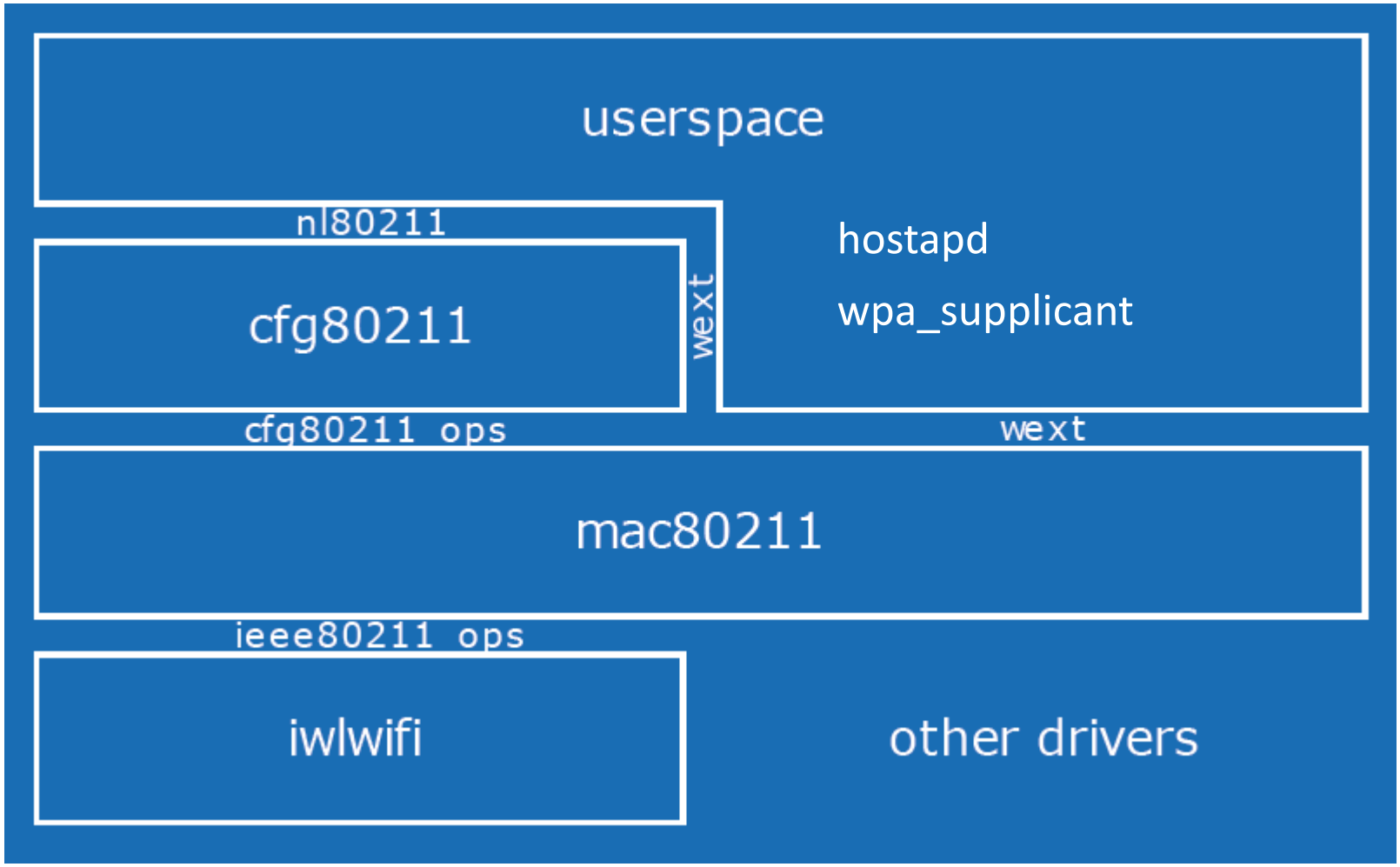
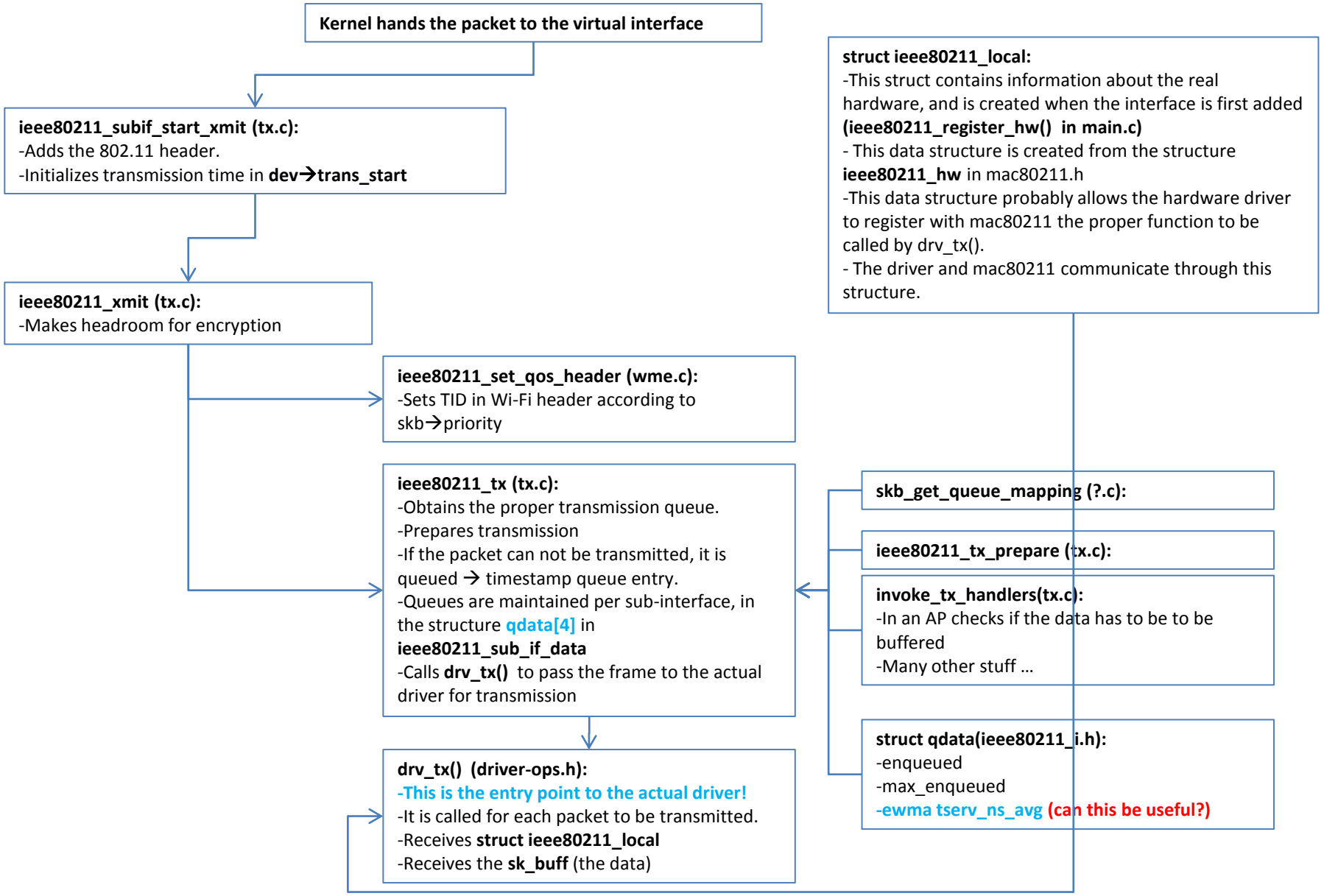
Linux Wi-Fi open source drivers -mac80211, ath9k/ath5k-

