

# **The mac80211 subsystem for kernel developers**

**Johannes Berg**

**`johannes@sipsolutions.net`**

# **The mac80211 subsystem for kernel developers**

by Johannes Berg

Copyright © 2007-2009 Johannes Berg

mac80211 is the Linux stack for 802.11 hardware that implements only partial functionality in hard- or firmware. This document defines the interface between mac80211 and low-level hardware drivers.

If you're reading this document and not the header file itself, it will be incomplete because not all documentation has been converted yet.

This documentation is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License version 2 as published by the Free Software Foundation.

This documentation 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 documentation; 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>I. The basic mac80211 driver interface.....</b>	<b>v</b>
1. Basic hardware handling.....	1
struct ieee80211_hw .....	1
enum ieee80211_hw_flags.....	3
SET_IEEE80211_DEV.....	6
SET_IEEE80211_PERM_ADDR.....	6
struct ieee80211_ops.....	7
ieee80211_alloc_hw .....	12
ieee80211_register_hw .....	13
ieee80211_get_tx_led_name.....	14
ieee80211_get_rx_led_name .....	15
ieee80211_get_assoc_led_name .....	16
ieee80211_get_radio_led_name .....	17
ieee80211_unregister_hw .....	17
ieee80211_free_hw .....	18
2. PHY configuration .....	21
struct ieee80211_conf .....	21
enum ieee80211_conf_flags.....	22
3. Virtual interfaces.....	25
struct ieee80211_if_init_conf .....	25
4. Receive and transmit processing.....	27
4.1. what should be here .....	27
4.2. Frame format.....	27
4.3. Packet alignment.....	27
4.4. Calling into mac80211 from interrupts.....	28
4.5. functions/definitions.....	28
struct ieee80211_rx_status.....	28
enum mac80211_rx_flags .....	30
struct ieee80211_tx_info.....	32
ieee80211_rx.....	33
ieee80211_rx_irqsafe .....	34
ieee80211_tx_status .....	35
ieee80211_tx_status_irqsafe .....	36
ieee80211_rts_get .....	37
ieee80211_rts_duration.....	38
ieee80211_ctstoself_get.....	39
ieee80211_ctstoself_duration .....	41
ieee80211_generic_frame_duration.....	42
ieee80211_wake_queue .....	43
ieee80211_stop_queue .....	44
ieee80211_wake_queues.....	45

ieee80211_stop_queues .....	46
5. Frame filtering.....	47
enum ieee80211_filter_flags .....	47
<b>II. Advanced driver interface.....</b>	<b>51</b>
6. Hardware crypto acceleration .....	53
enum set_key_cmd.....	53
struct ieee80211_key_conf .....	54
enum ieee80211_key_alg.....	56
enum ieee80211_key_flags .....	57
7. Powersave support .....	59
8. Beacon filter support.....	61
ieee80211_beacon_loss.....	62
9. Multiple queues and QoS support.....	63
struct ieee80211_tx_queue_params .....	63
struct ieee80211_tx_queue_stats .....	64
10. Access point mode support .....	67
ieee80211_get_buffered_bc .....	67
ieee80211_beacon_get.....	68
11. Supporting multiple virtual interfaces .....	71
12. Hardware scan offload .....	73
ieee80211_scan_completed .....	73
<b>III. Rate control interface.....</b>	<b>75</b>
13. dummy chapter.....	77
<b>IV. Internals.....</b>	<b>79</b>
14. Key handling .....	81
14.1. Key handling basics .....	81
14.2. MORE TBD .....	81
15. Receive processing.....	83
16. Transmit processing .....	85
17. Station info handling.....	87
17.1. Programming information.....	87
struct sta_info.....	87
enum ieee80211_sta_info_flags.....	92
17.2. STA information lifetime rules .....	94
18. Synchronisation.....	97

# I. The basic mac80211 driver interface

## Table of Contents

<b>1. Basic hardware handling.....</b>	<b>1</b>
<b>2. PHY configuration .....</b>	<b>21</b>
<b>3. Virtual interfaces.....</b>	<b>25</b>
<b>4. Receive and transmit processing .....</b>	<b>27</b>
<b>5. Frame filtering.....</b>	<b>47</b>

You should read and understand the information contained within this part of the book while implementing a driver. In some chapters, advanced usage is noted, that may be skipped at first.

This part of the book only covers station and monitor mode functionality, additional information required to implement the other modes is covered in the second part of the book.



# Chapter 1. Basic hardware handling

TBD

This chapter shall contain information on getting a hw struct allocated and registered with mac80211.

Since it is required to allocate rates/modes before registering a hw struct, this chapter shall also contain information on setting up the rate/mode structs.

Additionally, some discussion about the callbacks and the general programming model should be in here, including the definition of ieee80211\_ops which will be referred to a lot.

Finally, a discussion of hardware capabilities should be done with references to other parts of the book.

## struct ieee80211\_hw

**LINUX**

Kernel Hackers Manual November 2011

### Name

struct ieee80211\_hw — hardware information and state

### Synopsis

```
struct ieee80211_hw {
    struct ieee80211_conf conf;
    struct wiphy * wiphy;
    const char * rate_control_algorithm;
    void * priv;
    u32 flags;
    unsigned int extra_tx_headroom;
    int channel_change_time;
    int vif_data_size;
    int sta_data_size;
    u16 queues;
    u16 max_listen_interval;
    s8 max_signal;
    u8 max_rates;
    u8 max_rate_tries;
```

```
};
```

## Members

`conf`

struct `ieee80211_conf`, device configuration, don't use.

`wiphy`

This points to the struct `wiphy` allocated for this 802.11 PHY. You must fill in the `perm_addr` and `dev` members of this structure using `SET_IEEE80211_DEV` and `SET_IEEE80211_PERM_ADDR`. Additionally, all supported bands (with channels, bitrates) are registered here.

`rate_control_algorithm`

rate control algorithm for this hardware. If unset (NULL), the default algorithm will be used. Must be set before calling `ieee80211_register_hw`.

`priv`

pointer to private area that was allocated for driver use along with this structure.

`flags`

hardware flags, see enum `ieee80211_hw_flags`.

`extra_tx_headroom`

headroom to reserve in each transmit skb for use by the driver (e.g. for transmit headers.)

`channel_change_time`

time (in microseconds) it takes to change channels.

`vif_data_size`

size (in bytes) of the `drv_priv` data area within struct `ieee80211_vif`.

`sta_data_size`

size (in bytes) of the `drv_priv` data area within struct `ieee80211_sta`.



queues

number of available hardware transmit queues for data packets. WMM/QoS requires at least four, these queues need to have configurable access parameters.

max\_listen\_interval

max listen interval in units of beacon interval that HW supports

max\_signal

Maximum value for signal (rssi) in RX information, used only when *IEEE80211\_HW\_SIGNAL\_UNSPEC* or *IEEE80211\_HW\_SIGNAL\_DB*

max\_rates

maximum number of alternate rate retry stages

max\_rate\_tries

maximum number of tries for each stage

## Description

This structure contains the configuration and hardware information for an 802.11 PHY.

## enum ieee80211\_hw\_flags

### LINUX

Kernel Hackers Manual November 2011

### Name

enum ieee80211\_hw\_flags — hardware flags

### Synopsis

```
enum ieee80211_hw_flags {
```

```
IEEE80211_HW_RX_INCLUDES_FCS,  
IEEE80211_HW_HOST_BROADCAST_PS_BUFFERING,  
IEEE80211_HW_2GHZ_SHORT_SLOT_INCAPABLE,  
IEEE80211_HW_2GHZ_SHORT_PREAMBLE_INCAPABLE,  
IEEE80211_HW_SIGNAL_UNSPEC,  
IEEE80211_HW_SIGNAL_DBM,  
IEEE80211_HW_NOISE_DBM,  
IEEE80211_HW_SPECTRUM_MGMT,  
IEEE80211_HW_AMPDU_AGGREGATION,  
IEEE80211_HW_SUPPORTS_PS,  
IEEE80211_HW_PS_NULLFUNC_STACK,  
IEEE80211_HW_SUPPORTS_DYNAMIC_PS,  
IEEE80211_HW_MFP_CAPABLE,  
IEEE80211_HW_BEACON_FILTER,  
IEEE80211_HW_REPORTS_TX_ACK_STATUS  
};
```

## Constants

### IEEE80211\_HW\_RX\_INCLUDES\_FCS

Indicates that received frames passed to the stack include the FCS at the end.

### IEEE80211\_HW\_HOST\_BROADCAST\_PS\_BUFFERING

Some wireless LAN chipsets buffer broadcast/multicast frames for power saving stations in the hardware/firmware and others rely on the host system for such buffering. This option is used to configure the IEEE 802.11 upper layer to buffer broadcast and multicast frames when there are power saving stations so that the driver can fetch them with `ieee80211_get_buffered_bc`.

### IEEE80211\_HW\_2GHZ\_SHORT\_SLOT\_INCAPABLE

Hardware is not capable of short slot operation on the 2.4 GHz band.

### IEEE80211\_HW\_2GHZ\_SHORT\_PREAMBLE\_INCAPABLE

Hardware is not capable of receiving frames with short preamble on the 2.4 GHz band.

