMIT
2467         struct sd_flow_limit __rcu *flow_limit;
2468 #endif
2469         struct Qdisc            *output_queue;
2470         struct Qdisc            **output_queue_tailp;
2471         struct sk_buff          *completion_queue;
2472
2473 #ifdef CONFIG_RPS
2474         /* Elements below can be accessed between CPUs for RPS */
2475         struct call_single_data csd ____cacheline_aligned_in_smp;
2476         struct softnet_data     *rps_ipi_next;
2477         unsigned int            cpu;
2478         unsigned int            input_queue_head;
2479         unsigned int            input_queue_tail;
2480 #endif
2481         unsigned int            dropped;
2482         struct sk_buff_head     input_pkt_queue;
2483         struct napi_struct      backlog;
2484
2485 };
2486
2487 static inline void input_queue_head_incr(struct softnet_data *sd)
2488 {
2489 #ifdef CONFIG_RPS
2490         sd->input_queue_head++;
2491 #endif
2492 }
2493
2494 static inline void input_queue_tail_incr_save(struct softnet_data *sd,
2495                                               unsigned int *qtail)
2496 {
2497 #ifdef CONFIG_RPS
2498         *qtail = ++sd->input_queue_tail;
2499 #endif
2500 }
2501
2502 DECLARE_PER_CPU_ALIGNED(struct softnet_data, softnet_data);
2503
2504 void __netif_schedule(struct Qdisc *q);
2505 void netif_schedule_queue(struct netdev_queue *txq);
2506
2507 static inline void netif_tx_schedule_all(struct net_device *dev)
2508 {
2509         unsigned int i;
2510
2511         for (i = 0; i < dev->num_tx_queues; i++)
2512                 netif_schedule_queue(netdev_get_tx_queue(dev, i));
2513 }
2514
2515 static inline void netif_tx_start_queue(struct netdev_queue *dev_queue)
2516 {
2517         clear_bit(__QUEUE_STATE_DRV_XOFF, &dev_queue->state);
2518 }
2519
2520 /**
2521  *      netif_start_queue - allow transmit
2522  *      @dev: network device
2523  *
2524  *      Allow upper layers to call the device hard_start_xmit routine.
2525  */
2526 static inline void netif_start_queue(struct net_device *dev)
2527 {
2528         netif_tx_start_queue(netdev_get_tx_queue(dev, 0));
2529 }
2530
2531 static inline void netif_tx_start_all_queues(struct net_device *dev)
2532 {
2533         unsigned int i;
2534
2535         for (i = 0; i < dev->num_tx_queues; i++) {
```

```
2536                    struct netdev_queue *txq = netdev_get_tx_queue(dev, i);
2537                    netif_tx_start_queue(txq);
2538            }
2539    }
2540
2541    void netif_tx_wake_queue(struct netdev_queue *dev_queue);
2542
2543    /**
2544     *      netif_wake_queue - restart transmit
2545     *      @dev: network device
2546     *
2547     *      Allow upper layers to call the device hard_start_xmit routine.
2548     *      Used for flow control when transmit resources are available.
2549     */
2550    static inline void netif_wake_queue(struct net_device *dev)
2551    {
2552            netif_tx_wake_queue(netdev_get_tx_queue(dev, 0));
2553    }
2554
2555    static inline void netif_tx_wake_all_queues(struct net_device *dev)
2556    {
2557            unsigned int i;
2558
2559            for (i = 0; i < dev->num_tx_queues; i++) {
2560                    struct netdev_queue *txq = netdev_get_tx_queue(dev, i);
2561                    netif_tx_wake_queue(txq);
2562            }
2563    }
2564
2565    static inline void netif_tx_stop_queue(struct netdev_queue *dev_queue)
2566    {
2567            set_bit(__QUEUE_STATE_DRV_XOFF, &dev_queue->state);
2568    }
2569
2570    /**
2571     *      netif_stop_queue - stop transmitted packets
2572     *      @dev: network device
2573     *
2574     *      Stop upper layers calling the device hard_start_xmit routine.
2575     *      Used for flow control when transmit resources are unavailable.
2576     */
2577    static inline void netif_stop_queue(struct net_device *dev)
2578    {
2579            netif_tx_stop_queue(netdev_get_tx_queue(dev, 0));
2580    }
2581
2582    void netif_tx_stop_all_queues(struct net_device *dev);
2583
2584    static inline bool netif_tx_queue_stopped(const struct netdev_queue *dev_queue)
2585    {
2586            return test_bit(__QUEUE_STATE_DRV_XOFF, &dev_queue->state);
2587    }
2588
2589    /**
2590     *      netif_queue_stopped - test if transmit queue is flowblocked
2591     *      @dev: network device
2592     *
2593     *      Test if transmit queue on device is currently unable to send.
2594     */
2595    static inline bool netif_queue_stopped(const struct net_device *dev)
2596    {
2597            return netif_tx_queue_stopped(netdev_get_tx_queue(dev, 0));
2598    }
2599
2600    static inline bool netif_xmit_stopped(const struct netdev_queue *dev_queue)
2601    {
2602            return dev_queue->state & QUEUE_STATE_ANY_XOFF;
2603    }
2604
2605    static inline bool
2606    netif_xmit_frozen_or_stopped(const struct netdev_queue *dev_queue)
2607    {
2608            return dev_queue->state & QUEUE_STATE_ANY_XOFF_OR_FROZEN;
```

```
2609 }
2610
2611 static inline bool
2612 netif_xmit_frozen_or_drv_stopped(const struct netdev_queue *dev_queue)
2613 {
2614         return dev_queue->state & QUEUE_STATE_DRV_XOFF_OR_FROZEN;
2615 }
2616
2617 /**
2618  *      netdev_txq_bql_enqueue_prefetchw - prefetch bql data for write
2619  *      @dev_queue: pointer to transmit queue
2620  *
2621  * BQL enabled drivers might use this helper in their ndo_start_xmit(),
2622  * to give appropriate hint to the cpu.
2623  */
2624 static inline void netdev_txq_bql_enqueue_prefetchw(struct netdev_queue *dev_queue)
2625 {
2626 #ifdef CONFIG_BQL
2627         prefetchw(&dev_queue->dql.num_queued);
2628 #endif
2629 }
2630
2631 /**
2632  *      netdev_txq_bql_complete_prefetchw - prefetch bql data for write
2633  *      @dev_queue: pointer to transmit queue
2634  *
2635  * BQL enabled drivers might use this helper in their TX completion path,
2636  * to give appropriate hint to the cpu.
2637  */
2638 static inline void netdev_txq_bql_complete_prefetchw(struct netdev_queue *dev_queue)
2639 {
2640 #ifdef CONFIG_BQL
2641         prefetchw(&dev_queue->dql.limit);
2642 #endif
2643 }
2644
2645 static inline void netdev_tx_sent_queue(struct netdev_queue *dev_queue,
2646                                         unsigned int bytes)
2647 {
2648 #ifdef CONFIG_BQL
2649         dql_queued(&dev_queue->dql, bytes);
2650
2651         if (likely(dql_avail(&dev_queue->dql) >= 0))
2652                 return;
2653
2654         set_bit(__QUEUE_STATE_STACK_XOFF, &dev_queue->state);
2655
2656         /*
2657          * The XOFF flag must be set before checking the dql_avail below,
2658          * because in netdev_tx_completed_queue we update the dql_completed
2659          * before checking the XOFF flag.
2660          */
2661         smp_mb();
2662
2663         /* check again in case another CPU has just made room avail */
2664         if (unlikely(dql_avail(&dev_queue->dql) >= 0))
2665                 clear_bit(__QUEUE_STATE_STACK_XOFF, &dev_queue->state);
2666 #endif
2667 }
2668
2669 /**
2670  *      netdev_sent_queue - report the number of bytes queued to hardware
2671  *      @dev: network device
2672  *      @bytes: number of bytes queued to the hardware device queue
2673  *
2674  *      Report the number of bytes queued for sending/completion to the network
2675  *      device hardware queue. @bytes should be a good approximation and should
2676  *      exactly match netdev_completed_queue() @bytes
2677  */
2678 static inline void netdev_sent_queue(struct net_device *dev, unsigned int bytes)
2679 {
2680         netdev_tx_sent_queue(netdev_get_tx_queue(dev, 0), bytes);
2681 }
```

