VIETNAM NATIONAL UNIVERSITY – HOCHIMINH CITY
THE INTERNATIONAL UNIVERSITY
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Atheros ath5k wireless driver

Supervisor: Quan Le-Trung, Dr.techn

Student: Le Hoang Phuong – ITIU08029

A thesis submitted to the School of Computer Science and Engineering in partial fulfillment of the requirements for the degree of

Bachelor of Computer Science

Ho Chi Minh City, Viet Nam

2012

Atheros ath5k wireless driver

APPROVE BY

_____

Quan Le Trung, Dr Techn (Supervisor)

_____

_____

_____

_____

THESIS COMMITEE

Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor Quan Le-Trung, Dr.Techn for his patience, motivation, enthusiasm, and immense knowledge. His constant encouragement and support helped me to achieve my thesis.

Besides my advisor, my gratitude goes to the ath5k development team. Their experience and guidance give me a good starting point in this document.

My sincere thanks also go to the other lecturers of School of Computer Science and Engineering. Their experiences give me a treasure support since the beginning of my studies.

Last but not the least, I am grateful to my parent for supporting throughout my life.

Table of Contents

LIST OF FIGURES

ABSTRACT

Wireless is becoming the most popular type of communication nowadays. There are a variety of wireless devices in the market. The diversity of wireless devices leads to the incompatibility of wireless device drivers on different platforms. Fortunately, on open-source platform like Linux, this problem is solved and supported through open-source projects. In this thesis, we are going to explore such an open-source project: ath5k driver for WLAN card using Atheros chipsets. The focus is three-fold. Firstly, operations of WLAN MAC layer (IEEE 802.11), both in the infrastructure mode and the infrastructure-less (ad-hoc) mode, are described and analyzed. Secondly, the source code of ath5k device driver is analyzed and discussed on different characteristics. These ones include: i) data structures to be used in the ath5k, ii) reception and transmission packet flows, iii) configuration flow. Finally, realistic scenarios are presented to illustrate for the operations of ath5k device driver, and to collect statistical wireless information.

LIST OF ACRONYMS

ASIC: Application-Specific Integrated Circuit
API: Application Program Interface
PCI: Peripheral Component Interconnect
ISA: Industry Standard Architecture
TX: Transmit
RX: Receive
DMA: Direct Memory Access
AHB bus (Advanced High-performance Bus)
IMR: Interrupt mask register
ISR: Interrupt status registers
VEOL: Virtual End of List

I. WIRELESS DEVICE

1. Introduction

Firstly we need to know what wireless devices are. A wireless device is a device using radio network to perform operation. Radio network is a kind of network which participants are mobility nodes. In radio network, radio wave is used to transmit and receive data. So what are wireless devices? They are usually cards that can be plugged in computers or mobile devices.

2. Components of A Wireless Device

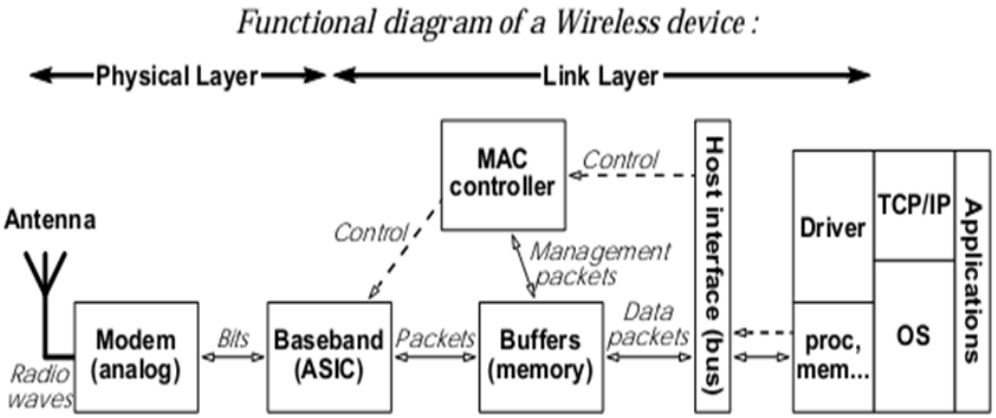


Figure 1: Components of a wireless device [1]

A wireless device is made from the following components:

- The radio modem includes antennas and modem. The task of radio modem is transmitting and receiving radio wave.
- The MAC controller controls how to send, receive and forward packets.
- Baseband is the chip which has the responsibility to perform MAC controller's operations.
- Buffers are additional components of MAC controller that store temporary packets.
- Host interface is position where the wireless card is plugged into device.
- Finally, a wireless device needs a "driver" to operate expectedly.

More details of a wireless device's components:

a. Radio Modem

Radio modem is the part which modulates data to the radio signal, transmits it and receives other transmissions. Radio modem is a combination of multiple components such as antennas, amplifications, frequency synthesizers, filters...The main characteristics of radio modems are frequency band, signal rate, modulation and transmitted power.

b. *MAC Controller*

"MAC controller runs *MAC protocol* which is used to provide *addressing* and *channel access control mechanisms*. MAC protocol makes it possible for several terminals or network nodes to communicate within a multiple access network that incorporates a shared medium". [2]

MAC controller is implemented mainly in an application-specific integrated circuit (ASIC) and/or a microcontroller on the card [1]. Most of critical functions are handled by the baseband of the radio modem. The card has some free memory (calling "buffer") for the MAC controller to temporarily store in-coming, out-going packets and data. This buffer is quite important because the MAC controller may need it to "compensate the PC and interface latency" [1]. Some management functions may be implemented in the card's driver.

The main characteristics of the MAC controller are *packet format*, *the channel access mechanisms* and *network management*.

c. *The host interface*

A card can be connected to a PC through its buses (ISA, PCI…) or through communication ports (USB, Ethernet…). This host interface allows the software (including the end-user applications and the drivers) to communicate with the MAC controller. Communication is done by the "buffer". Software writes packets to the buffer, the MAC controller reads and sends them.

The main characteristics of the host interface are the speed, the ability to process requests in parallel.

d. *The driver*

The end-user software does not directly communicate with the hardware. It needs a standard *application program interface* (API) to handle this. The driver will help the operating system to list the hardware into network standards.

The main functions of the driver are managing the hardware, answering the requests. In wireless devices, the driver sometimes implements some MAC's functions.

