NAME

IEEE80211 - 802.11 network layer

SYNOPSIS

#include <net80211/ieee80211_var.h>

void
ieee80211_ifattach(*struct ieee80211com *ic*);

void
ieee80211_ifdetach(*struct ieee80211com *ic*);

int
ieee80211_mhz2ieee(*u_int freq, u_int flags*);

int
ieee80211_chan2ieee(*struct ieee80211com *ic, const struct ieee80211_channel *c*);

u_int
ieee80211_ieee2mhz(*u_int chan, u_int flags*);

int
ieee80211_media_change(*struct ifnet *ifp*);

void
ieee80211_media_status(*struct ifnet *ifp, struct ifmediareq *imr*);

int
ieee80211_setmode(*struct ieee80211com *ic, enum ieee80211_phymode mode*);

enum ieee80211_phymode
ieee80211_chan2mode(*const struct ieee80211_channel *chan*);

int
ieee80211_rate2media(*struct ieee80211com *ic, int rate, enum ieee80211_phymode mode*);

int
ieee80211_media2rate(*int mword*);

DESCRIPTION

IEEE 802.11 device drivers are written to use the infrastructure provided by the **IEEE80211** software layer. This software provides a support framework for drivers that includes ifnet cloning, state management, and a user management API by which applications interact with 802.11 devices. Most drivers depend on the **IEEE80211** layer for protocol services but devices that off-load functionality may bypass the layer to connect directly to the device.

A **IEEE80211** device driver implements a virtual radio API that is exported to users through network interfaces (aka vaps) that are cloned from the underlying device. These interfaces have an operating mode (station, adhoc, hostap, wds, monitor, etc.) that is fixed for the lifetime of the interface. Devices that can support multiple concurrent interfaces allow multiple vaps to be cloned. This enables construction of interesting applications such as an AP vap and one or more WDS vaps or multiple AP vaps, each with a different security model. The **IEEE80211** layer virtualizes most 802.11 state and coordinates vap state changes including scheduling multiple vaps. State that is not virtualized includes the current channel and WME/WMM parameters. Protocol processing is typically handled entirely in the **IEEE80211** layer with drivers responsible purely for moving data between the host and device. Similarly, **IEEE80211** handles most ioctl(2) requests without entering the driver; instead drivers are notified of state changes that require their involvement.

The virtual radio interface defined by the **IEEE80211** layer means that drivers must be structured to follow specific rules. Drivers that support only a single interface at any time must still follow these rules.

Most of these functions require that attachment to the stack is performed before calling.

The **ieee80211_ifattach**() function attaches the wireless network interface *ic* to the 802.11 network stack layer. This function must be called before using any of the **IEEE80211** functions which need to store driver state across invocations.

The **ieee80211_ifdetach**() function frees any **IEEE80211** structures associated with the driver, and performs Ethernet and BPF detachment on behalf of the caller.

The **ieee80211_mhz2ieee**() utility function converts the frequency *freq* (specified in MHz) to an IEEE 802.11 channel number. The *flags* argument is a hint which specifies whether the frequency is in the 2GHz ISM band (*IEEE80211_CHAN_2GHZ*) or the 5GHz band (*IEEE80211_CHAN_5GHZ*); appropriate clipping of the result is then performed.

The **ieee80211_chan2ieee**() function converts the channel specified in **c* to an IEEE channel number for the driver *ic*. If the conversion would be invalid, an error message is printed to the system console. This function REQUIRES that the driver is hooked up to the **IEEE80211** subsystem.

The **ieee80211_ieee2mhz**() utility function converts the IEEE channel number *chan* to a frequency (in MHz). The *flags* argument is a hint which specifies whether the frequency is in the 2GHz ISM band (*IEEE80211_CHAN_2GHZ*) or the 5GHz band (*IEEE80211_CHAN_5GHZ*); appropriate clipping of the result is then performed.

The **ieee80211_media_status**() and **ieee80211_media_change**() functions are device-independent handlers for *ifmedia* commands and are not intended to be called directly.

