Linux Cross Reference

Free Electrons

Embedded Linux Experts

• *source navigation* • diff markup • identifier search • freetext search •

Version: 2.0.40 2.2.26 2.4.37 3.6 3.7 3.8 3.9 3.10 3.11 3.12 3.13 3.14 3.15 3.16 3.17 3.18 3.19 4.0 4.1 ***4.2***

Linux/include/linux/netdevice.h

```
/*
 * INET         An implementation of the TCP/IP protocol suite for the LINUX
 *              operating system.  INET is implemented using the  BSD Socket
 *              interface as the means of communication with the user level.
 *
 *              Definitions for the Interfaces handler.
 *
 * Version:     @(#)dev.h       1.0.10  08/12/93
 *
 * Authors:     Ross Biro
 *              Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
 *              Corey Minyard <wf-rch!minyard@relay.EU.net>
 *              Donald J. Becker, <becker@cesdis.gsfc.nasa.gov>
 *              Alan Cox, <alan@lxorguk.ukuu.org.uk>
 *              Bjorn Ekwall. <bj0rn@blox.se>
 *              Pekka Riikonen <priikone@poseidon.pspt.fi>
 *
 *              This program is free software; you can redistribute it and/or
 *              modify it under the terms of the GNU General Public License
 *              as published by the Free Software Foundation; either version
 *              2 of the License, or (at your option) any later version.
 *
 *              Moved to /usr/include/linux for NET3
 */
#ifndef _LINUX_NETDEVICE_H
#define _LINUX_NETDEVICE_H

#include <linux/timer.h>
#include <linux/bug.h>
#include <linux/delay.h>
#include <linux/atomic.h>
#include <linux/prefetch.h>
#include <asm/cache.h>
#include <asm/byteorder.h>

#include <linux/percpu.h>
#include <linux/rculist.h>
#include <linux/dmaengine.h>
#include <linux/workqueue.h>
#include <linux/dynamic_queue_limits.h>

#include <linux/ethtool.h>
#include <net/net_namespace.h>
#include <net/dsa.h>
#ifdef CONFIG_DCB
#include <net/dcbnl.h>
#endif
#include <net/netprio_cgroup.h>

#include <linux/netdev_features.h>
#include <linux/neighbour.h>
#include <uapi/linux/netdevice.h>
#include <uapi/linux/if_bonding.h>
```

```

struct netpoll_info;
struct device;
struct phy_device;
/* 802.11 specific */
struct wireless_dev;
/* 802.15.4 specific */
struct wpan_dev;
struct mpls_dev;

void netdev_set_default_ethtool_ops(struct net_device *dev,
                                    const struct ethtool_ops *ops);

/* Backlog congestion levels */
#define NET_RX_SUCCESS          0       /* keep 'em coming, baby */
#define NET_RX_DROP             1       /* packet dropped */

/*
 * Transmit return codes: transmit return codes originate from three different
 * namespaces:
 *
 * - qdisc return codes
 * - driver transmit return codes
 * - errno values
 *
 * Drivers are allowed to return any one of those in their hard_start_xmit()
 * function. Real network devices commonly used with qdiscs should only return
 * the driver transmit return codes though - when qdiscs are used, the actual
 * transmission happens asynchronously, so the value is not propagated to
 * higher layers. Virtual network devices transmit synchronously, in this case
 * the driver transmit return codes are consumed by dev_queue_xmit(), all
 * others are propagated to higher layers.
 */

/* qdisc ->enqueue() return codes. */
#define NET_XMIT_SUCCESS        0x00
#define NET_XMIT_DROP           0x01    /* skb dropped                  */
#define NET_XMIT_CN             0x02    /* congestion notification      */
#define NET_XMIT_POLICED        0x03    /* skb is shot by police        */
#define NET_XMIT_MASK           0x0f    /* qdisc flags in net/sch_generic.h */

/* NET_XMIT_CN is special. It does not guarantee that this packet is lost. It
 * indicates that the device will soon be dropping packets, or already drops
 * some packets of the same priority; prompting us to send less aggressively. */
#define net_xmit_eval(e)        ((e) == NET_XMIT_CN ? 0 : (e))
#define net_xmit_errno(e)       ((e) != NET_XMIT_CN ? -ENOBUFS : 0)

/* Driver transmit return codes */
#define NETDEV_TX_MASK          0xf0

enum netdev_tx {
        __NETDEV_TX_MIN  = INT_MIN,     /* make sure enum is signed */
        NETDEV_TX_OK     = 0x00,        /* driver took care of packet */
        NETDEV_TX_BUSY   = 0x10,        /* driver tx path was busy*/
        NETDEV_TX_LOCKED = 0x20,        /* driver tx lock was already taken */
};
typedef enum netdev_tx netdev_tx_t;

/*
 * Current order: NETDEV_TX_MASK > NET_XMIT_MASK >= 0 is significant;
 * hard_start_xmit() return < NET_XMIT_MASK means skb was consumed.
 */
static inline bool dev_xmit_complete(int rc)
{
        /*
         * Positive cases with an skb consumed by a driver:
         * - successful transmission (rc == NETDEV_TX_OK)
         * - error while transmitting (rc < 0)
         * - error while queueing to a different device (rc & NET_XMIT_MASK)
         */
        if (likely(rc < NET_XMIT_MASK))
                return true;

        return false;
}

/*
 *      Compute the worst case header length according to the protocols
 *      used.
 */

#if defined(CONFIG_WLAN) || IS_ENABLED(CONFIG_AX25)
# if defined(CONFIG_MAC80211_MESH)
#  define LL_MAX_HEADER 128
# else
#  define LL_MAX_HEADER 96
# endif
#else
# define LL_MAX_HEADER 32
#endif

#if !IS_ENABLED(CONFIG_NET_IPIP) && !IS_ENABLED(CONFIG_NET_IPGRE) && \
    !IS_ENABLED(CONFIG_IPV6_SIT) && !IS_ENABLED(CONFIG_IPV6_TUNNEL)
#define MAX_HEADER LL_MAX_HEADER
#else
#define MAX_HEADER (LL_MAX_HEADER + 48)
#endif

/*
 *      Old network device statistics. Fields are native words
 *      (unsigned long) so they can be read and written atomically.
 */

struct net_device_stats {
        unsigned long   rx_packets;
        unsigned long   tx_packets;
        unsigned long   rx_bytes;
        unsigned long   tx_bytes;
        unsigned long   rx_errors;
        unsigned long   tx_errors;
        unsigned long   rx_dropped;
        unsigned long   tx_dropped;
        unsigned long   multicast;
        unsigned long   collisions;
        unsigned long   rx_length_errors;
        unsigned long   rx_over_errors;
        unsigned long   rx_crc_errors;
        unsigned long   rx_frame_errors;
        unsigned long   rx_fifo_errors;
        unsigned long   rx_missed_errors;
        unsigned long   tx_aborted_errors;
        unsigned long   tx_carrier_errors;
        unsigned long   tx_fifo_errors;
        unsigned long   tx_heartbeat_errors;
        unsigned long   tx_window_errors;
        unsigned long   rx_compressed;
        unsigned long   tx_compressed;
};


#include <linux/cache.h>
#include <linux/skbuff.h>

#ifdef CONFIG_RPS
#include <linux/static_key.h>
extern struct static_key rps_needed;
#endif

struct neighbour;
struct neigh_parms;
struct sk_buff;

struct netdev_hw_addr {
        struct list_head        list;
        unsigned char           addr[MAX_ADDR_LEN];
        unsigned char           type;
```

```
#define NETDEV_HW_ADDR_T_LAN             1
#define NETDEV_HW_ADDR_T_SAN             2
#define NETDEV_HW_ADDR_T_SLAVE           3
#define NETDEV_HW_ADDR_T_UNICAST         4
#define NETDEV_HW_ADDR_T_MULTICAST       5
        bool                    global_use;
        int                     sync_cnt;
        int                     refcount;
        int                     synced;
        struct rcu_head         rcu_head;
};

struct netdev_hw_addr_list {
        struct list_head        list;
        int                     count;
};

#define netdev_hw_addr_list_count(l) ((l)->count)
#define netdev_hw_addr_list_empty(l) (netdev_hw_addr_list_count(l) == 0)
#define netdev_hw_addr_list_for_each(ha, l) \
        list_for_each_entry(ha, &(l)->list, list)

#define netdev_uc_count(dev) netdev_hw_addr_list_count(&(dev)->uc)
#define netdev_uc_empty(dev) netdev_hw_addr_list_empty(&(dev)->uc)
#define netdev_for_each_uc_addr(ha, dev) \
        netdev_hw_addr_list_for_each(ha, &(dev)->uc)

#define netdev_mc_count(dev) netdev_hw_addr_list_count(&(dev)->mc)
#define netdev_mc_empty(dev) netdev_hw_addr_list_empty(&(dev)->mc)
#define netdev_for_each_mc_addr(ha, dev) \
        netdev_hw_addr_list_for_each(ha, &(dev)->mc)

struct hh_cache {
        u16             hh_len;
        u16             __pad;
        seqlock_t       hh_lock;

        /* cached hardware header; allow for machine alignment needs.        */
#define HH_DATA_MOD     16
#define HH_DATA_OFF(__len) \
        (HH_DATA_MOD - (((__len - 1) & (HH_DATA_MOD - 1)) + 1))
#define HH_DATA_ALIGN(__len) \
        (((__len)+(HH_DATA_MOD-1))&~(HH_DATA_MOD - 1))
        unsigned long   hh_data[HH_DATA_ALIGN(LL_MAX_HEADER) / sizeof(long)];
};

/* Reserve HH_DATA_MOD byte aligned hard_header_len, but at least that much.
 * Alternative is:
 *   dev->hard_header_len ? (dev->hard_header_len +
 *                           (HH_DATA_MOD - 1)) & ~(HH_DATA_MOD - 1) : 0
 *
 * We could use other alignment values, but we must maintain the
 * relationship HH alignment <= LL alignment.
 */
#define LL_RESERVED_SPACE(dev) \
        ((((dev)->hard_header_len+(dev)->needed_headroom)&~(HH_DATA_MOD - 1)) + HH_DATA_MOD)
#define LL_RESERVED_SPACE_EXTRA(dev,extra) \
        ((((dev)->hard_header_len+(dev)->needed_headroom+(extra))&~(HH_DATA_MOD - 1)) + HH_DATA_MOD)

struct header_ops {
        int     (*create) (struct sk_buff *skb, struct net_device *dev,
                           unsigned short type, const void *daddr,
                           const void *saddr, unsigned int len);
        int     (*parse)(const struct sk_buff *skb, unsigned char *haddr);
        int     (*cache)(const struct neighbour *neigh, struct hh_cache *hh, __be16 type);
        void    (*cache_update)(struct hh_cache *hh,
                                const struct net_device *dev,
                                const unsigned char *haddr);
};

/* These flag bits are private to the generic network queueing
 * layer, they may not be explicitly referenced by any other
 * code.
 */

enum netdev_state_t {
        __LINK_STATE_START,
        __LINK_STATE_PRESENT,
        __LINK_STATE_NOCARRIER,
        __LINK_STATE_LINKWATCH_PENDING,
        __LINK_STATE_DORMANT,
};


/*
 * This structure holds at boot time configured netdevice settings. They
 * are then used in the device probing.
 */
struct netdev_boot_setup {
        char name[IFNAMSIZ];
        struct ifmap map;
};
#define NETDEV_BOOT_SETUP_MAX 8

int __init netdev_boot_setup(char *str);

/*
 * Structure for NAPI scheduling similar to tasklet but with weighting
 */
struct napi_struct {
        /* The poll_list must only be managed by the entity which
         * changes the state of the NAPI_STATE_SCHED bit.  This means
         * whoever atomically sets that bit can add this napi_struct
         * to the per-cpu poll_list, and whoever clears that bit
         * can remove from the list right before clearing the bit.
         */
        struct list_head        poll_list;

        unsigned long           state;
        int                     weight;
        unsigned int            gro_count;
        int                     (*poll)(struct napi_struct *, int);
#ifdef CONFIG_NETPOLL
        spinlock_t              poll_lock;
        int                     poll_owner;
#endif
        struct net_device       *dev;
        struct sk_buff          *gro_list;
        struct sk_buff          *skb;
        struct hrtimer          timer;
        struct list_head        dev_list;
        struct hlist_node       napi_hash_node;
        unsigned int            napi_id;
};

enum {
        NAPI_STATE_SCHED,       /* Poll is scheduled */
        NAPI_STATE_DISABLE,     /* Disable pending */
        NAPI_STATE_NPSVC,       /* Netpoll - don't dequeue from poll_list */
        NAPI_STATE_HASHED,      /* In NAPI hash */
};

enum gro_result {
        GRO_MERGED,
        GRO_MERGED_FREE,
        GRO_HELD,
        GRO_NORMAL,
        GRO_DROP,
};
typedef enum gro_result gro_result_t;

/*
 * enum rx_handler_result - Possible return values for rx_handlers.
 * @RX_HANDLER_CONSUMED: skb was consumed by rx_handler, do not process it
 * further.
 * @RX_HANDLER_ANOTHER: Do another round in receive path. This is indicated in
 * case skb->dev was changed by rx_handler.
 * @RX_HANDLER_EXACT: Force exact delivery, no wildcard.
 * @RX_HANDLER_PASS: Do nothing, passe the skb as if no rx_handler was called.
 *
 * rx_handlers are functions called from inside __netif_receive_skb(), to do
 * special processing of the skb, prior to delivery to protocol handlers.
 *
 * Currently, a net_device can only have a single rx_handler registered. Trying
 * to register a second rx_handler will return -EBUSY.
 *
 * To register a rx_handler on a net_device, use netdev_rx_handler_register().
 * To unregister a rx_handler on a net_device, use
 * netdev_rx_handler_unregister().
 *
 * Upon return, rx_handler is expected to tell __netif_receive_skb() what to
 * do with the skb.
 *
 * If the rx_handler consumed to skb in some way, it should return
 * RX_HANDLER_CONSUMED. This is appropriate when the rx_handler arranged for
 * the skb to be delivered in some other ways.
 *
 * If the rx_handler changed skb->dev, to divert the skb to another
 * net_device, it should return RX_HANDLER_ANOTHER. The rx_handler for the
 * new device will be called if it exists.
 *
 * If the rx_handler consider the skb should be ignored, it should return
 * RX_HANDLER_EXACT. The skb will only be delivered to protocol handlers that
 * are registered on exact device (ptype->dev == skb->dev).
 *
 * If the rx_handler didn't changed skb->dev, but want the skb to be normally
 * delivered, it should return RX_HANDLER_PASS.
 *
 * A device without a registered rx_handler will behave as if rx_handler
 * returned RX_HANDLER_PASS.
 */

enum rx_handler_result {
        RX_HANDLER_CONSUMED,
        RX_HANDLER_ANOTHER,
        RX_HANDLER_EXACT,
        RX_HANDLER_PASS,
};
typedef enum rx_handler_result rx_handler_result_t;
typedef rx_handler_result_t rx_handler_func_t(struct sk_buff **pskb);

void __napi_schedule(struct napi_struct *n);
void __napi_schedule_irqoff(struct napi_struct *n);

static inline bool napi_disable_pending(struct napi_struct *n)
{
        return test_bit(NAPI_STATE_DISABLE, &n->state);
}

/**
 *      napi_schedule_prep - check if napi can be scheduled
 *      @n: napi context
 *
 * Test if NAPI routine is already running, and if not mark
 * it as running.  This is used as a condition variable
 * insure only one NAPI poll instance runs.  We also make
 * sure there is no pending NAPI disable.
 */
static inline bool napi_schedule_prep(struct napi_struct *n)
{
        return !napi_disable_pending(n) &&
                !test_and_set_bit(NAPI_STATE_SCHED, &n->state);
}

/**
 *      napi_schedule - schedule NAPI poll
 *      @n: napi context
 *
 * Schedule NAPI poll routine to be called if it is not already
 * running.
 */
static inline void napi_schedule(struct napi_struct *n)
{
        if (napi_schedule_prep(n))
                __napi_schedule(n);
}

/**
 *      napi_schedule_irqoff - schedule NAPI poll
 *      @n: napi context
 *
 * Variant of napi_schedule(), assuming hard irqs are masked.
 */
static inline void napi_schedule_irqoff(struct napi_struct *n)
{
        if (napi_schedule_prep(n))
                __napi_schedule_irqoff(n);
}

/* Try to reschedule poll. Called by dev->poll() after napi_complete().  */
static inline bool napi_reschedule(struct napi_struct *napi)
{
        if (napi_schedule_prep(napi)) {
                __napi_schedule(napi);
                return true;
        }
        return false;
}

void __napi_complete(struct napi_struct *n);
void napi_complete_done(struct napi_struct *n, int work_done);
/**
 *      napi_complete - NAPI processing complete
 *      @n: napi context
 *
 * Mark NAPI processing as complete.
 * Consider using napi_complete_done() instead.
 */
static inline void napi_complete(struct napi_struct *n)
{
        return napi_complete_done(n, 0);
}

/**
 *      napi_by_id - lookup a NAPI by napi_id
 *      @napi_id: hashed napi_id
 *
 * lookup @napi_id in napi_hash table
 * must be called under rcu_read_lock()
 */
struct napi_struct *napi_by_id(unsigned int napi_id);

/**
 *      napi_hash_add - add a NAPI to global hashtable
 *      @napi: napi context
 *
 * generate a new napi_id and store a @napi under it in napi_hash
 */
void napi_hash_add(struct napi_struct *napi);

/**
 *      napi_hash_del - remove a NAPI from global table
 *      @napi: napi context
 *
 * Warning: caller must observe rcu grace period
 * before freeing memory containing @napi
 */
void napi_hash_del(struct napi_struct *napi);

/**
 *      napi_disable - prevent NAPI from scheduling
 *      @n: napi context
 *
 * Stop NAPI from being scheduled on this context.
 * Waits till any outstanding processing completes.
 */
void napi_disable(struct napi_struct *n);

/**
 *      napi_enable - enable NAPI scheduling
 *      @n: napi context
 *
 * Resume NAPI from being scheduled on this context.
 * Must be paired with napi_disable.
 */
static inline void napi_enable(struct napi_struct *n)
{
        BUG_ON(!test_bit(NAPI_STATE_SCHED, &n->state));
        smp_mb__before_atomic();
        clear_bit(NAPI_STATE_SCHED, &n->state);
}

#ifdef CONFIG_SMP
/**
 *      napi_synchronize - wait until NAPI is not running
 *      @n: napi context
 *
 * Wait until NAPI is done being scheduled on this context.
 * Waits till any outstanding processing completes but
 * does not disable future activations.