Daniel Camps Mur

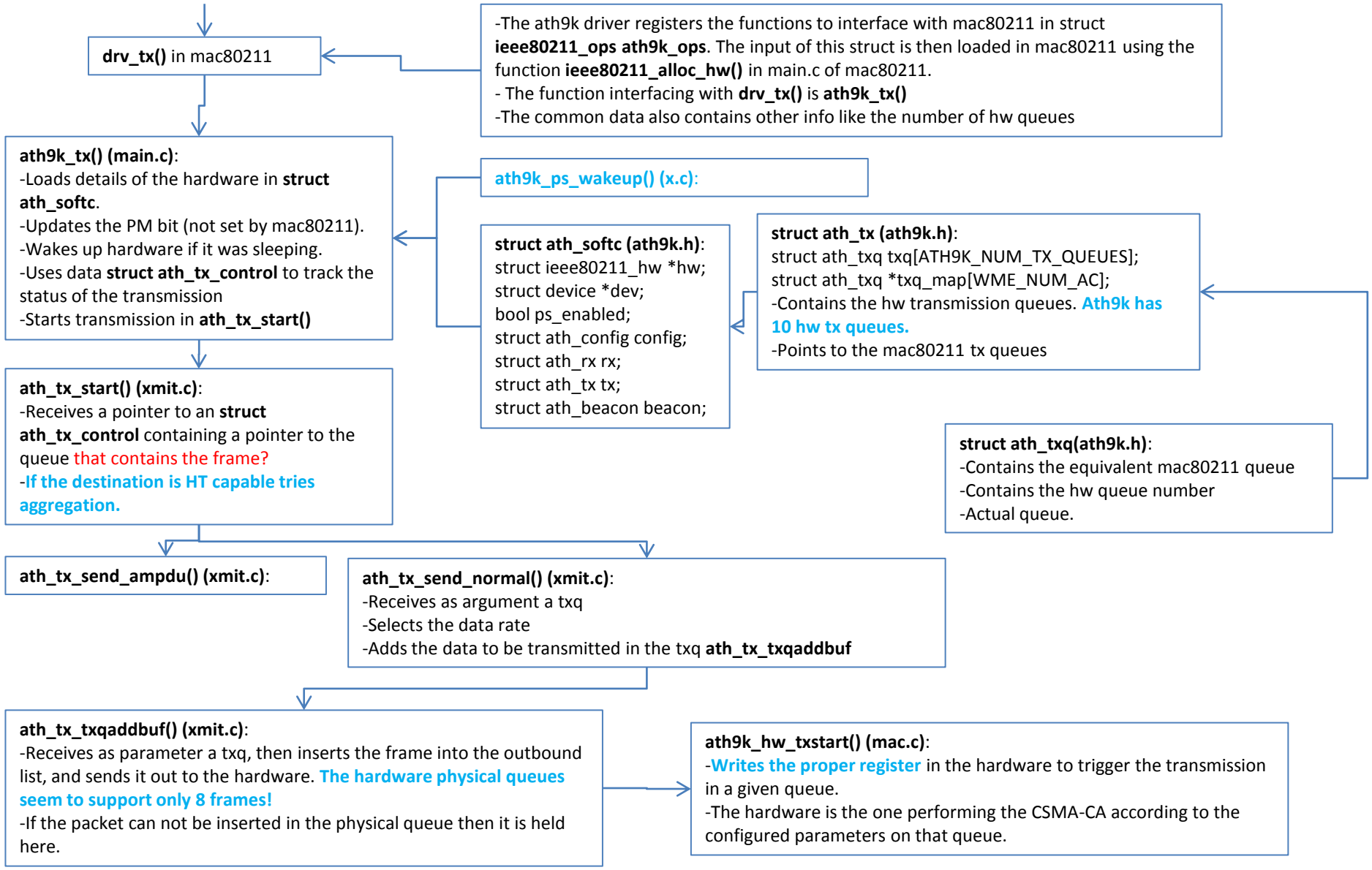
1. General Driver Overview



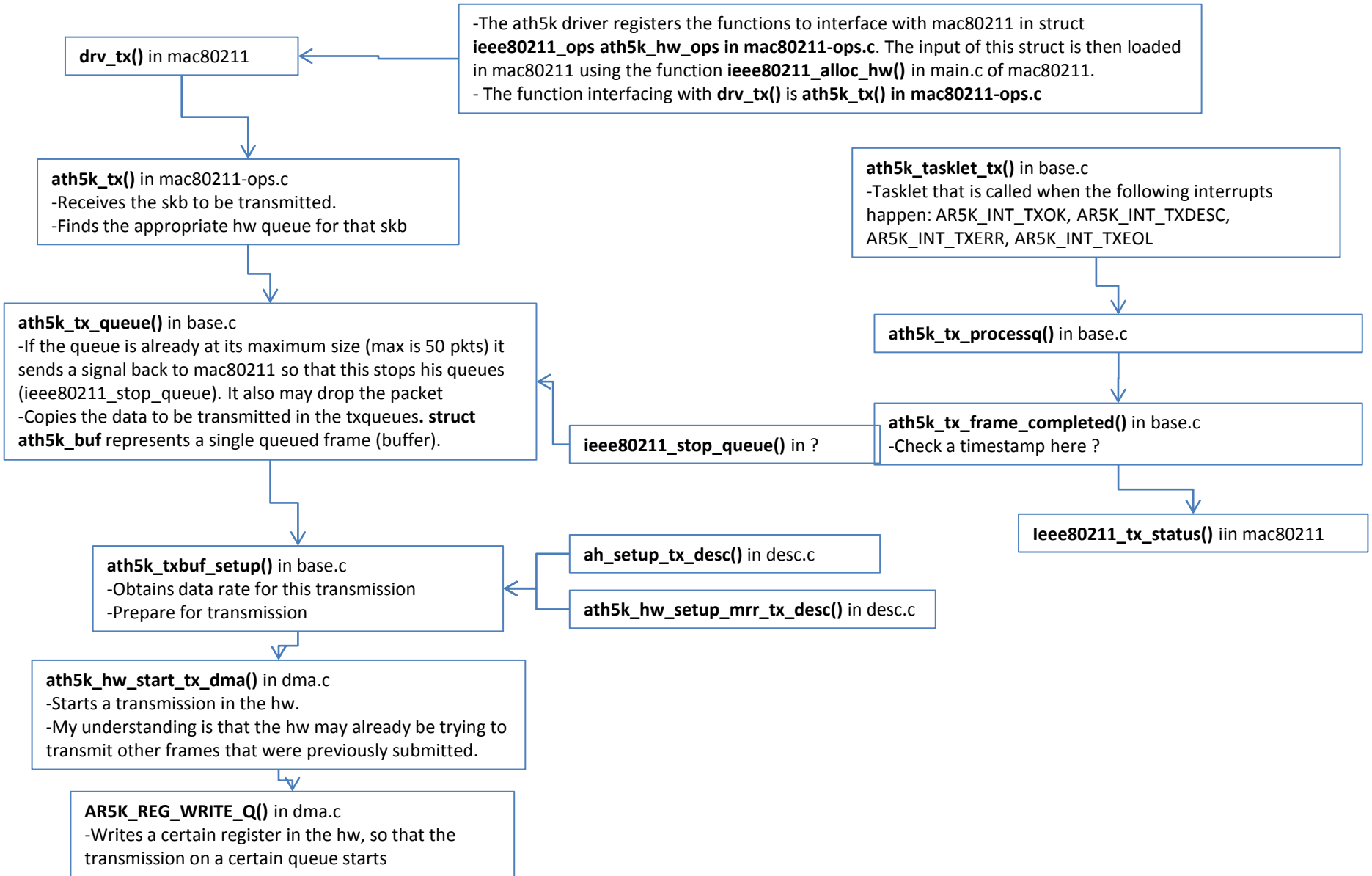
1. Transmission Path: kernel→mac80211→ath9k



1a. Transmission Path: ath9k→hardware



1b. Transmission Path: ath5k→hardware



1a. Reception Path: hardware→ath9k

When receiving a packet, and also for other reasons, the hardware sends an interruption that ath9k has previously registered. The function in charge of handling the interruption seems to be **irqreturn_t ath_isr** in main.c. This function discovers the type of interruption and asks the kernel to schedule the execution of a tasklet, **ath9k_tasklet** in main.c. This function in turn calls the the receive tasklet **ath_rx_tasklet()**

struct ieee80211_hdr (include/linux/ieee80211.h)

struct ieee80211_rx_status (mac80211.h)
u64 mactime : value in microseconds of the 64-bit Time Synchronization Function (TSF) timer when the first data symbol (MPDU) arrived at the hardware.
enum ieee80211_band band;
int rate_idx;
unsigned int rx_flags; …

struct ath_rx_status (mac.h)
u32 rs_tstamp; This is given by the hardware and eventually carried on to mactime in ieee80211_rx_status
u16 rs_datalen;
u8 rs_status;
u8 rs_phyerr;
int8_t rs_rssi;
u8 rs_keyix;
u8 rs_rate;
u8 rs_antenna;
…

