The 802.11 subsystems – for kernel developers

Explaining wireless 802.11 networking in the Linux kernel

Johannes Berg <johannes@sipsolutions.net>
Copyright © 2007-2009 Johannes Berg

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.

Abstract

These books attempt to give a description of the various subsystems that play a role in 802.11 wireless networking in Linux. Since these books are for kernel developers they attempts to document the structures and functions used in the kernel as well as giving a higher-level overview.

The reader is expected to be familiar with the 802.11 standard as published by the IEEE in 802.11-2007 (or possibly later versions). References to this standard will be given as "802.11-2007 8.1.5".

Table of Contents

The cfg80211 subsystem vi
1. Device registration 1
enum ieee80211_band 2
enum ieee80211_channel_flags 3
struct ieee80211_channel 5
enum ieee80211_rate_flags 7
struct ieee80211_rate 8
struct ieee80211_sta_ht_cap 9
struct ieee80211_supported_band 10
enum cfg80211_signal_type 11
enum wiphy_params_flags 12
enum wiphy_flags 13
struct wiphy 15
struct wireless_dev 20
wiphy_new 23
wiphy_register 24
wiphy_unregister 25
wiphy_free 26
wiphy_name 27
wiphy_dev 28
wiphy_priv 29
priv_to_wiphy 30
set_wiphy_dev 31
wdev_priv 32
struct ieee80211_iface_limit 33
struct ieee80211_iface_combination 34
cfg80211_check_combinations 36
2. Actions and configuration 37
struct cfg80211_ops 38
struct vif_params 46
struct key_params 47
enum survey_info_flags 48
struct survey_info 49
struct cfg80211_beacon_data 50
struct cfg80211_ap_settings 51
struct station_parameters 53
enum rate_info_flags 55
struct rate_info 56
struct station_info 57
enum monitor_flags 60
enum mpath_info_flags 61
struct mpath_info 62
struct bss_parameters 63
struct ieee80211_txq_params 64
struct cfg80211_crypto_settings 65
struct cfg80211_auth_request 66
struct cfg80211_assoc_request 67
struct cfg80211_deauth_request 68
struct cfg80211_disassoc_request 69
struct cfg80211_ibss_params 70
struct cfg80211_connect_params 72

struct cfg80211_pmksa 74
cfg80211_rx_mlme_mgmt 75
cfg80211_auth_timeout 76
cfg80211_rx_assoc_resp 77
cfg80211_assoc_timeout 78
cfg80211_tx_mlme_mgmt 79
cfg80211_ibss_joined 80
cfg80211_connect_result 81
cfg80211_roamed 82
cfg80211_disconnected 83
cfg80211_ready_on_channel 84
cfg80211_remain_on_channel_expired 85
cfg80211_new_sta 86
cfg80211_rx_mgmt 87
cfg80211_mgmt_tx_status 88
cfg80211_cqm_rssi_notify 89
cfg80211_cqm_pktloss_notify 90
cfg80211_michael_mic_failure 91
3. Scanning and BSS list handling 92
struct cfg80211_ssid 93
struct cfg80211_scan_request 94
cfg80211_scan_done 96
struct cfg80211_bss 97
cfg80211_inform_bss_width_frame 99
cfg80211_inform_bss_width 100
cfg80211_unlink_bss 101
cfg80211_find_ie 102
ieee80211_bss_get_ie 103
4. Utility functions 104
ieee80211_channel_to_frequency 105
ieee80211_frequency_to_channel 106
ieee80211_get_channel 107
ieee80211_get_response_rate 108
ieee80211_hdrlen 109
ieee80211_get_hdrlen_from_skb 110
struct ieee80211_radiotap_iterator 111
5. Data path helpers 113
ieee80211_data_to_8023 114
ieee80211_data_from_8023 115
ieee80211_amsdu_to_8023s 116
cfg80211_classify8021d 117
6. Regulatory enforcement infrastructure 118
regulatory_hint 119
wiphy_apply_custom_regulatory 120
freq_reg_info 121
7. RFkill integration 122
wiphy_rfkill_set_hw_state 123
wiphy_rfkill_start_polling 124
wiphy_rfkill_stop_polling 125
8. Test mode 126
cfg80211_testmode_alloc_reply_skb 127
cfg80211_testmode_reply 128
cfg80211_testmode_alloc_event_skb 129
cfg80211_testmode_event 130

The mac80211 subsystem cxxxi
I. The basic mac80211 driver interface 1
1. Basic hardware handling 3
struct ieee80211_hw 4
enum ieee80211_hw_flags 7
SET_IEEE80211_DEV 10
SET_IEEE80211_PERM_ADDR 11
struct ieee80211_ops 12
ieee80211_alloc_hw 23
ieee80211_register_hw 24
ieee80211_unregister_hw 25
ieee80211_free_hw 26
2. PHY configuration 27
struct ieee80211_conf 28
enum ieee80211_conf_flags 30
3. Virtual interfaces 31
struct ieee80211_vif 32
4. Receive and transmit processing 34
what should be here 34
Frame format 34
Packet alignment 34
Calling into mac80211 from interrupts 34
functions/definitions 35
5. Frame filtering 65
enum ieee80211_filter_flags 66
6. The mac80211 workqueue 67
ieee80211_queue_work 68
ieee80211_queue_delayed_work 69
II. Advanced driver interface 70
7. LED support 72
ieee80211_get_tx_led_name 73
ieee80211_get_rx_led_name 74
ieee80211_get_assoc_led_name 75
ieee80211_get_radio_led_name 76
struct ieee80211_tpt_blink 77
enum ieee80211_tpt_led_trigger_flags 78
ieee80211_create_tpt_led_trigger 79
8. Hardware crypto acceleration 80
enum set_key_cmd 81
struct ieee80211_key_conf 82
enum ieee80211_key_flags 83
ieee80211_get_tkip_p1k 85
ieee80211_get_tkip_p1k_iv 86
ieee80211_get_tkip_p2k 87
9. Powersave support 88
10. Beacon filter support 89
ieee80211_beacon_loss 90
11. Multiple queues and QoS support 91
struct ieee80211_tx_queue_params 92
12. Access point mode support 93
support for powersaving clients 93
13. Supporting multiple virtual interfaces 103
ieee80211_iterate_active_interfaces 104
ieee80211_iterate_active_interfaces_atomic 105

14. Station handling 106
struct ieee80211_sta 107
enum sta_notify_cmd 109
ieee80211_find_sta 110
ieee80211_find_sta_by_ifaddr 111
15. Hardware scan offload 112
ieee80211_scan_completed 113
16. Aggregation 114
TX A-MPDU aggregation 114
RX A-MPDU aggregation 114
17. Spatial Multiplexing Powersave (SMPS) 116
ieee80211_request_smps 117
enum ieee80211_smps_mode 118
III. Rate control interface 119
18. Rate Control API 121
ieee80211_start_tx_ba_session 122
ieee80211_start_tx_ba_cb_irqsafe 123
ieee80211_stop_tx_ba_session 124
ieee80211_stop_tx_ba_cb_irqsafe 125
enum ieee80211_rate_control_changed 126
struct ieee80211_tx_rate_control 127
rate_control_send_low 128
IV. Internals 129
19. Key handling 131
Key handling basics 131
MORE TBD 131
20. Receive processing 132
21. Transmit processing 133
22. Station info handling 134
Programming information 134
STA information lifetime rules 141
23. Aggregation 143
struct sta_ampdu_mlme 144
struct tid_ampdu_tx 145
struct tid_ampdu_rx 147
24. Synchronisation 149

The cfg80211 subsystem

The cfg80211 subsystem

Abstract

cfg80211 is the configuration API for 802.11 devices in Linux. It bridges userspace and drivers, and offers some utility functionality associated with 802.11. cfg80211 must, directly or indirectly via mac80211, be used by all modern wireless drivers in Linux, so that they offer a consistent API through nl80211. For backward compatibility, cfg80211 also offers wireless extensions to userspace, but hides them from drivers completely.

Additionally, cfg80211 contains code to help enforce regulatory spectrum use restrictions.

Table of Contents

1. Device registration 1
    - enum ieee80211_band 2
    - enum ieee80211_channel_flags 3
    - struct ieee80211_channel 5
    - enum ieee80211_rate_flags 7
    - struct ieee80211_rate 8
    - struct ieee80211_sta_ht_cap 9
    - struct ieee80211_supported_band 10
    - enum cfg80211_signal_type 11
    - enum wiphy_params_flags 12
    - enum wiphy_flags 13
    - struct wiphy 15
    - struct wireless_dev 20
    - wiphy_new 23
    - wiphy_register 24
    - wiphy_unregister 25
    - wiphy_free 26
    - wiphy_name 27
    - wiphy_dev 28
    - wiphy_priv 29
    - priv_to_wiphy 30
    - set_wiphy_dev 31
    - wdev_priv 32
    - struct ieee80211_iface_limit 33
    - struct ieee80211_iface_combination 34
    - cfg80211_check_combinations 36
2. Actions and configuration 37
    - struct cfg80211_ops 38
    - struct vif_params 46
    - struct key_params 47
    - enum survey_info_flags 48
    - struct survey_info 49
    - struct cfg80211_beacon_data 50
    - struct cfg80211_ap_settings 51
    - struct station_parameters 53
    - enum rate_info_flags 55
    - struct rate_info 56
    - struct station_info 57
    - enum monitor_flags 60
    - enum mpath_info_flags 61
    - struct mpath_info 62
    - struct bss_parameters 63
    - struct ieee80211_txq_params 64
    - struct cfg80211_crypto_settings 65
    - struct cfg80211_auth_request 66
    - struct cfg80211_assoc_request 67
    - struct cfg80211_deauth_request 68
    - struct cfg80211_disassoc_request 69
    - struct cfg80211_ibss_params 70
    - struct cfg80211_connect_params 72
    - struct cfg80211_pmksa 74

cfg80211_rx_mlme_mgmt 75
cfg80211_auth_timeout 76
cfg80211_rx_assoc_resp 77
cfg80211_assoc_timeout 78
cfg80211_tx_mlme_mgmt 79
cfg80211_ibss_joined 80
cfg80211_connect_result 81
cfg80211_roamed 82
cfg80211_disconnected 83
cfg80211_ready_on_channel 84
cfg80211_remain_on_channel_expired 85
cfg80211_new_sta 86
cfg80211_rx_mgmt 87
cfg80211_mgmt_tx_status 88
cfg80211_cqm_rssi_notify 89
cfg80211_cqm_pktloss_notify 90
cfg80211_michael_mic_failure 91
3. Scanning and BSS list handling 92
struct cfg80211_ssid 93
struct cfg80211_scan_request 94
cfg80211_scan_done 96
struct cfg80211_bss 97
cfg80211_inform_bss_width_frame 99
cfg80211_inform_bss_width 100
cfg80211_unlink_bss 101
cfg80211_find_ie 102
ieee80211_bss_get_ie 103
4. Utility functions 104
ieee80211_channel_to_frequency 105
ieee80211_frequency_to_channel 106
ieee80211_get_channel 107
ieee80211_get_response_rate 108
ieee80211_hdrlen 109
ieee80211_get_hdrlen_from_skb 110
struct ieee80211_radiotap_iterator 111
5. Data path helpers 113
ieee80211_data_to_8023 114
ieee80211_data_from_8023 115
ieee80211_amsdu_to_8023s 116
cfg80211_classify8021d 117
6. Regulatory enforcement infrastructure 118
regulatory_hint 119
wiphy_apply_custom_regulatory 120
freq_reg_info 121
7. RFkill integration 122
wiphy_rfkill_set_hw_state 123
wiphy_rfkill_start_polling 124
wiphy_rfkill_stop_polling 125
8. Test mode 126
cfg80211_testmode_alloc_reply_skb 127
cfg80211_testmode_reply 128
cfg80211_testmode_alloc_event_skb 129
cfg80211_testmode_event 130

Chapter 1. Device registration

In order for a driver to use cfg80211, it must register the hardware device with cfg80211. This happens through a number of hardware capability structs described below.

The fundamental structure for each device is the 'wiphy', of which each instance describes a physical wireless device connected to the system. Each such wiphy can have zero, one, or many virtual interfaces associated with it, which need to be identified as such by pointing the network interface's `ieee80211_ptr` pointer to a struct wireless_dev which further describes the wireless part of the interface, normally this struct is embedded in the network interface's private data area. Drivers can optionally allow creating or destroying virtual interfaces on the fly, but without at least one or the ability to create some the wireless device isn't useful.

Each wiphy structure contains device capability information, and also has a pointer to the various operations the driver offers. The definitions and structures here describe these capabilities in detail.

Name

enum ieee80211_band — supported frequency bands

Synopsis

```
enum ieee80211_band {
  IEEE80211_BAND_2GHZ,
  IEEE80211_BAND_5GHZ,
  IEEE80211_BAND_60GHZ,
  IEEE80211_NUM_BANDS
};
```

Constants

IEEE80211_BAND_2GHZ	2.4GHz ISM band
IEEE80211_BAND_5GHZ	around 5GHz band (4.9-5.7)
IEEE80211_BAND_60GHZ	around 60 GHz band (58.32 - 64.80 GHz)
IEEE80211_NUM_BANDS	number of defined bands

Device registration

The bands are assigned this way because the supported bitrates differ in these bands.

Name

enum ieee80211_channel_flags — channel flags

Synopsis

```
enum ieee80211_channel_flags {
  IEEE80211_CHAN_DISABLED,
  IEEE80211_CHAN_NO_IR,
  IEEE80211_CHAN_RADAR,
  IEEE80211_CHAN_NO_HT40PLUS,
  IEEE80211_CHAN_NO_HT40MINUS,
  IEEE80211_CHAN_NO_OFDM,
  IEEE80211_CHAN_NO_80MHZ,
  IEEE80211_CHAN_NO_160MHZ,
  IEEE80211_CHAN_INDOOR_ONLY,
  IEEE80211_CHAN_GO_CONCURRENT,
  IEEE80211_CHAN_NO_20MHZ,
  IEEE80211_CHAN_NO_10MHZ
};
```

Constants

IEEE80211_CHAN_DISABLED — This channel is disabled.

IEEE80211_CHAN_NO_IR — do not initiate radiation, this includes sending probe requests or beaconing.

IEEE80211_CHAN_RADAR — Radar detection is required on this channel.

IEEE80211_CHAN_NO_HT40PLUS — extension channel above this channel is not permitted.

IEEE80211_CHAN_NO_HT40MINUS — extension channel below this channel is not permitted.

IEEE80211_CHAN_NO_OFDM — OFDM is not allowed on this channel.

IEEE80211_CHAN_NO_80MHZ — If the driver supports 80 MHz on the band, this flag indicates that an 80 MHz channel cannot use this channel as the control or any of the secondary channels. This may be due to the driver or due to regulatory bandwidth restrictions.

IEEE80211_CHAN_NO_160MHZ — If the driver supports 160 MHz on the band, this flag indicates that an 160 MHz channel cannot use this channel as the control or any of the secondary channels. This may be due to the driver or due to regulatory bandwidth restrictions.

IEEE80211_CHAN_INDOOR_ONLY — see `NL80211_FREQUENCY_ATTR_INDOOR_ONLY`

IEEE80211_CHAN_GO_CONCURRENT — see `NL80211_FREQUENCY_ATTR_GO_CONCURRENT`

IEEE80211_CHAN_NO_20MHZ — 20 MHz bandwidth is not permitted on this channel.

IEEE80211_CHAN_NO_10MHZ — 10 MHz bandwidth is not permitted on this channel.

Description

Channel flags set by the regulatory control code.

Name

struct ieee80211_channel — channel definition

Synopsis

```
struct ieee80211_channel {
  enum ieee80211_band band;
  u16 center_freq;
  u16 hw_value;
  u32 flags;
  int max_antenna_gain;
  int max_power;
  int max_reg_power;
  bool beacon_found;
  u32 orig_flags;
  int orig_mag;
  int orig_mpwr;
  enum nl80211_dfs_state dfs_state;
  unsigned long dfs_state_entered;
  unsigned int dfs_cac_ms;
};
```

Members

band	band this channel belongs to.
center_freq	center frequency in MHz
hw_value	hardware-specific value for the channel
flags	channel flags from enum ieee80211_channel_flags.
max_antenna_gain	maximum antenna gain in dBi
max_power	maximum transmission power (in dBm)
max_reg_power	maximum regulatory transmission power (in dBm)
beacon_found	helper to regulatory code to indicate when a beacon has been found on this channel. Use `regulatory_hint_found_beacon` to enable this, this is useful only on 5 GHz band.
orig_flags	channel flags at registration time, used by regulatory code to support devices with additional restrictions
orig_mag	internal use
orig_mpwr	internal use
dfs_state	current state of this channel. Only relevant if radar is required on this channel.
dfs_state_entered	timestamp (jiffies) when the dfs state was entered.
dfs_cac_ms	DFS CAC time in milliseconds, this is valid for DFS channels.

Description

This structure describes a single channel for use with cfg80211.

Name

enum ieee80211_rate_flags — rate flags

Synopsis

```
enum ieee80211_rate_flags {
  IEEE80211_RATE_SHORT_PREAMBLE,
  IEEE80211_RATE_MANDATORY_A,
  IEEE80211_RATE_MANDATORY_B,
  IEEE80211_RATE_MANDATORY_G,
  IEEE80211_RATE_ERP_G,
  IEEE80211_RATE_SUPPORTS_5MHZ,
  IEEE80211_RATE_SUPPORTS_10MHZ
};
```

Constants

IEEE80211_RATE_SHORT_PREAMBLE	Hardware can send with short preamble on this bitrate; only relevant in 2.4GHz band and with CCK rates.
IEEE80211_RATE_MANDATORY_A	This bitrate is a mandatory rate when used with 802.11a (on the 5 GHz band); filled by the core code when registering the wiphy.
IEEE80211_RATE_MANDATORY_B	This bitrate is a mandatory rate when used with 802.11b (on the 2.4 GHz band); filled by the core code when registering the wiphy.
IEEE80211_RATE_MANDATORY_G	This bitrate is a mandatory rate when used with 802.11g (on the 2.4 GHz band); filled by the core code when registering the wiphy.
IEEE80211_RATE_ERP_G	This is an ERP rate in 802.11g mode.
IEEE80211_RATE_SUPPORTS_5MHZ	Rate can be used in 5 MHz mode
IEEE80211_RATE_SUPPORTS_10MHZ	Rate can be used in 10 MHz mode

Description

Hardware/specification flags for rates. These are structured in a way that allows using the same bitrate structure for different bands/PHY modes.

Name

struct ieee80211_rate — bitrate definition

Synopsis

```
struct ieee80211_rate {
  u32 flags;
  u16 bitrate;
  u16 hw_value;
  u16 hw_value_short;
};
```

Members

flags	rate-specific flags
bitrate	bitrate in units of 100 Kbps
hw_value	driver/hardware value for this rate
hw_value_short	driver/hardware value for this rate when short preamble is used

Description

This structure describes a bitrate that an 802.11 PHY can operate with. The two values *hw_value* and *hw_value_short* are only for driver use when pointers to this structure are passed around.

Name

struct ieee80211_sta_ht_cap — STA's HT capabilities

Synopsis

```
struct ieee80211_sta_ht_cap {
  u16 cap;
  bool ht_supported;
  u8 ampdu_factor;
  u8 ampdu_density;
  struct ieee80211_mcs_info mcs;
};
```

Members

cap	HT capabilities map as described in 802.11n spec
ht_supported	is HT supported by the STA
ampdu_factor	Maximum A-MPDU length factor
ampdu_density	Minimum A-MPDU spacing
mcs	Supported MCS rates

Description

This structure describes most essential parameters needed to describe 802.11n HT capabilities for an STA.

Name

struct ieee80211_supported_band — frequency band definition

Synopsis

```
struct ieee80211_supported_band {
  struct ieee80211_channel * channels;
  struct ieee80211_rate * bitrates;
  enum ieee80211_band band;
  int n_channels;
  int n_bitrates;
  struct ieee80211_sta_ht_cap ht_cap;
  struct ieee80211_sta_vht_cap vht_cap;
};
```

Members

channels	Array of channels the hardware can operate in in this band.
bitrates	Array of bitrates the hardware can operate with in this band. Must be sorted to give a valid “supported rates” IE, i.e. CCK rates first, then OFDM.
band	the band this structure represents
n_channels	Number of channels in *`channels`*
n_bitrates	Number of bitrates in *`bitrates`*
ht_cap	HT capabilities in this band
vht_cap	VHT capabilities in this band

Description

This structure describes a frequency band a wiphy is able to operate in.

Name

enum cfg80211_signal_type — signal type

Synopsis

```
enum cfg80211_signal_type {
  CFG80211_SIGNAL_TYPE_NONE,
  CFG80211_SIGNAL_TYPE_MBM,
  CFG80211_SIGNAL_TYPE_UNSPEC
};
```

Constants

CFG80211_SIGNAL_TYPE_NONE no signal strength information available

CFG80211_SIGNAL_TYPE_MBM signal strength in mBm (100*dBm)

CFG80211_SIGNAL_TYPE_UNSPEC signal strength, increasing from 0 through 100

Name

enum wiphy_params_flags — set_wiphy_params bitfield values

Synopsis

```
enum wiphy_params_flags {
  WIPHY_PARAM_RETRY_SHORT,
  WIPHY_PARAM_RETRY_LONG,
  WIPHY_PARAM_FRAG_THRESHOLD,
  WIPHY_PARAM_RTS_THRESHOLD,
  WIPHY_PARAM_COVERAGE_CLASS,
  WIPHY_PARAM_DYN_ACK
};
```

Constants

WIPHY_PARAM_RETRY_SHORT	wiphy->retry_short has changed
WIPHY_PARAM_RETRY_LONG	wiphy->retry_long has changed
WIPHY_PARAM_FRAG_THRESHOLD	wiphy->frag_threshold has changed
WIPHY_PARAM_RTS_THRESHOLD	wiphy->rts_threshold has changed
WIPHY_PARAM_COVERAGE_CLASS	coverage class changed
WIPHY_PARAM_DYN_ACK	dynack has been enabled

Name

enum wiphy_flags — wiphy capability flags

Synopsis

```
enum wiphy_flags {
  WIPHY_FLAG_NETNS_OK,
  WIPHY_FLAG_PS_ON_BY_DEFAULT,
  WIPHY_FLAG_4ADDR_AP,
  WIPHY_FLAG_4ADDR_STATION,
  WIPHY_FLAG_CONTROL_PORT_PROTOCOL,
  WIPHY_FLAG_IBSS_RSN,
  WIPHY_FLAG_MESH_AUTH,
  WIPHY_FLAG_SUPPORTS_SCHED_SCAN,
  WIPHY_FLAG_SUPPORTS_FW_ROAM,
  WIPHY_FLAG_AP_UAPSD,
  WIPHY_FLAG_SUPPORTS_TDLS,
  WIPHY_FLAG_TDLS_EXTERNAL_SETUP,
  WIPHY_FLAG_HAVE_AP_SME,
  WIPHY_FLAG_REPORTS_OBSS,
  WIPHY_FLAG_AP_PROBE_RESP_OFFLOAD,
  WIPHY_FLAG_OFFCHAN_TX,
  WIPHY_FLAG_HAS_REMAIN_ON_CHANNEL,
  WIPHY_FLAG_SUPPORTS_5_10_MHZ,
  WIPHY_FLAG_HAS_CHANNEL_SWITCH
};
```

Constants

WIPHY_FLAG_NETNS_OK	if not set, do not allow changing the netns of this wiphy at all
WIPHY_FLAG_PS_ON_BY_DEFAULT	if set to true, powersave will be enabled by default -- this flag will be set depending on the kernel's default on `wiphy_new`, but can be changed by the driver if it has a good reason to override the default
WIPHY_FLAG_4ADDR_AP	supports 4addr mode even on AP (with a single station on a VLAN interface)
WIPHY_FLAG_4ADDR_STATION	supports 4addr mode even as a station
WIPHY_FLAG_CONTROL_PORT_PROTOCOL	This device supports setting the control port protocol ethertype. The device also honours the control_port_no_encrypt flag.
WIPHY_FLAG_IBSS_RSN	The device supports IBSS RSN.
WIPHY_FLAG_MESH_AUTH	The device supports mesh authentication by routing auth frames to userspace. See *NL80211_MESH_SETUP_USERSPACE_AUTH*.
WIPHY_FLAG_SUPPORTS_SCHED_SCAN	The device supports scheduled scans.
WIPHY_FLAG_SUPPORTS_FW_ROAM	The device supports roaming feature in the firmware.
WIPHY_FLAG_AP_UAPSD	The device supports uapsd on AP.

WIPHY_FLAG_SUPPORTS_TDLS	The device supports TDLS (802.11z) operation.
WIPHY_FLAG_TDLS_EXTERNAL_SETUP	The device does not handle TDLS (802.11z) link setup/discovery operations internally. Setup, discovery and teardown packets should be sent through the *NL80211_CMD_TDLS_MGMT* command. When this flag is not set, *NL80211_CMD_TDLS_OPER* should be used for asking the driver/firmware to perform a TDLS operation.
WIPHY_FLAG_HAVE_AP_SME	device integrates AP SME
WIPHY_FLAG_REPORTS_OBSS	the device will report beacons from other BSSes when there are virtual interfaces in AP mode by calling `cfg80211_report_obss_beacon`.
WIPHY_FLAG_AP_PROBE_RESP_OFFLOAD	When operating as an AP, the device responds to probe-requests in hardware.
WIPHY_FLAG_OFFCHAN_TX	Device supports direct off-channel TX.
WIPHY_FLAG_HAS_REMAIN_ON_CHANNEL	Device supports remain-on-channel call.
WIPHY_FLAG_SUPPORTS_5_10_MHZ	Device supports 5 MHz and 10 MHz channels.
WIPHY_FLAG_HAS_CHANNEL_SWITCH	Device supports channel switch in beaconing mode (AP, IBSS, Mesh, ...).

