

# **Linux Networking and Network Devices APIs**

## **Linux Networking and Network Devices APIs**

This documentation 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.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

For more details see the file COPYING in the source distribution of Linux.

# Table of Contents

<b>1. Linux Networking .....</b>	<b>1</b>
1.1. Networking Base Types .....	1
enum sock_type .....	1
struct socket .....	2
1.2. Socket Buffer Functions.....	3
struct skb_shared_hwtstamps.....	3
struct sk_buff.....	5
skb_dst .....	7
skb_dst_set.....	8
skb_dst_is_noref .....	9
skb_queue_empty .....	10
skb_queue_is_last .....	10
skb_queue_is_first.....	11
skb_queue_next.....	12
skb_queue_prev .....	13
skb_get .....	14
skb_cloned .....	15
skb_header_cloned.....	15
skb_header_release .....	16
skb_shared.....	17
skb_share_check .....	18
skb_unshare.....	19
skb_peek .....	20
skb_peek_next.....	21
skb_peek_tail .....	22
skb_queue_len.....	22
__skb_queue_head_init.....	23
skb_queue_splice .....	24
skb_queue_splice_init.....	25
skb_queue_splice_tail.....	26
skb_queue_splice_tail_init.....	26
__skb_queue_after .....	27
__skb_fill_page_desc .....	28
skb_fill_page_desc.....	29
skb_headroom.....	31
skb_tailroom .....	31
skb_availroom.....	32
skb_reserve .....	33
pskb_trim_unique .....	34
skb_orphan.....	35
__dev_alloc_skb .....	36

netdev_alloc_skb.....	37
skb_frag_page.....	38
__skb_frag_ref.....	38
skb_frag_ref.....	39
__skb_frag_unref.....	40
skb_frag_unref.....	41
skb_frag_address.....	42
skb_frag_address_safe.....	42
__skb_frag_set_page.....	43
skb_frag_set_page.....	44
skb_frag_dma_map.....	45
skb_clone_writable.....	46
skb_cow.....	47
skb_cow_head.....	48
skb_padto.....	49
skb_linearize.....	50
skb_linearize_cow.....	51
skb_postpull_rcsum.....	52
pskb_trim_rcsum.....	53
skb_get_timestamp.....	54
skb_complete_tx_timestamp.....	55
skb_tx_timestamp.....	56
skb_checksum_complete.....	56
skb_checksum_none_assert.....	57
struct sock_common.....	58
struct sock.....	60
unlock_sock_fast.....	67
sk_filter_release.....	68
sk_wmem_alloc_get.....	69
sk_rmem_alloc_get.....	69
sk_has_allocations.....	70
wq_has_sleeper.....	71
sock_poll_wait.....	72
sk_eat_skb.....	73
sockfd_lookup.....	74
sock_release.....	75
kernel_recvmsg.....	76
sock_register.....	77
sock_unregister.....	78
__alloc_skb.....	79
build_skb.....	80
__netdev_alloc_skb.....	81
dev_alloc_skb.....	82

__kfree_skb.....	83
kfree_skb.....	84
consume_skb.....	85
skb_recycle .....	85
skb_recycle_check .....	86
skb_morph.....	87
skb_clone .....	88
skb_copy .....	89
__pskb_copy .....	90
pskb_expand_head.....	91
skb_copy_expand.....	93
skb_pad .....	94
skb_put.....	95
skb_push .....	96
skb_pull.....	97
skb_trim .....	97
__pskb_pull_tail.....	98
skb_copy_bits .....	99
skb_store_bits .....	101
skb_dequeue.....	102
skb_dequeue_tail.....	102
skb_queue_purge .....	103
skb_queue_head.....	104
skb_queue_tail .....	105
skb_unlink.....	106
skb_append .....	107
skb_insert.....	108
skb_split.....	109
skb_prepare_seq_read.....	110
skb_seq_read.....	111
skb_abort_seq_read.....	112
skb_find_text.....	113
skb_append_datato_frags.....	114
skb_pull_rcsum.....	115
skb_segment.....	116
skb_cow_data.....	117
skb_partial_csum_set.....	118
sk_alloc .....	119
sk_clone_lock .....	120
sk_wait_data .....	121
__sk_mem_schedule.....	122
__sk_mem_reclaim.....	123
lock_sock_fast.....	124

__skb_recv_datagram .....	124
skb_kill_datagram.....	126
skb_copy_datagram_iovec .....	127
skb_copy_datagram_const_iovec .....	128
skb_copy_datagram_from_iovec .....	129
skb_copy_and_csum_datagram_iovec.....	131
datagram_poll .....	132
sk_stream_write_space .....	133
sk_stream_wait_connect.....	134
sk_stream_wait_memory .....	134
1.3. Socket Filter .....	135
sk_filter .....	135
sk_run_filter .....	136
sk_chk_filter.....	137
sk_filter_release_rcu .....	138
sk_attach_filter .....	139
1.4. Generic Network Statistics .....	140
struct gnet_stats_basic .....	140
struct gnet_stats_rate_est .....	141
struct gnet_stats_queue .....	141
struct gnet_estimator.....	142
gnet_stats_start_copy_compat .....	143
gnet_stats_start_copy .....	144
gnet_stats_copy_basic.....	146
gnet_stats_copy_rate_est .....	146
gnet_stats_copy_queue .....	148
gnet_stats_copy_app .....	148
gnet_stats_finish_copy .....	150
gen_new_estimator .....	150
gen_kill_estimator.....	152
gen_replace_estimator .....	153
gen_estimator_active .....	154
1.5. SUN RPC subsystem .....	155
xdr_encode_opaque_fixed .....	155
xdr_encode_opaque .....	156
xdr_terminate_string.....	157
xdr_init_encode.....	157
xdr_reserve_space.....	158
xdr_write_pages.....	159
xdr_init_decode.....	160
xdr_init_decode_pages .....	161
xdr_set_scratch_buffer.....	162
xdr_inline_decode.....	163

xdr_read_pages .....	164
xdr_enter_page.....	165
svc_print_addr.....	166
svc_reserve.....	167
svc_find_xprt.....	168
svc_xprt_names.....	169
xprt_register_transport.....	170
xprt_unregister_transport.....	171
xprt_load_transport.....	172
xprt_reserve_xprt .....	173
xprt_release_xprt.....	174
xprt_release_xprt_cong.....	175
xprt_release_rqst_cong .....	175
xprt_adjust_cwnd.....	176
xprt_wake_pending_tasks.....	177
xprt_wait_for_buffer_space .....	178
xprt_write_space .....	178
xprt_set_retrans_timeout_def .....	179
xprt_disconnect_done .....	180
xprt_lookup_rqst.....	181
xprt_complete_rqst .....	181
rpc_wake_up .....	182
rpc_wake_up_status .....	183
rpc_wake_up_softconn_status .....	184
rpc_malloc.....	185
rpc_free .....	186
xdr_skb_read_bits.....	187
xdr_partial_copy_from_skb.....	188
csum_partial_copy_to_xdr.....	189
rpc_alloc_iostats .....	189
rpc_free_iostats .....	190
rpc_count_iostats .....	191
rpc_queue_upcall .....	192
rpc_mkpipe_dentry .....	193
rpc_unlink .....	194
rpcb_getport_async .....	195
rpc_bind_new_program .....	195
rpc_run_task.....	196
rpc_call_sync .....	197
rpc_call_async.....	198
rpc_peeraddr .....	199
rpc_peeraddr2str .....	200
rpc_localaddr.....	201

rpc_protocol .....	202
rpc_net_ns .....	203
rpc_max_payload .....	204
rpc_force_rebind .....	204
1.6. WiMAX .....	205
wimax_msg_alloc .....	205
wimax_msg_data_len .....	207
wimax_msg_data .....	208
wimax_msg_len .....	208
wimax_msg_send .....	209
wimax_msg .....	210
wimax_reset .....	212
wimax_report_rfkill_hw .....	213
wimax_report_rfkill_sw .....	214
wimax_rfkill .....	215
wimax_state_change .....	216
wimax_state_get .....	217
wimax_dev_init .....	218
wimax_dev_add .....	219
wimax_dev_rm .....	220
struct wimax_dev .....	221
enum wimax_st .....	225
<b>2. Network device support .....</b>	<b>229</b>
2.1. Driver Support .....	229
dev_add_pack .....	229
__dev_remove_pack .....	229
dev_remove_pack .....	230
netdev_boot_setup_check .....	231
__dev_get_by_name .....	232
dev_get_by_name_rcu .....	233
dev_get_by_name .....	234
__dev_get_by_index .....	235
dev_get_by_index_rcu .....	236
dev_get_by_index .....	237
dev_getbyhwaddr_rcu .....	238
dev_get_by_flags_rcu .....	239
dev_valid_name .....	240
dev_alloc_name .....	241
netdev_features_change .....	242
netdev_state_change .....	242
dev_load .....	243
dev_open .....	244
dev_close .....	245



dev_disable_lro .....	246
register_netdevice_notifier .....	247
unregister_netdevice_notifier .....	248
call_netdevice_notifiers .....	249
dev_forward_skb .....	250
netif_set_real_num_rx_queues .....	251
netif_device_detach .....	251
netif_device_attach .....	252
skb_gso_segment .....	253
dev_queue_xmit .....	254
rps_may_expire_flow .....	255
netif_rx .....	256
netdev_rx_handler_register .....	257
netdev_rx_handler_unregister .....	258
netif_receive_skb .....	259
__napi_schedule .....	260
register_gifconf .....	261
netdev_set_master .....	262
netdev_set_bond_master .....	263
dev_set_promiscuity .....	264
dev_set_allmulti .....	265
dev_get_flags .....	266
dev_change_flags .....	266
dev_set_mtu .....	267
dev_set_group .....	268
dev_set_mac_address .....	269
netdev_update_features .....	270
netdev_change_features .....	271
netif_stacked_transfer_operstate .....	272
register_netdevice .....	272
init_dummy_netdev .....	273
register_netdev .....	274
dev_get_stats .....	275
alloc_netdev_mqs .....	276
free_netdev .....	277
synchronize_net .....	278
unregister_netdevice_queue .....	279
unregister_netdevice_many .....	280
unregister_netdev .....	281
dev_change_net_namespace .....	282
netdev_increment_features .....	283
eth_header .....	284
eth_rebuild_header .....	285

eth_type_trans .....	286
eth_header_parse.....	287
eth_header_cache .....	287
eth_header_cache_update .....	288
eth_mac_addr .....	289
eth_change_mtu .....	290
ether_setup .....	291
alloc_etherdev_mqs .....	292
netif_carrier_on.....	293
netif_carrier_off .....	293
netif_notify_peers .....	294
is_zero_ether_addr.....	295
is_multicast_ether_addr .....	296
is_local_ether_addr.....	297
is_broadcast_ether_addr .....	297
is_unicast_ether_addr .....	298
is_valid_ether_addr.....	299
random_ether_addr .....	300
eth_hw_addr_random .....	301
compare_ether_addr.....	301
compare_ether_addr_64bits.....	302
is_etherdev_addr .....	303
compare_ether_header .....	304
napi_schedule_prep.....	305
napi_schedule.....	306
napi_disable .....	307
napi_enable .....	308
napi_synchronize .....	309
netdev_priv.....	309
netif_start_queue.....	310
netif_wake_queue .....	311
netif_stop_queue .....	312
netif_queue_stopped .....	313
netif_running.....	313
netif_start_subqueue .....	314
netif_stop_subqueue .....	315
__netif_subqueue_stopped.....	316
netif_wake_subqueue.....	317
netif_is_multiqueue.....	318
dev_put.....	319
dev_hold.....	319
netif_carrier_ok.....	320
netif_dormant_on .....	321

netif_dormant_off .....	322
netif_dormant .....	322
netif_oper_up .....	323
netif_device_present .....	324
netif_tx_lock .....	325
2.2. PHY Support .....	326
phy_print_status .....	326
phy_ethtool_sset .....	326
phy_mii_ioctl .....	327
phy_start_aneg .....	328
phy_start_interrupts .....	329
phy_stop_interrupts .....	330
phy_stop .....	331
phy_start .....	331
phy_clear_interrupt .....	332
phy_config_interrupt .....	333
phy_aneg_done .....	334
phy_find_setting .....	335
phy_find_valid .....	336
phy_sanitize_settings .....	336
phy_start_machine .....	337
phy_stop_machine .....	338
phy_force_reduction .....	339
phy_error .....	340
phy_interrupt .....	341
phy_enable_interrupts .....	342
phy_disable_interrupts .....	342
phy_change .....	343
phy_state_machine .....	344
get_phy_id .....	344
get_phy_device .....	345
phy_device_register .....	346
phy_find_first .....	347
phy_connect_direct .....	348
phy_connect .....	349
phy_disconnect .....	350
phy_attach .....	351
phy_detach .....	352
genphy_restart_aneg .....	352
genphy_config_aneg .....	353
genphy_update_link .....	354
genphy_read_status .....	355
phy_driver_register .....	356

phy_prepare_link .....	356
phy_attach_direct .....	357
genphy_config_advert .....	358
genphy_setup_forced .....	359
phy_probe .....	360
mdiobus_alloc_size .....	361
mdiobus_register .....	362
mdiobus_free .....	363
mdiobus_read .....	363
mdiobus_write .....	364
mdiobus_release .....	365
mdio_bus_match .....	366

# Chapter 1. Linux Networking

## 1.1. Networking Base Types

### enum sock\_type

**LINUX**

Kernel Hackers Manual January 2013

#### Name

enum sock\_type — Socket types

#### Synopsis

```
enum sock_type {  
    SOCK_STREAM,  
    SOCK_DGRAM,  
    SOCK_RAW,  
    SOCK_RDM,  
    SOCK_SEQPACKET,  
    SOCK_DCCP,  
    SOCK_PACKET  
};
```

#### Constants

SOCK\_STREAM

stream (connection) socket

SOCK\_DGRAM

datagram (conn.less) socket

SOCK\_RAW

raw socket

## SOCK\_RDM

reliably-delivered message

## SOCK\_SEQPACKET

sequential packet socket

## SOCK\_DCCP

Datagram Congestion Control Protocol socket

## SOCK\_PACKET

linux specific way of getting packets at the dev level. For writing rarp and other similar things on the user level.

## Description

When adding some new socket type please grep ARCH\_HAS\_SOCKET\_TYPE include/asm-\*/socket.h, at least MIPS overrides this enum for binary compat reasons.

# struct socket

## LINUX

Kernel Hackers Manual January 2013

## Name

`struct socket` — general BSD socket

## Synopsis

```
struct socket {
    socket_state state;
    short type;
    unsigned long flags;
    struct socket_wq __rcu * wq;
    struct file * file;
```

```
struct sock * sk;  
const struct proto_ops * ops;  
};
```

## Members

state

socket state (SS\_CONNECTED, etc)

type

socket type (SOCK\_STREAM, etc)

flags

socket flags (SOCK\_ASYNC\_NOSPACE, etc)

wq

wait queue for several uses

file

File back pointer for gc

sk

internal networking protocol agnostic socket representation

ops

protocol specific socket operations

## 1.2. Socket Buffer Functions

### struct skb\_shared\_hwtstamps

**LINUX**

## Name

`struct skb_shared_hwtstamps` — hardware time stamps

## Synopsis

```
struct skb_shared_hwtstamps {
    ktime_t hwtstamp;
    ktime_t syststamp;
};
```

## Members

`hwtstamp`

hardware time stamp transformed into duration since arbitrary point in time

`syststamp`

`hwtstamp` transformed to system time base

## Description

Software time stamps generated by `ktime_get_real` are stored in `skb->tstamp`. The relation between the different kinds of time

## stamps is as follows

`syststamp` and `tstamp` can be compared against each other in arbitrary combinations. The accuracy of a `syststamp/tstamp` “`syststamp` from other device” comparison is limited by the accuracy of the transformation into system time base. This depends on the device driver and its underlying hardware.

`hwtstamps` can only be compared against other `hwtstamps` from the same device.

This structure is attached to packets as part of the `skb_shared_info`. Use `skb_hwtstamps` to get a pointer.



# struct sk\_buff

## LINUX

Kernel Hackers Manual January 2013

## Name

struct sk\_buff — socket buffer

## Synopsis

```
struct sk_buff {
    struct sk_buff * next;
    struct sk_buff * prev;
    ktime_t tstamp;
    struct sock * sk;
    struct net_device * dev;
    char cb[48];
    unsigned long _skb_refdst;
#ifdef CONFIG_XFRM
    struct sec_path * sp;
#endif
    unsigned int len;
    unsigned int data_len;
    __u16 mac_len;
    __u16 hdr_len;
    union {unnamed_union};
    sk_buff_data_t transport_header;
    sk_buff_data_t network_header;
    sk_buff_data_t mac_header;
    sk_buff_data_t tail;
    sk_buff_data_t end;
    unsigned char * head;
    unsigned char * data;
    unsigned int truesize;
    atomic_t users;
};
```

## **Members**

next

Next buffer in list

prev

Previous buffer in list

tstamp

Time we arrived

sk

Socket we are owned by

dev

Device we arrived on/are leaving by

cb[48]

Control buffer. Free for use by every layer. Put private vars here

\_skb\_refdst

destination entry (with norefcnt bit)

sp

the security path, used for xfrm

len

Length of actual data

data\_len

Data length

mac\_len

Length of link layer header

hdr\_len

writable header length of cloned skb

{unnamed\_union}

anonymous

transport\_header  
    Transport layer header

network\_header  
    Network layer header

mac\_header  
    Link layer header

tail  
    Tail pointer

end  
    End pointer

head  
    Head of buffer

data  
    Data head pointer

truesize  
    Buffer size

users  
    User count - see {datagram,tcp}.c

## **skb\_dst**

### **LINUX**

Kernel Hackers Manual January 2013

### **Name**

skb\_dst — returns skb dst\_entry

## Synopsis

```
struct dst_entry * skb_dst (const struct sk_buff * skb);
```

## Arguments

*skb*

buffer

## Description

Returns *skb* *dst\_entry*, regardless of reference taken or not.

## skb\_dst\_set

### LINUX

Kernel Hackers Manual January 2013

## Name

*skb\_dst\_set* — sets *skb* *dst*

## Synopsis

```
void skb_dst_set (struct sk_buff * skb, struct dst_entry *  
dst);
```

## Arguments

*skb*

buffer

*dst*

dst entry

## Description

Sets *skb* *dst*, assuming a reference was taken on *dst* and should be released by `skb_dst_drop`

# skb\_dst\_is\_noref

## LINUX

Kernel Hackers Manual January 2013

## Name

`skb_dst_is_noref` — Test if *skb* *dst* isn't refcounted

## Synopsis

```
bool skb_dst_is_noref (const struct sk_buff * skb);
```

## Arguments

*skb*

buffer

# skb\_queue\_empty

## LINUX

Kernel Hackers Manual January 2013

### Name

`skb_queue_empty` — check if a queue is empty

### Synopsis

```
int skb_queue_empty (const struct sk_buff_head * list);
```

### Arguments

*list*

queue head

### Description

Returns true if the queue is empty, false otherwise.

# skb\_queue\_is\_last

## LINUX

## Name

`skb_queue_is_last` — check if `skb` is the last entry in the queue

## Synopsis

```
bool skb_queue_is_last (const struct sk_buff_head * list,  
const struct sk_buff * skb);
```

## Arguments

*list*

queue head

*skb*

buffer

## Description

Returns true if *skb* is the last buffer on the list.

# skb\_queue\_is\_first

## LINUX

## Name

`skb_queue_is_first` — check if `skb` is the first entry in the queue

## Synopsis

```
bool skb_queue_is_first (const struct sk_buff_head * list,  
const struct sk_buff * skb);
```

## Arguments

*list*

queue head

*skb*

buffer

## Description

Returns true if *skb* is the first buffer on the list.

## skb\_queue\_next

### LINUX

Kernel Hackers Manual January 2013

## Name

`skb_queue_next` — return the next packet in the queue

## Synopsis

```
struct sk_buff * skb_queue_next (const struct sk_buff_head *  
list, const struct sk_buff * skb);
```



## Arguments

*list*

queue head

*skb*

current buffer

## Description

Return the next packet in *list* after *skb*. It is only valid to call this if `skb_queue_is_last` evaluates to false.

# skb\_queue\_prev

## LINUX

Kernel Hackers Manual January 2013

## Name

`skb_queue_prev` — return the prev packet in the queue

## Synopsis

```
struct sk_buff * skb_queue_prev (const struct sk_buff_head *  
list, const struct sk_buff * skb);
```

## Arguments

*list*

queue head

*skb*

current buffer

## Description

Return the prev packet in *list* before *skb*. It is only valid to call this if `skb_queue_is_first` evaluates to false.

# skb\_get

## LINUX

Kernel Hackers Manual January 2013

## Name

`skb_get` — reference buffer

## Synopsis

```
struct sk_buff * skb_get (struct sk_buff * skb);
```

## Arguments

*skb*

buffer to reference

## Description

Makes another reference to a socket buffer and returns a pointer to the buffer.

# skb\_cloned

## LINUX

Kernel Hackers Manual January 2013

## Name

`skb_cloned` — is the buffer a clone

## Synopsis

```
int skb_cloned (const struct sk_buff * skb);
```

## Arguments

*skb*

buffer to check

## Description

Returns true if the buffer was generated with `skb_clone` and is one of multiple shared copies of the buffer. Cloned buffers are shared data so must not be written to under normal circumstances.

# skb\_header\_cloned

## LINUX

Kernel Hackers Manual January 2013

### Name

`skb_header_cloned` — is the header a clone

### Synopsis

```
int skb_header_cloned (const struct sk_buff * skb);
```

### Arguments

*skb*

buffer to check

### Description

Returns true if modifying the header part of the buffer requires the data to be copied.

# skb\_header\_release

## LINUX

Kernel Hackers Manual January 2013

### Name

`skb_header_release` — release reference to header

## Synopsis

```
void skb_header_release (struct sk_buff * skb);
```

## Arguments

*skb*

buffer to operate on

## Description

Drop a reference to the header part of the buffer. This is done by acquiring a payload reference. You must not read from the header part of `skb->data` after this.

## skb\_shared

### LINUX

Kernel Hackers Manual January 2013

## Name

`skb_shared` — is the buffer shared

## Synopsis

```
int skb_shared (const struct sk_buff * skb);
```

## Arguments

*skb*

buffer to check

## Description

Returns true if more than one person has a reference to this buffer.

# skb\_share\_check

## LINUX

Kernel Hackers Manual January 2013

## Name

`skb_share_check` — check if buffer is shared and if so clone it

## Synopsis

```
struct sk_buff * skb_share_check (struct sk_buff * skb, gfp_t  
pri);
```

## Arguments

*skb*

buffer to check

*pri*

priority for memory allocation

## Description

If the buffer is shared the buffer is cloned and the old copy drops a reference. A new clone with a single reference is returned. If the buffer is not shared the original buffer is returned. When being called from interrupt status or with spinlocks held `pri` must be `GFP_ATOMIC`.

`NULL` is returned on a memory allocation failure.

## skb\_unshare

### LINUX

Kernel Hackers Manual January 2013

## Name

`skb_unshare` — make a copy of a shared buffer

## Synopsis

```
struct sk_buff * skb_unshare (struct sk_buff * skb, gfp_t
pri);
```

## Arguments

*skb*

buffer to check

*pri*

priority for memory allocation

## Description

If the socket buffer is a clone then this function creates a new copy of the data, drops a reference count on the old copy and returns the new copy with the reference count at 1. If the buffer is not a clone the original buffer is returned. When called with a spinlock held or from interrupt state *pri* must be `GFP_ATOMIC`

`NULL` is returned on a memory allocation failure.

## skb\_peek

### LINUX

Kernel Hackers Manual January 2013

## Name

`skb_peek` — peek at the head of an `sk_buff_head`

## Synopsis

```
struct sk_buff * skb_peek (const struct sk_buff_head * list_);
```

## Arguments

*list\_*

list to peek at

## Description

Peek an `sk_buff`. Unlike most other operations you MUST be careful with this one. A peek leaves the buffer on the list and someone else may run off with it. You must hold the appropriate locks or have a private queue to do this.



Returns `NULL` for an empty list or a pointer to the head element. The reference count is not incremented and the reference is therefore volatile. Use with caution.

## skb\_peek\_next

### LINUX

Kernel Hackers Manual January 2013

### Name

`skb_peek_next` — peek skb following the given one from a queue

### Synopsis

```
struct sk_buff * skb_peek_next (struct sk_buff * skb, const  
struct sk_buff_head * list_);
```

### Arguments

*skb*

skb to start from

*list\_*

list to peek at

### Description

Returns `NULL` when the end of the list is met or a pointer to the next element. The reference count is not incremented and the reference is therefore volatile. Use with caution.

# skb\_peek\_tail

## LINUX

Kernel Hackers Manual January 2013

### Name

`skb_peek_tail` — peek at the tail of an `sk_buff_head`

### Synopsis

```
struct sk_buff * skb_peek_tail (const struct sk_buff_head *  
list_);
```

### Arguments

`list_`

list to peek at

### Description

Peek an `sk_buff`. Unlike most other operations you MUST be careful with this one. A peek leaves the buffer on the list and someone else may run off with it. You must hold the appropriate locks or have a private queue to do this.

Returns `NULL` for an empty list or a pointer to the tail element. The reference count is not incremented and the reference is therefore volatile. Use with caution.

# skb\_queue\_len

## LINUX

Kernel Hackers Manual January 2013

### Name

skb\_queue\_len — get queue length

### Synopsis

```
__u32 skb_queue_len (const struct sk_buff_head * list_);
```

### Arguments

*list\_*

list to measure

### Description

Return the length of an sk\_buff queue.

# \_\_skb\_queue\_head\_init

## LINUX

Kernel Hackers Manual January 2013

### Name

\_\_skb\_queue\_head\_init — initialize non-spinlock portions of sk\_buff\_head

## Synopsis

```
void __skb_queue_head_init (struct sk_buff_head * list);
```

## Arguments

*list*

queue to initialize

## Description

This initializes only the list and queue length aspects of an `sk_buff_head` object. This allows to initialize the list aspects of an `sk_buff_head` without reinitializing things like the spinlock. It can also be used for on-stack `sk_buff_head` objects where the spinlock is known to not be used.