### IEEE80211\_HW\_SIGNAL\_UNSPEC

Hardware can provide signal values but we don't know its units. We expect values between 0 and `max_signal`. If possible please provide dB or dBm instead.

#### IEEE80211\_HW\_SIGNAL\_DBM

Hardware gives signal values in dBm, decibel difference from one milliwatt. This is the preferred method since it is standardized between different devices. *max\_signal* does not need to be set.

#### IEEE80211\_HW\_NOISE\_DBM

Hardware can provide noise (radio interference) values in units dBm, decibel difference from one milliwatt.

#### IEEE80211\_HW\_SPECTRUM\_MGMT

Hardware supports spectrum management defined in 802.11h Measurement, Channel Switch, Quieting, TPC

#### IEEE80211\_HW\_AMPDU\_AGGREGATION

Hardware supports 11n A-MPDU aggregation.

#### IEEE80211\_HW\_SUPPORTS\_PS

Hardware has power save support (i.e. can go to sleep).

#### IEEE80211\_HW\_PS\_NULLFUNC\_STACK

Hardware requires nullfunc frame handling in stack, implies stack support for dynamic PS.

#### IEEE80211\_HW\_SUPPORTS\_DYNAMIC\_PS

Hardware has support for dynamic PS.

#### IEEE80211\_HW\_MFP\_CAPABLE

Hardware supports management frame protection (MFP, IEEE 802.11w).

#### IEEE80211\_HW\_BEACON\_FILTER

Hardware supports dropping of irrelevant beacon frames to avoid waking up cpu.

#### IEEE80211\_HW\_REPORTS\_TX\_ACK\_STATUS

Hardware can provide ack status reports of Tx frames to the stack.

## Description

These flags are used to indicate hardware capabilities to the stack. Generally, flags here should have their meaning done in a way that the simplest hardware doesn't

need setting any particular flags. There are some exceptions to this rule, however, so you are advised to review these flags carefully.

## SET\_IEEE80211\_DEV

### LINUX

Kernel Hackers Manual November 2011

### Name

SET\_IEEE80211\_DEV — set device for 802.11 hardware

### Synopsis

```
void SET_IEEE80211_DEV (struct ieee80211_hw * hw, struct
device * dev);
```

### Arguments

*hw*

the struct ieee80211\_hw to set the device for

*dev*

the struct device of this 802.11 device

## SET\_IEEE80211\_PERM\_ADDR

### LINUX

## Name

`SET_IEEE80211_PERM_ADDR` — set the permanent MAC address for 802.11 hardware

## Synopsis

```
void SET_IEEE80211_PERM_ADDR (struct ieee80211_hw * hw, u8 *  
addr);
```

## Arguments

*hw*

the struct `ieee80211_hw` to set the MAC address for

*addr*

the address to set

## struct `ieee80211_ops`

### LINUX

## Name

`struct ieee80211_ops` — callbacks from `mac80211` to the driver

## Synopsis

```
struct ieee80211_ops {
```

```
int (* tx) (struct ieee80211_hw *hw, struct sk_buff *skb);
int (* start) (struct ieee80211_hw *hw);
void (* stop) (struct ieee80211_hw *hw);
int (* add_interface) (struct ieee80211_hw *hw, struct ieee80211_if_init
void (* remove_interface) (struct ieee80211_hw *hw, struct ieee80211_if_
int (* config) (struct ieee80211_hw *hw, u32 changed);
void (* bss_info_changed) (struct ieee80211_hw *hw, struct ieee80211_vif
u64 (* prepare_multicast) (struct ieee80211_hw *hw, int mc_count, struct
void (* configure_filter) (struct ieee80211_hw *hw, unsigned int changed
int (* set_tim) (struct ieee80211_hw *hw, struct ieee80211_sta *sta, bool
int (* set_key) (struct ieee80211_hw *hw, enum set_key_cmd cmd, struct ie
void (* update_tkip_key) (struct ieee80211_hw *hw, struct ieee80211_key_c
int (* hw_scan) (struct ieee80211_hw *hw, struct cfg80211_scan_request *r
void (* sw_scan_start) (struct ieee80211_hw *hw);
void (* sw_scan_complete) (struct ieee80211_hw *hw);
int (* get_stats) (struct ieee80211_hw *hw, struct ieee80211_low_level_st
void (* get_tkip_seq) (struct ieee80211_hw *hw, u8 hw_key_idx, u32 *iv32,
int (* set_rts_threshold) (struct ieee80211_hw *hw, u32 value);
void (* sta_notify) (struct ieee80211_hw *hw, struct ieee80211_vif *vif,
int (* conf_tx) (struct ieee80211_hw *hw, u16 queue, const struct ieee80
int (* get_tx_stats) (struct ieee80211_hw *hw, struct ieee80211_tx_queue
u64 (* get_tsf) (struct ieee80211_hw *hw);
void (* set_tsf) (struct ieee80211_hw *hw, u64 tsf);
void (* reset_tsf) (struct ieee80211_hw *hw);
int (* tx_last_beacon) (struct ieee80211_hw *hw);
int (* ampdu_action) (struct ieee80211_hw *hw, enum ieee80211_ampdu_mlme
void (* rfkill_poll) (struct ieee80211_hw *hw);
#ifdef CONFIG_NL80211_TESTMODE
int (* testmode_cmd) (struct ieee80211_hw *hw, void *data, int len);
#endif
};
```

## Members

tx

Handler that 802.11 module calls for each transmitted frame. skb contains the buffer starting from the IEEE 802.11 header. The low-level driver should send the frame out based on configuration in the TX control data. This handler should, preferably, never fail and stop queues appropriately, more importantly, however, it must never fail for A-MPDU-queues. This function should return NETDEV\_TX\_OK except in very limited cases. Must be implemented and atomic.

`start`

Called before the first netdevice attached to the hardware is enabled. This should turn on the hardware and must turn on frame reception (for possibly enabled monitor interfaces.) Returns negative error codes, these may be seen in userspace, or zero. When the device is started it should not have a MAC address to avoid acknowledging frames before a non-monitor device is added. Must be implemented.

`stop`

Called after last netdevice attached to the hardware is disabled. This should turn off the hardware (at least it must turn off frame reception.) May be called right after `add_interface` if that rejects an interface. If you added any work onto the `mac80211` workqueue you should ensure to cancel it on this callback. Must be implemented.

`add_interface`

Called when a netdevice attached to the hardware is enabled. Because it is not called for monitor mode devices, `start` and `stop` must be implemented. The driver should perform any initialization it needs before the device can be enabled. The initial configuration for the interface is given in the `conf` parameter. The callback may refuse to add an interface by returning a negative error code (which will be seen in userspace.) Must be implemented.

`remove_interface`

Notifies a driver that an interface is going down. The `stop` callback is called after this if it is the last interface and no monitor interfaces are present. When all interfaces are removed, the MAC address in the hardware must be cleared so the device no longer acknowledges packets, the `mac_addr` member of the `conf` structure is, however, set to the MAC address of the device going away. Hence, this callback must be implemented.

`config`

Handler for configuration requests. IEEE 802.11 code calls this function to change hardware configuration, e.g., channel. This function should never fail but returns a negative error code if it does.

`bss_info_changed`

Handler for configuration requests related to BSS parameters that may vary during BSS's lifespan, and may affect low level driver (e.g. assoc/disassoc status, `erp` parameters). This function should not be used if no BSS has been set, unless for association indication. The `changed` parameter indicates which of the bss parameters has changed when a call is made.

`prepare_multicast`

Prepare for multicast filter configuration. This callback is optional, and its return value is passed to `configure_filter`. This callback must be atomic.

`configure_filter`

Configure the device's RX filter. See the section "Frame filtering" for more information. This callback must be implemented.

`set_tim`

Set TIM bit. `mac80211` calls this function when a TIM bit must be set or cleared for a given STA. Must be atomic.

`set_key`

See the section "Hardware crypto acceleration" This callback can sleep, and is only called between `add_interface` and `remove_interface` calls, i.e. while the given virtual interface is enabled. Returns a negative error code if the key can't be added.

`update_tkip_key`

See the section "Hardware crypto acceleration" This callback will be called in the context of Rx. Called for drivers which set `IEEE80211_KEY_FLAG_TKIP_REQ_RX_P1_KEY`.

`hw_scan`

Ask the hardware to service the scan request, no need to start the scan state machine in stack. The scan must honour the channel configuration done by the regulatory agent in the wiphy's registered bands. The hardware (or the driver) needs to make sure that power save is disabled. The `req ie/ie_len` members are rewritten by `mac80211` to contain the entire IEs after the SSID, so that drivers need not look at these at all but just send them after the SSID -- `mac80211` includes the (extended) supported rates and HT information (where applicable). When the scan finishes, `ieee80211_scan_completed` must be called; note that it also must be called when the scan cannot finish due to any error unless this callback returned a negative error code.

`sw_scan_start`

Notifier function that is called just before a software scan is started. Can be NULL, if the driver doesn't need this notification.

`sw_scan_complete`

Notifier function that is called just after a software scan finished. Can be NULL, if the driver doesn't need this notification.