II. MAC CONTROLLER

In this section, I am going to introduce some mechanisms of MAC controller such as: sending, receiving and forwarding packets through MAC controller in the Link Layer. "The main job of the MAC protocol is to regulate the usage of the medium, and this is done through *a channel access mechanism*" [1]. A channel access mechanism is the core of the MAC protocol. It ensures the main resources to be divided equally to all nodes. It also manages all sending, receiving and forwarding packets of all nodes. Here are three main classes of channel access mechanisms: TDMA, CSMA and Polling.

1. TDMA (Time Division Multiplex Access)

- The idea of TDMA is that a specific node (calling *a base station*) controls the operation of all other nodes. It has the responsibility to coordinate operation of all nodes. The channel is divided into time slots. These slots are fixed size. They are performed as frames and repeated over time. Each time slot is the time for a node to transmit data.
- How a node can take control the medium for transmission? It is very simple. Every node can get a certain number of slots where it can transmit data. The base station will give instructions for all nodes. Each node just follows exactly these instructions.
- The base station is performed into *a management frame* which is named "*beacon*" (Figure 2). Nodes not only follow instructions but also give request to get a connection. This is happen through *a service slot* in the medium which listens and receives request messages from nodes. The TDMA frame is organized as downlink (base station communicate to nodes) and uplink (nodes communicate to base station). Downlink and uplink are usually in different frequencies. The service slots may also be a separate channel.

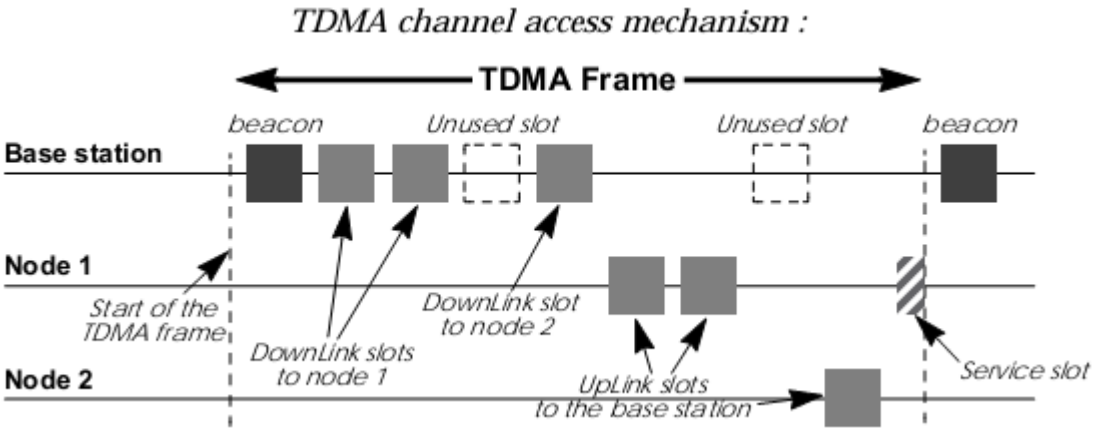


Figure 2: Operation of TDMA [1]

Advantages and Disadvantages of TDMA:

- TDMA is a connection oriented. It is suitable for phone network and phone applications because we can predict requirements for phones (fixed and identical bit rate for each cell phone).
- "TDMA is also very good to achieve low latency and guarantee of bandwidth" [1]. Since the base station manage all transmitting fairly between all nodes.
- There are also some disadvantages. TDMA is not very well to serve network applications. Those are very strict and inflexible applications. TCP/IP communication is un-predictable and generates bursting traffic while TDMA is connection oriented. TCP/IP communication also contains variable size packets which can be served by TDMA (using fixed size packets and symmetrical link).

2. CSMA/CA (Carrier Sense Multiple Avoidance/ Collision Avoidance)

- CSMA is used by most of modern Wireless LAN. CSMA specifies instructions for all nodes how to use the medium. The main ideas of CSMA/CA are "listening before talking" and "contention". This method is using *asynchronous messages passing* mechanism.
- CSMA/CA has the origin from CSMA/CD which is the base of Ethernet network. The main difference between these two mechanisms is CA "collision avoidance" and CD "collision detection". On the wire network, a sender can listen to whole network to detect the collision because transmissions have the same strength. In wireless network, this job is impossible. A node can still listen to the channel while transmitting but its own transmission can mask all other signals on the air. In conclusion, collision can be detected and protocol just tries to avoid it.

This is how the protocol works:

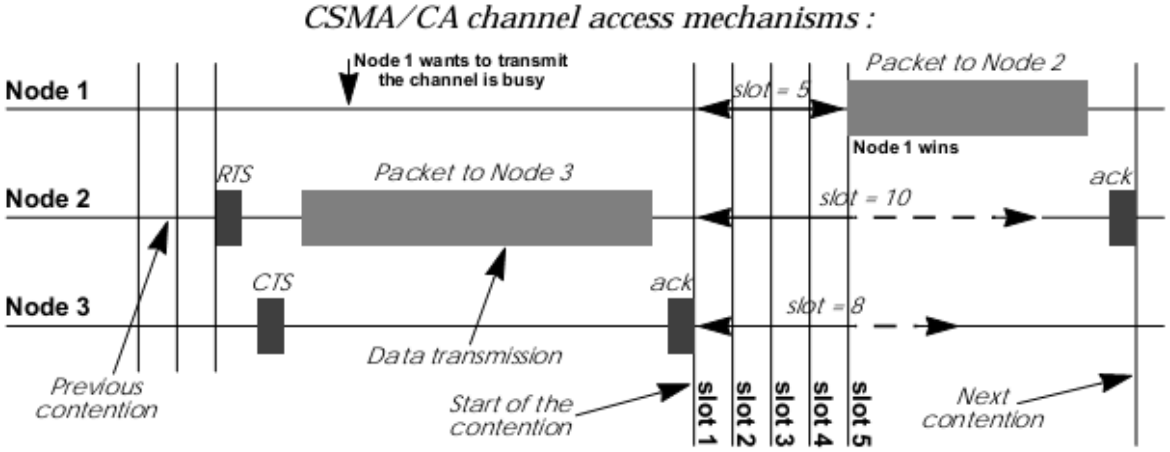


Figure 3: Operation of CSMA/CA [1]

A node begins its transmission by listening on channel (calling "carrier sense"). If channel is idle, node sends the first packet through the channel. If channel is busy (may be transmission of another node or the channel's interference), node will wait until the end of that transmission or no interference. It now starts the "contention". Every node picks up a random amount of time (calling contention time). When this time expires, if the channel is idle, the node sends the packet. The node, which can transmit its packet, is the node with the shortest contention time. The other nodes have to start again the contention when the transmission is over. Because the contention time is a random number, all nodes have the same probability to access the channel for transmission.