 */
static inline void napi_synchronize(const struct napi_struct *n)
{
        while (test_bit(NAPI_STATE_SCHED, &n->state))
                msleep(1);
}
#else
# define napi_synchronize(n)    barrier()
#endif

enum netdev_queue_state_t {
        __QUEUE_STATE_DRV_XOFF,
        __QUEUE_STATE_STACK_XOFF,
        __QUEUE_STATE_FROZEN,
};

#define QUEUE_STATE_DRV_XOFF    (1 << __QUEUE_STATE_DRV_XOFF)
#define QUEUE_STATE_STACK_XOFF  (1 << __QUEUE_STATE_STACK_XOFF)
#define QUEUE_STATE_FROZEN      (1 << __QUEUE_STATE_FROZEN)

#define QUEUE_STATE_ANY_XOFF    (QUEUE_STATE_DRV_XOFF | QUEUE_STATE_STACK_XOFF)
#define QUEUE_STATE_ANY_XOFF_OR_FROZEN (QUEUE_STATE_ANY_XOFF | \
                                        QUEUE_STATE_FROZEN)
#define QUEUE_STATE_DRV_XOFF_OR_FROZEN (QUEUE_STATE_DRV_XOFF | \
                                        QUEUE_STATE_FROZEN)

/*
 * __QUEUE_STATE_DRV_XOFF is used by drivers to stop the transmit queue.  The
 * netif_tx_* functions below are used to manipulate this flag.  The
 * __QUEUE_STATE_STACK_XOFF flag is used by the stack to stop the transmit
 * queue independently.  The netif_xmit_*stopped functions below are called
 * to check if the queue has been stopped by the driver or stack (either
 * of the XOFF bits are set in the state).  Drivers should not need to call
 * netif_xmit*stopped functions, they should only be using netif_tx_*.
 */

struct netdev_queue {
/*
 * read mostly part
 */
        struct net_device       *dev;
        struct Qdisc __rcu      *qdisc;
        struct Qdisc            *qdisc_sleeping;
#ifdef CONFIG_SYSFS
        struct kobject          kobj;
```

```
#endif
#if defined(CONFIG_XPS) && defined(CONFIG_NUMA)
        int                     numa_node;
#endif
/*
 * write mostly part
 */
        spinlock_t              _xmit_lock ____cacheline_aligned_in_smp;
        int                     xmit_lock_owner;
        /*
         * please use this field instead of dev->trans_start
         */
        unsigned long           trans_start;

        /*
         * Number of TX timeouts for this queue
         * (/sys/class/net/DEV/Q/trans_timeout)
         */
        unsigned long           trans_timeout;

        unsigned long           state;

#ifdef CONFIG_BQL
        struct dql              dql;
#endif
        unsigned long           tx_maxrate;
} ____cacheline_aligned_in_smp;

static inline int netdev_queue_numa_node_read(const struct netdev_queue *q)
{
#if defined(CONFIG_XPS) && defined(CONFIG_NUMA)
        return q->numa_node;
#else
        return NUMA_NO_NODE;
#endif
}

static inline void netdev_queue_numa_node_write(struct netdev_queue *q, int node)
{
#if defined(CONFIG_XPS) && defined(CONFIG_NUMA)
        q->numa_node = node;
#endif
}

#ifdef CONFIG_RPS
/*
 * This structure holds an RPS map which can be of variable length.  The
 * map is an array of CPUs.
 */
struct rps_map {
        unsigned int len;
        struct rcu_head rcu;
        u16 cpus[0];
};
#define RPS_MAP_SIZE(_num) (sizeof(struct rps_map) + ((_num) * sizeof(u16)))

/*
 * The rps_dev_flow structure contains the mapping of a flow to a CPU, the
 * tail pointer for that CPU's input queue at the time of last enqueue, and
 * a hardware filter index.
 */
struct rps_dev_flow {
        u16 cpu;
        u16 filter;
        unsigned int last_qtail;
};
#define RPS_NO_FILTER 0xffff

/*
 * The rps_dev_flow_table structure contains a table of flow mappings.
 */
struct rps_dev_flow_table {
        unsigned int mask;
        struct rcu_head rcu;
        struct rps_dev_flow flows[0];
};
#define RPS_DEV_FLOW_TABLE_SIZE(_num) (sizeof(struct rps_dev_flow_table) + \
    ((_num) * sizeof(struct rps_dev_flow)))

/*
 * The rps_sock_flow_table contains mappings of flows to the last CPU
 * on which they were processed by the application (set in recvmsg).
 * Each entry is a 32bit value. Upper part is the high order bits
 * of flow hash, lower part is cpu number.
 * rps_cpu_mask is used to partition the space, depending on number of
 * possible cpus : rps_cpu_mask = roundup_pow_of_two(nr_cpu_ids) - 1
 * For example, if 64 cpus are possible, rps_cpu_mask = 0x3f,
 * meaning we use 32-6=26 bits for the hash.
 */
struct rps_sock_flow_table {
        u32     mask;

        u32     ents[0] ____cacheline_aligned_in_smp;
};
#define RPS_SOCK_FLOW_TABLE_SIZE(_num) (offsetof(struct rps_sock_flow_table, ents[_num]))

#define RPS_NO_CPU 0xffff

extern u32 rps_cpu_mask;
extern struct rps_sock_flow_table __rcu *rps_sock_flow_table;

static inline void rps_record_sock_flow(struct rps_sock_flow_table *table,
                                        u32 hash)
{
        if (table && hash) {
                unsigned int index = hash & table->mask;
                u32 val = hash & ~rps_cpu_mask;

                /* We only give a hint, preemption can change cpu under us */
                val |= raw_smp_processor_id();

                if (table->ents[index] != val)
                        table->ents[index] = val;
        }
}

#ifdef CONFIG_RFS_ACCEL
bool rps_may_expire_flow(struct net_device *dev, u16 rxq_index, u32 flow_id,
                         u16 filter_id);
#endif
#endif /* CONFIG_RPS */

/* This structure contains an instance of an RX queue. */
struct netdev_rx_queue {
#ifdef CONFIG_RPS
        struct rps_map __rcu            *rps_map;
        struct rps_dev_flow_table __rcu *rps_flow_table;
#endif
        struct kobject                  kobj;
        struct net_device               *dev;
} ____cacheline_aligned_in_smp;

/*
 * RX queue sysfs structures and functions.
 */
struct rx_queue_attribute {
        struct attribute attr;
        ssize_t (*show)(struct netdev_rx_queue *queue,
            struct rx_queue_attribute *attr, char *buf);
        ssize_t (*store)(struct netdev_rx_queue *queue,
            struct rx_queue_attribute *attr, const char *buf, size_t len);
};

#ifdef CONFIG_XPS
/*
 * This structure holds an XPS map which can be of variable length.  The
 * map is an array of queues.
 */
struct xps_map {
        unsigned int len;
        unsigned int alloc_len;
        struct rcu_head rcu;
        u16 queues[0];
};
#define XPS_MAP_SIZE(_num) (sizeof(struct xps_map) + ((_num) * sizeof(u16)))
#define XPS_MIN_MAP_ALLOC ((L1_CACHE_BYTES - sizeof(struct xps_map))    \
    / sizeof(u16))

/*
 * This structure holds all XPS maps for device.  Maps are indexed by CPU.
 */
struct xps_dev_maps {
        struct rcu_head rcu;
        struct xps_map __rcu *cpu_map[0];
};
#define XPS_DEV_MAPS_SIZE (sizeof(struct xps_dev_maps) +                \
    (nr_cpu_ids * sizeof(struct xps_map *)))
#endif /* CONFIG_XPS */

#define TC_MAX_QUEUE    16
#define TC_BITMASK      15
/* HW offloaded queuing disciplines txq count and offset maps */
struct netdev_tc_txq {
        u16 count;
        u16 offset;
};

#if defined(CONFIG_FCOE) || defined(CONFIG_FCOE_MODULE)
/*
 * This structure is to hold information about the device
 * configured to run FCoE protocol stack.
 */
struct netdev_fcoe_hbainfo {
        char    manufacturer[64];
        char    serial_number[64];
        char    hardware_version[64];
        char    driver_version[64];
        char    optionrom_version[64];
        char    firmware_version[64];
        char    model[256];
        char    model_description[256];
};
#endif

#define MAX_PHYS_ITEM_ID_LEN 32

/* This structure holds a unique identifier to identify some
 * physical item (port for example) used by a netdevice.
 */
struct netdev_phys_item_id {
        unsigned char id[MAX_PHYS_ITEM_ID_LEN];
        unsigned char id_len;
};

typedef u16 (*select_queue_fallback_t)(struct net_device *dev,
                                       struct sk_buff *skb);

/*
 * This structure defines the management hooks for network devices.
 * The following hooks can be defined; unless noted otherwise, they are
 * optional and can be filled with a null pointer.
 *
 * int (*ndo_init)(struct net_device *dev);
 *     This function is called once when network device is registered.
 *     The network device can use this to any late stage initializaton
 *     or semantic validattion. It can fail with an error code which will
 *     be propogated back to register_netdev
 *
 * void (*ndo_uninit)(struct net_device *dev);
 *     This function is called when device is unregistered or when registration
 *     fails. It is not called if init fails.
 *
 * int (*ndo_open)(struct net_device *dev);
 *     This function is called when network device transistions to the up
 *     state.
 *
 * int (*ndo_stop)(struct net_device *dev);
 *     This function is called when network device transistions to the down
 *     state.
 *
 * netdev_tx_t (*ndo_start_xmit)(struct sk_buff *skb,
 *                               struct net_device *dev);
 *      Called when a packet needs to be transmitted.
 *      Returns NETDEV_TX_OK.  Can return NETDEV_TX_BUSY, but you should stop
 *      the queue before that can happen; it's for obsolete devices and weird
 *      corner cases, but the stack really does a non-trivial amount
 *      of useless work if you return NETDEV_TX_BUSY.
 *        (can also return NETDEV_TX_LOCKED iff NETIF_F_LLTX)
 *      Required can not be NULL.
 *
 * u16 (*ndo_select_queue)(struct net_device *dev, struct sk_buff *skb,
 *                         void *accel_priv, select_queue_fallback_t fallback);
 *      Called to decide which queue to when device supports multiple
 *      transmit queues.
 *
 * void (*ndo_change_rx_flags)(struct net_device *dev, int flags);
 *      This function is called to allow device receiver to make
 *      changes to configuration when multicast or promiscious is enabled.
 *
 * void (*ndo_set_rx_mode)(struct net_device *dev);
 *      This function is called device changes address list filtering.
 *      If driver handles unicast address filtering, it should set
 *      IFF_UNICAST_FLT to its priv_flags.
 *
 * int (*ndo_set_mac_address)(struct net_device *dev, void *addr);
 *      This function  is called when the Media Access Control address
 *      needs to be changed. If this interface is not defined, the
 *      mac address can not be changed.
 *
 * int (*ndo_validate_addr)(struct net_device *dev);
 *      Test if Media Access Control address is valid for the device.
 *
 * int (*ndo_do_ioctl)(struct net_device *dev, struct ifreq *ifr, int cmd);
 *      Called when a user request an ioctl which can't be handled by
 *      the generic interface code. If not defined ioctl's return
 *      not supported error code.
 *
 * int (*ndo_set_config)(struct net_device *dev, struct ifmap *map);
 *      Used to set network devices bus interface parameters. This interface
 *      is retained for legacy reason, new devices should use the bus
 *      interface (PCI) for low level management.
 *
 * int (*ndo_change_mtu)(struct net_device *dev, int new_mtu);
 *      Called when a user wants to change the Maximum Transfer Unit
 *      of a device. If not defined, any request to change MTU will
 *      will return an error.
 *
 * void (*ndo_tx_timeout)(struct net_device *dev);
 *      Callback uses when the transmitter has not made any progress
 *      for dev->watchdog ticks.
 *
 * struct rtnl_link_stats64* (*ndo_get_stats64)(struct net_device *dev,
 *                      struct rtnl_link_stats64 *storage);
 * struct net_device_stats* (*ndo_get_stats)(struct net_device *dev);
 *      Called when a user wants to get the network device usage
 *      statistics. Drivers must do one of the following:
 *      1. Define @ndo_get_stats64 to fill in a zero-initialised
 *         rtnl_link_stats64 structure passed by the caller.
 *      2. Define @ndo_get_stats to update a net_device_stats structure
 *         (which should normally be dev->stats) and return a pointer to
 *         it. The structure may be changed asynchronously only if each
 *         field is written atomically.
 *      3. Update dev->stats asynchronously and atomically, and define
 *         neither operation.
 *
 * int (*ndo_vlan_rx_add_vid)(struct net_device *dev, __be16 proto, u16 vid);
 *      If device support VLAN filtering this function is called when a
 *      VLAN id is registered.
 *
 * int (*ndo_vlan_rx_kill_vid)(struct net_device *dev, __be16 proto, u16 vid);
 *      If device support VLAN filtering this function is called when a
 *      VLAN id is unregistered.
 *
 * void (*ndo_poll_controller)(struct net_device *dev);
 *
 *      SR-IOV management functions.
 * int (*ndo_set_vf_mac)(struct net_device *dev, int vf, u8* mac);
 * int (*ndo_set_vf_vlan)(struct net_device *dev, int vf, u16 vlan, u8 qos);
 * int (*ndo_set_vf_rate)(struct net_device *dev, int vf, int min_tx_rate,
 *                        int max_tx_rate);
 * int (*ndo_set_vf_spoofchk)(struct net_device *dev, int vf, bool setting);
 * int (*ndo_get_vf_config)(struct net_device *dev,
 *                          int vf, struct ifla_vf_info *ivf);
 * int (*ndo_set_vf_link_state)(struct net_device *dev, int vf, int link_state);
 * int (*ndo_set_vf_port)(struct net_device *dev, int vf,
 *                        struct nlattr *port[]);
 *
 *      Enable or disable the VF ability to query its RSS Redirection Table and
 *      Hash Key. This is needed since on some devices VF share this information
 *      with PF and querying it may adduce a theoretical security risk.
 * int (*ndo_set_vf_rss_query_en)(struct net_device *dev, int vf, bool setting);
 * int (*ndo_get_vf_port)(struct net_device *dev, int vf, struct sk_buff *skb);
 * int (*ndo_setup_tc)(struct net_device *dev, u8 tc)
 *      Called to setup 'tc' number of traffic classes in the net device. This
 *      is always called from the stack with the rtnl lock held and netif tx
 *      queues stopped. This allows the netdevice to perform queue management
 *      safely.
 *
 *      Fiber Channel over Ethernet (FCoE) offload functions.
 * int (*ndo_fcoe_enable)(struct net_device *dev);
 *      Called when the FCoE protocol stack wants to start using LLD for FCoE
 *      so the underlying device can perform whatever needed configuration or
 *      initialization to support acceleration of FCoE traffic.
 *
 * int (*ndo_fcoe_disable)(struct net_device *dev);
 *      Called when the FCoE protocol stack wants to stop using LLD for FCoE
 *      so the underlying device can perform whatever needed clean-ups to
 *      stop supporting acceleration of FCoE traffic.
 *
 * int (*ndo_fcoe_ddp_setup)(struct net_device *dev, u16 xid,
 *                           struct scatterlist *sgl, unsigned int sgc);
 *      Called when the FCoE Initiator wants to initialize an I/O that
 *      is a possible candidate for Direct Data Placement (DDP). The LLD can
 *      perform necessary setup and returns 1 to indicate the device is set up
 *      successfully to perform DDP on this I/O, otherwise this returns 0.
 *
 * int (*ndo_fcoe_ddp_done)(struct net_device *dev,  u16 xid);
 *      Called when the FCoE Initiator/Target is done with the DDPed I/O as
 *      indicated by the FC exchange id 'xid', so the underlying device can
 *      clean up and reuse resources for later DDP requests.
 *
 * int (*ndo_fcoe_ddp_target)(struct net_device *dev, u16 xid,
 *                            struct scatterlist *sgl, unsigned int sgc);
 *      Called when the FCoE Target wants to initialize an I/O that
 *      is a possible candidate for Direct Data Placement (DDP). The LLD can
 *      perform necessary setup and returns 1 to indicate the device is set up
 *      successfully to perform DDP on this I/O, otherwise this returns 0.
 *
 * int (*ndo_fcoe_get_hbainfo)(struct net_device *dev,
 *                             struct netdev_fcoe_hbainfo *hbainfo);
 *      Called when the FCoE Protocol stack wants information on the underlying
 *      device. This information is utilized by the FCoE protocol stack to
 *      register attributes with Fiber Channel management service as per the
 *      FC-GS Fabric Device Management Information(FDMI) specification.
 *
 * int (*ndo_fcoe_get_wwn)(struct net_device *dev, u64 *wwn, int type);
 *      Called when the underlying device wants to override default World Wide
 *      Name (WWN) generation mechanism in FCoE protocol stack to pass its own
 *      World Wide Port Name (WWPN) or World Wide Node Name (WWNN) to the FCoE
 *      protocol stack to use.
 *
 *      RFS acceleration.
 * int (*ndo_rx_flow_steer)(struct net_device *dev, const struct sk_buff *skb,
 *                          u16 rxq_index, u32 flow_id);
 *      Set hardware filter for RFS.  rxq_index is the target queue index;
 *      flow_id is a flow ID to be passed to rps_may_expire_flow() later.
 *      Return the filter ID on success, or a negative error code.
 *
 *      Slave management functions (for bridge, bonding, etc).
 * int (*ndo_add_slave)(struct net_device *dev, struct net_device *slave_dev);
 *      Called to make another netdev an underling.
 *
 * int (*ndo_del_slave)(struct net_device *dev, struct net_device *slave_dev);
 *      Called to release previously enslaved netdev.
 *
 *      Feature/offload setting functions.
 * netdev_features_t (*ndo_fix_features)(struct net_device *dev,
 *              netdev_features_t features);
 *      Adjusts the requested feature flags according to device-specific
 *      constraints, and returns the resulting flags. Must not modify
 *      the device state.
 *
 * int (*ndo_set_features)(struct net_device *dev, netdev_features_t features);
 *      Called to update device configuration to new features. Passed
 *      feature set might be less than what was returned by ndo_fix_features()).
 *      Must return >0 or -errno if it changed dev->features itself.
 *
 * int (*ndo_fdb_add)(struct ndmsg *ndm, struct nlattr *tb[],
 *                    struct net_device *dev,
 *                    const unsigned char *addr, u16 vid, u16 flags)
 *      Adds an FDB entry to dev for addr.
 * int (*ndo_fdb_del)(struct ndmsg *ndm, struct nlattr *tb[],
 *                    struct net_device *dev,
 *                    const unsigned char *addr, u16 vid)
 *      Deletes the FDB entry from dev coresponding to addr.