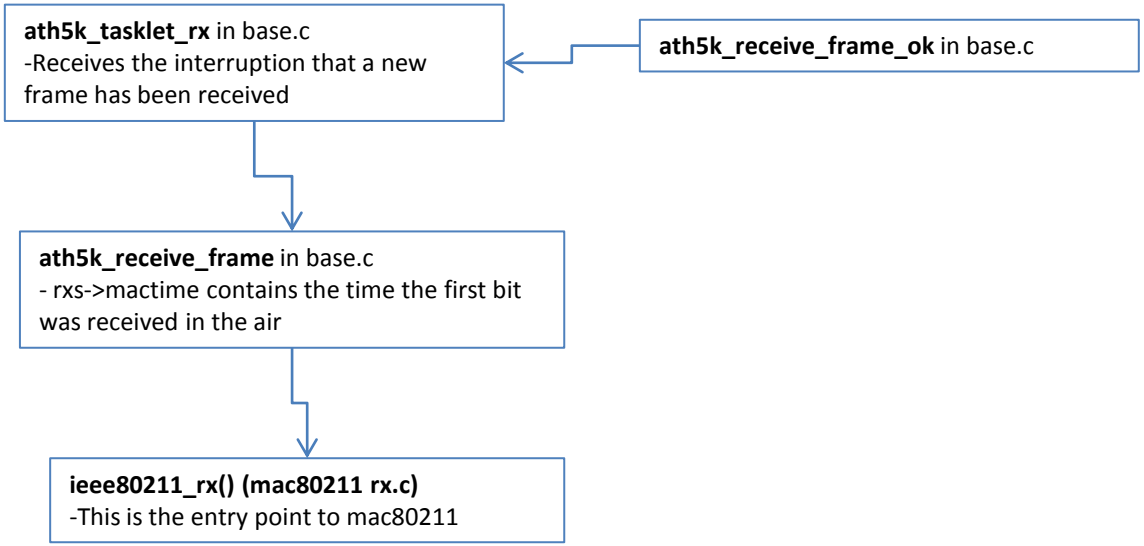
ath_rx_tasklet (recv.c)
-Obtains the frame header
-Obtains the current tsf value
-Records info about received packet in **struct ieee80211_rx_status**
-Insert received data in the receive buffer
-Creates a new skb to contain the received data
-Passes the skb to **ieee80211_rx()**
-Can we compute the airtime duration of the received frame in this function, as "airtime = tsf - rxs->mactime" ?

ath9k_rx_skb_preprocess (recv.c)
-Checks if the received data has CRC errors and in that case drops it. However, crypto errors are still passed up to mac80211
-Populates **ath_rx_status**