Name

struct wiphy — wireless hardware description

Synopsis

```
struct wiphy {
  u8 perm_addr[ETH_ALEN];
  u8 addr_mask[ETH_ALEN];
  struct mac_address * addresses;
  const struct ieee80211_txrx_stypes * mgmt_stypes;
  const struct ieee80211_iface_combination * iface_combinations;
  int n_iface_combinations;
  u16 software_iftypes;
  u16 n_addresses;
  u16 interface_modes;
  u16 max_acl_mac_addrs;
  u32 flags;
  u32 regulatory_flags;
  u32 features;
  u8 ext_features[DIV_ROUND_UP(NUM_NL80211_EXT_FEATURES# 8)];
  u32 ap_sme_capa;
  enum cfg80211_signal_type signal_type;
  int bss_priv_size;
  u8 max_scan_ssids;
  u8 max_sched_scan_ssids;
  u8 max_match_sets;
  u16 max_scan_ie_len;
  u16 max_sched_scan_ie_len;
  int n_cipher_suites;
  const u32 * cipher_suites;
  u8 retry_short;
  u8 retry_long;
  u32 frag_threshold;
  u32 rts_threshold;
  u8 coverage_class;
  char fw_version[ETHTOOL_FWVERS_LEN];
  u32 hw_version;
#ifdef CONFIG_PM
  const struct wiphy_wowlan_support * wowlan;
  struct cfg80211_wowlan * wowlan_config;
#endif
  u16 max_remain_on_channel_duration;
  u8 max_num_pmkids;
  u32 available_antennas_tx;
  u32 available_antennas_rx;
  u32 probe_resp_offload;
  const u8 * extended_capabilities;
  const u8 * extended_capabilities_mask;
  u8 extended_capabilities_len;
  const void * privid;
  struct ieee80211_supported_band * bands[IEEE80211_NUM_BANDS];
  void (* reg_notifier) (struct wiphy *wiphy,struct regulatory_request *request);
```

```
  const struct ieee80211_regdomain __rcu * regd;
  struct device dev;
  bool registered;
  struct dentry * debugfsdir;
  const struct ieee80211_ht_cap * ht_capa_mod_mask;
  const struct ieee80211_vht_cap * vht_capa_mod_mask;
  possible_net_t _net;
#ifdef CONFIG_CFG80211_WEXT
  const struct iw_handler_def * wext;
#endif
  const struct wiphy_coalesce_support * coalesce;
  const struct wiphy_vendor_command * vendor_commands;
  const struct nl80211_vendor_cmd_info * vendor_events;
  int n_vendor_commands;
  int n_vendor_events;
  u16 max_ap_assoc_sta;
  u8 max_num_csa_counters;
  u8 max_adj_channel_rssi_comp;
  char priv[0];
};
```

Members

perm_addr[ETH_ALEN] — permanent MAC address of this device

addr_mask[ETH_ALEN] — If the device supports multiple MAC addresses by masking, set this to a mask with variable bits set to 1, e.g. if the last four bits are variable then set it to 00-00-00-00-00-0f. The actual variable bits shall be determined by the interfaces added, with interfaces not matching the mask being rejected to be brought up.

addresses — If the device has more than one address, set this pointer to a list of addresses (6 bytes each). The first one will be used by default for perm_addr. In this case, the mask should be set to all-zeroes. In this case it is assumed that the device can handle the same number of arbitrary MAC addresses.

mgmt_stypes — bitmasks of frame subtypes that can be subscribed to or transmitted through nl80211, points to an array indexed by interface type

iface_combinations — Valid interface combinations array, should not list single interface types.

n_iface_combinations — number of entries in *`iface_combinations`* array.

software_iftypes — bitmask of software interface types, these are not subject to any restrictions since they are purely managed in SW.

n_addresses — number of addresses in *`addresses`*.

interface_modes — bitmask of interfaces types valid for this wiphy, must be set by driver

max_acl_mac_addrs — Maximum number of MAC addresses that the device supports for ACL.

flags	wiphy flags, see enum wiphy_flags
regulatory_flags	wiphy regulatory flags, see enum ieee80211_regulatory_flags
features	features advertised to nl80211, see enum nl80211_feature_flags.
ext_features[DIV_ROUND_UP(NUM_NL80211_EXT_FEATURES, 8)]	extended features advertised to nl80211, see enum nl80211_ext_feature_index.
ap_sme_capa	AP SME capabilities, flags from enum nl80211_ap_sme_features.
signal_type	signal type reported in struct cfg80211_bss.
bss_priv_size	each BSS struct has private data allocated with it, this variable determines its size
max_scan_ssids	maximum number of SSIDs the device can scan for in any given scan
max_sched_scan_ssids	maximum number of SSIDs the device can scan for in any given scheduled scan
max_match_sets	maximum number of match sets the device can handle when performing a scheduled scan, 0 if filtering is not supported.
max_scan_ie_len	maximum length of user-controlled IEs device can add to probe request frames transmitted during a scan, must not include fixed IEs like supported rates
max_sched_scan_ie_len	same as max_scan_ie_len, but for scheduled scans
n_cipher_suites	number of supported cipher suites
cipher_suites	supported cipher suites
retry_short	Retry limit for short frames (dot11ShortRetryLimit)
retry_long	Retry limit for long frames (dot11LongRetryLimit)
frag_threshold	Fragmentation threshold (dot11FragmentationThreshold); -1 = fragmentation disabled, only odd values >= 256 used
rts_threshold	RTS threshold (dot11RTSThreshold); -1 = RTS/CTS disabled
coverage_class	current coverage class
fw_version[ETHTOOL_FWVERS_LEN]	firmware version for ethtool reporting
hw_version	hardware version for ethtool reporting
wowlan	WoWLAN support information
wowlan_config	current WoWLAN configuration; this should usually not be used since access to it is necessarily racy, use the parameter passed to the `suspend` operation instead.
max_remain_on_channel_duration	Maximum time a remain-on-channel operation may request, if implemented.

max_num_pmkids	maximum number of PMKIDs supported by device
available_antennas_tx	bitmap of antennas which are available to be configured as TX antennas. Antenna configuration commands will be rejected unless this or *`available_antennas_rx`* is set.
available_antennas_rx	bitmap of antennas which are available to be configured as RX antennas. Antenna configuration commands will be rejected unless this or *`available_antennas_tx`* is set.
probe_resp_offload	Bitmap of supported protocols for probe response offloading. See enum nl80211_probe_resp_offload_support_attr. Only valid when the wiphy flag *`WIPHY_FLAG_AP_PROBE_RESP_OFFLOAD`* is set.
extended_capabilities	extended capabilities supported by the driver, additional capabilities might be supported by userspace; these are the 802.11 extended capabilities (“Extended Capabilities element”) and are in the same format as in the information element. See 802.11-2012 8.4.2.29 for the defined fields.
extended_capabilities_mask	mask of the valid values
extended_capabilities_len	length of the extended capabilities
privid	a pointer that drivers can use to identify if an arbitrary wiphy is theirs, e.g. in global notifiers
bands[IEEE80211_NUM_BANDS]	information about bands/channels supported by this device
reg_notifier	the driver's regulatory notification callback, note that if your driver uses `wiphy_apply_custom_regulatory` the reg_notifier's request can be passed as NULL
regd	the driver's regulatory domain, if one was requested via the `regulatory_hint` API. This can be used by the driver on the `reg_notifier` if it chooses to ignore future regulatory domain changes caused by other drivers.
dev	(virtual) struct device for this wiphy
registered	helps synchronize suspend/resume with wiphy unregister
debugfsdir	debugfs directory used for this wiphy, will be renamed automatically on wiphy renames
ht_capa_mod_mask	Specify what ht_cap values can be over-ridden. If null, then none can be over-ridden.
vht_capa_mod_mask	Specify what VHT capabilities can be over-ridden. If null, then none can be over-ridden.
_net	the network namespace this wiphy currently lives in
wext	wireless extension handlers
coalesce	packet coalescing support information

vendor_commands	array of vendor commands supported by the hardware
vendor_events	array of vendor events supported by the hardware
n_vendor_commands	number of vendor commands
n_vendor_events	number of vendor events
max_ap_assoc_sta	maximum number of associated stations supported in AP mode (including P2P GO) or 0 to indicate no such limit is advertised. The driver is allowed to advertise a theoretical limit that it can reach in some cases, but may not always reach.
max_num_csa_counters	Number of supported csa_counters in beacons and probe responses. This value should be set if the driver wishes to limit the number of csa counters. Default (0) means infinite.
max_adj_channel_rssi_comp	max offset of between the channel on which the frame was sent and the channel on which the frame was heard for which the reported rssi is still valid. If a driver is able to compensate the low rssi when a frame is heard on different channel, then it should set this variable to the maximal offset for which it can compensate. This value should be set in MHz.
priv[0]	driver private data (sized according to `wiphy_new` parameter)

Name

struct wireless_dev — wireless device state

Synopsis

```
struct wireless_dev {
  struct wiphy * wiphy;
  enum nl80211_iftype iftype;
  struct list_head list;
  struct net_device * netdev;
  u32 identifier;
  struct list_head mgmt_registrations;
  spinlock_t mgmt_registrations_lock;
  struct mutex mtx;
  bool use_4addr;
  bool p2p_started;
  u8 address[ETH_ALEN];
  u8 ssid[IEEE80211_MAX_SSID_LEN];
  u8 ssid_len;
  u8 mesh_id_len;
  u8 mesh_id_up_len;
  struct cfg80211_conn * conn;
  struct cfg80211_cached_keys * connect_keys;
  struct list_head event_list;
  spinlock_t event_lock;
  struct cfg80211_internal_bss * current_bss;
  struct cfg80211_chan_def preset_chandef;
  struct cfg80211_chan_def chandef;
  bool ibss_fixed;
  bool ibss_dfs_possible;
  bool ps;
  int ps_timeout;
  int beacon_interval;
  u32 ap_unexpected_nlportid;
  bool cac_started;
  unsigned long cac_start_time;
  unsigned int cac_time_ms;
  u32 owner_nlportid;
#ifdef CONFIG_CFG80211_WEXT
  struct wext;
#endif
};
```

Members

wiphy	pointer to hardware description
iftype	interface type
list	(private) Used to collect the interfaces
netdev	(private) Used to reference back to the netdev, may be NULL

identifier	(private) Identifier used in nl80211 to identify this wireless device if it has no netdev
mgmt_registrations	list of registrations for management frames
mgmt_registrations_lock	lock for the list
mtx	mutex used to lock data in this struct, may be used by drivers and some API functions require it held
use_4addr	indicates 4addr mode is used on this interface, must be set by driver (if supported) on add_interface BEFORE registering the netdev and may otherwise be used by driver read-only, will be update by cfg80211 on change_interface
p2p_started	true if this is a P2P Device that has been started
address[ETH_ALEN]	The address for this device, valid only if *netdev* is NULL
ssid[IEEE80211_MAX_SSID_LEN]	(private) Used by the internal configuration code
ssid_len	(private) Used by the internal configuration code
mesh_id_len	(private) Used by the internal configuration code
mesh_id_up_len	(private) Used by the internal configuration code
conn	(private) cfg80211 software SME connection state machine data
connect_keys	(private) keys to set after connection is established
event_list	(private) list for internal event processing
event_lock	(private) lock for event list
current_bss	(private) Used by the internal configuration code
preset_chandef	(private) Used by the internal configuration code to track the channel to be used for AP later
chandef	(private) Used by the internal configuration code to track the user-set channel definition.
ibss_fixed	(private) IBSS is using fixed BSSID
ibss_dfs_possible	(private) IBSS may change to a DFS channel
ps	powersave mode is enabled
ps_timeout	dynamic powersave timeout
beacon_interval	beacon interval used on this device for transmitting beacons, 0 when not valid
ap_unexpected_nlportid	(private) netlink port ID of application registered for unexpected class 3 frames (AP mode)
cac_started	true if DFS channel availability check has been started

cac_start_time	timestamp (jiffies) when the dfs state was entered.
cac_time_ms	CAC time in ms
owner_nlportid	(private) owner socket port ID
wext	(private) Used by the internal wireless extensions compat code

Description

For netdevs, this structure must be allocated by the driver that uses the ieee80211_ptr field in struct net_device (this is intentional so it can be allocated along with the netdev.) It need not be registered then as netdev registration will be intercepted by cfg80211 to see the new wireless device.

For non-netdev uses, it must also be allocated by the driver in response to the cfg80211 callbacks that require it, as there's no netdev registration in that case it may not be allocated outside of callback operations that return it.

Name

wiphy_new — create a new wiphy for use with cfg80211

Synopsis

```
struct wiphy * wiphy_new (const struct cfg80211_ops * ops, int sizeof_priv);
```

Arguments

ops	The configuration operations for this device
sizeof_priv	The size of the private area to allocate

Description

Create a new wiphy and associate the given operations with it. *sizeof_priv* bytes are allocated for private use.

Return

A pointer to the new wiphy. This pointer must be assigned to each netdev's ieee80211_ptr for proper operation.

Name

wiphy_register — register a wiphy with cfg80211

Synopsis

```
int wiphy_register (struct wiphy * wiphy);
```

Arguments

wiphy The wiphy to register.

Return

A non-negative wiphy index or a negative error code.

Name

wiphy_unregister — deregister a wiphy from cfg80211

Synopsis

```
void wiphy_unregister (struct wiphy * wiphy);
```

Arguments

wiphy The wiphy to unregister.

Description

After this call, no more requests can be made with this priv pointer, but the call may sleep to wait for an outstanding request that is being handled.

Name

wiphy_free — free wiphy

Synopsis

```
void wiphy_free (struct wiphy * wiphy);
```

Arguments

wiphy The wiphy to free

Name

wiphy_name — get wiphy name

Synopsis

```
const char * wiphy_name (const struct wiphy * wiphy);
```

Arguments

wiphy The wiphy whose name to return

Return

The name of *wiphy*.

Name

wiphy_dev — get wiphy dev pointer

Synopsis

```
struct device * wiphy_dev (struct wiphy * wiphy);
```

Arguments

wiphy The wiphy whose device struct to look up

Return

The dev of *wiphy*.

Name

wiphy_priv — return priv from wiphy

Synopsis

```
void * wiphy_priv (struct wiphy * wiphy);
```

Arguments

wiphy the wiphy whose priv pointer to return

Return

The priv of *wiphy*.

Name

priv_to_wiphy — return the wiphy containing the priv

Synopsis

```
struct wiphy * priv_to_wiphy (void * priv);
```

Arguments

priv a pointer previously returned by wiphy_priv

Return

The wiphy of *priv*.

Name

set_wiphy_dev — set device pointer for wiphy

Synopsis

```
void set_wiphy_dev (struct wiphy * wiphy, struct device * dev);
```

Arguments

wiphy The wiphy whose device to bind

dev The device to parent it to

Name

wdev_priv — return wiphy priv from wireless_dev

Synopsis

```
void * wdev_priv (struct wireless_dev * wdev);
```

Arguments

wdev The wireless device whose wiphy's priv pointer to return

Return

The wiphy priv of *wdev*.

Name

struct ieee80211_iface_limit — limit on certain interface types

Synopsis

```
struct ieee80211_iface_limit {
  u16 max;
  u16 types;
};
```

Members

max maximum number of interfaces of these types

types interface types (bits)

Name

struct ieee80211_iface_combination — possible interface combination

Synopsis

```
struct ieee80211_iface_combination {
  const struct ieee80211_iface_limit * limits;
  u32 num_different_channels;
  u16 max_interfaces;
  u8 n_limits;
  bool beacon_int_infra_match;
  u8 radar_detect_widths;
  u8 radar_detect_regions;
};
```

Members

limits	limits for the given interface types
num_different_channels	can use up to this many different channels
max_interfaces	maximum number of interfaces in total allowed in this group
n_limits	number of limitations
beacon_int_infra_match	In this combination, the beacon intervals between infrastructure and AP types must match. This is required only in special cases.
radar_detect_widths	bitmap of channel widths supported for radar detection
radar_detect_regions	bitmap of regions supported for radar detection

Description

With this structure the driver can describe which interface combinations it supports concurrently.

Examples

```
1. Allow #STA <= 1, #AP <= 1, matching BI, channels = 1, 2 total:

 struct ieee80211_iface_limit limits1[] = {
 { .max = 1, .types = BIT(NL80211_IFTYPE_STATION), },
 { .max = 1, .types = BIT(NL80211_IFTYPE_AP}, },
 };
 struct ieee80211_iface_combination combination1 = {
 .limits = limits1,
 .n_limits = ARRAY_SIZE(limits1),
 .max_interfaces = 2,
 .beacon_int_infra_match = true,
 };
```

2. Allow #{AP, P2P-GO} <= 8, channels = 1, 8 total:

```
 struct ieee80211_iface_limit limits2[] = {
 { .max = 8, .types = BIT(NL80211_IFTYPE_AP) |
        BIT(NL80211_IFTYPE_P2P_GO), },
 };
 struct ieee80211_iface_combination combination2 = {
 .limits = limits2,
 .n_limits = ARRAY_SIZE(limits2),
 .max_interfaces = 8,
 .num_different_channels = 1,
 };
```

3. Allow #STA <= 1, #{P2P-client,P2P-GO} <= 3 on two channels, 4 total.

This allows for an infrastructure connection and three P2P connections.

```
 struct ieee80211_iface_limit limits3[] = {
 { .max = 1, .types = BIT(NL80211_IFTYPE_STATION), },
 { .max = 3, .types = BIT(NL80211_IFTYPE_P2P_GO) |
        BIT(NL80211_IFTYPE_P2P_CLIENT), },
 };
 struct ieee80211_iface_combination combination3 = {
 .limits = limits3,
 .n_limits = ARRAY_SIZE(limits3),
 .max_interfaces = 4,
 .num_different_channels = 2,
 };
```

Name

cfg80211_check_combinations — check interface combinations

Synopsis

```
int cfg80211_check_combinations (struct wiphy * wiphy, const
int num_different_channels, const u8 radar_detect, const int
iftype_num[NUM_NL80211_IFTYPES]);
```

Arguments

wiphy	the wiphy
num_different_channels	the number of different channels we want to use for verification
radar_detect	a bitmap where each bit corresponds to a channel width where radar detection is needed, as in the definition of struct ieee80211_iface_combination.*radar_detect_widths*
iftype_num[NUM_NL80211_IFTYPES]	array with the numbers of interfaces of each interface type. The index is the interface type as specified in enum nl80211_iftype.

Description

This function can be called by the driver to check whether a combination of interfaces and their types are allowed according to the interface combinations.

Chapter 2. Actions and configuration

Each wireless device and each virtual interface offer a set of configuration operations and other actions that are invoked by userspace. Each of these actions is described in the operations structure, and the parameters these operations use are described separately.

Additionally, some operations are asynchronous and expect to get status information via some functions that drivers need to call.

Scanning and BSS list handling with its associated functionality is described in a separate chapter.

Name

struct cfg80211_ops — backend description for wireless configuration

Synopsis

```
struct cfg80211_ops {
  int (* suspend) (struct wiphy *wiphy, struct cfg80211_wowlan *wow);
  int (* resume) (struct wiphy *wiphy);
  void (* set_wakeup) (struct wiphy *wiphy, bool enabled);
  struct wireless_dev * (* add_virtual_intf) (struct wiphy *wiphy,const char *name
  int (* del_virtual_intf) (struct wiphy *wiphy,struct wireless_dev *wdev);
  int (* change_virtual_intf) (struct wiphy *wiphy,struct net_device *dev,enum nl8
  int (* add_key) (struct wiphy *wiphy, struct net_device *netdev,u8 key_index, bo
  int (* get_key) (struct wiphy *wiphy, struct net_device *netdev,u8 key_index, bo
  int (* del_key) (struct wiphy *wiphy, struct net_device *netdev,u8 key_index, bo
  int (* set_default_key) (struct wiphy *wiphy,struct net_device *netdev,u8 key_in
  int (* set_default_mgmt_key) (struct wiphy *wiphy,struct net_device *netdev,u8 k
  int (* start_ap) (struct wiphy *wiphy, struct net_device *dev,struct cfg80211_ap
  int (* change_beacon) (struct wiphy *wiphy, struct net_device *dev,struct cfg802
  int (* stop_ap) (struct wiphy *wiphy, struct net_device *dev);
  int (* add_station) (struct wiphy *wiphy, struct net_device *dev,const u8 *mac,s
  int (* del_station) (struct wiphy *wiphy, struct net_device *dev,struct station_
  int (* change_station) (struct wiphy *wiphy, struct net_device *dev,const u8 *ma
  int (* get_station) (struct wiphy *wiphy, struct net_device *dev,const u8 *mac,
  int (* dump_station) (struct wiphy *wiphy, struct net_device *dev,int idx, u8 *m
  int (* add_mpath) (struct wiphy *wiphy, struct net_device *dev,const u8 *dst, co
  int (* del_mpath) (struct wiphy *wiphy, struct net_device *dev,const u8 *dst);
  int (* change_mpath) (struct wiphy *wiphy, struct net_device *dev,const u8 *dst,
  int (* get_mpath) (struct wiphy *wiphy, struct net_device *dev,u8 *dst, u8 *next
  int (* dump_mpath) (struct wiphy *wiphy, struct net_device *dev,int idx, u8 *dst
  int (* get_mpp) (struct wiphy *wiphy, struct net_device *dev,u8 *dst, u8 *mpp, s
  int (* dump_mpp) (struct wiphy *wiphy, struct net_device *dev,int idx, u8 *dst,
  int (* get_mesh_config) (struct wiphy *wiphy,struct net_device *dev,struct mesh_
  int (* update_mesh_config) (struct wiphy *wiphy,struct net_device *dev, u32 mask
  int (* join_mesh) (struct wiphy *wiphy, struct net_device *dev,const struct mesh
  int (* leave_mesh) (struct wiphy *wiphy, struct net_device *dev);
  int (* join_ocb) (struct wiphy *wiphy, struct net_device *dev,struct ocb_setup *
  int (* leave_ocb) (struct wiphy *wiphy, struct net_device *dev);
  int (* change_bss) (struct wiphy *wiphy, struct net_device *dev,struct bss_param
  int (* set_txq_params) (struct wiphy *wiphy, struct net_device *dev,struct ieee8
  int (* libertas_set_mesh_channel) (struct wiphy *wiphy,struct net_device *dev,st
  int (* set_monitor_channel) (struct wiphy *wiphy,struct cfg80211_chan_def *chand
  int (* scan) (struct wiphy *wiphy,struct cfg80211_scan_request *request);
  int (* auth) (struct wiphy *wiphy, struct net_device *dev,struct cfg80211_auth_r
  int (* assoc) (struct wiphy *wiphy, struct net_device *dev,struct cfg80211_assoc
  int (* deauth) (struct wiphy *wiphy, struct net_device *dev,struct cfg80211_deau
  int (* disassoc) (struct wiphy *wiphy, struct net_device *dev,struct cfg80211_di
  int (* connect) (struct wiphy *wiphy, struct net_device *dev,struct cfg80211_con
  int (* disconnect) (struct wiphy *wiphy, struct net_device *dev,u16 reason_code)
  int (* join_ibss) (struct wiphy *wiphy, struct net_device *dev,struct cfg80211_i
  int (* leave_ibss) (struct wiphy *wiphy, struct net_device *dev);
  int (* set_mcast_rate) (struct wiphy *wiphy, struct net_device *dev,int rate[IEE
```

```
  int (* set_wiphy_params) (struct wiphy *wiphy, u32 changed);
  int (* set_tx_power) (struct wiphy *wiphy, struct wireless_dev *wdev,enum nl8021
  int (* get_tx_power) (struct wiphy *wiphy, struct wireless_dev *wdev,int *dbm);
  int (* set_wds_peer) (struct wiphy *wiphy, struct net_device *dev,const u8 *addr
  void (* rfkill_poll) (struct wiphy *wiphy);
#ifdef CONFIG_NL80211_TESTMODE
  int (* testmode_cmd) (struct wiphy *wiphy, struct wireless_dev *wdev,void *data,
  int (* testmode_dump) (struct wiphy *wiphy, struct sk_buff *skb,struct netlink_c
#endif
  int (* set_bitrate_mask) (struct wiphy *wiphy,struct net_device *dev,const u8 *p
  int (* dump_survey) (struct wiphy *wiphy, struct net_device *netdev,int idx, str
  int (* set_pmksa) (struct wiphy *wiphy, struct net_device *netdev,struct cfg8021
  int (* del_pmksa) (struct wiphy *wiphy, struct net_device *netdev,struct cfg8021
  int (* flush_pmksa) (struct wiphy *wiphy, struct net_device *netdev);
  int (* remain_on_channel) (struct wiphy *wiphy,struct wireless_dev *wdev,struct
  int (* cancel_remain_on_channel) (struct wiphy *wiphy,struct wireless_dev *wdev,
  int (* mgmt_tx) (struct wiphy *wiphy, struct wireless_dev *wdev,struct cfg80211_
  int (* mgmt_tx_cancel_wait) (struct wiphy *wiphy,struct wireless_dev *wdev,u64 c
  int (* set_power_mgmt) (struct wiphy *wiphy, struct net_device *dev,bool enabled
  int (* set_cqm_rssi_config) (struct wiphy *wiphy,struct net_device *dev,s32 rssi
  int (* set_cqm_txe_config) (struct wiphy *wiphy,struct net_device *dev,u32 rate,
  void (* mgmt_frame_register) (struct wiphy *wiphy,struct wireless_dev *wdev,u16
  int (* set_antenna) (struct wiphy *wiphy, u32 tx_ant, u32 rx_ant);
  int (* get_antenna) (struct wiphy *wiphy, u32 *tx_ant, u32 *rx_ant);
  int (* sched_scan_start) (struct wiphy *wiphy,struct net_device *dev,struct cfg8
  int (* sched_scan_stop) (struct wiphy *wiphy, struct net_device *dev);
  int (* set_rekey_data) (struct wiphy *wiphy, struct net_device *dev,struct cfg80
  int (* tdls_mgmt) (struct wiphy *wiphy, struct net_device *dev,const u8 *peer, u
  int (* tdls_oper) (struct wiphy *wiphy, struct net_device *dev,const u8 *peer, e
  int (* probe_client) (struct wiphy *wiphy, struct net_device *dev,const u8 *peer
  int (* set_noack_map) (struct wiphy *wiphy,struct net_device *dev,u16 noack_map)
  int (* get_channel) (struct wiphy *wiphy,struct wireless_dev *wdev,struct cfg802
  int (* start_p2p_device) (struct wiphy *wiphy,struct wireless_dev *wdev);
  void (* stop_p2p_device) (struct wiphy *wiphy,struct wireless_dev *wdev);
  int (* set_mac_acl) (struct wiphy *wiphy, struct net_device *dev,const struct cf
  int (* start_radar_detection) (struct wiphy *wiphy,struct net_device *dev,struct
  int (* update_ft_ies) (struct wiphy *wiphy, struct net_device *dev,struct cfg802
  int (* crit_proto_start) (struct wiphy *wiphy,struct wireless_dev *wdev,enum nl8
  void (* crit_proto_stop) (struct wiphy *wiphy,struct wireless_dev *wdev);
  int (* set_coalesce) (struct wiphy *wiphy,struct cfg80211_coalesce *coalesce);
  int (* channel_switch) (struct wiphy *wiphy,struct net_device *dev,struct cfg802
  int (* set_qos_map) (struct wiphy *wiphy,struct net_device *dev,struct cfg80211_
  int (* set_ap_chanwidth) (struct wiphy *wiphy, struct net_device *dev,struct cfg
  int (* add_tx_ts) (struct wiphy *wiphy, struct net_device *dev,u8 tsid, const u8
  int (* del_tx_ts) (struct wiphy *wiphy, struct net_device *dev,u8 tsid, const u8
  int (* tdls_channel_switch) (struct wiphy *wiphy,struct net_device *dev,const u8
  void (* tdls_cancel_channel_switch) (struct wiphy *wiphy,struct net_device *dev,
};
```

Members

suspend	wiphy device needs to be suspended. The variable *wow* will be NULL or contain the enabled Wake-on-Wireless triggers that are configured for the device.
resume	wiphy device needs to be resumed
set_wakeup	Called when WoWLAN is enabled/disabled, use this callback to call device_set_wakeup_enable to enable/disable wakeup from the device.
add_virtual_intf	create a new virtual interface with the given name, must set the struct wireless_dev's iftype. Beware: You must create the new netdev in the wiphy's network namespace! Returns the struct wireless_dev, or an ERR_PTR. For P2P device wdevs, the driver must also set the address member in the wdev.
del_virtual_intf	remove the virtual interface
change_virtual_intf	change type/configuration of virtual interface, keep the struct wireless_dev's iftype updated.
add_key	add a key with the given parameters. *mac_addr* will be NULL when adding a group key.
get_key	get information about the key with the given parameters. *mac_addr* will be NULL when requesting information for a group key. All pointers given to the *callback* function need not be valid after it returns. This function should return an error if it is not possible to retrieve the key, -ENOENT if it doesn't exist.
del_key	remove a key given the *mac_addr* (NULL for a group key) and *key_index*, return -ENOENT if the key doesn't exist.
set_default_key	set the default key on an interface
set_default_mgmt_key	set the default management frame key on an interface
start_ap	Start acting in AP mode defined by the parameters.
change_beacon	Change the beacon parameters for an access point mode interface. This should reject the call when AP mode wasn't started.
stop_ap	Stop being an AP, including stopping beaconing.
add_station	Add a new station.
del_station	Remove a station
change_station	Modify a given station. Note that flags changes are not much validated in cfg80211, in particular the auth/assoc/authorized flags might come to the driver in invalid combinations -- make sure to check them, also against the existing state! Drivers must call cfg80211_check_station_change to validate the information.

get_station	get station information for the station identified by *mac*
dump_station	dump station callback -- resume dump at index *idx*
add_mpath	add a fixed mesh path
del_mpath	delete a given mesh path
change_mpath	change a given mesh path
get_mpath	get a mesh path for the given parameters
dump_mpath	dump mesh path callback -- resume dump at index *idx*
get_mpp	get a mesh proxy path for the given parameters
dump_mpp	dump mesh proxy path callback -- resume dump at index *idx*
get_mesh_config	Get the current mesh configuration
update_mesh_config	Update mesh parameters on a running mesh. The mask is a bitfield which tells us which parameters to set, and which to leave alone.
join_mesh	join the mesh network with the specified parameters (invoked with the wireless_dev mutex held)
leave_mesh	leave the current mesh network (invoked with the wireless_dev mutex held)
join_ocb	join the OCB network with the specified parameters (invoked with the wireless_dev mutex held)
leave_ocb	leave the current OCB network (invoked with the wireless_dev mutex held)
change_bss	Modify parameters for a given BSS.
set_txq_params	Set TX queue parameters
libertas_set_mesh_channel	Only for backward compatibility for libertas, as it doesn't implement join_mesh and needs to set the channel to join the mesh instead.
set_monitor_channel	Set the monitor mode channel for the device. If other interfaces are active this callback should reject the configuration. If no interfaces are active or the device is down, the channel should be stored for when a monitor interface becomes active.
scan	Request to do a scan. If returning zero, the scan request is given the driver, and will be valid until passed to `cfg80211_scan_done`. For scan results, call `cfg80211_inform_bss`; you can call this outside the scan/scan_done bracket too.
auth	Request to authenticate with the specified peer (invoked with the wireless_dev mutex held)
assoc	Request to (re)associate with the specified peer (invoked with the wireless_dev mutex held)

deauth	Request to deauthenticate from the specified peer (invoked with the wireless_dev mutex held)
disassoc	Request to disassociate from the specified peer (invoked with the wireless_dev mutex held)
connect	Connect to the ESS with the specified parameters. When connected, call `cfg80211_connect_result` with status code `WLAN_STATUS_SUCCESS`. If the connection fails for some reason, call `cfg80211_connect_result` with the status from the AP. (invoked with the wireless_dev mutex held)
disconnect	Disconnect from the BSS/ESS. (invoked with the wireless_dev mutex held)
join_ibss	Join the specified IBSS (or create if necessary). Once done, call `cfg80211_ibss_joined`, also call that function when changing BSSID due to a merge. (invoked with the wireless_dev mutex held)
leave_ibss	Leave the IBSS. (invoked with the wireless_dev mutex held)
set_mcast_rate	Set the specified multicast rate (only if vif is in ADHOC or MESH mode)
set_wiphy_params	Notify that wiphy parameters have changed; *`changed`* bitfield (see enum wiphy_params_flags) describes which values have changed. The actual parameter values are available in struct wiphy. If returning an error, no value should be changed.
set_tx_power	set the transmit power according to the parameters, the power passed is in mBm, to get dBm use `MBM_TO_DBM`. The wdev may be `NULL` if power was set for the wiphy, and will always be `NULL` unless the driver supports per-vif TX power (as advertised by the nl80211 feature flag.)
get_tx_power	store the current TX power into the dbm variable; return 0 if successful
set_wds_peer	set the WDS peer for a WDS interface
rfkill_poll	polls the hw rfkill line, use cfg80211 reporting functions to adjust rfkill hw state
testmode_cmd	run a test mode command; *`wdev`* may be `NULL`
testmode_dump	Implement a test mode dump. The cb->args[2] and up may be used by the function, but 0 and 1 must not be touched. Additionally, return error codes other than -ENOBUFS and -ENOENT will terminate the dump and return to userspace with an error, so be careful. If any data was passed in from userspace then the data/len arguments will be present and point to the data contained in `NL80211_ATTR_TESTDATA`.
set_bitrate_mask	set the bitrate mask configuration
dump_survey	get site survey information.

set_pmksa	Cache a PMKID for a BSSID. This is mostly useful for fullmac devices running firmwares capable of generating the (re) association RSN IE. It allows for faster roaming between WPA2 BSSIDs.
del_pmksa	Delete a cached PMKID.
flush_pmksa	Flush all cached PMKIDs.
remain_on_channel	Request the driver to remain awake on the specified channel for the specified duration to complete an off-channel operation (e.g., public action frame exchange). When the driver is ready on the requested channel, it must indicate this with an event notification by calling `cfg80211_ready_on_channel`.
cancel_remain_on_channel	Cancel an on-going remain-on-channel operation. This allows the operation to be terminated prior to timeout based on the duration value.
mgmt_tx	Transmit a management frame.
mgmt_tx_cancel_wait	Cancel the wait time from transmitting a management frame on another channel
set_power_mgmt	Configure WLAN power management. A timeout value of -1 allows the driver to adjust the dynamic ps timeout value.
set_cqm_rssi_config	Configure connection quality monitor RSSI threshold.
set_cqm_txe_config	Configure connection quality monitor TX error thresholds.
mgmt_frame_register	Notify driver that a management frame type was registered. Note that this callback may not sleep, and cannot run concurrently with itself.
set_antenna	Set antenna configuration (tx_ant, rx_ant) on the device. Parameters are bitmaps of allowed antennas to use for TX/ RX. Drivers may reject TX/RX mask combinations they cannot support by returning -EINVAL (also see nl80211.h *`NL80211_ATTR_WIPHY_ANTENNA_TX`*).
get_antenna	Get current antenna configuration from device (tx_ant, rx_ant).
sched_scan_start	Tell the driver to start a scheduled scan.
sched_scan_stop	Tell the driver to stop an ongoing scheduled scan. This call must stop the scheduled scan and be ready for starting a new one before it returns, i.e. *`sched_scan_start`* may be called immediately after that again and should not fail in that case. The driver should not call `cfg80211_sched_scan_stopped` for a requested stop (when this method returns 0.)
set_rekey_data	give the data necessary for GTK rekeying to the driver
tdls_mgmt	Transmit a TDLS management frame.
tdls_oper	Perform a high-level TDLS operation (e.g. TDLS link setup).

probe_client	probe an associated client, must return a cookie that it later passes to `cfg80211_probe_status`.
set_noack_map	Set the NoAck Map for the TIDs.
get_channel	Get the current operating channel for the virtual interface. For monitor interfaces, it should return `NULL` unless there's a single current monitoring channel.
start_p2p_device	Start the given P2P device.
stop_p2p_device	Stop the given P2P device.
set_mac_acl	Sets MAC address control list in AP and P2P GO mode. Parameters include ACL policy, an array of MAC address of stations and the number of MAC addresses. If there is already a list in driver this new list replaces the existing one. Driver has to clear its ACL when number of MAC addresses entries is passed as 0. Drivers which advertise the support for MAC based ACL have to implement this callback.
start_radar_detection	Start radar detection in the driver.
update_ft_ies	Provide updated Fast BSS Transition information to the driver. If the SME is in the driver/firmware, this information can be used in building Authentication and Reassociation Request frames.
crit_proto_start	Indicates a critical protocol needs more link reliability for a given duration (milliseconds). The protocol is provided so the driver can take the most appropriate actions.
crit_proto_stop	Indicates critical protocol no longer needs increased link reliability. This operation can not fail.
set_coalesce	Set coalesce parameters.
channel_switch	initiate channel-switch procedure (with CSA). Driver is responsible for veryfing if the switch is possible. Since this is inherently tricky driver may decide to disconnect an interface later with `cfg80211_stop_iface`. This doesn't mean driver can accept everything. It should do it's best to verify requests and reject them as soon as possible.
set_qos_map	Set QoS mapping information to the driver
set_ap_chanwidth	Set the AP (including P2P GO) mode channel width for the given interface This is used e.g. for dynamic HT 20/40 MHz channel width changes during the lifetime of the BSS.
add_tx_ts	validate (if admitted_time is 0) or add a TX TS to the device with the given parameters; action frame exchange has been handled by userspace so this just has to modify the TX path to take the TS into account. If the admitted time is 0 just validate the parameters to make sure the session can be created at all; it is valid to just always return success for that but that may result in inefficient behaviour (handshake with the peer followed by immediate teardown when the addition is later rejected)

del_tx_ts	remove an existing TX TS
tdls_channel_switch	Start channel-switching with a TDLS peer. The driver is responsible for continually initiating channel-switching operations and returning to the base channel for communication with the AP.
tdls_cancel_channel_switch	Stop channel-switching with a TDLS peer. Both peers must be on the base channel when the call completes.