 * int (*ndo_fdb_dump)(struct sk_buff *skb, struct netlink_callback *cb,
 *                     struct net_device *dev, struct net_device *filter_dev,
 *                     int idx)
 *      Used to add FDB entries to dump requests. Implementers should add
 *      entries to skb and update idx with the number of entries.
 *
 * int (*ndo_bridge_setlink)(struct net_device *dev, struct nlmsghdr *nlh,
 *                           u16 flags)
 * int (*ndo_bridge_getlink)(struct sk_buff *skb, u32 pid, u32 seq,
 *                           struct net_device *dev, u32 filter_mask,
 *                           int nlflags)
 * int (*ndo_bridge_dellink)(struct net_device *dev, struct nlmsghdr *nlh,
 *                           u16 flags);
 *
 * int (*ndo_change_carrier)(struct net_device *dev, bool new_carrier);
 *      Called to change device carrier. Soft-devices (like dummy, team, etc)
 *      which do not represent real hardware may define this to allow their
 *      userspace components to manage their virtual carrier state. Devices
 *      that determine carrier state from physical hardware properties (eg
 *      network cables) or protocol-dependent mechanisms (eg
 *      USB_CDC_NOTIFY_NETWORK_CONNECTION) should NOT implement this function.
 *
 * int (*ndo_get_phys_port_id)(struct net_device *dev,
 *                             struct netdev_phys_item_id *ppid);
 *      Called to get ID of physical port of this device. If driver does
 *      not implement this, it is assumed that the hw is not able to have
 *      multiple net devices on single physical port.
 *
 * void (*ndo_add_vxlan_port)(struct  net_device *dev,
 *                            sa_family_t sa_family, __be16 port);
 *      Called by vxlan to notiy a driver about the UDP port and socket
 *      address family that vxlan is listnening to. It is called only when
 *      a new port starts listening. The operation is protected by the
 *      vxlan_net->sock_lock.
 *
 * void (*ndo_del_vxlan_port)(struct  net_device *dev,
 *                            sa_family_t sa_family, __be16 port);
 *      Called by vxlan to notify the driver about a UDP port and socket
 *      address family that vxlan is not listening to anymore. The operation
 *      is protected by the vxlan_net->sock_lock.
 *
 * void* (*ndo_dfwd_add_station)(struct net_device *pdev,
 *                               struct net_device *dev)
 *      Called by upper layer devices to accelerate switching or other
 *      station functionality into hardware. 'pdev is the lowerdev
 *      to use for the offload and 'dev' is the net device that will
 *      back the offload. Returns a pointer to the private structure
 *      the upper layer will maintain.
 * void (*ndo_dfwd_del_station)(struct net_device *pdev, void *priv)
 *      Called by upper layer device to delete the station created
 *      by 'ndo_dfwd_add_station'. 'pdev' is the net device backing
 *      the station and priv is the structure returned by the add
 *      operation.
 * netdev_tx_t (*ndo_dfwd_start_xmit)(struct sk_buff *skb,
 *                                    struct net_device *dev,
 *                                    void *priv);
 *      Callback to use for xmit over the accelerated station. This
 *      is used in place of ndo_start_xmit on accelerated net
 *      devices.
 * netdev_features_t (*ndo_features_check) (struct sk_buff *skb,
 *                                          struct net_device *dev
 *                                          netdev_features_t features);
 *      Called by core transmit path to determine if device is capable of
 *      performing offload operations on a given packet. This is to give
 *      the device an opportunity to implement any restrictions that cannot
 *      be otherwise expressed by feature flags. The check is called with
 *      the set of features that the stack has calculated and it returns
 *      those the driver believes to be appropriate.
 * int (*ndo_set_tx_maxrate)(struct net_device *dev,
 *                           int queue_index, u32 maxrate);
 *      Called when a user wants to set a max-rate limitation of specific
 *      TX queue.
 * int (*ndo_get_iflink)(const struct net_device *dev);
 *      Called to get the iflink value of this device.
 */
struct net_device_ops {
        int                     (*ndo_init)(struct net_device *dev);
        void                    (*ndo_uninit)(struct net_device *dev);
        int                     (*ndo_open)(struct net_device *dev);
        int                     (*ndo_stop)(struct net_device *dev);
        netdev_tx_t             (*ndo_start_xmit) (struct sk_buff *skb,
                                                   struct net_device *dev);
        u16                     (*ndo_select_queue)(struct net_device *dev,
                                                    struct sk_buff *skb,
                                                    void *accel_priv,
                                                    select_queue_fallback_t fallback);
        void                    (*ndo_change_rx_flags)(struct net_device *dev,
                                                       int flags);
        void                    (*ndo_set_rx_mode)(struct net_device *dev);
        int                     (*ndo_set_mac_address)(struct net_device *dev,
                                                       void *addr);
        int                     (*ndo_validate_addr)(struct net_device *dev);
        int                     (*ndo_do_ioctl)(struct net_device *dev,
                                                struct ifreq *ifr, int cmd);
        int                     (*ndo_set_config)(struct net_device *dev,
                                                  struct ifmap *map);
        int                     (*ndo_change_mtu)(struct net_device *dev,
                                                  int new_mtu);
        int                     (*ndo_neigh_setup)(struct net_device *dev,
                                                   struct neigh_parms *);
        void                    (*ndo_tx_timeout) (struct net_device *dev);

        struct rtnl_link_stats64* (*ndo_get_stats64)(struct net_device *dev,
                                                     struct rtnl_link_stats64 *storage);
        struct net_device_stats* (*ndo_get_stats)(struct net_device *dev);

        int                     (*ndo_vlan_rx_add_vid)(struct net_device *dev,
                                                       __be16 proto, u16 vid);
        int                     (*ndo_vlan_rx_kill_vid)(struct net_device *dev,
                                                        __be16 proto, u16 vid);
#ifdef CONFIG_NET_POLL_CONTROLLER
        void                    (*ndo_poll_controller)(struct net_device *dev);
        int                     (*ndo_netpoll_setup)(struct net_device *dev,
                                                     struct netpoll_info *info);
        void                    (*ndo_netpoll_cleanup)(struct net_device *dev);
#endif
#ifdef CONFIG_NET_RX_BUSY_POLL
        int                     (*ndo_busy_poll)(struct napi_struct *dev);
#endif
        int                     (*ndo_set_vf_mac)(struct net_device *dev,
                                                  int queue, u8 *mac);
        int                     (*ndo_set_vf_vlan)(struct net_device *dev,
                                                   int queue, u16 vlan, u8 qos);
        int                     (*ndo_set_vf_rate)(struct net_device *dev,
                                                   int vf, int min_tx_rate,
                                                   int max_tx_rate);
        int                     (*ndo_set_vf_spoofchk)(struct net_device *dev,
                                                       int vf, bool setting);
        int                     (*ndo_get_vf_config)(struct net_device *dev,
                                                     int vf,
                                                     struct ifla_vf_info *ivf);
        int                     (*ndo_set_vf_link_state)(struct net_device *dev,
                                                         int vf, int link_state);
        int                     (*ndo_get_vf_stats)(struct net_device *dev,
                                                    int vf,
                                                    struct ifla_vf_stats
                                                    *vf_stats);
        int                     (*ndo_set_vf_port)(struct net_device *dev,
                                                   int vf,
                                                   struct nlattr *port[]);
        int                     (*ndo_get_vf_port)(struct net_device *dev,
                                                   int vf, struct sk_buff *skb);
        int                     (*ndo_set_vf_rss_query_en)(
                                                   struct net_device *dev,
                                                   int vf, bool setting);
        int                     (*ndo_setup_tc)(struct net_device *dev, u8 tc);
#if IS_ENABLED(CONFIG_FCOE)
        int                     (*ndo_fcoe_enable)(struct net_device *dev);
        int                     (*ndo_fcoe_disable)(struct net_device *dev);
        int                     (*ndo_fcoe_ddp_setup)(struct net_device *dev,
                                                      u16 xid,
                                                      struct scatterlist *sgl,
                                                      unsigned int sgc);
        int                     (*ndo_fcoe_ddp_done)(struct net_device *dev,
                                                     u16 xid);
        int                     (*ndo_fcoe_ddp_target)(struct net_device *dev,
                                                       u16 xid,
                                                       struct scatterlist *sgl,
                                                       unsigned int sgc);
        int                     (*ndo_fcoe_get_hbainfo)(struct net_device *dev,
                                                        struct netdev_fcoe_hbainfo *hbainfo);
#endif

#if IS_ENABLED(CONFIG_LIBFCOE)
#define NETDEV_FCOE_WWNN 0
#define NETDEV_FCOE_WWPN 1
        int                     (*ndo_fcoe_get_wwn)(struct net_device *dev,
                                                    u64 *wwn, int type);
#endif

#ifdef CONFIG_RFS_ACCEL
        int                     (*ndo_rx_flow_steer)(struct net_device *dev,
                                                     const struct sk_buff *skb,
                                                     u16 rxq_index,
                                                     u32 flow_id);
#endif
        int                     (*ndo_add_slave)(struct net_device *dev,
                                                 struct net_device *slave_dev);
        int                     (*ndo_del_slave)(struct net_device *dev,
                                                 struct net_device *slave_dev);
        netdev_features_t       (*ndo_fix_features)(struct net_device *dev,
                                                    netdev_features_t features);
        int                     (*ndo_set_features)(struct net_device *dev,
                                                    netdev_features_t features);
        int                     (*ndo_neigh_construct)(struct neighbour *n);
        void                    (*ndo_neigh_destroy)(struct neighbour *n);

        int                     (*ndo_fdb_add)(struct ndmsg *ndm,
                                               struct nlattr *tb[],
                                               struct net_device *dev,
                                               const unsigned char *addr,
                                               u16 vid,
                                               u16 flags);
        int                     (*ndo_fdb_del)(struct ndmsg *ndm,
                                               struct nlattr *tb[],
                                               struct net_device *dev,
                                               const unsigned char *addr,
                                               u16 vid);
        int                     (*ndo_fdb_dump)(struct sk_buff *skb,
                                                struct netlink_callback *cb,
                                                struct net_device *dev,
                                                struct net_device *filter_dev,
                                                int idx);

        int                     (*ndo_bridge_setlink)(struct net_device *dev,
                                                      struct nlmsghdr *nlh,
                                                      u16 flags);
        int                     (*ndo_bridge_getlink)(struct sk_buff *skb,
                                                      u32 pid, u32 seq,
                                                      struct net_device *dev,
                                                      u32 filter_mask,
                                                      int nlflags);
        int                     (*ndo_bridge_dellink)(struct net_device *dev,
                                                      struct nlmsghdr *nlh,
                                                      u16 flags);
        int                     (*ndo_change_carrier)(struct net_device *dev,
                                                      bool new_carrier);
        int                     (*ndo_get_phys_port_id)(struct net_device *dev,
                                                        struct netdev_phys_item_id *ppid);
        int                     (*ndo_get_phys_port_name)(struct net_device *dev,
                                                          char *name, size_t len);
        void                    (*ndo_add_vxlan_port)(struct  net_device *dev,
                                                      sa_family_t sa_family,
                                                      __be16 port);
        void                    (*ndo_del_vxlan_port)(struct  net_device *dev,
                                                      sa_family_t sa_family,
                                                      __be16 port);

        void*                   (*ndo_dfwd_add_station)(struct net_device *pdev,
                                                        struct net_device *dev);
        void                    (*ndo_dfwd_del_station)(struct net_device *pdev,
                                                        void *priv);

        netdev_tx_t             (*ndo_dfwd_start_xmit) (struct sk_buff *skb,
                                                        struct net_device *dev,
                                                        void *priv);
        int                     (*ndo_get_lock_subclass)(struct net_device *dev);
        netdev_features_t       (*ndo_features_check) (struct sk_buff *skb,
                                                       struct net_device *dev,
                                                       netdev_features_t features);
        int                     (*ndo_set_tx_maxrate)(struct net_device *dev,
                                                      int queue_index,
                                                      u32 maxrate);
        int                     (*ndo_get_iflink)(const struct net_device *dev);
};

/**
 * enum net_device_priv_flags - &struct net_device priv_flags
 *
 * These are the &struct net_device, they are only set internally
 * by drivers and used in the kernel. These flags are invisible to
 * userspace, this means that the order of these flags can change
 * during any kernel release.
 *
 * You should have a pretty good reason to be extending these flags.
 *
 * @IFF_802_1Q_VLAN: 802.1Q VLAN device
 * @IFF_EBRIDGE: Ethernet bridging device
 * @IFF_SLAVE_INACTIVE: bonding slave not the curr. active
 * @IFF_MASTER_8023AD: bonding master, 802.3ad
 * @IFF_MASTER_ALB: bonding master, balance-alb
 * @IFF_BONDING: bonding master or slave
 * @IFF_SLAVE_NEEDARP: need ARPs for validation
 * @IFF_ISATAP: ISATAP interface (RFC4214)
 * @IFF_MASTER_ARPMON: bonding master, ARP mon in use
 * @IFF_WAN_HDLC: WAN HDLC device
 * @IFF_XMIT_DST_RELEASE: dev_hard_start_xmit() is allowed to
 *      release skb->dst
 * @IFF_DONT_BRIDGE: disallow bridging this ether dev
 * @IFF_DISABLE_NETPOLL: disable netpoll at run-time
 * @IFF_MACVLAN_PORT: device used as macvlan port
 * @IFF_BRIDGE_PORT: device used as bridge port
 * @IFF_OVS_DATAPATH: device used as Open vSwitch datapath port
 * @IFF_TX_SKB_SHARING: The interface supports sharing skbs on transmit
 * @IFF_UNICAST_FLT: Supports unicast filtering
 * @IFF_TEAM_PORT: device used as team port
 * @IFF_SUPP_NOFCS: device supports sending custom FCS
 * @IFF_LIVE_ADDR_CHANGE: device supports hardware address
 *      change when it's running
 * @IFF_MACVLAN: Macvlan device
 */
enum netdev_priv_flags {
        IFF_802_1Q_VLAN                 = 1<<0,
        IFF_EBRIDGE                     = 1<<1,
        IFF_SLAVE_INACTIVE              = 1<<2,
        IFF_MASTER_8023AD               = 1<<3,
        IFF_MASTER_ALB                  = 1<<4,
        IFF_BONDING                     = 1<<5,
        IFF_SLAVE_NEEDARP               = 1<<6,
        IFF_ISATAP                      = 1<<7,
        IFF_MASTER_ARPMON               = 1<<8,
        IFF_WAN_HDLC                    = 1<<9,
        IFF_XMIT_DST_RELEASE            = 1<<10,
        IFF_DONT_BRIDGE                 = 1<<11,
        IFF_DISABLE_NETPOLL             = 1<<12,
        IFF_MACVLAN_PORT                = 1<<13,
        IFF_BRIDGE_PORT                 = 1<<14,
        IFF_OVS_DATAPATH                = 1<<15,
        IFF_TX_SKB_SHARING              = 1<<16,
        IFF_UNICAST_FLT                 = 1<<17,
        IFF_TEAM_PORT                   = 1<<18,
        IFF_SUPP_NOFCS                  = 1<<19,
        IFF_LIVE_ADDR_CHANGE            = 1<<20,
        IFF_MACVLAN                     = 1<<21,
        IFF_XMIT_DST_RELEASE_PERM       = 1<<22,
        IFF_IPVLAN_MASTER               = 1<<23,
        IFF_IPVLAN_SLAVE                = 1<<24,
};

#define IFF_802_1Q_VLAN                 IFF_802_1Q_VLAN
#define IFF_EBRIDGE                     IFF_EBRIDGE
#define IFF_SLAVE_INACTIVE              IFF_SLAVE_INACTIVE
#define IFF_MASTER_8023AD               IFF_MASTER_8023AD
#define IFF_MASTER_ALB                  IFF_MASTER_ALB
#define IFF_BONDING                     IFF_BONDING
#define IFF_SLAVE_NEEDARP               IFF_SLAVE_NEEDARP
#define IFF_ISATAP                      IFF_ISATAP
#define IFF_MASTER_ARPMON               IFF_MASTER_ARPMON
#define IFF_WAN_HDLC                    IFF_WAN_HDLC
#define IFF_XMIT_DST_RELEASE            IFF_XMIT_DST_RELEASE
#define IFF_DONT_BRIDGE                 IFF_DONT_BRIDGE
#define IFF_DISABLE_NETPOLL             IFF_DISABLE_NETPOLL
#define IFF_MACVLAN_PORT                IFF_MACVLAN_PORT
#define IFF_BRIDGE_PORT                 IFF_BRIDGE_PORT
#define IFF_OVS_DATAPATH                IFF_OVS_DATAPATH
```

```
#define IFF_TX_SKB_SHARING              IFF_TX_SKB_SHARING
#define IFF_UNICAST_FLT                 IFF_UNICAST_FLT
#define IFF_TEAM_PORT                   IFF_TEAM_PORT
#define IFF_SUPP_NOFCS                  IFF_SUPP_NOFCS
#define IFF_LIVE_ADDR_CHANGE            IFF_LIVE_ADDR_CHANGE
#define IFF_MACVLAN                     IFF_MACVLAN
#define IFF_XMIT_DST_RELEASE_PERM       IFF_XMIT_DST_RELEASE_PERM
#define IFF_IPVLAN_MASTER               IFF_IPVLAN_MASTER
#define IFF_IPVLAN_SLAVE                IFF_IPVLAN_SLAVE

/**
 *      struct net_device - The DEVICE structure.
 *              Actually, this whole structure is a big mistake.  It mixes I/O
 *              data with strictly "high-level" data, and it has to know about
 *              almost every data structure used in the INET module.
 *
 *      @name:  This is the first field of the "visible" part of this structure
 *              (i.e. as seen by users in the "Space.c" file).  It is the name
 *              of the interface.
 *
 *      @name_hlist:    Device name hash chain, please keep it close to name[]
 *      @ifalias:       SNMP alias
 *      @mem_end:       Shared memory end
 *      @mem_start:     Shared memory start
 *      @base_addr:     Device I/O address
 *      @irq:           Device IRQ number
 *
 *      @carrier_changes:       Stats to monitor carrier on<->off transitions
 *
 *      @state:         Generic network queuing layer state, see netdev_state_t
 *      @dev_list:      The global list of network devices
 *      @napi_list:     List entry, that is used for polling napi devices
 *      @unreg_list:    List entry, that is used, when we are unregistering the
 *                      device, see the function unregister_netdev
 *      @close_list:    List entry, that is used, when we are closing the device
 *
 *      @adj_list:      Directly linked devices, like slaves for bonding
 *      @all_adj_list:  All linked devices, *including* neighbours
 *      @features:      Currently active device features
 *      @hw_features:   User-changeable features
 *
 *      @wanted_features:       User-requested features
 *      @vlan_features:         Mask of features inheritable by VLAN devices
 *
 *      @hw_enc_features:       Mask of features inherited by encapsulating devices
 *                              This field indicates what encapsulation
 *                              offloads the hardware is capable of doing,
 *                              and drivers will need to set them appropriately.