```
2682
2683 static inline void netdev_tx_completed_queue(struct netdev_queue *dev_queue,
2684                                               unsigned int pkts, unsigned int bytes)
2685 {
2686 #ifdef CONFIG_BQL
2687         if (unlikely(!bytes))
2688                 return;
2689
2690         dql_completed(&dev_queue->dql, bytes);
2691
2692         /*
2693          * Without the memory barrier there is a small possiblity that
2694          * netdev_tx_sent_queue will miss the update and cause the queue to
2695          * be stopped forever
2696          */
2697         smp_mb();
2698
2699         if (dql_avail(&dev_queue->dql) < 0)
2700                 return;
2701
2702         if (test_and_clear_bit(__QUEUE_STATE_STACK_XOFF, &dev_queue->state))
2703                 netif_schedule_queue(dev_queue);
2704 #endif
2705 }
2706
2707 /**
2708  *      netdev_completed_queue - report bytes and packets completed by device
2709  *      @dev: network device
2710  *      @pkts: actual number of packets sent over the medium
2711  *      @bytes: actual number of bytes sent over the medium
2712  *
2713  *      Report the number of bytes and packets transmitted by the network device
2714  *      hardware queue over the physical medium, @bytes must exactly match the
2715  *      @bytes amount passed to netdev_sent_queue()
2716  */
2717 static inline void netdev_completed_queue(struct net_device *dev,
2718                                           unsigned int pkts, unsigned int bytes)
2719 {
2720         netdev_tx_completed_queue(netdev_get_tx_queue(dev, 0), pkts, bytes);
2721 }
2722
2723 static inline void netdev_tx_reset_queue(struct netdev_queue *q)
2724 {
2725 #ifdef CONFIG_BQL
2726         clear_bit(__QUEUE_STATE_STACK_XOFF, &q->state);
2727         dql_reset(&q->dql);
2728 #endif
2729 }
2730
2731 /**
2732  *      netdev_reset_queue - reset the packets and bytes count of a network device
2733  *      @dev_queue: network device
2734  *
2735  *      Reset the bytes and packet count of a network device and clear the
2736  *      software flow control OFF bit for this network device
2737  */
2738 static inline void netdev_reset_queue(struct net_device *dev_queue)
2739 {
2740         netdev_tx_reset_queue(netdev_get_tx_queue(dev_queue, 0));
2741 }
2742
2743 /**
2744  *      netdev_cap_txqueue - check if selected tx queue exceeds device queues
2745  *      @dev: network device
2746  *      @queue_index: given tx queue index
2747  *
2748  *      Returns 0 if given tx queue index >= number of device tx queues,
2749  *      otherwise returns the originally passed tx queue index.
2750  */
2751 static inline u16 netdev_cap_txqueue(struct net_device *dev, u16 queue_index)
2752 {
2753         if (unlikely(queue_index >= dev->real_num_tx_queues)) {
2754                 net_warn_ratelimited("%s selects TX queue %d, but real number of TX queues is %d\n",
```

```
2755                                    dev->name, queue_index,
2756                                    dev->real_num_tx_queues);
2757                 return 0;
2758         }
2759
2760         return queue_index;
2761 }
2762
2763 /**
2764  *      netif_running - test if up
2765  *      @dev: network device
2766  *
2767  *      Test if the device has been brought up.
2768  */
2769 static inline bool netif_running(const struct net_device *dev)
2770 {
2771         return test_bit(__LINK_STATE_START, &dev->state);
2772 }
2773
2774 /*
2775  * Routines to manage the subqueues on a device.  We only need start
2776  * stop, and a check if it's stopped.  All other device management is
2777  * done at the overall netdevice level.
2778  * Also test the device if we're multiqueue.
2779  */
2780
2781 /**
2782  *      netif_start_subqueue - allow sending packets on subqueue
2783  *      @dev: network device
2784  *      @queue_index: sub queue index
2785  *
2786  * Start individual transmit queue of a device with multiple transmit queues.
2787  */
2788 static inline void netif_start_subqueue(struct net_device *dev, u16 queue_index)
2789 {
2790         struct netdev_queue *txq = netdev_get_tx_queue(dev, queue_index);
2791
2792         netif_tx_start_queue(txq);
2793 }
2794
2795 /**
2796  *      netif_stop_subqueue - stop sending packets on subqueue
2797  *      @dev: network device
2798  *      @queue_index: sub queue index
2799  *
2800  * Stop individual transmit queue of a device with multiple transmit queues.
2801  */
2802 static inline void netif_stop_subqueue(struct net_device *dev, u16 queue_index)
2803 {
2804         struct netdev_queue *txq = netdev_get_tx_queue(dev, queue_index);
2805         netif_tx_stop_queue(txq);
2806 }
2807
2808 /**
2809  *      netif_subqueue_stopped - test status of subqueue
2810  *      @dev: network device
2811  *      @queue_index: sub queue index
2812  *
2813  * Check individual transmit queue of a device with multiple transmit queues.
2814  */
2815 static inline bool __netif_subqueue_stopped(const struct net_device *dev,
2816                                             u16 queue_index)
2817 {
2818         struct netdev_queue *txq = netdev_get_tx_queue(dev, queue_index);
2819
2820         return netif_tx_queue_stopped(txq);
2821 }
2822
2823 static inline bool netif_subqueue_stopped(const struct net_device *dev,
2824                                           struct sk_buff *skb)
2825 {
2826         return __netif_subqueue_stopped(dev, skb_get_queue_mapping(skb));
2827 }
```

```
2828
2829 void netif_wake_subqueue(struct net_device *dev, u16 queue_index);
2830
2831 #ifdef CONFIG_XPS
2832 int netif_set_xps_queue(struct net_device *dev, const struct cpumask *mask,
2833                         u16 index);
2834 #else
2835 static inline int netif_set_xps_queue(struct net_device *dev,
2836                                       const struct cpumask *mask,
2837                                       u16 index)
2838 {
2839         return 0;
2840 }
2841 #endif
2842
2843 u16 __skb_tx_hash(const struct net_device *dev, struct sk_buff *skb,
2844                   unsigned int num_tx_queues);
2845
2846 /*
2847  * Returns a Tx hash for the given packet when dev->real_num_tx_queues is used
2848  * as a distribution range limit for the returned value.
2849  */
2850 static inline u16 skb_tx_hash(const struct net_device *dev,
2851                               struct sk_buff *skb)
2852 {
2853         return __skb_tx_hash(dev, skb, dev->real_num_tx_queues);
2854 }
2855
2856 /**
2857  *      netif_is_multiqueue - test if device has multiple transmit queues
2858  *      @dev: network device
2859  *
2860  * Check if device has multiple transmit queues
2861  */
2862 static inline bool netif_is_multiqueue(const struct net_device *dev)
2863 {
2864         return dev->num_tx_queues > 1;
2865 }
2866
2867 int netif_set_real_num_tx_queues(struct net_device *dev, unsigned int txq);
2868
2869 #ifdef CONFIG_SYSFS
2870 int netif_set_real_num_rx_queues(struct net_device *dev, unsigned int rxq);
2871 #else
2872 static inline int netif_set_real_num_rx_queues(struct net_device *dev,
2873                                                unsigned int rxq)
2874 {
2875         return 0;
2876 }
2877 #endif
2878
2879 #ifdef CONFIG_SYSFS
2880 static inline unsigned int get_netdev_rx_queue_index(
2881                 struct netdev_rx_queue *queue)
2882 {
2883         struct net_device *dev = queue->dev;
2884         int index = queue - dev->_rx;
2885
2886         BUG_ON(index >= dev->num_rx_queues);
2887         return index;
2888 }
2889 #endif
2890
2891 #define DEFAULT_MAX_NUM_RSS_QUEUES      (8)
2892 int netif_get_num_default_rss_queues(void);
2893
2894 enum skb_free_reason {
2895         SKB_REASON_CONSUMED,
2896         SKB_REASON_DROPPED,
2897 };
2898
2899 void __dev_kfree_skb_irq(struct sk_buff *skb, enum skb_free_reason reason);
2900 void __dev_kfree_skb_any(struct sk_buff *skb, enum skb_free_reason reason);
```