# skb\_queue\_splice

## LINUX

Kernel Hackers Manual January 2013

## Name

`skb_queue_splice` — join two skb lists, this is designed for stacks

## Synopsis

```
void skb_queue_splice (const struct sk_buff_head * list,  
struct sk_buff_head * head);
```

## Arguments

*list*

the new list to add

*head*

the place to add it in the first list

# skb\_queue\_splice\_init

## LINUX

Kernel Hackers Manual January 2013

## Name

`skb_queue_splice_init` — join two skb lists and reinitialise the emptied list

## Synopsis

```
void skb_queue_splice_init (struct sk_buff_head * list, struct  
sk_buff_head * head);
```

## Arguments

*list*

the new list to add

*head*

the place to add it in the first list

## Description

The list at *list* is reinitialised

# skb\_queue\_splice\_tail

## LINUX

Kernel Hackers Manual January 2013

## Name

`skb_queue_splice_tail` — join two skb lists, each list being a queue

## Synopsis

```
void skb_queue_splice_tail (const struct sk_buff_head * list,  
struct sk_buff_head * head);
```

## Arguments

*list*

the new list to add

*head*

the place to add it in the first list

# skb\_queue\_splice\_tail\_init

## LINUX

## Name

`skb_queue_splice_tail_init` — join two skb lists and reinitialise the emptied list

## Synopsis

```
void skb_queue_splice_tail_init (struct sk_buff_head * list,  
struct sk_buff_head * head);
```

## Arguments

*list*

the new list to add

*head*

the place to add it in the first list

## Description

Each of the lists is a queue. The list at *list* is reinitialised

**\_\_skb\_queue\_after**

**LINUX**

## Name

`__skb_queue_after` — queue a buffer at the list head

## Synopsis

```
void __skb_queue_after (struct sk_buff_head * list, struct  
sk_buff * prev, struct sk_buff * newsk);
```

## Arguments

*list*

list to use

*prev*

place after this buffer

*newsk*

buffer to queue

## Description

Queue a buffer into the middle of a list. This function takes no locks and you must therefore hold required locks before calling it.

A buffer cannot be placed on two lists at the same time.

# `__skb_fill_page_desc`

**LINUX**



## Name

`__skb_fill_page_desc` — initialise a paged fragment in an `skb`

## Synopsis

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

## Arguments

*skb*

buffer containing fragment to be initialised

*i*

paged fragment index to initialise

*page*

the page to use for this fragment

*off*

the offset to the data with *page*

*size*

the length of the data

## Description

Initialises the *i*'th fragment of *skb* to point to *size* bytes at offset *off* within *page*.

Does not take any additional reference on the fragment.

# skb\_fill\_page\_desc

## LINUX

Kernel Hackers Manual January 2013

### Name

`skb_fill_page_desc` — initialise a paged fragment in an `skb`

### Synopsis

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

### Arguments

*skb*

buffer containing fragment to be initialised

*i*

paged fragment index to initialise

*page*

the page to use for this fragment

*off*

the offset to the data with *page*

*size*

the length of the data

## Description

As per `__skb_fill_page_desc` -- initialises the *i*'th fragment of *skb* to point to size bytes at offset *off* within *page*. In addition updates *skb* such that *i* is the last fragment.

Does not take any additional reference on the fragment.

# skb\_headroom

## LINUX

Kernel Hackers Manual January 2013

## Name

`skb_headroom` — bytes at buffer head

## Synopsis

```
unsigned int skb_headroom (const struct sk_buff * skb);
```

## Arguments

*skb*

buffer to check

## Description

Return the number of bytes of free space at the head of an `sk_buff`.

# skb\_tailroom

## LINUX

Kernel Hackers Manual January 2013

### Name

`skb_tailroom` — bytes at buffer end

### Synopsis

```
int skb_tailroom (const struct sk_buff * skb);
```

### Arguments

*skb*

buffer to check

### Description

Return the number of bytes of free space at the tail of an `sk_buff`

# skb\_availroom

## LINUX

Kernel Hackers Manual January 2013

### Name

`skb_availroom` — bytes at buffer end

## Synopsis

```
int skb_availroom (const struct sk_buff * skb);
```

## Arguments

*skb*

buffer to check

## Description

Return the number of bytes of free space at the tail of an `sk_buff` allocated by `sk_stream_alloc`

# skb\_reserve

## LINUX

Kernel Hackers Manual January 2013

## Name

`skb_reserve` — adjust headroom

## Synopsis

```
void skb_reserve (struct sk_buff * skb, int len);
```

## Arguments

*skb*

buffer to alter

*len*

bytes to move

## Description

Increase the headroom of an empty `sk_buff` by reducing the tail room. This is only allowed for an empty buffer.

# pskb\_trim\_unique

## LINUX

Kernel Hackers Manual January 2013

## Name

`pskb_trim_unique` — remove end from a paged unique (not cloned) buffer

## Synopsis

```
void pskb_trim_unique (struct sk_buff * skb, unsigned int  
len);
```

## Arguments

*skb*

buffer to alter

*len*

new length

## Description

This is identical to `pskb_trim` except that the caller knows that the `skb` is not cloned so we should never get an error due to out- of-memory.

# skb\_orphan

## LINUX

Kernel Hackers Manual January 2013

## Name

`skb_orphan` — orphan a buffer

## Synopsis

```
void skb_orphan (struct sk_buff * skb);
```

## Arguments

*skb*

buffer to orphan

## Description

If a buffer currently has an owner then we call the owner's destructor function and make the *skb* unowned. The buffer continues to exist but is no longer charged to its former owner.

## \_\_dev\_alloc\_skb

### LINUX

Kernel Hackers Manual January 2013

## Name

`__dev_alloc_skb` — allocate an skbuff for receiving

## Synopsis

```
struct sk_buff * __dev_alloc_skb (unsigned int length, gfp_t
gfp_mask);
```

## Arguments

*length*

length to allocate

*gfp\_mask*

get\_free\_pages mask, passed to alloc\_skb

## Description

Allocate a new `sk_buff` and assign it a usage count of one. The buffer has unspecified headroom built in. Users should allocate the headroom they think they



need without accounting for the built in space. The built in space is used for optimisations.

NULL is returned if there is no free memory.

## netdev\_alloc\_skb

### LINUX

Kernel Hackers Manual January 2013

### Name

netdev\_alloc\_skb — allocate an skbuff for rx on a specific device

### Synopsis

```
struct sk_buff * netdev_alloc_skb (struct net_device * dev,  
unsigned int length);
```

### Arguments

*dev*

network device to receive on

*length*

length to allocate

### Description

Allocate a new sk\_buff and assign it a usage count of one. The buffer has unspecified headroom built in. Users should allocate the headroom they think they

need without accounting for the built in space. The built in space is used for optimisations.

NULL is returned if there is no free memory. Although this function allocates memory it can be called from an interrupt.

## skb\_frag\_page

### LINUX

Kernel Hackers Manual January 2013

### Name

`skb_frag_page` — retrieve the page referred to by a paged fragment

### Synopsis

```
struct page * skb_frag_page (const skb_frag_t * frag);
```

### Arguments

*frag*

the paged fragment

### Description

Returns the struct page associated with *frag*.

# \_\_skb\_frag\_ref

## LINUX

Kernel Hackers Manual January 2013

### Name

`__skb_frag_ref` — take an addition reference on a paged fragment.

### Synopsis

```
void __skb_frag_ref (skb_frag_t * frag);
```

### Arguments

*frag*

the paged fragment

### Description

Takes an additional reference on the paged fragment *frag*.

# skb\_frag\_ref

## LINUX

Kernel Hackers Manual January 2013

### Name

`skb_frag_ref` — take an addition reference on a paged fragment of an skb.

## Synopsis

```
void skb_frag_ref (struct sk_buff * skb, int f);
```

## Arguments

*skb*

the buffer

*f*

the fragment offset.

## Description

Takes an additional reference on the *f*'th paged fragment of *skb*.

## \_\_skb\_frag\_unref

### LINUX

Kernel Hackers Manual January 2013

## Name

`__skb_frag_unref` — release a reference on a paged fragment.

## Synopsis

```
void __skb_frag_unref (skb_frag_t * frag);
```

## Arguments

*frag*

the paged fragment

## Description

Releases a reference on the paged fragment *frag*.

# skb\_frag\_unref

## LINUX

Kernel Hackers Manual January 2013

## Name

`skb_frag_unref` — release a reference on a paged fragment of an `skb`.

## Synopsis

```
void skb_frag_unref (struct sk_buff * skb, int f);
```

## Arguments

*skb*

the buffer

*f*

the fragment offset

## Description

Releases a reference on the  $f$ 'th paged fragment of *skb*.

# skb\_frag\_address

## LINUX

Kernel Hackers Manual January 2013

## Name

`skb_frag_address` — gets the address of the data contained in a paged fragment

## Synopsis

```
void * skb_frag_address (const skb_frag_t * frag);
```

## Arguments

*frag*

the paged fragment buffer

## Description

Returns the address of the data within *frag*. The page must already be mapped.

# skb\_frag\_address\_safe

## LINUX

Kernel Hackers Manual January 2013

### Name

`skb_frag_address_safe` — gets the address of the data contained in a paged fragment

### Synopsis

```
void * skb_frag_address_safe (const skb_frag_t * frag);
```

### Arguments

*frag*

the paged fragment buffer

### Description

Returns the address of the data within *frag*. Checks that the page is mapped and returns `NULL` otherwise.

# \_\_skb\_frag\_set\_page

## LINUX

## Name

`__skb_frag_set_page` — sets the page contained in a paged fragment

## Synopsis

```
void __skb_frag_set_page (skb_frag_t * frag, struct page *  
page);
```

## Arguments

*frag*

the paged fragment

*page*

the page to set

## Description

Sets the fragment *frag* to contain *page*.

# skb\_frag\_set\_page

## LINUX

## Name

`skb_frag_set_page` — sets the page contained in a paged fragment of an skb



## Synopsis

```
void skb_frag_set_page (struct sk_buff * skb, int f, struct  
page * page);
```

## Arguments

*skb*

the buffer

*f*

the fragment offset

*page*

the page to set

## Description

Sets the *f*'th fragment of *skb* to contain *page*.

# skb\_frag\_dma\_map

**LINUX**

Kernel Hackers Manual January 2013

## Name

`skb_frag_dma_map` — maps a paged fragment via the DMA API

## Synopsis

```
dma_addr_t skb_frag_dma_map (struct device * dev, const
skb_frag_t * frag, size_t offset, size_t size, enum
dma_data_direction dir);
```

## Arguments

*dev*

the device to map the fragment to

*frag*

the paged fragment to map

*offset*

the offset within the fragment (starting at the fragment's own offset)

*size*

the number of bytes to map

*dir*

the direction of the mapping (PCI\_DMA\_\*)

## Description

Maps the page associated with *frag* to *device*.

## skb\_clone\_writable

**LINUX**

## Name

`skb_clone_writable` — is the header of a clone writable

## Synopsis

```
int skb_clone_writable (const struct sk_buff * skb, unsigned  
int len);
```

## Arguments

*skb*

buffer to check

*len*

length up to which to write

## Description

Returns true if modifying the header part of the cloned buffer does not requires the data to be copied.

## skb\_cow

**LINUX**

## Name

`skb_cow` — copy header of `skb` when it is required

## Synopsis

```
int skb_cow (struct sk_buff * skb, unsigned int headroom);
```

## Arguments

*skb*

buffer to cow

*headroom*

needed headroom

## Description

If the `skb` passed lacks sufficient headroom or its data part is shared, data is reallocated. If reallocation fails, an error is returned and original `skb` is not changed.

The result is `skb` with writable area `skb->head...skb->tail` and at least *headroom* of space at head.

## skb\_cow\_head

**LINUX**

## Name

`skb_cow_head` — `skb_cow` but only making the head writable

## Synopsis

```
int skb_cow_head (struct sk_buff * skb, unsigned int  
headroom);
```

## Arguments

*skb*

buffer to cow

*headroom*

needed headroom

## Description

This function is identical to `skb_cow` except that we replace the `skb_cloned` check by `skb_header_cloned`. It should be used when you only need to push on some header and do not need to modify the data.

## `skb_padto`

**LINUX**

## Name

`skb_padto` — pad an skbuff up to a minimal size

## Synopsis

```
int skb_padto (struct sk_buff * skb, unsigned int len);
```

## Arguments

*skb*

buffer to pad

*len*

minimal length

## Description

Pads up a buffer to ensure the trailing bytes exist and are blanked. If the buffer already contains sufficient data it is untouched. Otherwise it is extended. Returns zero on success. The `skb` is freed on error.

## `skb_linearize`

**LINUX**

## Name

`skb_linearize` — convert paged skb to linear one

## Synopsis

```
int skb_linearize (struct sk_buff * skb);
```

## Arguments

*skb*

buffer to linearize

## Description

If there is no free memory -ENOMEM is returned, otherwise zero is returned and the old skb data released.

# skb\_linearize\_cow

## LINUX

## Name

`skb_linearize_cow` — make sure skb is linear and writable

## Synopsis

```
int skb_linearize_cow (struct sk_buff * skb);
```

## Arguments

*skb*

buffer to process

## Description

If there is no free memory -ENOMEM is returned, otherwise zero is returned and the old *skb* data released.

# skb\_postpull\_rcsum

## LINUX

Kernel Hackers Manual January 2013

## Name

`skb_postpull_rcsum` — update checksum for received *skb* after pull

## Synopsis

```
void skb_postpull_rcsum (struct sk_buff * skb, const void *  
start, unsigned int len);
```



## Arguments

*skb*

buffer to update

*start*

start of data before pull

*len*

length of data pulled

## Description

After doing a pull on a received packet, you need to call this to update the CHECKSUM\_COMPLETE checksum, or set ip\_summed to CHECKSUM\_NONE so that it can be recomputed from scratch.

## pskb\_trim\_rcsum

### LINUX

Kernel Hackers Manual January 2013

## Name

`pskb_trim_rcsum` — trim received skb and update checksum

## Synopsis

```
int pskb_trim_rcsum (struct sk_buff * skb, unsigned int len);
```

## Arguments

*skb*

buffer to trim

*len*

new length

## Description

This is exactly the same as `pskb_trim` except that it ensures the checksum of received packets are still valid after the operation.

# skb\_get\_timestamp

## LINUX

Kernel Hackers Manual January 2013

## Name

`skb_get_timestamp` — get timestamp from a `skb`

## Synopsis

```
void skb_get_timestamp (const struct sk_buff * skb, struct  
timeval * stamp);
```

## Arguments

*skb*

skb to get stamp from

*stamp*

pointer to struct timeval to store stamp in

## Description

Timestamps are stored in the skb as offsets to a base timestamp. This function converts the offset back to a struct timeval and stores it in stamp.

# skb\_complete\_tx\_timestamp

## LINUX

Kernel Hackers Manual January 2013

## Name

skb\_complete\_tx\_timestamp — deliver cloned skb with tx timestamps

## Synopsis

```
void skb_complete_tx_timestamp (struct sk_buff * skb, struct
skb_shared_hwtstamps * hwtstamps);
```

## Arguments

*skb*

clone of the the original outgoing packet

*hwtstamps*

hardware time stamps, may be NULL if not available

## Description

PHY drivers may accept clones of transmitted packets for timestamping via their `phy_driver.txtstamp` method. These drivers must call this function to return the `skb` back to the stack, with or without a timestamp.

# skb\_tx\_timestamp

## LINUX

Kernel Hackers Manual January 2013

## Name

`skb_tx_timestamp` — Driver hook for transmit timestamping

## Synopsis

```
void skb_tx_timestamp (struct sk_buff * skb);
```

## Arguments

*skb*

A socket buffer.

## Description

Ethernet MAC Drivers should call this function in their `hard_xmit` function immediately before giving the `sk_buff` to the MAC hardware.

# skb\_checksum\_complete

## LINUX

Kernel Hackers Manual January 2013

### Name

`skb_checksum_complete` — Calculate checksum of an entire packet

### Synopsis

```
__sum16 skb_checksum_complete (struct sk_buff * skb);
```

### Arguments

*skb*

packet to process

### Description

This function calculates the checksum over the entire packet plus the value of `skb->csum`. The latter can be used to supply the checksum of a pseudo header as used by TCP/UDP. It returns the checksum.

For protocols that contain complete checksums such as ICMP/TCP/UDP, this function can be used to verify that checksum on received packets. In that case the function should return zero if the checksum is correct. In particular, this function will return zero if `skb->ip_summed` is `CHECKSUM_UNNECESSARY` which indicates that the hardware has already verified the correctness of the checksum.

# skb\_checksum\_none\_assert

## LINUX

Kernel Hackers Manual January 2013

### Name

`skb_checksum_none_assert` — make sure `skb` `ip_summed` is `CHECKSUM_NONE`

### Synopsis

```
void skb_checksum_none_assert (const struct sk_buff * skb);
```

### Arguments

*skb*

skb to check

### Description

fresh skbs have their `ip_summed` set to `CHECKSUM_NONE`. Instead of forcing `ip_summed` to `CHECKSUM_NONE`, we can use this helper, to document places where we make this assertion.

# struct sock\_common

## LINUX

## Name

`struct sock_common` — minimal network layer representation of sockets

## Synopsis

```
struct sock_common {
    __be32 skc_daddr;
    __be32 skc_rcv_saddr;
    union {unnamed_union};
    int skc_tx_queue_mapping;
    atomic_t skc_refcnt;
};
```

## Members

`skc_daddr`

Foreign IPv4 addr

`skc_rcv_saddr`

Bound local IPv4 addr

`{unnamed_union}`

anonymous

`skc_tx_queue_mapping`

tx queue number for this connection

`skc_refcnt`

reference count

## Description

This is the minimal network layer representation of sockets, the header for `struct sock` and `struct inet_timewait_sock`.

# struct sock

## LINUX

Kernel Hackers Manual January 2013

## Name

`struct sock` — network layer representation of sockets

## Synopsis

```
struct sock {
    struct sock_common __sk_common;
#define sk_node      __sk_common.skc_node
#define sk_nulls_node  __sk_common.skc_nulls_node
#define sk_refcnt    __sk_common.skc_refcnt
#define sk_tx_queue_mapping __sk_common.skc_tx_queue_mapping
#define sk_dontcopy_begin __sk_common.skc_dontcopy_begin
#define sk_dontcopy_end  __sk_common.skc_dontcopy_end
#define sk_hash      __sk_common.skc_hash
#define sk_family    __sk_common.skc_family
#define sk_state     __sk_common.skc_state
#define sk_reuse     __sk_common.skc_reuse
#define sk_bound_dev_if __sk_common.skc_bound_dev_if
#define sk_bind_node __sk_common.skc_bind_node
#define sk_prot      __sk_common.skc_prot
#define sk_net       __sk_common.skc_net
    socket_lock_t sk_lock;
    struct sk_buff_head sk_receive_queue;
    struct sk_backlog;
#define sk_rmem_alloc sk_backlog.rmem_alloc
    int sk_forward_alloc;
#ifdef CONFIG_RPS
    __u32 sk_rxhash;
#endif
    atomic_t sk_drops;
    int sk_rcvbuf;
    struct sk_filter __rcu * sk_filter;
    struct socket_wq __rcu * sk_wq;
#ifdef CONFIG_NET_DMA
```



```

    struct sk_buff_head sk_async_wait_queue;
#endif
#ifdef CONFIG_XFRM
    struct xfrm_policy * sk_policy[2];
#endif
    unsigned long sk_flags;
    struct dst_entry * sk_dst_cache;
    spinlock_t sk_dst_lock;
    atomic_t sk_wmem_alloc;
    atomic_t sk_omem_alloc;
    int sk_sndbuf;
    struct sk_buff_head sk_write_queue;
    unsigned int sk_shutdown:2;
    unsigned int sk_no_check:2;
    unsigned int sk_userlocks:4;
    unsigned int sk_protocol:8;
    unsigned int sk_type:16;
    int sk_wmem_queued;
    gfp_t sk_allocation;
    netdev_features_t sk_route_caps;
    netdev_features_t sk_route_nocaps;
    int sk_gso_type;
    unsigned int sk_gso_max_size;
    int sk_rcvlowat;
    unsigned long sk_lingertime;
    struct sk_buff_head sk_error_queue;
    struct proto * sk_prot_creator;
    rwlock_t sk_callback_lock;
    int sk_err;
    int sk_err_soft;
    unsigned short sk_ack_backlog;
    unsigned short sk_max_ack_backlog;
    __u32 sk_priority;
#ifdef CONFIG_CGROUPS
    __u32 sk_cgrp_prioidx;
#endif
    struct pid * sk_peer_pid;
    const struct cred * sk_peer_cred;
    long sk_rcvtimeo;
    long sk_sndtimeo;
    void * sk_protinfo;
    struct timer_list sk_timer;
    ktime_t sk_stamp;
    struct socket * sk_socket;
    void * sk_user_data;
    struct page * sk_sndmsg_page;
    struct sk_buff * sk_send_head;
    __u32 sk_sndmsg_off;

```

```
__s32 sk_peek_off;
int sk_write_pending;
#ifdef CONFIG_SECURITY
void * sk_security;
#endif
__u32 sk_mark;
u32 sk_classid;
struct cg_proto * sk_cgrp;
void (* sk_state_change) (struct sock *sk);
void (* sk_data_ready) (struct sock *sk, int bytes);
void (* sk_write_space) (struct sock *sk);
void (* sk_error_report) (struct sock *sk);
int (* sk_backlog_rcv) (struct sock *sk, struct sk_buff *skb);
void (* sk_destruct) (struct sock *sk);
};
```

## Members

`__sk_common`

shared layout with `inet_timewait_sock`

`sk_lock`

synchronizer

`sk_receive_queue`

incoming packets

`sk_backlog`

always used with the per-socket spinlock held

`sk_forward_alloc`

space allocated forward

`sk_rxhash`

flow hash received from netif layer

`sk_drops`

raw/udp drops counter

`sk_rcvbuf`

size of receive buffer in bytes

`sk_filter`

socket filtering instructions

`sk_wq`

sock wait queue and async head

`sk_async_wait_queue`

DMA copied packets

`sk_policy[2]`

flow policy

`sk_flags`

`SO_LINGER (l_onoff)`, `SO_BROADCAST`, `SO_KEEPAIVE`, `SO_OOINLINE`  
settings, `SO_TIMESTAMPING` settings

`sk_dst_cache`

destination cache

`sk_dst_lock`

destination cache lock

`sk_wmem_alloc`

transmit queue bytes committed

`sk_omem_alloc`

"o" is "option" or "other"

`sk_sndbuf`

size of send buffer in bytes

`sk_write_queue`

Packet sending queue

`sk_shutdown`

mask of `SEND_SHUTDOWN` and/or `RCV_SHUTDOWN`

`sk_no_check`

`SO_NO_CHECK` setting, whether or not checkup packets

`sk_userlocks`

`SO_SNDBUF` and `SO_RCVBUF` settings

`sk_protocol`

which protocol this socket belongs in this network family

`sk_type`

socket type (`SOCK_STREAM`, etc)

`sk_wmem_queued`

persistent queue size

`sk_allocation`

allocation mode

`sk_route_caps`

route capabilities (e.g. `NETIF_F_TSO`)

`sk_route_nocaps`

forbidden route capabilities (e.g. `NETIF_F_GSO_MASK`)

`sk_gso_type`

GSO type (e.g. `SKB_GSO_TCPV4`)

`sk_gso_max_size`

Maximum GSO segment size to build

`sk_rcvlowat`

`SO_RCVLOWAT` setting

`sk_lingertime`

`SO_LINGER` `l_linger` setting

`sk_error_queue`

rarely used

`sk_prot_creator`

`sk_prot` of original sock creator (see `ipv6_setsockopt`, `IPV6_ADDRFORM` for instance)

`sk_callback_lock`

used with the callbacks in the end of this struct

`sk_err`

last error

`sk_err_soft`

errors that don't cause failure but are the cause of a persistent failure not just 'timed out'

`sk_ack_backlog`

current listen backlog

`sk_max_ack_backlog`

listen backlog set in `listen`

`sk_priority`

`SO_PRIORITY` setting

`sk_cgrp_prioidx`

socket group's priority map index

`sk_peer_pid`

struct pid for this socket's peer

`sk_peer_cred`

`SO_PEERCRED` setting

`sk_rcvtimeo`

`SO_RCVTIMEO` setting

`sk_sndtimeo`

`SO_SNDTIMEO` setting

`sk_protinfo`

private area, net family specific, when not using slab

`sk_timer`

sock cleanup timer

<code>sk_stamp</code>	time stamp of last packet received
<code>sk_socket</code>	Identd and reporting IO signals
<code>sk_user_data</code>	RPC layer private data
<code>sk_sndmsg_page</code>	cached page for sendmsg
<code>sk_send_head</code>	front of stuff to transmit
<code>sk_sndmsg_off</code>	cached offset for sendmsg
<code>sk_peek_off</code>	current peek_offset value
<code>sk_write_pending</code>	a write to stream socket waits to start
<code>sk_security</code>	used by security modules
<code>sk_mark</code>	generic packet mark
<code>sk_classid</code>	this socket's cgroup classid
<code>sk_cgrp</code>	this socket's cgroup-specific proto data
<code>sk_state_change</code>	callback to indicate change in the state of the sock
<code>sk_data_ready</code>	callback to indicate there is data to be processed

`sk_write_space`

callback to indicate there is bf sending space available

`sk_error_report`

callback to indicate errors (e.g. `MSG_ERRQUEUE`)

`sk_backlog_rcv`

callback to process the backlog

`sk_destruct`

called at sock freeing time, i.e. when `all_refcnt == 0`

## unlock\_sock\_fast

### LINUX

Kernel Hackers Manual January 2013

### Name

`unlock_sock_fast` — complement of `lock_sock_fast`

### Synopsis

```
void unlock_sock_fast (struct sock * sk, bool slow);
```

### Arguments

*sk*

socket

*slow*

slow mode

## Description

fast unlock socket for user context. If slow mode is on, we call regular `release_sock`

# sk\_filter\_release

## LINUX

Kernel Hackers Manual January 2013

## Name

`sk_filter_release` — release a socket filter

## Synopsis

```
void sk_filter_release (struct sk_filter * fp);
```

## Arguments

*fp*

filter to remove

## Description

Remove a filter from a socket and release its resources.



# sk\_wmem\_alloc\_get

## LINUX

Kernel Hackers Manual January 2013

### Name

`sk_wmem_alloc_get` — returns write allocations

### Synopsis

```
int sk_wmem_alloc_get (const struct sock * sk);
```

### Arguments

*sk*

socket

### Description

Returns `sk_wmem_alloc` minus initial offset of one

# sk\_rmem\_alloc\_get

## LINUX

## Name

`sk_rmem_alloc_get` — returns read allocations

## Synopsis

```
int sk_rmem_alloc_get (const struct sock * sk);
```

## Arguments

*sk*

socket

## Description

Returns `sk_rmem_alloc`

# sk\_has\_allocations

## LINUX

## Name

`sk_has_allocations` — check if allocations are outstanding

## Synopsis

```
int sk_has_allocations (const struct sock * sk);
```

## Arguments

*sk*

socket

## Description

Returns true if socket has write or read allocations

## wq\_has\_sleeper

### LINUX

Kernel Hackers Manual January 2013

## Name

`wq_has_sleeper` — check if there are any waiting processes

## Synopsis

```
bool wq_has_sleeper (struct socket_wq * wq);
```

## Arguments

*wq*

struct socket\_wq

## Description

Returns true if socket\_wq has waiting processes

The purpose of the wq\_has\_sleeper and sock\_poll\_wait is to wrap the memory barrier call. They were added due to the race found within the tcp code.