 *
 *      @mpls_features: Mask of features inheritable by MPLS
 *
 *      @ifindex:       interface index
 *      @group:         The group, that the device belongs to
 *
 *      @stats:         Statistics struct, which was left as a legacy, use
 *                      rtnl_link_stats64 instead
 *
 *      @rx_dropped:    Dropped packets by core network,
 *                      do not use this in drivers
 *      @tx_dropped:    Dropped packets by core network,
 *                      do not use this in drivers
 *
 *      @wireless_handlers:     List of functions to handle Wireless Extensions,
 *                              instead of ioctl,
 *                              see <net/iw_handler.h> for details.
 *      @wireless_data: Instance data managed by the core of wireless extensions
 *
 *      @netdev_ops:    Includes several pointers to callbacks,
 *                      if one wants to override the ndo_*() functions
 *      @ethtool_ops:   Management operations
 *      @header_ops:    Includes callbacks for creating,parsing,caching,etc
 *                      of Layer 2 headers.
 *
 *      @flags:         Interface flags (a la BSD)
 *      @priv_flags:    Like 'flags' but invisible to userspace,
 *                      see if.h for the definitions
 *      @gflags:        Global flags ( kept as legacy )
 *      @padded:        How much padding added by alloc_netdev()
 *      @operstate:     RFC2863 operstate
 *      @link_mode:     Mapping policy to operstate
 *      @if_port:       Selectable AUI, TP, ...
 *      @dma:           DMA channel
 *      @mtu:           Interface MTU value
 *      @type:          Interface hardware type
 *      @hard_header_len: Hardware header length
 *
 *      @needed_headroom: Extra headroom the hardware may need, but not in all
 *                        cases can this be guaranteed
 *      @needed_tailroom: Extra tailroom the hardware may need, but not in all
 *                        cases can this be guaranteed. Some cases also use
 *                        LL_MAX_HEADER instead to allocate the skb
 *
 *      interface address info:
 *
 *      @perm_addr:             Permanent hw address
 *      @addr_assign_type:      Hw address assignment type
 *      @addr_len:              Hardware address length
 *      @neigh_priv_len;        Used in neigh_alloc(),
 *                              initialized only in atm/clip.c
 *      @dev_id:                Used to differentiate devices that share
 *                              the same link layer address
 *      @dev_port:              Used to differentiate devices that share
 *                              the same function
 *      @addr_list_lock:        XXX: need comments on this one
 *      @uc_promisc:            Counter, that indicates, that promiscuous mode
 *                              has been enabled due to the need to listen to
 *                              additional unicast addresses in a device that
 *                              does not implement ndo_set_rx_mode()
 *      @uc:                    unicast mac addresses
 *      @mc:                    multicast mac addresses
 *      @dev_addrs:             list of device hw addresses
 *      @queues_kset:           Group of all Kobjects in the Tx and RX queues
 *      @promiscuity:           Number of times, the NIC is told to work in
 *                              Promiscuous mode, if it becomes 0 the NIC will
 *                              exit from working in Promiscuous mode
 *      @allmulti:              Counter, enables or disables allmulticast mode
 *
 *      @vlan_info:     VLAN info
 *      @dsa_ptr:       dsa specific data
 *      @tipc_ptr:      TIPC specific data
 *      @atalk_ptr:     AppleTalk link
 *      @ip_ptr:        IPv4 specific data
 *      @dn_ptr:        DECnet specific data
 *      @ip6_ptr:       IPv6 specific data
 *      @ax25_ptr:      AX.25 specific data
 *      @ieee80211_ptr: IEEE 802.11 specific data, assign before registering
 *
 *      @last_rx:       Time of last Rx
 *      @dev_addr:      Hw address (before bcast,
 *                      because most packets are unicast)
 *
 *      @_rx:                   Array of RX queues
 *      @num_rx_queues:         Number of RX queues
 *                              allocated at register_netdev() time
 *      @real_num_rx_queues:    Number of RX queues currently active in device
 *
 *      @rx_handler:            handler for received packets
 *      @rx_handler_data:       XXX: need comments on this one
 *      @ingress_queue:         XXX: need comments on this one
 *      @broadcast:             hw bcast address
 *
 *      @rx_cpu_rmap:   CPU reverse-mapping for RX completion interrupts,
 *                      indexed by RX queue number. Assigned by driver.
 *                      This must only be set if the ndo_rx_flow_steer
 *                      operation is defined
 *      @index_hlist:           Device index hash chain
 *
 *      @_tx:                   Array of TX queues
 *      @num_tx_queues:         Number of TX queues allocated at alloc_netdev_mq() time
 *      @real_num_tx_queues:    Number of TX queues currently active in device
 *      @qdisc:                 Root qdisc from userspace point of view
 *      @tx_queue_len:          Max frames per queue allowed
 *      @tx_global_lock:        XXX: need comments on this one
 *
 *      @xps_maps:      XXX: need comments on this one
 *
 *      @trans_start:           Time (in jiffies) of last Tx
 *      @watchdog_timeo:        Represents the timeout that is used by
 *                              the watchdog ( see dev_watchdog() )
 *      @watchdog_timer:        List of timers
 *
 *      @pcpu_refcnt:           Number of references to this device
 *      @todo_list:             Delayed register/unregister
 *      @link_watch_list:       XXX: need comments on this one
 *
 *      @reg_state:             Register/unregister state machine
 *      @dismantle:             Device is going to be freed
 *      @rtnl_link_state:       This enum represents the phases of creating
 *                              a new link
 *
 *      @destructor:            Called from unregister,
 *                              can be used to call free_netdev
 *      @npinfo:                XXX: need comments on this one
 *      @nd_net:                Network namespace this network device is inside
 *
 *      @ml_priv:       Mid-layer private
 *      @lstats:        Loopback statistics
 *      @tstats:        Tunnel statistics
 *      @dstats:        Dummy statistics
 *      @vstats:        Virtual ethernet statistics
 *
 *      @garp_port:     GARP
 *      @mrp_port:      MRP
 *
 *      @dev:           Class/net/name entry
 *      @sysfs_groups:  Space for optional device, statistics and wireless
 *                      sysfs groups
 *
 *      @sysfs_rx_queue_group:  Space for optional per-rx queue attributes
 *      @rtnl_link_ops: Rtnl_link_ops
 *
 *      @gso_max_size:  Maximum size of generic segmentation offload
 *      @gso_max_segs:  Maximum number of segments that can be passed to the
 *                      NIC for GSO
 *      @gso_min_segs:  Minimum number of segments that can be passed to the
 *                      NIC for GSO
 *
 *      @dcbnl_ops:     Data Center Bridging netlink ops
 *      @num_tc:        Number of traffic classes in the net device
 *      @tc_to_txq:     XXX: need comments on this one
 *      @prio_tc_map    XXX: need comments on this one
 *
 *      @fcoe_ddp_xid:  Max exchange id for FCoE LRO by ddp
 *
 *      @priomap:       XXX: need comments on this one
 *      @phydev:        Physical device may attach itself
 *                      for hardware timestamping
 *
 *      @qdisc_tx_busylock:     XXX: need comments on this one
 *
 *      FIXME: cleanup struct net_device such that network protocol info
 *      moves out.
 */

struct net_device {
        char                    name[IFNAMSIZ];
        struct hlist_node       name_hlist;
        char                    *ifalias;
        /*
```

```
         *      I/O specific fields
         *      FIXME: Merge these and struct ifmap into one
         */
        unsigned long           mem_end;
        unsigned long           mem_start;
        unsigned long           base_addr;
        int                     irq;

        atomic_t                carrier_changes;

        /*
         *      Some hardware also needs these fields (state,dev_list,
         *      napi_list,unreg_list,close_list) but they are not
         *      part of the usual set specified in Space.c.
         */

        unsigned long           state;

        struct list_head        dev_list;
        struct list_head        napi_list;
        struct list_head        unreg_list;
        struct list_head        close_list;
        struct list_head        ptype_all;
        struct list_head        ptype_specific;

        struct {
                struct list_head upper;
                struct list_head lower;
        } adj_list;

        struct {
                struct list_head upper;
                struct list_head lower;
        } all_adj_list;

        netdev_features_t       features;
        netdev_features_t       hw_features;
        netdev_features_t       wanted_features;
        netdev_features_t       vlan_features;
        netdev_features_t       hw_enc_features;
        netdev_features_t       mpls_features;

        int                     ifindex;
        int                     group;

        struct net_device_stats stats;

        atomic_long_t           rx_dropped;
        atomic_long_t           tx_dropped;

#ifdef CONFIG_WIRELESS_EXT
        const struct iw_handler_def *   wireless_handlers;
        struct iw_public_data * wireless_data;
#endif
        const struct net_device_ops *netdev_ops;
        const struct ethtool_ops *ethtool_ops;
#ifdef CONFIG_NET_SWITCHDEV
        const struct switchdev_ops *switchdev_ops;
#endif

        const struct header_ops *header_ops;

        unsigned int            flags;
        unsigned int            priv_flags;

        unsigned short          gflags;
        unsigned short          padded;

        unsigned char           operstate;
        unsigned char           link_mode;

        unsigned char           if_port;
        unsigned char           dma;

        unsigned int            mtu;
        unsigned short          type;
        unsigned short          hard_header_len;

        unsigned short          needed_headroom;
        unsigned short          needed_tailroom;

        /* Interface address info. */
        unsigned char           perm_addr[MAX_ADDR_LEN];
        unsigned char           addr_assign_type;
        unsigned char           addr_len;
        unsigned short          neigh_priv_len;
        unsigned short          dev_id;
        unsigned short          dev_port;
        spinlock_t              addr_list_lock;
        unsigned char           name_assign_type;
        bool                    uc_promisc;
        struct netdev_hw_addr_list      uc;
        struct netdev_hw_addr_list      mc;
        struct netdev_hw_addr_list      dev_addrs;

#ifdef CONFIG_SYSFS
        struct kset             *queues_kset;
#endif
        unsigned int            promiscuity;
        unsigned int            allmulti;


        /* Protocol specific pointers */

#if IS_ENABLED(CONFIG_VLAN_8021Q)
        struct vlan_info __rcu  *vlan_info;
#endif
#if IS_ENABLED(CONFIG_NET_DSA)
        struct dsa_switch_tree  *dsa_ptr;
#endif
#if IS_ENABLED(CONFIG_TIPC)
        struct tipc_bearer __rcu *tipc_ptr;
#endif
        void                    *atalk_ptr;
        struct in_device __rcu  *ip_ptr;
        struct dn_dev __rcu     *dn_ptr;
        struct inet6_dev __rcu  *ip6_ptr;
        void                    *ax25_ptr;
        struct wireless_dev     *ieee80211_ptr;
        struct wpan_dev         *ieee802154_ptr;
#if IS_ENABLED(CONFIG_MPLS_ROUTING)
        struct mpls_dev __rcu   *mpls_ptr;
#endif

/*
 * Cache lines mostly used on receive path (including eth_type_trans())
 */
        unsigned long           last_rx;

        /* Interface address info used in eth_type_trans() */
        unsigned char           *dev_addr;


#ifdef CONFIG_SYSFS
        struct netdev_rx_queue  *_rx;

        unsigned int            num_rx_queues;
        unsigned int            real_num_rx_queues;

#endif

        unsigned long           gro_flush_timeout;
        rx_handler_func_t __rcu *rx_handler;
        void __rcu              *rx_handler_data;

#ifdef CONFIG_NET_CLS_ACT
```

```
        struct tcf_proto __rcu  *ingress_cl_list;
#endif
        struct netdev_queue __rcu *ingress_queue;
#ifdef CONFIG_NETFILTER_INGRESS
        struct list_head        nf_hooks_ingress;
#endif

        unsigned char           broadcast[MAX_ADDR_LEN];
#ifdef CONFIG_RFS_ACCEL
        struct cpu_rmap         *rx_cpu_rmap;
#endif
        struct hlist_node       index_hlist;

/*
 * Cache lines mostly used on transmit path
 */
        struct netdev_queue     *_tx ____cacheline_aligned_in_smp;
        unsigned int            num_tx_queues;
        unsigned int            real_num_tx_queues;
        struct Qdisc            *qdisc;
        unsigned long           tx_queue_len;
        spinlock_t              tx_global_lock;
        int                     watchdog_timeo;

#ifdef CONFIG_XPS
        struct xps_dev_maps __rcu *xps_maps;
#endif

        /* These may be needed for future network-power-down code. */

        /*
         * trans_start here is expensive for high speed devices on SMP,
         * please use netdev_queue->trans_start instead.
         */
        unsigned long           trans_start;

        struct timer_list       watchdog_timer;

        int __percpu            *pcpu_refcnt;
        struct list_head        todo_list;

        struct list_head        link_watch_list;

        enum { NETREG_UNINITIALIZED=0,
               NETREG_REGISTERED,       /* completed register_netdevice */
               NETREG_UNREGISTERING,    /* called unregister_netdevice */
               NETREG_UNREGISTERED,     /* completed unregister todo */
               NETREG_RELEASED,         /* called free_netdev */
               NETREG_DUMMY,            /* dummy device for NAPI poll */
        } reg_state:8;

        bool dismantle;

        enum {
                RTNL_LINK_INITIALIZED,
                RTNL_LINK_INITIALIZING,
        } rtnl_link_state:16;

        void (*destructor)(struct net_device *dev);

#ifdef CONFIG_NETPOLL
        struct netpoll_info __rcu       *npinfo;
#endif

        possible_net_t                  nd_net;

        /* mid-layer private */
        union {
                void                                    *ml_priv;
                struct pcpu_lstats __percpu             *lstats;
                struct pcpu_sw_netstats __percpu        *tstats;
                struct pcpu_dstats __percpu             *dstats;
                struct pcpu_vstats __percpu             *vstats;
        };

        struct garp_port __rcu  *garp_port;
        struct mrp_port __rcu   *mrp_port;

        struct device   dev;
        const struct attribute_group *sysfs_groups[4];
        const struct attribute_group *sysfs_rx_queue_group;

        const struct rtnl_link_ops *rtnl_link_ops;

        /* for setting kernel sock attribute on TCP connection setup */
#define GSO_MAX_SIZE            65536
        unsigned int            gso_max_size;
#define GSO_MAX_SEGS            65535
        u16                     gso_max_segs;
        u16                     gso_min_segs;
#ifdef CONFIG_DCB
        const struct dcbnl_rtnl_ops *dcbnl_ops;
#endif
        u8 num_tc;
        struct netdev_tc_txq tc_to_txq[TC_MAX_QUEUE];
        u8 prio_tc_map[TC_BITMASK + 1];

#if IS_ENABLED(CONFIG_FCOE)
        unsigned int            fcoe_ddp_xid;
#endif
#if IS_ENABLED(CONFIG_CGROUP_NET_PRIO)
        struct netprio_map __rcu *priomap;
#endif
        struct phy_device *phydev;
        struct lock_class_key *qdisc_tx_busylock;
};
#define to_net_dev(d) container_of(d, struct net_device, dev)

#define NETDEV_ALIGN            32

static inline
int netdev_get_prio_tc_map(const struct net_device *dev, u32 prio)
{
        return dev->prio_tc_map[prio & TC_BITMASK];
}

static inline
int netdev_set_prio_tc_map(struct net_device *dev, u8 prio, u8 tc)
{
        if (tc >= dev->num_tc)
                return -EINVAL;

        dev->prio_tc_map[prio & TC_BITMASK] = tc & TC_BITMASK;
        return 0;
}

static inline
void netdev_reset_tc(struct net_device *dev)
{
        dev->num_tc = 0;
        memset(dev->tc_to_txq, 0, sizeof(dev->tc_to_txq));
        memset(dev->prio_tc_map, 0, sizeof(dev->prio_tc_map));
}

static inline
int netdev_set_tc_queue(struct net_device *dev, u8 tc, u16 count, u16 offset)
{
        if (tc >= dev->num_tc)
                return -EINVAL;

        dev->tc_to_txq[tc].count = count;
        dev->tc_to_txq[tc].offset = offset;
        return 0;
}

static inline
```

```
int netdev_set_num_tc(struct net_device *dev, u8 num_tc)
{
        if (num_tc > TC_MAX_QUEUE)
                return -EINVAL;

        dev->num_tc = num_tc;
        return 0;
}

static inline
int netdev_get_num_tc(struct net_device *dev)
{
        return dev->num_tc;
}

static inline
struct netdev_queue *netdev_get_tx_queue(const struct net_device *dev,
                                         unsigned int index)
{
        return &dev->_tx[index];
}

static inline struct netdev_queue *skb_get_tx_queue(const struct net_device *dev,
                                                    const struct sk_buff *skb)
{
        return netdev_get_tx_queue(dev, skb_get_queue_mapping(skb));
}

static inline void netdev_for_each_tx_queue(struct net_device *dev,
                                            void (*f)(struct net_device *,
                                                      struct netdev_queue *,
                                                      void *),
                                            void *arg)
{
        unsigned int i;

        for (i = 0; i < dev->num_tx_queues; i++)
                f(dev, &dev->_tx[i], arg);
}

struct netdev_queue *netdev_pick_tx(struct net_device *dev,
                                    struct sk_buff *skb,
                                    void *accel_priv);

/*
 * Net namespace inlines
 */
static inline
struct net *dev_net(const struct net_device *dev)
{
        return read_pnet(&dev->nd_net);
}

static inline
void dev_net_set(struct net_device *dev, struct net *net)
{
        write_pnet(&dev->nd_net, net);
}

static inline bool netdev_uses_dsa(struct net_device *dev)
{
#if IS_ENABLED(CONFIG_NET_DSA)
        if (dev->dsa_ptr != NULL)
                return dsa_uses_tagged_protocol(dev->dsa_ptr);
#endif
        return false;
}

/**
 *      netdev_priv - access network device private data
 *      @dev: network device
 *
 * Get network device private data
 */
static inline void *netdev_priv(const struct net_device *dev)
{
        return (char *)dev + ALIGN(sizeof(struct net_device), NETDEV_ALIGN);
}

/* Set the sysfs physical device reference for the network logical device
 * if set prior to registration will cause a symlink during initialization.
 */
#define SET_NETDEV_DEV(net, pdev)       ((net)->dev.parent = (pdev))

/* Set the sysfs device type for the network logical device to allow
 * fine-grained identification of different network device types. For
 * example Ethernet, Wirelss LAN, Bluetooth, WiMAX etc.
 */
#define SET_NETDEV_DEVTYPE(net, devtype)        ((net)->dev.type = (devtype))

/* Default NAPI poll() weight
 * Device drivers are strongly advised to not use bigger value
 */
#define NAPI_POLL_WEIGHT 64

/**
 *      netif_napi_add - initialize a napi context
 *      @dev:  network device
 *      @napi: napi context
 *      @poll: polling function
 *      @weight: default weight
 *
 * netif_napi_add() must be used to initialize a napi context prior to calling
 * *any* of the other napi related functions.