```
2901
2902 /*
2903  * It is not allowed to call kfree_skb() or consume_skb() from hardware
2904  * interrupt context or with hardware interrupts being disabled.
2905  * (in_irq() || irqs_disabled())
2906  *
2907  * We provide four helpers that can be used in following contexts :
2908  *
2909  * dev_kfree_skb_irq(skb) when caller drops a packet from irq context,
2910  *  replacing kfree_skb(skb)
2911  *
2912  * dev_consume_skb_irq(skb) when caller consumes a packet from irq context.
2913  *  Typically used in place of consume_skb(skb) in TX completion path
2914  *
2915  * dev_kfree_skb_any(skb) when caller doesn't know its current irq context,
2916  *  replacing kfree_skb(skb)
2917  *
2918  * dev_consume_skb_any(skb) when caller doesn't know its current irq context,
2919  *  and consumed a packet. Used in place of consume_skb(skb)
2920  */
2921 static inline void dev_kfree_skb_irq(struct sk_buff *skb)
2922 {
2923         __dev_kfree_skb_irq(skb, SKB_REASON_DROPPED);
2924 }
2925
2926 static inline void dev_consume_skb_irq(struct sk_buff *skb)
2927 {
2928         __dev_kfree_skb_irq(skb, SKB_REASON_CONSUMED);
2929 }
2930
2931 static inline void dev_kfree_skb_any(struct sk_buff *skb)
2932 {
2933         __dev_kfree_skb_any(skb, SKB_REASON_DROPPED);
2934 }
2935
2936 static inline void dev_consume_skb_any(struct sk_buff *skb)
2937 {
2938         __dev_kfree_skb_any(skb, SKB_REASON_CONSUMED);
2939 }
2940
2941 int netif_rx(struct sk_buff *skb);
2942 int netif_rx_ni(struct sk_buff *skb);
2943 int netif_receive_skb_sk(struct sock *sk, struct sk_buff *skb);
2944 static inline int netif_receive_skb(struct sk_buff *skb)
2945 {
2946         return netif_receive_skb_sk(skb->sk, skb);
2947 }
2948 gro_result_t napi_gro_receive(struct napi_struct *napi, struct sk_buff *skb);
2949 void napi_gro_flush(struct napi_struct *napi, bool flush_old);
2950 struct sk_buff *napi_get_frags(struct napi_struct *napi);
2951 gro_result_t napi_gro_frags(struct napi_struct *napi);
2952 struct packet_offload *gro_find_receive_by_type(__be16 type);
2953 struct packet_offload *gro_find_complete_by_type(__be16 type);
2954
2955 static inline void napi_free_frags(struct napi_struct *napi)
2956 {
2957         kfree_skb(napi->skb);
2958         napi->skb = NULL;
2959 }
2960
2961 int netdev_rx_handler_register(struct net_device *dev,
2962                                rx_handler_func_t *rx_handler,
2963                                void *rx_handler_data);
2964 void netdev_rx_handler_unregister(struct net_device *dev);
2965
2966 bool dev_valid_name(const char *name);
2967 int dev_ioctl(struct net *net, unsigned int cmd, void __user *);
2968 int dev_ethtool(struct net *net, struct ifreq *);
2969 unsigned int dev_get_flags(const struct net_device *);
2970 int __dev_change_flags(struct net_device *, unsigned int flags);
2971 int dev_change_flags(struct net_device *, unsigned int);
2972 void __dev_notify_flags(struct net_device *, unsigned int old_flags,
2973                         unsigned int gchanges);
```

```
2974 int dev_change_name(struct net_device *, const char *);
2975 int dev_set_alias(struct net_device *, const char *, size_t);
2976 int dev_change_net_namespace(struct net_device *, struct net *, const char *);
2977 int dev_set_mtu(struct net_device *, int);
2978 void dev_set_group(struct net_device *, int);
2979 int dev_set_mac_address(struct net_device *, struct sockaddr *);
2980 int dev_change_carrier(struct net_device *, bool new_carrier);
2981 int dev_get_phys_port_id(struct net_device *dev,
2982                          struct netdev_phys_item_id *ppid);
2983 int dev_get_phys_port_name(struct net_device *dev,
2984                            char *name, size_t len);
2985 struct sk_buff *validate_xmit_skb_list(struct sk_buff *skb, struct net_device *dev);
2986 struct sk_buff *dev_hard_start_xmit(struct sk_buff *skb, struct net_device *dev,
2987                                     struct netdev_queue *txq, int *ret);
2988 int __dev_forward_skb(struct net_device *dev, struct sk_buff *skb);
2989 int dev_forward_skb(struct net_device *dev, struct sk_buff *skb);
2990 bool is_skb_forwardable(struct net_device *dev, struct sk_buff *skb);
2991
2992 extern int              netdev_budget;
2993
2994 /* Called by rtnetlink.c:rtnl_unlock() */
2995 void netdev_run_todo(void);
2996
2997 /**
2998  *      dev_put - release reference to device
2999  *      @dev: network device
3000  *
3001  * Release reference to device to allow it to be freed.
3002  */
3003 static inline void dev_put(struct net_device *dev)
3004 {
3005         this_cpu_dec(*dev->pcpu_refcnt);
3006 }
3007
3008 /**
3009  *      dev_hold - get reference to device
3010  *      @dev: network device
3011  *
3012  * Hold reference to device to keep it from being freed.
3013  */
3014 static inline void dev_hold(struct net_device *dev)
3015 {
3016         this_cpu_inc(*dev->pcpu_refcnt);
3017 }
3018
3019 /* Carrier loss detection, dial on demand. The functions netif_carrier_on
3020  * and _off may be called from IRQ context, but it is caller
3021  * who is responsible for serialization of these calls.
3022  *
3023  * The name carrier is inappropriate, these functions should really be
3024  * called netif_lowerlayer_*() because they represent the state of any
3025  * kind of lower layer not just hardware media.
3026  */
3027
3028 void linkwatch_init_dev(struct net_device *dev);
3029 void linkwatch_fire_event(struct net_device *dev);
3030 void linkwatch_forget_dev(struct net_device *dev);
3031
3032 /**
3033  *      netif_carrier_ok - test if carrier present
3034  *      @dev: network device
3035  *
3036  * Check if carrier is present on device
3037  */
3038 static inline bool netif_carrier_ok(const struct net_device *dev)
3039 {
3040         return !test_bit(__LINK_STATE_NOCARRIER, &dev->state);
3041 }
3042
3043 unsigned long dev_trans_start(struct net_device *dev);
3044
3045 void __netdev_watchdog_up(struct net_device *dev);
3046
```

```
3047 void netif_carrier_on(struct net_device *dev);
3048
3049 void netif_carrier_off(struct net_device *dev);
3050
3051 /**
3052  *      netif_dormant_on - mark device as dormant.
3053  *      @dev: network device
3054  *
3055  * Mark device as dormant (as per RFC2863).
3056  *
3057  * The dormant state indicates that the relevant interface is not
3058  * actually in a condition to pass packets (i.e., it is not 'up') but is
3059  * in a "pending" state, waiting for some external event.  For "on-
3060  * demand" interfaces, this new state identifies the situation where the
3061  * interface is waiting for events to place it in the up state.
3062  *
3063  */
3064 static inline void netif_dormant_on(struct net_device *dev)
3065 {
3066         if (!test_and_set_bit(__LINK_STATE_DORMANT, &dev->state))
3067                 linkwatch_fire_event(dev);
3068 }
3069
3070 /**
3071  *      netif_dormant_off - set device as not dormant.
3072  *      @dev: network device
3073  *
3074  * Device is not in dormant state.
3075  */
3076 static inline void netif_dormant_off(struct net_device *dev)
3077 {
3078         if (test_and_clear_bit(__LINK_STATE_DORMANT, &dev->state))
3079                 linkwatch_fire_event(dev);
3080 }
3081
3082 /**
3083  *      netif_dormant - test if carrier present
3084  *      @dev: network device
3085  *
3086  * Check if carrier is present on device
3087  */
3088 static inline bool netif_dormant(const struct net_device *dev)
3089 {
3090         return test_bit(__LINK_STATE_DORMANT, &dev->state);
3091 }
3092
3093
3094 /**
3095  *      netif_oper_up - test if device is operational
3096  *      @dev: network device
3097  *
3098  * Check if carrier is operational
3099  */
3100 static inline bool netif_oper_up(const struct net_device *dev)
3101 {
3102         return (dev->operstate == IF_OPER_UP ||
3103                 dev->operstate == IF_OPER_UNKNOWN /* backward compat */);
3104 }
3105
3106 /**
3107  *      netif_device_present - is device available or removed
3108  *      @dev: network device
3109  *
3110  * Check if device has not been removed from system.
3111  */
3112 static inline bool netif_device_present(struct net_device *dev)
3113 {
3114         return test_bit(__LINK_STATE_PRESENT, &dev->state);
3115 }
3116
3117 void netif_device_detach(struct net_device *dev);
3118
3119 void netif_device_attach(struct net_device *dev);
```