Advantages and disadvantages of CSMA/CA:

- "CSMA/CA suites well for TCP/IP network protocol. It adapts quite well with the variable condition of traffic and is quite robust against interferences" [1]. It can help to avoid collision on wireless network.
- But it also has some disadvantages. Firstly, CSMA/CA has no guarantee bandwidth since it has no a base station to manage transmission of nodes. Secondly, CSMA/CA can cause the large latency since it has a contention time which nodes have to wait for permission to access the channel.

3. Polling MAC

- The third channel access mechanism is polling. Polling actually stands between TDMA and CSMA/CA.
- Polling can work as a connection oriented. It works nearly the same with TDMA. It has a base station to control over the channel but the frame is no more fixed size. Variable size packets can transmit through the channel. A node just has to wait for a specific packet (naming "poll packet") to know that it now has the permission to send packets.
- Polling can also works as a connectionless. It can permanently poll all the nodes of the network to check if they have something to send. Or it can use reservation slots where each node can request a connection or to transmit a packet.

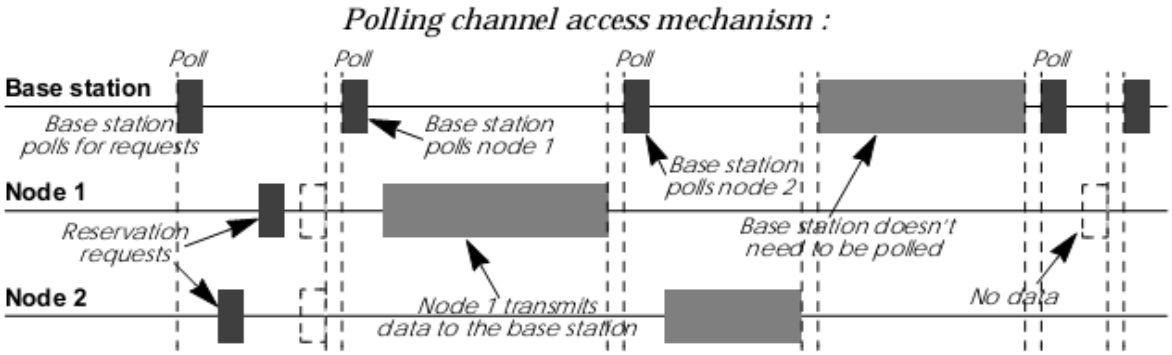


Figure 4: Operation of Polling MAC [1]

MAC protocol uses some additional techniques to help improving the performance of the channel access mechanism.

4. MAC retransmission

- In wireless network, there is a higher error-rate than in wire network. That is a high chance of packets being corrupted. To avoid losing packets on the air, MAC protocol also implement positive acknowledgement and MAC level transmissions.
- The idea is very simple. Each time a node receives a packet, it sends an ACK back to the transmitter to notice that it has received successfully the packet without errors. If after sending a packet, the transmitter does not receive an ACK, it knows that the packet was lost. It will automatically re-transmit the packet. Most MAC protocols use a stop and go mechanism. It means that they only transmit the next packet if the last packet has sent successfully (they got an ACK back). It is not the same with *window sliding mechanism* in TCP.
- The ACKs are embedded in the MAC protocol. So they guarantee not to collide with the contentions. The contentions start after the ACKs.

5. Fragmentation

- The MAC retransmission can help to solve packet being corrupted but their performance is not really good. If the transmitting packet is too large and contains only one error, the sender needs to transmit it entirely. This is why fragmentation being used. The principle is sending the big packets in small pieces over the medium.
- “There are lots of advantages of fragmentation. Firstly, in case of error the sender needs only to re-transmit one small fragment, so it is faster. Secondly, if the medium is very noisy, sending a small packet has a higher probability to get through without errors, the node increases its chance of success in bad conditions”. [1]

- Of course, no protocols only have advantages. The disadvantages of fragmentation are adding some overhead, because it duplicates packet headers in every fragment.

6. RTS/CTS

- One problem of transmission on radio waves is the attenuation of the signal. Because of this attenuation, we have a very common problem "*hidden nodes*".
- Hidden nodes in a wireless network refer to nodes that are out of range of other nodes or a collection of nodes. Take a physical star topology with an access point with many nodes surrounding it in a circular fashion: each node is within communication range of the AP, but the nodes cannot communicate with each other, as they do not have a physical connection to each other (Figure 5). Because transmissions are based on the carrier sense mechanism, those nodes ignore each other and may transmit at the same time. [1]

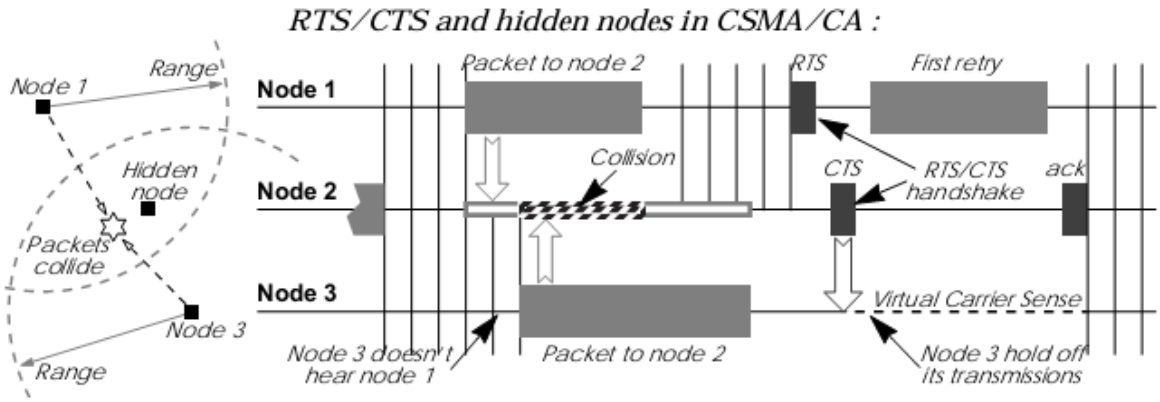


Figure 5: Hidden nodes and operation of RTS/CTS [1]