 */
void netif_napi_add(struct net_device *dev, struct napi_struct *napi,
                    int (*poll)(struct napi_struct *, int), int weight);

/**
 *  netif_napi_del - remove a napi context
 *  @napi: napi context
 *
 *  netif_napi_del() removes a napi context from the network device napi list
 */
void netif_napi_del(struct napi_struct *napi);

struct napi_gro_cb {
        /* Virtual address of skb_shinfo(skb)->frags[0].page + offset. */
        void *frag0;

        /* Length of frag0. */
        unsigned int frag0_len;

        /* This indicates where we are processing relative to skb->data. */
        int data_offset;

        /* This is non-zero if the packet cannot be merged with the new skb. */
        u16     flush;

        /* Save the IP ID here and check when we get to the transport layer */
        u16     flush_id;

        /* Number of segments aggregated. */
        u16     count;

        /* Start offset for remote checksum offload */
        u16     gro_remcsum_start;

        /* jiffies when first packet was created/queued */
        unsigned long age;

        /* Used in ipv6_gro_receive() and foo-over-udp */
        u16     proto;

        /* This is non-zero if the packet may be of the same flow. */
        u8      same_flow:1;

        /* Used in udp_gro_receive */
        u8      udp_mark:1;

        /* GRO checksum is valid */
        u8      csum_valid:1;

        /* Number of checksums via CHECKSUM_UNNECESSARY */
        u8      csum_cnt:3;

        /* Free the skb? */
        u8      free:2;
#define NAPI_GRO_FREE             1
#define NAPI_GRO_FREE_STOLEN_HEAD 2

        /* Used in foo-over-udp, set in udp[46]_gro_receive */
        u8      is_ipv6:1;

        /* 7 bit hole */

        /* used to support CHECKSUM_COMPLETE for tunneling protocols */
        __wsum  csum;

        /* used in skb_gro_receive() slow path */
        struct sk_buff *last;
};

#define NAPI_GRO_CB(skb) ((struct napi_gro_cb *)(skb)->cb)

struct packet_type {
        __be16                  type;   /* This is really htons(ether_type). */
        struct net_device       *dev;   /* NULL is wildcarded here           */
        int                     (*func) (struct sk_buff *,
                                         struct net_device *,
                                         struct packet_type *,
                                         struct net_device *);
        bool                    (*id_match)(struct packet_type *ptype,
                                            struct sock *sk);
        void                    *af_packet_priv;
        struct list_head        list;
};

struct offload_callbacks {
        struct sk_buff          *(*gso_segment)(struct sk_buff *skb,
                                                netdev_features_t features);
        struct sk_buff          **(*gro_receive)(struct sk_buff **head,
                                                 struct sk_buff *skb);
        int                     (*gro_complete)(struct sk_buff *skb, int nhoff);
};

struct packet_offload {
        __be16                   type;  /* This is really htons(ether_type). */
        u16                      priority;
        struct offload_callbacks callbacks;
        struct list_head         list;
};

struct udp_offload;

struct udp_offload_callbacks {
        struct sk_buff          **(*gro_receive)(struct sk_buff **head,
                                                 struct sk_buff *skb,
                                                 struct udp_offload *uoff);
        int                     (*gro_complete)(struct sk_buff *skb,
                                                int nhoff,
                                                struct udp_offload *uoff);
};

struct udp_offload {
        __be16                   port;
        u8                       ipproto;
        struct udp_offload_callbacks callbacks;
};
```

```

/* often modified stats are per cpu, other are shared (netdev->stats) */
struct pcpu_sw_netstats {
        u64     rx_packets;
        u64     rx_bytes;
        u64     tx_packets;
        u64     tx_bytes;
        struct u64_stats_sync   syncp;
};

#define netdev_alloc_pcpu_stats(type)                           \
({                                                              \
        typeof(type) __percpu *pcpu_stats = alloc_percpu(type); \
        if (pcpu_stats) {                                       \
                int __cpu;                                      \
                for_each_possible_cpu(__cpu) {                  \
                        typeof(type) *stat;                     \
                        stat = per_cpu_ptr(pcpu_stats, __cpu);  \
                        u64_stats_init(&stat->syncp);           \
                }                                               \
        }                                                       \
        pcpu_stats;                                             \
})

#include <linux/notifier.h>

/* netdevice notifier chain. Please remember to update the rtnetlink
 * notification exclusion list in rtnetlink_event() when adding new
 * types.
 */
#define NETDEV_UP       0x0001  /* For now you can't veto a device up/down */
#define NETDEV_DOWN     0x0002
#define NETDEV_REBOOT   0x0003  /* Tell a protocol stack a network interface
                                   detected a hardware crash and restarted
                                   - we can use this eg to kick tcp sessions
                                   once done */
#define NETDEV_CHANGE   0x0004  /* Notify device state change */
#define NETDEV_REGISTER 0x0005
#define NETDEV_UNREGISTER       0x0006
#define NETDEV_CHANGEMTU        0x0007 /* notify after mtu change happened */
#define NETDEV_CHANGEADDR       0x0008
#define NETDEV_GOING_DOWN       0x0009
#define NETDEV_CHANGENAME       0x000A
#define NETDEV_FEAT_CHANGE      0x000B
#define NETDEV_BONDING_FAILOVER 0x000C
#define NETDEV_PRE_UP           0x000D
#define NETDEV_PRE_TYPE_CHANGE  0x000E
#define NETDEV_POST_TYPE_CHANGE 0x000F
#define NETDEV_POST_INIT        0x0010
#define NETDEV_UNREGISTER_FINAL 0x0011
#define NETDEV_RELEASE          0x0012
#define NETDEV_NOTIFY_PEERS     0x0013
#define NETDEV_JOIN             0x0014
#define NETDEV_CHANGEUPPER      0x0015
#define NETDEV_RESEND_IGMP      0x0016
#define NETDEV_PRECHANGEMTU     0x0017 /* notify before mtu change happened */
#define NETDEV_CHANGEINFODATA   0x0018
#define NETDEV_BONDING_INFO     0x0019

int register_netdevice_notifier(struct notifier_block *nb);
int unregister_netdevice_notifier(struct notifier_block *nb);

struct netdev_notifier_info {
        struct net_device *dev;
};

struct netdev_notifier_change_info {
        struct netdev_notifier_info info; /* must be first */
        unsigned int flags_changed;
};

static inline void netdev_notifier_info_init(struct netdev_notifier_info *info,
                                             struct net_device *dev)
```

```
{
        info->dev = dev;
}

static inline struct net_device *
netdev_notifier_info_to_dev(const struct netdev_notifier_info *info)
{
        return info->dev;
}

int call_netdevice_notifiers(unsigned long val, struct net_device *dev);


extern rwlock_t                         dev_base_lock;          /* Device list lock */

#define for_each_netdev(net, d)         \
                list_for_each_entry(d, &(net)->dev_base_head, dev_list)
#define for_each_netdev_reverse(net, d) \
                list_for_each_entry_reverse(d, &(net)->dev_base_head, dev_list)
#define for_each_netdev_rcu(net, d)             \
                list_for_each_entry_rcu(d, &(net)->dev_base_head, dev_list)
#define for_each_netdev_safe(net, d, n) \
                list_for_each_entry_safe(d, n, &(net)->dev_base_head, dev_list)
#define for_each_netdev_continue(net, d)                \
                list_for_each_entry_continue(d, &(net)->dev_base_head, dev_list)
#define for_each_netdev_continue_rcu(net, d)            \
        list_for_each_entry_continue_rcu(d, &(net)->dev_base_head, dev_list)
#define for_each_netdev_in_bond_rcu(bond, slave)        \
                for_each_netdev_rcu(&init_net, slave)   \
                        if (netdev_master_upper_dev_get_rcu(slave) == (bond))
#define net_device_entry(lh)    list_entry(lh, struct net_device, dev_list)

static inline struct net_device *next_net_device(struct net_device *dev)
{
        struct list_head *lh;
        struct net *net;

        net = dev_net(dev);
        lh = dev->dev_list.next;
        return lh == &net->dev_base_head ? NULL : net_device_entry(lh);
}

static inline struct net_device *next_net_device_rcu(struct net_device *dev)
{
        struct list_head *lh;
        struct net *net;

        net = dev_net(dev);
        lh = rcu_dereference(list_next_rcu(&dev->dev_list));
        return lh == &net->dev_base_head ? NULL : net_device_entry(lh);
}

static inline struct net_device *first_net_device(struct net *net)
{
        return list_empty(&net->dev_base_head) ? NULL :
                net_device_entry(net->dev_base_head.next);
}

static inline struct net_device *first_net_device_rcu(struct net *net)
{
        struct list_head *lh = rcu_dereference(list_next_rcu(&net->dev_base_head));

        return lh == &net->dev_base_head ? NULL : net_device_entry(lh);
}

int netdev_boot_setup_check(struct net_device *dev);
unsigned long netdev_boot_base(const char *prefix, int unit);
struct net_device *dev_getbyhwaddr_rcu(struct net *net, unsigned short type,
                                       const char *hwaddr);
struct net_device *dev_getfirstbyhwtype(struct net *net, unsigned short type);
struct net_device *__dev_getfirstbyhwtype(struct net *net, unsigned short type);
void dev_add_pack(struct packet_type *pt);
void dev_remove_pack(struct packet_type *pt);
void __dev_remove_pack(struct packet_type *pt);
void dev_add_offload(struct packet_offload *po);
void dev_remove_offload(struct packet_offload *po);

int dev_get_iflink(const struct net_device *dev);
struct net_device *__dev_get_by_flags(struct net *net, unsigned short flags,
                                      unsigned short mask);
struct net_device *dev_get_by_name(struct net *net, const char *name);
struct net_device *dev_get_by_name_rcu(struct net *net, const char *name);
struct net_device *__dev_get_by_name(struct net *net, const char *name);
int dev_alloc_name(struct net_device *dev, const char *name);
int dev_open(struct net_device *dev);
int dev_close(struct net_device *dev);
int dev_close_many(struct list_head *head, bool unlink);
void dev_disable_lro(struct net_device *dev);
int dev_loopback_xmit(struct sock *sk, struct sk_buff *newskb);
int dev_queue_xmit_sk(struct sock *sk, struct sk_buff *skb);
static inline int dev_queue_xmit(struct sk_buff *skb)
{
        return dev_queue_xmit_sk(skb->sk, skb);
}
int dev_queue_xmit_accel(struct sk_buff *skb, void *accel_priv);
int register_netdevice(struct net_device *dev);
void unregister_netdevice_queue(struct net_device *dev, struct list_head *head);
void unregister_netdevice_many(struct list_head *head);
static inline void unregister_netdevice(struct net_device *dev)
{
        unregister_netdevice_queue(dev, NULL);
}

int netdev_refcnt_read(const struct net_device *dev);
void free_netdev(struct net_device *dev);
void netdev_freemem(struct net_device *dev);
void synchronize_net(void);
int init_dummy_netdev(struct net_device *dev);

DECLARE_PER_CPU(int, xmit_recursion);
static inline int dev_recursion_level(void)
{
        return this_cpu_read(xmit_recursion);
}

struct net_device *dev_get_by_index(struct net *net, int ifindex);
struct net_device *__dev_get_by_index(struct net *net, int ifindex);
struct net_device *dev_get_by_index_rcu(struct net *net, int ifindex);
int netdev_get_name(struct net *net, char *name, int ifindex);
int dev_restart(struct net_device *dev);
int skb_gro_receive(struct sk_buff **head, struct sk_buff *skb);

static inline unsigned int skb_gro_offset(const struct sk_buff *skb)
{
        return NAPI_GRO_CB(skb)->data_offset;
}

static inline unsigned int skb_gro_len(const struct sk_buff *skb)
{
        return skb->len - NAPI_GRO_CB(skb)->data_offset;
}

static inline void skb_gro_pull(struct sk_buff *skb, unsigned int len)
{
        NAPI_GRO_CB(skb)->data_offset += len;
}

static inline void *skb_gro_header_fast(struct sk_buff *skb,
                                        unsigned int offset)
{
        return NAPI_GRO_CB(skb)->frag0 + offset;
}

static inline int skb_gro_header_hard(struct sk_buff *skb, unsigned int hlen)
{
        return NAPI_GRO_CB(skb)->frag0_len < hlen;
```

```
}

static inline void *skb_gro_header_slow(struct sk_buff *skb, unsigned int hlen,
                                        unsigned int offset)
{
        if (!pskb_may_pull(skb, hlen))
                return NULL;

        NAPI_GRO_CB(skb)->frag0 = NULL;
        NAPI_GRO_CB(skb)->frag0_len = 0;
        return skb->data + offset;
}

static inline void *skb_gro_network_header(struct sk_buff *skb)
{
        return (NAPI_GRO_CB(skb)->frag0 ?: skb->data) +
               skb_network_offset(skb);
}

static inline void skb_gro_postpull_rcsum(struct sk_buff *skb,
                                        const void *start, unsigned int len)
{
        if (NAPI_GRO_CB(skb)->csum_valid)
                NAPI_GRO_CB(skb)->csum = csum_sub(NAPI_GRO_CB(skb)->csum,
                                                  csum_partial(start, len, 0));
}

/* GRO checksum functions. These are logical equivalents of the normal
 * checksum functions (in skbuff.h) except that they operate on the GRO
 * offsets and fields in sk_buff.
 */

__sum16 __skb_gro_checksum_complete(struct sk_buff *skb);

static inline bool skb_at_gro_remcsum_start(struct sk_buff *skb)
{
        return (NAPI_GRO_CB(skb)->gro_remcsum_start - skb_headroom(skb) ==
                skb_gro_offset(skb));
}

static inline bool __skb_gro_checksum_validate_needed(struct sk_buff *skb,
                                                      bool zero_okay,
                                                      __sum16 check)
{
        return ((skb->ip_summed != CHECKSUM_PARTIAL ||
                skb_checksum_start_offset(skb) <
                 skb_gro_offset(skb)) &&
                !skb_at_gro_remcsum_start(skb) &&
                NAPI_GRO_CB(skb)->csum_cnt == 0 &&
                (!zero_okay || check));
}

static inline __sum16 __skb_gro_checksum_validate_complete(struct sk_buff *skb,
                                                           __wsum psum)
{
        if (NAPI_GRO_CB(skb)->csum_valid &&
            !csum_fold(csum_add(psum, NAPI_GRO_CB(skb)->csum)))
                return 0;

        NAPI_GRO_CB(skb)->csum = psum;

        return __skb_gro_checksum_complete(skb);
}

static inline void skb_gro_incr_csum_unnecessary(struct sk_buff *skb)
{
        if (NAPI_GRO_CB(skb)->csum_cnt > 0) {
                /* Consume a checksum from CHECKSUM_UNNECESSARY */
                NAPI_GRO_CB(skb)->csum_cnt--;
        } else {
                /* Update skb for CHECKSUM_UNNECESSARY and csum_level when we
                 * verified a new top level checksum or an encapsulated one
                 * during GRO. This saves work if we fallback to normal path.
                 */
                __skb_incr_checksum_unnecessary(skb);
        }
}

#define __skb_gro_checksum_validate(skb, proto, zero_okay, check,       \
                                    compute_pseudo)                     \
({                                                                      \
        __sum16 __ret = 0;                                              \
        if (__skb_gro_checksum_validate_needed(skb, zero_okay, check))  \
                __ret = __skb_gro_checksum_validate_complete(skb,       \
                                compute_pseudo(skb, proto));            \
        if (__ret)                                                      \
                __skb_mark_checksum_bad(skb);                           \
        else                                                            \
                skb_gro_incr_csum_unnecessary(skb);                     \
        __ret;                                                          \
})

#define skb_gro_checksum_validate(skb, proto, compute_pseudo)           \
        __skb_gro_checksum_validate(skb, proto, false, 0, compute_pseudo)

#define skb_gro_checksum_validate_zero_check(skb, proto, check,         \
                                             compute_pseudo)            \
        __skb_gro_checksum_validate(skb, proto, true, check, compute_pseudo)

#define skb_gro_checksum_simple_validate(skb)                           \
        __skb_gro_checksum_validate(skb, 0, false, 0, null_compute_pseudo)

static inline bool __skb_gro_checksum_convert_check(struct sk_buff *skb)
{
        return (NAPI_GRO_CB(skb)->csum_cnt == 0 &&
                !NAPI_GRO_CB(skb)->csum_valid);
}

static inline void __skb_gro_checksum_convert(struct sk_buff *skb,
                                              __sum16 check, __wsum pseudo)
{
        NAPI_GRO_CB(skb)->csum = ~pseudo;
        NAPI_GRO_CB(skb)->csum_valid = 1;
}

#define skb_gro_checksum_try_convert(skb, proto, check, compute_pseudo) \
do {                                                                    \
        if (__skb_gro_checksum_convert_check(skb))                      \
                __skb_gro_checksum_convert(skb, check,                  \
                                           compute_pseudo(skb, proto)); \
} while (0)

struct gro_remcsum {
        int offset;
        __wsum delta;
};

static inline void skb_gro_remcsum_init(struct gro_remcsum *grc)
{
        grc->offset = 0;
        grc->delta = 0;
}

static inline void skb_gro_remcsum_process(struct sk_buff *skb, void *ptr,
                                           int start, int offset,
                                           struct gro_remcsum *grc,
                                           bool nopartial)
{
        __wsum delta;

        BUG_ON(!NAPI_GRO_CB(skb)->csum_valid);

        if (!nopartial) {
                NAPI_GRO_CB(skb)->gro_remcsum_start =
                    ((unsigned char *)ptr + start) - skb->head;
                return;
        }

        delta = remcsum_adjust(ptr, NAPI_GRO_CB(skb)->csum, start, offset);

        /* Adjust skb->csum since we changed the packet */
        NAPI_GRO_CB(skb)->csum = csum_add(NAPI_GRO_CB(skb)->csum, delta);

        grc->offset = (ptr + offset) - (void *)skb->head;
        grc->delta = delta;
}

static inline void skb_gro_remcsum_cleanup(struct sk_buff *skb,
                                           struct gro_remcsum *grc)
{
        if (!grc->delta)
                return;

        remcsum_unadjust((__sum16 *)(skb->head + grc->offset), grc->delta);
}

static inline int dev_hard_header(struct sk_buff *skb, struct net_device *dev,
                                  unsigned short type,
                                  const void *daddr, const void *saddr,
                                  unsigned int len)
{
        if (!dev->header_ops || !dev->header_ops->create)
                return 0;

        return dev->header_ops->create(skb, dev, type, daddr, saddr, len);
}

static inline int dev_parse_header(const struct sk_buff *skb,
                                   unsigned char *haddr)
{
        const struct net_device *dev = skb->dev;

        if (!dev->header_ops || !dev->header_ops->parse)
                return 0;
        return dev->header_ops->parse(skb, haddr);
}

typedef int gifconf_func_t(struct net_device * dev, char __user * bufptr, int len);
int register_gifconf(unsigned int family, gifconf_func_t *gifconf);
static inline int unregister_gifconf(unsigned int family)
{
        return register_gifconf(family, NULL);
}

#ifdef CONFIG_NET_FLOW_LIMIT
#define FLOW_LIMIT_HISTORY      (1 << 7)  /* must be ^2 and !overflow buckets */
struct sd_flow_limit {
        u64                     count;
        unsigned int            num_buckets;
        unsigned int            history_head;
        u16                     history[FLOW_LIMIT_HISTORY];
        u8                      buckets[];
};

extern int netdev_flow_limit_table_len;
#endif /* CONFIG_NET_FLOW_LIMIT */

/*
 * Incoming packets are placed on per-cpu queues
 */
struct softnet_data {
        struct list_head        poll_list;
        struct sk_buff_head     process_queue;

        /* stats */
        unsigned int            processed;
        unsigned int            time_squeeze;
        unsigned int            cpu_collision;
        unsigned int            received_rps;
```

```
#ifdef CONFIG_RPS
        struct softnet_data     *rps_ipi_list;
#endif
#ifdef CONFIG_NET_FLOW_LIMIT
        struct sd_flow_limit __rcu *flow_limit;
#endif
        struct Qdisc            *output_queue;
        struct Qdisc            **output_queue_tailp;
        struct sk_buff          *completion_queue;

#ifdef CONFIG_RPS
        /* Elements below can be accessed between CPUs for RPS */
        struct call_single_data csd ____cacheline_aligned_in_smp;
        struct softnet_data     *rps_ipi_next;
        unsigned int            cpu;
        unsigned int            input_queue_head;
        unsigned int            input_queue_tail;
#endif
        unsigned int            dropped;
        struct sk_buff_head     input_pkt_queue;
        struct napi_struct      backlog;

};

static inline void input_queue_head_incr(struct softnet_data *sd)
{
#ifdef CONFIG_RPS
        sd->input_queue_head++;
#endif
}

static inline void input_queue_tail_incr_save(struct softnet_data *sd,
                                              unsigned int *qtail)
{
#ifdef CONFIG_RPS
        *qtail = ++sd->input_queue_tail;
#endif
}

DECLARE_PER_CPU_ALIGNED(struct softnet_data, softnet_data);

void __netif_schedule(struct Qdisc *q);
void netif_schedule_queue(struct netdev_queue *txq);

static inline void netif_tx_schedule_all(struct net_device *dev)
{
        unsigned int i;

        for (i = 0; i < dev->num_tx_queues; i++)
                netif_schedule_queue(netdev_get_tx_queue(dev, i));
}

static inline void netif_tx_start_queue(struct netdev_queue *dev_queue)
{
        clear_bit(__QUEUE_STATE_DRV_XOFF, &dev_queue->state);
}

/**
 *      netif_start_queue - allow transmit
 *      @dev: network device
 *
 *      Allow upper layers to call the device hard_start_xmit routine.