```
3120
3121 /*
3122  * Network interface message level settings
3123  */
3124
3125 enum {
3126         NETIF_MSG_DRV           = 0x0001,
3127         NETIF_MSG_PROBE         = 0x0002,
3128         NETIF_MSG_LINK          = 0x0004,
3129         NETIF_MSG_TIMER         = 0x0008,
3130         NETIF_MSG_IFDOWN        = 0x0010,
3131         NETIF_MSG_IFUP          = 0x0020,
3132         NETIF_MSG_RX_ERR        = 0x0040,
3133         NETIF_MSG_TX_ERR        = 0x0080,
3134         NETIF_MSG_TX_QUEUED     = 0x0100,
3135         NETIF_MSG_INTR          = 0x0200,
3136         NETIF_MSG_TX_DONE       = 0x0400,
3137         NETIF_MSG_RX_STATUS     = 0x0800,
3138         NETIF_MSG_PKTDATA       = 0x1000,
3139         NETIF_MSG_HW            = 0x2000,
3140         NETIF_MSG_WOL           = 0x4000,
3141 };
3142
3143 #define netif_msg_drv(p)        ((p)->msg_enable & NETIF_MSG_DRV)
3144 #define netif_msg_probe(p)      ((p)->msg_enable & NETIF_MSG_PROBE)
3145 #define netif_msg_link(p)       ((p)->msg_enable & NETIF_MSG_LINK)
3146 #define netif_msg_timer(p)      ((p)->msg_enable & NETIF_MSG_TIMER)
3147 #define netif_msg_ifdown(p)     ((p)->msg_enable & NETIF_MSG_IFDOWN)
3148 #define netif_msg_ifup(p)       ((p)->msg_enable & NETIF_MSG_IFUP)
3149 #define netif_msg_rx_err(p)     ((p)->msg_enable & NETIF_MSG_RX_ERR)
3150 #define netif_msg_tx_err(p)     ((p)->msg_enable & NETIF_MSG_TX_ERR)
3151 #define netif_msg_tx_queued(p)  ((p)->msg_enable & NETIF_MSG_TX_QUEUED)
3152 #define netif_msg_intr(p)       ((p)->msg_enable & NETIF_MSG_INTR)
3153 #define netif_msg_tx_done(p)    ((p)->msg_enable & NETIF_MSG_TX_DONE)
3154 #define netif_msg_rx_status(p)  ((p)->msg_enable & NETIF_MSG_RX_STATUS)
3155 #define netif_msg_pktdata(p)    ((p)->msg_enable & NETIF_MSG_PKTDATA)
3156 #define netif_msg_hw(p)         ((p)->msg_enable & NETIF_MSG_HW)
3157 #define netif_msg_wol(p)        ((p)->msg_enable & NETIF_MSG_WOL)
3158
3159 static inline u32 netif_msg_init(int debug_value, int default_msg_enable_bits)
3160 {
3161         /* use default */
3162         if (debug_value < 0 || debug_value >= (sizeof(u32) * 8))
3163                 return default_msg_enable_bits;
3164         if (debug_value == 0)   /* no output */
3165                 return 0;
3166         /* set low N bits */
3167         return (1 << debug_value) - 1;
3168 }
3169
3170 static inline void __netif_tx_lock(struct netdev_queue *txq, int cpu)
3171 {
3172         spin_lock(&txq->_xmit_lock);
3173         txq->xmit_lock_owner = cpu;
3174 }
3175
3176 static inline void __netif_tx_lock_bh(struct netdev_queue *txq)
3177 {
3178         spin_lock_bh(&txq->_xmit_lock);
3179         txq->xmit_lock_owner = smp_processor_id();
3180 }
3181
3182 static inline bool __netif_tx_trylock(struct netdev_queue *txq)
3183 {
3184         bool ok = spin_trylock(&txq->_xmit_lock);
3185         if (likely(ok))
3186                 txq->xmit_lock_owner = smp_processor_id();
3187         return ok;
3188 }
3189
3190 static inline void __netif_tx_unlock(struct netdev_queue *txq)
3191 {
3192         txq->xmit_lock_owner = -1;
```

```
3193         spin_unlock(&txq->_xmit_lock);
3194 }
3195
3196 static inline void __netif_tx_unlock_bh(struct netdev_queue *txq)
3197 {
3198         txq->xmit_lock_owner = -1;
3199         spin_unlock_bh(&txq->_xmit_lock);
3200 }
3201
3202 static inline void txq_trans_update(struct netdev_queue *txq)
3203 {
3204         if (txq->xmit_lock_owner != -1)
3205                 txq->trans_start = jiffies;
3206 }
3207
3208 /**
3209  *      netif_tx_lock - grab network device transmit lock
3210  *      @dev: network device
3211  *
3212  * Get network device transmit lock
3213  */
3214 static inline void netif_tx_lock(struct net_device *dev)
3215 {
3216         unsigned int i;
3217         int cpu;
3218
3219         spin_lock(&dev->tx_global_lock);
3220         cpu = smp_processor_id();
3221         for (i = 0; i < dev->num_tx_queues; i++) {
3222                 struct netdev_queue *txq = netdev_get_tx_queue(dev, i);
3223
3224                 /* We are the only thread of execution doing a
3225                  * freeze, but we have to grab the _xmit_lock in
3226                  * order to synchronize with threads which are in
3227                  * the ->hard_start_xmit() handler and already
3228                  * checked the frozen bit.
3229                  */
3230                 __netif_tx_lock(txq, cpu);
3231                 set_bit(__QUEUE_STATE_FROZEN, &txq->state);
3232                 __netif_tx_unlock(txq);
3233         }
3234 }
3235
3236 static inline void netif_tx_lock_bh(struct net_device *dev)
3237 {
3238         local_bh_disable();
3239         netif_tx_lock(dev);
3240 }
3241
3242 static inline void netif_tx_unlock(struct net_device *dev)
3243 {
3244         unsigned int i;
3245
3246         for (i = 0; i < dev->num_tx_queues; i++) {
3247                 struct netdev_queue *txq = netdev_get_tx_queue(dev, i);
3248
3249                 /* No need to grab the _xmit_lock here.  If the
3250                  * queue is not stopped for another reason, we
3251                  * force a schedule.
3252                  */
3253                 clear_bit(__QUEUE_STATE_FROZEN, &txq->state);
3254                 netif_schedule_queue(txq);
3255         }
3256         spin_unlock(&dev->tx_global_lock);
3257 }
3258
3259 static inline void netif_tx_unlock_bh(struct net_device *dev)
3260 {
3261         netif_tx_unlock(dev);
3262         local_bh_enable();
3263 }
3264
3265 #define HARD_TX_LOCK(dev, txq, cpu) {                   \
```