Description

This struct is registered by fullmac card drivers and/or wireless stacks in order to handle configuration requests on their interfaces.

All callbacks except where otherwise noted should return 0 on success or a negative error code.

All operations are currently invoked under rtnl for consistency with the wireless extensions but this is subject to reevaluation as soon as this code is used more widely and we have a first user without wext.

Name

struct vif_params — describes virtual interface parameters

Synopsis

```
struct vif_params {
  int use_4addr;
  u8 macaddr[ETH_ALEN];
};
```

Members

use_4addr	use 4-address frames
macaddr[ETH_ALEN]	address to use for this virtual interface. If this parameter is set to zero address the driver may determine the address as needed. This feature is only fully supported by drivers that enable the `NL80211_FEATURE_MAC_ON_CREATE` flag. Others may support creating * only p2p devices with specified MAC.

Name

struct key_params — key information

Synopsis

```
struct key_params {
  const u8 * key;
  const u8 * seq;
  int key_len;
  int seq_len;
  u32 cipher;
};
```

Members

key — key material

seq — sequence counter (IV/PN) for TKIP and CCMP keys, only used with the `get_key` callback, must be in little endian, length given by *`seq_len`*.

key_len — length of key material

seq_len — length of *`seq`*.

cipher — cipher suite selector

Description

Information about a key

Name

enum survey_info_flags — survey information flags

Synopsis

```
enum survey_info_flags {
  SURVEY_INFO_NOISE_DBM,
  SURVEY_INFO_IN_USE,
  SURVEY_INFO_TIME,
  SURVEY_INFO_TIME_BUSY,
  SURVEY_INFO_TIME_EXT_BUSY,
  SURVEY_INFO_TIME_RX,
  SURVEY_INFO_TIME_TX,
  SURVEY_INFO_TIME_SCAN
};
```

Constants

SURVEY_INFO_NOISE_DBM	noise (in dBm) was filled in
SURVEY_INFO_IN_USE	channel is currently being used
SURVEY_INFO_TIME	active time (in ms) was filled in
SURVEY_INFO_TIME_BUSY	busy time was filled in
SURVEY_INFO_TIME_EXT_BUSY	extension channel busy time was filled in
SURVEY_INFO_TIME_RX	receive time was filled in
SURVEY_INFO_TIME_TX	transmit time was filled in
SURVEY_INFO_TIME_SCAN	scan time was filled in

Description

Used by the driver to indicate which info in struct survey_info it has filled in during the `get_survey`.

Name

struct survey_info — channel survey response

Synopsis

```
struct survey_info {
  struct ieee80211_channel * channel;
  u64 time;
  u64 time_busy;
  u64 time_ext_busy;
  u64 time_rx;
  u64 time_tx;
  u64 time_scan;
  u32 filled;
  s8 noise;
};
```

Members

channel	the channel this survey record reports, may be `NULL` for a single record to report global statistics
time	amount of time in ms the radio was turn on (on the channel)
time_busy	amount of time the primary channel was sensed busy
time_ext_busy	amount of time the extension channel was sensed busy
time_rx	amount of time the radio spent receiving data
time_tx	amount of time the radio spent transmitting data
time_scan	amount of time the radio spent for scanning
filled	bitflag of flags from enum survey_info_flags
noise	channel noise in dBm. This and all following fields are optional

Description

Used by `dump_survey` to report back per-channel survey information.

This structure can later be expanded with things like channel duty cycle etc.

Name

struct cfg80211_beacon_data — beacon data

Synopsis

```
struct cfg80211_beacon_data {
  const u8 * head;
  const u8 * tail;
  const u8 * beacon_ies;
  const u8 * proberesp_ies;
  const u8 * assocresp_ies;
  const u8 * probe_resp;
  size_t head_len;
  size_t tail_len;
  size_t beacon_ies_len;
  size_t proberesp_ies_len;
  size_t assocresp_ies_len;
  size_t probe_resp_len;
};
```

Members

head	head portion of beacon (before TIM IE) or `NULL` if not changed
tail	tail portion of beacon (after TIM IE) or `NULL` if not changed
beacon_ies	extra information element(s) to add into Beacon frames or `NULL`
proberesp_ies	extra information element(s) to add into Probe Response frames or `NULL`
assocresp_ies	extra information element(s) to add into (Re)Association Response frames or `NULL`
probe_resp	probe response template (AP mode only)
head_len	length of *`head`*
tail_len	length of *`tail`*
beacon_ies_len	length of beacon_ies in octets
proberesp_ies_len	length of proberesp_ies in octets
assocresp_ies_len	length of assocresp_ies in octets
probe_resp_len	length of probe response template (*`probe_resp`*)

Name

struct cfg80211_ap_settings — AP configuration

Synopsis

```
struct cfg80211_ap_settings {
  struct cfg80211_chan_def chandef;
  struct cfg80211_beacon_data beacon;
  int beacon_interval;
  int dtim_period;
  const u8 * ssid;
  size_t ssid_len;
  enum nl80211_hidden_ssid hidden_ssid;
  struct cfg80211_crypto_settings crypto;
  bool privacy;
  enum nl80211_auth_type auth_type;
  enum nl80211_smps_mode smps_mode;
  int inactivity_timeout;
  u8 p2p_ctwindow;
  bool p2p_opp_ps;
  const struct cfg80211_acl_data * acl;
};
```

Members

chandef	defines the channel to use
beacon	beacon data
beacon_interval	beacon interval
dtim_period	DTIM period
ssid	SSID to be used in the BSS (note: may be `NULL` if not provided from user space)
ssid_len	length of *`ssid`*
hidden_ssid	whether to hide the SSID in Beacon/Probe Response frames
crypto	crypto settings
privacy	the BSS uses privacy
auth_type	Authentication type (algorithm)
smps_mode	SMPS mode
inactivity_timeout	time in seconds to determine station's inactivity.
p2p_ctwindow	P2P CT Window
p2p_opp_ps	P2P opportunistic PS

acl	ACL configuration used by the drivers which has support for MAC address based access control

Description

Used to configure an AP interface.

Name

struct station_parameters — station parameters

Synopsis

```
struct station_parameters {
  const u8 * supported_rates;
  struct net_device * vlan;
  u32 sta_flags_mask;
  u32 sta_flags_set;
  u32 sta_modify_mask;
  int listen_interval;
  u16 aid;
  u8 supported_rates_len;
  u8 plink_action;
  u8 plink_state;
  const struct ieee80211_ht_cap * ht_capa;
  const struct ieee80211_vht_cap * vht_capa;
  u8 uapsd_queues;
  u8 max_sp;
  enum nl80211_mesh_power_mode local_pm;
  u16 capability;
  const u8 * ext_capab;
  u8 ext_capab_len;
  const u8 * supported_channels;
  u8 supported_channels_len;
  const u8 * supported_oper_classes;
  u8 supported_oper_classes_len;
  u8 opmode_notif;
  bool opmode_notif_used;
};
```

Members

supported_rates	supported rates in IEEE 802.11 format (or NULL for no change)
vlan	vlan interface station should belong to
sta_flags_mask	station flags that changed (bitmask of BIT(NL80211_STA_FLAG_...))
sta_flags_set	station flags values (bitmask of BIT(NL80211_STA_FLAG_...))
sta_modify_mask	bitmap indicating which parameters changed (for those that don't have a natural “no change” value), see enum station_parameters_apply_mask
listen_interval	listen interval or -1 for no change
aid	AID or zero for no change
supported_rates_len	number of supported rates

plink_action	plink action to take
plink_state	set the peer link state for a station
ht_capa	HT capabilities of station
vht_capa	VHT capabilities of station
uapsd_queues	bitmap of queues configured for uapsd. same format as the AC bitmap in the QoS info field
max_sp	max Service Period. same format as the MAX_SP in the QoS info field (but already shifted down)
local_pm	local link-specific mesh power save mode (no change when set to unknown)
capability	station capability
ext_capab	extended capabilities of the station
ext_capab_len	number of extended capabilities
supported_channels	supported channels in IEEE 802.11 format
supported_channels_len	number of supported channels
supported_oper_classes	supported oper classes in IEEE 802.11 format
supported_oper_classes_len	number of supported operating classes
opmode_notif	operating mode field from Operating Mode Notification
opmode_notif_used	information if operating mode field is used

Description

Used to change and create a new station.

Name

enum rate_info_flags — bitrate info flags

Synopsis

```
enum rate_info_flags {
  RATE_INFO_FLAGS_MCS,
  RATE_INFO_FLAGS_VHT_MCS,
  RATE_INFO_FLAGS_SHORT_GI,
  RATE_INFO_FLAGS_60G
};
```

Constants

RATE_INFO_FLAGS_MCS	mcs field filled with HT MCS
RATE_INFO_FLAGS_VHT_MCS	mcs field filled with VHT MCS
RATE_INFO_FLAGS_SHORT_GI	400ns guard interval
RATE_INFO_FLAGS_60G	60GHz MCS

Description

Used by the driver to indicate the specific rate transmission type for 802.11n transmissions.

Name

struct rate_info — bitrate information

Synopsis

```
struct rate_info {
  u8 flags;
  u8 mcs;
  u16 legacy;
  u8 nss;
  u8 bw;
};
```

Members

flags	bitflag of flags from enum rate_info_flags
mcs	mcs index if struct describes a 802.11n bitrate
legacy	bitrate in 100kbit/s for 802.11abg
nss	number of streams (VHT only)
bw	bandwidth (from enum rate_info_bw)

Description

Information about a receiving or transmitting bitrate

Name

struct station_info — station information

Synopsis

```
struct station_info {
  u32 filled;
  u32 connected_time;
  u32 inactive_time;
  u64 rx_bytes;
  u64 tx_bytes;
  u16 llid;
  u16 plid;
  u8 plink_state;
  s8 signal;
  s8 signal_avg;
  u8 chains;
  s8 chain_signal[IEEE80211_MAX_CHAINS];
  s8 chain_signal_avg[IEEE80211_MAX_CHAINS];
  struct rate_info txrate;
  struct rate_info rxrate;
  u32 rx_packets;
  u32 tx_packets;
  u32 tx_retries;
  u32 tx_failed;
  u32 rx_dropped_misc;
  struct sta_bss_parameters bss_param;
  struct nl80211_sta_flag_update sta_flags;
  int generation;
  const u8 * assoc_req_ies;
  size_t assoc_req_ies_len;
  u32 beacon_loss_count;
  s64 t_offset;
  enum nl80211_mesh_power_mode local_pm;
  enum nl80211_mesh_power_mode peer_pm;
  enum nl80211_mesh_power_mode nonpeer_pm;
  u32 expected_throughput;
  u64 rx_beacon;
  u8 rx_beacon_signal_avg;
  struct cfg80211_tid_stats pertid[IEEE80211_NUM_TIDS + 1];
};
```

Members

filled	bitflag of flags using the bits of enum nl80211_sta_info to indicate the relevant values in this struct for them
connected_time	time(in secs) since a station is last connected
inactive_time	time since last station activity (tx/rx) in milliseconds
rx_bytes	bytes (size of MPDUs) received from this station

tx_bytes	bytes (size of MPDUs) transmitted to this station
llid	mesh local link id
plid	mesh peer link id
plink_state	mesh peer link state
signal	The signal strength, type depends on the wiphy's signal_type. For CFG80211_SIGNAL_TYPE_MBM, value is expressed in _dBm_.
signal_avg	Average signal strength, type depends on the wiphy's signal_type. For CFG80211_SIGNAL_TYPE_MBM, value is expressed in _dBm_.
chains	bitmask for filled values in *chain_signal*, *chain_signal_avg*
chain_signal[IEEE80211_MAX_CHAINS]	per-chain signal strength of last received packet in dBm
chain_signal_avg[IEEE80211_MAX_CHAINS]	per-chain signal strength average in dBm
txrate	current unicast bitrate from this station
rxrate	current unicast bitrate to this station
rx_packets	packets (MSDUs & MMPDUs) received from this station
tx_packets	packets (MSDUs & MMPDUs) transmitted to this station
tx_retries	cumulative retry counts (MPDUs)
tx_failed	number of failed transmissions (MPDUs) (retries exceeded, no ACK)
rx_dropped_misc	Dropped for un-specified reason.
bss_param	current BSS parameters
sta_flags	station flags mask & values
generation	generation number for nl80211 dumps. This number should increase every time the list of stations changes, i.e. when a station is added or removed, so that userspace can tell whether it got a consistent snapshot.
assoc_req_ies	IEs from (Re)Association Request. This is used only when in AP mode with drivers that do not use user space MLME/ SME implementation. The information is provided for the `cfg80211_new_sta` calls to notify user space of the IEs.
assoc_req_ies_len	Length of assoc_req_ies buffer in octets.
beacon_loss_count	Number of times beacon loss event has triggered.
t_offset	Time offset of the station relative to this host.
local_pm	local mesh STA power save mode

peer_pm	peer mesh STA power save mode
nonpeer_pm	non-peer mesh STA power save mode
expected_throughput	expected throughput in kbps (including 802.11 headers) towards this station.
rx_beacon	number of beacons received from this peer
rx_beacon_signal_avg	signal strength average (in dBm) for beacons received from this peer
pertid[IEEE80211_NUM_TIDS + 1]	per-TID statistics, see struct cfg80211_tid_stats, using the last (IEEE80211_NUM_TIDS) index for MSDUs not encapsulated in QoS-MPDUs.

Description

Station information filled by driver for `get_station` and dump_station.

Name

enum monitor_flags — monitor flags

Synopsis

```
enum monitor_flags {
  MONITOR_FLAG_FCSFAIL,
  MONITOR_FLAG_PLCPFAIL,
  MONITOR_FLAG_CONTROL,
  MONITOR_FLAG_OTHER_BSS,
  MONITOR_FLAG_COOK_FRAMES,
  MONITOR_FLAG_ACTIVE
};
```

Constants

MONITOR_FLAG_FCSFAIL	pass frames with bad FCS
MONITOR_FLAG_PLCPFAIL	pass frames with bad PLCP
MONITOR_FLAG_CONTROL	pass control frames
MONITOR_FLAG_OTHER_BSS	disable BSSID filtering
MONITOR_FLAG_COOK_FRAMES	report frames after processing
MONITOR_FLAG_ACTIVE	active monitor, ACKs frames on its MAC address

Description

Monitor interface configuration flags. Note that these must be the bits according to the nl80211 flags.

Name

enum mpath_info_flags — mesh path information flags

Synopsis

```
enum mpath_info_flags {
  MPATH_INFO_FRAME_QLEN,
  MPATH_INFO_SN,
  MPATH_INFO_METRIC,
  MPATH_INFO_EXPTIME,
  MPATH_INFO_DISCOVERY_TIMEOUT,
  MPATH_INFO_DISCOVERY_RETRIES,
  MPATH_INFO_FLAGS
};
```

Constants

MPATH_INFO_FRAME_QLEN	*`frame_qlen`* filled
MPATH_INFO_SN	*`sn`* filled
MPATH_INFO_METRIC	*`metric`* filled
MPATH_INFO_EXPTIME	*`exptime`* filled
MPATH_INFO_DISCOVERY_TIMEOUT	*`discovery_timeout`* filled
MPATH_INFO_DISCOVERY_RETRIES	*`discovery_retries`* filled
MPATH_INFO_FLAGS	*`flags`* filled

Description

Used by the driver to indicate which info in struct mpath_info it has filled in during `get_station` or `dump_station`.

Name

struct mpath_info — mesh path information

Synopsis

```
struct mpath_info {
  u32 filled;
  u32 frame_qlen;
  u32 sn;
  u32 metric;
  u32 exptime;
  u32 discovery_timeout;
  u8 discovery_retries;
  u8 flags;
  int generation;
};
```

Members

filled	bitfield of flags from enum mpath_info_flags
frame_qlen	number of queued frames for this destination
sn	target sequence number
metric	metric (cost) of this mesh path
exptime	expiration time for the mesh path from now, in msecs
discovery_timeout	total mesh path discovery timeout, in msecs
discovery_retries	mesh path discovery retries
flags	mesh path flags
generation	generation number for nl80211 dumps. This number should increase every time the list of mesh paths changes, i.e. when a station is added or removed, so that userspace can tell whether it got a consistent snapshot.

Description

Mesh path information filled by driver for `get_mpath` and `dump_mpath`.

Name

struct bss_parameters — BSS parameters

Synopsis

```
struct bss_parameters {
  int use_cts_prot;
  int use_short_preamble;
  int use_short_slot_time;
  const u8 * basic_rates;
  u8 basic_rates_len;
  int ap_isolate;
  int ht_opmode;
  s8 p2p_ctwindow;
  s8 p2p_opp_ps;
};
```

Members

use_cts_prot	Whether to use CTS protection (0 = no, 1 = yes, -1 = do not change)
use_short_preamble	Whether the use of short preambles is allowed (0 = no, 1 = yes, -1 = do not change)
use_short_slot_time	Whether the use of short slot time is allowed (0 = no, 1 = yes, -1 = do not change)
basic_rates	basic rates in IEEE 802.11 format (or NULL for no change)
basic_rates_len	number of basic rates
ap_isolate	do not forward packets between connected stations
ht_opmode	HT Operation mode (u16 = opmode, -1 = do not change)
p2p_ctwindow	P2P CT Window (-1 = no change)
p2p_opp_ps	P2P opportunistic PS (-1 = no change)

Description

Used to change BSS parameters (mainly for AP mode).

Name

struct ieee80211_txq_params — TX queue parameters

Synopsis

```
struct ieee80211_txq_params {
  enum nl80211_ac ac;
  u16 txop;
  u16 cwmin;
  u16 cwmax;
  u8 aifs;
};
```

Members

ac	AC identifier
txop	Maximum burst time in units of 32 usecs, 0 meaning disabled
cwmin	Minimum contention window [a value of the form 2^n-1 in the range 1..32767]
cwmax	Maximum contention window [a value of the form 2^n-1 in the range 1..32767]
aifs	Arbitration interframe space [0..255]

Name

struct cfg80211_crypto_settings — Crypto settings

Synopsis

```
struct cfg80211_crypto_settings {
  u32 wpa_versions;
  u32 cipher_group;
  int n_ciphers_pairwise;
  u32 ciphers_pairwise[NL80211_MAX_NR_CIPHER_SUITES];
  int n_akm_suites;
  u32 akm_suites[NL80211_MAX_NR_AKM_SUITES];
  bool control_port;
  __be16 control_port_ethertype;
  bool control_port_no_encrypt;
};
```

Members

wpa_versions	indicates which, if any, WPA versions are enabled (from enum nl80211_wpa_versions)
cipher_group	group key cipher suite (or 0 if unset)
n_ciphers_pairwise	number of AP supported unicast ciphers
ciphers_pairwise[NL80211_MAX_NR_CIPHER_SUITES]	unicast key cipher suites
n_akm_suites	number of AKM suites
akm_suites[NL80211_MAX_NR_AKM_SUITES]	AKM suites
control_port	Whether user space controls IEEE 802.1X port, i.e., sets/clears `NL80211_STA_FLAG_AUTHORIZED`. If true, the driver is required to assume that the port is unauthorized until authorized by user space. Otherwise, port is marked authorized by default.
control_port_ethertype	the control port protocol that should be allowed through even on unauthorized ports
control_port_no_encrypt	TRUE to prevent encryption of control port protocol frames.

Name

struct cfg80211_auth_request — Authentication request data

Synopsis

```
struct cfg80211_auth_request {
  struct cfg80211_bss * bss;
  const u8 * ie;
  size_t ie_len;
  enum nl80211_auth_type auth_type;
  const u8 * key;
  u8 key_len;
  u8 key_idx;
  const u8 * sae_data;
  size_t sae_data_len;
};
```

Members

bss	The BSS to authenticate with, the callee must obtain a reference to it if it needs to keep it.
ie	Extra IEs to add to Authentication frame or `NULL`
ie_len	Length of ie buffer in octets
auth_type	Authentication type (algorithm)
key	WEP key for shared key authentication
key_len	length of WEP key for shared key authentication
key_idx	index of WEP key for shared key authentication
sae_data	Non-IE data to use with SAE or `NULL`. This starts with Authentication transaction sequence number field.
sae_data_len	Length of sae_data buffer in octets

Description

This structure provides information needed to complete IEEE 802.11 authentication.

Name

struct cfg80211_assoc_request — (Re)Association request data

Synopsis

```
struct cfg80211_assoc_request {
  struct cfg80211_bss * bss;
  const u8 * ie;
  const u8 * prev_bssid;
  size_t ie_len;
  struct cfg80211_crypto_settings crypto;
  bool use_mfp;
  u32 flags;
  struct ieee80211_ht_cap ht_capa;
  struct ieee80211_ht_cap ht_capa_mask;
  struct ieee80211_vht_cap vht_capa;
  struct ieee80211_vht_cap vht_capa_mask;
};
```

Members

bss	The BSS to associate with. If the call is successful the driver is given a reference that it must give back to `cfg80211_send_rx_assoc` or to `cfg80211_assoc_timeout`. To ensure proper refcounting, new association requests while already associating must be rejected.
ie	Extra IEs to add to (Re)Association Request frame or `NULL`
prev_bssid	previous BSSID, if not `NULL` use reassociate frame
ie_len	Length of ie buffer in octets
crypto	crypto settings
use_mfp	Use management frame protection (IEEE 802.11w) in this association
flags	See enum cfg80211_assoc_req_flags
ht_capa	HT Capabilities over-rides. Values set in ht_capa_mask will be used in ht_capa. Unsupported values will be ignored.
ht_capa_mask	The bits of ht_capa which are to be used.
vht_capa	VHT capability override
vht_capa_mask	VHT capability mask indicating which fields to use

Description

This structure provides information needed to complete IEEE 802.11 (re)association.

Name

struct cfg80211_deauth_request — Deauthentication request data

Synopsis

```
struct cfg80211_deauth_request {
  const u8 * bssid;
  const u8 * ie;
  size_t ie_len;
  u16 reason_code;
  bool local_state_change;
};
```

Members

bssid	the BSSID of the BSS to deauthenticate from
ie	Extra IEs to add to Deauthentication frame or `NULL`
ie_len	Length of ie buffer in octets
reason_code	The reason code for the deauthentication
local_state_change	if set, change local state only and do not set a deauth frame

Description

This structure provides information needed to complete IEEE 802.11 deauthentication.

Name

struct cfg80211_disassoc_request — Disassociation request data

Synopsis

```
struct cfg80211_disassoc_request {
  struct cfg80211_bss * bss;
  const u8 * ie;
  size_t ie_len;
  u16 reason_code;
  bool local_state_change;
};
```

Members

bss	the BSS to disassociate from
ie	Extra IEs to add to Disassociation frame or `NULL`
ie_len	Length of ie buffer in octets
reason_code	The reason code for the disassociation
local_state_change	This is a request for a local state only, i.e., no Disassociation frame is to be transmitted.

Description

This structure provides information needed to complete IEEE 802.11 disassocation.

Name

struct cfg80211_ibss_params — IBSS parameters

Synopsis

```
struct cfg80211_ibss_params {
  const u8 * ssid;
  const u8 * bssid;
  struct cfg80211_chan_def chandef;
  const u8 * ie;
  u8 ssid_len;
  u8 ie_len;
  u16 beacon_interval;
  u32 basic_rates;
  bool channel_fixed;
  bool privacy;
  bool control_port;
  bool userspace_handles_dfs;
  int mcast_rate[IEEE80211_NUM_BANDS];
  struct ieee80211_ht_cap ht_capa;
  struct ieee80211_ht_cap ht_capa_mask;
};
```

Members

ssid	The SSID, will always be non-null.
bssid	Fixed BSSID requested, maybe be `NULL`, if set do not search for IBSSs with a different BSSID.
chandef	defines the channel to use if no other IBSS to join can be found
ie	information element(s) to include in the beacon
ssid_len	The length of the SSID, will always be non-zero.
ie_len	length of that
beacon_interval	beacon interval to use
basic_rates	bitmap of basic rates to use when creating the IBSS
channel_fixed	The channel should be fixed -- do not search for IBSSs to join on other channels.
privacy	this is a protected network, keys will be configured after joining
control_port	whether user space controls IEEE 802.1X port, i.e., sets/clears `NL80211_STA_FLAG_AUTHORIZED`. If true, the driver is required to assume that the port is unauthorized until authorized by user space. Otherwise, port is marked authorized by default.
userspace_handles_dfs	whether user space controls DFS operation, i.e. changes the channel when a radar is detected. This is required to operate on DFS channels.

mcast_rate[IEEE80211_NUM_BANDS]	Per-band multicast rate index + 1 (0: disabled)
ht_capa	HT Capabilities over-rides. Values set in ht_capa_mask will be used in ht_capa. Un-supported values will be ignored.
ht_capa_mask	The bits of ht_capa which are to be used.

Description

This structure defines the IBSS parameters for the `join_ibss` method.

Name

struct cfg80211_connect_params — Connection parameters

Synopsis

```
struct cfg80211_connect_params {
  struct ieee80211_channel * channel;
  struct ieee80211_channel * channel_hint;
  const u8 * bssid;
  const u8 * bssid_hint;
  const u8 * ssid;
  size_t ssid_len;
  enum nl80211_auth_type auth_type;
  const u8 * ie;
  size_t ie_len;
  bool privacy;
  enum nl80211_mfp mfp;
  struct cfg80211_crypto_settings crypto;
  const u8 * key;
  u8 key_len;
  u8 key_idx;
  u32 flags;
  int bg_scan_period;
  struct ieee80211_ht_cap ht_capa;
  struct ieee80211_ht_cap ht_capa_mask;
  struct ieee80211_vht_cap vht_capa;
  struct ieee80211_vht_cap vht_capa_mask;
};
```

Members

channel	The channel to use or `NULL` if not specified (auto-select based on scan results)
channel_hint	The channel of the recommended BSS for initial connection or `NULL` if not specified
bssid	The AP BSSID or `NULL` if not specified (auto-select based on scan results)
bssid_hint	The recommended AP BSSID for initial connection to the BSS or `NULL` if not specified. Unlike the *`bssid`* parameter, the driver is allowed to ignore this *`bssid_hint`* if it has knowledge of a better BSS to use.
ssid	SSID
ssid_len	Length of ssid in octets
auth_type	Authentication type (algorithm)
ie	IEs for association request
ie_len	Length of assoc_ie in octets
privacy	indicates whether privacy-enabled APs should be used

mfp	indicate whether management frame protection is used
crypto	crypto settings
key	WEP key for shared key authentication
key_len	length of WEP key for shared key authentication
key_idx	index of WEP key for shared key authentication
flags	See enum cfg80211_assoc_req_flags
bg_scan_period	Background scan period in seconds or -1 to indicate that default value is to be used.
ht_capa	HT Capabilities over-rides. Values set in ht_capa_mask will be used in ht_capa. Un-supported values will be ignored.
ht_capa_mask	The bits of ht_capa which are to be used.
vht_capa	VHT Capability overrides
vht_capa_mask	The bits of vht_capa which are to be used.