 */
static inline void netif_start_queue(struct net_device *dev)
{
        netif_tx_start_queue(netdev_get_tx_queue(dev, 0));
}

static inline void netif_tx_start_all_queues(struct net_device *dev)
{
        unsigned int i;

        for (i = 0; i < dev->num_tx_queues; i++) {
                struct netdev_queue *txq = netdev_get_tx_queue(dev, i);
                netif_tx_start_queue(txq);
        }
}

void netif_tx_wake_queue(struct netdev_queue *dev_queue);

/**
 *      netif_wake_queue - restart transmit
 *      @dev: network device
 *
 *      Allow upper layers to call the device hard_start_xmit routine.
 *      Used for flow control when transmit resources are available.
 */
static inline void netif_wake_queue(struct net_device *dev)
{
        netif_tx_wake_queue(netdev_get_tx_queue(dev, 0));
}

static inline void netif_tx_wake_all_queues(struct net_device *dev)
{
        unsigned int i;

        for (i = 0; i < dev->num_tx_queues; i++) {
                struct netdev_queue *txq = netdev_get_tx_queue(dev, i);
                netif_tx_wake_queue(txq);
        }
}

static inline void netif_tx_stop_queue(struct netdev_queue *dev_queue)
{
        set_bit(__QUEUE_STATE_DRV_XOFF, &dev_queue->state);
}

/**
 *      netif_stop_queue - stop transmitted packets
 *      @dev: network device
 *
 *      Stop upper layers calling the device hard_start_xmit routine.
 *      Used for flow control when transmit resources are unavailable.
 */
static inline void netif_stop_queue(struct net_device *dev)
{
        netif_tx_stop_queue(netdev_get_tx_queue(dev, 0));
}

void netif_tx_stop_all_queues(struct net_device *dev);

static inline bool netif_tx_queue_stopped(const struct netdev_queue *dev_queue)
{
        return test_bit(__QUEUE_STATE_DRV_XOFF, &dev_queue->state);
}

/**
 *      netif_queue_stopped - test if transmit queue is flowblocked
 *      @dev: network device
 *
 *      Test if transmit queue on device is currently unable to send.
 */
static inline bool netif_queue_stopped(const struct net_device *dev)
{
        return netif_tx_queue_stopped(netdev_get_tx_queue(dev, 0));
}

static inline bool netif_xmit_stopped(const struct netdev_queue *dev_queue)
{
        return dev_queue->state & QUEUE_STATE_ANY_XOFF;
}

static inline bool
netif_xmit_frozen_or_stopped(const struct netdev_queue *dev_queue)
{
        return dev_queue->state & QUEUE_STATE_ANY_XOFF_OR_FROZEN;
```

```
}

static inline bool
netif_xmit_frozen_or_drv_stopped(const struct netdev_queue *dev_queue)
{
        return dev_queue->state & QUEUE_STATE_DRV_XOFF_OR_FROZEN;
}

/**
 *      netdev_txq_bql_enqueue_prefetchw - prefetch bql data for write
 *      @dev_queue: pointer to transmit queue
 *
 * BQL enabled drivers might use this helper in their ndo_start_xmit(),
 * to give appropriate hint to the cpu.
 */
static inline void netdev_txq_bql_enqueue_prefetchw(struct netdev_queue *dev_queue)
{
#ifdef CONFIG_BQL
        prefetchw(&dev_queue->dql.num_queued);
#endif
}

/**
 *      netdev_txq_bql_complete_prefetchw - prefetch bql data for write
 *      @dev_queue: pointer to transmit queue
 *
 * BQL enabled drivers might use this helper in their TX completion path,
 * to give appropriate hint to the cpu.
 */
static inline void netdev_txq_bql_complete_prefetchw(struct netdev_queue *dev_queue)
{
#ifdef CONFIG_BQL
        prefetchw(&dev_queue->dql.limit);
#endif
}

static inline void netdev_tx_sent_queue(struct netdev_queue *dev_queue,
                                        unsigned int bytes)
{
#ifdef CONFIG_BQL
        dql_queued(&dev_queue->dql, bytes);

        if (likely(dql_avail(&dev_queue->dql) >= 0))
                return;

        set_bit(__QUEUE_STATE_STACK_XOFF, &dev_queue->state);

        /*
         * The XOFF flag must be set before checking the dql_avail below,
         * because in netdev_tx_completed_queue we update the dql_completed
         * before checking the XOFF flag.
         */
        smp_mb();

        /* check again in case another CPU has just made room avail */
        if (unlikely(dql_avail(&dev_queue->dql) >= 0))
                clear_bit(__QUEUE_STATE_STACK_XOFF, &dev_queue->state);
#endif
}

/**
 *      netdev_sent_queue - report the number of bytes queued to hardware
 *      @dev: network device
 *      @bytes: number of bytes queued to the hardware device queue
 *
 *      Report the number of bytes queued for sending/completion to the network
 *      device hardware queue. @bytes should be a good approximation and should
 *      exactly match netdev_completed_queue() @bytes
 */
static inline void netdev_sent_queue(struct net_device *dev, unsigned int bytes)
{
        netdev_tx_sent_queue(netdev_get_tx_queue(dev, 0), bytes);
}
```

```

static inline void netdev_tx_completed_queue(struct netdev_queue *dev_queue,
                                             unsigned int pkts, unsigned int bytes)
{
#ifdef CONFIG_BQL
        if (unlikely(!bytes))
                return;

        dql_completed(&dev_queue->dql, bytes);

        /*
         * Without the memory barrier there is a small possiblity that
         * netdev_tx_sent_queue will miss the update and cause the queue to
         * be stopped forever
         */
        smp_mb();

        if (dql_avail(&dev_queue->dql) < 0)
                return;

        if (test_and_clear_bit(__QUEUE_STATE_STACK_XOFF, &dev_queue->state))
                netif_schedule_queue(dev_queue);
#endif
}

/**
 *      netdev_completed_queue - report bytes and packets completed by device
 *      @dev: network device
 *      @pkts: actual number of packets sent over the medium
 *      @bytes: actual number of bytes sent over the medium
 *
 *      Report the number of bytes and packets transmitted by the network device
 *      hardware queue over the physical medium, @bytes must exactly match the
 *      @bytes amount passed to netdev_sent_queue()
 */
static inline void netdev_completed_queue(struct net_device *dev,
                                          unsigned int pkts, unsigned int bytes)
{
        netdev_tx_completed_queue(netdev_get_tx_queue(dev, 0), pkts, bytes);
}

static inline void netdev_tx_reset_queue(struct netdev_queue *q)
{
#ifdef CONFIG_BQL
        clear_bit(__QUEUE_STATE_STACK_XOFF, &q->state);
        dql_reset(&q->dql);
#endif
}

/**
 *      netdev_reset_queue - reset the packets and bytes count of a network device
 *      @dev_queue: network device
 *
 *      Reset the bytes and packet count of a network device and clear the
 *      software flow control OFF bit for this network device
 */
static inline void netdev_reset_queue(struct net_device *dev_queue)
{
        netdev_tx_reset_queue(netdev_get_tx_queue(dev_queue, 0));
}

/**
 *      netdev_cap_txqueue - check if selected tx queue exceeds device queues
 *      @dev: network device
 *      @queue_index: given tx queue index
 *
 *      Returns 0 if given tx queue index >= number of device tx queues,
 *      otherwise returns the originally passed tx queue index.
 */
static inline u16 netdev_cap_txqueue(struct net_device *dev, u16 queue_index)
{
        if (unlikely(queue_index >= dev->real_num_tx_queues)) {
                net_warn_ratelimited("%s selects TX queue %d, but real number of TX queues is %d\n",
                                     dev->name, queue_index,
                                     dev->real_num_tx_queues);
                return 0;
        }

        return queue_index;
}

/**
 *      netif_running - test if up
 *      @dev: network device
 *
 *      Test if the device has been brought up.
 */
static inline bool netif_running(const struct net_device *dev)
{
        return test_bit(__LINK_STATE_START, &dev->state);
}

/*
 * Routines to manage the subqueues on a device.  We only need start
 * stop, and a check if it's stopped.  All other device management is
 * done at the overall netdevice level.
 * Also test the device if we're multiqueue.
 */

/**
 *      netif_start_subqueue - allow sending packets on subqueue
 *      @dev: network device
 *      @queue_index: sub queue index
 *
 * Start individual transmit queue of a device with multiple transmit queues.
 */
static inline void netif_start_subqueue(struct net_device *dev, u16 queue_index)
{
        struct netdev_queue *txq = netdev_get_tx_queue(dev, queue_index);

        netif_tx_start_queue(txq);
}

/**
 *      netif_stop_subqueue - stop sending packets on subqueue
 *      @dev: network device
 *      @queue_index: sub queue index
 *
 * Stop individual transmit queue of a device with multiple transmit queues.
 */
static inline void netif_stop_subqueue(struct net_device *dev, u16 queue_index)
{
        struct netdev_queue *txq = netdev_get_tx_queue(dev, queue_index);
        netif_tx_stop_queue(txq);
}

/**
 *      netif_subqueue_stopped - test status of subqueue
 *      @dev: network device
 *      @queue_index: sub queue index
 *
 * Check individual transmit queue of a device with multiple transmit queues.
 */
static inline bool __netif_subqueue_stopped(const struct net_device *dev,
                                            u16 queue_index)
{
        struct netdev_queue *txq = netdev_get_tx_queue(dev, queue_index);

        return netif_tx_queue_stopped(txq);
}

static inline bool netif_subqueue_stopped(const struct net_device *dev,
                                          struct sk_buff *skb)
{
        return __netif_subqueue_stopped(dev, skb_get_queue_mapping(skb));
}
```

```

void netif_wake_subqueue(struct net_device *dev, u16 queue_index);

#ifdef CONFIG_XPS
int netif_set_xps_queue(struct net_device *dev, const struct cpumask *mask,
                        u16 index);
#else
static inline int netif_set_xps_queue(struct net_device *dev,
                                      const struct cpumask *mask,
                                      u16 index)
{
        return 0;
}
#endif

u16 __skb_tx_hash(const struct net_device *dev, struct sk_buff *skb,
                  unsigned int num_tx_queues);

/*
 * Returns a Tx hash for the given packet when dev->real_num_tx_queues is used
 * as a distribution range limit for the returned value.
 */
static inline u16 skb_tx_hash(const struct net_device *dev,
                              struct sk_buff *skb)
{
        return __skb_tx_hash(dev, skb, dev->real_num_tx_queues);
}

/**
 *      netif_is_multiqueue - test if device has multiple transmit queues
 *      @dev: network device
 *
 * Check if device has multiple transmit queues
 */
static inline bool netif_is_multiqueue(const struct net_device *dev)
{
        return dev->num_tx_queues > 1;
}

int netif_set_real_num_tx_queues(struct net_device *dev, unsigned int txq);

#ifdef CONFIG_SYSFS
int netif_set_real_num_rx_queues(struct net_device *dev, unsigned int rxq);
#else
static inline int netif_set_real_num_rx_queues(struct net_device *dev,
                                                unsigned int rxq)
{
        return 0;
}
#endif

#ifdef CONFIG_SYSFS
static inline unsigned int get_netdev_rx_queue_index(
                struct netdev_rx_queue *queue)
{
        struct net_device *dev = queue->dev;
        int index = queue - dev->_rx;

        BUG_ON(index >= dev->num_rx_queues);
        return index;
}
#endif

#define DEFAULT_MAX_NUM_RSS_QUEUES      (8)
int netif_get_num_default_rss_queues(void);

enum skb_free_reason {
        SKB_REASON_CONSUMED,
        SKB_REASON_DROPPED,
};

void __dev_kfree_skb_irq(struct sk_buff *skb, enum skb_free_reason reason);
void __dev_kfree_skb_any(struct sk_buff *skb, enum skb_free_reason reason);
```

```

/*
 * It is not allowed to call kfree_skb() or consume_skb() from hardware
 * interrupt context or with hardware interrupts being disabled.
 * (in_irq() || irqs_disabled())
 *
 * We provide four helpers that can be used in following contexts :
 *
 * dev_kfree_skb_irq(skb) when caller drops a packet from irq context,
 *  replacing kfree_skb(skb)
 *
 * dev_consume_skb_irq(skb) when caller consumes a packet from irq context.
 *  Typically used in place of consume_skb(skb) in TX completion path
 *
 * dev_kfree_skb_any(skb) when caller doesn't know its current irq context,
 *  replacing kfree_skb(skb)
 *
 * dev_consume_skb_any(skb) when caller doesn't know its current irq context,
 *  and consumed a packet. Used in place of consume_skb(skb)
 */
static inline void dev_kfree_skb_irq(struct sk_buff *skb)
{
        __dev_kfree_skb_irq(skb, SKB_REASON_DROPPED);
}

static inline void dev_consume_skb_irq(struct sk_buff *skb)
{
        __dev_kfree_skb_irq(skb, SKB_REASON_CONSUMED);
}

static inline void dev_kfree_skb_any(struct sk_buff *skb)
{
        __dev_kfree_skb_any(skb, SKB_REASON_DROPPED);
}

static inline void dev_consume_skb_any(struct sk_buff *skb)
{
        __dev_kfree_skb_any(skb, SKB_REASON_CONSUMED);
}

int netif_rx(struct sk_buff *skb);
int netif_rx_ni(struct sk_buff *skb);
int netif_receive_skb_sk(struct sock *sk, struct sk_buff *skb);
static inline int netif_receive_skb(struct sk_buff *skb)
{
        return netif_receive_skb_sk(skb->sk, skb);
}
gro_result_t napi_gro_receive(struct napi_struct *napi, struct sk_buff *skb);
void napi_gro_flush(struct napi_struct *napi, bool flush_old);
struct sk_buff *napi_get_frags(struct napi_struct *napi);
gro_result_t napi_gro_frags(struct napi_struct *napi);
struct packet_offload *gro_find_receive_by_type(__be16 type);
struct packet_offload *gro_find_complete_by_type(__be16 type);

static inline void napi_free_frags(struct napi_struct *napi)
{
        kfree_skb(napi->skb);
        napi->skb = NULL;
}

int netdev_rx_handler_register(struct net_device *dev,
                               rx_handler_func_t *rx_handler,
                               void *rx_handler_data);
void netdev_rx_handler_unregister(struct net_device *dev);

bool dev_valid_name(const char *name);
int dev_ioctl(struct net *net, unsigned int cmd, void __user *);
int dev_ethtool(struct net *net, struct ifreq *);
unsigned int dev_get_flags(const struct net_device *);
int __dev_change_flags(struct net_device *, unsigned int flags);
int dev_change_flags(struct net_device *, unsigned int);
void __dev_notify_flags(struct net_device *, unsigned int old_flags,
                        unsigned int gchanges);
```

```
int dev_change_name(struct net_device *, const char *);
int dev_set_alias(struct net_device *, const char *, size_t);
int dev_change_net_namespace(struct net_device *, struct net *, const char *);
int dev_set_mtu(struct net_device *, int);
void dev_set_group(struct net_device *, int);
int dev_set_mac_address(struct net_device *, struct sockaddr *);
int dev_change_carrier(struct net_device *, bool new_carrier);
int dev_get_phys_port_id(struct net_device *dev,
                         struct netdev_phys_item_id *ppid);
int dev_get_phys_port_name(struct net_device *dev,
                           char *name, size_t len);
struct sk_buff *validate_xmit_skb_list(struct sk_buff *skb, struct net_device *dev);
struct sk_buff *dev_hard_start_xmit(struct sk_buff *skb, struct net_device *dev,
                                    struct netdev_queue *txq, int *ret);
int __dev_forward_skb(struct net_device *dev, struct sk_buff *skb);
int dev_forward_skb(struct net_device *dev, struct sk_buff *skb);
bool is_skb_forwardable(struct net_device *dev, struct sk_buff *skb);

extern int              netdev_budget;

/* Called by rtnetlink.c:rtnl_unlock() */
void netdev_run_todo(void);

/**
 *      dev_put - release reference to device
 *      @dev: network device
 *
 * Release reference to device to allow it to be freed.