```
3266         if ((dev->features & NETIF_F_LLTX) == 0) {      \
3267                 __netif_tx_lock(txq, cpu);              \
3268         }                                               \
3269 }
3270
3271 #define HARD_TX_TRYLOCK(dev, txq)                       \
3272         (((dev->features & NETIF_F_LLTX) == 0) ?        \
3273                 __netif_tx_trylock(txq) :               \
3274                 true )
3275
3276 #define HARD_TX_UNLOCK(dev, txq) {                      \
3277         if ((dev->features & NETIF_F_LLTX) == 0) {      \
3278                 __netif_tx_unlock(txq);                 \
3279         }                                               \
3280 }
3281
3282 static inline void netif_tx_disable(struct net_device *dev)
3283 {
3284         unsigned int i;
3285         int cpu;
3286
3287         local_bh_disable();
3288         cpu = smp_processor_id();
3289         for (i = 0; i < dev->num_tx_queues; i++) {
3290                 struct netdev_queue *txq = netdev_get_tx_queue(dev, i);
3291
3292                 __netif_tx_lock(txq, cpu);
3293                 netif_tx_stop_queue(txq);
3294                 __netif_tx_unlock(txq);
3295         }
3296         local_bh_enable();
3297 }
3298
3299 static inline void netif_addr_lock(struct net_device *dev)
3300 {
3301         spin_lock(&dev->addr_list_lock);
3302 }
3303
3304 static inline void netif_addr_lock_nested(struct net_device *dev)
3305 {
3306         int subclass = SINGLE_DEPTH_NESTING;
3307
3308         if (dev->netdev_ops->ndo_get_lock_subclass)
3309                 subclass = dev->netdev_ops->ndo_get_lock_subclass(dev);
3310
3311         spin_lock_nested(&dev->addr_list_lock, subclass);
3312 }
3313
3314 static inline void netif_addr_lock_bh(struct net_device *dev)
3315 {
3316         spin_lock_bh(&dev->addr_list_lock);
3317 }
3318
3319 static inline void netif_addr_unlock(struct net_device *dev)
3320 {
3321         spin_unlock(&dev->addr_list_lock);
3322 }
3323
3324 static inline void netif_addr_unlock_bh(struct net_device *dev)
3325 {
3326         spin_unlock_bh(&dev->addr_list_lock);
3327 }
3328
3329 /*
3330  * dev_addrs walker. Should be used only for read access. Call with
3331  * rcu_read_lock held.
3332  */
3333 #define for_each_dev_addr(dev, ha) \
3334                 list_for_each_entry_rcu(ha, &dev->dev_addrs.list, list)
3335
3336 /* These functions live elsewhere (drivers/net/net_init.c, but related) */
3337
3338 void ether_setup(struct net_device *dev);
```

```
3339
3340 /* Support for loadable net-drivers */
3341 struct net_device *alloc_netdev_mqs(int sizeof_priv, const char *name,
3342                                     unsigned char name_assign_type,
3343                                     void (*setup)(struct net_device *),
3344                                     unsigned int txqs, unsigned int rxqs);
3345 #define alloc_netdev(sizeof_priv, name, name_assign_type, setup) \
3346         alloc_netdev_mqs(sizeof_priv, name, name_assign_type, setup, 1, 1)
3347
3348 #define alloc_netdev_mq(sizeof_priv, name, name_assign_type, setup, count) \
3349         alloc_netdev_mqs(sizeof_priv, name, name_assign_type, setup, count, \
3350                          count)
3351
3352 int register_netdev(struct net_device *dev);
3353 void unregister_netdev(struct net_device *dev);
3354
3355 /* General hardware address lists handling functions */
3356 int __hw_addr_sync(struct netdev_hw_addr_list *to_list,
3357                    struct netdev_hw_addr_list *from_list, int addr_len);
3358 void __hw_addr_unsync(struct netdev_hw_addr_list *to_list,
3359                       struct netdev_hw_addr_list *from_list, int addr_len);
3360 int __hw_addr_sync_dev(struct netdev_hw_addr_list *list,
3361                        struct net_device *dev,
3362                        int (*sync)(struct net_device *, const unsigned char *),
3363                        int (*unsync)(struct net_device *,
3364                                      const unsigned char *));
3365 void __hw_addr_unsync_dev(struct netdev_hw_addr_list *list,
3366                           struct net_device *dev,
3367                           int (*unsync)(struct net_device *,
3368                                         const unsigned char *));
3369 void __hw_addr_init(struct netdev_hw_addr_list *list);
3370
3371 /* Functions used for device addresses handling */
3372 int dev_addr_add(struct net_device *dev, const unsigned char *addr,
3373                  unsigned char addr_type);
3374 int dev_addr_del(struct net_device *dev, const unsigned char *addr,
3375                  unsigned char addr_type);
3376 void dev_addr_flush(struct net_device *dev);
3377 int dev_addr_init(struct net_device *dev);
3378
3379 /* Functions used for unicast addresses handling */
3380 int dev_uc_add(struct net_device *dev, const unsigned char *addr);
3381 int dev_uc_add_excl(struct net_device *dev, const unsigned char *addr);
3382 int dev_uc_del(struct net_device *dev, const unsigned char *addr);
3383 int dev_uc_sync(struct net_device *to, struct net_device *from);
3384 int dev_uc_sync_multiple(struct net_device *to, struct net_device *from);
3385 void dev_uc_unsync(struct net_device *to, struct net_device *from);
3386 void dev_uc_flush(struct net_device *dev);
3387 void dev_uc_init(struct net_device *dev);
3388
3389 /**
3390  *  __dev_uc_sync - Synchonize device's unicast list
3391  *  @dev:  device to sync
3392  *  @sync: function to call if address should be added
3393  *  @unsync: function to call if address should be removed
3394  *
3395  *  Add newly added addresses to the interface, and release
3396  *  addresses that have been deleted.
3397  **/
3398 static inline int __dev_uc_sync(struct net_device *dev,
3399                                 int (*sync)(struct net_device *,
3400                                             const unsigned char *),
3401                                 int (*unsync)(struct net_device *,
3402                                               const unsigned char *))
3403 {
3404         return __hw_addr_sync_dev(&dev->uc, dev, sync, unsync);
3405 }
3406
3407 /**
3408  *  __dev_uc_unsync - Remove synchronized addresses from device
3409  *  @dev:  device to sync
3410  *  @unsync: function to call if address should be removed
3411  *
```

```
3412  *  Remove all addresses that were added to the device by dev_uc_sync().
3413  **/
3414 static inline void __dev_uc_unsync(struct net_device *dev,
3415                                    int (*unsync)(struct net_device *,
3416                                                  const unsigned char *))
3417 {
3418         __hw_addr_unsync_dev(&dev->uc, dev, unsync);
3419 }
3420
3421 /* Functions used for multicast addresses handling */
3422 int dev_mc_add(struct net_device *dev, const unsigned char *addr);
3423 int dev_mc_add_global(struct net_device *dev, const unsigned char *addr);
3424 int dev_mc_add_excl(struct net_device *dev, const unsigned char *addr);
3425 int dev_mc_del(struct net_device *dev, const unsigned char *addr);
3426 int dev_mc_del_global(struct net_device *dev, const unsigned char *addr);
3427 int dev_mc_sync(struct net_device *to, struct net_device *from);
3428 int dev_mc_sync_multiple(struct net_device *to, struct net_device *from);
3429 void dev_mc_unsync(struct net_device *to, struct net_device *from);
3430 void dev_mc_flush(struct net_device *dev);
3431 void dev_mc_init(struct net_device *dev);
3432
3433 /**
3434  *  __dev_mc_sync - Synchonize device's multicast list
3435  *  @dev:  device to sync
3436  *  @sync: function to call if address should be added
3437  *  @unsync: function to call if address should be removed
3438  *
3439  *  Add newly added addresses to the interface, and release
3440  *  addresses that have been deleted.
3441  **/
3442 static inline int __dev_mc_sync(struct net_device *dev,
3443                                 int (*sync)(struct net_device *,
3444                                             const unsigned char *),
3445                                 int (*unsync)(struct net_device *,
3446                                               const unsigned char *))
3447 {
3448         return __hw_addr_sync_dev(&dev->mc, dev, sync, unsync);
3449 }
3450
3451 /**
3452  *  __dev_mc_unsync - Remove synchronized addresses from device
3453  *  @dev:  device to sync
3454  *  @unsync: function to call if address should be removed
3455  *
3456  *  Remove all addresses that were added to the device by dev_mc_sync().
3457  **/
3458 static inline void __dev_mc_unsync(struct net_device *dev,
3459                                    int (*unsync)(struct net_device *,
3460                                                  const unsigned char *))
3461 {
3462         __hw_addr_unsync_dev(&dev->mc, dev, unsync);
3463 }
3464
3465 /* Functions used for secondary unicast and multicast support */
3466 void dev_set_rx_mode(struct net_device *dev);
3467 void __dev_set_rx_mode(struct net_device *dev);
3468 int dev_set_promiscuity(struct net_device *dev, int inc);
3469 int dev_set_allmulti(struct net_device *dev, int inc);
3470 void netdev_state_change(struct net_device *dev);
3471 void netdev_notify_peers(struct net_device *dev);
3472 void netdev_features_change(struct net_device *dev);
3473 /* Load a device via the kmod */
3474 void dev_load(struct net *net, const char *name);
3475 struct rtnl_link_stats64 *dev_get_stats(struct net_device *dev,
3476                                         struct rtnl_link_stats64 *storage);
3477 void netdev_stats_to_stats64(struct rtnl_link_stats64 *stats64,
3478                              const struct net_device_stats *netdev_stats);
3479
3480 extern int              netdev_max_backlog;
3481 extern int              netdev_tstamp_prequeue;
3482 extern int              weight_p;
3483 extern int              bpf_jit_enable;
3484
```