Description

This structure provides information needed to complete IEEE 802.11 authentication and association.

Name

struct cfg80211_pmksa — PMK Security Association

Synopsis

```
struct cfg80211_pmksa {
  const u8 * bssid;
  const u8 * pmkid;
};
```

Members

bssid The AP's BSSID.

pmkid The PMK material itself.

Description

This structure is passed to the set/`del_pmksa` method for PMKSA caching.

Name

cfg80211_rx_mlme_mgmt — notification of processed MLME management frame

Synopsis

```
void cfg80211_rx_mlme_mgmt (struct net_device * dev, const u8 * buf,
size_t len);
```

Arguments

dev network device

buf authentication frame (header + body)

len length of the frame data

Description

This function is called whenever an authentication, disassociation or deauthentication frame has been received and processed in station mode.

After being asked to authenticate via cfg80211_ops

:auth the driver must call either this function or cfg80211_auth_timeout.

After being asked to associate via cfg80211_ops

:assoc the driver must call either this function or cfg80211_auth_timeout. While connected, the driver must calls this for received and processed disassociation and deauthentication frames. If the frame couldn't be used because it was unprotected, the driver must call the function cfg80211_rx_unprot_mlme_mgmt instead.

This function may sleep. The caller must hold the corresponding wdev's mutex.

Name

cfg80211_auth_timeout — notification of timed out authentication

Synopsis

```
void cfg80211_auth_timeout (struct net_device * dev, const u8 * addr);
```

Arguments

dev network device

addr The MAC address of the device with which the authentication timed out

Description

This function may sleep. The caller must hold the corresponding wdev's mutex.

Name

cfg80211_rx_assoc_resp — notification of processed association response

Synopsis

```
void cfg80211_rx_assoc_resp (struct net_device * dev, struct
cfg80211_bss * bss, const u8 * buf, size_t len, int uapsd_queues);
```

Arguments

dev	network device
bss	the BSS that association was requested with, ownership of the pointer moves to cfg80211 in this call
buf	authentication frame (header + body)
len	length of the frame data
uapsd_queues	bitmap of ACs configured to uapsd. -1 if n/a.

After being asked to associate via cfg80211_ops

:assoc the driver must call either this function or `cfg80211_auth_timeout`.

This function may sleep. The caller must hold the corresponding wdev's mutex.

Name

cfg80211_assoc_timeout — notification of timed out association

Synopsis

```
void cfg80211_assoc_timeout (struct net_device * dev, struct
cfg80211_bss * bss);
```

Arguments

dev network device

bss The BSS entry with which association timed out.

Description

This function may sleep. The caller must hold the corresponding wdev's mutex.

Name

cfg80211_tx_mlme_mgmt — notification of transmitted deauth/disassoc frame

Synopsis

```
void cfg80211_tx_mlme_mgmt (struct net_device * dev, const u8 * buf,
size_t len);
```

Arguments

dev network device

buf 802.11 frame (header + body)

len length of the frame data

Description

This function is called whenever deauthentication has been processed in station mode. This includes both received deauthentication frames and locally generated ones. This function may sleep. The caller must hold the corresponding wdev's mutex.

Name

cfg80211_ibss_joined — notify cfg80211 that device joined an IBSS

Synopsis

```
void cfg80211_ibss_joined (struct net_device * dev, const u8 * bssid,
struct ieee80211_channel * channel, gfp_t gfp);
```

Arguments

dev	network device
bssid	the BSSID of the IBSS joined
channel	the channel of the IBSS joined
gfp	allocation flags

Description

This function notifies cfg80211 that the device joined an IBSS or switched to a different BSSID. Before this function can be called, either a beacon has to have been received from the IBSS, or one of the cfg80211_inform_bss{,_frame} functions must have been called with the locally generated beacon -- this guarantees that there is always a scan result for this IBSS. cfg80211 will handle the rest.

Name

cfg80211_connect_result — notify cfg80211 of connection result

Synopsis

```
void cfg80211_connect_result (struct net_device * dev, const u8 * bssid,
const u8 * req_ie, size_t req_ie_len, const u8 * resp_ie, size_t
resp_ie_len, u16 status, gfp_t gfp);
```

Arguments

dev	network device
bssid	the BSSID of the AP
req_ie	association request IEs (maybe be NULL)
req_ie_len	association request IEs length
resp_ie	association response IEs (may be NULL)
resp_ie_len	assoc response IEs length
status	status code, 0 for successful connection, use WLAN_STATUS_UNSPECIFIED_FAILURE if your device cannot give you the real status code for failures.
gfp	allocation flags

Description

It should be called by the underlying driver whenever connect has succeeded.

Name

cfg80211_roamed — notify cfg80211 of roaming

Synopsis

```
void cfg80211_roamed (struct net_device * dev, struct ieee80211_channel
* channel, const u8 * bssid, const u8 * req_ie, size_t req_ie_len, const
u8 * resp_ie, size_t resp_ie_len, gfp_t gfp);
```

Arguments

dev	network device
channel	the channel of the new AP
bssid	the BSSID of the new AP
req_ie	association request IEs (maybe be NULL)
req_ie_len	association request IEs length
resp_ie	association response IEs (may be NULL)
resp_ie_len	assoc response IEs length
gfp	allocation flags

Description

It should be called by the underlying driver whenever it roamed from one AP to another while connected.

Name

cfg80211_disconnected — notify cfg80211 that connection was dropped

Synopsis

```
void cfg80211_disconnected (struct net_device * dev, u16 reason, const
u8 * ie, size_t ie_len, gfp_t gfp);
```

Arguments

dev	network device
reason	reason code for the disconnection, set it to 0 if unknown
ie	information elements of the deauth/disassoc frame (may be NULL)
ie_len	length of IEs
gfp	allocation flags

Description

After it calls this function, the driver should enter an idle state and not try to connect to any AP any more.

Name

cfg80211_ready_on_channel — notification of remain_on_channel start

Synopsis

```
void cfg80211_ready_on_channel (struct wireless_dev * wdev, u64 cookie,
struct ieee80211_channel * chan, unsigned int duration, gfp_t gfp);
```

Arguments

wdev	wireless device
cookie	the request cookie
chan	The current channel (from remain_on_channel request)
duration	Duration in milliseconds that the driver intents to remain on the channel
gfp	allocation flags

Name

cfg80211_remain_on_channel_expired — remain_on_channel duration expired

Synopsis

```
void cfg80211_remain_on_channel_expired (struct wireless_dev * wdev,
u64 cookie, struct ieee80211_channel * chan, gfp_t gfp);
```

Arguments

wdev wireless device

cookie the request cookie

chan The current channel (from remain_on_channel request)

gfp allocation flags

Name

cfg80211_new_sta — notify userspace about station

Synopsis

```
void cfg80211_new_sta (struct net_device * dev, const u8 * mac_addr,
struct station_info * sinfo, gfp_t gfp);
```

Arguments

dev	the netdev
mac_addr	the station's address
sinfo	the station information
gfp	allocation flags

Name

cfg80211_rx_mgmt — notification of received, unprocessed management frame

Synopsis

```
bool cfg80211_rx_mgmt (struct wireless_dev * wdev, int freq, int sig_dbm,
const u8 * buf, size_t len, u32 flags);
```

Arguments

wdev	wireless device receiving the frame
freq	Frequency on which the frame was received in MHz
sig_dbm	signal strength in mBm, or 0 if unknown
buf	Management frame (header + body)
len	length of the frame data
flags	flags, as defined in enum nl80211_rxmgmt_flags

Description

This function is called whenever an Action frame is received for a station mode interface, but is not processed in kernel.

Return

`true` if a user space application has registered for this frame. For action frames, that makes it responsible for rejecting unrecognized action frames; `false` otherwise, in which case for action frames the driver is responsible for rejecting the frame.

Name

cfg80211_mgmt_tx_status — notification of TX status for management frame

Synopsis

```
void cfg80211_mgmt_tx_status (struct wireless_dev * wdev, u64 cookie,
const u8 * buf, size_t len, bool ack, gfp_t gfp);
```

Arguments

wdev wireless device receiving the frame

cookie Cookie returned by cfg80211_ops::mgmt_tx

buf Management frame (header + body)

len length of the frame data

ack Whether frame was acknowledged

gfp context flags

Description

This function is called whenever a management frame was requested to be

transmitted with cfg80211_ops

:mgmt_tx to report the TX status of the transmission attempt.

Name

cfg80211_cqm_rssi_notify — connection quality monitoring rssi event

Synopsis

```
void cfg80211_cqm_rssi_notify (struct net_device * dev, enum
nl80211_cqm_rssi_threshold_event rssi_event, gfp_t gfp);
```

Arguments

dev	network device
rssi_event	the triggered RSSI event
gfp	context flags

Description

This function is called when a configured connection quality monitoring rssi threshold reached event occurs.

Name

cfg80211_cqm_pktloss_notify — notify userspace about packetloss to peer

Synopsis

```
void cfg80211_cqm_pktloss_notify (struct net_device * dev, const u8 *
peer, u32 num_packets, gfp_t gfp);
```

Arguments

dev	network device
peer	peer's MAC address
num_packets	how many packets were lost -- should be a fixed threshold but probably no less than maybe 50, or maybe a throughput dependent threshold (to account for temporary interference)
gfp	context flags

Name

cfg80211_michael_mic_failure — notification of Michael MIC failure (TKIP)

Synopsis

```
void cfg80211_michael_mic_failure (struct net_device * dev, const u8
* addr, enum nl80211_key_type key_type, int key_id, const u8 * tsc,
gfp_t gfp);
```

Arguments

dev	network device
addr	The source MAC address of the frame
key_type	The key type that the received frame used
key_id	Key identifier (0..3). Can be -1 if missing.
tsc	The TSC value of the frame that generated the MIC failure (6 octets)
gfp	allocation flags

Description

This function is called whenever the local MAC detects a MIC failure in a received frame. This matches with MLME-MICHAELMICFAILURE.`indication` primitive.

Chapter 3. Scanning and BSS list handling

The scanning process itself is fairly simple, but cfg80211 offers quite a bit of helper functionality. To start a scan, the scan operation will be invoked with a scan definition. This scan definition contains the channels to scan, and the SSIDs to send probe requests for (including the wildcard, if desired). A passive scan is indicated by having no SSIDs to probe. Additionally, a scan request may contain extra information elements that should be added to the probe request. The IEs are guaranteed to be well-formed, and will not exceed the maximum length the driver advertised in the wiphy structure.

When scanning finds a BSS, cfg80211 needs to be notified of that, because it is responsible for maintaining the BSS list; the driver should not maintain a list itself. For this notification, various functions exist.

Since drivers do not maintain a BSS list, there are also a number of functions to search for a BSS and obtain information about it from the BSS structure cfg80211 maintains. The BSS list is also made available to userspace.

Name

struct cfg80211_ssid — SSID description

Synopsis

```
struct cfg80211_ssid {
  u8 ssid[IEEE80211_MAX_SSID_LEN];
  u8 ssid_len;
};
```

Members

ssid[IEEE80211_MAX_SSID_LEN]	the SSID
ssid_len	length of the ssid

Name

struct cfg80211_scan_request — scan request description

Synopsis

```
struct cfg80211_scan_request {
  struct cfg80211_ssid * ssids;
  int n_ssids;
  u32 n_channels;
  enum nl80211_bss_scan_width scan_width;
  const u8 * ie;
  size_t ie_len;
  u32 flags;
  u32 rates[IEEE80211_NUM_BANDS];
  struct wireless_dev * wdev;
  u8 mac_addr[ETH_ALEN];
  u8 mac_addr_mask[ETH_ALEN];
  struct wiphy * wiphy;
  unsigned long scan_start;
  bool aborted;
  bool notified;
  bool no_cck;
  struct ieee80211_channel * channels[0];
};
```

Members

ssids	SSIDs to scan for (active scan only)
n_ssids	number of SSIDs
n_channels	total number of channels to scan
scan_width	channel width for scanning
ie	optional information element(s) to add into Probe Request or `NULL`
ie_len	length of ie in octets
flags	bit field of flags controlling operation
rates[IEEE80211_NUM_BANDS]	bitmap of rates to advertise for each band
wdev	the wireless device to scan for
mac_addr[ETH_ALEN]	MAC address used with randomisation
mac_addr_mask[ETH_ALEN]	MAC address mask used with randomisation, bits that are 0 in the mask should be randomised, bits that are 1 should be taken from the *`mac_addr`*
wiphy	the wiphy this was for
scan_start	time (in jiffies) when the scan started

aborted	(internal) scan request was notified as aborted
notified	(internal) scan request was notified as done or aborted
no_cck	used to send probe requests at non CCK rate in 2GHz band
channels[0]	channels to scan on.

Name

cfg80211_scan_done — notify that scan finished

Synopsis

```
void cfg80211_scan_done (struct cfg80211_scan_request * request, bool
aborted);
```

Arguments

request the corresponding scan request

aborted set to true if the scan was aborted for any reason, userspace will be notified of that

Name

struct cfg80211_bss — BSS description

Synopsis

```
struct cfg80211_bss {
  struct ieee80211_channel * channel;
  enum nl80211_bss_scan_width scan_width;
  const struct cfg80211_bss_ies __rcu * ies;
  const struct cfg80211_bss_ies __rcu * beacon_ies;
  const struct cfg80211_bss_ies __rcu * proberesp_ies;
  struct cfg80211_bss * hidden_beacon_bss;
  s32 signal;
  u16 beacon_interval;
  u16 capability;
  u8 bssid[ETH_ALEN];
  u8 priv[0];
};
```

Members

channel	channel this BSS is on
scan_width	width of the control channel
ies	the information elements (Note that there is no guarantee that these are well-formed!); this is a pointer to either the beacon_ies or proberesp_ies depending on whether Probe Response frame has been received. It is always non-`NULL`.
beacon_ies	the information elements from the last Beacon frame (implementation note: if *`hidden_beacon_bss`* is set this struct doesn't own the beacon_ies, but they're just pointers to the ones from the *`hidden_beacon_bss`* struct)
proberesp_ies	the information elements from the last Probe Response frame
hidden_beacon_bss	in case this BSS struct represents a probe response from a BSS that hides the SSID in its beacon, this points to the BSS struct that holds the beacon data. *`beacon_ies`* is still valid, of course, and points to the same data as hidden_beacon_bss->beacon_ies in that case.
signal	signal strength value (type depends on the wiphy's signal_type)
beacon_interval	the beacon interval as from the frame
capability	the capability field in host byte order
bssid[ETH_ALEN]	BSSID of the BSS
priv[0]	private area for driver use, has at least wiphy->bss_priv_size bytes

Description

This structure describes a BSS (which may also be a mesh network) for use in scan results and similar.

Name

cfg80211_inform_bss_width_frame — inform cfg80211 of a received BSS frame

Synopsis

```
struct cfg80211_bss * cfg80211_inform_bss_width_frame (struct wiphy * wiphy, struct ieee80211_channel * rx_channel, enum nl80211_bss_scan_width scan_width, struct ieee80211_mgmt * mgmt, size_t len, s32 signal, gfp_t gfp);
```

Arguments

wiphy	the wiphy reporting the BSS
rx_channel	The channel the frame was received on
scan_width	width of the control channel
mgmt	the management frame (probe response or beacon)
len	length of the management frame
signal	the signal strength, type depends on the wiphy's signal_type
gfp	context flags

Description

This informs cfg80211 that BSS information was found and the BSS should be updated/added.

Return

A referenced struct, must be released with `cfg80211_put_bss`! Or `NULL` on error.

Name

cfg80211_inform_bss_width — inform cfg80211 of a new BSS

Synopsis

```
struct cfg80211_bss * cfg80211_inform_bss_width (struct wiphy * wiphy,
struct ieee80211_channel * rx_channel, enum nl80211_bss_scan_width
scan_width, enum cfg80211_bss_frame_type ftype, const u8 * bssid, u64
tsf, u16 capability, u16 beacon_interval, const u8 * ie, size_t ielen,
s32 signal, gfp_t gfp);
```

Arguments

wiphy	the wiphy reporting the BSS
rx_channel	The channel the frame was received on
scan_width	width of the control channel
ftype	frame type (if known)
bssid	the BSSID of the BSS
tsf	the TSF sent by the peer in the beacon/probe response (or 0)
capability	the capability field sent by the peer
beacon_interval	the beacon interval announced by the peer
ie	additional IEs sent by the peer
ielen	length of the additional IEs
signal	the signal strength, type depends on the wiphy's signal_type
gfp	context flags

Description

This informs cfg80211 that BSS information was found and the BSS should be updated/added.

Return

A referenced struct, must be released with `cfg80211_put_bss!` Or `NULL` on error.

Name

cfg80211_unlink_bss — unlink BSS from internal data structures

Synopsis

```
void cfg80211_unlink_bss (struct wiphy * wiphy, struct cfg80211_bss *
bss);
```

Arguments

wiphy the wiphy

bss the bss to remove

Description

This function removes the given BSS from the internal data structures thereby making it no longer show up in scan results etc. Use this function when you detect a BSS is gone. Normally BSSes will also time out, so it is not necessary to use this function at all.

Name

cfg80211_find_ie — find information element in data

Synopsis

```
const u8 * cfg80211_find_ie (u8 eid, const u8 * ies, int len);
```

Arguments

eid element ID

ies data consisting of IEs

len length of data

Return

NULL if the element ID could not be found or if the element is invalid (claims to be longer than the given data), or a pointer to the first byte of the requested element, that is the byte containing the element ID.

Note

There are no checks on the element length other than having to fit into the given data.

Name

ieee80211_bss_get_ie — find IE with given ID

Synopsis

```
const u8 * ieee80211_bss_get_ie (struct cfg80211_bss * bss, u8 ie);
```

Arguments

bss the bss to search

ie the IE ID

Description

Note that the return value is an RCU-protected pointer, so `rcu_read_lock` must be held when calling this function.

Return

`NULL` if not found.

Chapter 4. Utility functions

cfg80211 offers a number of utility functions that can be useful.

Name

ieee80211_channel_to_frequency — convert channel number to frequency

Synopsis

```
int ieee80211_channel_to_frequency (int chan, enum ieee80211_band band);
```

Arguments

chan channel number

band band, necessary due to channel number overlap

Return

The corresponding frequency (in MHz), or 0 if the conversion failed.

Name

ieee80211_frequency_to_channel — convert frequency to channel number

Synopsis

```
int ieee80211_frequency_to_channel (int freq);
```

Arguments

freq center frequency

Return

The corresponding channel, or 0 if the conversion failed.

Name

ieee80211_get_channel — get channel struct from wiphy for specified frequency

Synopsis

```
struct ieee80211_channel * ieee80211_get_channel (struct wiphy * wiphy,
int freq);
```

Arguments

wiphy the struct wiphy to get the channel for

freq the center frequency of the channel

Return

The channel struct from *wiphy* at *freq*.

Name

ieee80211_get_response_rate — get basic rate for a given rate

Synopsis

```
struct ieee80211_rate * ieee80211_get_response_rate (struct
ieee80211_supported_band * sband, u32 basic_rates, int bitrate);
```

Arguments

sband	the band to look for rates in
basic_rates	bitmap of basic rates
bitrate	the bitrate for which to find the basic rate

Return

The basic rate corresponding to a given bitrate, that is the next lower bitrate contained in the basic rate map, which is, for this function, given as a bitmap of indices of rates in the band's bitrate table.

Name

ieee80211_hdrlen — get header length in bytes from frame control

Synopsis

```
unsigned int __attribute_const__ ieee80211_hdrlen (__le16 fc);
```

Arguments

fc frame control field in little-endian format

Return

The header length in bytes.

Name

ieee80211_get_hdrlen_from_skb — get header length from data

Synopsis

```
unsigned int ieee80211_get_hdrlen_from_skb (const struct sk_buff * skb);
```

Arguments

skb the frame

Description

Given an skb with a raw 802.11 header at the data pointer this function returns the 802.11 header length.

Return

The 802.11 header length in bytes (not including encryption headers). Or 0 if the data in the sk_buff is too short to contain a valid 802.11 header.

Name

struct ieee80211_radiotap_iterator — tracks walk thru present radiotap args

Synopsis

```
struct ieee80211_radiotap_iterator {
  struct ieee80211_radiotap_header * _rtheader;
  const struct ieee80211_radiotap_vendor_namespaces * _vns;
  const struct ieee80211_radiotap_namespace * current_namespace;
  unsigned char * _arg;
  unsigned char * _next_ns_data;
  __le32 * _next_bitmap;
  unsigned char * this_arg;
  int this_arg_index;
  int this_arg_size;
  int is_radiotap_ns;
  int _max_length;
  int _arg_index;
  uint32_t _bitmap_shifter;
  int _reset_on_ext;
};
```

Members

_rtheader	pointer to the radiotap header we are walking through
_vns	vendor namespace definitions
current_namespace	pointer to the current namespace definition (or internally `NULL` if the current namespace is unknown)
_arg	next argument pointer
_next_ns_data	beginning of the next namespace's data
_next_bitmap	internal pointer to next present u32
this_arg	pointer to current radiotap arg; it is valid after each call to `ieee80211_radiotap_iterator_next` but also after `ieee80211_radiotap_iterator_init` where it will point to the beginning of the actual data portion
this_arg_index	index of current arg, valid after each successful call to `ieee80211_radiotap_iterator_next`
this_arg_size	length of the current arg, for convenience
is_radiotap_ns	indicates whether the current namespace is the default radiotap namespace or not
_max_length	length of radiotap header in cpu byte ordering
_arg_index	next argument index

_bitmap_shifter	internal shifter for curr u32 bitmap, b0 set == arg present
_reset_on_ext	internal; reset the arg index to 0 when going to the next bitmap word

Description

Describes the radiotap parser state. Fields prefixed with an underscore must not be used by users of the parser, only by the parser internally.

Chapter 5. Data path helpers

In addition to generic utilities, cfg80211 also offers functions that help implement the data path for devices that do not do the 802.11/802.3 conversion on the device.

Name

ieee80211_data_to_8023 — convert an 802.11 data frame to 802.3

Synopsis

```
int ieee80211_data_to_8023 (struct sk_buff * skb, const u8 * addr, enum
nl80211_iftype iftype);
```

Arguments

skb the 802.11 data frame

addr the device MAC address

iftype the virtual interface type

Return

0 on success. Non-zero on error.

Name

ieee80211_data_from_8023 — convert an 802.3 frame to 802.11

Synopsis

```
int ieee80211_data_from_8023 (struct sk_buff * skb, const u8 * addr,
enum nl80211_iftype iftype, const u8 * bssid, bool qos);
```

Arguments

skb	the 802.3 frame
addr	the device MAC address
iftype	the virtual interface type
bssid	the network bssid (used only for iftype STATION and ADHOC)
qos	build 802.11 QoS data frame

Return

0 on success, or a negative error code.

Name

ieee80211_amsdu_to_8023s — decode an IEEE 802.11n A-MSDU frame

Synopsis

```
void ieee80211_amsdu_to_8023s (struct sk_buff * skb, struct sk_buff_head
* list, const u8 * addr, enum nl80211_iftype iftype, const unsigned int
extra_headroom, bool has_80211_header);
```

Arguments

skb	The input IEEE 802.11n A-MSDU frame.
list	The output list of 802.3 frames. It must be allocated and initialized by by the caller.
addr	The device MAC address.
iftype	The device interface type.
extra_headroom	The hardware extra headroom for SKBs in the *list*.
has_80211_header	Set it true if SKB is with IEEE 802.11 header.

Description

Decode an IEEE 802.11n A-MSDU frame and convert it to a list of 802.3 frames. The *list* will be empty if the decode fails. The *skb* is consumed after the function returns.

Name

cfg80211_classify8021d — determine the 802.1p/1d tag for a data frame

Synopsis

```
unsigned int cfg80211_classify8021d (struct sk_buff * skb, struct
cfg80211_qos_map * qos_map);
```

Arguments

skb the data frame

qos_map Interworking QoS mapping or NULL if not in use

Return

The 802.1p/1d tag.

Chapter 6. Regulatory enforcement infrastructure

TODO

Name

regulatory_hint — driver hint to the wireless core a regulatory domain

Synopsis

```
int regulatory_hint (struct wiphy * wiphy, const char * alpha2);
```

Arguments

wiphy the wireless device giving the hint (used only for reporting conflicts)

alpha2 the ISO/IEC 3166 alpha2 the driver claims its regulatory domain should be in. If *rd* is set this should be NULL. Note that if you set this to NULL you should still set rd->alpha2 to some accepted alpha2.

Description

Wireless drivers can use this function to hint to the wireless core what it believes should be the current regulatory domain by giving it an ISO/IEC 3166 alpha2 country code it knows its regulatory domain should be in or by providing a completely build regulatory domain. If the driver provides an ISO/IEC 3166 alpha2 userspace will be queried for a regulatory domain structure for the respective country.

The wiphy must have been registered to cfg80211 prior to this call. For cfg80211 drivers this means you must first use `wiphy_register`, for mac80211 drivers you must first use `ieee80211_register_hw`.

Drivers should check the return value, its possible you can get an -ENOMEM.

Return

0 on success. -ENOMEM.

Name

wiphy_apply_custom_regulatory — apply a custom driver regulatory domain

Synopsis

```
void wiphy_apply_custom_regulatory (struct wiphy * wiphy, const struct
ieee80211_regdomain * regd);
```

Arguments

wiphy the wireless device we want to process the regulatory domain on

regd the custom regulatory domain to use for this wiphy

Description

Drivers can sometimes have custom regulatory domains which do not apply to a specific country. Drivers can use this to apply such custom regulatory domains. This routine must be called prior to wiphy registration. The custom regulatory domain will be trusted completely and as such previous default channel settings will be disregarded. If no rule is found for a channel on the regulatory domain the channel will be disabled. Drivers using this for a wiphy should also set the wiphy flag REGULATORY_CUSTOM_REG or cfg80211 will set it for the wiphy that called this helper.

Name

freq_reg_info — get regulatory information for the given frequency

Synopsis

```
const struct ieee80211_reg_rule * freq_reg_info (struct wiphy * wiphy,
u32 center_freq);
```

Arguments

wiphy	the wiphy for which we want to process this rule for
center_freq	Frequency in KHz for which we want regulatory information for

Description

Use this function to get the regulatory rule for a specific frequency on a given wireless device. If the device has a specific regulatory domain it wants to follow we respect that unless a country IE has been received and processed already.

Return

A valid pointer, or, when an error occurs, for example if no rule can be found, the return value is encoded using ERR_PTR. Use IS_ERR to check and PTR_ERR to obtain the numeric return value. The numeric return value will be -ERANGE if we determine the given center_freq does not even have a regulatory rule for a frequency range in the center_freq's band. See freq_in_rule_band for our current definition of a band -- this is purely subjective and right now it's 802.11 specific.

Chapter 7. RFkill integration

RFkill integration in cfg80211 is almost invisible to drivers, as cfg80211 automatically registers an rfkill instance for each wireless device it knows about. Soft kill is also translated into disconnecting and turning all interfaces off, drivers are expected to turn off the device when all interfaces are down.

However, devices may have a hard RFkill line, in which case they also need to interact with the rfkill subsystem, via cfg80211. They can do this with a few helper functions documented here.

Name

wiphy_rfkill_set_hw_state — notify cfg80211 about hw block state

Synopsis

```
void wiphy_rfkill_set_hw_state (struct wiphy * wiphy, bool blocked);
```

Arguments

wiphy the wiphy

blocked block status

Name

wiphy_rfkill_start_polling — start polling rfkill

Synopsis

```
void wiphy_rfkill_start_polling (struct wiphy * wiphy);
```

Arguments

wiphy the wiphy

Name

wiphy_rfkill_stop_polling — stop polling rfkill

Synopsis

```
void wiphy_rfkill_stop_polling (struct wiphy * wiphy);
```

Arguments

wiphy the wiphy

Chapter 8. Test mode

Test mode is a set of utility functions to allow drivers to interact with driver-specific tools to aid, for instance, factory programming.

This chapter describes how drivers interact with it, for more information see the nl80211 book's chapter on it.

Name

cfg80211_testmode_alloc_reply_skb — allocate testmode reply

Synopsis

```
struct sk_buff * cfg80211_testmode_alloc_reply_skb (struct wiphy *
wiphy, int approxlen);
```

Arguments

wiphy the wiphy

approxlen an upper bound of the length of the data that will be put into the skb

Description

This function allocates and pre-fills an skb for a reply to the testmode command. Since it is intended for a reply, calling it outside of the *testmode_cmd* operation is invalid.

The returned skb is pre-filled with the wiphy index and set up in a way that any data that is put into the skb (with skb_put, nla_put or similar) will end up being within the NL80211_ATTR_TESTDATA attribute, so all that needs to be done with the skb is adding data for the corresponding userspace tool which can then read that data out of the testdata attribute. You must not modify the skb in any other way.

When done, call cfg80211_testmode_reply with the skb and return its error code as the result of the *testmode_cmd* operation.

Return

An allocated and pre-filled skb. NULL if any errors happen.

Name

cfg80211_testmode_reply — send the reply skb

Synopsis

```
int cfg80211_testmode_reply (struct sk_buff * skb);
```

Arguments

skb The skb, must have been allocated with `cfg80211_testmode_alloc_reply_skb`

Description

Since calling this function will usually be the last thing before returning from the *`testmode_cmd`* you should return the error code. Note that this function consumes the skb regardless of the return value.