The **ieee80211_setmode**() function is called from within the 802.11 stack to change the mode of the driver's PHY; it is not intended to be called directly.

The **ieee80211_chan2mode**() function returns the PHY mode required for use with the channel *chan*. This is typically used when selecting a rate set, to be advertised in beacons, for example.

The **ieee80211_rate2media**() function converts the bit rate *rate* (measured in units of 0.5Mbps) to an *ifmedia* sub-type, for the device *ic* running in PHY mode *mode*. The **ieee80211_media2rate**() performs the reverse of this conversion, returning the bit rate (in 0.5Mbps units) corresponding to an *ifmedia* sub-type.

DATA STRUCTURES

The virtual radio architecture splits state between a single per-device *ieee80211com* structure and one or more *ieee80211vap* structures. Drivers are expected to setup various shared state in these structures at device attach and during vap creation but otherwise should treat them as read-only. The *ieee80211com* structure is allocated by the **IEEE80211** layer as adjunct data to a device's *ifnet*; it is accessed through the *if_l2com* structure member. The *ieee80211vap* structure is allocated by the driver in the "vap create" method and should be extended with any driver-private state. This technique of giving the driver control to allocate data structures is used for other **IEEE80211** data structures and should be exploited to maintain driver-private state together with public **IEEE80211** state.

The other main data structures are the station, or node, table that tracks peers in the local BSS, and the channel table that defines the current set of available radio channels. Both tables are bound to the *ieee80211com* structure and shared by all vaps. Long-lasting references to a node are counted to guard against premature reclamation. In particular every packet sent/received holds a node reference (either explicitly for transmit or implicitly on receive).

The *ieee80211com* and *ieee80211vap* structures also hold a collection of method pointers that drivers fill-in and/or override to take control of certain operations. These methods are the primary way drivers are bound to the **IEEE80211** layer and are described below.

DRIVER ATTACH/DETACH

Drivers attach to the **IEEE80211** layer with the **ieee80211_ifattach**() function. The driver is expected to allocate and setup any device-private data structures before passing control. The *ieee80211com* structure must be pre-initialized with state required to setup the **IEEE80211** layer:

ic_ifp Backpointer to the physical device's ifnet.

ic_caps Device/driver capabilities; see below for a complete description.

ic_channels Table of channels the device is capable of operating on. This is initially provided by the driver but may be changed through calls that change the regulatory state.

ic_nchan Number of entries in ic_channels.

On return from **ieee80211_ifattach**() the driver is expected to override default callback functions in the *ieee80211com* structure to register it's private routines. Methods marked with a "*" must be provided by the driver.

ic_vap_create*
: Create a vap instance of the specified type (operating mode). Any fixed BSSID and/or MAC address is provided. Drivers that support multi-bssid operation may honor the requested BSSID or assign their own.

ic_vap_delete*
: Destroy a vap instance created with ic_vap_create.

ic_getradiocaps
: Return the list of calibrated channels for the radio. The default method returns the current list of channels (space permitting).

ic_setregdomain
: Process a request to change regulatory state. The routine may reject a request or constrain changes (e.g. reduce transmit power caps). The default method accepts all proposed changes.

ic_send_mgmt
: Send an 802.11 management frame. The default method fabricates the frame using **IEEE80211** state and passes it to the driver through the ic_raw_xmit method.

ic_raw_xmit
: Transmit a raw 802.11 frame. The default method drops the frame and generates a message

on the console.

ic_updateslot
: Update hardware state after an 802.11 IFS slot time change. There is no default method; the pointer may be NULL in which case it will not be used.

ic_update_mcast
: Update hardware for a change in the multicast packet filter. The default method prints a console message.

ic_update_promisc
: Update hardware for a change in the promiscuous mode setting. The default method prints a console message.

ic_newassoc
: Update driver/device state for association to a new AP (in station mode) or when a new station associates (e.g. in AP mode). There is no default method; the pointer may be NULL in which case it will not be used.