- The principle to solve this problem is to use RTS/CTS. RTS/CTS are a handshaking process between a sending node and a receiving node. Before sending a packet, the sender sends a RTS and wait for the CTS from the receiver. When sender receives CTS, it understands that receiver is able to receive and accept the RTS. Sender starts transmitting.
- At the same time, every node in the range of the receiver can hear the CTS. It indicates that the transmitting is going on. The nodes hearing CTS are the nodes that can temporarily create collisions on the receiver. But after hearing the CTS, they will avoid accessing the channel even if their carrier sense indicates that the medium is free. Only the node that receives the CTS is able to send data.
- RTS/CTS help to lowers the overhead of the collisions on the medium. If two nodes attempt to transmit in the same slot of the contention

window, their RTS collide and they don't receive any CTS, so they lose only a RTS, whereas in the normal scenario they would have lost a whole packet. Because the RTS/CTS handshaking adds a significant overhead, usually it is not used for small packets or lightly loaded networks

7. Reservation and service slots

- The main problem of TDMA and Polling is that the base station does not know when the nodes want to transmit data. In CSMA/CA, nodes just wait to win a contention for transmitting. However, TDMA and Polling use a mechanism to solve this: service slot or reservation slot.
- The mechanism offers a period of time where nodes can send request to the base station about their traffic reservations. The base station uses its algorithm to decide when each node will have permission to transmit. This sending reservation time is called reservation slot or service slot.

8. Forwarding packets

- Forwarding mechanism is implemented in MAC protocol (name MAC level forwarding). Every node in the network can be used to replay the message on the air to the destination. This protocol does not rely on a fixed infrastructure but on all nodes on the path. How a node can find the optimal route? Routing protocols are used to solve this problem.
- IP routing protocols can be broadly classified into "source routing" or "next-hop routing" protocols. In source routing, the sender of a packet specifies the route that the packet must traverse to reach its final destination. Dynamic source routing (DSR) is proposing in [3] in which the sender node floods a route request (RREQ) probe in search of a route to the destination. Intermediate nodes that forward this request probe, append their identifiers to the probe. The probe arriving firstly at the destination is assumed to arrive on the optimal path. DSR uses this path for subsequent communication.
- In "next-hop routing", the forwarding mechanism uses management message to propagate network changes and topology information. From these messages, a node can calculate the optimal forwarding tables. Routing tables are the main keys for the forwarding mechanism, all actions base on these routing tables. In fact, every node of the network acts as an ad-hoc bridge.

III. ATHEROS ATH5K WIRELESS DRIVER

In this Section, I am going to explore an open-source project: ath5k driver for WLAN card using Atheros chipsets [4]. I am going to cover a specific topic: how to send/receive data, configure parameters of a wireless card using ath5k driver.

1. History of ath5k

Atheros chipsets have always been heavily favored by the open-source community because of their extensibility and because they are found in high-end cards provided by many manufacturers. ath5k is a completely *free and open source software* (FOSS) Linux driver for Atheros wireless cards. It is based on MadWifi [5] and the OpenHAL [6]. It is stable enough to be included in the Linux kernel, and like all modern wireless drivers on Linux, it uses mac80211. ath5k provides support for many devices that utilize the AR5XXX family of chipsets; however, it provides no USB support and no 802.11n support.

2. Structure of Atheros ath5k

a. Overview

Ath5k is a Linux wireless driver that is based on the MadWiFi [5] wireless driver and OpenHAL [6]. The main difference between MadWiFi and Ath5k is that Ath5k directly calls hardware functions and writes to the hardware registers of the Atheros wireless card. In the source code folder of ath5k (it is located in Linux kernel source code [7], folder path is **linux-…\driver\net\wireless\ath\ath5k**), there are 30 files:

ath5k.h: defines the structure of the hardware abstraction layer and contains the settings of the driver, like transmission rate, reception status, and driver mode…etc.

ahb.c: support for the AHB bus (Advanced High-performance Bus) got merged. Now ath5k can be used on AR231X and AR5312 embedded devices

ani.c, **ani.h**: perform Adaptive Noise Immunity functions which control noise immunity parameter depending on the amount of interference in the environment, increasing or reducing sensitivity as necessary.

attach.c: Attach/Detach Functions and helpers

base.c, **base.h**: main files of ath5k, contain functions that are responsible for the transmission of packets, reception of packets, driver initialization etc.

caps.c: fill the capabilities struct that store in the EEPROM

debug.c, **debug.h**: implement functions that use for debugging.

desc.c, **desc.h**: hardware descriptor functions which handle the processing of the low-level hardware descriptors that the card reads and writes via DMA for each TX and RX.

dma.c: DMA and interrupt masking functions which setup descriptor pointers (rxdp/txdp), start/stop DMA engine and handle queue setup for 5210 chipset (rest are handled on qcu.c). Also we setup interrupt mask register (IMR) and read the various interrupt status registers (ISR).

eeprom.c, **eeprom.h:** EEPROM access functions and helpers

gpio.c: GPIO/LED functions which control the 6 bidirectional GPIO pins provided by the hardware.

initvals.c: fills in the registers in the wireless card with initial values.in reg.h

led.c: control the LEDs of wireless card.

mac80211-ops.c: implements MAC80211 functions

pcu.c: Protocol control unit is responsible to maintain various protocol properties before a frame is send and after a frame is received to/from baseband

phy.c: PHY related functions which handle the low-level functions related to baseband and analog frontend (RF) parts. This is by far the most complex part of the hadrware code so make sure you know what you are doing.

qcu.c: Queue Control Unit (QCU)/DCF Control Unit (DCU) functions which setup parameters for the 12 available TX queues. Each queue has a matching QCU that controls when the queue will get triggered and multiple QCUs can be mapped to a single DCU that controls the various DFS parameters for the various queues. In our setup we have a 1:1 mapping between QCUs and DCUs allowing us to have different DFS settings for each queue. When a frame goes into a TX queue, QCU decides when it'll trigger a transmission based on various criteria (such as how many data we have inside it's buffer or -if it's a beacon queue- if it's time to fire up the queue based on TSF etc.), DCU adds back off, IFSes etc. and then a scheduler (arbitrator) decides the priority of each QCU based on its configuration (e.g. beacons are always transmitted when they leave DCU bypassing all other frames from other queues waiting to be transmitted). After a frame leaves the DCU it goes to PCU for further processing and then to PHY for the actual transmission.

reg.h: holds the values for the hardware registers of Atheros 5212, 5211, and 5210 cards and other hardware functionalities.