## Consider following tcp code paths

CPU1 CPU2

```
sys_select receive packet ... .. __add_wait_queue update tp->rcv_nxt ... ..  
tp->rcv_nxt check sock_def_readable ... { schedule rcu_read_lock; wq =  
rcu_dereference(sk->sk_wq); if (wq && waitqueue_active(wq->wait))  
wake_up_interruptible(wq->wait) ... }
```

The race for tcp fires when the \_\_add\_wait\_queue changes done by CPU1 stay in its cache, and so does the tp->rcv\_nxt update on CPU2 side. The CPU1 could then endup calling schedule and sleep forever if there are no more data on the socket.

## sock\_poll\_wait

**LINUX**

Kernel Hackers Manual January 2013

### Name

sock\_poll\_wait — place memory barrier behind the poll\_wait call.

## Synopsis

```
void sock_poll_wait (struct file * filp, wait_queue_head_t *  
wait_address, poll_table * p);
```

## Arguments

*filp*

file

*wait\_address*

socket wait queue

*p*

poll\_table

## Description

See the comments in the `wq_has_sleeper` function.

## sk\_eat\_skb

**LINUX**

Kernel Hackers Manual January 2013

## Name

`sk_eat_skb` — Release a skb if it is no longer needed

## Synopsis

```
void sk_eat_skb (struct sock * sk, struct sk_buff * skb, int
copied_early);
```

## Arguments

*sk*

socket to eat this skb from

*skb*

socket buffer to eat

*copied\_early*

flag indicating whether DMA operations copied this data early

## Description

This routine must be called with interrupts disabled or with the socket locked so that the sk\_buff queue operation is ok.

# sockfd\_lookup

## LINUX

Kernel Hackers Manual January 2013

## Name

sockfd\_lookup — Go from a file number to its socket slot

## Synopsis

```
struct socket * sockfd_lookup (int fd, int * err);
```

## Arguments

*fd*

file handle

*err*

pointer to an error code return

## Description

The file handle passed in is locked and the socket it is bound too is returned. If an error occurs the *err* pointer is overwritten with a negative *errno* code and *NULL* is returned. The function checks for both invalid handles and passing a handle which is not a socket.

On a success the socket object pointer is returned.

## sock\_release

### LINUX

Kernel Hackers Manual January 2013

### Name

`sock_release` — close a socket

## Synopsis

```
void sock_release (struct socket * sock);
```

## Arguments

*sock*

socket to close

## Description

The socket is released from the protocol stack if it has a release callback, and the inode is then released if the socket is bound to an inode not a file.

# kernel\_recvmsg

## LINUX

Kernel Hackers Manual January 2013

## Name

`kernel_recvmsg` — Receive a message from a socket (kernel space)

## Synopsis

```
int kernel_recvmsg (struct socket * sock, struct msghdr * msg,  
struct kvec * vec, size_t num, size_t size, int flags);
```



## Arguments

*sock*

The socket to receive the message from

*msg*

Received message

*vec*

Input s/g array for message data

*num*

Size of input s/g array

*size*

Number of bytes to read

*flags*

Message flags (MSG\_DONTWAIT, etc...)

## Description

On return the msg structure contains the scatter/gather array passed in the vec argument. The array is modified so that it consists of the unfilled portion of the original array.

The returned value is the total number of bytes received, or an error.

## sock\_register

**LINUX**

Kernel Hackers Manual January 2013

## Name

sock\_register — add a socket protocol handler

## Synopsis

```
int sock_register (const struct net_proto_family * ops);
```

## Arguments

*ops*

description of protocol

## Description

This function is called by a protocol handler that wants to advertise its address family, and have it linked into the socket interface. The value `ops->family` corresponds to the socket system call protocol family.

## sock\_unregister

### LINUX

Kernel Hackers Manual January 2013

## Name

`sock_unregister` — remove a protocol handler

## Synopsis

```
void sock_unregister (int family);
```

## Arguments

*family*

protocol family to remove

## Description

This function is called by a protocol handler that wants to remove its address family, and have it unlinked from the new socket creation.

If protocol handler is a module, then it can use module reference counts to protect against new references. If protocol handler is not a module then it needs to provide its own protection in the ops->create routine.

## \_\_alloc\_skb

### LINUX

Kernel Hackers Manual January 2013

## Name

`__alloc_skb` — allocate a network buffer

## Synopsis

```
struct sk_buff * __alloc_skb (unsigned int size, gfp_t
gfp_mask, int fclone, int node);
```

## Arguments

*size*

size to allocate

*gfp\_mask*

allocation mask

*fclone*

allocate from fclone cache instead of head cache and allocate a cloned (child) skb

*node*

numa node to allocate memory on

## Description

Allocate a new sk\_buff. The returned buffer has no headroom and a tail room of size bytes. The object has a reference count of one. The return is the buffer. On a failure the return is NULL.

Buffers may only be allocated from interrupts using a *gfp\_mask* of GFP\_ATOMIC.

# build\_skb

## LINUX

Kernel Hackers Manual January 2013

## Name

build\_skb — build a network buffer

## Synopsis

```
struct sk_buff * build_skb (void * data);
```

## Arguments

*data*

data buffer provided by caller

## Description

Allocate a new `sk_buff`. Caller provides space holding head and `skb_shared_info`. *data* must have been allocated by `kmalloc`. The return is the new `skb` buffer. On a failure the return is `NULL`, and *data* is not freed.

## Notes

Before IO, driver allocates only data buffer where NIC put incoming frame. Driver should add room at head (`NET_SKB_PAD`) and MUST add room at tail (`SKB_DATA_ALIGN(skb_shared_info)`). After IO, driver calls `build_skb`, to allocate `sk_buff` and populate it before giving packet to stack. RX rings only contains data buffers, not full `skbs`.

## \_\_netdev\_alloc\_skb

### LINUX

Kernel Hackers Manual January 2013

## Name

`__netdev_alloc_skb` — allocate an `skbuff` for rx on a specific device

## Synopsis

```
struct sk_buff * __netdev_alloc_skb (struct net_device * dev,
unsigned int length, gfp_t gfp_mask);
```

## Arguments

*dev*

network device to receive on

*length*

length to allocate

*gfp\_mask*

get\_free\_pages mask, passed to alloc\_skb

## Description

Allocate a new sk\_buff and assign it a usage count of one. The buffer has unspecified headroom built in. Users should allocate the headroom they think they need without accounting for the built in space. The built in space is used for optimisations.

NULL is returned if there is no free memory.

## dev\_alloc\_skb

### LINUX

Kernel Hackers Manual January 2013

### Name

dev\_alloc\_skb — allocate an skbuff for receiving

### Synopsis

```
struct sk_buff * dev_alloc_skb (unsigned int length);
```

## Arguments

*length*

length to allocate

## Description

Allocate a new `sk_buff` and assign it a usage count of one. The buffer has unspecified headroom built in. Users should allocate the headroom they think they need without accounting for the built in space. The built in space is used for optimisations.

`NULL` is returned if there is no free memory. Although this function allocates memory it can be called from an interrupt.

## **\_\_kfree\_skb**

### **LINUX**

Kernel Hackers Manual January 2013

## **Name**

`__kfree_skb` — private function

## **Synopsis**

```
void __kfree_skb (struct sk_buff * skb);
```

## Arguments

*skb*

buffer

## Description

Free an `sk_buff`. Release anything attached to the buffer. Clean the state. This is an internal helper function. Users should always call `kfree_skb`

# kfree\_skb

## LINUX

Kernel Hackers Manual January 2013

## Name

`kfree_skb` — free an `sk_buff`

## Synopsis

```
void kfree_skb (struct sk_buff * skb);
```

## Arguments

*skb*

buffer to free



## Description

Drop a reference to the buffer and free it if the usage count has hit zero.

# consume\_skb

## LINUX

Kernel Hackers Manual January 2013

## Name

consume\_skb — free an skbuff

## Synopsis

```
void consume_skb (struct sk_buff * skb);
```

## Arguments

*skb*

buffer to free

## Description

Drop a ref to the buffer and free it if the usage count has hit zero Functions identically to kfree\_skb, but kfree\_skb assumes that the frame is being dropped after a failure and notes that

# skb\_recycle

**LINUX**

Kernel Hackers Manual January 2013

## Name

`skb_recycle` — clean up an skb for reuse

## Synopsis

```
void skb_recycle (struct sk_buff * skb);
```

## Arguments

*skb*

buffer

## Description

Recycles the skb to be reused as a receive buffer. This function does any necessary reference count dropping, and cleans up the skbuff as if it just came from `__alloc_skb`.

# skb\_recycle\_check

**LINUX**

## Name

`skb_recycle_check` — check if skb can be reused for receive

## Synopsis

```
bool skb_recycle_check (struct sk_buff * skb, int skb_size);
```

## Arguments

*skb*

buffer

*skb\_size*

minimum receive buffer size

## Description

Checks that the skb passed in is not shared or cloned, and that it is linear and its head portion at least as large as `skb_size` so that it can be recycled as a receive buffer. If these conditions are met, this function does any necessary reference count dropping and cleans up the skbuff as if it just came from `__alloc_skb`.

## skb\_morph

**LINUX**

## Name

`skb_morph` — morph one skb into another

## Synopsis

```
struct sk_buff * skb_morph (struct sk_buff * dst, struct  
sk_buff * src);
```

## Arguments

*dst*

the skb to receive the contents

*src*

the skb to supply the contents

## Description

This is identical to `skb_clone` except that the target skb is supplied by the user.

The target skb is returned upon exit.

## skb\_clone

**LINUX**

## Name

`skb_clone` — duplicate an `sk_buff`

## Synopsis

```
struct sk_buff * skb_clone (struct sk_buff * skb, gfp_t  
gfp_mask);
```

## Arguments

*skb*

buffer to clone

*gfp\_mask*

allocation priority

## Description

Duplicate an `sk_buff`. The new one is not owned by a socket. Both copies share the same packet data but not structure. The new buffer has a reference count of 1. If the allocation fails the function returns `NULL` otherwise the new buffer is returned.

If this function is called from an interrupt `gfp_mask` must be `GFP_ATOMIC`.

## `skb_copy`

**LINUX**

## Name

`skb_copy` — create private copy of an `sk_buff`

## Synopsis

```
struct sk_buff * skb_copy (const struct sk_buff * skb, gfp_t  
gfp_mask);
```

## Arguments

*skb*

buffer to copy

*gfp\_mask*

allocation priority

## Description

Make a copy of both an `sk_buff` and its data. This is used when the caller wishes to modify the data and needs a private copy of the data to alter. Returns `NULL` on failure or the pointer to the buffer on success. The returned buffer has a reference count of 1.

As by-product this function converts non-linear `sk_buff` to linear one, so that `sk_buff` becomes completely private and caller is allowed to modify all the data of returned buffer. This means that this function is not recommended for use in circumstances when only header is going to be modified. Use `pskb_copy` instead.

# \_\_pskb\_copy

## LINUX

Kernel Hackers Manual January 2013

### Name

`__pskb_copy` — create copy of an `sk_buff` with private head.

### Synopsis

```
struct sk_buff * __pskb_copy (struct sk_buff * skb, int  
headroom, gfp_t gfp_mask);
```

### Arguments

*skb*

buffer to copy

*headroom*

headroom of new skb

*gfp\_mask*

allocation priority

### Description

Make a copy of both an `sk_buff` and part of its data, located in header. Fragmented data remain shared. This is used when the caller wishes to modify only header of `sk_buff` and needs private copy of the header to alter. Returns `NULL` on failure or the pointer to the buffer on success. The returned buffer has a reference count of 1.

# pskb\_expand\_head

## LINUX

Kernel Hackers Manual January 2013

### Name

`pskb_expand_head` — reallocate header of `sk_buff`

### Synopsis

```
int pskb_expand_head (struct sk_buff * skb, int nhead, int  
ntail, gfp_t gfp_mask);
```

### Arguments

*skb*

buffer to reallocate

*nhead*

room to add at head

*ntail*

room to add at tail

*gfp\_mask*

allocation priority

### Description

Expands (or creates identical copy, if `nhead` and `ntail` are zero) header of `skb`. `sk_buff` itself is not changed. `sk_buff` MUST have reference count of 1. Returns zero in the case of success or error, if expansion failed. In the last case, `sk_buff` is not changed.



All the pointers pointing into skb header may change and must be reloaded after call to this function.

# skb\_copy\_expand

## LINUX

Kernel Hackers Manual January 2013

### Name

skb\_copy\_expand — copy and expand sk\_buff

### Synopsis

```
struct sk_buff * skb_copy_expand (const struct sk_buff * skb,  
int newheadroom, int newtailroom, gfp_t gfp_mask);
```

### Arguments

*skb*

buffer to copy

*newheadroom*

new free bytes at head

*newtailroom*

new free bytes at tail

*gfp\_mask*

allocation priority

## Description

Make a copy of both an `sk_buff` and its data and while doing so allocate additional space.

This is used when the caller wishes to modify the data and needs a private copy of the data to alter as well as more space for new fields. Returns `NULL` on failure or the pointer to the buffer on success. The returned buffer has a reference count of 1.

You must pass `GFP_ATOMIC` as the allocation priority if this function is called from an interrupt.

## skb\_pad

### LINUX

Kernel Hackers Manual January 2013

### Name

`skb_pad` — zero pad the tail of an `skb`

### Synopsis

```
int skb_pad (struct sk_buff * skb, int pad);
```

### Arguments

*skb*

buffer to pad

*pad*

space to pad

## Description

Ensure that a buffer is followed by a padding area that is zero filled. Used by network drivers which may DMA or transfer data beyond the buffer end onto the wire.

May return error in out of memory cases. The skb is freed on error.

# skb\_put

## LINUX

Kernel Hackers Manual January 2013

## Name

skb\_put — add data to a buffer

## Synopsis

```
unsigned char * skb_put (struct sk_buff * skb, unsigned int len);
```

## Arguments

*skb*

buffer to use

*len*

amount of data to add

## Description

This function extends the used data area of the buffer. If this would exceed the total buffer size the kernel will panic. A pointer to the first byte of the extra data is returned.

## skb\_push

### LINUX

Kernel Hackers Manual January 2013

## Name

`skb_push` — add data to the start of a buffer

## Synopsis

```
unsigned char * skb_push (struct sk_buff * skb, unsigned int  
len);
```

## Arguments

*skb*

buffer to use

*len*

amount of data to add

## Description

This function extends the used data area of the buffer at the buffer start. If this would exceed the total buffer headroom the kernel will panic. A pointer to the first

byte of the extra data is returned.

## skb\_pull

### LINUX

Kernel Hackers Manual January 2013

### Name

`skb_pull` — remove data from the start of a buffer

### Synopsis

```
unsigned char * skb_pull (struct sk_buff * skb, unsigned int  
len);
```

### Arguments

*skb*

buffer to use

*len*

amount of data to remove

### Description

This function removes data from the start of a buffer, returning the memory to the headroom. A pointer to the next data in the buffer is returned. Once the data has been pulled future pushes will overwrite the old data.

# skb\_trim

## LINUX

Kernel Hackers Manual January 2013

### Name

`skb_trim` — remove end from a buffer

### Synopsis

```
void skb_trim (struct sk_buff * skb, unsigned int len);
```

### Arguments

*skb*

buffer to alter

*len*

new length

### Description

Cut the length of a buffer down by removing data from the tail. If the buffer is already under the length specified it is not modified. The `skb` must be linear.

# \_\_pskb\_pull\_tail

## LINUX

## Name

`__pskb_pull_tail` — advance tail of skb header

## Synopsis

```
unsigned char * __pskb_pull_tail (struct sk_buff * skb, int delta);
```

## Arguments

*skb*

buffer to reallocate

*delta*

number of bytes to advance tail

## Description

The function makes a sense only on a fragmented `sk_buff`, it expands header moving its tail forward and copying necessary data from fragmented part.

`sk_buff` MUST have reference count of 1.

Returns `NULL` (and `sk_buff` does not change) if pull failed or value of new tail of `skb` in the case of success.

All the pointers pointing into `skb` header may change and must be reloaded after call to this function.

# skb\_copy\_bits

## LINUX

Kernel Hackers Manual January 2013

### Name

`skb_copy_bits` — copy bits from skb to kernel buffer

### Synopsis

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

### Arguments

*skb*

source skb

*offset*

offset in source

*to*

destination buffer

*len*

number of bytes to copy

### Description

Copy the specified number of bytes from the source skb to the destination buffer.

CAUTION ! : If its prototype is ever changed, check `arch/{*}/net/{*}.S` files, since it is called from BPF assembly code.



# skb\_store\_bits

## LINUX

Kernel Hackers Manual January 2013

### Name

`skb_store_bits` — store bits from kernel buffer to skb

### Synopsis

```
int skb_store_bits (struct sk_buff * skb, int offset, const  
void * from, int len);
```

### Arguments

*skb*

destination buffer

*offset*

offset in destination

*from*

source buffer

*len*

number of bytes to copy

## Description

Copy the specified number of bytes from the source buffer to the destination skb. This function handles all the messy bits of traversing fragment lists and such.

# skb\_dequeue

## LINUX

Kernel Hackers Manual January 2013

## Name

skb\_dequeue — remove from the head of the queue

## Synopsis

```
struct sk_buff * skb_dequeue (struct sk_buff_head * list);
```

## Arguments

*list*

list to dequeue from

## Description

Remove the head of the list. The list lock is taken so the function may be used safely with other locking list functions. The head item is returned or `NULL` if the list is empty.

# skb\_dequeue\_tail

## LINUX

Kernel Hackers Manual January 2013

### Name

`skb_dequeue_tail` — remove from the tail of the queue

### Synopsis

```
struct sk_buff * skb_dequeue_tail (struct sk_buff_head *  
list);
```

### Arguments

*list*

list to dequeue from

### Description

Remove the tail of the list. The list lock is taken so the function may be used safely with other locking list functions. The tail item is returned or `NULL` if the list is empty.

# skb\_queue\_purge

## LINUX

## Name

`skb_queue_purge` — empty a list

## Synopsis

```
void skb_queue_purge (struct sk_buff_head * list);
```

## Arguments

*list*

list to empty

## Description

Delete all buffers on an `sk_buff` list. Each buffer is removed from the list and one reference dropped. This function takes the list lock and is atomic with respect to other list locking functions.

# skb\_queue\_head

## LINUX

## Name

`skb_queue_head` — queue a buffer at the list head

## Synopsis

```
void skb_queue_head (struct sk_buff_head * list, struct
sk_buff * newsk);
```

## Arguments

*list*

list to use

*newsk*

buffer to queue

## Description

Queue a buffer at the start of the list. This function takes the list lock and can be used safely with other locking `sk_buff` functions safely.

A buffer cannot be placed on two lists at the same time.

## skb\_queue\_tail

### LINUX

Kernel Hackers Manual January 2013

## Name

`skb_queue_tail` — queue a buffer at the list tail

## Synopsis

```
void skb_queue_tail (struct sk_buff_head * list, struct  
sk_buff * newsk);
```

## Arguments

*list*

list to use

*newsk*

buffer to queue

## Description

Queue a buffer at the tail of the list. This function takes the list lock and can be used safely with other locking `sk_buff` functions safely.

A buffer cannot be placed on two lists at the same time.

## skb\_unlink

### LINUX

Kernel Hackers Manual January 2013

## Name

`skb_unlink` — remove a buffer from a list

## Synopsis

```
void skb_unlink (struct sk_buff * skb, struct sk_buff_head *  
list);
```

## Arguments

*skb*

buffer to remove

*list*

list to use

## Description

Remove a packet from a list. The list locks are taken and this function is atomic with respect to other list locked calls

You must know what list the SKB is on.

## skb\_append

**LINUX**

Kernel Hackers Manual January 2013

## Name

skb\_append — append a buffer

## Synopsis

```
void skb_append (struct sk_buff * old, struct sk_buff * newsk,  
struct sk_buff_head * list);
```

## Arguments

*old*

buffer to insert after

*newsk*

buffer to insert

*list*

list to use

## Description

Place a packet after a given packet in a list. The list locks are taken and this function is atomic with respect to other list locked calls. A buffer cannot be placed on two lists at the same time.

## skb\_insert

**LINUX**

Kernel Hackers Manual January 2013

## Name

`skb_insert` — insert a buffer



## Synopsis

```
void skb_insert (struct sk_buff * old, struct sk_buff * newsk,
struct sk_buff_head * list);
```

## Arguments

*old*

buffer to insert before

*newsk*

buffer to insert

*list*

list to use

## Description

Place a packet before a given packet in a list. The list locks are taken and this function is atomic with respect to other list locked calls.

A buffer cannot be placed on two lists at the same time.

## skb\_split

**LINUX**

Kernel Hackers Manual January 2013

## Name

`skb_split` — Split fragmented skb to two parts at length len.

## Synopsis

```
void skb_split (struct sk_buff * skb, struct sk_buff * skb1,  
const u32 len);
```

## Arguments

*skb*

the buffer to split

*skb1*

the buffer to receive the second part

*len*

new length for *skb*

## skb\_prepare\_seq\_read

### LINUX

Kernel Hackers Manual January 2013

## Name

`skb_prepare_seq_read` — Prepare a sequential read of *skb* data

## Synopsis

```
void skb_prepare_seq_read (struct sk_buff * skb, unsigned int  
from, unsigned int to, struct skb_seq_state * st);
```

## Arguments

*skb*

the buffer to read

*from*

lower offset of data to be read

*to*

upper offset of data to be read

*st*

state variable

## Description

Initializes the specified state variable. Must be called before invoking `skb_seq_read` for the first time.

# skb\_seq\_read

## LINUX

Kernel Hackers Manual January 2013

## Name

`skb_seq_read` — Sequentially read skb data

## Synopsis

```
unsigned int skb_seq_read (unsigned int consumed, const u8 **  
data, struct skb_seq_state * st);
```

## Arguments

*consumed*

number of bytes consumed by the caller so far

*data*

destination pointer for data to be returned

*st*

state variable

## Description

Reads a block of skb data at consumed relative to the lower offset specified to `skb_prepare_seq_read`. Assigns the head of the data block to `data` and returns the length of the block or 0 if the end of the skb data or the upper offset has been reached.

The caller is not required to consume all of the data returned, i.e. `consumed` is typically set to the number of bytes already consumed and the next call to `skb_seq_read` will return the remaining part of the block.

## Note 1

The size of each block of data returned can be arbitrary, this limitation is the cost for zerocopy sequential reads of potentially non linear data.

## Note 2

Fragment lists within fragments are not implemented at the moment, `state->root_skb` could be replaced with a stack for this purpose.

## skb\_abort\_seq\_read

**LINUX**

## Name

`skb_abort_seq_read` — Abort a sequential read of skb data

## Synopsis

```
void skb_abort_seq_read (struct skb_seq_state * st);
```

## Arguments

*st*

state variable

## Description

Must be called if `skb_seq_read` was not called until it returned 0.

# skb\_find\_text

## LINUX

## Name

`skb_find_text` — Find a text pattern in skb data

## Synopsis

```
unsigned int skb_find_text (struct sk_buff * skb, unsigned int
    from, unsigned int to, struct ts_config * config, struct
    ts_state * state);
```

## Arguments

*skb*

the buffer to look in

*from*

search offset

*to*

search limit

*config*

textsearch configuration

*state*

uninitialized textsearch state variable

## Description

Finds a pattern in the *skb* data according to the specified textsearch configuration. Use `textsearch_next` to retrieve subsequent occurrences of the pattern. Returns the offset to the first occurrence or `UINT_MAX` if no match was found.

## skb\_append\_datato\_fragments

**LINUX**

## Name

`skb_append_datato_frags` — append the user data to a skb

## Synopsis

```
int skb_append_datato_frags (struct sock * sk, struct sk_buff  
* skb, int (*getfrag) (void *from, char *to, int offset, int  
len, int odd, struct sk_buff *skb), void * from, int length);
```

## Arguments

*sk*

sock structure

*skb*

skb structure to be appened with user data.

*getfrag*

call back function to be used for getting the user data

*from*

pointer to user message iov

*length*

length of the iov message

## Description

This procedure append the user data in the fragment part of the skb if any page alloc fails user this procedure returns -ENOMEM

# skb\_pull\_rcsum

## LINUX

Kernel Hackers Manual January 2013

### Name

`skb_pull_rcsum` — pull skb and update receive checksum

### Synopsis

```
unsigned char * skb_pull_rcsum (struct sk_buff * skb, unsigned  
int len);
```

### Arguments

*skb*

buffer to update

*len*

length of data pulled

### Description

This function performs an `skb_pull` on the packet and updates the `CHECKSUM_COMPLETE` checksum. It should be used on receive path processing instead of `skb_pull` unless you know that the checksum difference is zero (e.g., a valid IP header) or you are setting `ip_summed` to `CHECKSUM_NONE`.



# skb\_segment

## LINUX

Kernel Hackers Manual January 2013

## Name

`skb_segment` — Perform protocol segmentation on `skb`.

## Synopsis

```
struct sk_buff * skb_segment (struct sk_buff * skb,
netdev_features_t features);
```

## Arguments

*skb*

buffer to segment

*features*

features for the output path (see `dev->features`)

## Description

This function performs segmentation on the given `skb`. It returns a pointer to the first in a list of new skbs for the segments. In case of error it returns `ERR_PTR(err)`.

# skb\_cow\_data

## LINUX

## Name

`skb_cow_data` — Check that a socket buffer's data buffers are writable

## Synopsis

```
int skb_cow_data (struct sk_buff * skb, int tailbits, struct
sk_buff ** trailer);
```

## Arguments

*skb*

The socket buffer to check.

*tailbits*

Amount of trailing space to be added

*trailer*

Returned pointer to the *skb* where the *tailbits* space begins

## Description

Make sure that the data buffers attached to a socket buffer are writable. If they are not, private copies are made of the data buffers and the socket buffer is set to use these instead.

If *tailbits* is given, make sure that there is space to write *tailbits* bytes of data beyond current end of socket buffer. *trailer* will be set to point to the *skb* in which this space begins.

The number of scatterlist elements required to completely map the COW'd and extended socket buffer will be returned.

# skb\_partial\_csum\_set

## LINUX

Kernel Hackers Manual January 2013

### Name

`skb_partial_csum_set` — set up and verify partial csum values for packet

### Synopsis

```
bool skb_partial_csum_set (struct sk_buff * skb, u16 start,  
u16 off);
```

### Arguments

*skb*

the skb to set

*start*

the number of bytes after `skb->data` to start checksumming.