 */
static inline void dev_put(struct net_device *dev)
{
        this_cpu_dec(*dev->pcpu_refcnt);
}

/**
 *      dev_hold - get reference to device
 *      @dev: network device
 *
 * Hold reference to device to keep it from being freed.
 */
static inline void dev_hold(struct net_device *dev)
{
        this_cpu_inc(*dev->pcpu_refcnt);
}

/* Carrier loss detection, dial on demand. The functions netif_carrier_on
 * and _off may be called from IRQ context, but it is caller
 * who is responsible for serialization of these calls.
 *
 * The name carrier is inappropriate, these functions should really be
 * called netif_lowerlayer_*() because they represent the state of any
 * kind of lower layer not just hardware media.
 */

void linkwatch_init_dev(struct net_device *dev);
void linkwatch_fire_event(struct net_device *dev);
void linkwatch_forget_dev(struct net_device *dev);

/**
 *      netif_carrier_ok - test if carrier present
 *      @dev: network device
 *
 * Check if carrier is present on device
 */
static inline bool netif_carrier_ok(const struct net_device *dev)
{
        return !test_bit(__LINK_STATE_NOCARRIER, &dev->state);
}

unsigned long dev_trans_start(struct net_device *dev);

void __netdev_watchdog_up(struct net_device *dev);

void netif_carrier_on(struct net_device *dev);

void netif_carrier_off(struct net_device *dev);

/**
 *      netif_dormant_on - mark device as dormant.
 *      @dev: network device
 *
 * Mark device as dormant (as per RFC2863).
 *
 * The dormant state indicates that the relevant interface is not
 * actually in a condition to pass packets (i.e., it is not 'up') but is
 * in a "pending" state, waiting for some external event.  For "on-
 * demand" interfaces, this new state identifies the situation where the
 * interface is waiting for events to place it in the up state.
 *
 */
static inline void netif_dormant_on(struct net_device *dev)
{
        if (!test_and_set_bit(__LINK_STATE_DORMANT, &dev->state))
                linkwatch_fire_event(dev);
}

/**
 *      netif_dormant_off - set device as not dormant.
 *      @dev: network device
 *
 * Device is not in dormant state.
 */
static inline void netif_dormant_off(struct net_device *dev)
{
        if (test_and_clear_bit(__LINK_STATE_DORMANT, &dev->state))
                linkwatch_fire_event(dev);
}

/**
 *      netif_dormant - test if carrier present
 *      @dev: network device
 *
 * Check if carrier is present on device
 */
static inline bool netif_dormant(const struct net_device *dev)
{
        return test_bit(__LINK_STATE_DORMANT, &dev->state);
}


/**
 *      netif_oper_up - test if device is operational
 *      @dev: network device
 *
 * Check if carrier is operational
 */
static inline bool netif_oper_up(const struct net_device *dev)
{
        return (dev->operstate == IF_OPER_UP ||
                dev->operstate == IF_OPER_UNKNOWN /* backward compat */);
}

/**
 *      netif_device_present - is device available or removed
 *      @dev: network device
 *
 * Check if device has not been removed from system.
 */
static inline bool netif_device_present(struct net_device *dev)
{
        return test_bit(__LINK_STATE_PRESENT, &dev->state);
}

void netif_device_detach(struct net_device *dev);

void netif_device_attach(struct net_device *dev);
```

```

/*
 * Network interface message level settings
 */

enum {
        NETIF_MSG_DRV           = 0x0001,
        NETIF_MSG_PROBE         = 0x0002,
        NETIF_MSG_LINK          = 0x0004,
        NETIF_MSG_TIMER         = 0x0008,
        NETIF_MSG_IFDOWN        = 0x0010,
        NETIF_MSG_IFUP          = 0x0020,
        NETIF_MSG_RX_ERR        = 0x0040,
        NETIF_MSG_TX_ERR        = 0x0080,
        NETIF_MSG_TX_QUEUED     = 0x0100,
        NETIF_MSG_INTR          = 0x0200,
        NETIF_MSG_TX_DONE       = 0x0400,
        NETIF_MSG_RX_STATUS     = 0x0800,
        NETIF_MSG_PKTDATA       = 0x1000,
        NETIF_MSG_HW            = 0x2000,
        NETIF_MSG_WOL           = 0x4000,
};

#define netif_msg_drv(p)        ((p)->msg_enable & NETIF_MSG_DRV)
#define netif_msg_probe(p)      ((p)->msg_enable & NETIF_MSG_PROBE)
#define netif_msg_link(p)       ((p)->msg_enable & NETIF_MSG_LINK)
#define netif_msg_timer(p)      ((p)->msg_enable & NETIF_MSG_TIMER)
#define netif_msg_ifdown(p)     ((p)->msg_enable & NETIF_MSG_IFDOWN)
#define netif_msg_ifup(p)       ((p)->msg_enable & NETIF_MSG_IFUP)
#define netif_msg_rx_err(p)     ((p)->msg_enable & NETIF_MSG_RX_ERR)
#define netif_msg_tx_err(p)     ((p)->msg_enable & NETIF_MSG_TX_ERR)
#define netif_msg_tx_queued(p)  ((p)->msg_enable & NETIF_MSG_TX_QUEUED)
#define netif_msg_intr(p)       ((p)->msg_enable & NETIF_MSG_INTR)
#define netif_msg_tx_done(p)    ((p)->msg_enable & NETIF_MSG_TX_DONE)
#define netif_msg_rx_status(p)  ((p)->msg_enable & NETIF_MSG_RX_STATUS)
#define netif_msg_pktdata(p)    ((p)->msg_enable & NETIF_MSG_PKTDATA)
#define netif_msg_hw(p)         ((p)->msg_enable & NETIF_MSG_HW)
#define netif_msg_wol(p)        ((p)->msg_enable & NETIF_MSG_WOL)

static inline u32 netif_msg_init(int debug_value, int default_msg_enable_bits)
{
        /* use default */
        if (debug_value < 0 || debug_value >= (sizeof(u32) * 8))
                return default_msg_enable_bits;
        if (debug_value == 0)   /* no output */
                return 0;
        /* set low N bits */
        return (1 << debug_value) - 1;
}

static inline void __netif_tx_lock(struct netdev_queue *txq, int cpu)
{
        spin_lock(&txq->_xmit_lock);
        txq->xmit_lock_owner = cpu;
}

static inline void __netif_tx_lock_bh(struct netdev_queue *txq)
{
        spin_lock_bh(&txq->_xmit_lock);
        txq->xmit_lock_owner = smp_processor_id();
}

static inline bool __netif_tx_trylock(struct netdev_queue *txq)
{
        bool ok = spin_trylock(&txq->_xmit_lock);
        if (likely(ok))
                txq->xmit_lock_owner = smp_processor_id();
        return ok;
}

static inline void __netif_tx_unlock(struct netdev_queue *txq)
{
        txq->xmit_lock_owner = -1;
        spin_unlock(&txq->_xmit_lock);
}

static inline void __netif_tx_unlock_bh(struct netdev_queue *txq)
{
        txq->xmit_lock_owner = -1;
        spin_unlock_bh(&txq->_xmit_lock);
}

static inline void txq_trans_update(struct netdev_queue *txq)
{
        if (txq->xmit_lock_owner != -1)
                txq->trans_start = jiffies;
}

/**
 *      netif_tx_lock - grab network device transmit lock
 *      @dev: network device
 *
 * Get network device transmit lock
 */
static inline void netif_tx_lock(struct net_device *dev)
{
        unsigned int i;
        int cpu;

        spin_lock(&dev->tx_global_lock);
        cpu = smp_processor_id();
        for (i = 0; i < dev->num_tx_queues; i++) {
                struct netdev_queue *txq = netdev_get_tx_queue(dev, i);

                /* We are the only thread of execution doing a
                 * freeze, but we have to grab the _xmit_lock in
                 * order to synchronize with threads which are in
                 * the ->hard_start_xmit() handler and already
                 * checked the frozen bit.
                 */
                __netif_tx_lock(txq, cpu);
                set_bit(__QUEUE_STATE_FROZEN, &txq->state);
                __netif_tx_unlock(txq);
        }
}

static inline void netif_tx_lock_bh(struct net_device *dev)
{
        local_bh_disable();
        netif_tx_lock(dev);
}

static inline void netif_tx_unlock(struct net_device *dev)
{
        unsigned int i;

        for (i = 0; i < dev->num_tx_queues; i++) {
                struct netdev_queue *txq = netdev_get_tx_queue(dev, i);

                /* No need to grab the _xmit_lock here.  If the
                 * queue is not stopped for another reason, we
                 * force a schedule.
                 */
                clear_bit(__QUEUE_STATE_FROZEN, &txq->state);
                netif_schedule_queue(txq);
        }
        spin_unlock(&dev->tx_global_lock);
}

static inline void netif_tx_unlock_bh(struct net_device *dev)
{
        netif_tx_unlock(dev);
        local_bh_enable();
}

#define HARD_TX_LOCK(dev, txq, cpu) {                   \
        if ((dev->features & NETIF_F_LLTX) == 0) {      \
                __netif_tx_lock(txq, cpu);              \
        }                                               \
}

#define HARD_TX_TRYLOCK(dev, txq)                       \
        (((dev->features & NETIF_F_LLTX) == 0) ?        \
                __netif_tx_trylock(txq) :               \
                true )

#define HARD_TX_UNLOCK(dev, txq) {                      \
        if ((dev->features & NETIF_F_LLTX) == 0) {      \
                __netif_tx_unlock(txq);                 \
        }                                               \
}

static inline void netif_tx_disable(struct net_device *dev)
{
        unsigned int i;
        int cpu;

        local_bh_disable();
        cpu = smp_processor_id();
        for (i = 0; i < dev->num_tx_queues; i++) {
                struct netdev_queue *txq = netdev_get_tx_queue(dev, i);

                __netif_tx_lock(txq, cpu);
                netif_tx_stop_queue(txq);
                __netif_tx_unlock(txq);
        }
        local_bh_enable();
}

static inline void netif_addr_lock(struct net_device *dev)
{
        spin_lock(&dev->addr_list_lock);
}

static inline void netif_addr_lock_nested(struct net_device *dev)
{
        int subclass = SINGLE_DEPTH_NESTING;

        if (dev->netdev_ops->ndo_get_lock_subclass)
                subclass = dev->netdev_ops->ndo_get_lock_subclass(dev);

        spin_lock_nested(&dev->addr_list_lock, subclass);
}

static inline void netif_addr_lock_bh(struct net_device *dev)
{
        spin_lock_bh(&dev->addr_list_lock);
}

static inline void netif_addr_unlock(struct net_device *dev)
{
        spin_unlock(&dev->addr_list_lock);
}

static inline void netif_addr_unlock_bh(struct net_device *dev)
{
        spin_unlock_bh(&dev->addr_list_lock);
}

/*
 * dev_addrs walker. Should be used only for read access. Call with
 * rcu_read_lock held.
 */
#define for_each_dev_addr(dev, ha) \
                list_for_each_entry_rcu(ha, &dev->dev_addrs.list, list)

/* These functions live elsewhere (drivers/net/net_init.c, but related) */

void ether_setup(struct net_device *dev);
```

```

/* Support for loadable net-drivers */
struct net_device *alloc_netdev_mqs(int sizeof_priv, const char *name,
                                    unsigned char name_assign_type,
                                    void (*setup)(struct net_device *),
                                    unsigned int txqs, unsigned int rxqs);
#define alloc_netdev(sizeof_priv, name, name_assign_type, setup) \
        alloc_netdev_mqs(sizeof_priv, name, name_assign_type, setup, 1, 1)

#define alloc_netdev_mq(sizeof_priv, name, name_assign_type, setup, count) \
        alloc_netdev_mqs(sizeof_priv, name, name_assign_type, setup, count, \
                         count)

int register_netdev(struct net_device *dev);
void unregister_netdev(struct net_device *dev);

/* General hardware address lists handling functions */
int __hw_addr_sync(struct netdev_hw_addr_list *to_list,
                   struct netdev_hw_addr_list *from_list, int addr_len);
void __hw_addr_unsync(struct netdev_hw_addr_list *to_list,
                      struct netdev_hw_addr_list *from_list, int addr_len);
int __hw_addr_sync_dev(struct netdev_hw_addr_list *list,
                       struct net_device *dev,
                       int (*sync)(struct net_device *, const unsigned char *),
                       int (*unsync)(struct net_device *,
                                     const unsigned char *));
void __hw_addr_unsync_dev(struct netdev_hw_addr_list *list,
                          struct net_device *dev,
                          int (*unsync)(struct net_device *,
                                        const unsigned char *));
void __hw_addr_init(struct netdev_hw_addr_list *list);

/* Functions used for device addresses handling */
int dev_addr_add(struct net_device *dev, const unsigned char *addr,
                 unsigned char addr_type);
int dev_addr_del(struct net_device *dev, const unsigned char *addr,
                 unsigned char addr_type);
void dev_addr_flush(struct net_device *dev);
int dev_addr_init(struct net_device *dev);

/* Functions used for unicast addresses handling */
int dev_uc_add(struct net_device *dev, const unsigned char *addr);
int dev_uc_add_excl(struct net_device *dev, const unsigned char *addr);
int dev_uc_del(struct net_device *dev, const unsigned char *addr);
int dev_uc_sync(struct net_device *to, struct net_device *from);
int dev_uc_sync_multiple(struct net_device *to, struct net_device *from);
void dev_uc_unsync(struct net_device *to, struct net_device *from);
void dev_uc_flush(struct net_device *dev);
void dev_uc_init(struct net_device *dev);

/**
 *  __dev_uc_sync - Synchonize device's unicast list
 *  @dev:  device to sync
 *  @sync: function to call if address should be added
 *  @unsync: function to call if address should be removed
 *
 *  Add newly added addresses to the interface, and release
 *  addresses that have been deleted.
 **/
static inline int __dev_uc_sync(struct net_device *dev,
                                int (*sync)(struct net_device *,
                                            const unsigned char *),
                                int (*unsync)(struct net_device *,
                                              const unsigned char *))
{
        return __hw_addr_sync_dev(&dev->uc, dev, sync, unsync);
}

/**
 *  __dev_uc_unsync - Remove synchronized addresses from device
 *  @dev:  device to sync
 *  @unsync: function to call if address should be removed
 *
 *  Remove all addresses that were added to the device by dev_uc_sync().
 **/
static inline void __dev_uc_unsync(struct net_device *dev,
                                   int (*unsync)(struct net_device *,
                                                 const unsigned char *))
{
        __hw_addr_unsync_dev(&dev->uc, dev, unsync);
}

/* Functions used for multicast addresses handling */
int dev_mc_add(struct net_device *dev, const unsigned char *addr);
int dev_mc_add_global(struct net_device *dev, const unsigned char *addr);
int dev_mc_add_excl(struct net_device *dev, const unsigned char *addr);
int dev_mc_del(struct net_device *dev, const unsigned char *addr);
int dev_mc_del_global(struct net_device *dev, const unsigned char *addr);
int dev_mc_sync(struct net_device *to, struct net_device *from);
int dev_mc_sync_multiple(struct net_device *to, struct net_device *from);
void dev_mc_unsync(struct net_device *to, struct net_device *from);
void dev_mc_flush(struct net_device *dev);
void dev_mc_init(struct net_device *dev);

/**
 *  __dev_mc_sync - Synchonize device's multicast list
 *  @dev:  device to sync
 *  @sync: function to call if address should be added
 *  @unsync: function to call if address should be removed
 *
 *  Add newly added addresses to the interface, and release
 *  addresses that have been deleted.
 **/
static inline int __dev_mc_sync(struct net_device *dev,
                                int (*sync)(struct net_device *,
                                            const unsigned char *),
                                int (*unsync)(struct net_device *,
                                              const unsigned char *))
{
        return __hw_addr_sync_dev(&dev->mc, dev, sync, unsync);
}

/**
 *  __dev_mc_unsync - Remove synchronized addresses from device
 *  @dev:  device to sync
 *  @unsync: function to call if address should be removed
 *
 *  Remove all addresses that were added to the device by dev_mc_sync().