`get_stats`

Return low-level statistics. Returns zero if statistics are available.

`get_tkip_seq`

If your device implements TKIP encryption in hardware this callback should be provided to read the TKIP transmit IVs (both IV32 and IV16) for the given key from hardware.

`set_rts_threshold`

Configuration of RTS threshold (if device needs it)

`sta_notify`

Notifies low level driver about addition, removal or power state transition of an associated station, AP, IBSS/WDS/mesh peer etc. Must be atomic.

`conf_tx`

Configure TX queue parameters (EDCF (aifs, cw\_min, cw\_max), bursting) for a hardware TX queue. Returns a negative error code on failure.

`get_tx_stats`

Get statistics of the current TX queue status. This is used to get number of currently queued packets (queue length), maximum queue size (limit), and total number of packets sent using each TX queue (count). The 'stats' pointer points to an array that has hw->queues items.

`get_tsf`

Get the current TSF timer value from firmware/hardware. Currently, this is only used for IBSS mode BSSID merging and debugging. Is not a required function.

`set_tsf`

Set the TSF timer to the specified value in the firmware/hardware. Currently, this is only used for IBSS mode debugging. Is not a required function.

`reset_tsf`

Reset the TSF timer and allow firmware/hardware to synchronize with other STAs in the IBSS. This is only used in IBSS mode. This function is optional if the firmware/hardware takes full care of TSF synchronization.

`tx_last_beacon`

Determine whether the last IBSS beacon was sent by us. This is needed only

for IBSS mode and the result of this function is used to determine whether to reply to Probe Requests. Returns non-zero if this device sent the last beacon.

#### `ampdu_action`

Perform a certain A-MPDU action The RA/TID combination determines the destination and TID we want the ampdu action to be performed for. The action is defined through `ieee80211_ampdu_mlme_action`. Starting sequence number (*ssn*) is the first frame we expect to perform the action on. Notice that TX/RX\_STOP can pass NULL for this parameter. Returns a negative error code on failure.

#### `rfkill_poll`

Poll rfkill hardware state. If you need this, you also need to set `wiphy->rfkill_poll` to `true` before registration, and need to call `wiphy_rfkill_set_hw_state` in the callback.

#### `testmode_cmd`

Implement a `cfg80211` test mode command.

## Description

This structure contains various callbacks that the driver may handle or, in some cases, must handle, for example to configure the hardware to a new channel or to transmit a frame.

## `ieee80211_alloc_hw`

### LINUX

Kernel Hackers Manual November 2011

### Name

`ieee80211_alloc_hw` — Allocate a new hardware device

## Synopsis

```
struct ieee80211_hw * ieee80211_alloc_hw (size_t
priv_data_len, const struct ieee80211_ops * ops);
```

## Arguments

*priv\_data\_len*

length of private data

*ops*

callbacks for this device

## Description

This must be called once for each hardware device. The returned pointer must be used to refer to this device when calling other functions. `mac80211` allocates a private data area for the driver pointed to by *priv* in struct `ieee80211_hw`, the size of this area is given as *priv\_data\_len*.

## ieee80211\_register\_hw

### LINUX

Kernel Hackers Manual November 2011

### Name

`ieee80211_register_hw` — Register hardware device

## Synopsis

```
int ieee80211_register_hw (struct ieee80211_hw * hw);
```

## Arguments

*hw*

the device to register as returned by `ieee80211_alloc_hw`

## Description

You must call this function before any other functions in `mac80211`. Note that before a hardware can be registered, you need to fill the contained wiphy's information.

## ieee80211\_get\_tx\_led\_name

### LINUX

Kernel Hackers Manual November 2011

## Name

`ieee80211_get_tx_led_name` — get name of TX LED

## Synopsis

```
char * ieee80211_get_tx_led_name (struct ieee80211_hw * hw);
```

## Arguments

*hw*

the hardware to get the LED trigger name for

## Description

mac80211 creates a transmit LED trigger for each wireless hardware that can be used to drive LEDs if your driver registers a LED device. This function returns the name (or `NULL` if not configured for LEDs) of the trigger so you can automatically link the LED device.

# ieee80211\_get\_rx\_led\_name

## LINUX

Kernel Hackers Manual November 2011

## Name

`ieee80211_get_rx_led_name` — get name of RX LED

## Synopsis

```
char * ieee80211_get_rx_led_name (struct ieee80211_hw * hw);
```

## Arguments

*hw*

the hardware to get the LED trigger name for

## Description

mac80211 creates a receive LED trigger for each wireless hardware that can be used to drive LEDs if your driver registers a LED device. This function returns the name (or `NULL` if not configured for LEDs) of the trigger so you can automatically link the LED device.

## ieee80211\_get\_assoc\_led\_name

### LINUX

Kernel Hackers Manual November 2011

### Name

`ieee80211_get_assoc_led_name` — get name of association LED

### Synopsis

```
char * ieee80211_get_assoc_led_name (struct ieee80211_hw *  
hw);
```

### Arguments

*hw*

the hardware to get the LED trigger name for

### Description

mac80211 creates a association LED trigger for each wireless hardware that can be used to drive LEDs if your driver registers a LED device. This function returns the

name (or `NULL` if not configured for LEDs) of the trigger so you can automatically link the LED device.

## ieee80211\_get\_radio\_led\_name

### LINUX

Kernel Hackers Manual November 2011

### Name

`ieee80211_get_radio_led_name` — get name of radio LED

### Synopsis

```
char * ieee80211_get_radio_led_name (struct ieee80211_hw *  
hw);
```

### Arguments

*hw*

the hardware to get the LED trigger name for

### Description

mac80211 creates a radio change LED trigger for each wireless hardware that can be used to drive LEDs if your driver registers a LED device. This function returns the name (or `NULL` if not configured for LEDs) of the trigger so you can automatically link the LED device.

# ieee80211\_unregister\_hw

## LINUX

Kernel Hackers Manual November 2011

### Name

ieee80211\_unregister\_hw — Unregister a hardware device

### Synopsis

```
void ieee80211_unregister_hw (struct ieee80211_hw * hw);
```

### Arguments

*hw*

the hardware to unregister

### Description

This function instructs mac80211 to free allocated resources and unregister netdevices from the networking subsystem.

# ieee80211\_free\_hw

## LINUX



## Name

`ieee80211_free_hw` — free hardware descriptor

## Synopsis

```
void ieee80211_free_hw (struct ieee80211_hw * hw);
```

## Arguments

*hw*

the hardware to free

## Description

This function frees everything that was allocated, including the private data for the driver. You must call `ieee80211_unregister_hw` before calling this function.



# Chapter 2. PHY configuration

TBD

This chapter should describe PHY handling including start/stop callbacks and the various structures used.

## struct ieee80211\_conf

**LINUX**

Kernel Hackers Manual November 2011

### Name

struct ieee80211\_conf — configuration of the device

### Synopsis

```
struct ieee80211_conf {
    u32 flags;
    int power_level;
    int dynamic_ps_timeout;
    int max_sleep_period;
    u16 listen_interval;
    u8 long_frame_max_tx_count;
    u8 short_frame_max_tx_count;
    struct ieee80211_channel * channel;
    enum nl80211_channel_type channel_type;
};
```

### Members

flags

configuration flags defined above

power\_level

requested transmit power (in dBm)

`dynamic_ps_timeout`

The dynamic powersave timeout (in ms), see the powersave documentation below. This variable is valid only when the CONF\_PS flag is set.

`max_sleep_period`

the maximum number of beacon intervals to sleep for before checking the beacon for a TIM bit (managed mode only); this value will be only achievable between DTIM frames, the hardware needs to check for the multicast traffic bit in DTIM beacons. This variable is valid only when the CONF\_PS flag is set.

`listen_interval`

listen interval in units of beacon interval

`long_frame_max_tx_count`

Maximum number of transmissions for a “long” frame (a frame not RTS protected), called “dot11LongRetryLimit” in 802.11, but actually means the number of transmissions not the number of retries

`short_frame_max_tx_count`

Maximum number of transmissions for a “short” frame, called “dot11ShortRetryLimit” in 802.11, but actually means the number of transmissions not the number of retries

`channel`

the channel to tune to

`channel_type`

the channel (HT) type

## Description

This struct indicates how the driver shall configure the hardware.

## enum ieee80211\_conf\_flags

**LINUX**

## Name

enum ieee80211\_conf\_flags — configuration flags

## Synopsis

```
enum ieee80211_conf_flags {
    IEEE80211_CONF_RADIOTAP,
    IEEE80211_CONF_PS,
    IEEE80211_CONF_IDLE
};
```

## Constants

IEEE80211\_CONF\_RADIOTAP

add radiotap header at receive time (if supported)

IEEE80211\_CONF\_PS

Enable 802.11 power save mode (managed mode only)

IEEE80211\_CONF\_IDLE

The device is running, but idle; if the flag is set the driver should be prepared to handle configuration requests but may turn the device off as much as possible. Typically, this flag will be set when an interface is set UP but not associated or scanning, but it can also be unset in that case when monitor interfaces are active.