```
3485 bool netdev_has_upper_dev(struct net_device *dev, struct net_device *upper_dev);
3486 struct net_device *netdev_upper_get_next_dev_rcu(struct net_device *dev,
3487                                                  struct list_head **iter);
3488 struct net_device *netdev_all_upper_get_next_dev_rcu(struct net_device *dev,
3489                                                  struct list_head **iter);
3490
3491 /* iterate through upper list, must be called under RCU read lock */
3492 #define netdev_for_each_upper_dev_rcu(dev, updev, iter) \
3493         for (iter = &(dev)->adj_list.upper, \
3494              updev = netdev_upper_get_next_dev_rcu(dev, &(iter)); \
3495              updev; \
3496              updev = netdev_upper_get_next_dev_rcu(dev, &(iter)))
3497
3498 /* iterate through upper list, must be called under RCU read lock */
3499 #define netdev_for_each_all_upper_dev_rcu(dev, updev, iter) \
3500         for (iter = &(dev)->all_adj_list.upper, \
3501              updev = netdev_all_upper_get_next_dev_rcu(dev, &(iter)); \
3502              updev; \
3503              updev = netdev_all_upper_get_next_dev_rcu(dev, &(iter)))
3504
3505 void *netdev_lower_get_next_private(struct net_device *dev,
3506                                     struct list_head **iter);
3507 void *netdev_lower_get_next_private_rcu(struct net_device *dev,
3508                                         struct list_head **iter);
3509
3510 #define netdev_for_each_lower_private(dev, priv, iter) \
3511         for (iter = (dev)->adj_list.lower.next, \
3512              priv = netdev_lower_get_next_private(dev, &(iter)); \
3513              priv; \
3514              priv = netdev_lower_get_next_private(dev, &(iter)))
3515
3516 #define netdev_for_each_lower_private_rcu(dev, priv, iter) \
3517         for (iter = &(dev)->adj_list.lower, \
3518              priv = netdev_lower_get_next_private_rcu(dev, &(iter)); \
3519              priv; \
3520              priv = netdev_lower_get_next_private_rcu(dev, &(iter)))
3521
3522 void *netdev_lower_get_next(struct net_device *dev,
3523                                 struct list_head **iter);
3524 #define netdev_for_each_lower_dev(dev, ldev, iter) \
3525         for (iter = &(dev)->adj_list.lower, \
3526              ldev = netdev_lower_get_next(dev, &(iter)); \
3527              ldev; \
3528              ldev = netdev_lower_get_next(dev, &(iter)))
3529
3530 void *netdev_adjacent_get_private(struct list_head *adj_list);
3531 void *netdev_lower_get_first_private_rcu(struct net_device *dev);
3532 struct net_device *netdev_master_upper_dev_get(struct net_device *dev);
3533 struct net_device *netdev_master_upper_dev_get_rcu(struct net_device *dev);
3534 int netdev_upper_dev_link(struct net_device *dev, struct net_device *upper_dev);
3535 int netdev_master_upper_dev_link(struct net_device *dev,
3536                                  struct net_device *upper_dev);
3537 int netdev_master_upper_dev_link_private(struct net_device *dev,
3538                                          struct net_device *upper_dev,
3539                                          void *private);
3540 void netdev_upper_dev_unlink(struct net_device *dev,
3541                              struct net_device *upper_dev);
3542 void netdev_adjacent_rename_links(struct net_device *dev, char *oldname);
3543 void *netdev_lower_dev_get_private(struct net_device *dev,
3544                                    struct net_device *lower_dev);
3545
3546 /* RSS keys are 40 or 52 bytes long */
3547 #define NETDEV_RSS_KEY_LEN 52
3548 extern u8 netdev_rss_key[NETDEV_RSS_KEY_LEN];
3549 void netdev_rss_key_fill(void *buffer, size_t len);
3550
3551 int dev_get_nest_level(struct net_device *dev,
3552                        bool (*type_check)(struct net_device *dev));
3553 int skb_checksum_help(struct sk_buff *skb);
3554 struct sk_buff *__skb_gso_segment(struct sk_buff *skb,
3555                                   netdev_features_t features, bool tx_path);
3556 struct sk_buff *skb_mac_gso_segment(struct sk_buff *skb,
3557                                     netdev_features_t features);
```

```
3558
3559 struct netdev_bonding_info {
3560         ifslave slave;
3561         ifbond  master;
3562 };
3563
3564 struct netdev_notifier_bonding_info {
3565         struct netdev_notifier_info info; /* must be first */
3566         struct netdev_bonding_info  bonding_info;
3567 };
3568
3569 void netdev_bonding_info_change(struct net_device *dev,
3570                                 struct netdev_bonding_info *bonding_info);
3571
3572 static inline
3573 struct sk_buff *skb_gso_segment(struct sk_buff *skb, netdev_features_t features)
3574 {
3575         return __skb_gso_segment(skb, features, true);
3576 }
3577 __be16 skb_network_protocol(struct sk_buff *skb, int *depth);
3578
3579 static inline bool can_checksum_protocol(netdev_features_t features,
3580                                          __be16 protocol)
3581 {
3582         return ((features & NETIF_F_GEN_CSUM) ||
3583                 ((features & NETIF_F_V4_CSUM) &&
3584                  protocol == htons(ETH_P_IP)) ||
3585                 ((features & NETIF_F_V6_CSUM) &&
3586                  protocol == htons(ETH_P_IPV6)) ||
3587                 ((features & NETIF_F_FCOE_CRC) &&
3588                  protocol == htons(ETH_P_FCOE)));
3589 }
3590
3591 #ifdef CONFIG_BUG
3592 void netdev_rx_csum_fault(struct net_device *dev);
3593 #else
3594 static inline void netdev_rx_csum_fault(struct net_device *dev)
3595 {
3596 }
3597 #endif
3598 /* rx skb timestamps */
3599 void net_enable_timestamp(void);
3600 void net_disable_timestamp(void);
3601
3602 #ifdef CONFIG_PROC_FS
3603 int __init dev_proc_init(void);
3604 #else
3605 #define dev_proc_init() 0
3606 #endif
3607
3608 static inline netdev_tx_t __netdev_start_xmit(const struct net_device_ops *ops,
3609                                               struct sk_buff *skb, struct net_device *dev,
3610                                               bool more)
3611 {
3612         skb->xmit_more = more ? 1 : 0;
3613         return ops->ndo_start_xmit(skb, dev);
3614 }
3615
3616 static inline netdev_tx_t netdev_start_xmit(struct sk_buff *skb, struct net_device *dev,
3617                                             struct netdev_queue *txq, bool more)
3618 {
3619         const struct net_device_ops *ops = dev->netdev_ops;
3620         int rc;
3621
3622         rc = __netdev_start_xmit(ops, skb, dev, more);
3623         if (rc == NETDEV_TX_OK)
3624                 txq_trans_update(txq);
3625
3626         return rc;
3627 }
3628
3629 int netdev_class_create_file_ns(struct class_attribute *class_attr,
3630                                 const void *ns);
```