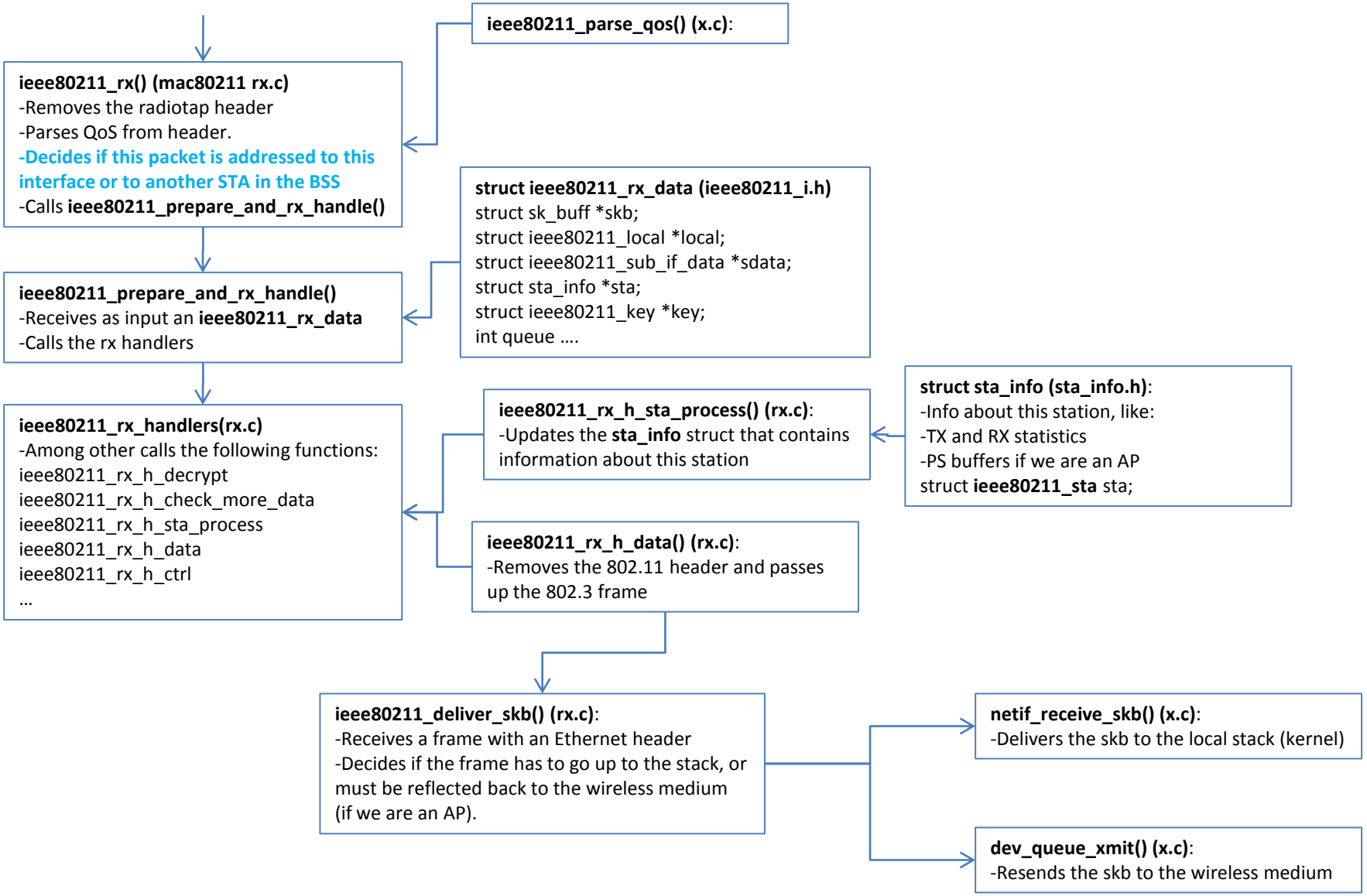
ath9k_rx_skb_postprocess (recv.c)
-Remove padding from the received header