## Description

Flags to define PHY configuration options



# Chapter 3. Virtual interfaces

TBD

This chapter should describe virtual interface basics that are relevant to the driver (VLANs, MGMT etc are not.) It should explain the use of the `add_iface/remove_iface` callbacks as well as the interface configuration callbacks.

Things related to AP mode should be discussed there.

Things related to supporting multiple interfaces should be in the appropriate chapter, a BIG FAT note should be here about this though and the recommendation to allow only a single interface in STA mode at first!

## struct ieee80211\_if\_init\_conf

### LINUX

Kernel Hackers Manual November 2011

### Name

`struct ieee80211_if_init_conf` — initial configuration of an interface

### Synopsis

```
struct ieee80211_if_init_conf {
    enum nl80211_iftype type;
    struct ieee80211_vif * vif;
    void * mac_addr;
};
```

### Members

type

one of `enum nl80211_iftype` constants. Determines the type of added/removed interface.

`vif`

pointer to a driver-use per-interface structure. The pointer itself is also used for various functions including `ieee80211_beacon_get` and `ieee80211_get_buffered_bc`.

`mac_addr`

pointer to MAC address of the interface. This pointer is valid until the interface is removed (i.e. it cannot be used after `remove_interface` callback was called for this interface).

## Description

This structure is used in `add_interface` and `remove_interface` callbacks of struct `ieee80211_hw`.

When you allow multiple interfaces to be added to your PHY, take care that the hardware can actually handle multiple MAC addresses. However, also take care that when there's no interface left with `mac_addr != NULL` you remove the MAC address from the device to avoid acknowledging packets in pure monitor mode.



# Chapter 4. Receive and transmit processing

## 4.1. what should be here

TBD

This should describe the receive and transmit paths in mac80211/the drivers as well as transmit status handling.

## 4.2. Frame format

As a general rule, when frames are passed between mac80211 and the driver, they start with the IEEE 802.11 header and include the same octets that are sent over the air except for the FCS which should be calculated by the hardware.

There are, however, various exceptions to this rule for advanced features:

The first exception is for hardware encryption and decryption offload where the IV/ICV may or may not be generated in hardware.

Secondly, when the hardware handles fragmentation, the frame handed to the driver from mac80211 is the MSDU, not the MPDU.

Finally, for received frames, the driver is able to indicate that it has filled a radiotap header and put that in front of the frame; if it does not do so then mac80211 may add this under certain circumstances.

## 4.3. Packet alignment

Drivers always need to pass packets that are aligned to two-byte boundaries to the stack.

Additionally, should, if possible, align the payload data in a way that guarantees that the contained IP header is aligned to a four-byte boundary. In the case of regular frames, this simply means aligning the payload to a four-byte boundary (because either the IP header is directly contained, or IV/RFC1042 headers that have a length divisible by four are in front of it).

With A-MSDU frames, however, the payload data address must yield two modulo four because there are 14-byte 802.3 headers within the A-MSDU frames that push the IP header further back to a multiple of four again. Thankfully, the specs were sane enough this time around to require padding each A-MSDU subframe to a length that is a multiple of four.

Padding like Atheros hardware adds which is inbetween the 802.11 header and the payload is not supported, the driver is required to move the 802.11 header to be directly in front of the payload in that case.

## 4.4. Calling into mac80211 from interrupts

Only `ieee80211_tx_status_irqsafe` and `ieee80211_rx_irqsafe` can be called in hardware interrupt context. The low-level driver must not call any other functions in hardware interrupt context. If there is a need for such call, the low-level driver should first ACK the interrupt and perform the IEEE 802.11 code call after this, e.g. from a scheduled workqueue or even tasklet function.

NOTE: If the driver opts to use the `_irqsafe` functions, it may not also use the non-IRQ-safe functions!

## 4.5. functions/definitions

### struct ieee80211\_rx\_status

#### LINUX

Kernel Hackers Manual November 2011

#### Name

struct `ieee80211_rx_status` — receive status

#### Synopsis

```
struct ieee80211_rx_status {
    u64 mactime;
    enum ieee80211_band band;
```

```
int freq;  
int signal;  
int noise;  
int qual;  
int antenna;  
int rate_idx;  
int flag;  
};
```

## Members

mactime

value in microseconds of the 64-bit Time Synchronization Function (TSF) timer when the first data symbol (MPDU) arrived at the hardware.

band

the active band when this frame was received

freq

frequency the radio was tuned to when receiving this frame, in MHz

signal

signal strength when receiving this frame, either in dBm, in dB or unspecified depending on the hardware capabilities flags `IEEE80211_HW_SIGNAL_*`

noise

noise when receiving this frame, in dBm.

qual

overall signal quality indication, in percent (0-100).

antenna

antenna used

rate\_idx

index of data rate into band's supported rates or MCS index if HT rates are use (RX\_FLAG\_HT)

flag

RX\_FLAG\_\*

## Description

The low-level driver should provide this information (the subset supported by hardware) to the 802.11 code with each received frame, in the skb's control buffer (cb).

## enum mac80211\_rx\_flags

### LINUX

Kernel Hackers Manual November 2011

### Name

enum mac80211\_rx\_flags — receive flags

### Synopsis

```
enum mac80211_rx_flags {
    RX_FLAG_MMIC_ERROR,
    RX_FLAG_DECRYPTED,
    RX_FLAG_RADIOTAP,
    RX_FLAG_MMIC_STRIPPED,
    RX_FLAG_IV_STRIPPED,
    RX_FLAG_FAILED_FCS_CRC,
    RX_FLAG_FAILED_PLCP_CRC,
    RX_FLAG_TSFT,
    RX_FLAG_SHORTPRE,
    RX_FLAG_HT,
    RX_FLAG_40MHZ,
    RX_FLAG_SHORT_GI
};
```

## Constants

### RX\_FLAG\_MMIC\_ERROR

Michael MIC error was reported on this frame. Use together with  
RX\_FLAG\_MMIC\_STRIPPED.

### RX\_FLAG\_DECRYPTED

This frame was decrypted in hardware.

### RX\_FLAG\_RADIOTAP

This frame starts with a radiotap header.

### RX\_FLAG\_MMIC\_STRIPPED

the Michael MIC is stripped off this frame, verification has been done by the  
hardware.

### RX\_FLAG\_IV\_STRIPPED

The IV/ICV are stripped from this frame. If this flag is set, the stack cannot do  
any replay detection hence the driver or hardware will have to do that.

### RX\_FLAG\_FAILED\_FCS\_CRC

Set this flag if the FCS check failed on the frame.

### RX\_FLAG\_FAILED\_PLCP\_CRC

Set this flag if the PCLP check failed on the frame.

### RX\_FLAG\_TSFT

The timestamp passed in the RX status (*mtime* field) is valid. This is useful  
in monitor mode and necessary for beacon frames to enable IBSS merging.

### RX\_FLAG\_SHORTPRE

Short preamble was used for this frame

### RX\_FLAG\_HT

HT MCS was used and *rate\_idx* is MCS index

### RX\_FLAG\_40MHZ

HT40 (40 MHz) was used

RX\_FLAG\_SHORT\_GI

Short guard interval was used

## Description

These flags are used with the *flag* member of struct `ieee80211_rx_status`.

# struct `ieee80211_tx_info`

## LINUX

Kernel Hackers Manual November 2011

## Name

struct `ieee80211_tx_info` — skb transmit information

## Synopsis

```
struct ieee80211_tx_info {
    u32 flags;
    u8 band;
    u8 antenna_sel_tx;
    u8 pad[2];
    union {unnamed_union};
};
```

## Members

`flags`

transmit info flags, defined above

`band`

the band to transmit on (use for checking for races)

antenna\_sel\_tx

antenna to use, 0 for automatic diversity

pad[2]

padding, ignore

{unnamed\_union}

anonymous

## Description

This structure is placed in `skb->cb` for three uses: (1) mac80211 TX control - mac80211 tells the driver what to do (2) driver internal use (if applicable) (3) TX status information - driver tells mac80211 what happened

The TX control's sta pointer is only valid during the `->tx` call, it may be NULL.

## ieee80211\_rx

### LINUX

Kernel Hackers Manual November 2011

## Name

`ieee80211_rx` — receive frame

## Synopsis

```
void ieee80211_rx (struct ieee80211_hw * hw, struct sk_buff *  
skb);
```

## Arguments

*hw*

the hardware this frame came in on

*skb*

the buffer to receive, owned by mac80211 after this call

## Description

Use this function to hand received frames to mac80211. The receive buffer in *skb* must start with an IEEE 802.11 header or a radiotap header if `RX_FLAG_RADIOTAP` is set in the *status* flags.

This function may not be called in IRQ context. Calls to this function for a single hardware must be synchronized against each other. Calls to this function and `ieee80211_rx_irqsafe` may not be mixed for a single hardware.

Note that right now, this function must be called with softirqs disabled.

# ieee80211\_rx\_irqsafe

## LINUX

Kernel Hackers Manual November 2011

## Name

`ieee80211_rx_irqsafe` — receive frame

## Synopsis

```
void ieee80211_rx_irqsafe (struct ieee80211_hw * hw, struct  
sk_buff * skb);
```



## Arguments

*hw*

the hardware this frame came in on

*skb*

the buffer to receive, owned by mac80211 after this call

## Description

Like `ieee80211_rx` but can be called in IRQ context (internally defers to a tasklet.)