*off*

the offset from `start` to place the checksum.

### Description

For untrusted partially-checksummed packets, we need to make sure the values for `skb->csum_start` and `skb->csum_offset` are valid so we don't oops.

This function checks and sets those values and `skb->ip_summed`: if this returns false you should drop the packet.

# sk\_alloc

## LINUX

Kernel Hackers Manual January 2013

### Name

`sk_alloc` — All socket objects are allocated here

### Synopsis

```
struct sock * sk_alloc (struct net * net, int family, gfp_t  
priority, struct proto * prot);
```

### Arguments

*net*

the applicable net namespace

*family*

protocol family

*priority*

for allocation (GFP\_KERNEL, GFP\_ATOMIC, etc)

*prot*

struct proto associated with this new sock instance

# sk\_clone\_lock

## LINUX

## Name

`sk_clone_lock` — clone a socket, and lock its clone

## Synopsis

```
struct sock * sk_clone_lock (const struct sock * sk, const
gfp_t priority);
```

## Arguments

*sk*

the socket to clone

*priority*

for allocation (GFP\_KERNEL, GFP\_ATOMIC, etc)

## Description

Caller must unlock socket even in error path (`bh_unlock_sock(newsk)`)

# sk\_wait\_data

## LINUX

## Name

`sk_wait_data` — wait for data to arrive at `sk_receive_queue`

## Synopsis

```
int sk_wait_data (struct sock * sk, long * timeo);
```

## Arguments

*sk*

sock to wait on

*timeo*

for how long

## Description

Now socket state including `sk->sk_err` is changed only under lock, hence we may omit checks after joining wait queue. We check receive queue before `schedule` only as optimization; it is very likely that `release_sock` added new data.

## \_\_sk\_mem\_schedule

### LINUX

Kernel Hackers Manual January 2013

## Name

`__sk_mem_schedule` — increase `sk_forward_alloc` and `memory_allocated`

## Synopsis

```
int __sk_mem_schedule (struct sock * sk, int size, int kind);
```

## Arguments

*sk*

socket

*size*

memory size to allocate

*kind*

allocation type

## Description

If *kind* is `SK_MEM_SEND`, it means `wmem` allocation. Otherwise it means `rmem` allocation. This function assumes that protocols which have `memory_pressure` use `sk_wmem_queued` as write buffer accounting.

## \_\_sk\_mem\_reclaim

### LINUX

Kernel Hackers Manual January 2013

### Name

`__sk_mem_reclaim` — reclaim memory\_allocated

## Synopsis

```
void __sk_mem_reclaim (struct sock * sk);
```

## Arguments

*sk*

socket

## lock\_sock\_fast

### LINUX

Kernel Hackers Manual January 2013

## Name

`lock_sock_fast` — fast version of `lock_sock`

## Synopsis

```
bool lock_sock_fast (struct sock * sk);
```

## Arguments

*sk*

socket

## Description

This version should be used for very small section, where process wont block return false if fast path is taken `sk_lock.slock` locked, owned = 0, BH disabled return true if slow path is taken `sk_lock.slock` unlocked, owned = 1, BH enabled



# \_\_skb\_recv\_datagram

## LINUX

Kernel Hackers Manual January 2013

### Name

`__skb_recv_datagram` — Receive a datagram skbuff

### Synopsis

```
struct sk_buff * __skb_recv_datagram (struct sock * sk,
unsigned flags, int * peeked, int * off, int * err);
```

### Arguments

*sk*

socket

*flags*

MSG\_ flags

*peeked*

returns non-zero if this packet has been seen before

*off*

an offset in bytes to peek skb from. Returns an offset within an skb where data actually starts

*err*

error code returned

## Description

Get a datagram skbuff, understands the peeking, nonblocking wakeups and possible races. This replaces identical code in packet, raw and udp, as well as the IPX AX.25 and Appletalk. It also finally fixes the long standing peek and read race for datagram sockets. If you alter this routine remember it must be re-entrant.

This function will lock the socket if a skb is returned, so the caller needs to unlock the socket in that case (usually by calling `skb_free_datagram`)

\* It does not lock socket since today. This function is \* free of race conditions. This measure should/can improve \* significantly datagram socket latencies at high loads, \* when data copying to user space takes lots of time. \* (BTW I've just killed the last `cli` in IP/IPv6/core/netlink/packet \* 8) Great win.) \* --ANK (980729)

The order of the tests when we find no data waiting are specified quite explicitly by POSIX 1003.1g, don't change them without having the standard around please.

## skb\_kill\_datagram

### LINUX

Kernel Hackers Manual January 2013

### Name

`skb_kill_datagram` — Free a datagram skbuff forcibly

### Synopsis

```
int skb_kill_datagram (struct sock * sk, struct sk_buff * skb,
unsigned int flags);
```

## Arguments

*sk*

socket

*skb*

datagram skbuff

*flags*

MSG\_ flags

## Description

This function frees a datagram skbuff that was received by `skb_recv_datagram`. The `flags` argument must match the one used for `skb_recv_datagram`.

If the `MSG_PEEK` flag is set, and the packet is still on the receive queue of the socket, it will be taken off the queue before it is freed.

This function currently only disables BH when acquiring the `sk_receive_queue` lock. Therefore it must not be used in a context where that lock is acquired in an IRQ context.

It returns 0 if the packet was removed by us.

## skb\_copy\_datagram\_iovec

### LINUX

Kernel Hackers Manual January 2013

### Name

`skb_copy_datagram_iovec` — Copy a datagram to an iovec.

## Synopsis

```
int skb_copy_datagram_iovec (const struct sk_buff * skb, int
offset, struct iovec * to, int len);
```

## Arguments

*skb*

buffer to copy

*offset*

offset in the buffer to start copying from

*to*

io vector to copy to

*len*

amount of data to copy from buffer to iovec

## Note

the iovec is modified during the copy.

# skb\_copy\_datagram\_const\_iovec

## LINUX

Kernel Hackers Manual January 2013

## Name

skb\_copy\_datagram\_const\_iovec — Copy a datagram to an iovec.

## Synopsis

```
int skb_copy_datagram_const_iovec (const struct sk_buff * skb,  
int offset, const struct iovec * to, int to_offset, int len);
```

## Arguments

*skb*

buffer to copy

*offset*

offset in the buffer to start copying from

*to*

io vector to copy to

*to\_offset*

offset in the io vector to start copying to

*len*

amount of data to copy from buffer to iovec

## Description

Returns 0 or -EFAULT.

## Note

the iovec is not modified during the copy.

# skb\_copy\_datagram\_from\_iovec

## LINUX

Kernel Hackers Manual January 2013

### Name

`skb_copy_datagram_from_iovec` — Copy a datagram from an iovec.

### Synopsis

```
int skb_copy_datagram_from_iovec (struct sk_buff * skb, int
offset, const struct iovec * from, int from_offset, int len);
```

### Arguments

*skb*

buffer to copy

*offset*

offset in the buffer to start copying to

*from*

io vector to copy to

*from\_offset*

offset in the io vector to start copying from

*len*

amount of data to copy to buffer from iovec

### Description

Returns 0 or -EFAULT.

## Note

the iovec is not modified during the copy.

# skb\_copy\_and\_csum\_datagram\_iovec

## LINUX

Kernel Hackers Manual January 2013

## Name

`skb_copy_and_csum_datagram_iovec` — Copy and checksum skb to user iovec.

## Synopsis

```
int skb_copy_and_csum_datagram_iovec (struct sk_buff * skb,
int hlen, struct iovec * iov);
```

## Arguments

*skb*

skbuff

*hlen*

hardware length

*iov*

io vector

## Description

Caller `_must_` check that `skb` will fit to this `iovec`.

## Returns

0 - success. -EINVAL - checksum failure. -EFAULT - fault during copy. Beware, in this case `iovec` can be modified!

# datagram\_poll

## LINUX

Kernel Hackers Manual January 2013

## Name

`datagram_poll` — generic datagram poll

## Synopsis

```
unsigned int datagram_poll (struct file * file, struct socket
* sock, poll_table * wait);
```

## Arguments

*file*

file struct

*sock*

socket



*wait*

poll table

## Datagram poll

Again totally generic. This also handles sequenced packet sockets providing the socket receive queue is only ever holding data ready to receive.

## Note

when you *\_don't\_* use this routine for this protocol, and you use a different write policy from `sock_writeable` then please supply your own `write_space` callback.

# sk\_stream\_write\_space

## LINUX

Kernel Hackers Manual January 2013

## Name

`sk_stream_write_space` — stream socket `write_space` callback.

## Synopsis

```
void sk_stream_write_space (struct sock * sk);
```

## Arguments

*sk*

socket

## FIXME

write proper description

# sk\_stream\_wait\_connect

## LINUX

Kernel Hackers Manual January 2013

## Name

`sk_stream_wait_connect` — Wait for a socket to get into the connected state

## Synopsis

```
int sk_stream_wait_connect (struct sock * sk, long * timeo_p);
```

## Arguments

*sk*

sock to wait on

*timeo\_p*

for how long to wait

## Description

Must be called with the socket locked.

# sk\_stream\_wait\_memory

## LINUX

Kernel Hackers Manual January 2013

### Name

`sk_stream_wait_memory` — Wait for more memory for a socket

### Synopsis

```
int sk_stream_wait_memory (struct sock * sk, long * timeo_p);
```

### Arguments

*sk*

socket to wait for memory

*timeo\_p*

for how long

## 1.3. Socket Filter

### sk\_filter

## LINUX

## Name

`sk_filter` — run a packet through a socket filter

## Synopsis

```
int sk_filter (struct sock * sk, struct sk_buff * skb);
```

## Arguments

*sk*

sock associated with `sk_buff`

*skb*

buffer to filter

## Description

Run the filter code and then cut `skb->data` to correct size returned by `sk_run_filter`. If `pkt_len` is 0 we toss packet. If `skb->len` is smaller than `pkt_len` we keep whole `skb->data`. This is the socket level wrapper to `sk_run_filter`. It returns 0 if the packet should be accepted or `-EPERM` if the packet should be tossed.

## `sk_run_filter`

**LINUX**

## Name

`sk_run_filter` — run a filter on a socket

## Synopsis

```
unsigned int sk_run_filter (const struct sk_buff * skb, const  
struct sock_filter * fentry);
```

## Arguments

*skb*

buffer to run the filter on

*fentry*

filter to apply

## Description

Decode and apply filter instructions to the `skb->data`. Return length to keep, 0 for none. *skb* is the data we are filtering, *filter* is the array of filter instructions. Because all jumps are guaranteed to be before last instruction, and last instruction guaranteed to be a RET, we dont need to check `flen`. (We used to pass to this function the length of filter)

## `sk_chk_filter`

**LINUX**

## Name

`sk_chk_filter` — verify socket filter code

## Synopsis

```
int sk_chk_filter (struct sock_filter * filter, unsigned int  
flen);
```

## Arguments

*filter*

filter to verify

*flen*

length of filter

## Description

Check the user's filter code. If we let some ugly filter code slip through kaboom! The filter must contain no references or jumps that are out of range, no illegal instructions, and must end with a RET instruction.

All jumps are forward as they are not signed.

Returns 0 if the rule set is legal or -EINVAL if not.

## `sk_filter_release_rcu`

**LINUX**

## Name

`sk_filter_release_rcu` — Release a socket filter by `rcu_head`

## Synopsis

```
void sk_filter_release_rcu (struct rcu_head * rcu);
```

## Arguments

*rcu*

`rcu_head` that contains the `sk_filter` to free

# sk\_attach\_filter

## LINUX

## Name

`sk_attach_filter` — attach a socket filter

## Synopsis

```
int sk_attach_filter (struct sock_fprog * fprog, struct sock *  
sk);
```

## Arguments

*fprog*

the filter program

*sk*

the socket to use

## Description

Attach the user's filter code. We first run some sanity checks on it to make sure it does not explode on us later. If an error occurs or there is insufficient memory for the filter a negative `errno` code is returned. On success the return is zero.

## 1.4. Generic Network Statistics

### struct gnet\_stats\_basic

#### LINUX

Kernel Hackers Manual January 2013

#### Name

`struct gnet_stats_basic` — byte/packet throughput statistics

#### Synopsis

```
struct gnet_stats_basic {  
    __u64 bytes;  
    __u32 packets;  
};
```



## Members

bytes

number of seen bytes

packets

number of seen packets

## struct gnet\_stats\_rate\_est

### LINUX

Kernel Hackers Manual January 2013

## Name

struct gnet\_stats\_rate\_est — rate estimator

## Synopsis

```
struct gnet_stats_rate_est {  
    __u32 bps;  
    __u32 pps;  
};
```

## Members

bps

current byte rate

pps

current packet rate

# struct gnet\_stats\_queue

## LINUX

Kernel Hackers Manual January 2013

### Name

struct gnet\_stats\_queue — queuing statistics

### Synopsis

```
struct gnet_stats_queue {  
    __u32 qlen;  
    __u32 backlog;  
    __u32 drops;  
    __u32 requeues;  
    __u32 overlimits;  
};
```

### Members

qlen

queue length

backlog

backlog size of queue

drops

number of dropped packets

requeues

number of requeues

overlimits

number of enqueues over the limit

# struct gnet\_estimator

## LINUX

Kernel Hackers Manual January 2013

### Name

struct gnet\_estimator — rate estimator configuration

### Synopsis

```
struct gnet_estimator {  
    signed char interval;  
    unsigned char ewma_log;  
};
```

### Members

interval

sampling period

ewma\_log

the log of measurement window weight

# gnet\_stats\_start\_copy\_compat

## LINUX

Kernel Hackers Manual January 2013

### Name

gnet\_stats\_start\_copy\_compat — start dumping procedure in compatibility mode

## Synopsis

```
int gnet_stats_start_copy_compat (struct sk_buff * skb, int
type, int tc_stats_type, int xstats_type, spinlock_t * lock,
struct gnet_dump * d);
```

## Arguments

*skb*

socket buffer to put statistics TLVs into

*type*

TLV type for top level statistic TLV

*tc\_stats\_type*

TLV type for backward compatibility struct tc\_stats TLV

*xstats\_type*

TLV type for backward compatibility xstats TLV

*lock*

statistics lock

*d*

dumping handle

## Description

Initializes the dumping handle, grabs the statistic lock and appends an empty TLV header to the socket buffer for use a container for all other statistic TLVS.

The dumping handle is marked to be in backward compatibility mode telling all `gnet_stats_copy_XXX` functions to fill a local copy of struct tc\_stats.

Returns 0 on success or -1 if the room in the socket buffer was not sufficient.

# gnet\_stats\_start\_copy

## LINUX

Kernel Hackers Manual January 2013

## Name

`gnet_stats_start_copy` — start dumping procedure in compatibility mode

## Synopsis

```
int gnet_stats_start_copy (struct sk_buff * skb, int type,
spinlock_t * lock, struct gnet_dump * d);
```

## Arguments

*skb*

socket buffer to put statistics TLVs into

*type*

TLV type for top level statistic TLV

*lock*

statistics lock

*d*

dumping handle

## Description

Initializes the dumping handle, grabs the statistic lock and appends an empty TLV header to the socket buffer for use a container for all other statistic TLVS.

Returns 0 on success or -1 if the room in the socket buffer was not sufficient.

# **gnet\_stats\_copy\_basic**

## **LINUX**

Kernel Hackers Manual January 2013

### **Name**

`gnet_stats_copy_basic` — copy basic statistics into statistic TLV

### **Synopsis**

```
int gnet_stats_copy_basic (struct gnet_dump * d, struct  
gnet_stats_basic_packed * b);
```

### **Arguments**

*d*

dumping handle

*b*

basic statistics

### **Description**

Appends the basic statistics to the top level TLV created by `gnet_stats_start_copy`.

Returns 0 on success or -1 with the statistic lock released if the room in the socket buffer was not sufficient.

# **gnet\_stats\_copy\_rate\_est**

## **LINUX**

Kernel Hackers Manual January 2013

### **Name**

`gnet_stats_copy_rate_est` — copy rate estimator statistics into statistics TLV

### **Synopsis**

```
int gnet_stats_copy_rate_est (struct gnet_dump * d, const
struct gnet_stats_basic_packed * b, struct gnet_stats_rate_est
* r);
```

### **Arguments**

<i>d</i>	dumping handle
<i>b</i>	basic statistics
<i>r</i>	rate estimator statistics

### **Description**

Appends the rate estimator statistics to the top level TLV created by `gnet_stats_start_copy`.

Returns 0 on success or -1 with the statistic lock released if the room in the socket buffer was not sufficient.

# **gnet\_stats\_copy\_queue**

## **LINUX**

Kernel Hackers Manual January 2013

### **Name**

`gnet_stats_copy_queue` — copy queue statistics into statistics TLV

### **Synopsis**

```
int gnet_stats_copy_queue (struct gnet_dump * d, struct
gnet_stats_queue * q);
```

### **Arguments**

*d*

dumping handle

*q*

queue statistics

### **Description**

Appends the queue statistics to the top level TLV created by `gnet_stats_start_copy`.

Returns 0 on success or -1 with the statistic lock released if the room in the socket buffer was not sufficient.



# **gnet\_stats\_copy\_app**

## **LINUX**

Kernel Hackers Manual January 2013

### **Name**

`gnet_stats_copy_app` — copy application specific statistics into statistics TLV

### **Synopsis**

```
int gnet_stats_copy_app (struct gnet_dump * d, void * st, int len);
```

### **Arguments**

*d*

dumping handle

*st*

application specific statistics data

*len*

length of data

### **Description**

Appends the application sepecific statistics to the top level TLV created by `gnet_stats_start_copy` and remembers the data for XSTATS if the dumping handle is in backward compatibility mode.

Returns 0 on success or -1 with the statistic lock released if the room in the socket buffer was not sufficient.

# **gnet\_stats\_finish\_copy**

## **LINUX**

Kernel Hackers Manual January 2013

### **Name**

`gnet_stats_finish_copy` — finish dumping procedure

### **Synopsis**

```
int gnet_stats_finish_copy (struct gnet_dump * d);
```

### **Arguments**

*d*

dumping handle

### **Description**

Corrects the length of the top level TLV to include all TLVs added by `gnet_stats_copy_XXX` calls. Adds the backward compatibility TLVs if `gnet_stats_start_copy_compat` was used and releases the statistics lock.

Returns 0 on success or -1 with the statistic lock released if the room in the socket buffer was not sufficient.

# gen\_new\_estimator

## LINUX

Kernel Hackers Manual January 2013

## Name

`gen_new_estimator` — create a new rate estimator

## Synopsis

```
int gen_new_estimator (struct gnet_stats_basic_packed *
    bstats, struct gnet_stats_rate_est * rate_est, spinlock_t *
    stats_lock, struct nlattr * opt);
```

## Arguments

*bstats*

basic statistics

*rate\_est*

rate estimator statistics

*stats\_lock*

statistics lock

*opt*

rate estimator configuration TLV

## Description

Creates a new rate estimator with *bstats* as source and *rate\_est* as destination. A new timer with the interval specified in the configuration TLV is created. Upon each interval, the latest statistics will be read from *bstats* and the estimated rate will be stored in *rate\_est* with the statistics lock grabbed during this period.

Returns 0 on success or a negative error code.

# gen\_kill\_estimator

## LINUX

Kernel Hackers Manual January 2013

### Name

`gen_kill_estimator` — remove a rate estimator

### Synopsis

```
void gen_kill_estimator (struct gnet_stats_basic_packed *  
    bstats, struct gnet_stats_rate_est * rate_est);
```

### Arguments

*bstats*

basic statistics

*rate\_est*

rate estimator statistics

### Description

Removes the rate estimator specified by *bstats* and *rate\_est*.

### Note

Caller should respect an RCU grace period before freeing *stats\_lock*

# gen\_replace\_estimator

## LINUX

Kernel Hackers Manual January 2013

### Name

`gen_replace_estimator` — replace rate estimator configuration

### Synopsis

```
int gen_replace_estimator (struct gnet_stats_basic_packed *  
    bstats, struct gnet_stats_rate_est * rate_est, spinlock_t *  
    stats_lock, struct nlattr * opt);
```

### Arguments

*bstats*

basic statistics

*rate\_est*

rate estimator statistics

*stats\_lock*

statistics lock

*opt*

rate estimator configuration TLV

## Description

Replaces the configuration of a rate estimator by calling `gen_kill_estimator` and `gen_new_estimator`.

Returns 0 on success or a negative error code.

# gen\_estimator\_active

## LINUX

Kernel Hackers Manual January 2013

## Name

`gen_estimator_active` — test if estimator is currently in use

## Synopsis

```
bool gen_estimator_active (const struct
gnet_stats_basic_packed * bstats, const struct
gnet_stats_rate_est * rate_est);
```

## Arguments

*bstats*

basic statistics

*rate\_est*

rate estimator statistics

## Description

Returns true if estimator is active, and false if not.

## 1.5. SUN RPC subsystem

### xdr\_encode\_opaque\_fixed

#### LINUX

Kernel Hackers Manual January 2013

#### Name

`xdr_encode_opaque_fixed` — Encode fixed length opaque data

#### Synopsis

```
__be32 * xdr_encode_opaque_fixed (__be32 * p, const void *  
ptr, unsigned int nbytes);
```

#### Arguments

*p*

pointer to current position in XDR buffer.

*ptr*

pointer to data to encode (or NULL)

*nbytes*

size of data.

## Description

Copy the array of data of length `nbytes` at `ptr` to the XDR buffer at position `p`, then align to the next 32-bit boundary by padding with zero bytes (see RFC1832).

## Note

if `ptr` is `NULL`, only the padding is performed.

Returns the updated current XDR buffer position

# xdr\_encode\_opaque

## LINUX

Kernel Hackers Manual January 2013

## Name

`xdr_encode_opaque` — Encode variable length opaque data

## Synopsis

```
__be32 * xdr_encode_opaque (__be32 * p, const void * ptr,  
unsigned int nbytes);
```

## Arguments

*p*

pointer to current position in XDR buffer.

*ptr*

pointer to data to encode (or `NULL`)



*nbytes*

size of data.

## Description

Returns the updated current XDR buffer position

# xdr\_terminate\_string

## LINUX

Kernel Hackers Manual January 2013

## Name

`xdr_terminate_string` — '\0'-terminate a string residing in an `xdr_buf`

## Synopsis

```
void xdr_terminate_string (struct xdr_buf * buf, const u32  
len);
```

## Arguments

*buf*

XDR buffer where string resides

*len*

length of string, in bytes

# xdr\_init\_encode

## LINUX

Kernel Hackers Manual January 2013

### Name

`xdr_init_encode` — Initialize a struct `xdr_stream` for sending data.

### Synopsis

```
void xdr_init_encode (struct xdr_stream * xdr, struct xdr_buf  
* buf, __be32 * p);
```

### Arguments

*xdr*

pointer to `xdr_stream` struct

*buf*

pointer to XDR buffer in which to encode data

*p*

current pointer inside XDR buffer

### Note

at the moment the RPC client only passes the length of our scratch buffer in the `xdr_buf`'s header `kvec`. Previously this meant we needed to call `xdr_adjust_iovec` after encoding the data. With the new scheme, the `xdr_stream` manages the details of the buffer length, and takes care of adjusting the `kvec` length for us.

# xdr\_reserve\_space

## LINUX

Kernel Hackers Manual January 2013

### Name

`xdr_reserve_space` — Reserve buffer space for sending

### Synopsis

```
__be32 * xdr_reserve_space (struct xdr_stream * xdr, size_t  
nbytes);
```

### Arguments

*xdr*

pointer to `xdr_stream`

*nbytes*

number of bytes to reserve

### Description

Checks that we have enough buffer space to encode 'nbytes' more bytes of data. If so, update the total `xdr_buf` length, and adjust the length of the current `kvec`.

# xdr\_write\_pages

## LINUX

## Name

`xdr_write_pages` — Insert a list of pages into an XDR buffer for sending

## Synopsis

```
void xdr_write_pages (struct xdr_stream * xdr, struct page **  
pages, unsigned int base, unsigned int len);
```

## Arguments

*xdr*

pointer to `xdr_stream`

*pages*

list of pages

*base*

offset of first byte

*len*

length of data in bytes

## `xdr_init_decode`

**LINUX**

## Name

`xdr_init_decode` — Initialize an `xdr_stream` for decoding data.

## Synopsis

```
void xdr_init_decode (struct xdr_stream * xdr, struct xdr_buf  
* buf, __be32 * p);
```

## Arguments

*xdr*

pointer to `xdr_stream` struct

*buf*

pointer to XDR buffer from which to decode data

*p*

current pointer inside XDR buffer

## `xdr_init_decode_pages`

### LINUX

## Name

`xdr_init_decode_pages` — Initialize an `xdr_stream` for decoding data.

## Synopsis

```
void xdr_init_decode_pages (struct xdr_stream * xdr, struct  
xdr_buf * buf, struct page ** pages, unsigned int len);
```

## Arguments

*xdr*

pointer to xdr\_stream struct

*buf*

pointer to XDR buffer from which to decode data

*pages*

list of pages to decode into

*len*

length in bytes of buffer in pages

## xdr\_set\_scratch\_buffer

### LINUX

Kernel Hackers Manual January 2013

### Name

`xdr_set_scratch_buffer` — Attach a scratch buffer for decoding data.

## Synopsis

```
void xdr_set_scratch_buffer (struct xdr_stream * xdr, void *  
buf, size_t buflen);
```

## Arguments

*xdr*

pointer to xdr\_stream struct

*buf*

pointer to an empty buffer

*buflen*

size of 'buf'

## Description

The scratch buffer is used when decoding from an array of pages. If an `xdr_inline_decode` call spans across page boundaries, then we copy the data into the scratch buffer in order to allow linear access.

## xdr\_inline\_decode

**LINUX**

Kernel Hackers Manual January 2013

## Name

`xdr_inline_decode` — Retrieve XDR data to decode

## Synopsis

```
__be32 * xdr_inline_decode (struct xdr_stream * xdr, size_t
nbytes);
```

## Arguments

*xdr*

pointer to xdr\_stream struct

*nbytes*

number of bytes of data to decode

## Description

Check if the input buffer is long enough to enable us to decode 'nbytes' more bytes of data starting at the current position. If so return the current pointer, then update the current pointer position.

## xdr\_read\_pages

### LINUX

Kernel Hackers Manual January 2013

## Name

`xdr_read_pages` — Ensure page-based XDR data to decode is aligned at current pointer position



## Synopsis

```
void xdr_read_pages (struct xdr_stream * xdr, unsigned int  
len);
```

## Arguments

*xdr*

pointer to xdr\_stream struct

*len*

number of bytes of page data

## Description

Moves data beyond the current pointer position from the XDR head[] buffer into the page list. Any data that lies beyond current position + “len” bytes is moved into the XDR tail[].