Return

An error code or 0 on success.

Name

cfg80211_testmode_alloc_event_skb — allocate testmode event

Synopsis

```
struct sk_buff * cfg80211_testmode_alloc_event_skb (struct wiphy *
wiphy, int approxlen, gfp_t gfp);
```

Arguments

wiphy the wiphy

approxlen an upper bound of the length of the data that will be put into the skb

gfp allocation flags

Description

This function allocates and pre-fills an skb for an event on the testmode multicast group.

The returned skb is set up in the same way as with `cfg80211_testmode_alloc_reply_skb` but prepared for an event. As there, you should simply add data to it that will then end up in the `NL80211_ATTR_TESTDATA` attribute. Again, you must not modify the skb in any other way.

When done filling the skb, call `cfg80211_testmode_event` with the skb to send the event.

Return

An allocated and pre-filled skb. `NULL` if any errors happen.

Name

cfg80211_testmode_event — send the event

Synopsis

```
void cfg80211_testmode_event (struct sk_buff * skb, gfp_t gfp);
```

Arguments

skb The skb, must have been allocated with `cfg80211_testmode_alloc_event_skb`

gfp allocation flags

Description

This function sends the given *skb*, which must have been allocated by `cfg80211_testmode_alloc_event_skb`, as an event. It always consumes it.

The mac80211 subsystem

The mac80211 subsystem

Abstract

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.

Table of Contents

I. The basic mac80211 driver interface 1
1. Basic hardware handling 3
struct ieee80211_hw 4
enum ieee80211_hw_flags 7
SET_IEEE80211_DEV 10
SET_IEEE80211_PERM_ADDR 11
struct ieee80211_ops 12
ieee80211_alloc_hw 23
ieee80211_register_hw 24
ieee80211_unregister_hw 25
ieee80211_free_hw 26
2. PHY configuration 27
struct ieee80211_conf 28
enum ieee80211_conf_flags 30
3. Virtual interfaces 31
struct ieee80211_vif 32
4. Receive and transmit processing 34
what should be here 34
Frame format 34
Packet alignment 34
Calling into mac80211 from interrupts 34
functions/definitions 35
5. Frame filtering 65
enum ieee80211_filter_flags 66
6. The mac80211 workqueue 67
ieee80211_queue_work 68
ieee80211_queue_delayed_work 69
II. Advanced driver interface 70
7. LED support 72
ieee80211_get_tx_led_name 73
ieee80211_get_rx_led_name 74
ieee80211_get_assoc_led_name 75
ieee80211_get_radio_led_name 76
struct ieee80211_tpt_blink 77
enum ieee80211_tpt_led_trigger_flags 78
ieee80211_create_tpt_led_trigger 79
8. Hardware crypto acceleration 80
enum set_key_cmd 81
struct ieee80211_key_conf 82
enum ieee80211_key_flags 83
ieee80211_get_tkip_p1k 85
ieee80211_get_tkip_p1k_iv 86
ieee80211_get_tkip_p2k 87
9. Powersave support 88
10. Beacon filter support 89
ieee80211_beacon_loss 90
11. Multiple queues and QoS support 91
struct ieee80211_tx_queue_params 92
12. Access point mode support 93
support for powersaving clients 93
13. Supporting multiple virtual interfaces 103

ieee80211_iterate_active_interfaces 104
ieee80211_iterate_active_interfaces_atomic 105
14. Station handling 106
struct ieee80211_sta 107
enum sta_notify_cmd 109
ieee80211_find_sta 110
ieee80211_find_sta_by_ifaddr 111
15. Hardware scan offload 112
ieee80211_scan_completed 113
16. Aggregation 114
TX A-MPDU aggregation 114
RX A-MPDU aggregation 114
17. Spatial Multiplexing Powersave (SMPS) 116
ieee80211_request_smps 117
enum ieee80211_smps_mode 118
III. Rate control interface 119
18. Rate Control API 121
ieee80211_start_tx_ba_session 122
ieee80211_start_tx_ba_cb_irqsafe 123
ieee80211_stop_tx_ba_session 124
ieee80211_stop_tx_ba_cb_irqsafe 125
enum ieee80211_rate_control_changed 126
struct ieee80211_tx_rate_control 127
rate_control_send_low 128
IV. Internals 129
19. Key handling 131
Key handling basics 131
MORE TBD 131
20. Receive processing 132
21. Transmit processing 133
22. Station info handling 134
Programming information 134
STA information lifetime rules 141
23. Aggregation 143
struct sta_ampdu_mlme 144
struct tid_ampdu_tx 145
struct tid_ampdu_rx 147
24. Synchronisation 149

Part I. The basic mac80211 driver interface

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.

Table of Contents

1. Basic hardware handling 3
    - struct ieee80211_hw 4
    - enum ieee80211_hw_flags 7
    - SET_IEEE80211_DEV 10
    - SET_IEEE80211_PERM_ADDR 11
    - struct ieee80211_ops 12
    - ieee80211_alloc_hw 23
    - ieee80211_register_hw 24
    - ieee80211_unregister_hw 25
    - ieee80211_free_hw 26
2. PHY configuration 27
    - struct ieee80211_conf 28
    - enum ieee80211_conf_flags 30
3. Virtual interfaces 31
    - struct ieee80211_vif 32
4. Receive and transmit processing 34
    - what should be here 34
    - Frame format 34
    - Packet alignment 34
    - Calling into mac80211 from interrupts 34
    - functions/definitions 35
5. Frame filtering 65
    - enum ieee80211_filter_flags 66
6. The mac80211 workqueue 67
    - ieee80211_queue_work 68
    - ieee80211_queue_delayed_work 69

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.

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;
  unsigned int extra_beacon_tailroom;
  int vif_data_size;
  int sta_data_size;
  int chanctx_data_size;
  int txq_data_size;
  u16 queues;
  u16 max_listen_interval;
  s8 max_signal;
  u8 max_rates;
  u8 max_report_rates;
  u8 max_rate_tries;
  u8 max_rx_aggregation_subframes;
  u8 max_tx_aggregation_subframes;
  u8 offchannel_tx_hw_queue;
  u8 radiotap_mcs_details;
  u16 radiotap_vht_details;
  netdev_features_t netdev_features;
  u8 uapsd_queues;
  u8 uapsd_max_sp_len;
  u8 n_cipher_schemes;
  const struct ieee80211_cipher_scheme * cipher_schemes;
  int txq_ac_max_pending;
};
```

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.)
extra_beacon_tailroom	tailroom to reserve in each beacon tx skb. Can be used by drivers to add extra IEs.
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.
chanctx_data_size	size (in bytes) of the drv_priv data area within struct ieee80211_chanctx_conf.
txq_data_size	size (in bytes) of the drv_priv data area within *`struct`* ieee80211_txq.
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 the hw can handle.
max_report_rates	maximum number of alternate rate retry stages the hw can report back.
max_rate_tries	maximum number of tries for each stage
max_rx_aggregation_subframes	maximum buffer size (number of sub-frames) to be used for A-MPDU block ack receiver aggregation. This is only relevant if the device has restrictions on the number of subframes, if it relies on mac80211 to do reordering it shouldn't be set.
max_tx_aggregation_subframes	maximum number of subframes in an aggregate an HT driver will transmit, used by the peer as a hint to size its reorder buffer.
offchannel_tx_hw_queue	HW queue ID to use for offchannel TX (if `IEEE80211_HW_QUEUE_CONTROL` is set)
radiotap_mcs_details	lists which MCS information can the HW reports, by default it is set to _MCS, _GI and _BW but doesn't include _FMT. Use `IEEE80211_RADIOTAP_MCS_HAVE_*` values, only adding _BW is supported today.
radiotap_vht_details	lists which VHT MCS information the HW reports, the default is _GI | _BANDWIDTH. Use the `IEEE80211_RADIOTAP_VHT_KNOWN_*` values.
netdev_features	netdev features to be set in each netdev created from this HW. Note only HW checksum features are currently compatible with mac80211. Other feature bits will be rejected.

uapsd_queues	This bitmap is included in (re)association frame to indicate for each access category if it is uAPSD trigger-enabled and delivery-enabled. Use IEEE80211_WMM_IE_STA_QOSINFO_AC_* to set this bitmap. Each bit corresponds to different AC. Value '1' in specific bit means that corresponding AC is both trigger- and delivery-enabled. '0' means neither enabled.
uapsd_max_sp_len	maximum number of total buffered frames the WMM AP may deliver to a WMM STA during any Service Period triggered by the WMM STA. Use IEEE80211_WMM_IE_STA_QOSINFO_SP_* for correct values.
n_cipher_schemes	a size of an array of cipher schemes definitions.
cipher_schemes	a pointer to an array of cipher scheme definitions supported by HW.
txq_ac_max_pending	maximum number of frames per AC pending in all txq entries for a vif.

Description

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

Name

enum ieee80211_hw_flags — hardware flags

Synopsis

```
enum ieee80211_hw_flags {
  IEEE80211_HW_HAS_RATE_CONTROL,
  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_NEED_DTIM_BEFORE_ASSOC,
  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_WANT_MONITOR_VIF,
  IEEE80211_HW_NO_AUTO_VIF,
  IEEE80211_HW_SW_CRYPTO_CONTROL,
  IEEE80211_HW_REPORTS_TX_ACK_STATUS,
  IEEE80211_HW_CONNECTION_MONITOR,
  IEEE80211_HW_QUEUE_CONTROL,
  IEEE80211_HW_SUPPORTS_PER_STA_GTK,
  IEEE80211_HW_AP_LINK_PS,
  IEEE80211_HW_TX_AMPDU_SETUP_IN_HW,
  IEEE80211_HW_SUPPORTS_RC_TABLE,
  IEEE80211_HW_P2P_DEV_ADDR_FOR_INTF,
  IEEE80211_HW_TIMING_BEACON_ONLY,
  IEEE80211_HW_SUPPORTS_HT_CCK_RATES,
  IEEE80211_HW_CHANCTX_STA_CSA,
  IEEE80211_HW_SUPPORTS_CLONED_SKBS,
  IEEE80211_SINGLE_HW_SCAN_ON_ALL_BANDS
};
```

Constants

IEEE80211_HW_HAS_RATE_CONTROL
: The hardware or firmware includes rate control, and cannot be controlled by the stack. As such, no rate control algorithm should be instantiated, and the TX rate reported to userspace will be taken from the TX status instead of the rate control algorithm. Note that this requires that the driver implement a number of callbacks so it has the correct information, it needs to have the *set_rts_threshold* callback and must look at the BSS config *use_cts_prot* for G/N protection, *use_short_slot* for slot timing in 2.4 GHz and *use_short_preamble* for preambles for CCK frames.

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_NEED_DTIM_BEFORE_ASSOC — Driver needs to get data from beacon before association (i.e. dtim_period).

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_WANT_MONITOR_VIF — The driver would like to be informed of a virtual monitor interface when monitor interfaces are the only active interfaces.

IEEE80211_HW_NO_AUTO_VIF — The driver would like for no wlanX to be created. It is expected user-space will create vifs as desired (and thus have them named as desired).

IEEE80211_HW_SW_CRYPTO_CONTROL — The driver wants to control which of the crypto algorithms can be done in software - so don't automatically try to fall back to it if hardware crypto fails, but do so only if the driver returns 1. This also forces the driver to advertise its supported cipher suites.

IEEE80211_HW_REPORTS_TX_ACK_STATUS — Hardware can provide ack status reports of Tx frames to the stack.

IEEE80211_HW_CONNECTION_MONITOR — The hardware performs its own connection monitoring, including periodic keep-alives to the AP and probing the AP on beacon loss.

IEEE80211_HW_QUEUE_CONTROL
: The driver wants to control per-interface queue mapping in order to use different queues (not just one per AC) for different virtual interfaces. See the doc section on HW queue control for more details.

IEEE80211_HW_SUPPORTS_PER_STA_GTK
: The device's crypto engine supports per-station GTKs as used by IBSS RSN or during fast transition. If the device doesn't support per-station GTKs, but can be asked not to decrypt group addressed frames, then IBSS RSN support is still possible but software crypto will be used. Advertise the wiphy flag only in that case.

IEEE80211_HW_AP_LINK_PS
: When operating in AP mode the device autonomously manages the PS status of connected stations. When this flag is set mac80211 will not trigger PS mode for connected stations based on the PM bit of incoming frames. Use `ieee80211_start_ps`/ `ieee8021_end_ps` to manually configure the PS mode of connected stations.

IEEE80211_HW_TX_AMPDU_SETUP_IN_HW
: The device handles TX A-MPDU session setup strictly in HW. mac80211 should not attempt to do this in software.

IEEE80211_HW_SUPPORTS_RC_TABLE
: The driver supports using a rate selection table provided by the rate control algorithm.

IEEE80211_HW_P2P_DEV_ADDR_FOR_INTF
: Use the P2P Device address for any P2P Interface. This will be honoured even if more than one interface is supported.

IEEE80211_HW_TIMING_BEACON_ONLY
: Use sync timing from beacon frames only, to allow getting TBTT of a DTIM beacon.

IEEE80211_HW_SUPPORTS_HT_CCK_RATES
: Hardware supports mixing HT/CCK rates and can cope with CCK rates in an aggregation session (e.g. by not using aggregation for such frames.)

IEEE80211_HW_CHANCTX_STA_CSA
: Support 802.11h based channel-switch (CSA) for a single active channel while using channel contexts. When support is not enabled the default action is to disconnect when getting the CSA frame.

IEEE80211_HW_SUPPORTS_CLONED_SKBS
: The driver will never modify the payload or tailroom of TX skbs without copying them first.

IEEE80211_SINGLE_HW_SCAN_ON_ALL_BANDS
: The hardware supports scanning on all bands in one command, mac80211 doesn't have to run separate scans per band.

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.

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

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

Name

struct ieee80211_ops — callbacks from mac80211 to the driver

Synopsis

```
struct ieee80211_ops {
  void (* tx) (struct ieee80211_hw *hw,struct ieee80211_tx_control *control,struct
  int (* start) (struct ieee80211_hw *hw);
  void (* stop) (struct ieee80211_hw *hw);
#ifdef CONFIG_PM
  int (* suspend) (struct ieee80211_hw *hw, struct cfg80211_wowlan *wowlan);
  int (* resume) (struct ieee80211_hw *hw);
  void (* set_wakeup) (struct ieee80211_hw *hw, bool enabled);
#endif
  int (* add_interface) (struct ieee80211_hw *hw,struct ieee80211_vif *vif);
  int (* change_interface) (struct ieee80211_hw *hw,struct ieee80211_vif *vif,enum
  void (* remove_interface) (struct ieee80211_hw *hw,struct ieee80211_vif *vif);
  int (* config) (struct ieee80211_hw *hw, u32 changed);
  void (* bss_info_changed) (struct ieee80211_hw *hw,struct ieee80211_vif *vif,str
  int (* start_ap) (struct ieee80211_hw *hw, struct ieee80211_vif *vif);
  void (* stop_ap) (struct ieee80211_hw *hw, struct ieee80211_vif *vif);
  u64 (* prepare_multicast) (struct ieee80211_hw *hw,struct netdev_hw_addr_list *m
  void (* configure_filter) (struct ieee80211_hw *hw,unsigned int changed_flags,un
  int (* set_tim) (struct ieee80211_hw *hw, struct ieee80211_sta *sta,bool set);
  int (* set_key) (struct ieee80211_hw *hw, enum set_key_cmd cmd,struct ieee80211_
  void (* update_tkip_key) (struct ieee80211_hw *hw,struct ieee80211_vif *vif,stru
  void (* set_rekey_data) (struct ieee80211_hw *hw,struct ieee80211_vif *vif,struc
  void (* set_default_unicast_key) (struct ieee80211_hw *hw,struct ieee80211_vif *
  int (* hw_scan) (struct ieee80211_hw *hw, struct ieee80211_vif *vif,struct ieee8
  void (* cancel_hw_scan) (struct ieee80211_hw *hw,struct ieee80211_vif *vif);
  int (* sched_scan_start) (struct ieee80211_hw *hw,struct ieee80211_vif *vif,stru
  int (* sched_scan_stop) (struct ieee80211_hw *hw,struct ieee80211_vif *vif);
  void (* sw_scan_start) (struct ieee80211_hw *hw,struct ieee80211_vif *vif,const
  void (* sw_scan_complete) (struct ieee80211_hw *hw,struct ieee80211_vif *vif);
  int (* get_stats) (struct ieee80211_hw *hw,struct ieee80211_low_level_stats *sta
  void (* get_tkip_seq) (struct ieee80211_hw *hw, u8 hw_key_idx,u32 *iv32, u16 *iv
  int (* set_frag_threshold) (struct ieee80211_hw *hw, u32 value);
  int (* set_rts_threshold) (struct ieee80211_hw *hw, u32 value);
  int (* sta_add) (struct ieee80211_hw *hw, struct ieee80211_vif *vif,struct ieee8
  int (* sta_remove) (struct ieee80211_hw *hw, struct ieee80211_vif *vif,struct ie
#ifdef CONFIG_MAC80211_DEBUGFS
  void (* sta_add_debugfs) (struct ieee80211_hw *hw,struct ieee80211_vif *vif,stru
  void (* sta_remove_debugfs) (struct ieee80211_hw *hw,struct ieee80211_vif *vif,s
#endif
  void (* sta_notify) (struct ieee80211_hw *hw, struct ieee80211_vif *vif,enum sta
  int (* sta_state) (struct ieee80211_hw *hw, struct ieee80211_vif *vif,struct iee
  void (* sta_pre_rcu_remove) (struct ieee80211_hw *hw,struct ieee80211_vif *vif,s
  void (* sta_rc_update) (struct ieee80211_hw *hw,struct ieee80211_vif *vif,struct
  void (* sta_rate_tbl_update) (struct ieee80211_hw *hw,struct ieee80211_vif *vif,
  void (* sta_statistics) (struct ieee80211_hw *hw,struct ieee80211_vif *vif,struc
  int (* conf_tx) (struct ieee80211_hw *hw,struct ieee80211_vif *vif, u16 ac,const
  u64 (* get_tsf) (struct ieee80211_hw *hw, struct ieee80211_vif *vif);
```

```
  void (* set_tsf) (struct ieee80211_hw *hw, struct ieee80211_vif *vif,u64 tsf);
  void (* reset_tsf) (struct ieee80211_hw *hw, struct ieee80211_vif *vif);
  int (* tx_last_beacon) (struct ieee80211_hw *hw);
  int (* ampdu_action) (struct ieee80211_hw *hw,struct ieee80211_vif *vif,enum iee
  int (* get_survey) (struct ieee80211_hw *hw, int idx,struct survey_info *survey)
  void (* rfkill_poll) (struct ieee80211_hw *hw);
  void (* set_coverage_class) (struct ieee80211_hw *hw, s16 coverage_class);
#ifdef CONFIG_NL80211_TESTMODE
  int (* testmode_cmd) (struct ieee80211_hw *hw, struct ieee80211_vif *vif,void *d
  int (* testmode_dump) (struct ieee80211_hw *hw, struct sk_buff *skb,struct netli
#endif
  void (* flush) (struct ieee80211_hw *hw, struct ieee80211_vif *vif,u32 queues, b
  void (* channel_switch) (struct ieee80211_hw *hw,struct ieee80211_vif *vif,struc
  int (* set_antenna) (struct ieee80211_hw *hw, u32 tx_ant, u32 rx_ant);
  int (* get_antenna) (struct ieee80211_hw *hw, u32 *tx_ant, u32 *rx_ant);
  int (* remain_on_channel) (struct ieee80211_hw *hw,struct ieee80211_vif *vif,str
  int (* cancel_remain_on_channel) (struct ieee80211_hw *hw);
  int (* set_ringparam) (struct ieee80211_hw *hw, u32 tx, u32 rx);
  void (* get_ringparam) (struct ieee80211_hw *hw,u32 *tx, u32 *tx_max, u32 *rx, u
  bool (* tx_frames_pending) (struct ieee80211_hw *hw);
  int (* set_bitrate_mask) (struct ieee80211_hw *hw, struct ieee80211_vif *vif,con
  void (* event_callback) (struct ieee80211_hw *hw,struct ieee80211_vif *vif,const
  void (* allow_buffered_frames) (struct ieee80211_hw *hw,struct ieee80211_sta *st
  void (* release_buffered_frames) (struct ieee80211_hw *hw,struct ieee80211_sta *
  int (* get_et_sset_count) (struct ieee80211_hw *hw,struct ieee80211_vif *vif, in
  void (* get_et_stats) (struct ieee80211_hw *hw,struct ieee80211_vif *vif,struct
  void (* get_et_strings) (struct ieee80211_hw *hw,struct ieee80211_vif *vif,u32 s
  void (* mgd_prepare_tx) (struct ieee80211_hw *hw,struct ieee80211_vif *vif);
  void (* mgd_protect_tdls_discover) (struct ieee80211_hw *hw,struct ieee80211_vif
  int (* add_chanctx) (struct ieee80211_hw *hw,struct ieee80211_chanctx_conf *ctx)
  void (* remove_chanctx) (struct ieee80211_hw *hw,struct ieee80211_chanctx_conf *
  void (* change_chanctx) (struct ieee80211_hw *hw,struct ieee80211_chanctx_conf *
  int (* assign_vif_chanctx) (struct ieee80211_hw *hw,struct ieee80211_vif *vif,st
  void (* unassign_vif_chanctx) (struct ieee80211_hw *hw,struct ieee80211_vif *vif
  int (* switch_vif_chanctx) (struct ieee80211_hw *hw,struct ieee80211_vif_chanctx
  void (* reconfig_complete) (struct ieee80211_hw *hw,enum ieee80211_reconfig_type
#if IS_ENABLED(CONFIG_IPV6)
  void (* ipv6_addr_change) (struct ieee80211_hw *hw,struct ieee80211_vif *vif,str
#endif
  void (* channel_switch_beacon) (struct ieee80211_hw *hw,struct ieee80211_vif *vi
  int (* pre_channel_switch) (struct ieee80211_hw *hw,struct ieee80211_vif *vif,st
  int (* post_channel_switch) (struct ieee80211_hw *hw,struct ieee80211_vif *vif);
  int (* join_ibss) (struct ieee80211_hw *hw, struct ieee80211_vif *vif);
  void (* leave_ibss) (struct ieee80211_hw *hw, struct ieee80211_vif *vif);
  u32 (* get_expected_throughput) (struct ieee80211_sta *sta);
  int (* get_txpower) (struct ieee80211_hw *hw, struct ieee80211_vif *vif,int *dbm
  int (* tdls_channel_switch) (struct ieee80211_hw *hw,struct ieee80211_vif *vif,s
  void (* tdls_cancel_channel_switch) (struct ieee80211_hw *hw,struct ieee80211_vi
  void (* tdls_recv_channel_switch) (struct ieee80211_hw *hw,struct ieee80211_vif
  void (* wake_tx_queue) (struct ieee80211_hw *hw,struct ieee80211_txq *txq);
};
```

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. Must be 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 and can sleep.

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 and can sleep.

suspend

Suspend the device; mac80211 itself will quiesce before and stop transmitting and doing any other configuration, and then ask the device to suspend. This is only invoked when WoWLAN is configured, otherwise the device is deconfigured completely and reconfigured at resume time. The driver may also impose special conditions under which it wants to use the “normal” suspend (deconfigure), say if it only supports WoWLAN when the device is associated. In this case, it must return 1 from this function.

resume

If WoWLAN was configured, this indicates that mac80211 is now resuming its operation, after this the device must be fully functional again. If this returns an error, the only way out is to also unregister the device. If it returns 1, then mac80211 will also go through the regular complete restart on resume.

set_wakeup

Enable or disable wakeup when WoWLAN configuration is modified. The reason is that `device_set_wakeup_enable` is supposed to be called when the configuration changes, not only in `suspend`.

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 and can sleep.

change_interface

Called when a netdevice changes type. This callback is optional, but only if it is supported can interface types be switched while the

	interface is UP. The callback may sleep. Note that while an interface is being switched, it will not be found by the interface iteration callbacks.
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. It can sleep.
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. The callback can sleep.
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. The callback can sleep.
start_ap	Start operation on the AP interface, this is called after all the information in bss_conf is set and beacon can be retrieved. A channel context is bound before this is called. Note that if the driver uses software scan or ROC, this (and *stop_ap*) isn't called when the AP is just “paused” for scanning/ROC, which is indicated by the beacon being disabled/enabled via *bss_info_changed*.
stop_ap	Stop operation on the AP interface.
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 and can sleep.
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 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. The callback can sleep.
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. The callback must be atomic.
set_rekey_data	If the device supports GTK rekeying, for example while the host is suspended, it can assign this callback to retrieve the

	data necessary to do GTK rekeying, this is the KEK, KCK and replay counter. After rekeying was done it should (for example during resume) notify userspace of the new replay counter using `ieee80211_gtk_rekey_notify`.
set_default_unicast_key	Set the default (unicast) key index, useful for WEP when the device sends data packets autonomously, e.g. for ARP offloading. The index can be 0-3, or -1 for unsetting it.
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. The callback can sleep.
cancel_hw_scan	Ask the low-level tp cancel the active hw scan. The driver should ask the hardware to cancel the scan (if possible), but the scan will be completed only after the driver will call `ieee80211_scan_completed`. This callback is needed for wowlan, to prevent enqueueing a new scan_work after the low-level driver was already suspended. The callback can sleep.
sched_scan_start	Ask the hardware to start scanning repeatedly at specific intervals. The driver must call the `ieee80211_sched_scan_results` function whenever it finds results. This process will continue until sched_scan_stop is called.
sched_scan_stop	Tell the hardware to stop an ongoing scheduled scan. In this case, `ieee80211_sched_scan_stopped` must not be called.
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. The mac_addr parameter allows supporting NL80211_SCAN_FLAG_RANDOM_ADDR, the driver may set the NL80211_FEATURE_SCAN_RANDOM_MAC_ADDR flag if it can use this parameter. The callback can sleep.
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. The callback can sleep.
get_stats	Return low-level statistics. Returns zero if statistics are available. The callback can sleep.
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. The callback must be atomic.

set_frag_threshold	Configuration of fragmentation threshold. Assign this if the device does fragmentation by itself; if this callback is implemented then the stack will not do fragmentation. The callback can sleep.
set_rts_threshold	Configuration of RTS threshold (if device needs it) The callback can sleep.
sta_add	Notifies low level driver about addition of an associated station, AP, IBSS/WDS/mesh peer etc. This callback can sleep.
sta_remove	Notifies low level driver about removal of an associated station, AP, IBSS/WDS/mesh peer etc. Note that after the callback returns it isn't safe to use the pointer, not even RCU protected; no RCU grace period is guaranteed between returning here and freeing the station. See *sta_pre_rcu_remove* if needed. This callback can sleep.
sta_add_debugfs	Drivers can use this callback to add debugfs files when a station is added to mac80211's station list. This callback and *sta_remove_debugfs* should be within a CONFIG_MAC80211_DEBUGFS conditional. This callback can sleep.
sta_remove_debugfs	Remove the debugfs files which were added using *sta_add_debugfs*. This callback can sleep.
sta_notify	Notifies low level driver about power state transition of an associated station, AP, IBSS/WDS/mesh peer etc. For a VIF operating in AP mode, this callback will not be called when the flag IEEE80211_HW_AP_LINK_PS is set. Must be atomic.
sta_state	Notifies low level driver about state transition of a station (which can be the AP, a client, IBSS/WDS/mesh peer etc.) This callback is mutually exclusive with *sta_add*/*sta_remove*. It must not fail for down transitions but may fail for transitions up the list of states. Also note that after the callback returns it isn't safe to use the pointer, not even RCU protected - no RCU grace period is guaranteed between returning here and freeing the station. See *sta_pre_rcu_remove* if needed. The callback can sleep.
sta_pre_rcu_remove	Notify driver about station removal before RCU synchronisation. This is useful if a driver needs to have station pointers protected using RCU, it can then use this call to clear the pointers instead of waiting for an RCU grace period to elapse in *sta_state*. The callback can sleep.
sta_rc_update	Notifies the driver of changes to the bitrates that can be used to transmit to the station. The changes are advertised with bits from enum ieee80211_rate_control_changed and the values are reflected in the station data. This callback should only be used when the driver uses hardware rate control (IEEE80211_HW_HAS_RATE_CONTROL) since otherwise the rate control algorithm is notified directly. Must be atomic.
sta_rate_tbl_update	Notifies the driver that the rate table changed. This is only used if the configured rate control algorithm actually uses the new rate table API, and is therefore optional. Must be atomic.

sta_statistics	Get statistics for this station. For example with beacon filtering, the statistics kept by mac80211 might not be accurate, so let the driver pre-fill the statistics. The driver can fill most of the values (indicating which by setting the filled bitmap), but not all of them make sense - see the source for which ones are possible. Statistics that the driver doesn't fill will be filled by mac80211. The callback can sleep.
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. The callback can sleep.
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. The callback can sleep.
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. The callback can sleep.
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. The callback can sleep.
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. The callback can sleep.
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. The *`buf_size`* parameter is only valid when the action is set to `IEEE80211_AMPDU_TX_OPERATIONAL` and indicates the peer's reorder buffer size (number of subframes) for this session -- the driver may neither send aggregates containing more subframes than this nor send aggregates in a way that lost frames would exceed the buffer size. If just limiting the aggregate size, this would be
get_survey	Return per-channel survey information
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. The callback can sleep.
set_coverage_class	Set slot time for given coverage class as specified in IEEE 802.11-2007 section 17.3.8.6 and modify ACK timeout accordingly; coverage class equals to -1 to enable ACK timeout estimation algorithm (dynack). To disable dynack set valid value for coverage class. This callback is not required and may sleep.

testmode_cmd	Implement a cfg80211 test mode command. The passed *vif* may be NULL. The callback can sleep.
testmode_dump	Implement a cfg80211 test mode dump. The callback can sleep.
flush	Flush all pending frames from the hardware queue, making sure that the hardware queues are empty. The *queues* parameter is a bitmap of queues to flush, which is useful if different virtual interfaces use different hardware queues; it may also indicate all queues. If the parameter *drop* is set to true, pending frames may be dropped. Note that vif can be NULL. The callback can sleep.
channel_switch	Drivers that need (or want) to offload the channel switch operation for CSAs received from the AP may implement this callback. They must then call ieee80211_chswitch_done to indicate completion of the channel switch.
set_antenna	Set antenna configuration (tx_ant, rx_ant) on the device. Parameters are bitmaps of allowed antennas to use for TX/ RX. Drivers may reject TX/RX mask combinations they cannot support by returning -EINVAL (also see nl80211.h *NL80211_ATTR_WIPHY_ANTENNA_TX*).
get_antenna	Get current antenna configuration from device (tx_ant, rx_ant).
remain_on_channel	Starts an off-channel period on the given channel, must call back to ieee80211_ready_on_channel when on that channel. Note that normal channel traffic is not stopped as this is intended for hw offload. Frames to transmit on the off-channel channel are transmitted normally except for the IEEE80211_TX_CTL_TX_OFFCHAN flag. When the duration (which will always be non-zero) expires, the driver must call ieee80211_remain_on_channel_expired. Note that this callback may be called while the device is in IDLE and must be accepted in this case. This callback may sleep.
cancel_remain_on_channel	Requests that an ongoing off-channel period is aborted before it expires. This callback may sleep.
set_ringparam	Set tx and rx ring sizes.
get_ringparam	Get tx and rx ring current and maximum sizes.
tx_frames_pending	Check if there is any pending frame in the hardware queues before entering power save.
set_bitrate_mask	Set a mask of rates to be used for rate control selection when transmitting a frame. Currently only legacy rates are handled. The callback can sleep.
event_callback	Notify driver about any event in mac80211. See enum ieee80211_event_type for the different types. The callback can sleep.
allow_buffered_frames	Prepare device to allow the given number of frames to go out to the given station. The frames will be sent by mac80211 via the usual

	TX path after this call. The TX information for frames released will also have the `IEEE80211_TX_CTL_NO_PS_BUFFER` flag set and the last one will also have `IEEE80211_TX_STATUS_EOSP` set. In case frames from multiple TIDs are released and the driver might reorder them between the TIDs, it must set the `IEEE80211_TX_STATUS_EOSP` flag on the last frame and clear it on all others and also handle the EOSP bit in the QoS header correctly. Alternatively, it can also call the `ieee80211_sta_eosp` function. The *`tids`* parameter is a bitmap and tells the driver which TIDs the frames will be on; it will at most have two bits set. This callback must be atomic.
release_buffered_frames	Release buffered frames according to the given parameters. In the case where the driver buffers some frames for sleeping stations mac80211 will use this callback to tell the driver to release some frames, either for PS-poll or uAPSD. Note that if the *`more_data`* parameter is `false` the driver must check if there are more frames on the given TIDs, and if there are more than the frames being released then it must still set the more-data bit in the frame. If the *`more_data`* parameter is `true`, then of course the more-data bit must always be set. The *`tids`* parameter tells the driver which TIDs to release frames from, for PS-poll it will always have only a single bit set. In the case this is used for a PS-poll initiated release, the *`num_frames`* parameter will always be 1 so code can be shared. In this case the driver must also set `IEEE80211_TX_STATUS_EOSP` flag on the TX status (and must report TX status) so that the PS-poll period is properly ended. This is used to avoid sending multiple responses for a retried PS-poll frame. In the case this is used for uAPSD, the *`num_frames`* parameter may be bigger than one, but the driver may send fewer frames (it must send at least one, however). In this case it is also responsible for setting the EOSP flag in the QoS header of the frames. Also, when the service period ends, the driver must set `IEEE80211_TX_STATUS_EOSP` on the last frame in the SP. Alternatively, it may call the function `ieee80211_sta_eosp` to inform mac80211 of the end of the SP. This callback must be atomic.
get_et_sset_count	Ethtool API to get string-set count.
get_et_stats	Ethtool API to get a set of u64 stats.
get_et_strings	Ethtool API to get a set of strings to describe stats and perhaps other supported types of ethtool data-sets.
mgd_prepare_tx	Prepare for transmitting a management frame for association before associated. In multi-channel scenarios, a virtual interface is bound to a channel before it is associated, but as it isn't associated yet it need not necessarily be given airtime, in particular since any transmission to a P2P GO needs to be synchronized against the GO's powersave state. mac80211 will call this function before transmitting a management frame prior to having successfully associated to allow the driver to give it channel time for the transmission, to get a response and to be able to synchronize with the GO. The callback will be called before each transmission and

	upon return mac80211 will transmit the frame right away. The callback is optional and can (should!) sleep.
mgd_protect_tdls_discover	Protect a TDLS discovery session. After sending a TDLS discovery-request, we expect a reply to arrive on the AP's channel. We must stay on the channel (no PSM, scan, etc.), since a TDLS setup-response is a direct packet not buffered by the AP. mac80211 will call this function just before the transmission of a TDLS discovery-request. The recommended period of protection is at least 2 * (DTIM period). The callback is optional and can sleep.
add_chanctx	Notifies device driver about new channel context creation.
remove_chanctx	Notifies device driver about channel context destruction.
change_chanctx	Notifies device driver about channel context changes that may happen when combining different virtual interfaces on the same channel context with different settings
assign_vif_chanctx	Notifies device driver about channel context being bound to vif. Possible use is for hw queue remapping.
unassign_vif_chanctx	Notifies device driver about channel context being unbound from vif.
switch_vif_chanctx	switch a number of vifs from one chanctx to another, as specified in the list of *ieee80211_vif_chanctx_switch* passed to the driver, according to the mode defined in ieee80211_chanctx_switch_mode.
reconfig_complete	Called after a call to `ieee80211_restart_hw` and during resume, when the reconfiguration has completed. This can help the driver implement the reconfiguration step (and indicate mac80211 is ready to receive frames). This callback may sleep.
ipv6_addr_change	IPv6 address assignment on the given interface changed. Currently, this is only called for managed or P2P client interfaces. This callback is optional; it must not sleep.
channel_switch_beacon	Starts a channel switch to a new channel. Beacons are modified to include CSA or ECSA IEs before calling this function. The corresponding count fields in these IEs must be decremented, and when they reach 1 the driver must call `ieee80211_csa_finish`. Drivers which use `ieee80211_beacon_get` get the csa counter decremented by mac80211, but must check if it is 1 using `ieee80211_csa_is_complete` after the beacon has been transmitted and then call `ieee80211_csa_finish`. If the CSA count starts as zero or 1, this function will not be called, since there won't be any time to beacon before the switch anyway.
pre_channel_switch	This is an optional callback that is called before a channel switch procedure is started (ie. when a STA gets a CSA or an userspace initiated channel-switch), allowing the driver to prepare for the channel switch.

post_channel_switch	This is an optional callback that is called after a channel switch procedure is completed, allowing the driver to go back to a normal configuration.
join_ibss	Join an IBSS (on an IBSS interface); this is called after all information in bss_conf is set up and the beacon can be retrieved. A channel context is bound before this is called.
leave_ibss	Leave the IBSS again.
get_expected_throughput	extract the expected throughput towards the specified station. The returned value is expressed in Kbps. It returns 0 if the RC algorithm does not have proper data to provide.
get_txpower	get current maximum tx power (in dBm) based on configuration and hardware limits.
tdls_channel_switch	Start channel-switching with a TDLS peer. The driver is responsible for continually initiating channel-switching operations and returning to the base channel for communication with the AP. The driver receives a channel-switch request template and the location of the switch-timing IE within the template as part of the invocation. The template is valid only within the call, and the driver can optionally copy the skb for further re-use.
tdls_cancel_channel_switch	Stop channel-switching with a TDLS peer. Both peers must be on the base channel when the call completes.
tdls_recv_channel_switch	a TDLS channel-switch related frame (request or response) has been received from a remote peer. The driver gets parameters parsed from the incoming frame and may use them to continue an ongoing channel-switch operation. In addition, a channel-switch response template is provided, together with the location of the switch-timing IE within the template. The skb can only be used within the function call.
wake_tx_queue	Called when new packets have been added to the queue.

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.

possible with a buf_size of 8

- TX: 1.....7 - RX: 2....7 (lost frame #1) - TX: 8..1... which is invalid since #1 was now re-transmitted well past the buffer size of 8. Correct ways to retransmit #1 would be: - TX: 1 or 18 or 81 Even “189” would be wrong since 1 could be lost again.

Returns a negative error code on failure. The callback can sleep.

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*.

Return

A pointer to the new hardware device, or NULL on error.

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.

Return

0 on success. An error code otherwise.

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.

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.

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 ps_dtim_period;
  u8 long_frame_max_tx_count;
  u8 short_frame_max_tx_count;
  struct cfg80211_chan_def chandef;
  bool radar_enabled;
  enum ieee80211_smps_mode smps_mode;
};
```