 **/
static inline void __dev_mc_unsync(struct net_device *dev,
                                   int (*unsync)(struct net_device *,
                                                 const unsigned char *))
{
        __hw_addr_unsync_dev(&dev->mc, dev, unsync);
}

/* Functions used for secondary unicast and multicast support */
void dev_set_rx_mode(struct net_device *dev);
void __dev_set_rx_mode(struct net_device *dev);
int dev_set_promiscuity(struct net_device *dev, int inc);
int dev_set_allmulti(struct net_device *dev, int inc);
void netdev_state_change(struct net_device *dev);
void netdev_notify_peers(struct net_device *dev);
void netdev_features_change(struct net_device *dev);
/* Load a device via the kmod */
void dev_load(struct net *net, const char *name);
struct rtnl_link_stats64 *dev_get_stats(struct net_device *dev,
                                        struct rtnl_link_stats64 *storage);
void netdev_stats_to_stats64(struct rtnl_link_stats64 *stats64,
                             const struct net_device_stats *netdev_stats);

extern int              netdev_max_backlog;
extern int              netdev_tstamp_prequeue;
extern int              weight_p;
extern int              bpf_jit_enable;

```

```
bool netdev_has_upper_dev(struct net_device *dev, struct net_device *upper_dev);
struct net_device *netdev_upper_get_next_dev_rcu(struct net_device *dev,
                                                     struct list_head **iter);
struct net_device *netdev_all_upper_get_next_dev_rcu(struct net_device *dev,
                                                     struct list_head **iter);

/* iterate through upper list, must be called under RCU read lock */
#define netdev_for_each_upper_dev_rcu(dev, updev, iter) \
        for (iter = &(dev)->adj_list.upper, \
             updev = netdev_upper_get_next_dev_rcu(dev, &(iter)); \
             updev; \
             updev = netdev_upper_get_next_dev_rcu(dev, &(iter)))

/* iterate through upper list, must be called under RCU read lock */
#define netdev_for_each_all_upper_dev_rcu(dev, updev, iter) \
        for (iter = &(dev)->all_adj_list.upper, \
             updev = netdev_all_upper_get_next_dev_rcu(dev, &(iter)); \
             updev; \
             updev = netdev_all_upper_get_next_dev_rcu(dev, &(iter)))

void *netdev_lower_get_next_private(struct net_device *dev,
                                    struct list_head **iter);
void *netdev_lower_get_next_private_rcu(struct net_device *dev,
                                        struct list_head **iter);

#define netdev_for_each_lower_private(dev, priv, iter) \
        for (iter = (dev)->adj_list.lower.next, \
             priv = netdev_lower_get_next_private(dev, &(iter)); \
             priv; \
             priv = netdev_lower_get_next_private(dev, &(iter)))

#define netdev_for_each_lower_private_rcu(dev, priv, iter) \
        for (iter = &(dev)->adj_list.lower, \
             priv = netdev_lower_get_next_private_rcu(dev, &(iter)); \
             priv; \
             priv = netdev_lower_get_next_private_rcu(dev, &(iter)))

void *netdev_lower_get_next(struct net_device *dev,
                                struct list_head **iter);
#define netdev_for_each_lower_dev(dev, ldev, iter) \
        for (iter = &(dev)->adj_list.lower, \
             ldev = netdev_lower_get_next(dev, &(iter)); \
             ldev; \
             ldev = netdev_lower_get_next(dev, &(iter)))

void *netdev_adjacent_get_private(struct list_head *adj_list);
void *netdev_lower_get_first_private_rcu(struct net_device *dev);
struct net_device *netdev_master_upper_dev_get(struct net_device *dev);
struct net_device *netdev_master_upper_dev_get_rcu(struct net_device *dev);
int netdev_upper_dev_link(struct net_device *dev, struct net_device *upper_dev);
int netdev_master_upper_dev_link(struct net_device *dev,
                                 struct net_device *upper_dev);
int netdev_master_upper_dev_link_private(struct net_device *dev,
                                         struct net_device *upper_dev,
                                         void *private);
void netdev_upper_dev_unlink(struct net_device *dev,
                             struct net_device *upper_dev);
void netdev_adjacent_rename_links(struct net_device *dev, char *oldname);
void *netdev_lower_dev_get_private(struct net_device *dev,
                                   struct net_device *lower_dev);

/* RSS keys are 40 or 52 bytes long */
#define NETDEV_RSS_KEY_LEN 52
extern u8 netdev_rss_key[NETDEV_RSS_KEY_LEN];
void netdev_rss_key_fill(void *buffer, size_t len);

int dev_get_nest_level(struct net_device *dev,
                       bool (*type_check)(struct net_device *dev));
int skb_checksum_help(struct sk_buff *skb);
struct sk_buff *__skb_gso_segment(struct sk_buff *skb,
                                  netdev_features_t features, bool tx_path);
struct sk_buff *skb_mac_gso_segment(struct sk_buff *skb,
                                    netdev_features_t features);
```

```

struct netdev_bonding_info {
        ifslave slave;
        ifbond  master;
};

struct netdev_notifier_bonding_info {
        struct netdev_notifier_info info; /* must be first */
        struct netdev_bonding_info  bonding_info;
};

void netdev_bonding_info_change(struct net_device *dev,
                                struct netdev_bonding_info *bonding_info);

static inline
struct sk_buff *skb_gso_segment(struct sk_buff *skb, netdev_features_t features)
{
        return __skb_gso_segment(skb, features, true);
}
__be16 skb_network_protocol(struct sk_buff *skb, int *depth);

static inline bool can_checksum_protocol(netdev_features_t features,
                                         __be16 protocol)
{
        return ((features & NETIF_F_GEN_CSUM) ||
                ((features & NETIF_F_V4_CSUM) &&
                 protocol == htons(ETH_P_IP)) ||
                ((features & NETIF_F_V6_CSUM) &&
                 protocol == htons(ETH_P_IPV6)) ||
                ((features & NETIF_F_FCOE_CRC) &&
                 protocol == htons(ETH_P_FCOE)));
}

#ifdef CONFIG_BUG
void netdev_rx_csum_fault(struct net_device *dev);
#else
static inline void netdev_rx_csum_fault(struct net_device *dev)
{
}
#endif
/* rx skb timestamps */
void net_enable_timestamp(void);
void net_disable_timestamp(void);

#ifdef CONFIG_PROC_FS
int __init dev_proc_init(void);
#else
#define dev_proc_init() 0
#endif

static inline netdev_tx_t __netdev_start_xmit(const struct net_device_ops *ops,
                                              struct sk_buff *skb, struct net_device *dev,
                                              bool more)
{
        skb->xmit_more = more ? 1 : 0;
        return ops->ndo_start_xmit(skb, dev);
}

static inline netdev_tx_t netdev_start_xmit(struct sk_buff *skb, struct net_device *dev,
                                            struct netdev_queue *txq, bool more)
{
        const struct net_device_ops *ops = dev->netdev_ops;
        int rc;

        rc = __netdev_start_xmit(ops, skb, dev, more);
        if (rc == NETDEV_TX_OK)
                txq_trans_update(txq);

        return rc;
}

int netdev_class_create_file_ns(struct class_attribute *class_attr,
                                const void *ns);
```

```
void netdev_class_remove_file_ns(struct class_attribute *class_attr,
                                 const void *ns);

static inline int netdev_class_create_file(struct class_attribute *class_attr)
{
        return netdev_class_create_file_ns(class_attr, NULL);
}

static inline void netdev_class_remove_file(struct class_attribute *class_attr)
{
        netdev_class_remove_file_ns(class_attr, NULL);
}

extern struct kobj_ns_type_operations net_ns_type_operations;

const char *netdev_drivername(const struct net_device *dev);

void linkwatch_run_queue(void);

static inline netdev_features_t netdev_intersect_features(netdev_features_t f1,
                                                          netdev_features_t f2)
{
        if (f1 & NETIF_F_GEN_CSUM)
                f1 |= (NETIF_F_ALL_CSUM & ~NETIF_F_GEN_CSUM);
        if (f2 & NETIF_F_GEN_CSUM)
                f2 |= (NETIF_F_ALL_CSUM & ~NETIF_F_GEN_CSUM);
        f1 &= f2;
        if (f1 & NETIF_F_GEN_CSUM)
                f1 &= ~(NETIF_F_ALL_CSUM & ~NETIF_F_GEN_CSUM);

        return f1;
}

static inline netdev_features_t netdev_get_wanted_features(
        struct net_device *dev)
{
        return (dev->features & ~dev->hw_features) | dev->wanted_features;
}
netdev_features_t netdev_increment_features(netdev_features_t all,
        netdev_features_t one, netdev_features_t mask);

/* Allow TSO being used on stacked device :
 * Performing the GSO segmentation before last device
 * is a performance improvement.
 */
static inline netdev_features_t netdev_add_tso_features(netdev_features_t features,
                                                        netdev_features_t mask)
{
        return netdev_increment_features(features, NETIF_F_ALL_TSO, mask);
}

int __netdev_update_features(struct net_device *dev);
void netdev_update_features(struct net_device *dev);
void netdev_change_features(struct net_device *dev);

void netif_stacked_transfer_operstate(const struct net_device *rootdev,
                                        struct net_device *dev);

netdev_features_t passthru_features_check(struct sk_buff *skb,
                                          struct net_device *dev,
                                          netdev_features_t features);
netdev_features_t netif_skb_features(struct sk_buff *skb);

static inline bool net_gso_ok(netdev_features_t features, int gso_type)
{
        netdev_features_t feature = gso_type << NETIF_F_GSO_SHIFT;

        /* check flags correspondence */
        BUILD_BUG_ON(SKB_GSO_TCPV4   != (NETIF_F_TSO >> NETIF_F_GSO_SHIFT));
        BUILD_BUG_ON(SKB_GSO_UDP     != (NETIF_F_UFO >> NETIF_F_GSO_SHIFT));
        BUILD_BUG_ON(SKB_GSO_DODGY   != (NETIF_F_GSO_ROBUST >> NETIF_F_GSO_SHIFT));
        BUILD_BUG_ON(SKB_GSO_TCP_ECN != (NETIF_F_TSO_ECN >> NETIF_F_GSO_SHIFT));
        BUILD_BUG_ON(SKB_GSO_TCPV6   != (NETIF_F_TSO6 >> NETIF_F_GSO_SHIFT));
        BUILD_BUG_ON(SKB_GSO_FCOE    != (NETIF_F_FSO >> NETIF_F_GSO_SHIFT));
        BUILD_BUG_ON(SKB_GSO_GRE     != (NETIF_F_GSO_GRE >> NETIF_F_GSO_SHIFT));
        BUILD_BUG_ON(SKB_GSO_GRE_CSUM != (NETIF_F_GSO_GRE_CSUM >> NETIF_F_GSO_SHIFT));
        BUILD_BUG_ON(SKB_GSO_IPIP    != (NETIF_F_GSO_IPIP >> NETIF_F_GSO_SHIFT));
        BUILD_BUG_ON(SKB_GSO_SIT     != (NETIF_F_GSO_SIT >> NETIF_F_GSO_SHIFT));
        BUILD_BUG_ON(SKB_GSO_UDP_TUNNEL != (NETIF_F_GSO_UDP_TUNNEL >> NETIF_F_GSO_SHIFT));
        BUILD_BUG_ON(SKB_GSO_UDP_TUNNEL_CSUM != (NETIF_F_GSO_UDP_TUNNEL_CSUM >> NETIF_F_GSO_SHIFT));
        BUILD_BUG_ON(SKB_GSO_TUNNEL_REMCSUM != (NETIF_F_GSO_TUNNEL_REMCSUM >> NETIF_F_GSO_SHIFT));

        return (features & feature) == feature;
}

static inline bool skb_gso_ok(struct sk_buff *skb, netdev_features_t features)
{
        return net_gso_ok(features, skb_shinfo(skb)->gso_type) &&
               (!skb_has_frag_list(skb) || (features & NETIF_F_FRAGLIST));
}

static inline bool netif_needs_gso(struct sk_buff *skb,
                                   netdev_features_t features)
{
        return skb_is_gso(skb) && (!skb_gso_ok(skb, features) ||
                unlikely((skb->ip_summed != CHECKSUM_PARTIAL) &&
                         (skb->ip_summed != CHECKSUM_UNNECESSARY)));
}

static inline void netif_set_gso_max_size(struct net_device *dev,
                                          unsigned int size)
{
        dev->gso_max_size = size;
}

static inline void skb_gso_error_unwind(struct sk_buff *skb, __be16 protocol,
                                        int pulled_hlen, u16 mac_offset,
                                        int mac_len)
{
        skb->protocol = protocol;
        skb->encapsulation = 1;
        skb_push(skb, pulled_hlen);
        skb_reset_transport_header(skb);
        skb->mac_header = mac_offset;
        skb->network_header = skb->mac_header + mac_len;
        skb->mac_len = mac_len;
}

static inline bool netif_is_macvlan(struct net_device *dev)
{
        return dev->priv_flags & IFF_MACVLAN;
}

static inline bool netif_is_macvlan_port(struct net_device *dev)
{
        return dev->priv_flags & IFF_MACVLAN_PORT;
}

static inline bool netif_is_ipvlan(struct net_device *dev)
{
        return dev->priv_flags & IFF_IPVLAN_SLAVE;
}

static inline bool netif_is_ipvlan_port(struct net_device *dev)
{
        return dev->priv_flags & IFF_IPVLAN_MASTER;
}

static inline bool netif_is_bond_master(struct net_device *dev)
{
        return dev->flags & IFF_MASTER && dev->priv_flags & IFF_BONDING;
}

static inline bool netif_is_bond_slave(struct net_device *dev)
{
        return dev->flags & IFF_SLAVE && dev->priv_flags & IFF_BONDING;
```

```
}

static inline bool netif_supports_nofcs(struct net_device *dev)
{
        return dev->priv_flags & IFF_SUPP_NOFCS;
}

/* This device needs to keep skb dst for qdisc enqueue or ndo_start_xmit() */
static inline void netif_keep_dst(struct net_device *dev)
{
        dev->priv_flags &= ~(IFF_XMIT_DST_RELEASE | IFF_XMIT_DST_RELEASE_PERM);
}

extern struct pernet_operations __net_initdata loopback_net_ops;

/* Logging, debugging and troubleshooting/diagnostic helpers. */

/* netdev_printk helpers, similar to dev_printk */

static inline const char *netdev_name(const struct net_device *dev)
{
        if (!dev->name[0] || strchr(dev->name, '%'))
                return "(unnamed net_device)";
        return dev->name;
}

static inline const char *netdev_reg_state(const struct net_device *dev)
{
        switch (dev->reg_state) {
        case NETREG_UNINITIALIZED: return " (uninitialized)";
        case NETREG_REGISTERED: return "";
        case NETREG_UNREGISTERING: return " (unregistering)";
        case NETREG_UNREGISTERED: return " (unregistered)";
        case NETREG_RELEASED: return " (released)";
        case NETREG_DUMMY: return " (dummy)";
        }

        WARN_ONCE(1, "%s: unknown reg_state %d\n", dev->name, dev->reg_state);
        return " (unknown)";
}

__printf(3, 4)
void netdev_printk(const char *level, const struct net_device *dev,
                   const char *format, ...);
__printf(2, 3)
void netdev_emerg(const struct net_device *dev, const char *format, ...);
__printf(2, 3)
void netdev_alert(const struct net_device *dev, const char *format, ...);
__printf(2, 3)
void netdev_crit(const struct net_device *dev, const char *format, ...);
__printf(2, 3)
void netdev_err(const struct net_device *dev, const char *format, ...);
__printf(2, 3)
void netdev_warn(const struct net_device *dev, const char *format, ...);
__printf(2, 3)
void netdev_notice(const struct net_device *dev, const char *format, ...);
__printf(2, 3)
void netdev_info(const struct net_device *dev, const char *format, ...);

#define MODULE_ALIAS_NETDEV(device) \
        MODULE_ALIAS("netdev-" device)

#if defined(CONFIG_DYNAMIC_DEBUG)
#define netdev_dbg(__dev, format, args...)                      \
do {                                                            \
        dynamic_netdev_dbg(__dev, format, ##args);              \
} while (0)
#elif defined(DEBUG)
#define netdev_dbg(__dev, format, args...)                      \
        netdev_printk(KERN_DEBUG, __dev, format, ##args)
#else
#define netdev_dbg(__dev, format, args...)                      \
({                                                              \
        if (0)                                                  \
                netdev_printk(KERN_DEBUG, __dev, format, ##args); \
})
#endif

#if defined(VERBOSE_DEBUG)
#define netdev_vdbg     netdev_dbg
#else

#define netdev_vdbg(dev, format, args...)                       \
({                                                              \
        if (0)                                                  \
                netdev_printk(KERN_DEBUG, dev, format, ##args); \
        0;                                                      \
})
#endif

/*
 * netdev_WARN() acts like dev_printk(), but with the key difference
 * of using a WARN/WARN_ON to get the message out, including the
 * file/line information and a backtrace.
 */
#define netdev_WARN(dev, format, args...)                       \
        WARN(1, "netdevice: %s%s\n" format, netdev_name(dev),   \
             netdev_reg_state(dev), ##args)

/* netif printk helpers, similar to netdev_printk */

#define netif_printk(priv, type, level, dev, fmt, args...)      \
do {                                                            \
        if (netif_msg_##type(priv))                             \
                netdev_printk(level, (dev), fmt, ##args);       \
} while (0)

#define netif_level(level, priv, type, dev, fmt, args...)       \
do {                                                            \
        if (netif_msg_##type(priv))                             \
                netdev_##level(dev, fmt, ##args);               \
} while (0)

#define netif_emerg(priv, type, dev, fmt, args...)              \
        netif_level(emerg, priv, type, dev, fmt, ##args)
#define netif_alert(priv, type, dev, fmt, args...)              \
        netif_level(alert, priv, type, dev, fmt, ##args)
#define netif_crit(priv, type, dev, fmt, args...)               \
        netif_level(crit, priv, type, dev, fmt, ##args)
#define netif_err(priv, type, dev, fmt, args...)                \
        netif_level(err, priv, type, dev, fmt, ##args)
#define netif_warn(priv, type, dev, fmt, args...)               \
        netif_level(warn, priv, type, dev, fmt, ##args)
#define netif_notice(priv, type, dev, fmt, args...)             \
        netif_level(notice, priv, type, dev, fmt, ##args)
#define netif_info(priv, type, dev, fmt, args...)               \
        netif_level(info, priv, type, dev, fmt, ##args)

#if defined(CONFIG_DYNAMIC_DEBUG)
#define netif_dbg(priv, type, netdev, format, args...)          \
do {                                                            \
        if (netif_msg_##type(priv))                             \
                dynamic_netdev_dbg(netdev, format, ##args);     \
} while (0)
#elif defined(DEBUG)
#define netif_dbg(priv, type, dev, format, args...)             \
        netif_printk(priv, type, KERN_DEBUG, dev, format, ##args)
#else
#define netif_dbg(priv, type, dev, format, args...)                     \
({                                                                      \
        if (0)                                                          \
                netif_printk(priv, type, KERN_DEBUG, dev, format, ##args); \
        0;                                                              \
})
#endif

```

```
#if defined(VERBOSE_DEBUG)
#define netif_vdbg      netif_dbg
#else
#define netif_vdbg(priv, type, dev, format, args...)            \
({                                                              \
        if (0)                                                  \
                netif_printk(priv, type, KERN_DEBUG, dev, format, ##args); \
        0;                                                      \
})
#endif

/*
 *      The list of packet types we will receive (as opposed to discard)
 *      and the routines to invoke.
 *
 *      Why 16. Because with 16 the only overlap we get on a hash of the
 *      low nibble of the protocol value is RARP/SNAP/X.25.
 *
 *      NOTE:  That is no longer true with the addition of VLAN tags.  Not
 *             sure which should go first, but I bet it won't make much
 *             difference if we are running VLANs.  The good news is that
 *             this protocol won't be in the list unless compiled in, so
 *             the average user (w/out VLANs) will not be adversely affected.
 *             --BLG
 *
 *              0800    IP
 *              8100    802.1Q VLAN
 *              0001    802.3
 *              0002    AX.25
 *              0004    802.2
 *              8035    RARP
 *              0005    SNAP
 *              0805    X.25
 *              0806    ARP
 *              8137    IPX
 *              0009    Localtalk
 *              86DD    IPv6
 */
#define PTYPE_HASH_SIZE (16)
#define PTYPE_HASH_MASK (PTYPE_HASH_SIZE - 1)

#endif  /* _LINUX_NETDEVICE_H */
```

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

- Home
- Development
- Services
- Training
- Docs
- Community
- Company
- Blog