Calls to this function and `ieee80211_rx` may not be mixed for a single hardware.

# ieee80211\_tx\_status

## LINUX

Kernel Hackers Manual November 2011

## Name

`ieee80211_tx_status` — transmit status callback

## Synopsis

```
void ieee80211_tx_status (struct ieee80211_hw * hw, struct  
sk_buff * skb);
```

## Arguments

*hw*

the hardware the frame was transmitted by

*skb*

the frame that was transmitted, owned by mac80211 after this call

## Description

Call this function for all transmitted frames after they have been transmitted. It is permissible to not call this function for multicast frames but this can affect statistics.

This function may not be called in IRQ context. Calls to this function for a single hardware must be synchronized against each other. Calls to this function and `ieee80211_tx_status_irqsafe` may not be mixed for a single hardware.

# ieee80211\_tx\_status\_irqsafe

## LINUX

Kernel Hackers Manual November 2011

## Name

`ieee80211_tx_status_irqsafe` — IRQ-safe transmit status callback

## Synopsis

```
void ieee80211_tx_status_irqsafe (struct ieee80211_hw * hw,  
struct sk_buff * skb);
```

## Arguments

*hw*

the hardware the frame was transmitted by

*skb*

the frame that was transmitted, owned by mac80211 after this call

## Description

Like `ieee80211_tx_status` but can be called in IRQ context (internally defers to a tasklet.)

Calls to this function and `ieee80211_tx_status` may not be mixed for a single hardware.

# ieee80211\_rts\_get

## LINUX

Kernel Hackers Manual November 2011

## Name

`ieee80211_rts_get` — RTS frame generation function

## Synopsis

```
void ieee80211_rts_get (struct ieee80211_hw * hw, struct
ieee80211_vif * vif, const void * frame, size_t frame_len,
const struct ieee80211_tx_info * frame_txctl, struct
ieee80211_rts * rts);
```

## Arguments

*hw*

pointer obtained from `ieee80211_alloc_hw`.

*vif*

struct `ieee80211_vif` pointer from struct `ieee80211_if_init_conf`.

*frame*

pointer to the frame that is going to be protected by the RTS.

*frame\_len*

the frame length (in octets).

*frame\_txctl*

struct `ieee80211_tx_info` of the frame.

*rts*

The buffer where to store the RTS frame.

## Description

If the RTS frames are generated by the host system (i.e., not in hardware/firmware), the low-level driver uses this function to receive the next RTS frame from the 802.11 code. The low-level is responsible for calling this function before and RTS frame is needed.

## ieee80211\_rts\_duration

### LINUX

Kernel Hackers Manual November 2011

## Name

`ieee80211_rts_duration` — Get the duration field for an RTS frame

## Synopsis

```
__le16 ieee80211_rts_duration (struct ieee80211_hw * hw,  
struct ieee80211_vif * vif, size_t frame_len, const struct  
ieee80211_tx_info * frame_txctl);
```

## Arguments

*hw*

pointer obtained from `ieee80211_alloc_hw`.

*vif*

struct `ieee80211_vif` pointer from struct `ieee80211_if_init_conf`.

*frame\_len*

the length of the frame that is going to be protected by the RTS.

*frame\_txctl*

struct `ieee80211_tx_info` of the frame.

## Description

If the RTS is generated in firmware, but the host system must provide the duration field, the low-level driver uses this function to receive the duration field value in little-endian byteorder.

## ieee80211\_ctstoself\_get

**LINUX**

## Name

`ieee80211_ctstoself_get` — CTS-to-self frame generation function

## Synopsis

```
void ieee80211_ctstoself_get (struct ieee80211_hw * hw, struct  
ieee80211_vif * vif, const void * frame, size_t frame_len,  
const struct ieee80211_tx_info * frame_txctl, struct  
ieee80211_cts * cts);
```

## Arguments

*hw*

pointer obtained from `ieee80211_alloc_hw`.

*vif*

struct `ieee80211_vif` pointer from struct `ieee80211_if_init_conf`.

*frame*

pointer to the frame that is going to be protected by the CTS-to-self.

*frame\_len*

the frame length (in octets).

*frame\_txctl*

struct `ieee80211_tx_info` of the frame.

*cts*

The buffer where to store the CTS-to-self frame.

## Description

If the CTS-to-self frames are generated by the host system (i.e., not in hardware/firmware), the low-level driver uses this function to receive the next CTS-to-self frame from the 802.11 code. The low-level is responsible for calling this function before and CTS-to-self frame is needed.

## ieee80211\_ctstoself\_duration

### LINUX

Kernel Hackers Manual November 2011

## Name

`ieee80211_ctstoself_duration` — Get the duration field for a CTS-to-self frame

## Synopsis

```
__le16 ieee80211_ctstoself_duration (struct ieee80211_hw * hw,
struct ieee80211_vif * vif, size_t frame_len, const struct
ieee80211_tx_info * frame_txctl);
```

## Arguments

*hw*

pointer obtained from `ieee80211_alloc_hw`.

*vif*

struct `ieee80211_vif` pointer from struct `ieee80211_if_init_conf`.

*frame\_len*

the length of the frame that is going to be protected by the CTS-to-self.

*frame\_txctl*

struct ieee80211\_tx\_info of the frame.

## Description

If the CTS-to-self is generated in firmware, but the host system must provide the duration field, the low-level driver uses this function to receive the duration field value in little-endian byteorder.

# ieee80211\_generic\_frame\_duration

## LINUX

Kernel Hackers Manual November 2011

## Name

`ieee80211_generic_frame_duration` — Calculate the duration field for a frame

## Synopsis

```
__le16 ieee80211_generic_frame_duration (struct ieee80211_hw *  
hw, struct ieee80211_vif * vif, size_t frame_len, struct  
ieee80211_rate * rate);
```

## Arguments

*hw*

pointer obtained from `ieee80211_alloc_hw`.



*vif*

struct ieee80211\_vif pointer from struct ieee80211\_if\_init\_conf.

*frame\_len*

the length of the frame.

*rate*

the rate at which the frame is going to be transmitted.

## Description

Calculate the duration field of some generic frame, given its length and transmission rate (in 100kbps).

# ieee80211\_wake\_queue

## LINUX

Kernel Hackers Manual November 2011

## Name

ieee80211\_wake\_queue — wake specific queue

## Synopsis

```
void ieee80211_wake_queue (struct ieee80211_hw * hw, int  
queue);
```

## Arguments

*hw*

pointer as obtained from `ieee80211_alloc_hw`.

*queue*

queue number (counted from zero).

## Description

Drivers should use this function instead of `netif_wake_queue`.

# ieee80211\_stop\_queue

## LINUX

Kernel Hackers Manual November 2011

## Name

`ieee80211_stop_queue` — stop specific queue

## Synopsis

```
void ieee80211_stop_queue (struct ieee80211_hw * hw, int  
queue);
```

## Arguments

*hw*

pointer as obtained from `ieee80211_alloc_hw`.

*queue*

queue number (counted from zero).

## Description

Drivers should use this function instead of `netif_stop_queue`.

# ieee80211\_wake\_queues

## LINUX

Kernel Hackers Manual November 2011

## Name

`ieee80211_wake_queues` — wake all queues

## Synopsis

```
void ieee80211_wake_queues (struct ieee80211_hw * hw);
```

## Arguments

*hw*

pointer as obtained from `ieee80211_alloc_hw`.

## Description

Drivers should use this function instead of `netif_wake_queue`.

# ieee80211\_stop\_queues

## LINUX

Kernel Hackers Manual November 2011

### Name

`ieee80211_stop_queues` — stop all queues

### Synopsis

```
void ieee80211_stop_queues (struct ieee80211_hw * hw);
```

### Arguments

*hw*

pointer as obtained from `ieee80211_alloc_hw`.

### Description

Drivers should use this function instead of `netif_stop_queue`.

# Chapter 5. Frame filtering

mac80211 requires to see many management frames for proper operation, and users may want to see many more frames when in monitor mode. However, for best CPU usage and power consumption, having as few frames as possible percolate through the stack is desirable. Hence, the hardware should filter as much as possible.

To achieve this, mac80211 uses filter flags (see below) to tell the driver's `configure_filter` function which frames should be passed to mac80211 and which should be filtered out.

Before `configure_filter` is invoked, the `prepare_multicast` callback is invoked with the parameters `mc_count` and `mc_list` for the combined multicast address list of all virtual interfaces. It's use is optional, and it returns a u64 that is passed to `configure_filter`. Additionally, `configure_filter` has the arguments `changed_flags` telling which flags were changed and `total_flags` with the new flag states.

If your device has no multicast address filters your driver will need to check both the `FIF_ALLMULTI` flag and the `mc_count` parameter to see whether multicast frames should be accepted or dropped.

All unsupported flags in `total_flags` must be cleared. Hardware does not support a flag if it is incapable of `_passing_` the frame to the stack. Otherwise the driver must ignore the flag, but not clear it. You must `_only_` clear the flag (announce no support for the flag to mac80211) if you are not able to pass the packet type to the stack (so the hardware always filters it). So for example, you should clear `FIF_CONTROL`, if your hardware always filters control frames. If your hardware always passes control frames to the kernel and is incapable of filtering them, you do `_not_` clear the `FIF_CONTROL` flag. This rule applies to all other FIF flags as well.