ic_node_alloc
: Allocate and initialize a *ieee80211_node* structure. This method cannot sleep. The default method allocates zero'd memory using malloc(9). Drivers should override this method to allocate extended storage for their own needs. Memory allocated by the driver must be tagged with M_80211_NODE to balance the memory allocation statistics.

ic_node_free
: Reclaim storage of a node allocated by ic_node_alloc. Drivers are expected to *interpose* their own method to cleanup private state but must call through this method to allow **IEEE80211** to reclaim it's private state.

ic_node_cleanup
: Cleanup state in a *ieee80211_node* created by ic_node_alloc. This operation is distinguished from ic_node_free in that it may be called long before the node is actually reclaimed to cleanup adjunct state. This can happen, for example, when a node must not be reclaimed due to references held by packets in the transmit queue. Drivers typically interpose ic_node_cleanup instead of ic_node_free.

ic_node_age
: Age, and potentially reclaim, resources associated with a node. The default method ages frames on the power-save queue (in AP mode) and pending frames in the receive reorder

queues (for stations using A-MPDU).

ic_node_drain
: Reclaim all optional resources associated with a node. This call is used to free up resources when they are in short supply.

ic_node_getrssi
: Return the Receive Signal Strength Indication (RSSI) in .5 dBm units for the specified node. This interface returns a subset of the information returned by ic_node_getsignal. The default method calculates a filtered average over the last ten samples passed in to ieee80211_input(9) or ieee80211_input_all(9).

ic_node_getsignal
: Return the RSSI and noise floor (in .5 dBm units) for a station. The default method calculates RSSI as described above; the noise floor returned is the last value supplied to ieee80211_input(9) or ieee80211_input_all(9).

ic_node_getmimoinfo
: Return MIMO radio state for a station in support of the IEEE80211_IOC_STA_INFO ioctl request. The default method returns nothing.

ic_scan_start*
: Prepare driver/hardware state for scanning. This callback is done in a sleepable context.

ic_scan_end*
: Restore driver/hardware state after scanning completes. This callback is done in a sleepable context.

ic_set_channel*
: Set the current radio channel using *ic_curchan*. This callback is done in a sleepable context.

ic_scan_curchan
: Start scanning on a channel. This method is called immediately after each channel change and must initiate the work to scan a channel and schedule a timer to advance to the next channel in the scan list. This callback is done in a sleepable context. The default method handles active scan work (e.g. sending ProbeRequest frames), and schedules a call to ieee80211_scan_next(9) according to the maximum dwell time for the channel. Drivers that off-load scan work to firmware typically use this method to trigger per-channel scan activity.

ic_scan_mindwell

Handle reaching the minimum dwell time on a channel when scanning. This event is triggered when one or more stations have been found on a channel and the minimum dwell time has been reached. This callback is done in a sleepable context. The default method signals the scan machinery to advance to the next channel as soon as possible. Drivers can use this method to preempt further work (e.g. if scanning is handled by firmware) or ignore the request to force maximum dwell time on a channel.

ic_recv_action

Process a received Action frame. The default method points to ieee80211_recv_action(9) which provides a mechanism for setting up handlers for each Action frame class.

ic_send_action

Transmit an Action frame. The default method points to ieee80211_send_action(9) which provides a mechanism for setting up handlers for each Action frame class.

ic_ampdu_enable

Check if transmit A-MPDU should be enabled for the specified station and AC. The default method checks a per-AC traffic rate against a per-vap threshold to decide if A-MPDU should be enabled. This method also rate-limits ADDBA requests so that requests are not made too frequently when a receiver has limited resources.

ic_addba_request

Request A-MPDU transmit aggregation. The default method sets up local state and issues an ADDBA Request Action frame. Drivers may interpose this method if they need to setup private state for handling transmit A-MPDU.

ic_addb_response

Process a received ADDBA Response Action frame and setup resources as needed for doing transmit A-MPDU.

ic_addb_stop

Shutdown an A-MPDU transmit stream for the specified station and AC. The default method reclaims local state after sending a DelBA Action frame.

ic_bar_response

Process a response to a transmitted BAR control frame.

ic_ampdu_rx_start

Prepare to receive A-MPDU data from the specified station for the TID.

ic_ampdu_rx_stop
Terminate receipt of A-MPDU data from the specified station for the TID.