reset.c: reset functions which implement the main reset routine, used to bring the card to a working state and ready to receive. We also handle

routines that don't fit on other places such as clock, sleep and power control

rfbuffer.h: some special registers on the RF chip that control various operation settings related mostly to the analog parts (e.g. channel, gain adjustment etc.)

rfgain.c: RF Gain optimization

rfkill.c: RFKILL support for ath5k

trace.h: add some trace points for ath5k, these trace points can be also used for debugging.

b. Data structures

Here is a brief description of some associated structures which plays an important role in the analysis process of ath5k source code. Some data structures, which I have set in **bold** style, are the most important data structures to analyze ath5k source code.

ath5k_hw *ah: This structure represents the ath hardware. It provides "Driver state associated with an instance of a device". It contains pointers to multiple structures such as ieee80211_hw, ieee80211_channel, ath5k_desc, ath5k_txq, ath5k_txq_info, and function pointers of transmit and receive descriptors etc.

```
struct ath5k_hw {
      struct ath_common       common;

      struct pci_dev        *pdev;
      struct device         *dev;      /* for dma mapping */
      int irq;
      u16 devid;
      void __iomem          *iobase;   /* address of the device
*/
      struct mutex          lock;      /* dev-level lock */
```

struct ieee80211_hw is the most important data structure, it represents hardware information and state, driver calls function ieee80211_alloc_hw() to allocate it.

```
      struct ieee80211_hw   *hw;       /* IEEE 802.11 common */
      struct ieee80211_supported_band
sbands[IEEE80211_NUM_BANDS];
      struct ieee80211_channel channels[ATH_CHAN_MAX];
      struct ieee80211_rate
      rates[IEEE80211_NUM_BANDS][AR5K_MAX_RATES];
      s8
      rate_idx[IEEE80211_NUM_BANDS][AR5K_MAX_RATES];
      enum nl80211_iftype   opmode;    /*Operation mode*/
```

```
...

     struct ath5k_buf    *bufptr;  /* allocated buffer ptr
*/
     struct ath5k_desc   *desc;          /* TX/RX
descriptors */
     dma_addr_t      desc_daddr;     /* DMA (physical) address
*/
     size_t               desc_len;  /* size of TX/RX
descriptors */


     struct list_head    txbuf;          /* transmit buffer
*/
     spinlock_t      txbuflock;
     unsigned int         txbuf_len; /* buf count in txbuf list
*/
     struct ath5k_txq    txqs[AR5K_NUM_TX_QUEUES];     /*
tx queues */
     struct tasklet_struct txtq;      /* tx intr tasklet */

     struct ath5k_txq_info ah_txq[AR5K_NUM_TX_QUEUES];
     u32             ah_txq_status;
     u32             ah_txq_imr_txok;
     u32             ah_txq_imr_txerr;
     u32             ah_txq_imr_txurn;
     u32             ah_txq_imr_txdesc;
     u32             ah_txq_imr_txeol;
     u32             ah_txq_imr_cbrorn;
     u32             ah_txq_imr_cbrurn;
     u32             ah_txq_imr_qtrig;
     u32             ah_txq_imr_nofrm;

     u32             ah_txq_isr_txok_all;
     u32             ah_txq_isr_txurn;
     u32             ah_txq_isr_qcborn;
     u32             ah_txq_isr_qcburn;
     u32             ah_txq_isr_qtrig;

...etc.
```
(for more details, see in **\...\driver\net\wireless\ath\ath5k\ath5k.h**)
```
};
```

Hardware Descriptor:

There are two kinds of descriptors for RX and TX: control descriptor and status descriptor. They are defined in desc.h

- Firstly, control descriptors tell the wireless card how to send or receive a packet, where to read/write it from/to etc.

struct ath5k_hw_rx_ctl - Common hardware RX control descriptor

```
struct ath5k_hw_rx_ctl {
     u32   rx_control_0; //RX control word 0
     u32   rx_control_1; //RX control word 1
} __packed __aligned(4);
```

Since there are multiple types of chipset such as AR5210 / AR5211 (which using a 2-Word TX control descriptor) or AR5212 and later chips (which using a 4-Word TX control descriptor), developers have to build their own data structures for each type of chipset. Descriptor format is not exactly the same for each MAC chip version so they have function pointers on &**struct ath5k_hw** that is initialized at runtime based on the chip used.

struct ath5k_hw_4w_tx_ctl – using for 5212 wireless cards 4-word TX control descriptor

```
struct ath5k_hw_4w_tx_ctl {
     u32   tx_control_0; // TX control word 0
     u32   tx_control_1; // TX control word 1
     u32   tx_control_2; // TX control word 2
     u32   tx_control_3; // TX control word 3
} __packed __aligned(4);
```

struct ath5k_hw_2w_tx_ctl - using for 5210/5211 wireless cards 2-word TX control descriptor

```
struct ath5k_hw_2w_tx_ctl {
     u32   tx_control_0; //TX control word 0
     u32   tx_control_1; //TX control word 1
} __packed __aligned(4);
```

- Secondly, status descriptors that contain information about how the packet was sent or received (errors included).

struct ath5k_hw_rx_status: Common hardware RX status descriptor

```
struct ath5k_hw_rx_status {
     u32   rx_status_0;
     u32   rx_status_1;
} __packed __aligned(4);
```

struct ath5k_hw_tx_status - Common TX status descriptor

```
struct ath5k_hw_tx_status {
   u32   tx_status_0;
   u32   tx_status_1;
} __packed __aligned(4);
```

DMA DESCRIPTOR: use by DMA Engine (see Appendix B)

It is Atheros hardware DMA descriptor. It consists of ds_link: Physical address of the next descriptor, ds_data: Physical address of data buffer (skb), and ud: Union containing hw_5xxx_tx_desc structs and hw_all_rx_desc (since there are multiple types of data structure for each chipset family above, *ath5k_desc* combines of all type of descriptors above for easily coding). This is read and written to by the hardware.