```
3631 void netdev_class_remove_file_ns(struct class_attribute *class_attr,
3632                                   const void *ns);
3633
3634 static inline int netdev_class_create_file(struct class_attribute *class_attr)
3635 {
3636         return netdev_class_create_file_ns(class_attr, NULL);
3637 }
3638
3639 static inline void netdev_class_remove_file(struct class_attribute *class_attr)
3640 {
3641         netdev_class_remove_file_ns(class_attr, NULL);
3642 }
3643
3644 extern struct kobj_ns_type_operations net_ns_type_operations;
3645
3646 const char *netdev_drivername(const struct net_device *dev);
3647
3648 void linkwatch_run_queue(void);
3649
3650 static inline netdev_features_t netdev_intersect_features(netdev_features_t f1,
3651                                                           netdev_features_t f2)
3652 {
3653         if (f1 & NETIF_F_GEN_CSUM)
3654                 f1 |= (NETIF_F_ALL_CSUM & ~NETIF_F_GEN_CSUM);
3655         if (f2 & NETIF_F_GEN_CSUM)
3656                 f2 |= (NETIF_F_ALL_CSUM & ~NETIF_F_GEN_CSUM);
3657         f1 &= f2;
3658         if (f1 & NETIF_F_GEN_CSUM)
3659                 f1 &= ~(NETIF_F_ALL_CSUM & ~NETIF_F_GEN_CSUM);
3660
3661         return f1;
3662 }
3663
3664 static inline netdev_features_t netdev_get_wanted_features(
3665         struct net_device *dev)
3666 {
3667         return (dev->features & ~dev->hw_features) | dev->wanted_features;
3668 }
3669 netdev_features_t netdev_increment_features(netdev_features_t all,
3670         netdev_features_t one, netdev_features_t mask);
3671
3672 /* Allow TSO being used on stacked device :
3673  * Performing the GSO segmentation before last device
3674  * is a performance improvement.
3675  */
3676 static inline netdev_features_t netdev_add_tso_features(netdev_features_t features,
3677                                                         netdev_features_t mask)
3678 {
3679         return netdev_increment_features(features, NETIF_F_ALL_TSO, mask);
3680 }
3681
3682 int __netdev_update_features(struct net_device *dev);
3683 void netdev_update_features(struct net_device *dev);
3684 void netdev_change_features(struct net_device *dev);
3685
3686 void netif_stacked_transfer_operstate(const struct net_device *rootdev,
3687                                       struct net_device *dev);
3688
3689 netdev_features_t passthru_features_check(struct sk_buff *skb,
3690                                           struct net_device *dev,
3691                                           netdev_features_t features);
3692 netdev_features_t netif_skb_features(struct sk_buff *skb);
3693
3694 static inline bool net_gso_ok(netdev_features_t features, int gso_type)
3695 {
3696         netdev_features_t feature = gso_type << NETIF_F_GSO_SHIFT;
3697
3698         /* check flags correspondence */
3699         BUILD_BUG_ON(SKB_GSO_TCPV4   != (NETIF_F_TSO >> NETIF_F_GSO_SHIFT));
3700         BUILD_BUG_ON(SKB_GSO_UDP     != (NETIF_F_UFO >> NETIF_F_GSO_SHIFT));
3701         BUILD_BUG_ON(SKB_GSO_DODGY   != (NETIF_F_GSO_ROBUST >> NETIF_F_GSO_SHIFT));
3702         BUILD_BUG_ON(SKB_GSO_TCP_ECN != (NETIF_F_TSO_ECN >> NETIF_F_GSO_SHIFT));
3703         BUILD_BUG_ON(SKB_GSO_TCPV6   != (NETIF_F_TSO6 >> NETIF_F_GSO_SHIFT));
```

```
3704         BUILD_BUG_ON(SKB_GSO_FCOE     != (NETIF_F_FSO >> NETIF_F_GSO_SHIFT));
3705         BUILD_BUG_ON(SKB_GSO_GRE      != (NETIF_F_GSO_GRE >> NETIF_F_GSO_SHIFT));
3706         BUILD_BUG_ON(SKB_GSO_GRE_CSUM != (NETIF_F_GSO_GRE_CSUM >> NETIF_F_GSO_SHIFT));
3707         BUILD_BUG_ON(SKB_GSO_IPIP     != (NETIF_F_GSO_IPIP >> NETIF_F_GSO_SHIFT));
3708         BUILD_BUG_ON(SKB_GSO_SIT      != (NETIF_F_GSO_SIT >> NETIF_F_GSO_SHIFT));
3709         BUILD_BUG_ON(SKB_GSO_UDP_TUNNEL != (NETIF_F_GSO_UDP_TUNNEL >> NETIF_F_GSO_SHIFT));
3710         BUILD_BUG_ON(SKB_GSO_UDP_TUNNEL_CSUM != (NETIF_F_GSO_UDP_TUNNEL_CSUM >> NETIF_F_GSO_SHIFT));
3711         BUILD_BUG_ON(SKB_GSO_TUNNEL_REMCSUM != (NETIF_F_GSO_TUNNEL_REMCSUM >> NETIF_F_GSO_SHIFT));
3712
3713         return (features & feature) == feature;
3714 }
3715
3716 static inline bool skb_gso_ok(struct sk_buff *skb, netdev_features_t features)
3717 {
3718         return net_gso_ok(features, skb_shinfo(skb)->gso_type) &&
3719                (!skb_has_frag_list(skb) || (features & NETIF_F_FRAGLIST));
3720 }
3721
3722 static inline bool netif_needs_gso(struct sk_buff *skb,
3723                                    netdev_features_t features)
3724 {
3725         return skb_is_gso(skb) && (!skb_gso_ok(skb, features) ||
3726                 unlikely((skb->ip_summed != CHECKSUM_PARTIAL) &&
3727                          (skb->ip_summed != CHECKSUM_UNNECESSARY)));
3728 }
3729
3730 static inline void netif_set_gso_max_size(struct net_device *dev,
3731                                           unsigned int size)
3732 {
3733         dev->gso_max_size = size;
3734 }
3735
3736 static inline void skb_gso_error_unwind(struct sk_buff *skb, __be16 protocol,
3737                                         int pulled_hlen, u16 mac_offset,
3738                                         int mac_len)
3739 {
3740         skb->protocol = protocol;
3741         skb->encapsulation = 1;
3742         skb_push(skb, pulled_hlen);
3743         skb_reset_transport_header(skb);
3744         skb->mac_header = mac_offset;
3745         skb->network_header = skb->mac_header + mac_len;
3746         skb->mac_len = mac_len;
3747 }
3748
3749 static inline bool netif_is_macvlan(struct net_device *dev)
3750 {
3751         return dev->priv_flags & IFF_MACVLAN;
3752 }
3753
3754 static inline bool netif_is_macvlan_port(struct net_device *dev)
3755 {
3756         return dev->priv_flags & IFF_MACVLAN_PORT;
3757 }
3758
3759 static inline bool netif_is_ipvlan(struct net_device *dev)
3760 {
3761         return dev->priv_flags & IFF_IPVLAN_SLAVE;
3762 }
3763
3764 static inline bool netif_is_ipvlan_port(struct net_device *dev)
3765 {
3766         return dev->priv_flags & IFF_IPVLAN_MASTER;
3767 }
3768
3769 static inline bool netif_is_bond_master(struct net_device *dev)
3770 {
3771         return dev->flags & IFF_MASTER && dev->priv_flags & IFF_BONDING;
3772 }
3773
3774 static inline bool netif_is_bond_slave(struct net_device *dev)
3775 {
3776         return dev->flags & IFF_SLAVE && dev->priv_flags & IFF_BONDING;
```