## enum ieee80211\_filter\_flags

### LINUX

Kernel Hackers Manual November 2011

### Name

enum ieee80211\_filter\_flags — hardware filter flags

## Synopsis

```
enum ieee80211_filter_flags {  
    FIF_PROMISC_IN_BSS,  
    FIF_ALLMULTI,  
    FIF_FCSFAIL,  
    FIF_PLCPFAIL,  
    FIF_BCN_PRBRESP_PROMISC,  
    FIF_CONTROL,  
    FIF_OTHER_BSS,  
    FIF_PSPOLL  
};
```

## Constants

### FIF\_PROMISC\_IN\_BSS

promiscuous mode within your BSS, think of the BSS as your network segment and then this corresponds to the regular ethernet device promiscuous mode.

### FIF\_ALLMULTI

pass all multicast frames, this is used if requested by the user or if the hardware is not capable of filtering by multicast address.

### FIF\_FCSFAIL

pass frames with failed FCS (but you need to set the RX\_FLAG\_FAILED\_FCS\_CRC for them)

### FIF\_PLCPFAIL

pass frames with failed PLCP CRC (but you need to set the RX\_FLAG\_FAILED\_PLCP\_CRC for them)

### FIF\_BCN\_PRBRESP\_PROMISC

This flag is set during scanning to indicate to the hardware that it should not filter beacons or probe responses by BSSID. Filtering them can greatly reduce the amount of processing mac80211 needs to do and the amount of CPU wakeups, so you should honour this flag if possible.

### FIF\_CONTROL

pass control frames (except for PS Poll), if PROMISC\_IN\_BSS is not set then only those addressed to this station.

**FIF\_OTHER\_BSS**

pass frames destined to other BSSes

**FIF\_PSPOLL**

pass PS Poll frames, if PROMISC\_IN\_BSS is not set then only those addressed to this station.

## **Frame filtering**

These flags determine what the filter in hardware should be programmed to let through and what should not be passed to the stack. It is always safe to pass more frames than requested, but this has negative impact on power consumption.





# II. Advanced driver interface

## Table of Contents

<b>6. Hardware crypto acceleration .....</b>	<b>53</b>
<b>7. Powersave support .....</b>	<b>59</b>
<b>8. Beacon filter support.....</b>	<b>61</b>
<b>9. Multiple queues and QoS support.....</b>	<b>63</b>
<b>10. Access point mode support.....</b>	<b>67</b>
<b>11. Supporting multiple virtual interfaces.....</b>	<b>71</b>
<b>12. Hardware scan offload.....</b>	<b>73</b>

Information contained within this part of the book is of interest only for advanced interaction of mac80211 with drivers to exploit more hardware capabilities and improve performance.



# Chapter 6. Hardware crypto acceleration

mac80211 is capable of taking advantage of many hardware acceleration designs for encryption and decryption operations.

The `set_key` callback in the struct `ieee80211_ops` for a given device is called to enable hardware acceleration of encryption and decryption. The callback takes a `sta` parameter that will be `NULL` for default keys or keys used for transmission only, or point to the station information for the peer for individual keys. Multiple transmission keys with the same key index may be used when VLANs are configured for an access point.

When transmitting, the TX control data will use the `hw_key_idx` selected by the driver by modifying the struct `ieee80211_key_conf` pointed to by the `key` parameter to the `set_key` function.

The `set_key` call for the `SET_KEY` command should return 0 if the key is now in use, `-EOPNOTSUPP` or `-ENOSPC` if it couldn't be added; if you return 0 then `hw_key_idx` must be assigned to the hardware key index, you are free to use the full u8 range.

When the cmd is `DISABLE_KEY` then it must succeed.

Note that it is permissible to not decrypt a frame even if a key for it has been uploaded to hardware, the stack will not make any decision based on whether a key has been uploaded or not but rather based on the receive flags.

The struct `ieee80211_key_conf` structure pointed to by the `key` parameter is guaranteed to be valid until another call to `set_key` removes it, but it can only be used as a cookie to differentiate keys.

In TKIP some HW need to be provided a phase 1 key, for RX decryption acceleration (i.e. iwlwifi). Those drivers should provide `update_tkip_key` handler. The `update_tkip_key` call updates the driver with the new phase 1 key. This happens everytime the iv16 wraps around (every 65536 packets). The `set_key` call will happen only once for each key (unless the AP did rekeying), it will not include a valid phase 1 key. The valid phase 1 key is provided by `update_tkip_key` only. The trigger that makes mac80211 call this handler is software decryption with wrap around of iv16.

# enum set\_key\_cmd

## LINUX

Kernel Hackers Manual November 2011

### Name

enum set\_key\_cmd — key command

### Synopsis

```
enum set_key_cmd {  
    SET_KEY,  
    DISABLE_KEY  
};
```

### Constants

SET\_KEY

a key is set

DISABLE\_KEY

a key must be disabled

### Description

Used with the `set_key` callback in struct `ieee80211_ops`, this indicates whether a key is being removed or added.

# struct ieee80211\_key\_conf

## LINUX

## Name

`struct ieee80211_key_conf` — key information

## Synopsis

```
struct ieee80211_key_conf {
    enum ieee80211_key_alg alg;
    u8 icv_len;
    u8 iv_len;
    u8 hw_key_idx;
    u8 flags;
    s8 keyidx;
    u8 keylen;
    u8 key[0];
};
```

## Members

`alg`

The key algorithm.

`icv_len`

The ICV length for this key type

`iv_len`

The IV length for this key type

`hw_key_idx`

To be set by the driver, this is the key index the driver wants to be given when a frame is transmitted and needs to be encrypted in hardware.

`flags`

key flags, see `enum ieee80211_key_flags`.

`keyidx`

the key index (0-3)

keylen

key material length

key[0]

key material. For ALG\_TKIP the key is encoded as a 256-bit (32 byte)

## Description

This key information is given by mac80211 to the driver by the `set_key` callback in struct `ieee80211_ops`.

## data block

- Temporal Encryption Key (128 bits) - Temporal Authenticator Tx MIC Key (64 bits) - Temporal Authenticator Rx MIC Key (64 bits)

## enum ieee80211\_key\_alg

### LINUX

Kernel Hackers Manual November 2011

## Name

enum `ieee80211_key_alg` — key algorithm

## Synopsis

```
enum ieee80211_key_alg {  
    ALG_WEP,  
    ALG_TKIP,  
    ALG_CCMP,  
    ALG_AES_CMAC  
};
```

## Constants

ALG\_WEP

WEP40 or WEP104

ALG\_TKIP

TKIP

ALG\_CCMP

CCMP (AES)

ALG\_AES\_CMAC

AES-128-CMAC

## enum ieee80211\_key\_flags

### LINUX

Kernel Hackers Manual November 2011

## Name

enum ieee80211\_key\_flags — key flags

## Synopsis

```
enum ieee80211_key_flags {  
    IEEE80211_KEY_FLAG_WMM_STA,  
    IEEE80211_KEY_FLAG_GENERATE_IV,  
    IEEE80211_KEY_FLAG_GENERATE_MMIC,  
    IEEE80211_KEY_FLAG_PAIRWISE,  
    IEEE80211_KEY_FLAG_SW_MGMT  
};
```

## Constants

### IEEE80211\_KEY\_FLAG\_WMM\_STA

Set by mac80211, this flag indicates that the STA this key will be used with could be using QoS.

### IEEE80211\_KEY\_FLAG\_GENERATE\_IV

This flag should be set by the driver to indicate that it requires IV generation for this particular key.

### IEEE80211\_KEY\_FLAG\_GENERATE\_MMIC

This flag should be set by the driver for a TKIP key if it requires Michael MIC generation in software.

### IEEE80211\_KEY\_FLAG\_PAIRWISE

Set by mac80211, this flag indicates that the key is pairwise rather than a shared key.

### IEEE80211\_KEY\_FLAG\_SW\_MGMT

This flag should be set by the driver for a CCMP key if it requires CCMP encryption of management frames (MFP) to be done in software.

## Description

These flags are used for communication about keys between the driver and mac80211, with the *flags* parameter of struct `ieee80211_key_conf`.



# Chapter 7. Powersave support

mac80211 has support for various powersave implementations.

First, it can support hardware that handles all powersaving by itself, such hardware should simply set the `IEEE80211_HW_SUPPORTS_PS` hardware flag. In that case, it will be told about the desired powersave mode depending on the association status, and the driver must take care of sending nullfunc frames when necessary, i.e. when entering and leaving powersave mode. The driver is required to look at the AID in beacons and signal to the AP that it woke up when it finds traffic directed to it. This mode supports dynamic PS by simply enabling/disabling PS.

Additionally, such hardware may set the `IEEE80211_HW_SUPPORTS_DYNAMIC_PS` flag to indicate that it can support dynamic PS mode itself (see below).