## xdr\_enter\_page

### LINUX

Kernel Hackers Manual January 2013

## Name

xdr\_enter\_page — decode data from the XDR page

## Synopsis

```
void xdr_enter_page (struct xdr_stream * xdr, unsigned int  
len);
```

## Arguments

*xdr*

pointer to xdr\_stream struct

*len*

number of bytes of page data

## Description

Moves data beyond the current pointer position from the XDR head[] buffer into the page list. Any data that lies beyond current position + “len” bytes is moved into the XDR tail[]. The current pointer is then repositioned at the beginning of the first XDR page.

## svc\_print\_addr

### LINUX

Kernel Hackers Manual January 2013

## Name

svc\_print\_addr — Format rq\_addr field for printing

## Synopsis

```
char * svc_print_addr (struct svc_rqst * rqstp, char * buf,
size_t len);
```

## Arguments

*rqstp*

svc\_rqst struct containing address to print

*buf*

target buffer for formatted address

*len*

length of target buffer

## svc\_reserve

### LINUX

Kernel Hackers Manual January 2013

## Name

`svc_reserve` — change the space reserved for the reply to a request.

## Synopsis

```
void svc_reserve (struct svc_rqst * rqstp, int space);
```

## Arguments

*rqstp*

The request in question

*space*

new max space to reserve

## Description

Each request reserves some space on the output queue of the transport to make sure the reply fits. This function reduces that reserved space to be the amount of space used already, plus *space*.

## svc\_find\_xprt

### LINUX

Kernel Hackers Manual January 2013

## Name

`svc_find_xprt` — find an RPC transport instance

## Synopsis

```
struct svc_xprt * svc_find_xprt (struct svc_serv * serv, const  
char * xcl_name, struct net * net, const sa_family_t af, const  
unsigned short port);
```

## Arguments

*serv*

pointer to `svc_serv` to search

*xcl\_name*

C string containing transport's class name

*net*

owner net pointer

*af*

Address family of transport's local address

*port*

transport's IP port number

## Description

Return the transport instance pointer for the endpoint accepting connections/peer traffic from the specified transport class, address family and port.

Specifying 0 for the address family or port is effectively a wild-card, and will result in matching the first transport in the service's list that has a matching class name.

## svc\_xprt\_names

### LINUX

Kernel Hackers Manual January 2013

## Name

`svc_xprt_names` — format a buffer with a list of transport names

## Synopsis

```
int svc_xprt_names (struct svc_serv * serv, char * buf, const  
int buflen);
```

## Arguments

*serv*

pointer to an RPC service

*buf*

pointer to a buffer to be filled in

*buflen*

length of buffer to be filled in

## Description

Fills in *buf* with a string containing a list of transport names, each name terminated with '\n'.

Returns positive length of the filled-in string on success; otherwise a negative errno value is returned if an error occurs.

# xprt\_register\_transport

**LINUX**

Kernel Hackers Manual January 2013

## Name

`xprt_register_transport` — register a transport implementation

## Synopsis

```
int xprt_register_transport (struct xprt_class * transport);
```

## Arguments

*transport*

transport to register

## Description

If a transport implementation is loaded as a kernel module, it can call this interface to make itself known to the RPC client.

### 0

transport successfully registered -EEXIST: transport already registered -EINVAL: transport module being unloaded

## xprt\_unregister\_transport

### LINUX

Kernel Hackers Manual January 2013

## Name

`xprt_unregister_transport` — unregister a transport implementation

## Synopsis

```
int xprt_unregister_transport (struct xprt_class * transport);
```

## Arguments

*transport*

transport to unregister

**0**

transport successfully unregistered -ENOENT: transport never registered

## xprt\_load\_transport

**LINUX**

Kernel Hackers Manual January 2013

## Name

`xprt_load_transport` — load a transport implementation

## Synopsis

```
int xprt_load_transport (const char * transport_name);
```



## Arguments

*transport\_name*

transport to load

0

transport successfully loaded -ENOENT: transport module not available

## xprt\_reserve\_xprt

**LINUX**

Kernel Hackers Manual January 2013

## Name

xprt\_reserve\_xprt — serialize write access to transports

## Synopsis

```
int xprt_reserve_xprt (struct rpc_xprt * xprt, struct rpc_task  
* task);
```

## Arguments

*xprt*

pointer to the target transport

*task*

task that is requesting access to the transport

## Description

This prevents mixing the payload of separate requests, and prevents transport connects from colliding with writes. No congestion control is provided.

# xprt\_release\_xprt

## LINUX

Kernel Hackers Manual January 2013

## Name

`xprt_release_xprt` — allow other requests to use a transport

## Synopsis

```
void xprt_release_xprt (struct rpc_xprt * xprt, struct  
rpc_task * task);
```

## Arguments

*xprt*

transport with other tasks potentially waiting

*task*

task that is releasing access to the transport

## Description

Note that “task” can be NULL. No congestion control is provided.

# xprt\_release\_xprt\_cong

## LINUX

Kernel Hackers Manual January 2013

### Name

`xprt_release_xprt_cong` — allow other requests to use a transport

### Synopsis

```
void xprt_release_xprt_cong (struct rpc_xprt * xprt, struct  
rpc_task * task);
```

### Arguments

*xprt*

transport with other tasks potentially waiting

*task*

task that is releasing access to the transport

### Description

Note that “task” can be NULL. Another task is awoken to use the transport if the transport’s congestion window allows it.

# xprt\_release\_rqst\_cong

## LINUX

## Name

`xprt_release_rqst_cong` — housekeeping when request is complete

## Synopsis

```
void xprt_release_rqst_cong (struct rpc_task * task);
```

## Arguments

*task*

RPC request that recently completed

## Description

Useful for transports that require congestion control.

# xprt\_adjust\_cwnd

## LINUX

## Name

`xprt_adjust_cwnd` — adjust transport congestion window

## Synopsis

```
void xprt_adjust_cwnd (struct rpc_task * task, int result);
```

## Arguments

*task*

recently completed RPC request used to adjust window

*result*

result code of completed RPC request

## Description

We use a time-smoothed congestion estimator to avoid heavy oscillation.

# xprt\_wake\_pending\_tasks

## LINUX

Kernel Hackers Manual January 2013

## Name

`xprt_wake_pending_tasks` — wake all tasks on a transport's pending queue

## Synopsis

```
void xprt_wake_pending_tasks (struct rpc_xprt * xprt, int  
status);
```

## Arguments

*xprt*

transport with waiting tasks

*status*

result code to plant in each task before waking it

## xprt\_wait\_for\_buffer\_space

### LINUX

Kernel Hackers Manual January 2013

## Name

`xprt_wait_for_buffer_space` — wait for transport output buffer to clear

## Synopsis

```
void xprt_wait_for_buffer_space (struct rpc_task * task,  
rpc_action action);
```

## Arguments

*task*

task to be put to sleep

*action*

function pointer to be executed after wait

# xprt\_write\_space

## LINUX

Kernel Hackers Manual January 2013

### Name

`xprt_write_space` — wake the task waiting for transport output buffer space

### Synopsis

```
void xprt_write_space (struct rpc_xprt * xprt);
```

### Arguments

*xprt*

transport with waiting tasks

### Description

Can be called in a soft IRQ context, so `xprt_write_space` never sleeps.

# xprt\_set\_retrans\_timeout\_def

## LINUX

Kernel Hackers Manual January 2013

### Name

`xprt_set_retrans_timeout_def` — set a request's retransmit timeout

## Synopsis

```
void xprt_set_retrans_timeout_def (struct rpc_task * task);
```

## Arguments

*task*

task whose timeout is to be set

## Description

Set a request's retransmit timeout based on the transport's default timeout parameters. Used by transports that don't adjust the retransmit timeout based on round-trip time estimation.

# xprt\_disconnect\_done

## LINUX

Kernel Hackers Manual January 2013

## Name

`xprt_disconnect_done` — mark a transport as disconnected

## Synopsis

```
void xprt_disconnect_done (struct rpc_xprt * xprt);
```



## Arguments

*xprt*

transport to flag for disconnect

## xprt\_lookup\_rqst

### LINUX

Kernel Hackers Manual January 2013

## Name

`xprt_lookup_rqst` — find an RPC request corresponding to an XID

## Synopsis

```
struct rpc_rqst * xprt_lookup_rqst (struct rpc_xprt * xprt,  
__be32 xid);
```

## Arguments

*xprt*

transport on which the original request was transmitted

*xid*

RPC XID of incoming reply

# xprt\_complete\_rqst

## LINUX

Kernel Hackers Manual January 2013

### Name

`xprt_complete_rqst` — called when reply processing is complete

### Synopsis

```
void xprt_complete_rqst (struct rpc_task * task, int copied);
```

### Arguments

*task*

RPC request that recently completed

*copied*

actual number of bytes received from the transport

### Description

Caller holds transport lock.

# rpc\_wake\_up

## LINUX

## Name

`rpc_wake_up` — wake up all `rpc_tasks`

## Synopsis

```
void rpc_wake_up (struct rpc_wait_queue * queue);
```

## Arguments

*queue*

`rpc_wait_queue` on which the tasks are sleeping

## Description

Grabs `queue->lock`

# rpc\_wake\_up\_status

## LINUX

## Name

`rpc_wake_up_status` — wake up all `rpc_tasks` and set their status value.

## Synopsis

```
void rpc_wake_up_status (struct rpc_wait_queue * queue, int  
status);
```

## Arguments

*queue*

rpc\_wait\_queue on which the tasks are sleeping

*status*

status value to set

## Description

Grabs queue->lock

# rpc\_wake\_up\_softconn\_status

## LINUX

Kernel Hackers Manual January 2013

## Name

rpc\_wake\_up\_softconn\_status — wake up all SOFTCONN rpc\_tasks and set their status value.

## Synopsis

```
void rpc_wake_up_softconn_status (struct rpc_wait_queue *  
queue, int status);
```

## Arguments

*queue*

rpc\_wait\_queue on which the tasks are sleeping

*status*

status value to set

## Description

Grabs queue->lock

# rpc\_malloc

## LINUX

Kernel Hackers Manual January 2013

## Name

rpc\_malloc — allocate an RPC buffer

## Synopsis

```
void * rpc_malloc (struct rpc_task * task, size_t size);
```

## Arguments

*task*

RPC task that will use this buffer

*size*

requested byte size

## Description

To prevent `rpciod` from hanging, this allocator never sleeps, returning `NULL` if the request cannot be serviced immediately. The caller can arrange to sleep in a way that is safe for `rpciod`.

Most requests are 'small' (under 2KiB) and can be serviced from a mempool, ensuring that NFS reads and writes can always proceed, and that there is good locality of reference for these buffers.

In order to avoid memory starvation triggering more writebacks of NFS requests, we avoid using `GFP_KERNEL`.

## rpc\_free

### LINUX

Kernel Hackers Manual January 2013

### Name

`rpc_free` — free buffer allocated via `rpc_malloc`

### Synopsis

```
void rpc_free (void * buffer);
```

## Arguments

*buffer*

buffer to free

# xdr\_skb\_read\_bits

## LINUX

Kernel Hackers Manual January 2013

## Name

`xdr_skb_read_bits` — copy some data bits from skb to internal buffer

## Synopsis

```
size_t xdr_skb_read_bits (struct xdr_skb_reader * desc, void *  
to, size_t len);
```

## Arguments

*desc*

sk\_buff copy helper

*to*

copy destination

*len*

number of bytes to copy

## Description

Possibly called several times to iterate over an `sk_buff` and copy data out of it.

# xdr\_partial\_copy\_from\_skb

## LINUX

Kernel Hackers Manual January 2013

## Name

`xdr_partial_copy_from_skb` — copy data out of an `skb`

## Synopsis

```
ssize_t xdr_partial_copy_from_skb (struct xdr_buf * xdr,
unsigned int base, struct xdr_skb_reader * desc,
xdr_skb_read_actor copy_actor);
```

## Arguments

*xdr*

target XDR buffer

*base*

starting offset

*desc*

`sk_buff` copy helper

*copy\_actor*

virtual method for copying data



# csum\_partial\_copy\_to\_xdr

## LINUX

Kernel Hackers Manual January 2013

### Name

`csum_partial_copy_to_xdr` — checksum and copy data

### Synopsis

```
int csum_partial_copy_to_xdr (struct xdr_buf * xdr, struct
sk_buff * skb);
```

### Arguments

*xdr*

target XDR buffer

*skb*

source skb

### Description

We have set things up such that we perform the checksum of the UDP packet in parallel with the copies into the RPC client iovec. -DaveM

# rpc\_alloc\_iostats

## LINUX

Kernel Hackers Manual January 2013

### Name

`rpc_alloc_iostats` — allocate an `rpc_iostats` structure

### Synopsis

```
struct rpc_iostats * rpc_alloc_iostats (struct rpc_clnt *  
clnt);
```

### Arguments

*clnt*

RPC program, version, and xprt

# rpc\_free\_iostats

## LINUX

Kernel Hackers Manual January 2013

### Name

`rpc_free_iostats` — release an `rpc_iostats` structure

## Synopsis

```
void rpc_free_iostats (struct rpc_iostats * stats);
```

## Arguments

*stats*

doomed rpc\_iostats structure

## rpc\_count\_iostats

**LINUX**

Kernel Hackers Manual January 2013

## Name

rpc\_count\_iostats — tally up per-task stats

## Synopsis

```
void rpc_count_iostats (const struct rpc_task * task, struct  
rpc_iostats * stats);
```

## Arguments

*task*

completed rpc\_task

*stats*

array of stat structures

## Description

Relies on the caller for serialization.

# rpc\_queue\_upcall

## LINUX

Kernel Hackers Manual January 2013

## Name

`rpc_queue_upcall` — queue an upcall message to userspace

## Synopsis

```
int rpc_queue_upcall (struct rpc_pipe * pipe, struct
rpc_pipe_msg * msg);
```

## Arguments

*pipe*

-- undescribed --

*msg*

message to queue

## Description

Call with an *inode* created by `rpc_mkpipe` to queue an upcall. A userspace process may then later read the upcall by performing a read on an open file for this *inode*. It is up to the caller to initialize the fields of *msg* (other than *msg->list*) appropriately.

## rpc\_mkpipe\_dentry

### LINUX

Kernel Hackers Manual January 2013

## Name

`rpc_mkpipe_dentry` — make an `rpc_pipefs` file for kernel<->userspace communication

## Synopsis

```
struct dentry * rpc_mkpipe_dentry (struct dentry * parent,
const char * name, void * private, struct rpc_pipe * pipe);
```

## Arguments

*parent*

dentry of directory to create new “pipe” in

*name*

name of pipe

*private*

private data to associate with the pipe, for the caller’s use

*pipe*

-- undescribed --

## Description

Data is made available for userspace to read by calls to `rpc_queue_upcall`. The actual reads will result in calls to `ops->upcall`, which will be called with the file pointer, message, and userspace buffer to copy to.

Writes can come at any time, and do not necessarily have to be responses to upcalls. They will result in calls to `msg->downcall`.

The *private* argument passed here will be available to all these methods from the file pointer, via `RPC_I(file->f_dentry->d_inode)->private`.

## rpc\_unlink

**LINUX**

Kernel Hackers Manual January 2013

### Name

`rpc_unlink` — remove a pipe

### Synopsis

```
int rpc_unlink (struct dentry * dentry);
```

### Arguments

*dentry*

dentry for the pipe, as returned from `rpc_mkpipe`

## Description

After this call, lookups will no longer find the pipe, and any attempts to read or write using preexisting opens of the pipe will return -EPIPE.

# rpcb\_getport\_async

## LINUX

Kernel Hackers Manual January 2013

## Name

`rpcb_getport_async` — obtain the port for a given RPC service on a given host

## Synopsis

```
void rpcb_getport_async (struct rpc_task * task);
```

## Arguments

*task*

task that is waiting for portmapper request

## Description

This one can be called for an ongoing RPC request, and can be used in an async (rpciod) context.

# rpc\_bind\_new\_program

## LINUX

Kernel Hackers Manual January 2013

### Name

`rpc_bind_new_program` — bind a new RPC program to an existing client

### Synopsis

```
struct rpc_clnt * rpc_bind_new_program (struct rpc_clnt * old,  
const struct rpc_program * program, u32 vers);
```

### Arguments

*old*

old rpc\_client

*program*

rpc program to set

*vers*

rpc program version

### Description

Clones the rpc client and sets up a new RPC program. This is mainly of use for enabling different RPC programs to share the same transport. The Sun NFSv2/v3 ACL protocol can do this.



# rpc\_run\_task

## LINUX

Kernel Hackers Manual January 2013

### Name

`rpc_run_task` — Allocate a new RPC task, then run `rpc_execute` against it

### Synopsis

```
struct rpc_task * rpc_run_task (const struct rpc_task_setup *  
task_setup_data);
```

### Arguments

*task\_setup\_data*

pointer to task initialisation data

# rpc\_call\_sync

## LINUX

Kernel Hackers Manual January 2013

### Name

`rpc_call_sync` — Perform a synchronous RPC call

## Synopsis

```
int rpc_call_sync (struct rpc_clnt * clnt, const struct  
rpc_message * msg, int flags);
```

## Arguments

*clnt*

pointer to RPC client

*msg*

RPC call parameters

*flags*

RPC call flags

## rpc\_call\_async

### LINUX

Kernel Hackers Manual January 2013

## Name

`rpc_call_async` — Perform an asynchronous RPC call

## Synopsis

```
int rpc_call_async (struct rpc_clnt * clnt, const struct  
rpc_message * msg, int flags, const struct rpc_call_ops *  
tk_ops, void * data);
```

## Arguments

*clnt*

pointer to RPC client

*msg*

RPC call parameters

*flags*

RPC call flags

*tk\_ops*

RPC call ops

*data*

user call data

## rpc\_peeraddr

### LINUX

Kernel Hackers Manual January 2013

### Name

`rpc_peeraddr` — extract remote peer address from `clnt`'s `xprt`

### Synopsis

```
size_t rpc_peeraddr (struct rpc_clnt * clnt, struct sockaddr *  
buf, size_t bufsize);
```

## Arguments

*clnt*

RPC client structure

*buf*

target buffer

*bufsize*

length of target buffer

## Description

Returns the number of bytes that are actually in the stored address.

# rpc\_peeraddr2str

## LINUX

Kernel Hackers Manual January 2013

## Name

`rpc_peeraddr2str` — return remote peer address in printable format

## Synopsis

```
const char * rpc_peeraddr2str (struct rpc_clnt * clnt, enum  
rpc_display_format_t format);
```

## Arguments

*clnt*

RPC client structure

*format*

address format

## NB

the lifetime of the memory referenced by the returned pointer is the same as the `rpc_xprt` itself. As long as the caller uses this pointer, it must hold the RCU read lock.

## rpc\_localaddr

### LINUX

Kernel Hackers Manual January 2013

## Name

`rpc_localaddr` — discover local endpoint address for an RPC client

## Synopsis

```
int rpc_localaddr (struct rpc_clnt * clnt, struct sockaddr *  
buf, size_t buflen);
```

## Arguments

*clnt*

RPC client structure

*buf*

target buffer

*buflen*

size of target buffer, in bytes

## Description

Returns zero and fills in “buf” and “buflen” if successful; otherwise, a negative errno is returned.

This works even if the underlying transport is not currently connected, or if the upper layer never previously provided a source address.

## The result of this function call is transient

multiple calls in succession may give different results, depending on how local networking configuration changes over time.

# rpc\_protocol

## LINUX

Kernel Hackers Manual January 2013

## Name

`rpc_protocol` — Get transport protocol number for an RPC client

## Synopsis

```
int rpc_protocol (struct rpc_clnt * clnt);
```

## Arguments

*clnt*

RPC client to query

## rpc\_net\_ns

### LINUX

Kernel Hackers Manual January 2013

## Name

`rpc_net_ns` — Get the network namespace for this RPC client

## Synopsis

```
struct net * rpc_net_ns (struct rpc_clnt * clnt);
```

## Arguments

*clnt*

RPC client to query

# rpc\_max\_payload

## LINUX

Kernel Hackers Manual January 2013

### Name

`rpc_max_payload` — Get maximum payload size for a transport, in bytes

### Synopsis

```
size_t rpc_max_payload (struct rpc_clnt * clnt);
```

### Arguments

*clnt*

RPC client to query

### Description

For stream transports, this is one RPC record fragment (see RFC 1831), as we don't support multi-record requests yet. For datagram transports, this is the size of an IP packet minus the IP, UDP, and RPC header sizes.

# rpc\_force\_rebind

## LINUX



## Name

`rpc_force_rebind` — force transport to check that remote port is unchanged

## Synopsis

```
void rpc_force_rebind (struct rpc_clnt * clnt);
```

## Arguments

*clnt*

client to rebind

## 1.6. WiMAX

### wimax\_msg\_alloc

#### LINUX

## Name

`wimax_msg_alloc` — Create a new skb for sending a message to userspace

## Synopsis

```
struct sk_buff * wimax_msg_alloc (struct wimax_dev *  
wimax_dev, const char * pipe_name, const void * msg, size_t  
size, gfp_t gfp_flags);
```

## Arguments

*wimax\_dev*

WiMAX device descriptor

*pipe\_name*

"named pipe" the message will be sent to

*msg*

pointer to the message data to send

*size*

size of the message to send (in bytes), including the header.

*gfp\_flags*

flags for memory allocation.

## Returns

0 if ok, negative errno code on error

## Description

Allocates an skb that will contain the message to send to user space over the messaging pipe and initializes it, copying the payload.

Once this call is done, you can deliver it with `wimax_msg_send`.

## IMPORTANT

Don't use `skb_push/skb_pull/skb_reserve` on the `skb`, as `wimax_msg_send` depends on `skb->data` being placed at the beginning of the user message.

Unlike other WiMAX stack calls, this call can be used way early, even before `wimax_dev_add` is called, as long as the `wimax_dev->net_dev` pointer is set to point to a proper `net_dev`. This is so that drivers can use it early in case they need to send stuff around or communicate with user space.

## wimax\_msg\_data\_len

### LINUX

Kernel Hackers Manual January 2013

### Name

`wimax_msg_data_len` — Return a pointer and size of a message's payload

### Synopsis

```
const void * wimax_msg_data_len (struct sk_buff * msg, size_t
* size);
```

### Arguments

*msg*

Pointer to a message created with `wimax_msg_alloc`

*size*

Pointer to where to store the message's size

## Description

Returns the pointer to the message data.

## wimax\_msg\_data

### LINUX

Kernel Hackers Manual January 2013

## Name

`wimax_msg_data` — Return a pointer to a message's payload

## Synopsis

```
const void * wimax_msg_data (struct sk_buff * msg);
```

## Arguments

*msg*

Pointer to a message created with `wimax_msg_alloc`

## wimax\_msg\_len

### LINUX

## Name

`wimax_msg_len` — Return a message's payload length

## Synopsis

```
ssize_t wimax_msg_len (struct sk_buff * msg);
```

## Arguments

*msg*

Pointer to a message created with `wimax_msg_alloc`

# wimax\_msg\_send

## LINUX

## Name

`wimax_msg_send` — Send a pre-allocated message to user space

## Synopsis

```
int wimax_msg_send (struct wimax_dev * wimax_dev, struct  
sk_buff * skb);
```

## Arguments

*wimax\_dev*

WiMAX device descriptor

*skb*

struct sk\_buff returned by `wimax_msg_alloc`. Note the ownership of *skb* is transferred to this function.

## Returns

0 if ok, < 0 errno code on error

## Description

Sends a free-form message that was preallocated with `wimax_msg_alloc` and filled up.

Assumes that once you pass an *skb* to this function for sending, it owns it and will release it when done (on success).

## IMPORTANT

Don't use `skb_push/skb_pull/skb_reserve` on the *skb*, as `wimax_msg_send` depends on `skb->data` being placed at the beginning of the user message.

Unlike other WiMAX stack calls, this call can be used way early, even before `wimax_dev_add` is called, as long as the `wimax_dev->net_dev` pointer is set to point to a proper `net_dev`. This is so that drivers can use it early in case they need to send stuff around or communicate with user space.

## wimax\_msg

**LINUX**

## Name

`wimax_msg` — Send a message to user space

## Synopsis

```
int wimax_msg (struct wimax_dev * wimax_dev, const char *  
pipe_name, const void * buf, size_t size, gfp_t gfp_flags);
```

## Arguments

*wimax\_dev*

WiMAX device descriptor (properly referenced)

*pipe\_name*

"named pipe" the message will be sent to

*buf*

pointer to the message to send.

*size*

size of the buffer pointed to by *buf* (in bytes).

*gfp\_flags*

flags for memory allocation.

## Returns

0 if ok, negative errno code on error.

## Description

Sends a free-form message to user space on the device *wimax\_dev*.

## NOTES

Once the *skb* is given to this function, who will own it and will release it when done (unless it returns error).

## wimax\_reset

### LINUX

Kernel Hackers Manual January 2013

### Name

`wimax_reset` — Reset a WiMAX device

### Synopsis

```
int wimax_reset (struct wimax_dev * wimax_dev);
```

### Arguments

*wimax\_dev*

WiMAX device descriptor

### Returns

0 if ok and a warm reset was done (the device still exists in the system).

-ENODEV if a cold/bus reset had to be done (device has disconnected and reconnected, so current handle is not valid any more).

-EINVAL if the device is not even registered.

Any other negative error code shall be considered as non-recoverable.



## Description

Called when wanting to reset the device for any reason. Device is taken back to power on status.

This call blocks; on successful return, the device has completed the reset process and is ready to operate.

# wimax\_report\_rfkill\_hw

## LINUX

Kernel Hackers Manual January 2013

## Name

`wimax_report_rfkill_hw` — Reports changes in the hardware RF switch

## Synopsis

```
void wimax_report_rfkill_hw (struct wimax_dev * wimax_dev,  
enum wimax_rf_state state);
```

## Arguments

*wimax\_dev*

WiMAX device descriptor

*state*

New state of the RF Kill switch. `WIMAX_RF_ON` radio on, `WIMAX_RF_OFF` radio off.

## Description

When the device detects a change in the state of the hardware RF switch, it must call this function to let the WiMAX kernel stack know that the state has changed so it can be properly propagated.

The WiMAX stack caches the state (the driver doesn't need to). As well, as the change is propagated it will come back as a request to change the software state to mirror the hardware state.

If the device doesn't have a hardware kill switch, just report it on initialization as always on (`WIMAX_RF_ON`, radio on).

## wimax\_report\_rfkill\_sw

### LINUX

Kernel Hackers Manual January 2013

### Name

`wimax_report_rfkill_sw` — Reports changes in the software RF switch

### Synopsis

```
void wimax_report_rfkill_sw (struct wimax_dev * wimax_dev,  
enum wimax_rf_state state);
```

### Arguments

*wimax\_dev*

WiMAX device descriptor

*state*

New state of the RF kill switch. `WIMAX_RF_ON` radio on, `WIMAX_RF_OFF` radio off.