Once the **IEEE80211** layer is attached to a driver there are two more steps typically done to complete the work:

1. Setup "radiotap support" for capturing raw 802.11 packets that pass through the device. This is done with a call to ieee80211_radiotap_attach(9).

2. Do any final device setup like enabling interrupts.

State is torn down and reclaimed with a call to **ieee80211_ifdetach**(). Note this call may result in multiple callbacks into the driver so it should be done before any critical driver state is reclaimed. On return from **ieee80211_ifdetach**() all associated vaps and ifnet structures are reclaimed or inaccessible to user applications so it is safe to teardown driver state without worry about being re-entered. The driver is responsible for calling if_free(9) on the ifnet it allocated for the physical device.

DRIVER CAPABILITIES

Driver/device capabilities are specified using several sets of flags in the *ieee80211com* structure. General capabilities are specified by *ic_caps*. Hardware cryptographic capabilities are specified by *ic_cryptocaps*. 802.11n capabilities, if any, are specified by *ic_htcaps*. The **IEEE80211** layer propagates a subset of these capabilities to each vap through the equivalent fields: *iv_caps*, *iv_cryptocaps*, and *iv_htcaps*. The following general capabilities are defined:

IEEE80211_C_STA	Device is capable of operating in station (aka Infrastructure) mode.
IEEE80211_C_8023ENCAP	Device requires 802.3-encapsulated frames be passed for transmit. By default **IEEE80211** will encapsulate all outbound frames as 802.11 frames (without a PLCP header).
IEEE80211_C_FF	Device supports Atheros Fast-Frames.
IEEE80211_C_TURBOP	Device supports Atheros Dynamic Turbo mode.
IEEE80211_C_IBSS	Device is capable of operating in adhoc/IBSS mode.
IEEE80211_C_PMGT	Device supports dynamic power-management (aka power save) in station mode.
IEEE80211_C_HOSTAP	Device is capable of operating as an Access Point in Infrastructure mode.

IEEE80211_C_AHDEMO Device is capable of operating in Adhoc Demo mode. In this mode the device is used purely to send/receive raw 802.11 frames.

IEEE80211_C_SWRETRY Device supports software retry of transmitted frames.

IEEE80211_C_TXPMGT Device support dynamic transmit power changes on transmitted frames; also known as Transmit Power Control (TPC).

IEEE80211_C_SHSLOT Device supports short slot time operation (for 802.11g).

IEEE80211_C_SHPREAMBLE
Device supports short preamble operation (for 802.11g).

IEEE80211_C_MONITOR Device is capable of operating in monitor mode.

IEEE80211_C_DFS Device supports radar detection and/or DFS. DFS protocol support can be handled by **IEEE80211** but the device must be capable of detecting radar events.

IEEE80211_C_MBSS Device is capable of operating in MeshBSS (MBSS) mode (as defined by 802.11s Draft 3.0).

IEEE80211_C_WPA1 Device supports WPA1 operation.

IEEE80211_C_WPA2 Device supports WPA2/802.11i operation.

IEEE80211_C_BURST Device supports frame bursting.

IEEE80211_C_WME Device supports WME/WMM operation (at the moment this is mostly support for sending and receiving QoS frames with EDCF).

IEEE80211_C_WDS Device supports transmit/receive of 4-address frames.

IEEE80211_C_BGSCAN Device supports background scanning.

IEEE80211_C_TXFRAG Device supports transmit of fragmented 802.11 frames.

IEEE80211_C_TDMA Device is capable of operating in TDMA mode.

The follow general crypto capabilities are defined. In general **IEEE80211** will fall-back to software

support when a device is not capable of hardware acceleration of a cipher. This can be done on a per-key basis. **IEEE80211** can also handle software Michael calculation combined with hardware AES acceleration.

IEEE80211_CRYPTO_WEP
: Device supports hardware WEP cipher.