Other hardware designs cannot send nullfunc frames by themselves and also need software support for parsing the TIM bitmap. This is also supported by mac80211 by combining the `IEEE80211_HW_SUPPORTS_PS` and `IEEE80211_HW_PS_NULLFUNC_STACK` flags. The hardware is of course still required to pass up beacons. The hardware is still required to handle waking up for multicast traffic; if it cannot the driver must handle that as best as it can, mac80211 is too slow.

Dynamic powersave mode is an extension to normal powersave mode in which the hardware stays awake for a user-specified period of time after sending a frame so that reply frames need not be buffered and therefore delayed to the next wakeup. This can either be supported by hardware, in which case the driver needs to look at the `dynamic_ps_timeout` hardware configuration value, or by the stack if all nullfunc handling is in the stack.



# Chapter 8. Beacon filter support

Some hardware have beacon filter support to reduce host cpu wakeups which will reduce system power consumption. It usually works so that the firmware creates a checksum of the beacon but omits all constantly changing elements (TSF, TIM etc). Whenever the checksum changes the beacon is forwarded to the host, otherwise it will be just dropped. That way the host will only receive beacons where some relevant information (for example ERP protection or WMM settings) have changed.

Beacon filter support is advertised with the `IEEE80211_HW_BEACON_FILTER` hardware capability. The driver needs to enable beacon filter support whenever power save is enabled, that is `IEEE80211_CONF_PS` is set. When power save is enabled, the stack will not check for beacon loss and the driver needs to notify about loss of beacons with `ieee80211_beacon_loss`.

The time (or number of beacons missed) until the firmware notifies the driver of a beacon loss event (which in turn causes the driver to call `ieee80211_beacon_loss`) should be configurable and will be controlled by `mac80211` and the roaming algorithm in the future.

Since there may be constantly changing information elements that nothing in the software stack cares about, we will, in the future, have `mac80211` tell the driver which information elements are interesting in the sense that we want to see changes in them. This will include - a list of information element IDs - a list of OUIs for the vendor information element

Ideally, the hardware would filter out any beacons without changes in the requested elements, but if it cannot support that it may, at the expense of some efficiency, filter out only a subset. For example, if the device doesn't support checking for OUIs it should pass up all changes in all vendor information elements.

Note that change, for the sake of simplification, also includes information elements appearing or disappearing from the beacon.

Some hardware supports an "ignore list" instead, just make sure nothing that was requested is on the ignore list, and include commonly changing information element IDs in the ignore list, for example 11 (BSS load) and the various vendor-assigned IEs with unknown contents (128, 129, 133-136, 149, 150, 155, 156, 173, 176, 178, 179, 219); for forward compatibility it could also include some currently unused IDs.

In addition to these capabilities, hardware should support notifying the host of changes in the beacon RSSI. This is relevant to implement roaming when no traffic is flowing (when traffic is flowing we see the RSSI of the received data packets). This can consist in notifying the host when the RSSI changes significantly or when it drops below or rises above configurable thresholds. In the future these thresholds

will also be configured by mac80211 (which gets them from userspace) to implement them as the roaming algorithm requires.

If the hardware cannot implement this, the driver should ask it to periodically pass beacon frames to the host so that software can do the signal strength threshold checking.

## ieee80211\_beacon\_loss

### LINUX

Kernel Hackers Manual November 2011

### Name

`ieee80211_beacon_loss` — inform hardware does not receive beacons

### Synopsis

```
void ieee80211_beacon_loss (struct ieee80211_vif * vif);
```

### Arguments

*vif*

struct `ieee80211_vif` pointer from struct `ieee80211_if_init_conf`.

### Description

When beacon filtering is enabled with `IEEE80211_HW_BEACON_FILTERING` and `IEEE80211_CONF_PS` is set, the driver needs to inform whenever the hardware is not receiving beacons with this function.

# Chapter 9. Multiple queues and QoS support

TBD

## struct ieee80211\_tx\_queue\_params

**LINUX**

Kernel Hackers Manual November 2011

### Name

struct ieee80211\_tx\_queue\_params — transmit queue configuration

### Synopsis

```
struct ieee80211_tx_queue_params {
    u16 txop;
    u16 cw_min;
    u16 cw_max;
    u8 aifs;
};
```

### Members

txop

maximum burst time in units of 32 usecs, 0 meaning disabled

cw\_min

minimum contention window [a value of the form  $2^n - 1$  in the range 1..32767]

cw\_max

maximum contention window [like *cw\_min*]

aifs

arbitration interframe space [0..255]

## Description

The information provided in this structure is required for QoS transmit queue configuration. Cf. IEEE 802.11 7.3.2.29.

# struct ieee80211\_tx\_queue\_stats

## LINUX

Kernel Hackers Manual November 2011

## Name

struct ieee80211\_tx\_queue\_stats — transmit queue statistics

## Synopsis

```
struct ieee80211_tx_queue_stats {  
    unsigned int len;  
    unsigned int limit;  
    unsigned int count;  
};
```

## Members

len

number of packets in queue

limit

queue length limit

count

number of frames sent





# Chapter 10. Access point mode support

TBD

Some parts of the `if_conf` should be discussed here instead

Insert notes about VLAN interfaces with hw crypto here or in the hw crypto chapter.

## ieee80211\_get\_buffered\_bc

### LINUX

Kernel Hackers Manual November 2011

### Name

`ieee80211_get_buffered_bc` — accessing buffered broadcast and multicast frames

### Synopsis

```
struct sk_buff * ieee80211_get_buffered_bc (struct  
ieee80211_hw * hw, struct ieee80211_vif * vif);
```

### Arguments

*hw*

pointer as obtained from `ieee80211_alloc_hw`.

*vif*

struct `ieee80211_vif` pointer from struct `ieee80211_if_init_conf`.

## Description

Function for accessing buffered broadcast and multicast frames. If hardware/firmware does not implement buffering of broadcast/multicast frames when power saving is used, 802.11 code buffers them in the host memory. The low-level driver uses this function to fetch next buffered frame. In most cases, this is used when generating beacon frame. This function returns a pointer to the next buffered skb or NULL if no more buffered frames are available.

## Note

buffered frames are returned only after DTIM beacon frame was generated with `ieee80211_beacon_get` and the low-level driver must thus call `ieee80211_beacon_get` first. `ieee80211_get_buffered_bc` returns NULL if the previous generated beacon was not DTIM, so the low-level driver does not need to check for DTIM beacons separately and should be able to use common code for all beacons.

# ieee80211\_beacon\_get

## LINUX

Kernel Hackers Manual November 2011

## Name

`ieee80211_beacon_get` — beacon generation function

## Synopsis

```
struct sk_buff * ieee80211_beacon_get (struct ieee80211_hw *  
hw, struct ieee80211_vif * vif);
```

## Arguments

*hw*

pointer obtained from `ieee80211_alloc_hw`.

*vif*

struct `ieee80211_vif` pointer from struct `ieee80211_if_init_conf`.

## Description

If the beacon frames are generated by the host system (i.e., not in hardware/firmware), the low-level driver uses this function to receive the next beacon frame from the 802.11 code. The low-level is responsible for calling this function before beacon data is needed (e.g., based on hardware interrupt). Returned `skb` is used only once and low-level driver is responsible for freeing it.



# Chapter 11. Supporting multiple virtual interfaces

TBD

Note: WDS with identical MAC address should almost always be OK

Insert notes about having multiple virtual interfaces with different MAC addresses here, note which configurations are supported by mac80211, add notes about supporting hw crypto with it.



# Chapter 12. Hardware scan offload

TBD

## ieee80211\_scan\_completed

### LINUX

Kernel Hackers Manual November 2011

### Name

`ieee80211_scan_completed` — completed hardware scan

### Synopsis

```
void ieee80211_scan_completed (struct ieee80211_hw * hw, bool  
aborted);
```

### Arguments

*hw*

the hardware that finished the scan

*aborted*

set to true if scan was aborted

### Description

When hardware scan offload is used (i.e. the `hw_scan` callback is assigned) this function needs to be called by the driver to notify mac80211 that the scan finished.





# III. Rate control interface

## Table of Contents

13. dummy chapter .....	77
-------------------------	----

TBD

This part of the book describes the rate control algorithm interface and how it relates to mac80211 and drivers.



# Chapter 13. dummy chapter

TBD



# IV. Internals

## Table of Contents

<b>14. Key handling.....</b>	<b>81</b>
<b>15. Receive processing.....</b>	<b>83</b>
<b>16. Transmit processing .....</b>	<b>85</b>
<b>17. Station info handling.....</b>	<b>87</b>
<b>18. Synchronisation.....</b>	<b>97</b>

TBD

This part of the book describes mac80211 internals.



# Chapter 14. Key handling

## 14.1. Key handling basics

Key handling in mac80211 is done based on per-interface (`sub_if_data`) keys and per-station keys. Since each station belongs to an interface, each station key also belongs to that interface.

Hardware acceleration is done on a best-effort basis, for each key that is eligible the hardware is asked to enable that key but if it cannot do that the key is simply kept for software encryption. There is currently no way of knowing this except by looking into debugfs.

All key operations are protected internally so you can call them at any time.