ieee80211_rx() (mac80211 rx.c)
-This is the entry point to mac80211

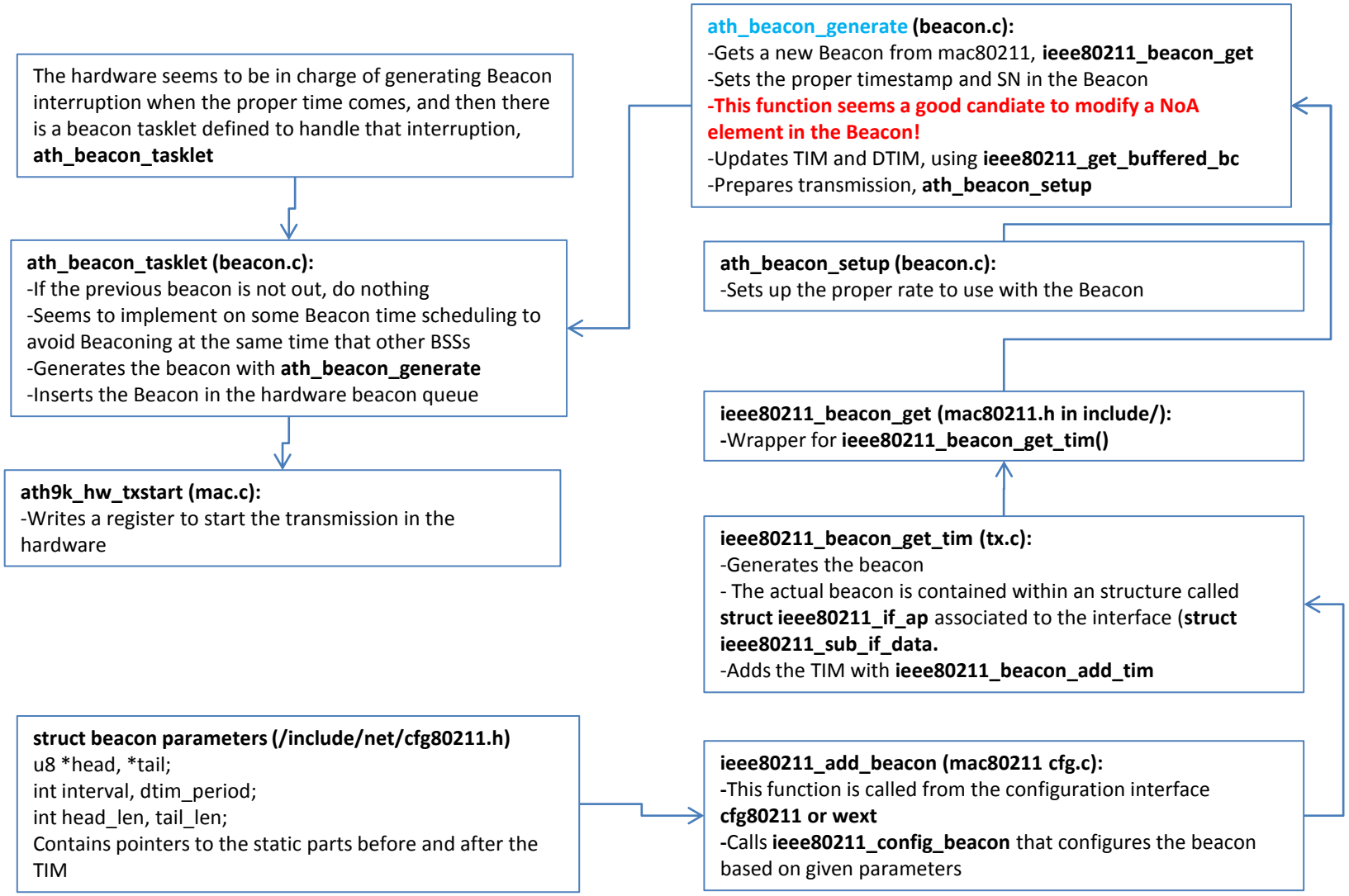
1b. Reception Path: hardware→ath5k



1. Reception Path: mac80211→kernel



1. Beacon Tx path: from ath9k to mac80211



But where is the Beacon actually constructed ? This is done by hostapd, next …

1. Beacon set up: hostapd/wpa_supplicant

- hostpad creates all the STATIC template of the Beacon frame (i.e., SSID, supported rates, …) and then passes it down to mac80211.
- The low level driver (ath9k) is the one in control of the DYNAMIC parts of the Beacon (SNs, Timestamp, TIM).

ieee802_11_set_beacon (beacon.c)
-This function allocates all the IEs in the HEAD and TAIL parts of the Beacon, which then will be used by mac80211 to construct the Beacon and pass it down to the driver.
-In P2P mode hostapd adds to the Beacon a P2P IE and calls the function **hostapd_eid_p2p_manage**

hostapd_eid_p2p_manage (p2p_hostapd.c)
-This function builds the P2P IE that goes in the Beacon
-Right now the function only adds the P2P Manageability element within the P2P IE
-We need to modify this function to also add a Notice of Absence Element in the Beacon with one NoA Descriptor.
-Then in ath_beacon_generate() from ath9k we have to be able to access the memory allocated to the NoA element and overwrite it with the duration/interval values computed by our algorithm.

- When including the NoA element in the Beacon, we should reuse the definition already provided by hostap. Note that the current P2P implementation can already send a P2P Presence Request frame which contains NoA descriptors.

p2p_build_presence_req (hostap/src/p2p.c)
-Build a presence request which is an Action frame that includes a NoA IE

p2p_add_noa (hostap/src/p2p_build.c)
-Adds an NoA element to a **struct wpa_buf**

struct p2p_noa_desc (p2p_i.h)
u8 count_type;
u32 duration;
u32 interval;
u32 start_time;

1. Beacon Rx path: mac80211