Members

flags	configuration flags defined above
power_level	requested transmit power (in dBm), backward compatibility value only that is set to the minimum of all interfaces
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
ps_dtim_period	The DTIM period of the AP we're connected to, for use in power saving. Power saving will not be enabled until a beacon has been received and the DTIM period is known.
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
chandef	the channel definition to tune to

radar_enabled	whether radar detection is enabled
smps_mode	spatial multiplexing powersave mode; note that `IEEE80211_SMPS_STATIC` is used when the device is not configured for an HT channel. Note that this is only valid if channel contexts are not used, otherwise each channel context has the number of chains listed.

Description

This struct indicates how the driver shall configure the hardware.

Name

enum ieee80211_conf_flags — configuration flags

Synopsis

```
enum ieee80211_conf_flags {
  IEEE80211_CONF_MONITOR,
  IEEE80211_CONF_PS,
  IEEE80211_CONF_IDLE,
  IEEE80211_CONF_OFFCHANNEL
};
```

Constants

IEEE80211_CONF_MONITOR	there's a monitor interface present -- use this to determine for example whether to calculate timestamps for packets or not, do not use instead of filter flags!
IEEE80211_CONF_PS	Enable 802.11 power save mode (managed mode only). This is the power save mode defined by IEEE 802.11-2007 section 11.2, meaning that the hardware still wakes up for beacons, is able to transmit frames and receive the possible acknowledgment frames. Not to be confused with hardware specific wakeup/sleep states, driver is responsible for that. See the section “Powersave support” for more.
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.
IEEE80211_CONF_OFFCHANNEL	The device is currently not on its main operating channel.

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!

Name

struct ieee80211_vif — per-interface data

Synopsis

```
struct ieee80211_vif {
  enum nl80211_iftype type;
  struct ieee80211_bss_conf bss_conf;
  u8 addr[ETH_ALEN];
  bool p2p;
  bool csa_active;
  u8 cab_queue;
  u8 hw_queue[IEEE80211_NUM_ACS];
  struct ieee80211_txq * txq;
  struct ieee80211_chanctx_conf __rcu * chanctx_conf;
  u32 driver_flags;
#ifdef CONFIG_MAC80211_DEBUGFS
  struct dentry * debugfs_dir;
#endif
  u8 drv_priv[0];
};
```

Members

type	type of this virtual interface
bss_conf	BSS configuration for this interface, either our own or the BSS we're associated to
addr[ETH_ALEN]	address of this interface
p2p	indicates whether this AP or STA interface is a p2p interface, i.e. a GO or p2p-sta respectively
csa_active	marks whether a channel switch is going on. Internally it is write-protected by sdata_lock and local->mtx so holding either is fine for read access.
cab_queue	content-after-beacon (DTIM beacon really) queue, AP mode only
hw_queue[IEEE80211_NUM_ACS]	hardware queue for each AC
txq	the multicast data TX queue (if driver uses the TXQ abstraction)
chanctx_conf	The channel context this interface is assigned to, or `NULL` when it is not assigned. This pointer is RCU-protected due to the TX path needing to access it; even though the netdev carrier will always be off when it is `NULL` there can still be races and packets could be processed after it switches back to `NULL`.
driver_flags	flags/capabilities the driver has for this interface, these need to be set (or cleared) when the interface is added or, if supported by

	the driver, the interface type is changed at runtime, mac80211 will never touch this field
debugfs_dir	debugfs dentry, can be used by drivers to create own per interface debug files. Note that it will be NULL for the virtual monitor interface (if that is requested.)
drv_priv[0]	data area for driver use, will always be aligned to sizeof(void *).

Description

Data in this structure is continually present for driver use during the life of a virtual interface.

Chapter 4. Receive and transmit processing

what should be here

TBD

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

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.

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). If the payload data is not properly aligned and the architecture doesn't support efficient unaligned operations, mac80211 will align the data.

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 between 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.

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!

functions/definitions

Name

struct ieee80211_rx_status — receive status

Synopsis

```
struct ieee80211_rx_status {
  u64 mactime;
  u32 device_timestamp;
  u32 ampdu_reference;
  u32 flag;
  u16 freq;
  u8 vht_flag;
  u8 rate_idx;
  u8 vht_nss;
  u8 rx_flags;
  u8 band;
  u8 antenna;
  s8 signal;
  u8 chains;
  s8 chain_signal[IEEE80211_MAX_CHAINS];
  u8 ampdu_delimiter_crc;
};
```

Members

mactime	value in microseconds of the 64-bit Time Synchronization Function (TSF) timer when the first data symbol (MPDU) arrived at the hardware.
device_timestamp	arbitrary timestamp for the device, mac80211 doesn't use it but can store it and pass it back to the driver for synchronisation
ampdu_reference	A-MPDU reference number, must be a different value for each A-MPDU but the same for each subframe within one A-MPDU
flag	RX_FLAG_*
freq	frequency the radio was tuned to when receiving this frame, in MHz
vht_flag	RX_VHT_FLAG_*
rate_idx	index of data rate into band's supported rates or MCS index if HT or VHT is used (RX_FLAG_HT/RX_FLAG_VHT)
vht_nss	number of streams (VHT only)
rx_flags	internal RX flags for mac80211
band	the active band when this frame was received
antenna	antenna used
signal	signal strength when receiving this frame, either in dBm, in dB or unspecified depending on the hardware capabilities flags *IEEE80211_HW_SIGNAL_**

chains	bitmask of receive chains for which separate signal strength values were filled.
chain_signal[IEEE80211_MAX_CHAINS]	per-chain signal strength, in dBm (unlike *signal*, doesn't support dB or unspecified units)
ampdu_delimiter_crc	A-MPDU delimiter CRC

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).

Name

enum mac80211_rx_flags — receive flags

Synopsis

```
enum mac80211_rx_flags {
  RX_FLAG_MMIC_ERROR,
  RX_FLAG_DECRYPTED,
  RX_FLAG_MMIC_STRIPPED,
  RX_FLAG_IV_STRIPPED,
  RX_FLAG_FAILED_FCS_CRC,
  RX_FLAG_FAILED_PLCP_CRC,
  RX_FLAG_MACTIME_START,
  RX_FLAG_SHORTPRE,
  RX_FLAG_HT,
  RX_FLAG_40MHZ,
  RX_FLAG_SHORT_GI,
  RX_FLAG_NO_SIGNAL_VAL,
  RX_FLAG_HT_GF,
  RX_FLAG_AMPDU_DETAILS,
  RX_FLAG_AMPDU_REPORT_ZEROLEN,
  RX_FLAG_AMPDU_IS_ZEROLEN,
  RX_FLAG_AMPDU_LAST_KNOWN,
  RX_FLAG_AMPDU_IS_LAST,
  RX_FLAG_AMPDU_DELIM_CRC_ERROR,
  RX_FLAG_AMPDU_DELIM_CRC_KNOWN,
  RX_FLAG_MACTIME_END,
  RX_FLAG_VHT,
  RX_FLAG_LDPC,
  RX_FLAG_STBC_MASK,
  RX_FLAG_10MHZ,
  RX_FLAG_5MHZ,
  RX_FLAG_AMSDU_MORE,
  RX_FLAG_RADIOTAP_VENDOR_DATA
};
```

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_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_MACTIME_START	The timestamp passed in the RX status (*mactime* field) is valid and contains the time the first symbol of the MPDU was received. This is useful in monitor mode and for proper 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
RX_FLAG_NO_SIGNAL_VAL	The signal strength value is not present. Valid only for data frames (mainly A-MPDU)
RX_FLAG_HT_GF	This frame was received in a HT-greenfield transmission, if the driver fills this value it should add `IEEE80211_RADIOTAP_MCS_HAVE_FMT` to hw.radiotap_mcs_details to advertise that fact
RX_FLAG_AMPDU_DETAILS	A-MPDU details are known, in particular the reference number (*ampdu_reference*) must be populated and be a distinct number for each A-MPDU
RX_FLAG_AMPDU_REPORT_ZEROLEN	Driver reports 0-length subframes
RX_FLAG_AMPDU_IS_ZEROLEN	This is a zero-length subframe, for monitoring purposes only
RX_FLAG_AMPDU_LAST_KNOWN	last subframe is known, should be set on all subframes of a single A-MPDU
RX_FLAG_AMPDU_IS_LAST	this subframe is the last subframe of the A-MPDU
RX_FLAG_AMPDU_DELIM_CRC_ERROR	A delimiter CRC error has been detected on this subframe
RX_FLAG_AMPDU_DELIM_CRC_KNOWN	The delimiter CRC field is known (the CRC is stored in the *ampdu_delimiter_crc* field)
RX_FLAG_MACTIME_END	The timestamp passed in the RX status (*mactime* field) is valid and contains the time the last symbol of the MPDU (including FCS) was received.
RX_FLAG_VHT	VHT MCS was used and rate_index is MCS index
RX_FLAG_LDPC	LDPC was used
RX_FLAG_STBC_MASK	STBC 2 bit bitmask. 1 - Nss=1, 2 - Nss=2, 3 - Nss=3
RX_FLAG_10MHZ	10 MHz (half channel) was used
RX_FLAG_5MHZ	5 MHz (quarter channel) was used
RX_FLAG_AMSDU_MORE	Some drivers may prefer to report separate A-MSDU subframes instead of a one huge frame for performance reasons. All, but the last MSDU from an A-MSDU should have this flag set. E.g. if an A-MSDU has 3 frames, the first 2 must have the flag set, while the 3rd (last) one must not have this flag set. The flag is used to deal

with retransmission/duplication recovery properly since A-MSDU subframes share the same sequence number. Reported subframes can be either regular MSDU or singly A-MSDUs. Subframes must not be interleaved with other frames.

RX_FLAG_RADIOTAP_VENDOR_DATA This frame contains vendor-specific radiotap data in the skb->data (before the frame) as described by the struct ieee80211_vendor_radiotap.

Description

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

Name

enum mac80211_tx_info_flags — flags to describe transmission information/status

Synopsis

```
enum mac80211_tx_info_flags {
  IEEE80211_TX_CTL_REQ_TX_STATUS,
  IEEE80211_TX_CTL_ASSIGN_SEQ,
  IEEE80211_TX_CTL_NO_ACK,
  IEEE80211_TX_CTL_CLEAR_PS_FILT,
  IEEE80211_TX_CTL_FIRST_FRAGMENT,
  IEEE80211_TX_CTL_SEND_AFTER_DTIM,
  IEEE80211_TX_CTL_AMPDU,
  IEEE80211_TX_CTL_INJECTED,
  IEEE80211_TX_STAT_TX_FILTERED,
  IEEE80211_TX_STAT_ACK,
  IEEE80211_TX_STAT_AMPDU,
  IEEE80211_TX_STAT_AMPDU_NO_BACK,
  IEEE80211_TX_CTL_RATE_CTRL_PROBE,
  IEEE80211_TX_INTFL_OFFCHAN_TX_OK,
  IEEE80211_TX_INTFL_NEED_TXPROCESSING,
  IEEE80211_TX_INTFL_RETRIED,
  IEEE80211_TX_INTFL_DONT_ENCRYPT,
  IEEE80211_TX_CTL_NO_PS_BUFFER,
  IEEE80211_TX_CTL_MORE_FRAMES,
  IEEE80211_TX_INTFL_RETRANSMISSION,
  IEEE80211_TX_INTFL_MLME_CONN_TX,
  IEEE80211_TX_INTFL_NL80211_FRAME_TX,
  IEEE80211_TX_CTL_LDPC,
  IEEE80211_TX_CTL_STBC,
  IEEE80211_TX_CTL_TX_OFFCHAN,
  IEEE80211_TX_INTFL_TKIP_MIC_FAILURE,
  IEEE80211_TX_CTL_NO_CCK_RATE,
  IEEE80211_TX_STATUS_EOSP,
  IEEE80211_TX_CTL_USE_MINRATE,
  IEEE80211_TX_CTL_DONTFRAG,
  IEEE80211_TX_STAT_NOACK_TRANSMITTED
};
```

Constants

IEEE80211_TX_CTL_REQ_TX_STATUS
: require TX status callback for this frame.

IEEE80211_TX_CTL_ASSIGN_SEQ
: The driver has to assign a sequence number to this frame, taking care of not overwriting the fragment number and increasing the sequence number only when the IEEE80211_TX_CTL_FIRST_FRAGMENT flag is set. mac80211 will properly assign sequence numbers to QoS-data frames but cannot do so correctly for non-QoS-data and management frames because beacons need them from that counter as well and mac80211 cannot guarantee proper sequencing. If this flag is set, the driver should instruct the hardware to assign a sequence number to

	the frame or assign one itself. Cf. IEEE 802.11-2007 7.1.3.4.1 paragraph 3. This flag will always be set for beacons and always be clear for frames without a sequence number field.
IEEE80211_TX_CTL_NO_ACK	tell the low level not to wait for an ack
IEEE80211_TX_CTL_CLEAR_PS_FILT	clear powersave filter for destination station
IEEE80211_TX_CTL_FIRST_FRAGMENT	this is a first fragment of the frame
IEEE80211_TX_CTL_SEND_AFTER_DTIM	send this frame after DTIM beacon
IEEE80211_TX_CTL_AMPDU	this frame should be sent as part of an A-MPDU
IEEE80211_TX_CTL_INJECTED	Frame was injected, internal to mac80211.
IEEE80211_TX_STAT_TX_FILTERED	The frame was not transmitted because the destination STA was in powersave mode. Note that to avoid race conditions, the filter must be set by the hardware or firmware upon receiving a frame that indicates that the station went to sleep (must be done on device to filter frames already on the queue) and may only be unset after mac80211 gives the OK for that by setting the IEEE80211_TX_CTL_CLEAR_PS_FILT (see above), since only then is it guaranteed that no more frames are in the hardware queue.
IEEE80211_TX_STAT_ACK	Frame was acknowledged
IEEE80211_TX_STAT_AMPDU	The frame was aggregated, so status is for the whole aggregation.
IEEE80211_TX_STAT_AMPDU_NO_BACK	no block ack was returned, so consider using block ack request (BAR).
IEEE80211_TX_CTL_RATE_CTRL_PROBE	internal to mac80211, can be set by rate control algorithms to indicate probe rate, will be cleared for fragmented frames (except on the last fragment)
IEEE80211_TX_INTFL_OFFCHAN_TX_OK	Internal to mac80211. Used to indicate that a frame can be transmitted while the queues are stopped for off-channel operation.
IEEE80211_TX_INTFL_NEED_TXPROCESSING	completely internal to mac80211, used to indicate that a pending frame requires TX processing before it can be sent out.
IEEE80211_TX_INTFL_RETRIED	completely internal to mac80211, used to indicate that a frame was already retried due to PS
IEEE80211_TX_INTFL_DONT_ENCRYPT	completely internal to mac80211, used to indicate frame should not be encrypted
IEEE80211_TX_CTL_NO_PS_BUFFER	This frame is a response to a poll frame (PS-Poll or uAPSD) or a non-bufferable MMPDU and must be sent although the station is in powersave mode.
IEEE80211_TX_CTL_MORE_FRAMES	More frames will be passed to the transmit function after the current frame, this can be used by drivers to kick the DMA queue only if unset or when the queue gets full.

IEEE80211_TX_INTFL_RETRANSMISSION	This frame is being retransmitted after TX status because the destination was asleep, it must not be modified again (no seqno assignment, crypto, etc.)
IEEE80211_TX_INTFL_MLME_CONN_TX	This frame was transmitted by the MLME code for connection establishment, this indicates that its status should kick the MLME state machine.
IEEE80211_TX_INTFL_NL80211_FRAME_TX	Frame was requested through nl80211 MLME command (internal to mac80211 to figure out whether to send TX status to user space)
IEEE80211_TX_CTL_LDPC	tells the driver to use LDPC for this frame
IEEE80211_TX_CTL_STBC	Enables Space-Time Block Coding (STBC) for this frame and selects the maximum number of streams that it can use.
IEEE80211_TX_CTL_TX_OFFCHAN	Marks this packet to be transmitted on the off-channel channel when a remain-on-channel offload is done in hardware -- normal packets still flow and are expected to be handled properly by the device.
IEEE80211_TX_INTFL_TKIP_MIC_FAILURE	Marks this packet to be used for TKIP testing. It will be sent out with incorrect Michael MIC key to allow TKIP countermeasures to be tested.
IEEE80211_TX_CTL_NO_CCK_RATE	This frame will be sent at non CCK rate. This flag is actually used for management frame especially for P2P frames not being sent at CCK rate in 2GHz band.
IEEE80211_TX_STATUS_EOSP	This packet marks the end of service period, when its status is reported the service period ends. For frames in an SP that mac80211 transmits, it is already set; for driver frames the driver may set this flag. It is also used to do the same for PS-Poll responses.
IEEE80211_TX_CTL_USE_MINRATE	This frame will be sent at lowest rate. This flag is used to send nullfunc frame at minimum rate when the nullfunc is used for connection monitoring purpose.
IEEE80211_TX_CTL_DONTFRAG	Don't fragment this packet even if it would be fragmented by size (this is optional, only used for monitor injection).
IEEE80211_TX_STAT_NOACK_TRANSMITTED	A frame that was marked with IEEE80211_TX_CTL_NO_ACK has been successfully transmitted without any errors (like issues specific to the driver/HW). This flag must not be set for frames that don't request no-ack behaviour with IEEE80211_TX_CTL_NO_ACK.

Description

These flags are used with the *flags* member of ieee80211_tx_info.

Note

If you have to add new flags to the enumeration, then don't forget to update `IEEE80211_TX_TEMPORARY_FLAGS` when necessary.

Name

enum mac80211_tx_control_flags — flags to describe transmit control

Synopsis

```
enum mac80211_tx_control_flags {
  IEEE80211_TX_CTRL_PORT_CTRL_PROTO,
  IEEE80211_TX_CTRL_PS_RESPONSE
};
```

Constants

IEEE80211_TX_CTRL_PORT_CTRL_PROTO
: this frame is a port control protocol frame (e.g. EAP)

IEEE80211_TX_CTRL_PS_RESPONSE
: This frame is a response to a poll frame (PS-Poll or uAPSD).

Description

These flags are used in tx_info->control.flags.

Name

enum mac80211_rate_control_flags — per-rate flags set by the Rate Control algorithm.

Synopsis

```
enum mac80211_rate_control_flags {
  IEEE80211_TX_RC_USE_RTS_CTS,
  IEEE80211_TX_RC_USE_CTS_PROTECT,
  IEEE80211_TX_RC_USE_SHORT_PREAMBLE,
  IEEE80211_TX_RC_MCS,
  IEEE80211_TX_RC_GREEN_FIELD,
  IEEE80211_TX_RC_40_MHZ_WIDTH,
  IEEE80211_TX_RC_DUP_DATA,
  IEEE80211_TX_RC_SHORT_GI,
  IEEE80211_TX_RC_VHT_MCS,
  IEEE80211_TX_RC_80_MHZ_WIDTH,
  IEEE80211_TX_RC_160_MHZ_WIDTH
};
```

Constants

IEEE80211_TX_RC_USE_RTS_CTS — Use RTS/CTS exchange for this rate.

IEEE80211_TX_RC_USE_CTS_PROTECT — CTS-to-self protection is required. This is set if the current BSS requires ERP protection.

IEEE80211_TX_RC_USE_SHORT_PREAMBLE — Use short preamble.

IEEE80211_TX_RC_MCS — HT rate.

IEEE80211_TX_RC_GREEN_FIELD — Indicates whether this rate should be used in Greenfield mode.

IEEE80211_TX_RC_40_MHZ_WIDTH — Indicates if the Channel Width should be 40 MHz.

IEEE80211_TX_RC_DUP_DATA — The frame should be transmitted on both of the adjacent 20 MHz channels, if the current channel type is NL80211_CHAN_HT40MINUS or NL80211_CHAN_HT40PLUS.

IEEE80211_TX_RC_SHORT_GI — Short Guard interval should be used for this rate.

IEEE80211_TX_RC_VHT_MCS — VHT MCS rate, in this case the idx field is split into a higher 4 bits (Nss) and lower 4 bits (MCS number)

IEEE80211_TX_RC_80_MHZ_WIDTH — Indicates 80 MHz transmission

IEEE80211_TX_RC_160_MHZ_WIDTH — Indicates 160 MHz transmission (80+80 isn't supported yet)

Description

These flags are set by the Rate control algorithm for each rate during tx, in the *flags* member of struct ieee80211_tx_rate.

Name

struct ieee80211_tx_rate — rate selection/status

Synopsis

```
struct ieee80211_tx_rate {
  s8 idx;
  u16 count:5;
  u16 flags:11;
};
```

Members

idx rate index to attempt to send with

count number of tries in this rate before going to the next rate

flags rate control flags (enum mac80211_rate_control_flags)

Description

A value of -1 for *idx* indicates an invalid rate and, if used in an array of retry rates, that no more rates should be tried.