Within mac80211, key references are, just as STA structure references, protected by RCU. Note, however, that some things are unprotected, namely the `key->sta` dereferences within the hardware acceleration functions. This means that `sta_info_destroy` must flush the key todo list.

All the direct key list manipulation functions must not sleep because they can operate on STA info structs that are protected by RCU.

## 14.2. MORE TBD

TBD





# Chapter 15. Receive processing

TBD



# Chapter 16. Transmit processing

TBD



# Chapter 17. Station info handling

## 17.1. Programming information

### struct sta\_info

#### LINUX

Kernel Hackers Manual November 2011

#### Name

struct sta\_info — STA information

#### Synopsis

```
struct sta_info {
    struct list_head list;
    struct sta_info * hnext;
    struct ieee80211_local * local;
    struct ieee80211_sub_if_data * sdata;
    struct ieee80211_key * key;
    struct rate_control_ref * rate_ctrl;
    void * rate_ctrl_priv;
    spinlock_t lock;
    spinlock_t flaglock;
    u16 listen_interval;
    u8 pin_status;
    u32 flags;
    struct sk_buff_head ps_tx_buf;
    struct sk_buff_head tx_filtered;
    unsigned long rx_packets;
    unsigned long rx_bytes;
    unsigned long wep_weak_iv_count;
    unsigned long last_rx;
    unsigned long num_duplicates;
    unsigned long rx_fragments;
    unsigned long rx_dropped;
    int last_signal;
    int last_qual;
```

```
int last_noise;
__le16 last_seq_ctrl[NUM_RX_DATA_QUEUES];
unsigned long tx_filtered_count;
unsigned long tx_retry_failed;
unsigned long tx_retry_count;
unsigned int fail_avg;
unsigned long tx_packets;
unsigned long tx_bytes;
unsigned long tx_fragments;
struct ieee80211_tx_rate last_tx_rate;
u16 tid_seq[IEEE80211_QOS_CTL_TID_MASK + 1];
struct sta_ampdu_mlme ampdu_mlme;
u8 timer_to_tid[STA_TID_NUM];
#ifdef CONFIG_MAC80211_MESH
    __le16 llid;
    __le16 plid;
    __le16 reason;
    u8 plink_retries;
    bool ignore_plink_timer;
    bool plink_timer_was_running;
    enum plink_state plink_state;
    u32 plink_timeout;
    struct timer_list plink_timer;
#endif
#ifdef CONFIG_MAC80211_DEBUGFS
    struct sta_info_debugfsdentries debugfs;
#endif
    struct ieee80211_sta sta;
};
```

## Members

list

global linked list entry

hnext

hash table linked list pointer

local

pointer to the global information

sdata

virtual interface this station belongs to

key  
peer key negotiated with this station, if any

rate\_ctrl  
rate control algorithm reference

rate\_ctrl\_priv  
rate control private per-STA pointer

lock  
used for locking all fields that require locking, see comments in the header file.

flaglock  
spinlock for flags accesses

listen\_interval  
listen interval of this station, when we're acting as AP

pin\_status  
used internally for pinning a STA struct into memory

flags  
STA flags, see enum `ieee80211_sta_info_flags`

ps\_tx\_buf  
buffer of frames to transmit to this station when it leaves power saving state

tx\_filtered  
buffer of frames we already tried to transmit but were filtered by hardware due to STA having entered power saving state

rx\_packets  
Number of MSDUs received from this STA

rx\_bytes  
Number of bytes received from this STA

wep\_weak\_iv\_count  
number of weak WEP IVs received from this station

## *Chapter 17. Station info handling*

last\_rx

time (in jiffies) when last frame was received from this STA

num\_duplicates

number of duplicate frames received from this STA

rx\_fragments

number of received MPDUs

rx\_dropped

number of dropped MPDUs from this STA

last\_signal

signal of last received frame from this STA

last\_qual

qual of last received frame from this STA

last\_noise

noise of last received frame from this STA

last\_seq\_ctrl[NUM\_RX\_DATA\_QUEUES]

last received seq/frag number from this STA (per RX queue)

tx\_filtered\_count

number of frames the hardware filtered for this STA

tx\_retry\_failed

number of frames that failed retry

tx\_retry\_count

total number of retries for frames to this STA

fail\_avg

moving percentage of failed MSDUs

tx\_packets

number of RX/TX MSDUs

tx\_bytes

number of bytes transmitted to this STA



tx\_fragments

number of transmitted MPDUs

last\_tx\_rate

rate used for last transmit, to report to userspace as “the” transmit rate

tid\_seq[IEEE80211\_QOS\_CTL\_TID\_MASK + 1]

per-TID sequence numbers for sending to this STA

ampdu\_mlme

A-MPDU state machine state

timer\_to\_tid[STA\_TID\_NUM]

identity mapping to ID timers

llid

Local link ID

plid

Peer link ID

reason

Cancel reason on PLINK\_HOLDING state

plink\_retries

Retries in establishment

ignore\_plink\_timer

ignore the peer-link timer (used internally)

plink\_timer\_was\_running

used by suspend/resume to restore timers

plink\_state

peer link state

plink\_timeout

timeout of peer link

plink\_timer

peer link watch timer

debugfs

debug filesystem info

sta

station information we share with the driver

## Description

This structure collects information about a station that mac80211 is communicating with.

## enum ieee80211\_sta\_info\_flags

### LINUX

Kernel Hackers Manual November 2011

## Name

enum ieee80211\_sta\_info\_flags — Stations flags

## Synopsis

```
enum ieee80211_sta_info_flags {
    WLAN_STA_AUTH,
    WLAN_STA_ASSOC,
    WLAN_STA_PS,
    WLAN_STA_AUTHORIZED,
    WLAN_STA_SHORT_PREAMBLE,
    WLAN_STA_ASSOC_AP,
    WLAN_STA_WME,
    WLAN_STA_WDS,
    WLAN_STA_CLEAR_PS_FILT,
    WLAN_STA_MFP,
    WLAN_STA_SUSPEND
};
```

## Constants

WLAN\_STA\_AUTH

Station is authenticated.

WLAN\_STA\_ASSOC

Station is associated.

WLAN\_STA\_PS

Station is in power-save mode

WLAN\_STA\_AUTHORIZED

Station is authorized to send/receive traffic. This bit is always checked so needs to be enabled for all stations when virtual port control is not in use.

WLAN\_STA\_SHORT\_PREAMBLE

Station is capable of receiving short-preamble frames.

WLAN\_STA\_ASSOC\_AP

We're associated to that station, it is an AP.

WLAN\_STA\_WME

Station is a QoS-STA.

WLAN\_STA\_WDS

Station is one of our WDS peers.

WLAN\_STA\_CLEAR\_PS\_FILT

Clear PS filter in hardware (using the IEEE80211\_TX\_CTL\_CLEAR\_PS\_FILT control flag) when the next frame to this station is transmitted.

WLAN\_STA\_MFP

Management frame protection is used with this STA.

WLAN\_STA\_SUSPEND

Set/cleared during a suspend/resume cycle. Used to deny ADDBA requests (both TX and RX).

## Description

These flags are used with struct `sta_info`'s `flags` member.

## 17.2. STA information lifetime rules

STA info structures (struct `sta_info`) are managed in a hash table for faster lookup and a list for iteration. They are managed using RCU, i.e. access to the list and hash table is protected by RCU.

Upon allocating a STA info structure with `sta_info_alloc`, the caller owns that structure. It must then either destroy it using `sta_info_destroy` (which is pretty useless) or insert it into the hash table using `sta_info_insert` which demotes the reference from ownership to a regular RCU-protected reference; if the function is called without protection by an RCU critical section the reference is instantly invalidated. Note that the caller may not do much with the STA info before inserting it, in particular, it may not start any mesh peer link management or add encryption keys.

When the insertion fails (`sta_info_insert` returns non-zero), the structure will have been freed by `sta_info_insert`!

sta entries are added by `mac80211` when you establish a link with a peer. This means different things for the different type of interfaces we support. For a regular station this mean we add the AP sta when we receive an association response from the AP. For IBSS this occurs when we receive a probe response or a beacon from target IBSS network. For WDS we add the sta for the peer immediately upon device open. When using AP mode we add stations for each respective station upon request from userspace through `nl80211`.

Because there are debugfs entries for each station, and adding those must be able to sleep, it is also possible to “pin” a station entry, that means it can be removed from the hash table but not be freed. See the comment in `__sta_info_unlink` for more information, this is an internal capability only.

In order to remove a STA info structure, the caller needs to first unlink it (`sta_info_unlink`) from the list and hash tables and then destroy it; `sta_info_destroy` will wait for an RCU grace period to elapse before actually freeing it. Due to the pinning and the possibility of multiple callers trying to remove the same STA info at the same time, `sta_info_unlink` can clear the STA info pointer it is passed to indicate that the STA info is owned by somebody else now.

If `sta_info_unlink` did not clear the pointer then the caller owns the STA info structure now and is responsible of destroying it with a call to `sta_info_destroy`.

In all other cases, there is no concept of ownership on a STA entry, each structure is owned by the global hash table/list until it is removed. All users of the structure need to be RCU protected so that the structure won't be freed before they are done using it.



# Chapter 18. Synchronisation

TBD

Locking, lots of RCU