## Description

Reports changes in the software RF switch state to the the WiMAX stack.

The main use is during initialization, so the driver can query the device for its current software radio kill switch state and feed it to the system.

On the side, the device does not change the software state by itself. In practice, this can happen, as the device might decide to switch (in software) the radio off for different reasons.

# wimax\_rfkill

## LINUX

Kernel Hackers Manual January 2013

## Name

`wimax_rfkill` — Set the software RF switch state for a WiMAX device

## Synopsis

```
int wimax_rfkill (struct wimax_dev * wimax_dev, enum
wimax_rf_state state);
```

## Arguments

*wimax\_dev*

WiMAX device descriptor

*state*

New RF state.

## Returns

$\geq 0$  toggle state if ok,  $< 0$  errno code on error. The toggle state is returned as a bitmap, bit 0 being the hardware RF state, bit 1 the software RF state.

0 means disabled (WIMAX\_RF\_ON, radio on), 1 means enabled radio off (WIMAX\_RF\_OFF).

## Description

Called by the user when he wants to request the WiMAX radio to be switched on (WIMAX\_RF\_ON) or off (WIMAX\_RF\_OFF). With WIMAX\_RF\_QUERY, just the current state is returned.

## NOTE

This call will block until the operation is complete.

# wimax\_state\_change

## LINUX

Kernel Hackers Manual January 2013

## Name

wimax\_state\_change — Set the current state of a WiMAX device

## Synopsis

```
void wimax_state_change (struct wimax_dev * wimax_dev, enum  
wimax_st new_state);
```

## Arguments

*wimax\_dev*

WiMAX device descriptor (properly referenced)

*new\_state*

New state to switch to

## Description

This implements the state changes for the wimax devices. It will

- verify that the state transition is legal (for now it'll just print a warning if not) according to the table in linux/wimax.h's documentation for 'enum wimax\_st'.
- perform the actions needed for leaving the current state and whichever are needed for entering the new state.
- issue a report to user space indicating the new state (and an optional payload with information about the new state).

## NOTE

*wimax\_dev* must be locked

## wimax\_state\_get

**LINUX**

## Name

`wimax_state_get` — Return the current state of a WiMAX device

## Synopsis

```
enum wimax_st wimax_state_get (struct wimax_dev * wimax_dev);
```

## Arguments

*wimax\_dev*

WiMAX device descriptor

## Returns

Current state of the device according to its driver.

# wimax\_dev\_init

## LINUX

## Name

`wimax_dev_init` — initialize a newly allocated instance

## Synopsis

```
void wimax_dev_init (struct wimax_dev * wimax_dev);
```

## Arguments

*wimax\_dev*

WiMAX device descriptor to initialize.

## Description

Initializes fields of a freshly allocated *wimax\_dev* instance. This function assumes that after allocation, the memory occupied by *wimax\_dev* was zeroed.

# wimax\_dev\_add

## LINUX

Kernel Hackers Manual January 2013

## Name

`wimax_dev_add` — Register a new WiMAX device

## Synopsis

```
int wimax_dev_add (struct wimax_dev * wimax_dev, struct  
net_device * net_dev);
```

## Arguments

*wimax\_dev*

WiMAX device descriptor (as embedded in your *net\_dev*'s priv data). You must have called `wimax_dev_init` on it before.

*net\_dev*

net device the *wimax\_dev* is associated with. The function expects `SET_NETDEV_DEV` and `register_netdev` were already called on it.

## Description

Registers the new WiMAX device, sets up the user-kernel control interface (generic netlink) and common WiMAX infrastructure.

Note that the parts that will allow interaction with user space are setup at the very end, when the rest is in place, as once that happens, the driver might get user space control requests via netlink or from debugfs that might translate into calls into `wimax_dev->op_*`.

## wimax\_dev\_rm

### LINUX

Kernel Hackers Manual January 2013

### Name

`wimax_dev_rm` — Unregister an existing WiMAX device

### Synopsis

```
void wimax_dev_rm (struct wimax_dev * wimax_dev);
```



## Arguments

`wimax_dev`

WiMAX device descriptor

## Description

Unregisters a WiMAX device previously registered for use with `wimax_add_rm`.

IMPORTANT! Must call before calling `unregister_netdev`.

After this function returns, you will not get any more user space control requests (via netlink or debugfs) and thus to `wimax_dev->ops`.

Reentrancy control is ensured by setting the state to `__WIMAX_ST_QUIESCING`. `rkill` operations coming through `wimax_*rfkill*()` will be stopped by the quiescing state; ops coming from the `rkill` subsystem will be stopped by the support being removed by `wimax_rfkill_rm`.

## struct wimax\_dev

### LINUX

Kernel Hackers Manual January 2013

## Name

`struct wimax_dev` — Generic WiMAX device

## Synopsis

```
struct wimax_dev {
    struct net_device * net_dev;
    struct list_head id_table_node;
    struct mutex mutex;
    struct mutex mutex_reset;
    enum wimax_st state;
    int (* op_msg_from_user) (struct wimax_dev *wimax_dev, const char *, const
    int (* op_rfkill_sw_toggle) (struct wimax_dev *wimax_dev, enum wimax_rf_s
```

```
int (* op_reset) (struct wimax_dev *wimax_dev);
struct rfkill * rfkill;
unsigned rf_hw;
unsigned rf_sw;
char name[32];
struct dentry * debugfs_dentry;
};
```

## Members

net\_dev

[fill] Pointer to the struct net\_device this WiMAX device implements.

id\_table\_node

[private] link to the list of wimax devices kept by id-table.c. Protected by it's own spinlock.

mutex

[private] Serializes all concurrent access and execution of operations.

mutex\_reset

[private] Serializes reset operations. Needs to be a different mutex because as part of the reset operation, the driver has to call back into the stack to do things such as state change, that require wimax\_dev->mutex.

state

[private] Current state of the WiMAX device.

op\_msg\_from\_user

[fill] Driver-specific operation to handle a raw message from user space to the driver. The driver can send messages to user space using with wimax\_msg\_to\_user.

op\_rfkill\_sw\_toggle

[fill] Driver-specific operation to act on userspace (or any other agent) requesting the WiMAX device to change the RF Kill software switch (WIMAX\_RF\_ON or WIMAX\_RF\_OFF). If such hardware support is not present, it is assumed the radio cannot be switched off and it is always on (and the stack will error out when trying to switch it off). In such case, this function pointer can be left as NULL.

op\_reset

[fill] Driver specific operation to reset the device. This operation should always attempt first a warm reset that does not disconnect the device from the bus and return 0. If that fails, it should resort to some sort of cold or bus reset (even if it implies a bus disconnection and device disappearance). In that case, -ENODEV should be returned to indicate the device is gone. This operation has to be synchronous, and return only when the reset is complete. In case of having had to resort to bus/cold reset implying a device disconnection, the call is allowed to return immediately.

rfkill

[private] integration into the RF-Kill infrastructure.

rf\_hw

[private] State of the hardware radio switch (OFF/ON)

rf\_sw

[private] State of the software radio switch (OFF/ON)

name[32]

[fill] A way to identify this device. We need to register a name with many subsystems (rfkill, workqueue creation, etc). We can't use the network device name as that might change and in some instances we don't know it yet (until we don't call `register_netdev`). So we generate an unique one using the driver name and device bus id, place it here and use it across the board. Recommended naming: DRIVERNAME-BUSNAME:BUSID (dev->bus->name, dev->bus\_id).

debugfs\_dentry

[private] Used to hook up a debugfs entry. This shows up in the debugfs root as wimax\:\DEVICENAME.

## NOTE

wimax\_dev->mutex is NOT locked when this op is being called; however, wimax\_dev->mutex\_reset IS locked to ensure serialization of calls to wimax\_reset. See wimax\_reset's documentation.

## Description

This structure defines a common interface to access all WiMAX devices from different vendors and provides a common API as well as a free-form device-specific messaging channel.

## Usage

1. Embed a struct `wimax_dev` at *\*the beginning\** the network device structure so that `netdev_priv` points to it.
2. `memset` it to zero
3. Initialize with `wimax_dev_init`. This will leave the WiMAX device in the `__WIMAX_ST_NULL` state.
4. Fill all the fields marked with [fill]; once called `wimax_dev_add`, those fields CANNOT be modified.
5. Call `wimax_dev_add` *\*after\** registering the network device. This will leave the WiMAX device in the `WIMAX_ST_DOWN` state. Protect the driver's `net_device->open` against succeeding if the wimax device state is lower than `WIMAX_ST_DOWN`.
6. Select when the device is going to be turned on/initialized; for example, it could be initialized on 'ifconfig up' (when the `netdev op 'open'` is called on the driver).

When the device is initialized (at 'ifconfig up' time, or right after calling `wimax_dev_add` from `_probe`, make sure the following steps are taken

- a. Move the device to `WIMAX_ST_UNINITIALIZED`. This is needed so some API calls that shouldn't work until the device is ready can be blocked.
- b. Initialize the device. Make sure to turn the SW radio switch off and move the device to state `WIMAX_ST_RADIO_OFF` when done. When just initialized, a device should be left in RADIO OFF state until user space devices to turn it on.
- c. Query the device for the state of the hardware rfkill switch and call `wimax_rfkill_report_hw` and `wimax_rfkill_report_sw` as needed. See below.

`wimax_dev_rm` undoes before unregistering the network device. Once `wimax_dev_add` is called, the driver can get called on the `wimax_dev->op_*` function pointers

## CONCURRENCY

The stack provides a mutex for each device that will disallow API calls happening concurrently; thus, op calls into the driver through the `wimax_dev->op*()` function pointers will always be serialized and *\*never\** concurrent.

For locking, take `wimax_dev->mutex` is taken; (most) operations in the API have to check for `wimax_dev_is_ready` to return 0 before continuing (this is done internally).

## REFERENCE COUNTING

The WiMAX device is reference counted by the associated network device. The only operation that can be used to reference the device is `wimax_dev_get_by_genl_info`, and the reference it acquires has to be released with `dev_put(wimax_dev->net_dev)`.

## RFKILL

At startup, both HW and SW radio switchess are assumed to be off.

At initialization time [after calling `wimax_dev_add`], have the driver query the device for the status of the software and hardware RF kill switches and call `wimax_report_rfkill_hw` and `wimax_rfkill_report_sw` to indicate their state. If any is missing, just call it to indicate it is ON (radio always on).

Whenever the driver detects a change in the state of the RF kill switches, it should call `wimax_report_rfkill_hw` or `wimax_report_rfkill_sw` to report it to the stack.

## enum wimax\_st

### LINUX

Kernel Hackers Manual January 2013

### Name

`enum wimax_st` — The different states of a WiMAX device

## Synopsis

```
enum wimax_st {
    __WIMAX_ST_NULL,
    WIMAX_ST_DOWN,
    __WIMAX_ST QUIESCING,
    WIMAX_ST_UNINITIALIZED,
    WIMAX_ST_RADIO_OFF,
    WIMAX_ST_READY,
    WIMAX_ST_SCANNING,
    WIMAX_ST_CONNECTING,
    WIMAX_ST_CONNECTED,
    __WIMAX_ST_INVALID
};
```

## Constants

### `__WIMAX_ST_NULL`

The device structure has been allocated and zeroed, but still `wimax_dev_add` hasn't been called. There is no state.

### `WIMAX_ST_DOWN`

The device has been registered with the WiMAX and networking stacks, but it is not initialized (normally that is done with 'ifconfig DEV up' [or equivalent], which can upload firmware and enable communications with the device). In this state, the device is powered down and using as less power as possible. This state is the default after a call to `wimax_dev_add`. It is ok to have drivers move directly to `WIMAX_ST_UNINITIALIZED` or `WIMAX_ST_RADIO_OFF` in `_probe` after the call to `wimax_dev_add`. It is recommended that the driver leaves this state when calling 'ifconfig DEV up' and enters it back on 'ifconfig DEV down'.

### `__WIMAX_ST QUIESCING`

The device is being torn down, so no API operations are allowed to proceed except the ones needed to complete the device clean up process.

### `WIMAX_ST_UNINITIALIZED`

[optional] Communication with the device is setup, but the device still requires some configuration before being operational. Some WiMAX API calls might work.

#### WIMAX\_ST\_RADIO\_OFF

The device is fully up; radio is off (wether by hardware or software switches). It is recommended to always leave the device in this state after initialization.

#### WIMAX\_ST\_READY

The device is fully up and radio is on.

#### WIMAX\_ST\_SCANNING

[optional] The device has been instructed to scan. In this state, the device cannot be actively connected to a network.

#### WIMAX\_ST\_CONNECTING

The device is connecting to a network. This state exists because in some devices, the connect process can include a number of negotiations between user space, kernel space and the device. User space needs to know what the device is doing. If the connect sequence in a device is atomic and fast, the device can transition directly to CONNECTED

#### WIMAX\_ST\_CONNECTED

The device is connected to a network.

#### \_\_WIMAX\_ST\_INVALID

This is an invalid state used to mark the maximum numeric value of states.

## Description

Transitions from one state to another one are atomic and can only be caused in kernel space with `wimax_state_change`. To read the state, use `wimax_state_get`.

States starting with `__` are internal and shall not be used or referred to by drivers or userspace. They look ugly, but that's the point -- if any use is made non-internal to the stack, it is easier to catch on review.

All API operations [with well defined exceptions] will take the device mutex before starting and then check the state. If the state is `__WIMAX_ST_NULL`, `WIMAX_ST_DOWN`, `WIMAX_ST_UNINITIALIZED` or `__WIMAX_ST QUIESCING`, it will drop the lock and quit with `-EINVAL`, `-ENOMEDIUM`, `-ENOTCONN` or `-ESHUTDOWN`.

The order of the definitions is important, so we can do numerical comparisons (eg: `< WIMAX_ST_RADIO_OFF` means the device is not ready to operate).





# Chapter 2. Network device support

## 2.1. Driver Support

### dev\_add\_pack

#### LINUX

Kernel Hackers Manual January 2013

#### Name

dev\_add\_pack — add packet handler

#### Synopsis

```
void dev_add_pack (struct packet_type * pt);
```

#### Arguments

*pt*

packet type declaration

#### Description

Add a protocol handler to the networking stack. The passed `packet_type` is linked into kernel lists and may not be freed until it has been removed from the kernel lists.

This call does not sleep therefore it can not guarantee all CPU's that are in middle of receiving packets will see the new packet type (until the next received packet).

# \_\_dev\_remove\_pack

## LINUX

Kernel Hackers Manual January 2013

### Name

`__dev_remove_pack` — remove packet handler

### Synopsis

```
void __dev_remove_pack (struct packet_type * pt);
```

### Arguments

*pt*

packet type declaration

### Description

Remove a protocol handler that was previously added to the kernel protocol handlers by `dev_add_pack`. The passed `packet_type` is removed from the kernel lists and can be freed or reused once this function returns.

The packet type might still be in use by receivers and must not be freed until after all the CPU's have gone through a quiescent state.

# dev\_remove\_pack

## LINUX

## Name

`dev_remove_pack` — remove packet handler

## Synopsis

```
void dev_remove_pack (struct packet_type * pt);
```

## Arguments

*pt*

packet type declaration

## Description

Remove a protocol handler that was previously added to the kernel protocol handlers by `dev_add_pack`. The passed `packet_type` is removed from the kernel lists and can be freed or reused once this function returns.

This call sleeps to guarantee that no CPU is looking at the packet type after return.

# netdev\_boot\_setup\_check

## LINUX

## Name

`netdev_boot_setup_check` — check boot time settings

## Synopsis

```
int netdev_boot_setup_check (struct net_device * dev);
```

## Arguments

*dev*

the netdevice

## Description

Check boot time settings for the device. The found settings are set for the device to be used later in the device probing. Returns 0 if no settings found, 1 if they are.

## \_\_dev\_get\_by\_name

### LINUX

Kernel Hackers Manual January 2013

## Name

`__dev_get_by_name` — find a device by its name

## Synopsis

```
struct net_device * __dev_get_by_name (struct net * net, const  
char * name);
```

## Arguments

*net*

the applicable net namespace

*name*

name to find

## Description

Find an interface by name. Must be called under RTNL semaphore or *dev\_base\_lock*. If the name is found a pointer to the device is returned. If the name is not found then `NULL` is returned. The reference counters are not incremented so the caller must be careful with locks.

## dev\_get\_by\_name\_rcu

### LINUX

Kernel Hackers Manual January 2013

## Name

`dev_get_by_name_rcu` — find a device by its name

## Synopsis

```
struct net_device * dev_get_by_name_rcu (struct net * net,
const char * name);
```

## Arguments

*net*

the applicable net namespace

*name*

name to find

## Description

Find an interface by name. If the name is found a pointer to the device is returned. If the name is not found then `NULL` is returned. The reference counters are not incremented so the caller must be careful with locks. The caller must hold RCU lock.

# dev\_get\_by\_name

## LINUX

Kernel Hackers Manual January 2013

## Name

`dev_get_by_name` — find a device by its name

## Synopsis

```
struct net_device * dev_get_by_name (struct net * net, const  
char * name);
```

## Arguments

*net*

the applicable net namespace

*name*

name to find

## Description

Find an interface by name. This can be called from any context and does its own locking. The returned handle has the usage count incremented and the caller must use `dev_put` to release it when it is no longer needed. `NULL` is returned if no matching device is found.

## \_\_dev\_get\_by\_index

### LINUX

Kernel Hackers Manual January 2013

## Name

`__dev_get_by_index` — find a device by its ifindex

## Synopsis

```
struct net_device * __dev_get_by_index (struct net * net, int
ifindex);
```

## Arguments

*net*

the applicable net namespace

*ifindex*

index of device

## Description

Search for an interface by index. Returns `NULL` if the device is not found or a pointer to the device. The device has not had its reference counter increased so the caller must be careful about locking. The caller must hold either the RTNL semaphore or *dev\_base\_lock*.

# dev\_get\_by\_index\_rcu

## LINUX

Kernel Hackers Manual January 2013

## Name

`dev_get_by_index_rcu` — find a device by its `ifindex`

## Synopsis

```
struct net_device * dev_get_by_index_rcu (struct net * net,  
int ifindex);
```



## Arguments

*net*

the applicable net namespace

*ifindex*

index of device

## Description

Search for an interface by index. Returns `NULL` if the device is not found or a pointer to the device. The device has not had its reference counter increased so the caller must be careful about locking. The caller must hold RCU lock.

## dev\_get\_by\_index

### LINUX

Kernel Hackers Manual January 2013

## Name

`dev_get_by_index` — find a device by its `ifindex`

## Synopsis

```
struct net_device * dev_get_by_index (struct net * net, int
ifindex);
```

## Arguments

*net*

the applicable net namespace

*ifindex*

index of device

## Description

Search for an interface by index. Returns NULL if the device is not found or a pointer to the device. The device returned has had a reference added and the pointer is safe until the user calls `dev_put` to indicate they have finished with it.

# dev\_getbyhwaddr\_rcu

## LINUX

Kernel Hackers Manual January 2013

## Name

`dev_getbyhwaddr_rcu` — find a device by its hardware address

## Synopsis

```
struct net_device * dev_getbyhwaddr_rcu (struct net * net,  
unsigned short type, const char * ha);
```

## Arguments

*net*

the applicable net namespace

*type*

media type of device

*ha*

hardware address

## Description

Search for an interface by MAC address. Returns NULL if the device is not found or a pointer to the device. The caller must hold RCU or RTNL. The returned device has not had its ref count increased and the caller must therefore be careful about locking

# dev\_get\_by\_flags\_rcu

## LINUX

Kernel Hackers Manual January 2013

## Name

`dev_get_by_flags_rcu` — find any device with given flags

## Synopsis

```
struct net_device * dev_get_by_flags_rcu (struct net * net,  
unsigned short if_flags, unsigned short mask);
```

## Arguments

*net*

the applicable net namespace

*if\_flags*

IFF\_\* values

*mask*

bitmask of bits in *if\_flags* to check

## Description

Search for any interface with the given flags. Returns NULL if a device is not found or a pointer to the device. Must be called inside `rcu_read_lock`, and result refcount is unchanged.

## dev\_valid\_name

### LINUX

Kernel Hackers Manual January 2013

## Name

`dev_valid_name` — check if name is okay for network device

## Synopsis

```
bool dev_valid_name (const char * name);
```

## Arguments

*name*

name string

## Description

Network device names need to be valid file names to to allow sysfs to work. We also disallow any kind of whitespace.

# dev\_alloc\_name

## LINUX

Kernel Hackers Manual January 2013

## Name

dev\_alloc\_name — allocate a name for a device

## Synopsis

```
int dev_alloc_name (struct net_device * dev, const char *  
name);
```

## Arguments

*dev*

device

*name*

name format string

## Description

Passed a format string - eg “%d” it will try and find a suitable id. It scans list of devices to build up a free map, then chooses the first empty slot. The caller must hold the dev\_base or rtnl lock while allocating the name and adding the device in order to avoid duplicates. Limited to bits\_per\_byte \* page size devices (ie 32K on most platforms). Returns the number of the unit assigned or a negative errno code.

## netdev\_features\_change

### LINUX

Kernel Hackers Manual January 2013

## Name

`netdev_features_change` — device changes features

## Synopsis

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

## Arguments

*dev*

device to cause notification

## Description

Called to indicate a device has changed features.

# netdev\_state\_change

## LINUX

Kernel Hackers Manual January 2013

### Name

netdev\_state\_change — device changes state

### Synopsis

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

### Arguments

*dev*

device to cause notification

### Description

Called to indicate a device has changed state. This function calls the notifier chains for netdev\_chain and sends a NEWLINK message to the routing socket.

# dev\_load

## LINUX

## Name

`dev_load` — load a network module

## Synopsis

```
void dev_load (struct net * net, const char * name);
```

## Arguments

*net*

the applicable net namespace

*name*

name of interface

## Description

If a network interface is not present and the process has suitable privileges this function loads the module. If module loading is not available in this kernel then it becomes a nop.

## dev\_open

**LINUX**



## Name

`dev_open` — prepare an interface for use.

## Synopsis

```
int dev_open (struct net_device * dev);
```

## Arguments

*dev*

device to open

## Description

Takes a device from down to up state. The device's private open function is invoked and then the multicast lists are loaded. Finally the device is moved into the up state and a `NETDEV_UP` message is sent to the netdev notifier chain.

Calling this function on an active interface is a nop. On a failure a negative errno code is returned.

## `dev_close`

### LINUX

## Name

`dev_close` — shutdown an interface.

## Synopsis

```
int dev_close (struct net_device * dev);
```

## Arguments

*dev*

device to shutdown

## Description

This function moves an active device into down state. A `NETDEV_GOING_DOWN` is sent to the netdev notifier chain. The device is then deactivated and finally a `NETDEV_DOWN` is sent to the notifier chain.

## dev\_disable\_lro

### LINUX

Kernel Hackers Manual January 2013

## Name

`dev_disable_lro` — disable Large Receive Offload on a device

## Synopsis

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

## Arguments

*dev*

device

## Description

Disable Large Receive Offload (LRO) on a net device. Must be called under RTNL. This is needed if received packets may be forwarded to another interface.

# register\_netdevice\_notifier

## LINUX

Kernel Hackers Manual January 2013

## Name

`register_netdevice_notifier` — register a network notifier block

## Synopsis

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

## Arguments

*nb*

notifier

## Description

Register a notifier to be called when network device events occur. The notifier passed is linked into the kernel structures and must not be reused until it has been unregistered. A negative errno code is returned on a failure.

When registered all registration and up events are replayed to the new notifier to allow device to have a race free view of the network device list.

# unregister\_netdevice\_notifier

## LINUX

Kernel Hackers Manual January 2013

## Name

`unregister_netdevice_notifier` — unregister a network notifier block

## Synopsis

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

## Arguments

*nb*

notifier

## Description

Unregister a notifier previously registered by `register_netdevice_notifier`. The notifier is unlinked into the kernel structures and may then be reused. A negative errno code is returned on a failure.

After unregistering unregister and down device events are synthesized for all devices on the device list to the removed notifier to remove the need for special case cleanup code.

## call\_netdevice\_notifiers

### LINUX

Kernel Hackers Manual January 2013

### Name

`call_netdevice_notifiers` — call all network notifier blocks

### Synopsis

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

### Arguments

*val*

value passed unmodified to notifier function

*dev*

net\_device pointer passed unmodified to notifier function

### Description

Call all network notifier blocks. Parameters and return value are as for `raw_notifier_call_chain`.

# dev\_forward\_skb

## LINUX

Kernel Hackers Manual January 2013

### Name

`dev_forward_skb` — loopback an skb to another netif

### Synopsis

```
int dev_forward_skb (struct net_device * dev, struct sk_buff *  
skb);
```

### Arguments

*dev*

destination network device

*skb*

buffer to forward

### return values

NET\_RX\_SUCCESS (no congestion) NET\_RX\_DROP (packet was dropped, but freed)

`dev_forward_skb` can be used for injecting an skb from the `start_xmit` function of one device into the receive queue of another device.

The receiving device may be in another namespace, so we have to clear all information in the skb that could impact namespace isolation.

# netif\_set\_real\_num\_rx\_queues

## LINUX

Kernel Hackers Manual January 2013

### Name

`netif_set_real_num_rx_queues` — set actual number of RX queues used

### Synopsis

```
int netif_set_real_num_rx_queues (struct net_device * dev,  
unsigned int rxq);
```

### Arguments

*dev*

Network device

*rxq*

Actual number of RX queues

### Description

This must be called either with the `rtnl_lock` held or before registration of the net device. Returns 0 on success, or a negative error code. If called before registration, it always succeeds.

# netif\_device\_detach

## LINUX

Kernel Hackers Manual January 2013

### Name

`netif_device_detach` — mark device as removed

### Synopsis

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

### Arguments

*dev*

network device

### Description

Mark device as removed from system and therefore no longer available.

# netif\_device\_attach

## LINUX

Kernel Hackers Manual January 2013

### Name

`netif_device_attach` — mark device as attached



## Synopsis

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

## Arguments

*dev*

network device

## Description

Mark device as attached from system and restart if needed.

# skb\_gso\_segment

## LINUX

Kernel Hackers Manual January 2013

## Name

`skb_gso_segment` — Perform segmentation on `skb`.

## Synopsis

```
struct sk_buff * skb_gso_segment (struct sk_buff * skb,  
netdev_features_t features);
```

## Arguments

*skb*

buffer to segment

*features*

features for the output path (see dev->features)

## Description

This function segments the given skb and returns a list of segments.

It may return NULL if the skb requires no segmentation. This is only possible when GSO is used for verifying header integrity.