struct ath5k_hw_5210_tx_desc - 5210/5211 hardware TX descriptor

```
struct ath5k_hw_5210_tx_desc {
      struct ath5k_hw_2w_tx_ctl   tx_ctl;
      struct ath5k_hw_tx_status   tx_stat;
} __packed __aligned(4);
```

struct ath5k_hw_5212_tx_desc - 5212 hardware TX descriptor

```
struct ath5k_hw_5212_tx_desc {
      struct ath5k_hw_4w_tx_ctl   tx_ctl;
      struct ath5k_hw_tx_status   tx_stat;
} __packed __aligned(4);
```

struct ath5k_hw_all_rx_desc - Common hardware RX descriptor {

```
      struct ath5k_hw_rx_ctl          rx_ctl;
      struct ath5k_hw_rx_status   rx_stat;
} __packed __aligned(4);
```

struct ath5k_desc - Atheros hardware DMA descriptor

```
struct ath5k_desc {
   u32   ds_link;
   u32   ds_data;

   union {
         struct ath5k_hw_5210_tx_desc  ds_tx5210;
         struct ath5k_hw_5212_tx_desc  ds_tx5212;
         struct ath5k_hw_all_rx_desc   ds_rx;
   } ud;
} __packed __aligned(4);
```

RX/TX Report Descriptor: contains information to report upper layer. They get filled by the hardware on each RX/TX attempt.

struct ath5k_rx_status : RX Status descriptor

```
struct ath5k_rx_status {
      u16   rs_datalen; // Data length
      u16   rs_tstamp; // Timestamp
      u8    rs_status; // Status code
```

```
      u8    rs_phyerr; // PHY error mask
      s8    rs_rssi; // RSSI in 0.5dbm units
      u8    rs_keyix; // Index to the key used for decrypting
      u8    rs_rate; // Rate used to decode the frame
      u8    rs_antenna; // Antenna used to receive the frame
      u8    rs_more; // Indicates this is a frame fragment (Fast
frames)
};
```

struct ath5k_tx_status - TX Status descriptor. TX status descriptor gets filled by the hw on each transmission attempt.

```
struct ath5k_tx_status {
   u16   ts_seqnum; // Sequence number
   u16   ts_tstamp; // Timestamp
   u8    ts_status; // Status code
   u8    ts_final_idx; // Final transmission series index
   u8    ts_final_retry; // Final retry count
   s8    ts_rssi; // RSSI for received ACK
   u8    ts_shortretry; // Short retry count
   u8    ts_virtcol; // Virtual collision count
   u8    ts_antenna; //Antenna used
};
```

QUEUES

struct ath5k_txq - Transmit queue state. Struct ath5k_hw has 10 queues of type "struct ath5k_txq" which denotes the queue in hardware. One of these exists for each hardware transmit queue. It has fields such as queue number, number of queued buffers, max allowed number of queued buffers, etc. Each of the 10 such queues in struct ath5k_hw are initialized in ath5k_init() function in base.c.

This structure pointer is also passed to the function ath5k_tx_queue() in base.c from ath5k_tx() in mac80211-ops.c. Using this structure, it is checked in ath5k_tx_queue() function if the queue is already at its maximum size or not …

```
struct ath5k_txq {
   unsigned int       qnum; // Hardware q number
   u32          *link; // Link ptr in last TX desc
   struct list_head   q; // Transmit queue (&struct list_head)
   spinlock_t         lock; // Lock on q and link
   bool          setup; // Is the queue configured
   int           txq_len; // Number of queued buffers
   int           txq_max; // Max allowed num of queued buffers
   bool          txq_poll_mark; // Used to check if queue got
stuck
   unsigned int       txq_stuck; // Queue stuck counter
};
```

`struct ath5k_txq_info` - A data struct to hold TX queue's parameters like queue type(enum ath5k_tx_queue), subtype(enum ath5k_tx_queue_subtype), cwmin, cwmax, aifs, transmission queue flags, constant bit rate period, and waiting time of the queue after ready is enabled. This structure is populated in the function ath5k_txq_setup which is called from the init function of the driver module - ath5k_init. Thus when the driver module is inserted, the queue parameters are initialized.

```
struct ath5k_txq_info {
   enum ath5k_tx_queue tqi_type; // One of enum ath5k_tx_queue
   enum ath5k_tx_queue_subtype tqi_subtype; // One of enum
ath5k_tx_queue_subtype
   u16   tqi_flags; // TX queue flags (see above)
   u8    tqi_aifs; // Arbitrated Inter-frame Space
   u16   tqi_cw_min; // Minimum Contention Window
   u16   tqi_cw_max; // Maximum Contention Window
   u32   tqi_cbr_period; // Constant bit rate period
   u32   tqi_cbr_overflow_limit;
   u32   tqi_burst_time;
   u32   tqi_ready_time; // Time queue waits after an event when
RDYTIME is enabled
};
```

BUFFER: stores frame data and all the information of a frame data.

`ath5k_buf : TX and RX buffer. It represents a single queued frame(buffer).It consists of pointers to sk_buff and ath5k_desc.`

```
struct ath5k_buf {
      struct list_head list;
      struct ath5k_desc     *desc;     /* virtual addr of desc */
      dma_addr_t       daddr;     /* physical addr of desc */
      struct sk_buff        *skb; /* skbuff for buf */
      dma_addr_t       skbaddr;/* physical addr of skb data */
};
```

INTERRUPT VALUE

These Interrupt Value is triggered by function `ath5k_hw_set_imr()` (in dma.c). The wireless card reads “chipset-dependent interrupt mask flags” and writes them to the interrupt mask register (IMR). This IMR then is read by function `ath5k_hw_get_isr()`.

RX Interrupt value

They are located in file ath5k.h. Some of them:

- AR5K_INT_RXOK: Frame successfully received
- AR5K_INT_RXERR: Frame reception failed
- AR5K_INT_RXEOL: Reached "End Of List", means we need more RX descriptors

- AR5K_INT_RXORN: Indicates we got RX FIFO overrun. Note that Rx overrun is not always fatal, on some chips we can continue operation without resetting the card, that's why %AR5K_INT_FATAL is not common for all chips.

TX Interrupt value

- AR5K_INT_TXOK: Frame transmission success.
- AR5K_INT_TXDESC: Request TX descriptor/Read TX status descriptor.
- AR5K_INT_TXERR: Frame transmission failure.
- AR5K_INT_TXEOL: Received End Of List for VEOL (Virtual End Of List). The Queue Control Unit (QCU) signals an EOL interrupt only if a descriptor's LinkPtr is NULL.