```
3777 }
3778
3779 static inline bool netif_supports_nofcs(struct net_device *dev)
3780 {
3781         return dev->priv_flags & IFF_SUPP_NOFCS;
3782 }
3783
3784 /* This device needs to keep skb dst for qdisc enqueue or ndo_start_xmit() */
3785 static inline void netif_keep_dst(struct net_device *dev)
3786 {
3787         dev->priv_flags &= ~(IFF_XMIT_DST_RELEASE | IFF_XMIT_DST_RELEASE_PERM);
3788 }
3789
3790 extern struct pernet_operations __net_initdata loopback_net_ops;
3791
3792 /* Logging, debugging and troubleshooting/diagnostic helpers. */
3793
3794 /* netdev_printk helpers, similar to dev_printk */
3795
3796 static inline const char *netdev_name(const struct net_device *dev)
3797 {
3798         if (!dev->name[0] || strchr(dev->name, '%'))
3799                 return "(unnamed net_device)";
3800         return dev->name;
3801 }
3802
3803 static inline const char *netdev_reg_state(const struct net_device *dev)
3804 {
3805         switch (dev->reg_state) {
3806         case NETREG_UNINITIALIZED: return " (uninitialized)";
3807         case NETREG_REGISTERED: return "";
3808         case NETREG_UNREGISTERING: return " (unregistering)";
3809         case NETREG_UNREGISTERED: return " (unregistered)";
3810         case NETREG_RELEASED: return " (released)";
3811         case NETREG_DUMMY: return " (dummy)";
3812         }
3813
3814         WARN_ONCE(1, "%s: unknown reg_state %d\n", dev->name, dev->reg_state);
3815         return " (unknown)";
3816 }
3817
3818 __printf(3, 4)
3819 void netdev_printk(const char *level, const struct net_device *dev,
3820                    const char *format, ...);
3821 __printf(2, 3)
3822 void netdev_emerg(const struct net_device *dev, const char *format, ...);
3823 __printf(2, 3)
3824 void netdev_alert(const struct net_device *dev, const char *format, ...);
3825 __printf(2, 3)
3826 void netdev_crit(const struct net_device *dev, const char *format, ...);
3827 __printf(2, 3)
3828 void netdev_err(const struct net_device *dev, const char *format, ...);
3829 __printf(2, 3)
3830 void netdev_warn(const struct net_device *dev, const char *format, ...);
3831 __printf(2, 3)
3832 void netdev_notice(const struct net_device *dev, const char *format, ...);
3833 __printf(2, 3)
3834 void netdev_info(const struct net_device *dev, const char *format, ...);
3835
3836 #define MODULE_ALIAS_NETDEV(device) \
3837         MODULE_ALIAS("netdev-" device)
3838
3839 #if defined(CONFIG_DYNAMIC_DEBUG)
3840 #define netdev_dbg(__dev, format, args...)                      \
3841 do {                                                            \
3842         dynamic_netdev_dbg(__dev, format, ##args);              \
3843 } while (0)
3844 #elif defined(DEBUG)
3845 #define netdev_dbg(__dev, format, args...)                      \
3846         netdev_printk(KERN_DEBUG, __dev, format, ##args)
3847 #else
3848 #define netdev_dbg(__dev, format, args...)                      \
3849 ({                                                              \
```

```
3850         if (0)                                                  \
3851                 netdev_printk(KERN_DEBUG, __dev, format, ##args); \
3852 })
3853 #endif
3854
3855 #if defined(VERBOSE_DEBUG)
3856 #define netdev_vdbg     netdev_dbg
3857 #else
3858
3859 #define netdev_vdbg(dev, format, args...)                       \
3860 ({                                                              \
3861         if (0)                                                  \
3862                 netdev_printk(KERN_DEBUG, dev, format, ##args); \
3863         0;                                                      \
3864 })
3865 #endif
3866
3867 /*
3868  * netdev_WARN() acts like dev_printk(), but with the key difference
3869  * of using a WARN/WARN_ON to get the message out, including the
3870  * file/line information and a backtrace.
3871  */
3872 #define netdev_WARN(dev, format, args...)                       \
3873         WARN(1, "netdevice: %s%s\n" format, netdev_name(dev),   \
3874              netdev_reg_state(dev), ##args)
3875
3876 /* netif printk helpers, similar to netdev_printk */
3877
3878 #define netif_printk(priv, type, level, dev, fmt, args...)      \
3879 do {                                                            \
3880         if (netif_msg_##type(priv))                             \
3881                 netdev_printk(level, (dev), fmt, ##args);       \
3882 } while (0)
3883
3884 #define netif_level(level, priv, type, dev, fmt, args...)       \
3885 do {                                                            \
3886         if (netif_msg_##type(priv))                             \
3887                 netdev_##level(dev, fmt, ##args);               \
3888 } while (0)
3889
3890 #define netif_emerg(priv, type, dev, fmt, args...)              \
3891         netif_level(emerg, priv, type, dev, fmt, ##args)
3892 #define netif_alert(priv, type, dev, fmt, args...)              \
3893         netif_level(alert, priv, type, dev, fmt, ##args)
3894 #define netif_crit(priv, type, dev, fmt, args...)               \
3895         netif_level(crit, priv, type, dev, fmt, ##args)
3896 #define netif_err(priv, type, dev, fmt, args...)                \
3897         netif_level(err, priv, type, dev, fmt, ##args)
3898 #define netif_warn(priv, type, dev, fmt, args...)               \
3899         netif_level(warn, priv, type, dev, fmt, ##args)
3900 #define netif_notice(priv, type, dev, fmt, args...)             \
3901         netif_level(notice, priv, type, dev, fmt, ##args)
3902 #define netif_info(priv, type, dev, fmt, args...)               \
3903         netif_level(info, priv, type, dev, fmt, ##args)
3904
3905 #if defined(CONFIG_DYNAMIC_DEBUG)
3906 #define netif_dbg(priv, type, netdev, format, args...)          \
3907 do {                                                            \
3908         if (netif_msg_##type(priv))                             \
3909                 dynamic_netdev_dbg(netdev, format, ##args);     \
3910 } while (0)
3911 #elif defined(DEBUG)
3912 #define netif_dbg(priv, type, dev, format, args...)             \
3913         netif_printk(priv, type, KERN_DEBUG, dev, format, ##args)
3914 #else
3915 #define netif_dbg(priv, type, dev, format, args...)                     \
3916 ({                                                                      \
3917         if (0)                                                          \
3918                 netif_printk(priv, type, KERN_DEBUG, dev, format, ##args); \
3919         0;                                                              \
3920 })
3921 #endif
3922
```

```
3923 #if defined(VERBOSE_DEBUG)
3924 #define netif_vdbg      netif_dbg
3925 #else
3926 #define netif_vdbg(priv, type, dev, format, args...)            \
3927 ({                                                              \
3928         if (0)                                                 \
3929                 netif_printk(priv, type, KERN_DEBUG, dev, format, ##args); \
3930         0;                                                     \
3931 })
3932 #endif
3933
3934 /*
3935  *      The list of packet types we will receive (as opposed to discard)
3936  *      and the routines to invoke.
3937  *
3938  *      Why 16. Because with 16 the only overlap we get on a hash of the
3939  *      low nibble of the protocol value is RARP/SNAP/X.25.
3940  *
3941  *      NOTE:  That is no longer true with the addition of VLAN tags.  Not
3942  *             sure which should go first, but I bet it won't make much
3943  *             difference if we are running VLANs.  The good news is that
3944  *             this protocol won't be in the list unless compiled in, so
3945  *             the average user (w/out VLANs) will not be adversely affected.
3946  *             --BLG
3947  *
3948  *              0800    IP
3949  *              8100    802.1Q VLAN
3950  *              0001    802.3
3951  *              0002    AX.25
3952  *              0004    802.2
3953  *              8035    RARP
3954  *              0005    SNAP
3955  *              0805    X.25
3956  *              0806    ARP
3957  *              8137    IPX
3958  *              0009    Localtalk
3959  *              86DD    IPv6
3960  */
3961 #define PTYPE_HASH_SIZE (16)
3962 #define PTYPE_HASH_MASK (PTYPE_HASH_SIZE - 1)
3963
3964 #endif  /* _LINUX_NETDEVICE_H */
3965
```