# dev\_queue\_xmit

## LINUX

Kernel Hackers Manual January 2013

## Name

dev\_queue\_xmit — transmit a buffer

## Synopsis

```
int dev_queue_xmit (struct sk_buff * skb);
```

## Arguments

*skb*

buffer to transmit

## Description

Queue a buffer for transmission to a network device. The caller must have set the device and priority and built the buffer before calling this function. The function can be called from an interrupt.

A negative errno code is returned on a failure. A success does not guarantee the frame will be transmitted as it may be dropped due to congestion or traffic shaping.

----- I notice this method can also return errors from the queue disciplines, including NET\_XMIT\_DROP, which is a positive value. So, errors can also be positive.

Regardless of the return value, the skb is consumed, so it is currently difficult to retry a send to this method. (You can bump the ref count before sending to hold a reference for retry if you are careful.)

When calling this method, interrupts MUST be enabled. This is because the BH enable code must have IRQs enabled so that it will not deadlock. --BLG

## rps\_may\_expire\_flow

### LINUX

Kernel Hackers Manual January 2013

### Name

`rps_may_expire_flow` — check whether an RFS hardware filter may be removed

## Synopsis

```
bool rps_may_expire_flow (struct net_device * dev, u16
rxq_index, u32 flow_id, u16 filter_id);
```

## Arguments

*dev*

Device on which the filter was set

*rxq\_index*

RX queue index

*flow\_id*

Flow ID passed to `ndo_rx_flow_steer`

*filter\_id*

Filter ID returned by `ndo_rx_flow_steer`

## Description

Drivers that implement `ndo_rx_flow_steer` should periodically call this function for each installed filter and remove the filters for which it returns `true`.

## netif\_rx

### LINUX

Kernel Hackers Manual January 2013

## Name

`netif_rx` — post buffer to the network code

## Synopsis

```
int netif_rx (struct sk_buff * skb);
```

## Arguments

*skb*

buffer to post

## Description

This function receives a packet from a device driver and queues it for the upper (protocol) levels to process. It always succeeds. The buffer may be dropped during processing for congestion control or by the protocol layers.

## return values

NET\_RX\_SUCCESS (no congestion) NET\_RX\_DROP (packet was dropped)

# netdev\_rx\_handler\_register

## LINUX

Kernel Hackers Manual January 2013

## Name

`netdev_rx_handler_register` — register receive handler

## Synopsis

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

## Arguments

*dev*

device to register a handler for

*rx\_handler*

receive handler to register

*rx\_handler\_data*

data pointer that is used by rx handler

## Description

Register a receive handler for a device. This handler will then be called from `__netif_receive_skb`. A negative errno code is returned on a failure.

The caller must hold the `rtnl_mutex`.

For a general description of `rx_handler`, see enum `rx_handler_result`.

# netdev\_rx\_handler\_unregister

## LINUX

Kernel Hackers Manual January 2013

## Name

`netdev_rx_handler_unregister` — unregister receive handler

## Synopsis

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

## Arguments

*dev*

device to unregister a handler from

## Description

Unregister a receive handler from a device.

The caller must hold the `rtnl_mutex`.

# netif\_receive\_skb

## LINUX

Kernel Hackers Manual January 2013

## Name

`netif_receive_skb` — process receive buffer from network

## Synopsis

```
int netif_receive_skb (struct sk_buff * skb);
```

## Arguments

*skb*

buffer to process

## Description

`netif_receive_skb` is the main receive data processing function. It always succeeds. The buffer may be dropped during processing for congestion control or by the protocol layers.

This function may only be called from softirq context and interrupts should be enabled.

Return values (usually ignored):

## NET\_RX\_SUCCESS

no congestion

## NET\_RX\_DROP

packet was dropped

## \_\_napi\_schedule

### LINUX

Kernel Hackers Manual January 2013

## Name

`__napi_schedule` — schedule for receive



## Synopsis

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

## Arguments

*n*

entry to schedule

## Description

The entry's receive function will be scheduled to run

# register\_gifconf

## LINUX

Kernel Hackers Manual January 2013

## Name

register\_gifconf — register a SIOCGIF handler

## Synopsis

```
int register_gifconf (unsigned int family, gifconf_func_t *  
gifconf);
```

## Arguments

*family*

Address family

*gifconf*

Function handler

## Description

Register protocol dependent address dumping routines. The handler that is passed must not be freed or reused until it has been replaced by another handler.

# netdev\_set\_master

## LINUX

Kernel Hackers Manual January 2013

## Name

`netdev_set_master` — set up master pointer

## Synopsis

```
int netdev_set_master (struct net_device * slave, struct  
net_device * master);
```

## Arguments

*slave*

slave device

*master*

new master device

## Description

Changes the master device of the slave. Pass `NULL` to break the bonding. The caller must hold the RTNL semaphore. On a failure a negative `errno` code is returned. On success the reference counts are adjusted and the function returns zero.

# netdev\_set\_bond\_master

## LINUX

Kernel Hackers Manual January 2013

## Name

`netdev_set_bond_master` — set up bonding master/slave pair

## Synopsis

```
int netdev_set_bond_master (struct net_device * slave, struct
net_device * master);
```

## Arguments

*slave*

slave device

*master*

new master device

## Description

Changes the master device of the slave. Pass `NULL` to break the bonding. The caller must hold the RTNL semaphore. On a failure a negative errno code is returned. On success `RTM_NEWLINK` is sent to the routing socket and the function returns zero.

# dev\_set\_promiscuity

## LINUX

Kernel Hackers Manual January 2013

## Name

`dev_set_promiscuity` — update promiscuity count on a device

## Synopsis

```
int dev_set_promiscuity (struct net_device * dev, int inc);
```

## Arguments

*dev*

device

*inc*

modifier

## Description

Add or remove promiscuity from a device. While the count in the device remains above zero the interface remains promiscuous. Once it hits zero the device reverts

back to normal filtering operation. A negative *inc* value is used to drop promiscuity on the device. Return 0 if successful or a negative *errno* code on error.

## dev\_set\_allmulti

### LINUX

Kernel Hackers Manual January 2013

### Name

`dev_set_allmulti` — update allmulti count on a device

### Synopsis

```
int dev_set_allmulti (struct net_device * dev, int inc);
```

### Arguments

*dev*

device

*inc*

modifier

### Description

Add or remove reception of all multicast frames to a device. While the count in the device remains above zero the interface remains listening to all interfaces. Once it hits zero the device reverts back to normal filtering operation. A negative *inc* value is used to drop the counter when releasing a resource needing all multicasts. Return 0 if successful or a negative *errno* code on error.

# dev\_get\_flags

## LINUX

Kernel Hackers Manual January 2013

### Name

`dev_get_flags` — get flags reported to userspace

### Synopsis

```
unsigned dev_get_flags (const struct net_device * dev);
```

### Arguments

*dev*

device

### Description

Get the combination of flag bits exported through APIs to userspace.

# dev\_change\_flags

## LINUX

## Name

`dev_change_flags` — change device settings

## Synopsis

```
int dev_change_flags (struct net_device * dev, unsigned int  
    flags);
```

## Arguments

*dev*

device

*flags*

device state flags

## Description

Change settings on device based state flags. The flags are in the userspace exported format.

## `dev_set_mtu`

**LINUX**

## Name

`dev_set_mtu` — Change maximum transfer unit

## Synopsis

```
int dev_set_mtu (struct net_device * dev, int new_mtu);
```

## Arguments

*dev*

device

*new\_mtu*

new transfer unit

## Description

Change the maximum transfer size of the network device.

# dev\_set\_group

## LINUX

## Name

`dev_set_group` — Change group this device belongs to



## Synopsis

```
void dev_set_group (struct net_device * dev, int new_group);
```

## Arguments

*dev*

device

*new\_group*

group this device should belong to

## dev\_set\_mac\_address

### LINUX

Kernel Hackers Manual January 2013

## Name

`dev_set_mac_address` — Change Media Access Control Address

## Synopsis

```
int dev_set_mac_address (struct net_device * dev, struct  
sockaddr * sa);
```

## Arguments

*dev*

device

*sa*

new address

## Description

Change the hardware (MAC) address of the device

# netdev\_update\_features

## LINUX

Kernel Hackers Manual January 2013

## Name

`netdev_update_features` — recalculate device features

## Synopsis

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

## Arguments

*dev*

the device to check

## Description

Recalculate dev->features set and send notifications if it has changed. Should be called after driver or hardware dependent conditions might have changed that influence the features.

# netdev\_change\_features

## LINUX

Kernel Hackers Manual January 2013

## Name

netdev\_change\_features — recalculate device features

## Synopsis

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

## Arguments

*dev*

the device to check

## Description

Recalculate dev->features set and send notifications even if they have not changed. Should be called instead of netdev\_update\_features if also dev->vlan\_features might have changed to allow the changes to be propagated to stacked VLAN devices.

# netif\_stacked\_transfer\_operstate

## LINUX

Kernel Hackers Manual January 2013

### Name

`netif_stacked_transfer_operstate` — transfer operstate

### Synopsis

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

### Arguments

*rootdev*

the root or lower level device to transfer state from

*dev*

the device to transfer operstate to

### Description

Transfer operational state from root to device. This is normally called when a stacking relationship exists between the root device and the device(a leaf device).

# register\_netdevice

## LINUX

## Name

`register_netdevice` — register a network device

## Synopsis

```
int register_netdevice (struct net_device * dev);
```

## Arguments

*dev*

device to register

## Description

Take a completed network device structure and add it to the kernel interfaces. A `NETDEV_REGISTER` message is sent to the netdev notifier chain. 0 is returned on success. A negative errno code is returned on a failure to set up the device, or if the name is a duplicate.

Callers must hold the `rtnl` semaphore. You may want `register_netdev` instead of this.

## BUGS

The locking appears insufficient to guarantee two parallel registers will not get the same name.

# init\_dummy\_netdev

## LINUX

Kernel Hackers Manual January 2013

### Name

`init_dummy_netdev` — init a dummy network device for NAPI

### Synopsis

```
int init_dummy_netdev (struct net_device * dev);
```

### Arguments

*dev*

device to init

### Description

This takes a network device structure and initialize the minimum amount of fields so it can be used to schedule NAPI polls without registering a full blown interface. This is to be used by drivers that need to tie several hardware interfaces to a single NAPI poll scheduler due to HW limitations.

# register\_netdev

## LINUX

## Name

`register_netdev` — register a network device

## Synopsis

```
int register_netdev (struct net_device * dev);
```

## Arguments

*dev*

device to register

## Description

Take a completed network device structure and add it to the kernel interfaces. A `NETDEV_REGISTER` message is sent to the netdev notifier chain. 0 is returned on success. A negative errno code is returned on a failure to set up the device, or if the name is a duplicate.

This is a wrapper around `register_netdevice` that takes the `rtnl` semaphore and expands the device name if you passed a format string to `alloc_netdev`.

## dev\_get\_stats

**LINUX**

## Name

`dev_get_stats` — get network device statistics

## Synopsis

```
struct rtnl_link_stats64 * dev_get_stats (struct net_device *  
dev, struct rtnl_link_stats64 * storage);
```

## Arguments

*dev*

device to get statistics from

*storage*

place to store stats

## Description

Get network statistics from device. Return *storage*. The device driver may provide its own method by setting `dev->netdev_ops->get_stats64` or `dev->netdev_ops->get_stats`; otherwise the internal statistics structure is used.

## `alloc_netdev_mqs`

**LINUX**



## Name

`alloc_netdev_mqs` — allocate network device

## Synopsis

```
struct net_device * alloc_netdev_mqs (int sizeof_priv, const  
char * name, void (*setup) (struct net_device *), unsigned int  
txqs, unsigned int rxqs);
```

## Arguments

*sizeof\_priv*

size of private data to allocate space for

*name*

device name format string

*setup*

callback to initialize device

*txqs*

the number of TX subqueues to allocate

*rxqs*

the number of RX subqueues to allocate

## Description

Allocates a struct `net_device` with private data area for driver use and performs basic initialization. Also allocates subqueue structs for each queue on the device.

# free\_netdev

## LINUX

Kernel Hackers Manual January 2013

### Name

`free_netdev` — free network device

### Synopsis

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

### Arguments

*dev*

device

### Description

This function does the last stage of destroying an allocated device interface. The reference to the device object is released. If this is the last reference then it will be freed.

# synchronize\_net

## LINUX

## Name

`synchronize_net` — Synchronize with packet receive processing

## Synopsis

```
void synchronize_net ( void );
```

## Arguments

*void*

no arguments

## Description

Wait for packets currently being received to be done. Does not block later packets from starting.

# unregister\_netdevice\_queue

## LINUX

## Name

`unregister_netdevice_queue` — remove device from the kernel

## Synopsis

```
void unregister_netdevice_queue (struct net_device * dev,  
struct list_head * head);
```

## Arguments

*dev*

device

*head*

list

## Description

This function shuts down a device interface and removes it from the kernel tables. If head not NULL, device is queued to be unregistered later.

Callers must hold the rtnl semaphore. You may want `unregister_netdev` instead of this.

# unregister\_netdevice\_many

## LINUX

Kernel Hackers Manual January 2013

## Name

`unregister_netdevice_many` — unregister many devices

## Synopsis

```
void unregister_netdevice_many (struct list_head * head);
```

## Arguments

*head*

list of devices

# unregister\_netdev

## LINUX

Kernel Hackers Manual January 2013

## Name

`unregister_netdev` — remove device from the kernel

## Synopsis

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

## Arguments

*dev*

device

## Description

This function shuts down a device interface and removes it from the kernel tables.

This is just a wrapper for `unregister_netdevice` that takes the `rtnl` semaphore. In general you want to use this and not `unregister_netdevice`.

# dev\_change\_net\_namespace

## LINUX

Kernel Hackers Manual January 2013

## Name

`dev_change_net_namespace` — move device to different nethost namespace

## Synopsis

```
int dev_change_net_namespace (struct net_device * dev, struct
net * net, const char * pat);
```

## Arguments

*dev*

device

*net*

network namespace

*pat*

If not NULL name pattern to try if the current device name is already taken in the destination network namespace.

## Description

This function shuts down a device interface and moves it to a new network namespace. On success 0 is returned, on a failure a netagive errno code is returned.

Callers must hold the rtnl semaphore.

## netdev\_increment\_features

### LINUX

Kernel Hackers Manual January 2013

### Name

`netdev_increment_features` — increment feature set by one

### Synopsis

```
netdev_features_t netdev_increment_features (netdev_features_t
all, netdev_features_t one, netdev_features_t mask);
```

### Arguments

*all*

current feature set

*one*

new feature set

*mask*

mask feature set

## Description

Computes a new feature set after adding a device with feature set *one* to the master device with current feature set *all*. Will not enable anything that is off in *mask*. Returns the new feature set.

# eth\_header

## LINUX

Kernel Hackers Manual January 2013

## Name

`eth_header` — create the Ethernet header

## Synopsis

```
int eth_header (struct sk_buff * skb, struct net_device * dev,
unsigned short type, const void * daddr, const void * saddr,
unsigned len);
```

## Arguments

*skb*

buffer to alter

*dev*

source device

*type*

Ethernet type field



*daddr*

destination address (NULL leave destination address)

*saddr*

source address (NULL use device source address)

*len*

packet length ( $\leq$  `skb->len`)

## Description

Set the protocol type. For a packet of type `ETH_P_802_3/2` we put the length in here instead.

# eth\_rebuild\_header

## LINUX

Kernel Hackers Manual January 2013

## Name

`eth_rebuild_header` — rebuild the Ethernet MAC header.

## Synopsis

```
int eth_rebuild_header (struct sk_buff * skb);
```

## Arguments

*skb*

socket buffer to update

## Description

This is called after an ARP or IPV6 ndisc it's resolution on this sk\_buff. We now let protocol (ARP) fill in the other fields.

This routine CANNOT use cached dst->neigh! Really, it is used only when dst->neigh is wrong.

# eth\_type\_trans

## LINUX

Kernel Hackers Manual January 2013

## Name

`eth_type_trans` — determine the packet's protocol ID.

## Synopsis

```
__be16 eth_type_trans (struct sk_buff * skb, struct net_device  
* dev);
```

## Arguments

*skb*

received socket data

*dev*

receiving network device

## Description

The rule here is that we assume 802.3 if the type field is short enough to be a length. This is normal practice and works for any 'now in use' protocol.

# eth\_header\_parse

## LINUX

Kernel Hackers Manual January 2013

## Name

`eth_header_parse` — extract hardware address from packet

## Synopsis

```
int eth_header_parse (const struct sk_buff * skb, unsigned
char * haddr);
```

## Arguments

*skb*

packet to extract header from

*haddr*

destination buffer

# eth\_header\_cache

## LINUX

Kernel Hackers Manual January 2013

### Name

`eth_header_cache` — fill cache entry from neighbour

### Synopsis

```
int eth_header_cache (const struct neighbour * neigh, struct  
hh_cache * hh, __be16 type);
```

### Arguments

*neigh*

source neighbour

*hh*

destination cache entry

*type*

Ethernet type field Create an Ethernet header template from the neighbour.

# eth\_header\_cache\_update

## LINUX

## Name

`eth_header_cache_update` — update cache entry

## Synopsis

```
void eth_header_cache_update (struct hh_cache * hh, const  
struct net_device * dev, const unsigned char * haddr);
```

## Arguments

*hh*

destination cache entry

*dev*

network device

*haddr*

new hardware address

## Description

Called by Address Resolution module to notify changes in address.

## `eth_mac_addr`

**LINUX**

## Name

`eth_mac_addr` — set new Ethernet hardware address

## Synopsis

```
int eth_mac_addr (struct net_device * dev, void * p);
```

## Arguments

*dev*

network device

*p*

socket address Change hardware address of device.

## Description

This doesn't change hardware matching, so needs to be overridden for most real devices.

# eth\_change\_mtu

## LINUX

## Name

`eth_change_mtu` — set new MTU size

## Synopsis

```
int eth_change_mtu (struct net_device * dev, int new_mtu);
```

## Arguments

*dev*

network device

*new\_mtu*

new Maximum Transfer Unit

## Description

Allow changing MTU size. Needs to be overridden for devices supporting jumbo frames.

# ether\_setup

## LINUX

Kernel Hackers Manual January 2013

## Name

`ether_setup` — setup Ethernet network device

## Synopsis

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

## Arguments

*dev*

network device Fill in the fields of the device structure with Ethernet-generic values.

## alloc\_etherdev\_mqs

### LINUX

Kernel Hackers Manual January 2013

## Name

`alloc_etherdev_mqs` — Allocates and sets up an Ethernet device

## Synopsis

```
struct net_device * alloc_etherdev_mqs (int sizeof_priv,  
unsigned int txqs, unsigned int rxqs);
```

## Arguments

*sizeof\_priv*

Size of additional driver-private structure to be allocated for this Ethernet device

*txqs*

The number of TX queues this device has.

*rxqs*

The number of RX queues this device has.



## Description

Fill in the fields of the device structure with Ethernet-generic values. Basically does everything except registering the device.

Constructs a new net device, complete with a private data area of size (sizeof\_priv). A 32-byte (not bit) alignment is enforced for this private data area.

# netif\_carrier\_on

## LINUX

Kernel Hackers Manual January 2013

## Name

netif\_carrier\_on — set carrier

## Synopsis

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

## Arguments

*dev*

network device

## Description

Device has detected that carrier.

# netif\_carrier\_off

## LINUX

Kernel Hackers Manual January 2013

### Name

`netif_carrier_off` — clear carrier

### Synopsis

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

### Arguments

*dev*

network device

### Description

Device has detected loss of carrier.

# netif\_notify\_peers

## LINUX

Kernel Hackers Manual January 2013

### Name

`netif_notify_peers` — notify network peers about existence of *dev*

## Synopsis

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

## Arguments

*dev*

network device

## Description

Generate traffic such that interested network peers are aware of *dev*, such as by generating a gratuitous ARP. This may be used when a device wants to inform the rest of the network about some sort of reconfiguration such as a failover event or virtual machine migration.

# is\_zero\_ether\_addr

## LINUX

Kernel Hackers Manual January 2013

## Name

`is_zero_ether_addr` — Determine if give Ethernet address is all zeros.

## Synopsis

```
int is_zero_ether_addr (const u8 * addr);
```

## Arguments

*addr*

Pointer to a six-byte array containing the Ethernet address

## Description

Return true if the address is all zeroes.

# is\_multicast\_ether\_addr

## LINUX

Kernel Hackers Manual January 2013

## Name

`is_multicast_ether_addr` — Determine if the Ethernet address is a multicast.

## Synopsis

```
int is_multicast_ether_addr (const u8 * addr);
```

## Arguments

*addr*

Pointer to a six-byte array containing the Ethernet address

## Description

Return true if the address is a multicast address. By definition the broadcast address is also a multicast address.

# is\_local\_ether\_addr

## LINUX

Kernel Hackers Manual January 2013

## Name

`is_local_ether_addr` — Determine if the Ethernet address is locally-assigned one (IEEE 802).

## Synopsis

```
int is_local_ether_addr (const u8 * addr);
```

## Arguments

*addr*

Pointer to a six-byte array containing the Ethernet address

## Description

Return true if the address is a local address.

# is\_broadcast\_ether\_addr

## LINUX

Kernel Hackers Manual January 2013

### Name

`is_broadcast_ether_addr` — Determine if the Ethernet address is broadcast

### Synopsis

```
int is_broadcast_ether_addr (const u8 * addr);
```

### Arguments

*addr*

Pointer to a six-byte array containing the Ethernet address

### Description

Return true if the address is the broadcast address.

# is\_unicast\_ether\_addr

## LINUX

Kernel Hackers Manual January 2013

### Name

`is_unicast_ether_addr` — Determine if the Ethernet address is unicast

## Synopsis

```
int is_unicast_ether_addr (const u8 * addr);
```

## Arguments

*addr*

Pointer to a six-byte array containing the Ethernet address

## Description

Return true if the address is a unicast address.

## is\_valid\_ether\_addr

### LINUX

Kernel Hackers Manual January 2013

## Name

`is_valid_ether_addr` — Determine if the given Ethernet address is valid

## Synopsis

```
int is_valid_ether_addr (const u8 * addr);
```

## Arguments

*addr*

Pointer to a six-byte array containing the Ethernet address

## Description

Check that the Ethernet address (MAC) is not 00:00:00:00:00:00, is not a multicast address, and is not FF:FF:FF:FF:FF:FF.

Return true if the address is valid.

# random\_ether\_addr

## LINUX

Kernel Hackers Manual January 2013

## Name

`random_ether_addr` — Generate software assigned random Ethernet address

## Synopsis

```
void random_ether_addr (u8 * addr);
```

## Arguments

*addr*

Pointer to a six-byte array containing the Ethernet address



## Description

Generate a random Ethernet address (MAC) that is not multicast and has the local assigned bit set.

# eth\_hw\_addr\_random

## LINUX

Kernel Hackers Manual January 2013

## Name

`eth_hw_addr_random` — Generate software assigned random Ethernet and set device flag

## Synopsis

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

## Arguments

*dev*

pointer to `net_device` structure

## Description

Generate a random Ethernet address (MAC) to be used by a net device and set `addr_assign_type` so the state can be read by `sysfs` and be used by userspace.

# compare\_ether\_addr

## LINUX

Kernel Hackers Manual January 2013

### Name

`compare_ether_addr` — Compare two Ethernet addresses

### Synopsis

```
unsigned compare_ether_addr (const u8 * addr1, const u8 *  
addr2);
```

### Arguments

*addr1*

Pointer to a six-byte array containing the Ethernet address

*addr2*

Pointer other six-byte array containing the Ethernet address

### Description

Compare two ethernet addresses, returns 0 if equal, non-zero otherwise. Unlike `memcmp`, it doesn't return a value suitable for sorting.

# compare\_ether\_addr\_64bits

## LINUX

## Name

`compare_ether_addr_64bits` — Compare two Ethernet addresses

## Synopsis

```
unsigned compare_ether_addr_64bits (const u8 addr1[6+2], const
u8 addr2[6+2]);
```

## Arguments

*addr1*[6+2]

Pointer to an array of 8 bytes

*addr2*[6+2]

Pointer to an other array of 8 bytes

## Description

Compare two ethernet addresses, returns 0 if equal, non-zero otherwise. Unlike `memcmp`, it doesn't return a value suitable for sorting. The function doesn't need any conditional branches and possibly uses word memory accesses on CPU allowing cheap unaligned memory reads. arrays = { byte1, byte2, byte3, byte4, byte6, byte7, pad1, pad2 }

Please note that alignment of `addr1` & `addr2` is only guaranted to be 16 bits.

## is\_etherdev\_addr

**LINUX**

## Name

`is_etherdev_addr` — Tell if given Ethernet address belongs to the device.

## Synopsis

```
bool is_etherdev_addr (const struct net_device * dev, const u8  
addr[6 + 2]);
```

## Arguments

*dev*

Pointer to a device structure

*addr[6 + 2]*

Pointer to a six-byte array containing the Ethernet address

## Description

Compare passed address with all addresses of the device. Return true if the address if one of the device addresses.

Note that this function calls `compare_ether_addr_64bits` so take care of the right padding.

## `compare_ether_header`

**LINUX**

## Name

`compare_ether_header` — Compare two Ethernet headers

## Synopsis

```
unsigned long compare_ether_header (const void * a, const void  
* b);
```

## Arguments

*a*

Pointer to Ethernet header

*b*

Pointer to Ethernet header

## Description

Compare two ethernet headers, returns 0 if equal. This assumes that the network header (i.e., IP header) is 4-byte aligned OR the platform can handle unaligned access. This is the case for all packets coming into `netif_receive_skb` or similar entry points.

## `napi_schedule_prep`

**LINUX**

## Name

`napi_schedule_prep` — check if napi can be scheduled

## Synopsis

```
bool napi_schedule_prep (struct napi_struct * n);
```

## Arguments

*n*

napi context

## Description

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.

# napi\_schedule

## LINUX

## Name

`napi_schedule` — schedule NAPI poll

## Synopsis

```
void napi_schedule (struct napi_struct * n);
```

## Arguments

*n*

napi context

## Description

Schedule NAPI poll routine to be called if it is not already running.

## napi\_disable

### LINUX

Kernel Hackers Manual January 2013

## Name

`napi_disable` — prevent NAPI from scheduling

## Synopsis

```
void napi_disable (struct napi_struct * n);
```

## Arguments

*n*

napi context

## Description

Stop NAPI from being scheduled on this context. Waits till any outstanding processing completes.

# napi\_enable

## LINUX

Kernel Hackers Manual January 2013

## Name

`napi_enable` — enable NAPI scheduling

## Synopsis

```
void napi_enable (struct napi_struct * n);
```

## Arguments

*n*

napi context



## Description

Resume NAPI from being scheduled on this context. Must be paired with `napi_disable`.