3. Brief overview of A Linux Wireless Driver:

To understand the operation of ath5k driver, we need to know the basic operation of a Linux wireless driver. In general, wireless drivers follow a typical route of processing:

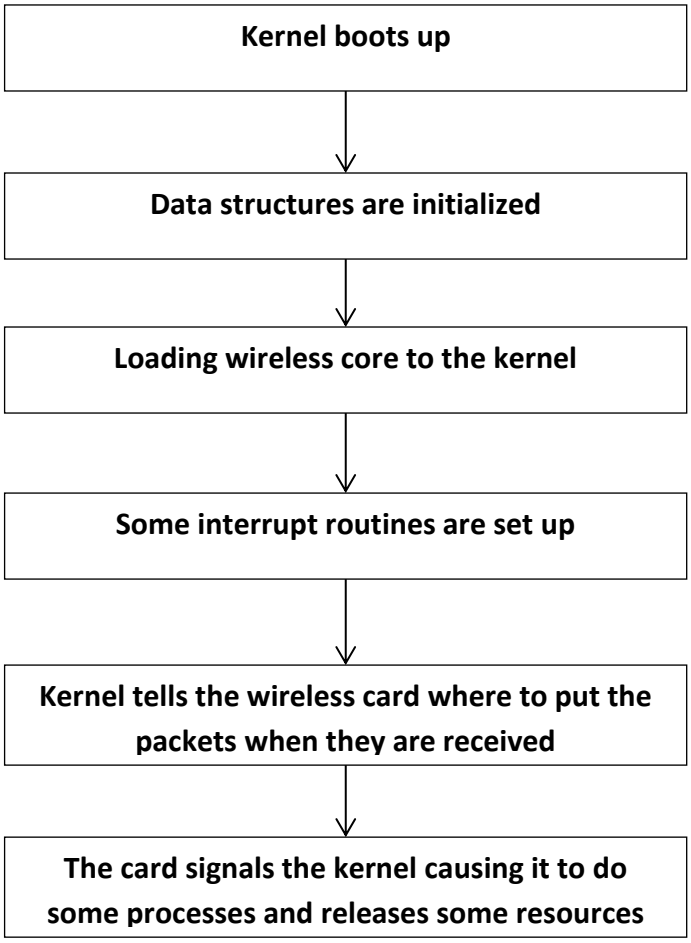


Figure 6: Basic operation of a Linux wireless driver

When the kernel boots up, it discovers the network card and sets up its own data structures such as hardware descriptors, queues etc. There are multiple queues for receiving and sending data while the kernel tries to process receiving and sending packets. All queues are independent so it is safe for a process to work with a queue backlog. In default, interrupts (happening when a wireless card sends/receives frames) are disabled when a core works with its queue backlog to prevent wasteful context switching. The wireless card also writes packets directly to memory (through DMA engine – see in Appendix B), there are no need of resources that are used to move the packets from one place to another.

All the processes above need an exact coordination between the kernel and the WLAN card. How to get this exact coordination since

every piece of hardware is different? The kernel creates a standard interface between itself and the underlying hardware through a “driver”. Driver contains wireless cores. Each core has a stable functionality. All wireless cores have to be loaded to the kernel by loading module functions. These module functions in the driver provide the abstract kernel functions by implementing them for the actual hardware. The driver uses whatever tricks available to fulfill a request. Network drivers typically give the hardware a region of memory, which describes separate memory locations it can write packets to, and carefully allocates/re-allocates resources as needed. These are typical operation of a wireless driver. Now I am going to explore ath5k source code.

4. Reception, Transmission and Configuration Path of ath5k

a. Initialization

Initialization functions have responsibility to allocate the resource such as hardware descriptors (TX and RX descriptors) and tasklets etc. The driver needs them to perform operation. Initialization Functions (e.g. ath5k_init(), ath5k_reset()) are called at the boot time:

- **`ath5k_init()`** in base.c calls some functions as:
- Allocates tx and rx descriptors using **`ath5k_desc_alloc()`** in base.c and the associated buffers.

```
/*
 * Allocate tx+rx descriptors and populate the lists.
 */
ret = ath5k_desc_alloc(ah);
if (ret) {
      ATH5K_ERR(ah, "can't allocate descriptors\n");
      goto err;
}
```

- Initializes tx queues (queues that use for transmission) by calling: **ath5k_txq_setup.**
- Initializes rx/tx etc **tasklets**.

```
tasklet_init(&ah->rxtq, ath5k_tasklet_rx, (unsigned long)ah);
tasklet_init(&ah->txtq, ath5k_tasklet_tx, (unsigned long)ah);
tasklet_init(&ah->beacontq, ath5k_tasklet_beacon, (unsigned
long)ah);
tasklet_init(&ah->ani_tasklet, ath5k_tasklet_ani, (unsigned
long)ah);
```

What is “tasklet”?

“Tasklets resemble kernel timers in some ways. They are always run at interrupt time, they always run on the same CPU that schedules them,

and they receive an unsigned long argument. Unlike kernel timers, however, you can't ask to execute the function at a specific time. By scheduling a tasklet, you simply ask for it to be executed at a later time chosen by the kernel. This behavior is especially useful with interrupt handlers, where the hardware interrupt must be managed as quickly as possible, but most of the data management can be safely delayed to a later time. Actually, a tasklet, just like a kernel timer, is executed (in atomic mode) in the context of a "soft interrupt," a kernel mechanism that executes asynchronous tasks with hardware interrupts enabled". [8]

- **`ath5k_reset()`** in base.c calls some functions as:

```
* ret = ath5k_hw_reset(ah, ah->opmode, ah->curchan, fast,
skip_pcu);
```

- **`ath5k_hw_reset`** has the responsibility to:
  * Sets the channel
  * Initializes QCUs/DCUs/PCU
  * Initializes DMA engine

```
/*
 * Configure QCUs/DCUs
 */
ret = ath5k_hw_init_queues(ah);
if (ret)
     return ret;
```

```
/*
 * Initialize DMA/Interrupts
 */
ath5k_hw_dma_init(ah);
```

```
/*
 * Initialize PCU
 */
ath5k_hw_pcu_init(ah, op_mode);
```

```
/*
 * Initialize PHY
 */
ret = ath5k_hw_phy_init(ah, channel, mode, false);
if (ret) {
     ATH5K_ERR(ah,
          "failed to initialize PHY (%i) !\n", ret);
     return ret;
}
```

- **`ath5k_reset`** also calls functions to start the PCU engine (Protocol Control Unit) to enable packet reception and processing by calling: `ath5k_rx_start(ah)`.