When used for transmit status reporting, the driver should always report the rate along with the flags it used.

struct ieee80211_tx_info contains an array of these structs in the control information, and it will be filled by the rate control algorithm according to what should be sent. For example, if this array contains, in the format { <idx>, <count> } the information { 3, 2 }, { 2, 2 }, { 1, 4 }, { -1, 0 }, { -1, 0 } then this means that the frame should be transmitted up to twice at rate 3, up to twice at rate 2, and up to four times at rate 1 if it doesn't get acknowledged. Say it gets acknowledged by the peer after the fifth attempt, the status information should then contain { 3, 2 }, { 2, 2 }, { 1, 1 }, { -1, 0 } ... since it was transmitted twice at rate 3, twice at rate 2 and once at rate 1 after which we received an acknowledgement.

Name

struct ieee80211_tx_info — skb transmit information

Synopsis

```
struct ieee80211_tx_info {
  u32 flags;
  u8 band;
  u8 hw_queue;
  u16 ack_frame_id;
  union {unnamed_union};
};
```

Members

flags	transmit info flags, defined above
band	the band to transmit on (use for checking for races)
hw_queue	HW queue to put the frame on, `skb_get_queue_mapping` gives the AC
ack_frame_id	internal frame ID for TX status, used internally
{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

Name

ieee80211_tx_info_clear_status — clear TX status

Synopsis

```
void ieee80211_tx_info_clear_status (struct ieee80211_tx_info * info);
```

Arguments

info The struct ieee80211_tx_info to be cleared.

Description

When the driver passes an skb back to mac80211, it must report a number of things in TX status. This function clears everything in the TX status but the rate control information (it does clear the count since you need to fill that in anyway).

NOTE

You can only use this function if you do NOT use info->driver_data! Use info->rate_driver_data instead if you need only the less space that allows.

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. In case of a paged *skb* is used, the driver is recommended to put the ieee80211 header of the frame on the linear part of the *skb* to avoid memory allocation and/or memcpy by the stack.

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, `ieee80211_rx_ni` and `ieee80211_rx_irqsafe` may not be mixed for a single hardware. Must not run concurrently with `ieee80211_tx_status` or `ieee80211_tx_status_ni`.

In process context use instead `ieee80211_rx_ni`.

Name

ieee80211_rx_ni — receive frame (in process context)

Synopsis

```
void ieee80211_rx_ni (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 process context (internally disables bottom halves).

Calls to this function, `ieee80211_rx` and `ieee80211_rx_irqsafe` may not be mixed for a single hardware. Must not run concurrently with `ieee80211_tx_status` or `ieee80211_tx_status_ni`.

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, `ieee80211_rx` or `ieee80211_rx_ni` may not be mixed for a single hardware.Must not run concurrently with `ieee80211_tx_status` or `ieee80211_tx_status_ni`.

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, `ieee80211_tx_status_ni` and `ieee80211_tx_status_irqsafe` may not be mixed for a single hardware. Must not run concurrently with `ieee80211_rx` or `ieee80211_rx_ni`.

Name

ieee80211_tx_status_ni — transmit status callback (in process context)

Synopsis

```
void ieee80211_tx_status_ni (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 process context.

Calls to this function, `ieee80211_tx_status` and `ieee80211_tx_status_irqsafe` may not be mixed for a single hardware.

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, `ieee80211_tx_status` and `ieee80211_tx_status_ni` may not be mixed for a single hardware.

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 the add_interface callback.
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.

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 the add_interface callback.
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.

Return

The duration.

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 the add_interface callback.
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.

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 the add_interface callback.
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.

Return

The duration.

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, enum ieee80211_band band, size_t frame_len,
struct ieee80211_rate * rate);
```

Arguments

hw	pointer obtained from `ieee80211_alloc_hw`.
vif	struct ieee80211_vif pointer from the add_interface callback.
band	the band to calculate the frame duration on
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).

Return

The duration.

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.

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.

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.

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.

Name

ieee80211_queue_stopped — test status of the queue

Synopsis

```
int ieee80211_queue_stopped (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.

Return

`true` if the queue is stopped. `false` otherwise.

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.

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,
  FIF_PROBE_REQ
};
```

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.
FIF_PROBE_REQ	pass probe request frames

HW queue control

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.

Chapter 6. The mac80211 workqueue

mac80211 provides its own workqueue for drivers and internal mac80211 use. The workqueue is a single threaded workqueue and can only be accessed by helpers for sanity checking. Drivers must ensure all work added onto the mac80211 workqueue should be cancelled on the driver `stop` callback.

mac80211 will flushed the workqueue upon interface removal and during suspend.

All work performed on the mac80211 workqueue must not acquire the RTNL lock.

Name

ieee80211_queue_work — add work onto the mac80211 workqueue

Synopsis

```
void ieee80211_queue_work (struct ieee80211_hw * hw, struct work_struct
* work);
```

Arguments

hw the hardware struct for the interface we are adding work for

work the work we want to add onto the mac80211 workqueue

Description

Drivers and mac80211 use this to add work onto the mac80211 workqueue. This helper ensures drivers are not queueing work when they should not be.

Name

ieee80211_queue_delayed_work — add work onto the mac80211 workqueue

Synopsis

```
void ieee80211_queue_delayed_work (struct ieee80211_hw * hw, struct
delayed_work * dwork, unsigned long delay);
```

Arguments

hw the hardware struct for the interface we are adding work for

dwork delayable work to queue onto the mac80211 workqueue

delay number of jiffies to wait before queueing

Description

Drivers and mac80211 use this to queue delayed work onto the mac80211 workqueue.

Part II. Advanced driver interface

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.

Table of Contents

7. LED support 72
ieee80211_get_tx_led_name 73
ieee80211_get_rx_led_name 74
ieee80211_get_assoc_led_name 75
ieee80211_get_radio_led_name 76
struct ieee80211_tpt_blink 77
enum ieee80211_tpt_led_trigger_flags 78
ieee80211_create_tpt_led_trigger 79
8. Hardware crypto acceleration 80
enum set_key_cmd 81
struct ieee80211_key_conf 82
enum ieee80211_key_flags 83
ieee80211_get_tkip_p1k 85
ieee80211_get_tkip_p1k_iv 86
ieee80211_get_tkip_p2k 87
9. Powersave support 88
10. Beacon filter support 89
ieee80211_beacon_loss 90
11. Multiple queues and QoS support 91
struct ieee80211_tx_queue_params 92
12. Access point mode support 93
support for powersaving clients 93
13. Supporting multiple virtual interfaces 103
ieee80211_iterate_active_interfaces 104
ieee80211_iterate_active_interfaces_atomic 105
14. Station handling 106
struct ieee80211_sta 107
enum sta_notify_cmd 109
ieee80211_find_sta 110
ieee80211_find_sta_by_ifaddr 111
15. Hardware scan offload 112
ieee80211_scan_completed 113
16. Aggregation 114
TX A-MPDU aggregation 114
RX A-MPDU aggregation 114
17. Spatial Multiplexing Powersave (SMPS) 116
ieee80211_request_smps 117
enum ieee80211_smps_mode 118

Chapter 7. LED support

Mac80211 supports various ways of blinking LEDs. Wherever possible, device LEDs should be exposed as LED class devices and hooked up to the appropriate trigger, which will then be triggered appropriately by mac80211.

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.

Return

The name of the LED trigger. `NULL` if not configured for LEDs.

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.

Return

The name of the LED trigger. `NULL` if not configured for LEDs.

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.

Return

The name of the LED trigger. `NULL` if not configured for LEDs.

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.

Return

The name of the LED trigger. `NULL` if not configured for LEDs.

Name

struct ieee80211_tpt_blink — throughput blink description

Synopsis

```
struct ieee80211_tpt_blink {
  int throughput;
  int blink_time;
};
```

Members

throughput	throughput in Kbit/sec
blink_time	blink time in milliseconds (full cycle, ie. one off + one on period)

Name

enum ieee80211_tpt_led_trigger_flags — throughput trigger flags

Synopsis

```
enum ieee80211_tpt_led_trigger_flags {
  IEEE80211_TPT_LEDTRIG_FL_RADIO,
  IEEE80211_TPT_LEDTRIG_FL_WORK,
  IEEE80211_TPT_LEDTRIG_FL_CONNECTED
};
```

Constants

IEEE80211_TPT_LEDTRIG_FL_RADIO
: enable blinking with radio

IEEE80211_TPT_LEDTRIG_FL_WORK
: enable blinking when working

IEEE80211_TPT_LEDTRIG_FL_CONNECTED
: enable blinking when at least one interface is connected in some way, including being an AP

Name

ieee80211_create_tpt_led_trigger — create throughput LED trigger

Synopsis

```
char * ieee80211_create_tpt_led_trigger (struct ieee80211_hw * hw,
unsigned int flags, const struct ieee80211_tpt_blink * blink_table,
unsigned int blink_table_len);
```

Arguments

hw	the hardware to create the trigger for
flags	trigger flags, see enum ieee80211_tpt_led_trigger_flags
blink_table	the blink table -- needs to be ordered by throughput
blink_table_len	size of the blink table

Return

NULL (in case of error, or if no LED triggers are configured) or the name of the new trigger.

Note

This function must be called before ieee80211_register_hw.

Chapter 8. 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.

Note that in the case that the *`IEEE80211_HW_SW_CRYPTO_CONTROL`* flag is set, mac80211 will not automatically fall back to software crypto if enabling hardware crypto failed. The `set_key` call may also return the value 1 to permit this specific key/algorithm to be done in software.

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 every time 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.

The `set_default_unicast_key` call updates the default WEP key index configured to the hardware for WEP encryption type. This is required for devices that support offload of data packets (e.g. ARP responses).

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.

Name

struct ieee80211_key_conf — key information

Synopsis

```
struct ieee80211_key_conf {
  u32 cipher;
  u8 icv_len;
  u8 iv_len;
  u8 hw_key_idx;
  u8 flags;
  s8 keyidx;
  u8 keylen;
  u8 key[0];
};
```

Members

cipher	The key's cipher suite selector.
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)

Name

enum ieee80211_key_flags — key flags

Synopsis

```
enum ieee80211_key_flags {
  IEEE80211_KEY_FLAG_GENERATE_IV_MGMT,
  IEEE80211_KEY_FLAG_GENERATE_IV,
  IEEE80211_KEY_FLAG_GENERATE_MMIC,
  IEEE80211_KEY_FLAG_PAIRWISE,
  IEEE80211_KEY_FLAG_SW_MGMT_TX,
  IEEE80211_KEY_FLAG_PUT_IV_SPACE,
  IEEE80211_KEY_FLAG_RX_MGMT,
  IEEE80211_KEY_FLAG_RESERVE_TAILROOM
};
```

Constants

IEEE80211_KEY_FLAG_GENERATE_IV_MGMT
: This flag should be set by the driver for a CCMP/GCMP key to indicate that is requires IV generation only for managment frames (MFP).

IEEE80211_KEY_FLAG_GENERATE_IV
: This flag should be set by the driver to indicate that it requires IV generation for this particular key. Setting this flag does not necessarily mean that SKBs will have sufficient tailroom for ICV or MIC.

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 then a shared key.

IEEE80211_KEY_FLAG_SW_MGMT_TX
: This flag should be set by the driver for a CCMP/GCMP key if it requires CCMP/GCMP encryption of management frames (MFP) to be done in software.

IEEE80211_KEY_FLAG_PUT_IV_SPACE
: This flag should be set by the driver if space should be prepared for the IV, but the IV itself should not be generated. Do not set together with *IEEE80211_KEY_FLAG_GENERATE_IV* on the same key. Setting this flag does not necessarily mean that SKBs will have sufficient tailroom for ICV or MIC.

IEEE80211_KEY_FLAG_RX_MGMT
: This key will be used to decrypt received management frames. The flag can help drivers that have a hardware crypto implementation that doesn't deal with management frames properly by allowing them to not upload the keys to hardware and fall back to software crypto. Note that this flag deals only with RX, if your crypto engine can't deal with TX you can also set the `IEEE80211_KEY_FLAG_SW_MGMT_TX` flag to encrypt such frames in SW.

IEEE80211_KEY_FLAG_RESERVE_TAILROOM This flag should be set by the driver for a key to indicate that sufficient tailroom must always be reserved for ICV or MIC, even when HW encryption is enabled.

Description

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

Name

ieee80211_get_tkip_p1k — get a TKIP phase 1 key

Synopsis

```
void ieee80211_get_tkip_p1k (struct ieee80211_key_conf * keyconf, struct
sk_buff * skb, u16 * p1k);
```

Arguments

keyconf	the parameter passed with the set key
skb	the packet to take the IV32 value from that will be encrypted with this P1K
p1k	a buffer to which the key will be written, as 5 u16 values

Description

This function returns the TKIP phase 1 key for the IV32 taken from the given packet.

Name

ieee80211_get_tkip_p1k_iv — get a TKIP phase 1 key for IV32

Synopsis

```
void ieee80211_get_tkip_p1k_iv (struct ieee80211_key_conf * keyconf,
u32 iv32, u16 * p1k);
```

Arguments

keyconf the parameter passed with the set key

iv32 IV32 to get the P1K for

p1k a buffer to which the key will be written, as 5 u16 values

Description

This function returns the TKIP phase 1 key for the given IV32.

Name

ieee80211_get_tkip_p2k — get a TKIP phase 2 key

Synopsis

```
void ieee80211_get_tkip_p2k (struct ieee80211_key_conf * keyconf, struct
sk_buff * skb, u8 * p2k);
```

Arguments

keyconf	the parameter passed with the set key
skb	the packet to take the IV32/IV16 values from that will be encrypted with this key
p2k	a buffer to which the key will be written, 16 bytes

Description

This function computes the TKIP RC4 key for the IV values in the packet.

Chapter 9. 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 with the `IEEE80211_CONF_PS` flag depending on the association status. The hardware must take care of sending nullfunc frames when necessary, i.e. when entering and leaving powersave mode. The hardware 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.

`IEEE80211_CONF_PS` flag enabled means that the powersave mode defined in IEEE 802.11-2007 section 11.2 is enabled. This is not to be confused with hardware wakeup and sleep states. Driver is responsible for waking up the hardware before issuing commands to the hardware and putting it back to sleep at appropriate times.

When PS is enabled, hardware needs to wakeup for beacons and receive the buffered multicast/broadcast frames after the beacon. Also it must be possible to send frames and receive the acknowledment frame.

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 to do that.

Dynamic powersave is an extension to normal powersave 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. It's compromise of getting good enough latency when there's data traffic and still saving significantly power in idle periods.

Dynamic powersave is simply supported by mac80211 enabling and disabling PS based on traffic. Driver needs to only set `IEEE80211_HW_SUPPORTS_PS` flag and mac80211 will handle everything automatically. Additionally, hardware having support for the dynamic PS feature may set the `IEEE80211_HW_SUPPORTS_DYNAMIC_PS` flag to indicate that it can support dynamic PS mode itself. The driver needs to look at the *`dynamic_ps_timeout`* hardware configuration value and use it that value whenever `IEEE80211_CONF_PS` is set. In this case mac80211 will disable dynamic PS feature in stack and will just keep `IEEE80211_CONF_PS` enabled whenever user has enabled powersave.

Driver informs U-APSD client support by enabling `IEEE80211_VIF_SUPPORTS_UAPSD` flag. The mode is configured through the uapsd parameter in `conf_tx` operation. Hardware needs to send the QoS Nullfunc frames and stay awake until the service period has ended. To utilize U-APSD, dynamic powersave is disabled for voip AC and all frames from that AC are transmitted with powersave enabled.

Note: U-APSD client mode is not yet supported with `IEEE80211_HW_PS_NULLFUNC_STACK`.

Chapter 10. 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_VIF_BEACON_FILTER` interface 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.

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 the add_interface callback.

Description

When beacon filtering is enabled with IEEE80211_VIF_BEACON_FILTER and IEEE80211_CONF_PS is set, the driver needs to inform whenever the hardware is not receiving beacons with this function.

Chapter 11. Multiple queues and QoS support

TBD

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;
  bool acm;
  bool uapsd;
};
```

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]

acm — is mandatory admission control required for the access category

uapsd — is U-APSD mode enabled for the queue

Description

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

Chapter 12. 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.

support for powersaving clients

In order to implement AP and P2P GO modes, mac80211 has support for client powersaving, both “legacy” PS (PS-Poll/null data) and uAPSD. There currently is no support for sAPSD.

There is one assumption that mac80211 makes, namely that a client will not poll with PS-Poll and trigger with uAPSD at the same time. Both are supported, and both can be used by the same client, but they can't be used concurrently by the same client. This simplifies the driver code.

The first thing to keep in mind is that there is a flag for complete driver implementation: `IEEE80211_HW_AP_LINK_PS`. If this flag is set, mac80211 expects the driver to handle most of the state machine for powersaving clients and will ignore the PM bit in incoming frames. Drivers then use `ieee80211_sta_ps_transition` to inform mac80211 of stations' powersave transitions. In this mode, mac80211 also doesn't handle PS-Poll/uAPSD.

In the mode without `IEEE80211_HW_AP_LINK_PS`, mac80211 will check the PM bit in incoming frames for client powersave transitions. When a station goes to sleep, we will stop transmitting to it. There is, however, a race condition: a station might go to sleep while there is data buffered on hardware queues. If the device has support for this it will reject frames, and the driver should give the frames back to mac80211 with the `IEEE80211_TX_STAT_TX_FILTERED` flag set which will cause mac80211 to retry the frame when the station wakes up. The driver is also notified of powersave transitions by calling its *`sta_notify`* callback.

When the station is asleep, it has three choices: it can wake up, it can PS-Poll, or it can possibly start a uAPSD service period. Waking up is implemented by simply transmitting all buffered (and filtered) frames to the station. This is the easiest case. When the station sends a PS-Poll or a uAPSD trigger frame, mac80211 will inform the driver of this with the *`allow_buffered_frames`* callback; this callback is optional. mac80211 will then transmit the frames as usual and set the `IEEE80211_TX_CTL_NO_PS_BUFFER` on each frame. The last frame in the service period (or the only response to a PS-Poll) also has `IEEE80211_TX_STATUS_EOSP` set to indicate that it ends the service period; as this frame must have TX status report it also sets `IEEE80211_TX_CTL_REQ_TX_STATUS`. When TX status is reported for this frame, the service period is marked has having ended and a new one can be started by the peer.

Additionally, non-bufferable MMPDUs can also be transmitted by mac80211 with the `IEEE80211_TX_CTL_NO_PS_BUFFER` set in them.

Another race condition can happen on some devices like iwlwifi when there are frames queued for the station and it wakes up or polls; the frames that are already queued could end up being transmitted first instead, causing reordering and/or wrong processing of the EOSP. The cause is that allowing frames to be transmitted to a certain station is out-of-band communication to the device. To allow this problem to be solved, the driver can call `ieee80211_sta_block_awake` if frames are buffered when it is notified

that the station went to sleep. When all these frames have been filtered (see above), it must call the function again to indicate that the station is no longer blocked.

If the driver buffers frames in the driver for aggregation in any way, it must use the `ieee80211_sta_set_buffered` call when it is notified of the station going to sleep to inform mac80211 of any TIDs that have frames buffered. Note that when a station wakes up this information is reset (hence the requirement to call it when informed of the station going to sleep). Then, when a service period starts for any reason, *`release_buffered_frames`* is called with the number of frames to be released and which TIDs they are to come from. In this case, the driver is responsible for setting the EOSP (for uAPSD) and MORE_DATA bits in the released frames, to help the *`more_data`* parameter is passed to tell the driver if there is more data on other TIDs -- the TIDs to release frames from are ignored since mac80211 doesn't know how many frames the buffers for those TIDs contain.

If the driver also implement GO mode, where absence periods may shorten service periods (or abort PS-Poll responses), it must filter those response frames except in the case of frames that are buffered in the driver -- those must remain buffered to avoid reordering. Because it is possible that no frames are released in this case, the driver must call `ieee80211_sta_eosp` to indicate to mac80211 that the service period ended anyway.

Finally, if frames from multiple TIDs are released from mac80211 but the driver might reorder them, it must clear & set the flags appropriately (only the last frame may have `IEEE80211_TX_STATUS_EOSP`) and also take care of the EOSP and MORE_DATA bits in the frame. The driver may also use `ieee80211_sta_eosp` in this case.

Note that if the driver ever buffers frames other than QoS-data frames, it must take care to never send a non-QoS-data frame as the last frame in a service period, adding a QoS-nulldata frame after a non-QoS-data frame if needed.

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 the add_interface callback.

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.

Return

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.

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 the add_interface callback.

Description

See `ieee80211_beacon_get_tim`.

Return

See `ieee80211_beacon_get_tim`.

Name

ieee80211_sta_eosp — notify mac80211 about end of SP

Synopsis

```
void ieee80211_sta_eosp (struct ieee80211_sta * pubsta);
```

Arguments

pubsta the station

Description

When a device transmits frames in a way that it can't tell mac80211 in the TX status about the EOSP, it must clear the `IEEE80211_TX_STATUS_EOSP` bit and call this function instead. This applies for PS-Poll as well as uAPSD.

Note that just like with `_tx_status` and `_rx` drivers must not mix calls to irqsafe/non-irqsafe versions, this function must not be mixed with those either. Use the all irqsafe, or all non-irqsafe, don't mix!

NB

the _irqsafe version of this function doesn't exist, no driver needs it right now. Don't call this function if you'd need the _irqsafe version, look at the git history and restore the _irqsafe version!

Name

enum ieee80211_frame_release_type — frame release reason

Synopsis

```
enum ieee80211_frame_release_type {
  IEEE80211_FRAME_RELEASE_PSPOLL,
  IEEE80211_FRAME_RELEASE_UAPSD
};
```

Constants

IEEE80211_FRAME_RELEASE_PSPOLL
: frame released for PS-Poll

IEEE80211_FRAME_RELEASE_UAPSD
: frame(s) released due to frame received on trigger-enabled AC

Name

ieee80211_sta_ps_transition — PS transition for connected sta

Synopsis

```
int ieee80211_sta_ps_transition (struct ieee80211_sta * sta, bool start);
```

Arguments

sta currently connected sta

start start or stop PS

Description

When operating in AP mode with the `IEEE80211_HW_AP_LINK_PS` flag set, use this function to inform mac80211 about a connected station entering/leaving PS mode.

This function may not be called in IRQ context or with softirqs enabled.

Calls to this function for a single hardware must be synchronized against each other.

Return

0 on success. -EINVAL when the requested PS mode is already set.

Name

ieee80211_sta_ps_transition_ni — PS transition for connected sta (in process context)

Synopsis

```
int ieee80211_sta_ps_transition_ni (struct ieee80211_sta * sta, bool start);
```

Arguments

sta currently connected sta

start start or stop PS

Description

Like `ieee80211_sta_ps_transition` but can be called in process context (internally disables bottom halves). Concurrent call restriction still applies.

Return

Like `ieee80211_sta_ps_transition`.

Name

ieee80211_sta_set_buffered — inform mac80211 about driver-buffered frames

Synopsis

```
void ieee80211_sta_set_buffered (struct ieee80211_sta * sta, u8 tid,
bool buffered);
```

Arguments

sta struct ieee80211_sta pointer for the sleeping station

tid the TID that has buffered frames

buffered indicates whether or not frames are buffered for this TID

Description

If a driver buffers frames for a powersave station instead of passing them back to mac80211 for retransmission, the station may still need to be told that there are buffered frames via the TIM bit.

This function informs mac80211 whether or not there are frames that are buffered in the driver for a given TID; mac80211 can then use this data to set the TIM bit (NOTE: This may call back into the driver's set_tim call! Beware of the locking!)

If all frames are released to the station (due to PS-poll or uAPSD) then the driver needs to inform mac80211 that there no longer are frames buffered. However, when the station wakes up mac80211 assumes that all buffered frames will be transmitted and clears this data, drivers need to make sure they inform mac80211 about all buffered frames on the sleep transition (`sta_notify` with `STA_NOTIFY_SLEEP`).

Note that technically mac80211 only needs to know this per AC, not per TID, but since driver buffering will inevitably happen per TID (since it is related to aggregation) it is easier to make mac80211 map the TID to the AC as required instead of keeping track in all drivers that use this API.

Name

ieee80211_sta_block_awake — block station from waking up

Synopsis

```
void ieee80211_sta_block_awake (struct ieee80211_hw * hw, struct
ieee80211_sta * pubsta, bool block);
```

Arguments

hw the hardware

pubsta the station

block whether to block or unblock

Description

Some devices require that all frames that are on the queues for a specific station that went to sleep are flushed before a poll response or frames after the station woke up can be delivered to that it. Note that such frames must be rejected by the driver as filtered, with the appropriate status flag.

This function allows implementing this mode in a race-free manner.

To do this, a driver must keep track of the number of frames still enqueued for a specific station. If this number is not zero when the station goes to sleep, the driver must call this function to force mac80211 to consider the station to be asleep regardless of the station's actual state. Once the number of outstanding frames reaches zero, the driver must call this function again to unblock the station. That will cause mac80211 to be able to send ps-poll responses, and if the station queried in the meantime then frames will also be sent out as a result of this. Additionally, the driver will be notified that the station woke up some time after it is unblocked, regardless of whether the station actually woke up while blocked or not.

Chapter 13. 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.

Name

ieee80211_iterate_active_interfaces — iterate active interfaces

Synopsis

```
void ieee80211_iterate_active_interfaces (struct ieee80211_hw * hw, u32
iter_flags, void (*iterator) (void *data, u8 *mac, struct ieee80211_vif
*vif), void * data);
```

Arguments

hw	the hardware struct of which the interfaces should be iterated over
iter_flags	iteration flags, see enum ieee80211_interface_iteration_flags
iterator	the iterator function to call
data	first argument of the iterator function

Description

This function iterates over the interfaces associated with a given hardware that are currently active and calls the callback for them. This function allows the iterator function to sleep, when the iterator function is atomic *ieee80211_iterate_active_interfaces_atomic* can be used. Does not iterate over a new interface during add_interface.

Name

ieee80211_iterate_active_interfaces_atomic — iterate active interfaces

Synopsis

```
void ieee80211_iterate_active_interfaces_atomic (struct ieee80211_hw *
hw, u32 iter_flags, void (*iterator) (void *data, u8 *mac, struct
ieee80211_vif *vif), void * data);
```

Arguments

hw	the hardware struct of which the interfaces should be iterated over
iter_flags	iteration flags, see enum ieee80211_interface_iteration_flags
iterator	the iterator function to call, cannot sleep
data	first argument of the iterator function

Description

This function iterates over the interfaces associated with a given hardware that are currently active and calls the callback for them. This function requires the iterator callback function to be atomic, if that is not desired, use *ieee80211_iterate_active_interfaces* instead. Does not iterate over a new interface during add_interface.

Chapter 14. Station handling

TODO

Name

struct ieee80211_sta — station table entry

Synopsis

```
struct ieee80211_sta {
  u32 supp_rates[IEEE80211_NUM_BANDS];
  u8 addr[ETH_ALEN];
  u16 aid;
  struct ieee80211_sta_ht_cap ht_cap;
  struct ieee80211_sta_vht_cap vht_cap;
  bool wme;
  u8 uapsd_queues;
  u8 max_sp;
  u8 rx_nss;
  enum ieee80211_sta_rx_bandwidth bandwidth;
  enum ieee80211_smps_mode smps_mode;
  struct ieee80211_sta_rates __rcu * rates;
  bool tdls;
  bool tdls_initiator;
  bool mfp;
  struct ieee80211_txq * txq[IEEE80211_NUM_TIDS];
  u8 drv_priv[0];
};
```

Members

supp_rates[IEEE80211_NUM_BANDS]	Bitmap of supported rates (per band)
addr[ETH_ALEN]	MAC address
aid	AID we assigned to the station if we're an AP
ht_cap	HT capabilities of this STA; restricted to our own capabilities
vht_cap	VHT capabilities of this STA; restricted to our own capabilities
wme	indicates whether the STA supports QoS/WME (if local devices does, otherwise always false)
uapsd_queues	bitmap of queues configured for uapsd. Only valid if wme is supported.
max_sp	max Service Period. Only valid if wme is supported.
rx_nss	in HT/VHT, the maximum number of spatial streams the station can receive at the moment, changed by operating mode notifications and capabilities. The value is only valid after the station moves to associated state.
bandwidth	current bandwidth the station can receive with
smps_mode	current SMPS mode (off, static or dynamic)

rates	rate control selection table
tdls	indicates whether the STA is a TDLS peer
tdls_initiator	indicates the STA is an initiator of the TDLS link. Only valid if the STA is a TDLS peer in the first place.
mfp	indicates whether the STA uses management frame protection or not.
txq[IEEE80211_NUM_TIDS]	per-TID data TX queues (if driver uses the TXQ abstraction)
drv_priv[0]	data area for driver use, will always be aligned to sizeof(void *), size is determined in hw information.

Description

A station table entry represents a station we are possibly communicating with. Since stations are RCU-managed in mac80211, any ieee80211_sta pointer you get access to must either be protected by `rcu_read_lock` explicitly or implicitly, or you must take good care to not use such a pointer after a call to your sta_remove callback that removed it.

Name

enum sta_notify_cmd — sta notify command

Synopsis

```
enum sta_notify_cmd {
  STA_NOTIFY_SLEEP,
  STA_NOTIFY_AWAKE
};
```

Constants

STA_NOTIFY_SLEEP a station is now sleeping

STA_NOTIFY_AWAKE a sleeping station woke up

Description

Used with the `sta_notify` callback in struct ieee80211_ops, this indicates if an associated station made a power state transition.

Name

ieee80211_find_sta — find a station

Synopsis

```
struct ieee80211_sta * ieee80211_find_sta (struct ieee80211_vif * vif,
const u8 * addr);
```

Arguments

vif virtual interface to look for station on

addr station's address

Return

The station, if found. NULL otherwise.

Note

This function must be called under RCU lock and the resulting pointer is only valid under RCU lock as well.

Name

ieee80211_find_sta_by_ifaddr — find a station on hardware

Synopsis

```
struct ieee80211_sta * ieee80211_find_sta_by_ifaddr (struct
ieee80211_hw * hw, const u8 * addr, const u8 * localaddr);
```

Arguments

hw	pointer as obtained from `ieee80211_alloc_hw`
addr	remote station's address
localaddr	local address (vif->sdata->vif.addr). Use NULL for 'any'.