# napi\_synchronize

## LINUX

Kernel Hackers Manual January 2013

## Name

`napi_synchronize` — wait until NAPI is not running

## Synopsis

```
void napi_synchronize (const struct napi_struct * n);
```

## Arguments

*n*

napi context

## Description

Wait until NAPI is done being scheduled on this context. Waits till any outstanding processing completes but does not disable future activations.

# netdev\_priv

## LINUX

Kernel Hackers Manual January 2013

### Name

`netdev_priv` — access network device private data

### Synopsis

```
void * netdev_priv (const struct net_device * dev);
```

### Arguments

*dev*

network device

### Description

Get network device private data

# netif\_start\_queue

## LINUX

Kernel Hackers Manual January 2013

### Name

`netif_start_queue` — allow transmit

## Synopsis

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

## Arguments

*dev*

network device

## Description

Allow upper layers to call the device `hard_start_xmit` routine.

## netif\_wake\_queue

### LINUX

Kernel Hackers Manual January 2013

## Name

`netif_wake_queue` — restart transmit

## Synopsis

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

## Arguments

*dev*

network device

## Description

Allow upper layers to call the device `hard_start_xmit` routine. Used for flow control when transmit resources are available.

# netif\_stop\_queue

## LINUX

Kernel Hackers Manual January 2013

## Name

`netif_stop_queue` — stop transmitted packets

## Synopsis

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

## Arguments

*dev*

network device

## Description

Stop upper layers calling the device `hard_start_xmit` routine. Used for flow control when transmit resources are unavailable.

# netif\_queue\_stopped

## LINUX

Kernel Hackers Manual January 2013

## Name

`netif_queue_stopped` — test if transmit queue is flowblocked

## Synopsis

```
bool netif_queue_stopped (const struct net_device * dev);
```

## Arguments

*dev*

network device

## Description

Test if transmit queue on device is currently unable to send.

# netif\_running

## LINUX

Kernel Hackers Manual January 2013

### Name

`netif_running` — test if up

### Synopsis

```
bool netif_running (const struct net_device * dev);
```

### Arguments

*dev*

network device

### Description

Test if the device has been brought up.

# netif\_start\_subqueue

## LINUX

Kernel Hackers Manual January 2013

### Name

`netif_start_subqueue` — allow sending packets on subqueue

## Synopsis

```
void netif_start_subqueue (struct net_device * dev, u16  
queue_index);
```

## Arguments

*dev*

network device

*queue\_index*

sub queue index

## Description

Start individual transmit queue of a device with multiple transmit queues.

# netif\_stop\_subqueue

## LINUX

Kernel Hackers Manual January 2013

## Name

`netif_stop_subqueue` — stop sending packets on subqueue

## Synopsis

```
void netif_stop_subqueue (struct net_device * dev, u16  
queue_index);
```

## Arguments

*dev*

network device

*queue\_index*

sub queue index

## Description

Stop individual transmit queue of a device with multiple transmit queues.

# \_\_netif\_subqueue\_stopped

## LINUX

Kernel Hackers Manual January 2013

## Name

`__netif_subqueue_stopped` — test status of subqueue

## Synopsis

```
bool __netif_subqueue_stopped (const struct net_device * dev,  
                               u16 queue_index);
```



## Arguments

*dev*

network device

*queue\_index*

sub queue index

## Description

Check individual transmit queue of a device with multiple transmit queues.

# netif\_wake\_subqueue

## LINUX

Kernel Hackers Manual January 2013

## Name

`netif_wake_subqueue` — allow sending packets on subqueue

## Synopsis

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

## Arguments

*dev*

network device

`queue_index`

sub queue index

## Description

Resume individual transmit queue of a device with multiple transmit queues.

# netif\_is\_multiqueue

## LINUX

Kernel Hackers Manual January 2013

## Name

`netif_is_multiqueue` — test if device has multiple transmit queues

## Synopsis

```
bool netif_is_multiqueue (const struct net_device * dev);
```

## Arguments

*dev*

network device

## Description

Check if device has multiple transmit queues

# dev\_put

## LINUX

Kernel Hackers Manual January 2013

### Name

`dev_put` — release reference to device

### Synopsis

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

### Arguments

*dev*

network device

### Description

Release reference to device to allow it to be freed.

# dev\_hold

## LINUX

Kernel Hackers Manual January 2013

### Name

`dev_hold` — get reference to device

## Synopsis

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

## Arguments

*dev*

network device

## Description

Hold reference to device to keep it from being freed.

## netif\_carrier\_ok

### LINUX

Kernel Hackers Manual January 2013

## Name

`netif_carrier_ok` — test if carrier present

## Synopsis

```
bool netif_carrier_ok (const struct net_device * dev);
```

## Arguments

*dev*

network device

## Description

Check if carrier is present on device

# netif\_dormant\_on

## LINUX

Kernel Hackers Manual January 2013

## Name

`netif_dormant_on` — mark device as dormant.

## Synopsis

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

## Arguments

*dev*

network device

## Description

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.

## netif\_dormant\_off

### LINUX

Kernel Hackers Manual January 2013

### Name

`netif_dormant_off` — set device as not dormant.

### Synopsis

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

### Arguments

*dev*

network device

### Description

Device is not in dormant state.

# netif\_dormant

## LINUX

Kernel Hackers Manual January 2013

### Name

`netif_dormant` — test if carrier present

### Synopsis

```
bool netif_dormant (const struct net_device * dev);
```

### Arguments

*dev*

network device

### Description

Check if carrier is present on device

# netif\_oper\_up

## LINUX

Kernel Hackers Manual January 2013

### Name

`netif_oper_up` — test if device is operational

## Synopsis

```
bool netif_oper_up (const struct net_device * dev);
```

## Arguments

*dev*

network device

## Description

Check if carrier is operational

## netif\_device\_present

### LINUX

Kernel Hackers Manual January 2013

## Name

`netif_device_present` — is device available or removed

## Synopsis

```
bool netif_device_present (struct net_device * dev);
```



## Arguments

*dev*

network device

## Description

Check if device has not been removed from system.

# netif\_tx\_lock

## LINUX

Kernel Hackers Manual January 2013

## Name

`netif_tx_lock` — grab network device transmit lock

## Synopsis

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

## Arguments

*dev*

network device

## Description

Get network device transmit lock

## 2.2. PHY Support

### phy\_print\_status

#### LINUX

Kernel Hackers Manual January 2013

#### Name

`phy_print_status` — Convenience function to print out the current phy status

#### Synopsis

```
void phy_print_status (struct phy_device * phydev);
```

#### Arguments

*phydev*

the `phy_device` struct

### phy\_ethtool\_sset

#### LINUX

## Name

`phy_ethtool_sset` — generic ethtool sset function, handles all the details

## Synopsis

```
int phy_ethtool_sset (struct phy_device * phydev, struct
ethtool_cmd * cmd);
```

## Arguments

*phydev*

target `phy_device` struct

*cmd*

`ethtool_cmd`

## A few notes about parameter checking

- We don't set port or transceiver, so we don't care what they were set to. -  
`phy_start_aneg` will make sure forced settings are sane, and choose the next best  
ones from the ones selected, so we don't care if ethtool tries to give us bad values.

## `phy_mii_ioctl`

**LINUX**

## Name

`phy_mii_ioctl` — generic PHY MII ioctl interface

## Synopsis

```
int phy_mii_ioctl (struct phy_device * phydev, struct ifreq *  
ifr, int cmd);
```

## Arguments

*phydev*

the `phy_device` struct

*ifr*

struct `ifreq` for socket ioctl's

*cmd*

ioctl cmd to execute

## Description

Note that this function is currently incompatible with the PHYCONTROL layer. It changes registers without regard to current state. Use at own risk.

## `phy_start_aneg`

**LINUX**

## Name

`phy_start_aneg` — start auto-negotiation for this PHY device

## Synopsis

```
int phy_start_aneg (struct phy_device * phydev);
```

## Arguments

*phydev*

the `phy_device` struct

## Description

Sanitizes the settings (if we're not autonegotiating them), and then calls the driver's `config_aneg` function. If the PHYCONTROL Layer is operating, we change the state to reflect the beginning of Auto-negotiation or forcing.

# phy\_start\_interrupts

## LINUX

## Name

`phy_start_interrupts` — request and enable interrupts for a PHY device

## Synopsis

```
int phy_start_interrupts (struct phy_device * phydev);
```

## Arguments

*phydev*

target phy\_device struct

## Description

Request the interrupt for the given PHY. If this fails, then we set irq to PHY\_POLL. Otherwise, we enable the interrupts in the PHY. This should only be called with a valid IRQ number. Returns 0 on success or < 0 on error.

# phy\_stop\_interrupts

## LINUX

Kernel Hackers Manual January 2013

## Name

`phy_stop_interrupts` — disable interrupts from a PHY device

## Synopsis

```
int phy_stop_interrupts (struct phy_device * phydev);
```

## Arguments

*phydev*

target phy\_device struct

## phy\_stop

**LINUX**

Kernel Hackers Manual January 2013

## Name

`phy_stop` — Bring down the PHY link, and stop checking the status

## Synopsis

```
void phy_stop (struct phy_device * phydev);
```

## Arguments

*phydev*

target phy\_device struct

## phy\_start

**LINUX**

## Name

`phy_start` — start or restart a PHY device

## Synopsis

```
void phy_start (struct phy_device * phydev);
```

## Arguments

*phydev*

target `phy_device` struct

## Description

Indicates the attached device's readiness to handle PHY-related work. Used during startup to start the PHY, and after a call to `phy_stop` to resume operation. Also used to indicate the MDIO bus has cleared an error condition.

# phy\_clear\_interrupt

## LINUX

## Name

`phy_clear_interrupt` — Ack the phy device's interrupt



## Synopsis

```
int phy_clear_interrupt (struct phy_device * phydev);
```

## Arguments

*phydev*

the `phy_device` struct

## Description

If the *phydev* driver has an `ack_interrupt` function, call it to ack and clear the phy device's interrupt.

Returns 0 on success on < 0 on error.

# phy\_config\_interrupt

## LINUX

Kernel Hackers Manual January 2013

## Name

`phy_config_interrupt` — configure the PHY device for the requested interrupts

## Synopsis

```
int phy_config_interrupt (struct phy_device * phydev, u32  
interrupts);
```

## Arguments

*phydev*

the `phy_device` struct

*interrupts*

interrupt flags to configure for this *phydev*

## Description

Returns 0 on success on < 0 on error.

# phy\_aneg\_done

## LINUX

Kernel Hackers Manual January 2013

## Name

`phy_aneg_done` — return auto-negotiation status

## Synopsis

```
int phy_aneg_done (struct phy_device * phydev);
```

## Arguments

*phydev*

target `phy_device` struct

## Description

Reads the status register and returns 0 either if auto-negotiation is incomplete, or if there was an error. Returns BMSR\_ANEGCOMPLETE if auto-negotiation is done.

# phy\_find\_setting

## LINUX

Kernel Hackers Manual January 2013

## Name

`phy_find_setting` — find a PHY settings array entry that matches speed & duplex

## Synopsis

```
int phy_find_setting (int speed, int duplex);
```

## Arguments

*speed*

speed to match

*duplex*

duplex to match

## Description

Searches the settings array for the setting which matches the desired speed and duplex, and returns the index of that setting. Returns the index of the last setting if none of the others match.

# phy\_find\_valid

## LINUX

Kernel Hackers Manual January 2013

### Name

`phy_find_valid` — find a PHY setting that matches the requested features  
mask

### Synopsis

```
int phy_find_valid (int idx, u32 features);
```

### Arguments

*idx*

The first index in `settings[]` to search

*features*

A mask of the valid settings

### Description

Returns the index of the first valid setting less than or equal to the one pointed to by `idx`, as determined by the mask in `features`. Returns the index of the last setting if nothing else matches.

# phy\_sanitize\_settings

## LINUX

Kernel Hackers Manual January 2013

### Name

`phy_sanitize_settings` — make sure the PHY is set to supported speed and duplex

### Synopsis

```
void phy_sanitize_settings (struct phy_device * phydev);
```

### Arguments

*phydev*

the target `phy_device` struct

### Description

Make sure the PHY is set to supported speeds and duplexes. Drop down by one in this order: 1000/FULL, 1000/HALF, 100/FULL, 100/HALF, 10/FULL, 10/HALF.

# phy\_start\_machine

## LINUX

## Name

`phy_start_machine` — start PHY state machine tracking

## Synopsis

```
void phy_start_machine (struct phy_device * phydev, void  
(*handler) (struct net_device *));
```

## Arguments

*phydev*

the `phy_device` struct

*handler*

callback function for state change notifications

## Description

The PHY infrastructure can run a state machine which tracks whether the PHY is starting up, negotiating, etc. This function starts the timer which tracks the state of the PHY. If you want to be notified when the state changes, pass in the callback *handler*, otherwise, pass NULL. If you want to maintain your own state machine, do not call this function.

## `phy_stop_machine`

**LINUX**

## Name

`phy_stop_machine` — stop the PHY state machine tracking

## Synopsis

```
void phy_stop_machine (struct phy_device * phydev);
```

## Arguments

*phydev*

target `phy_device` struct

## Description

Stops the state machine timer, sets the state to UP (unless it wasn't up yet). This function must be called BEFORE `phy_detach`.

# phy\_force\_reduction

## LINUX

## Name

`phy_force_reduction` — reduce PHY speed/duplex settings by one step

## Synopsis

```
void phy_force_reduction (struct phy_device * phydev);
```

## Arguments

*phydev*

target phy\_device struct

## Description

Reduces the speed/duplex settings by one notch, in this order-- 1000/FULL, 1000/HALF, 100/FULL, 100/HALF, 10/FULL, 10/HALF. The function bottoms out at 10/HALF.

## phy\_error

### LINUX

Kernel Hackers Manual January 2013

## Name

`phy_error` — enter HALTED state for this PHY device

## Synopsis

```
void phy_error (struct phy_device * phydev);
```



## Arguments

*phydev*

target `phy_device` struct

## Description

Moves the PHY to the HALTED state in response to a read or write error, and tells the controller the link is down. Must not be called from interrupt context, or while the `phydev->lock` is held.

## phy\_interrupt

### LINUX

Kernel Hackers Manual January 2013

## Name

`phy_interrupt` — PHY interrupt handler

## Synopsis

```
irqreturn_t phy_interrupt (int irq, void * phy_dat);
```

## Arguments

*irq*

interrupt line

*phy\_dat*

`phy_device` pointer

## Description

When a PHY interrupt occurs, the handler disables interrupts, and schedules a work task to clear the interrupt.

# phy\_enable\_interrupts

## LINUX

Kernel Hackers Manual January 2013

## Name

`phy_enable_interrupts` — Enable the interrupts from the PHY side

## Synopsis

```
int phy_enable_interrupts (struct phy_device * phydev);
```

## Arguments

*phydev*

target phy\_device struct

# phy\_disable\_interrupts

## LINUX

## Name

`phy_disable_interrupts` — Disable the PHY interrupts from the PHY side

## Synopsis

```
int phy_disable_interrupts (struct phy_device * phydev);
```

## Arguments

*phydev*

target `phy_device` struct

# phy\_change

## LINUX

## Name

`phy_change` — Scheduled by the `phy_interrupt/timer` to handle PHY changes

## Synopsis

```
void phy_change (struct work_struct * work);
```

## Arguments

*work*

work\_struct that describes the work to be done

# phy\_state\_machine

## LINUX

Kernel Hackers Manual January 2013

## Name

phy\_state\_machine — Handle the state machine

## Synopsis

```
void phy_state_machine (struct work_struct * work);
```

## Arguments

*work*

work\_struct that describes the work to be done

# get\_phy\_id

## LINUX

## Name

`get_phy_id` — reads the specified `addr` for its ID.

## Synopsis

```
int get_phy_id (struct mii_bus * bus, int addr, u32 * phy_id);
```

## Arguments

*bus*

the target MII bus

*addr*

PHY address on the MII bus

*phy\_id*

where to store the ID retrieved.

## Description

Reads the ID registers of the PHY at *addr* on the *bus*, stores it in *phy\_id* and returns zero on success.

## get\_phy\_device

**LINUX**

## Name

`get_phy_device` — reads the specified PHY device and returns its `phy_device` struct

## Synopsis

```
struct phy_device * get_phy_device (struct mii_bus * bus, int  
addr);
```

## Arguments

*bus*

the target MII bus

*addr*

PHY address on the MII bus

## Description

Reads the ID registers of the PHY at *addr* on the *bus*, then allocates and returns the `phy_device` to represent it.

# phy\_device\_register

**LINUX**

## Name

`phy_device_register` — Register the phy device on the MDIO bus

## Synopsis

```
int phy_device_register (struct phy_device * phydev);
```

## Arguments

*phydev*

phy\_device structure to be added to the MDIO bus

# phy\_find\_first

## LINUX

## Name

`phy_find_first` — finds the first PHY device on the bus

## Synopsis

```
struct phy_device * phy_find_first (struct mii_bus * bus);
```

## Arguments

*bus*

the target MII bus

## phy\_connect\_direct

### LINUX

Kernel Hackers Manual January 2013

## Name

`phy_connect_direct` — connect an ethernet device to a specific `phy_device`

## Synopsis

```
int phy_connect_direct (struct net_device * dev, struct
phy_device * phydev, void (*handler) (struct net_device *),
u32 flags, phy_interface_t interface);
```

## Arguments

*dev*

the network device to connect

*phydev*

the pointer to the phy device

*handler*

callback function for state change notifications



*flags*

PHY device's dev\_flags

*interface*

PHY device's interface

## phy\_connect

### LINUX

Kernel Hackers Manual January 2013

### Name

`phy_connect` — connect an ethernet device to a PHY device

### Synopsis

```
struct phy_device * phy_connect (struct net_device * dev,
const char * bus_id, void (*handler) (struct net_device *),
u32 flags, phy_interface_t interface);
```

### Arguments

*dev*

the network device to connect

*bus\_id*

the id string of the PHY device to connect

*handler*

callback function for state change notifications

*flags*

PHY device's dev\_flags

*interface*

PHY device's interface

## Description

Convenience function for connecting ethernet devices to PHY devices. The default behavior is for the PHY infrastructure to handle everything, and only notify the connected driver when the link status changes. If you don't want, or can't use the provided functionality, you may choose to call only the subset of functions which provide the desired functionality.

# phy\_disconnect

**LINUX**

Kernel Hackers Manual January 2013

## Name

`phy_disconnect` — disable interrupts, stop state machine, and detach a PHY device

## Synopsis

```
void phy_disconnect (struct phy_device * phydev);
```

## Arguments

*phydev*

target phy\_device struct

## phy\_attach

### LINUX

Kernel Hackers Manual January 2013

## Name

`phy_attach` — attach a network device to a particular PHY device

## Synopsis

```
struct phy_device * phy_attach (struct net_device * dev, const  
char * bus_id, u32 flags, phy_interface_t interface);
```

## Arguments

*dev*

network device to attach

*bus\_id*

Bus ID of PHY device to attach

*flags*

PHY device's dev\_flags

*interface*

PHY device's interface

## Description

Same as `phy_attach_direct` except that a PHY `bus_id` string is passed instead of a pointer to a struct `phy_device`.

# phy\_detach

## LINUX

Kernel Hackers Manual January 2013

## Name

`phy_detach` — detach a PHY device from its network device

## Synopsis

```
void phy_detach (struct phy_device * phydev);
```

## Arguments

*phydev*

target `phy_device` struct

# genphy\_restart\_aneg

## LINUX

Kernel Hackers Manual January 2013

### Name

genphy\_restart\_aneg — Enable and Restart Autonegotiation

### Synopsis

```
int genphy_restart_aneg (struct phy_device * phydev);
```

### Arguments

*phydev*

target phy\_device struct

# genphy\_config\_aneg

## LINUX

Kernel Hackers Manual January 2013

### Name

genphy\_config\_aneg — restart auto-negotiation or write BMCR

## Synopsis

```
int genphy_config_aneg (struct phy_device * phydev);
```

## Arguments

*phydev*

target phy\_device struct

## Description

If auto-negotiation is enabled, we configure the advertising, and then restart auto-negotiation. If it is not enabled, then we write the BMCR.

# genphy\_update\_link

## LINUX

Kernel Hackers Manual January 2013

## Name

`genphy_update_link` — update link status in *phydev*

## Synopsis

```
int genphy_update_link (struct phy_device * phydev);
```

## Arguments

*phydev*

target phy\_device struct

## Description

Update the value in `phydev->link` to reflect the current link value. In order to do this, we need to read the status register twice, keeping the second value.

# genphy\_read\_status

## LINUX

Kernel Hackers Manual January 2013

## Name

`genphy_read_status` — check the link status and update current link state

## Synopsis

```
int genphy_read_status (struct phy_device * phydev);
```

## Arguments

*phydev*

target phy\_device struct

## Description

Check the link, then figure out the current state by comparing what we advertise with what the link partner advertises. Start by checking the gigabit possibilities, then move on to 10/100.

# phy\_driver\_register

## LINUX

Kernel Hackers Manual January 2013

## Name

`phy_driver_register` — register a `phy_driver` with the PHY layer

## Synopsis

```
int phy_driver_register (struct phy_driver * new_driver);
```

## Arguments

*new\_driver*

new `phy_driver` to register

# phy\_prepare\_link

## LINUX



## Name

`phy_prepare_link` — prepares the PHY layer to monitor link status

## Synopsis

```
void phy_prepare_link (struct phy_device * phydev, void  
(*handler) (struct net_device *));
```

## Arguments

*phydev*

target `phy_device` struct

*handler*

callback function for link status change notifications

## Description

Tells the PHY infrastructure to handle the gory details on monitoring link status (whether through polling or an interrupt), and to call back to the connected device driver when the link status changes. If you want to monitor your own link state, don't call this function.

## `phy_attach_direct`

**LINUX**

## Name

`phy_attach_direct` — attach a network device to a given PHY device pointer

## Synopsis

```
int phy_attach_direct (struct net_device * dev, struct
phy_device * phydev, u32 flags, phy_interface_t interface);
```

## Arguments

*dev*

network device to attach

*phydev*

Pointer to `phy_device` to attach

*flags*

PHY device's `dev_flags`

*interface*

PHY device's interface

## Description

Called by drivers to attach to a particular PHY device. The `phy_device` is found, and properly hooked up to the `phy_driver`. If no driver is attached, then the `genphy_driver` is used. The `phy_device` is given a ptr to the attaching device, and given a callback for link status change. The `phy_device` is returned to the attaching driver.

# genphy\_config\_advert

## LINUX

Kernel Hackers Manual January 2013

### Name

`genphy_config_advert` — sanitize and advertise auto-negotiation parameters

### Synopsis

```
int genphy_config_advert (struct phy_device * phydev);
```

### Arguments

*phydev*

target phy\_device struct

### Description

Writes MII\_ADVERTISE with the appropriate values, after sanitizing the values to make sure we only advertise what is supported. Returns < 0 on error, 0 if the PHY's advertisement hasn't changed, and > 0 if it has changed.

# genphy\_setup\_forced

## LINUX

## Name

`genphy_setup_forced` — configures/forces speed/duplex from *phydev*

## Synopsis

```
int genphy_setup_forced (struct phy_device * phydev);
```

## Arguments

*phydev*

target `phy_device` struct

## Description

Configures MII\_BMCR to force speed/duplex to the values in *phydev*. Assumes that the values are valid. Please see `phy_sanitize_settings`.

# phy\_probe

## LINUX

## Name

`phy_probe` — probe and init a PHY device

## Synopsis

```
int phy_probe (struct device * dev);
```

## Arguments

*dev*

device to probe and init

## Description

Take care of setting up the `phy_device` structure, set the state to `READY` (the driver's init function should set it to `STARTING` if needed).

# mdiobus\_alloc\_size

## LINUX

Kernel Hackers Manual January 2013

## Name

`mdiobus_alloc_size` — allocate a `mii_bus` structure

## Synopsis

```
struct mii_bus * mdiobus_alloc_size (size_t size);
```

## Arguments

*size*

extra amount of memory to allocate for private storage. If non-zero, then `bus->priv` is points to that memory.

## Description

called by a bus driver to allocate an `mii_bus` structure to fill in.

# mdiobus\_register

## LINUX

Kernel Hackers Manual January 2013

## Name

`mdiobus_register` — bring up all the PHYs on a given bus and attach them to `bus`

## Synopsis

```
int mdiobus_register (struct mii_bus * bus);
```

## Arguments

*bus*

target `mii_bus`

## Description

Called by a bus driver to bring up all the PHYs on a given bus, and attach them to the bus.

Returns 0 on success or < 0 on error.

## mdiobus\_free

### LINUX

Kernel Hackers Manual January 2013

## Name

`mdiobus_free` — free a struct `mii_bus`

## Synopsis

```
void mdiobus_free (struct mii_bus * bus);
```

## Arguments

*bus*

`mii_bus` to free

## Description

This function releases the reference to the underlying device object in the `mii_bus`. If this is the last reference, the `mii_bus` will be freed.

# mdiobus\_read

## LINUX

Kernel Hackers Manual January 2013

### Name

`mdiobus_read` — Convenience function for reading a given MII mgmt register

### Synopsis

```
int mdiobus_read (struct mii_bus * bus, int addr, u32 regnum);
```

### Arguments

*bus*

the `mii_bus` struct

*addr*

the phy address

*regnum*

register number to read

### NOTE

MUST NOT be called from interrupt context, because the bus read/write functions may wait for an interrupt to conclude the operation.



# mdiobus\_write

## LINUX

Kernel Hackers Manual January 2013

### Name

`mdiobus_write` — Convenience function for writing a given MII mgmt register

### Synopsis

```
int mdiobus_write (struct mii_bus * bus, int addr, u32 regnum,
u16 val);
```

### Arguments

*bus*

the `mii_bus` struct

*addr*

the phy address

*regnum*

register number to write

*val*

value to write to *regnum*

### NOTE

MUST NOT be called from interrupt context, because the bus read/write functions may wait for an interrupt to conclude the operation.

# mdiobus\_release

## LINUX

Kernel Hackers Manual January 2013

### Name

`mdiobus_release` — mii\_bus device release callback

### Synopsis

```
void mdiobus_release (struct device * d);
```

### Arguments

*d*

the target struct device that contains the mii\_bus

### Description

called when the last reference to an mii\_bus is dropped, to free the underlying memory.

# mdio\_bus\_match

## LINUX

## Name

`mdio_bus_match` — determine if given PHY driver supports the given PHY device

## Synopsis

```
int mdio_bus_match (struct device * dev, struct device_driver  
* drv);
```

## Arguments

*dev*

target PHY device

*drv*

given PHY driver

## Description

Given a PHY device, and a PHY driver, return 1 if the driver supports the device. Otherwise, return 0.