IEEE80211_CRYPTO_TKIP
: Device supports hardware TKIP cipher.

IEEE80211_CRYPTO_AES_OCB
: Device supports hardware AES-OCB cipher.

IEEE80211_CRYPTO_AES_CCM
: Device supports hardware AES-CCM cipher.

IEEE80211_CRYPTO_TKIPMIC
: Device supports hardware Michael for use with TKIP.

IEEE80211_CRYPTO_CKIP
: Devices supports hardware CKIP cipher.

The follow general 802.11n capabilities are defined. The first capabilities are defined exactly as they appear in the 802.11n specification. Capabilities beginning with IEEE80211_HTC_AMPDU are used solely by the **IEEE80211** layer.

IEEE80211_HTCAP_CHWIDTH40
: Device supports 20/40 channel width operation.

IEEE80211_HTCAP_SMPS_DYNAMIC
: Device supports dynamic SM power save operation.

IEEE80211_HTCAP_SMPS_ENA
: Device supports static SM power save operation.

IEEE80211_HTCAP_GREENFIELD
: Device supports Greenfield preamble.

IEEE80211_HTCAP_SHORTGI20
: Device supports Short Guard Interval on 20MHz channels.

IEEE80211_HTCAP_SHORTGI40
: Device supports Short Guard Interval on 40MHz channels.

IEEE80211_HTCAP_TXSTBC
: Device supports Space Time Block Convolution (STBC) for transmit.

IEEE80211_HTCAP_RXSTBC_1STREAM
: Device supports 1 spatial stream for STBC receive.

IEEE80211_HTCAP_RXSTBC_2STREAM
: Device supports 1-2 spatial streams for STBC receive.

IEEE80211_HTCAP_RXSTBC_3STREAM
: Device supports 1-3 spatial streams for STBC receive.

IEEE80211_HTCAP_MAXAMSDU_7935
: Device supports A-MSDU frames up to 7935 octets.

IEEE80211_HTCAP_MAXAMSDU_3839
: Device supports A-MSDU frames up to 3839 octets.

IEEE80211_HTCAP_DSSSCCK40
: Device supports use of DSSS/CCK on 40MHz channels.

IEEE80211_HTCAP_PSMP
: Device supports PSMP.

IEEE80211_HTCAP_40INTOLERANT
: Device is intolerant of 40MHz wide channel use.

IEEE80211_HTCAP_LSIGTXOPPROT
: Device supports L-SIG TXOP protection.

IEEE80211_HTC_AMPDU
: Device supports A-MPDU aggregation. Note that any 802.11n compliant device must support A-MPDU receive so this implicitly means support for *transmit* of A-MPDU frames.

IEEE80211_HTC_AMSDU
: Device supports A-MSDU aggregation. Note that any 802.11n compliant device must support A-MSDU receive so this implicitly means support for *transmit* of A-MSDU frames.

IEEE80211_HTC_HT Device supports High Throughput (HT) operation. This capability must be set to enable 802.11n functionality in **IEEE80211**.

IEEE80211_HTC_SMPS Device supports MIMO Power Save operation.

IEEE80211_HTC_RIFS Device supports Reduced Inter Frame Spacing (RIFS).

SEE ALSO

ioctl(2), ieee80211_amrr(9), ieee80211_beacon(9), ieee80211_bmiss(9), ieee80211_crypto(9), ieee80211_ddb(9), ieee80211_input(9), ieee80211_node(9), ieee80211_output(9), ieee80211_proto(9), ieee80211_radiotap(9), ieee80211_regdomain(9), ieee80211_scan(9), ieee80211_vap(9), ifnet(9), malloc(9)

HISTORY

The **IEEE80211** series of functions first appeared in NetBSD 1.5, and were later ported to FreeBSD 4.6. This man page was updated with the information from NetBSD **IEEE80211** man page.

AUTHORS

The original NetBSD **IEEE80211** man page was written by Bruce M. Simpson <*bms@FreeBSD.org*> and Darron Broad <*darron@kewl.org*>.