Return

The station, if found. `NULL` otherwise.

Note

This function must be called under RCU lock and the resulting pointer is only valid under RCU lock as well.

NOTE

You may pass NULL for localaddr, but then you will just get the first STA that matches the remote address 'addr'. We can have multiple STA associated with multiple logical stations (e.g. consider a station connecting to another BSSID on the same AP hardware without disconnecting first). In this case, the result of this method with localaddr NULL is not reliable.

DO NOT USE THIS FUNCTION with localaddr NULL if at all possible.

Chapter 15. Hardware scan offload

TBD

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. This function can be called from any context, including hardirq context.

Chapter 16. Aggregation

TX A-MPDU aggregation

Aggregation on the TX side requires setting the hardware flag `IEEE80211_HW_AMPDU_AGGREGATION`. The driver will then be handed packets with a flag indicating A-MPDU aggregation. The driver or device is responsible for actually aggregating the frames, as well as deciding how many and which to aggregate.

When TX aggregation is started by some subsystem (usually the rate control algorithm would be appropriate) by calling the `ieee80211_start_tx_ba_session` function, the driver will be notified via its *`ampdu_action`* function, with the `IEEE80211_AMPDU_TX_START` action.

In response to that, the driver is later required to call the `ieee80211_start_tx_ba_cb_irqsafe` function, which will really start the aggregation session after the peer has also responded. If the peer responds negatively, the session will be stopped again right away. Note that it is possible for the aggregation session to be stopped before the driver has indicated that it is done setting it up, in which case it must not indicate the setup completion.

Also note that, since we also need to wait for a response from the peer, the driver is notified of the completion of the handshake by the `IEEE80211_AMPDU_TX_OPERATIONAL` action to the *`ampdu_action`* callback.

Similarly, when the aggregation session is stopped by the peer or something calling `ieee80211_stop_tx_ba_session`, the driver's *`ampdu_action`* function will be called with the action `IEEE80211_AMPDU_TX_STOP`. In this case, the call must not fail, and the driver must later call `ieee80211_stop_tx_ba_cb_irqsafe`. Note that the sta can get destroyed before the BA tear down is complete.

RX A-MPDU aggregation

Aggregation on the RX side requires only implementing the *`ampdu_action`* callback that is invoked to start/stop any block-ack sessions for RX aggregation.

When RX aggregation is started by the peer, the driver is notified via *`ampdu_action`* function, with the `IEEE80211_AMPDU_RX_START` action, and may reject the request in which case a negative response is sent to the peer, if it accepts it a positive response is sent.

While the session is active, the device/driver are required to de-aggregate frames and pass them up one by one to mac80211, which will handle the reorder buffer.

When the aggregation session is stopped again by the peer or ourselves, the driver's *`ampdu_action`* function will be called with the action `IEEE80211_AMPDU_RX_STOP`. In this case, the call must not fail.

Name

enum ieee80211_ampdu_mlme_action — A-MPDU actions

Synopsis

```
enum ieee80211_ampdu_mlme_action {
  IEEE80211_AMPDU_RX_START,
  IEEE80211_AMPDU_RX_STOP,
  IEEE80211_AMPDU_TX_START,
  IEEE80211_AMPDU_TX_STOP_CONT,
  IEEE80211_AMPDU_TX_STOP_FLUSH,
  IEEE80211_AMPDU_TX_STOP_FLUSH_CONT,
  IEEE80211_AMPDU_TX_OPERATIONAL
};
```

Constants

IEEE80211_AMPDU_RX_START — start RX aggregation

IEEE80211_AMPDU_RX_STOP — stop RX aggregation

IEEE80211_AMPDU_TX_START — start TX aggregation

IEEE80211_AMPDU_TX_STOP_CONT — stop TX aggregation but continue transmitting queued packets, now unaggregated. After all packets are transmitted the driver has to call `ieee80211_stop_tx_ba_cb_irqsafe`.

IEEE80211_AMPDU_TX_STOP_FLUSH — stop TX aggregation and flush all packets, called when the station is removed. There's no need or reason to call `ieee80211_stop_tx_ba_cb_irqsafe` in this case as mac80211 assumes the session is gone and removes the station.

IEEE80211_AMPDU_TX_STOP_FLUSH_CONT — called when TX aggregation is stopped but the driver hasn't called `ieee80211_stop_tx_ba_cb_irqsafe` yet and now the connection is dropped and the station will be removed. Drivers should clean up and drop remaining packets when this is called.

IEEE80211_AMPDU_TX_OPERATIONAL — TX aggregation has become operational

Description

These flags are used with the `ampdu_action` callback in struct ieee80211_ops to indicate which action is needed.

Note that drivers MUST be able to deal with a TX aggregation session being stopped even before they OK'ed starting it by calling ieee80211_start_tx_ba_cb_irqsafe, because the peer might receive the addBA frame and send a delBA right away!

Chapter 17. Spatial Multiplexing Powersave (SMPS)

SMPS (Spatial multiplexing power save) is a mechanism to conserve power in an 802.11n implementation. For details on the mechanism and rationale, please refer to 802.11 (as amended by 802.11n-2009) “11.2.3 SM power save”.

The mac80211 implementation is capable of sending action frames to update the AP about the station's SMPS mode, and will instruct the driver to enter the specific mode. It will also announce the requested SMPS mode during the association handshake. Hardware support for this feature is required, and can be indicated by hardware flags.

The default mode will be “automatic”, which nl80211/cfg80211 defines to be dynamic SMPS in (regular) powersave, and SMPS turned off otherwise.

To support this feature, the driver must set the appropriate hardware support flags, and handle the SMPS flag to the `config` operation. It will then with this mechanism be instructed to enter the requested SMPS mode while associated to an HT AP.

Name

ieee80211_request_smps — request SM PS transition

Synopsis

```
void ieee80211_request_smps (struct ieee80211_vif * vif, enum ieee80211_smps_mode smps_mode);
```

Arguments

vif struct ieee80211_vif pointer from the add_interface callback.

smps_mode new SM PS mode

Description

This allows the driver to request an SM PS transition in managed mode. This is useful when the driver has more information than the stack about possible interference, for example by bluetooth.

Name

enum ieee80211_smps_mode — spatial multiplexing power save mode

Synopsis

```
enum ieee80211_smps_mode {
  IEEE80211_SMPS_AUTOMATIC,
  IEEE80211_SMPS_OFF,
  IEEE80211_SMPS_STATIC,
  IEEE80211_SMPS_DYNAMIC,
  IEEE80211_SMPS_NUM_MODES
};
```

Constants

IEEE80211_SMPS_AUTOMATIC	automatic
IEEE80211_SMPS_OFF	off
IEEE80211_SMPS_STATIC	static
IEEE80211_SMPS_DYNAMIC	dynamic
IEEE80211_SMPS_NUM_MODES	internal, don't use

Part III. Rate control interface

TBD

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

Table of Contents

18. Rate Control API 121
ieee80211_start_tx_ba_session 122
ieee80211_start_tx_ba_cb_irqsafe 123
ieee80211_stop_tx_ba_session 124
ieee80211_stop_tx_ba_cb_irqsafe 125
enum ieee80211_rate_control_changed 126
struct ieee80211_tx_rate_control 127
rate_control_send_low 128

Chapter 18. Rate Control API

TBD

Name

ieee80211_start_tx_ba_session — Start a tx Block Ack session.

Synopsis

```
int ieee80211_start_tx_ba_session (struct ieee80211_sta * sta, u16 tid,
u16 timeout);
```

Arguments

sta the station for which to start a BA session

tid the TID to BA on.

timeout session timeout value (in TUs)

Return

success if addBA request was sent, failure otherwise

Although mac80211/low level driver/user space application can estimate the need to start aggregation on a certain RA/TID, the session level will be managed by the mac80211.

Name

ieee80211_start_tx_ba_cb_irqsafe — low level driver ready to aggregate.

Synopsis

```
void ieee80211_start_tx_ba_cb_irqsafe (struct ieee80211_vif * vif, const
u8 * ra, u16 tid);
```

Arguments

vif struct ieee80211_vif pointer from the add_interface callback

ra receiver address of the BA session recipient.

tid the TID to BA on.

Description

This function must be called by low level driver once it has finished with preparations for the BA session. It can be called from any context.

Name

ieee80211_stop_tx_ba_session — Stop a Block Ack session.

Synopsis

```
int ieee80211_stop_tx_ba_session (struct ieee80211_sta * sta, u16 tid);
```

Arguments

sta the station whose BA session to stop

tid the TID to stop BA.

Return

negative error if the TID is invalid, or no aggregation active

Although mac80211/low level driver/user space application can estimate the need to stop aggregation on a certain RA/TID, the session level will be managed by the mac80211.

Name

ieee80211_stop_tx_ba_cb_irqsafe — low level driver ready to stop aggregate.

Synopsis

```
void ieee80211_stop_tx_ba_cb_irqsafe (struct ieee80211_vif * vif, const
u8 * ra, u16 tid);
```

Arguments

vif struct ieee80211_vif pointer from the add_interface callback

ra receiver address of the BA session recipient.

tid the desired TID to BA on.

Description

This function must be called by low level driver once it has finished with preparations for the BA session tear down. It can be called from any context.

Name

enum ieee80211_rate_control_changed — flags to indicate what changed

Synopsis

```
enum ieee80211_rate_control_changed {
  IEEE80211_RC_BW_CHANGED,
  IEEE80211_RC_SMPS_CHANGED,
  IEEE80211_RC_SUPP_RATES_CHANGED,
  IEEE80211_RC_NSS_CHANGED
};
```

Constants

IEEE80211_RC_BW_CHANGED	The bandwidth that can be used to transmit to this station changed. The actual bandwidth is in the station information -- for HT20/40 the IEEE80211_HT_CAP_SUP_WIDTH_20_40 flag changes, for HT and VHT the bandwidth field changes.
IEEE80211_RC_SMPS_CHANGED	The SMPS state of the station changed.
IEEE80211_RC_SUPP_RATES_CHANGED	The supported rate set of this peer changed (in IBSS mode) due to discovering more information about the peer.
IEEE80211_RC_NSS_CHANGED	N_SS (number of spatial streams) was changed by the peer

Name

struct ieee80211_tx_rate_control — rate control information for/from RC algo

Synopsis

```
struct ieee80211_tx_rate_control {
  struct ieee80211_hw * hw;
  struct ieee80211_supported_band * sband;
  struct ieee80211_bss_conf * bss_conf;
  struct sk_buff * skb;
  struct ieee80211_tx_rate reported_rate;
  bool rts;
  bool short_preamble;
  u8 max_rate_idx;
  u32 rate_idx_mask;
  u8 * rate_idx_mcs_mask;
  bool bss;
};
```

Members

hw	The hardware the algorithm is invoked for.
sband	The band this frame is being transmitted on.
bss_conf	the current BSS configuration
skb	the skb that will be transmitted, the control information in it needs to be filled in
reported_rate	The rate control algorithm can fill this in to indicate which rate should be reported to userspace as the current rate and used for rate calculations in the mesh network.
rts	whether RTS will be used for this frame because it is longer than the RTS threshold
short_preamble	whether mac80211 will request short-preamble transmission if the selected rate supports it
max_rate_idx	user-requested maximum (legacy) rate (deprecated; this will be removed once drivers get updated to use rate_idx_mask)
rate_idx_mask	user-requested (legacy) rate mask
rate_idx_mcs_mask	user-requested MCS rate mask (NULL if not in use)
bss	whether this frame is sent out in AP or IBSS mode

Name

rate_control_send_low — helper for drivers for management/no-ack frames

Synopsis

```
bool rate_control_send_low (struct ieee80211_sta * sta, void * priv_sta,
struct ieee80211_tx_rate_control * txrc);
```

Arguments

sta	struct ieee80211_sta pointer to the target destination. Note that this may be null.
priv_sta	private rate control structure. This may be null.
txrc	rate control information we sholud populate for mac80211.

Description

Rate control algorithms that agree to use the lowest rate to send management frames and NO_ACK data with the respective hw retries should use this in the beginning of their mac80211 get_rate callback. If true is returned the rate control can simply return. If false is returned we guarantee that sta and sta and priv_sta is not null.

Rate control algorithms wishing to do more intelligent selection of rate for multicast/broadcast frames may choose to not use this.

Part IV. Internals

TBD

This part of the book describes mac80211 internals.

Table of Contents

19. Key handling 131
Key handling basics 131
MORE TBD 131
20. Receive processing 132
21. Transmit processing 133
22. Station info handling 134
Programming information 134
STA information lifetime rules 141
23. Aggregation 143
struct sta_ampdu_mlme 144
struct tid_ampdu_tx 145
struct tid_ampdu_rx 147
24. Synchronisation 149

Chapter 19. Key handling

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 algorithms that are implemented in software, for each key the hardware is asked to enable that key for offloading but if it cannot do that the key is simply kept for software encryption (unless it is for an algorithm that isn't implemented in software). There is currently no way of knowing whether a key is handled in SW or HW except by looking into debugfs.

All key management is internally protected by a mutex. Within all other parts of 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 remove the key which waits for an RCU grace period.

MORE TBD

TBD

Chapter 20. Receive processing

TBD

Chapter 21. Transmit processing

TBD

Chapter 22. Station info handling

Programming information

Name

struct sta_info — STA information

Synopsis

```
struct sta_info {
  struct list_head list;
  struct list_head free_list;
  struct rcu_head rcu_head;
  struct rhash_head hash_node;
  struct ieee80211_local * local;
  struct ieee80211_sub_if_data * sdata;
  struct ieee80211_key __rcu * gtk[NUM_DEFAULT_KEYS + NUM_DEFAULT_MGMT_KEYS];
  struct ieee80211_key __rcu * ptk[NUM_DEFAULT_KEYS];
  u8 gtk_idx;
  u8 ptk_idx;
  struct rate_control_ref * rate_ctrl;
  void * rate_ctrl_priv;
  spinlock_t lock;
  struct work_struct drv_deliver_wk;
  u16 listen_interval;
  bool dead;
  bool uploaded;
  enum ieee80211_sta_state sta_state;
  unsigned long _flags;
  spinlock_t ps_lock;
  struct sk_buff_head ps_tx_buf[IEEE80211_NUM_ACS];
  struct sk_buff_head tx_filtered[IEEE80211_NUM_ACS];
  unsigned long driver_buffered_tids;
  unsigned long txq_buffered_tids;
  unsigned long rx_packets;
  u64 rx_bytes;
  unsigned long last_rx;
  long last_connected;
  unsigned long num_duplicates;
  unsigned long rx_fragments;
  unsigned long rx_dropped;
  int last_signal;
  struct ewma avg_signal;
  int last_ack_signal;
  u8 chains;
  s8 chain_signal_last[IEEE80211_MAX_CHAINS];
  struct ewma chain_signal_avg[IEEE80211_MAX_CHAINS];
  __le16 last_seq_ctrl[IEEE80211_NUM_TIDS + 1];
  unsigned long tx_filtered_count;
  unsigned long tx_retry_failed;
  unsigned long tx_retry_count;
  unsigned int fail_avg;
  u32 tx_fragments;
  u64 tx_packets[IEEE80211_NUM_ACS];
  u64 tx_bytes[IEEE80211_NUM_ACS];
  struct ieee80211_tx_rate last_tx_rate;
```

```
  int last_rx_rate_idx;
  u32 last_rx_rate_flag;
  u32 last_rx_rate_vht_flag;
  u8 last_rx_rate_vht_nss;
  u16 tid_seq[IEEE80211_QOS_CTL_TID_MASK + 1];
  u64 tx_msdu[IEEE80211_NUM_TIDS + 1];
  u64 tx_msdu_retries[IEEE80211_NUM_TIDS + 1];
  u64 tx_msdu_failed[IEEE80211_NUM_TIDS + 1];
  u64 rx_msdu[IEEE80211_NUM_TIDS + 1];
  struct sta_ampdu_mlme ampdu_mlme;
  u8 timer_to_tid[IEEE80211_NUM_TIDS];
#ifdef CONFIG_MAC80211_MESH
  u16 llid;
  u16 plid;
  u16 reason;
  u8 plink_retries;
  enum nl80211_plink_state plink_state;
  u32 plink_timeout;
  struct timer_list plink_timer;
  s64 t_offset;
  s64 t_offset_setpoint;
  enum nl80211_mesh_power_mode local_pm;
  enum nl80211_mesh_power_mode peer_pm;
  enum nl80211_mesh_power_mode nonpeer_pm;
#endif
#ifdef CONFIG_MAC80211_DEBUGFS
  struct sta_info_debugfsdentries debugfs;
#endif
  enum ieee80211_sta_rx_bandwidth cur_max_bandwidth;
  unsigned int lost_packets;
  unsigned int beacon_loss_count;
  enum ieee80211_smps_mode known_smps_mode;
  const struct ieee80211_cipher_scheme * cipher_scheme;
  unsigned long last_tdls_pkt_time;
  u8 reserved_tid;
  struct ieee80211_sta sta;
};
```

Members

list	global linked list entry
free_list	list entry for keeping track of stations to free
rcu_head	RCU head used for freeing this station struct
hash_node	hash node for rhashtable
local	pointer to the global information
sdata	virtual interface this station belongs to
gtk[NUM_DEFAULT_KEYS + NUM_DEFAULT_MGMT_KEYS]	group keys negotiated with this station, if any
ptk[NUM_DEFAULT_KEYS]	peer keys negotiated with this station, if any

gtk_idx	last installed group key index
ptk_idx	last installed peer key index
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.
drv_deliver_wk	used for delivering frames after driver PS unblocking
listen_interval	listen interval of this station, when we're acting as AP
dead	set to true when sta is unlinked
uploaded	set to true when sta is uploaded to the driver
sta_state	duplicates information about station state (for debug)
_flags	STA flags, see enum ieee80211_sta_info_flags, do not use directly
ps_lock	used for powersave (when mac80211 is the AP) related locking
ps_tx_buf[IEEE80211_NUM_ACS]	buffers (per AC) of frames to transmit to this station when it leaves power saving state or polls
tx_filtered[IEEE80211_NUM_ACS]	buffers (per AC) of frames we already tried to transmit but were filtered by hardware due to STA having entered power saving state, these are also delivered to the station when it leaves powersave or polls for frames
driver_buffered_tids	bitmap of TIDs the driver has data buffered on
txq_buffered_tids	bitmap of TIDs that mac80211 has txq data buffered on
rx_packets	Number of MSDUs received from this STA
rx_bytes	Number of bytes received from this STA
last_rx	time (in jiffies) when last frame was received from this STA
last_connected	time (in seconds) when a station got connected
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
avg_signal	moving average of signal of received frames from this STA
last_ack_signal	signal of last received Ack frame from this STA
chains	chains ever used for RX from this station

chain_signal_last[IEEE80211_MAX_CHAINS]	last signal (per chain)
chain_signal_avg[IEEE80211_MAX_CHAINS]	signal average (per chain)
last_seq_ctrl[IEEE80211_NUM_TIDS + 1]	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_fragments	number of transmitted MPDUs
tx_packets[IEEE80211_NUM_ACS]	number of RX/TX MSDUs
tx_bytes[IEEE80211_NUM_ACS]	number of bytes transmitted to this STA
last_tx_rate	rate used for last transmit, to report to userspace as “the” transmit rate
last_rx_rate_idx	rx status rate index of the last data packet
last_rx_rate_flag	rx status flag of the last data packet
last_rx_rate_vht_flag	rx status vht flag of the last data packet
last_rx_rate_vht_nss	rx status nss of last data packet
tid_seq[IEEE80211_QOS_CTL_TID_MASK + 1]	TID sequence numbers for sending to this STA
tx_msdu[IEEE80211_NUM_TIDS + 1]	MSDUs transmitted to this station, using IEEE80211_NUM_TID entry for non-QoS frames
tx_msdu_retries[IEEE80211_NUM_TIDS + 1]	MSDU retries for transmissions to to this station, using IEEE80211_NUM_TID entry for non-QoS frames
tx_msdu_failed[IEEE80211_NUM_TIDS + 1]	MSDU failures for transmissions to to this station, using IEEE80211_NUM_TID entry for non-QoS frames
rx_msdu[IEEE80211_NUM_TIDS + 1]	MSDUs received from this station, using IEEE80211_NUM_TID entry for non-QoS frames
ampdu_mlme	A-MPDU state machine state
timer_to_tid[IEEE80211_NUM_TIDS]	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

plink_state	peer link state
plink_timeout	timeout of peer link
plink_timer	peer link watch timer
t_offset	timing offset relative to this host
t_offset_setpoint	reference timing offset of this sta to be used when calculating clockdrift
local_pm	local link-specific power save mode
peer_pm	peer-specific power save mode towards local STA
nonpeer_pm	STA power save mode towards non-peer neighbors
debugfs	debug filesystem info
cur_max_bandwidth	maximum bandwidth to use for TX to the station, taken from HT/VHT capabilities or VHT operating mode notification
lost_packets	number of consecutive lost packets
beacon_loss_count	number of times beacon loss has triggered
known_smps_mode	the smps_mode the client thinks we are in. Relevant for AP only.
cipher_scheme	optional cipher scheme for this station
last_tdls_pkt_time	holds the time in jiffies of last TDLS pkt ACKed
reserved_tid	reserved TID (if any, otherwise IEEE80211_TID_UNRESERVED)
sta	station information we share with the driver

Description

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

Name

enum ieee80211_sta_info_flags — Stations flags

Synopsis

```
enum ieee80211_sta_info_flags {
  WLAN_STA_AUTH,
  WLAN_STA_ASSOC,
  WLAN_STA_PS_STA,
  WLAN_STA_AUTHORIZED,
  WLAN_STA_SHORT_PREAMBLE,
  WLAN_STA_WDS,
  WLAN_STA_CLEAR_PS_FILT,
  WLAN_STA_MFP,
  WLAN_STA_BLOCK_BA,
  WLAN_STA_PS_DRIVER,
  WLAN_STA_PSPOLL,
  WLAN_STA_TDLS_PEER,
  WLAN_STA_TDLS_PEER_AUTH,
  WLAN_STA_TDLS_INITIATOR,
  WLAN_STA_TDLS_CHAN_SWITCH,
  WLAN_STA_TDLS_OFF_CHANNEL,
  WLAN_STA_UAPSD,
  WLAN_STA_SP,
  WLAN_STA_4ADDR_EVENT,
  WLAN_STA_INSERTED,
  WLAN_STA_RATE_CONTROL,
  WLAN_STA_TOFFSET_KNOWN,
  WLAN_STA_MPSP_OWNER,
  WLAN_STA_MPSP_RECIPIENT,
  WLAN_STA_PS_DELIVER
};
```

Constants

WLAN_STA_AUTH	Station is authenticated.
WLAN_STA_ASSOC	Station is associated.
WLAN_STA_PS_STA	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_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_BLOCK_BA	Used to deny ADDBA requests (both TX and RX) during suspend/resume and station removal.
WLAN_STA_PS_DRIVER	driver requires keeping this station in power-save mode logically to flush frames that might still be in the queues
WLAN_STA_PSPOLL	Station sent PS-poll while driver was keeping station in power-save mode, reply when the driver unblocks.
WLAN_STA_TDLS_PEER	Station is a TDLS peer.
WLAN_STA_TDLS_PEER_AUTH	This TDLS peer is authorized to send direct packets. This means the link is enabled.
WLAN_STA_TDLS_INITIATOR	We are the initiator of the TDLS link with this station.
WLAN_STA_TDLS_CHAN_SWITCH	This TDLS peer supports TDLS channel-switching
WLAN_STA_TDLS_OFF_CHANNEL	The local STA is currently off-channel with this TDLS peer
WLAN_STA_UAPSD	Station requested unscheduled SP while driver was keeping station in power-save mode, reply when the driver unblocks the station.
WLAN_STA_SP	Station is in a service period, so don't try to reply to other uAPSD trigger frames or PS-Poll.
WLAN_STA_4ADDR_EVENT	4-addr event was already sent for this frame.
WLAN_STA_INSERTED	This station is inserted into the hash table.
WLAN_STA_RATE_CONTROL	rate control was initialized for this station.
WLAN_STA_TOFFSET_KNOWN	toffset calculated for this station is valid.
WLAN_STA_MPSP_OWNER	local STA is owner of a mesh Peer Service Period.
WLAN_STA_MPSP_RECIPIENT	local STA is recipient of a MPSP.
WLAN_STA_PS_DELIVER	station woke up, but we're still blocking TX until pending frames are delivered

Description

These flags are used with struct sta_info's *`flags`* member, but only indirectly with `set_sta_flag` and friends.

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 insert it into the hash table using either `sta_info_insert` or `sta_info_insert_rcu`; only in the latter case (which acquires an rcu read section but must not be called from within one) will the pointer

still be valid after the call. 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`!

Station 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 get to know about a peer on the same IBSS. 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.

In order to remove a STA info structure, various sta_info_destroy_*() calls are available.

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 23. Aggregation

Name

struct sta_ampdu_mlme — STA aggregation information.

Synopsis

```
struct sta_ampdu_mlme {
  struct mutex mtx;
  struct tid_ampdu_rx __rcu * tid_rx[IEEE80211_NUM_TIDS];
  unsigned long tid_rx_timer_expired[BITS_TO_LONGS(IEEE80211_NUM_TIDS)];
  unsigned long tid_rx_stop_requested[BITS_TO_LONGS(IEEE80211_NUM_TIDS)];
  struct work_struct work;
  struct tid_ampdu_tx __rcu * tid_tx[IEEE80211_NUM_TIDS];
  struct tid_ampdu_tx * tid_start_tx[IEEE80211_NUM_TIDS];
  unsigned long last_addba_req_time[IEEE80211_NUM_TIDS];
  u8 addba_req_num[IEEE80211_NUM_TIDS];
  u8 dialog_token_allocator;
};
```

Members

mtx — mutex to protect all TX data (except non-NULL assignments to tid_tx[idx], which are protected by the sta spinlock) tid_start_tx is also protected by sta->lock.

tid_rx[IEEE80211_NUM_TIDS] — aggregation info for Rx per TID -- RCU protected

tid_rx_timer_expired[BITS_TO_LONGS(IEEE80211_NUM_TIDS)] — bitmap indicating on which TIDs the RX timer expired until the work for it runs

tid_rx_stop_requested[BITS_TO_LONGS(IEEE80211_NUM_TIDS)] — bitmap indicating which BA sessions per TID the driver requested to close until the work for it runs

work — work struct for starting/stopping aggregation

tid_tx[IEEE80211_NUM_TIDS] — aggregation info for Tx per TID

tid_start_tx[IEEE80211_NUM_TIDS] — sessions where start was requested

last_addba_req_time[IEEE80211_NUM_TIDS] — timestamp of the last addBA request.

addba_req_num[IEEE80211_NUM_TIDS] — number of times addBA request has been sent.

dialog_token_allocator — dialog token enumerator for each new session;

Name

struct tid_ampdu_tx — TID aggregation information (Tx).

Synopsis

```
struct tid_ampdu_tx {
  struct rcu_head rcu_head;
  struct timer_list session_timer;
  struct timer_list addba_resp_timer;
  struct sk_buff_head pending;
  unsigned long state;
  unsigned long last_tx;
  u16 timeout;
  u8 dialog_token;
  u8 stop_initiator;
  bool tx_stop;
  u8 buf_size;
  u16 failed_bar_ssn;
  bool bar_pending;
};
```

Members

rcu_head	rcu head for freeing structure
session_timer	check if we keep Tx-ing on the TID (by timeout value)
addba_resp_timer	timer for peer's response to addba request
pending	pending frames queue -- use sta's spinlock to protect
state	session state (see above)
last_tx	jiffies of last tx activity
timeout	session timeout value to be filled in ADDBA requests
dialog_token	dialog token for aggregation session
stop_initiator	initiator of a session stop
tx_stop	TX DelBA frame when stopping
buf_size	reorder buffer size at receiver
failed_bar_ssn	ssn of the last failed BAR tx attempt
bar_pending	BAR needs to be re-sent

Description

This structure's lifetime is managed by RCU, assignments to the array holding it must hold the aggregation mutex.

The TX path can access it under RCU lock-free if, and only if, the state has the flag `HT_AGG_STATE_OPERATIONAL` set. Otherwise, the TX path must also acquire the spinlock and re-check the state, see comments in the tx code touching it.

Name

struct tid_ampdu_rx — TID aggregation information (Rx).

Synopsis

```
struct tid_ampdu_rx {
  struct rcu_head rcu_head;
  spinlock_t reorder_lock;
  struct sk_buff_head * reorder_buf;
  unsigned long * reorder_time;
  struct timer_list session_timer;
  struct timer_list reorder_timer;
  unsigned long last_rx;
  u16 head_seq_num;
  u16 stored_mpdu_num;
  u16 ssn;
  u16 buf_size;
  u16 timeout;
  u8 dialog_token;
  bool auto_seq;
  bool removed;
};
```

Members

rcu_head	RCU head used for freeing this struct
reorder_lock	serializes access to reorder buffer, see below.
reorder_buf	buffer to reorder incoming aggregated MPDUs. An MPDU may be an A-MSDU with individually reported subframes.
reorder_time	jiffies when skb was added
session_timer	check if peer keeps Tx-ing on the TID (by timeout value)
reorder_timer	releases expired frames from the reorder buffer.
last_rx	jiffies of last rx activity
head_seq_num	head sequence number in reordering buffer.
stored_mpdu_num	number of MPDUs in reordering buffer
ssn	Starting Sequence Number expected to be aggregated.
buf_size	buffer size for incoming A-MPDUs
timeout	reset timer value (in TUs).
dialog_token	dialog token for aggregation session
auto_seq	used for offloaded BA sessions to automatically pick head_seq_and and ssn.

removed this session is removed (but might have been found due to RCU)

Description

This structure's lifetime is managed by RCU, assignments to the array holding it must hold the aggregation mutex.

The *reorder_lock* is used to protect the members of this struct, except for *timeout*, *buf_size* and *dialog_token*, which are constant across the lifetime of the struct (the dialog token being used only for debugging).

Chapter 24. Synchronisation

TBD

Locking, lots of RCU