b. Reception Path

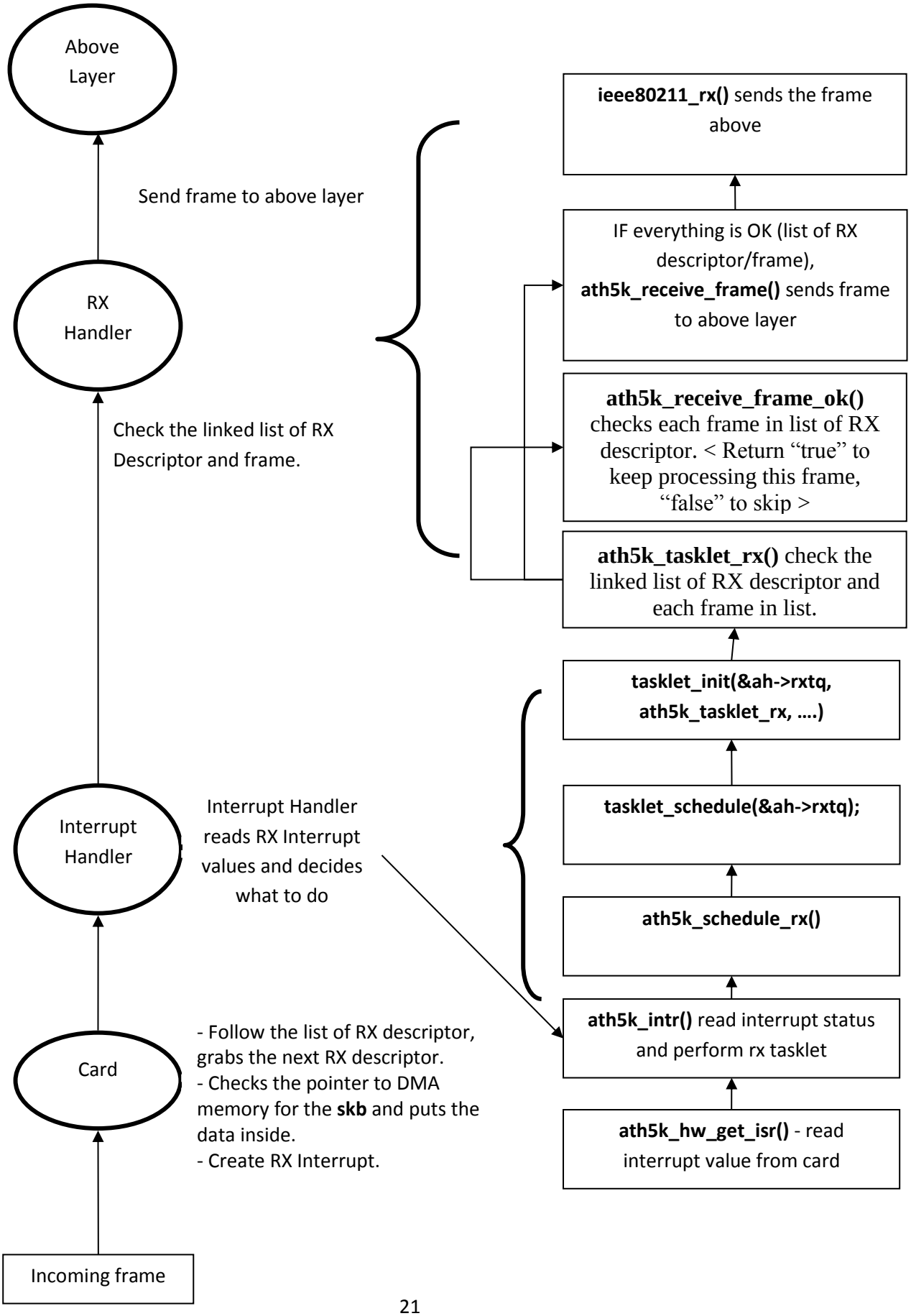

Above Layer
Send frame to above layer
RX Handler
Check the linked list of RX Descriptor and frame.
Interrupt Handler
Interrupt Handler reads RX Interrupt values and decides what to do
Card
- Follow the list of RX descriptor, grabs the next RX descriptor.
- Checks the pointer to DMA memory for the skb and puts the data inside.
- Create RX Interrupt.
Incoming frame
ieee80211_rx() sends the frame above
IF everything is OK (list of RX descriptor/frame), ath5k_receive_frame() sends frame to above layer
ath5k_receive_frame_ok() checks each frame in list of RX descriptor. < Return "true" to keep processing this frame, "false" to skip >
ath5k_tasklet_rx() check the linked list of RX descriptor and each frame in list.
tasklet_init(&ah->rxtq, ath5k_tasklet_rx, ….)
tasklet_schedule(&ah->rxtq);
ath5k_schedule_rx()
ath5k_intr() read interrupt status and perform rx tasklet
ath5k_hw_get_isr() - read interrupt value from card

Figure 7: Reception Path from ath5k driver to mac80211

- After driver calls initialization functions, it is now working properly. Process of receiving a frame is performed at run time. A frame is received by the wireless card (this includes normal frames, badly decoded frames etc.). The card follows the list of RX descriptors, grabs the next descriptor, checks the pointer to DMA memory (see how DMA engine works in Appendix) for the **`skb`** and puts the data inside. If this went fine, it issues an RXOK interrupt. If it ran out of descriptors it issues an RXEOL interrupt. If it ran out of buffer space - packet was larger than the memory allocated for it- it issues an RXORN interrupt indicating an RX overrun. If the packet was not received fine (with decoding errors etc.), it issues an RXERR interrupt. This process is shown in this part of the Figure 8.

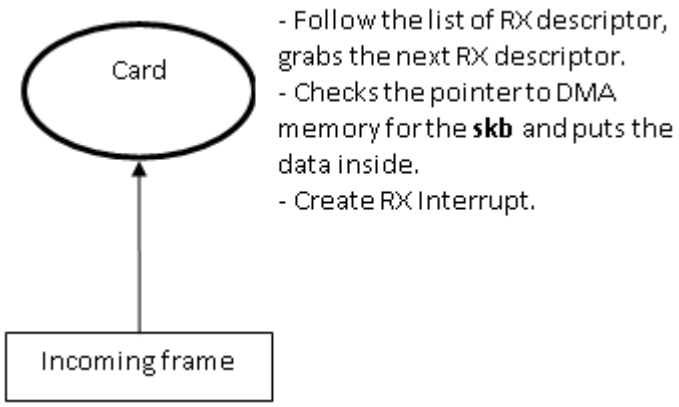


Figure 8: Frame comes to wireless card

- After the wireless card receives frame, function `ath5k_intr()` in file base.c is called.

Inside `ath5k_intr()`:

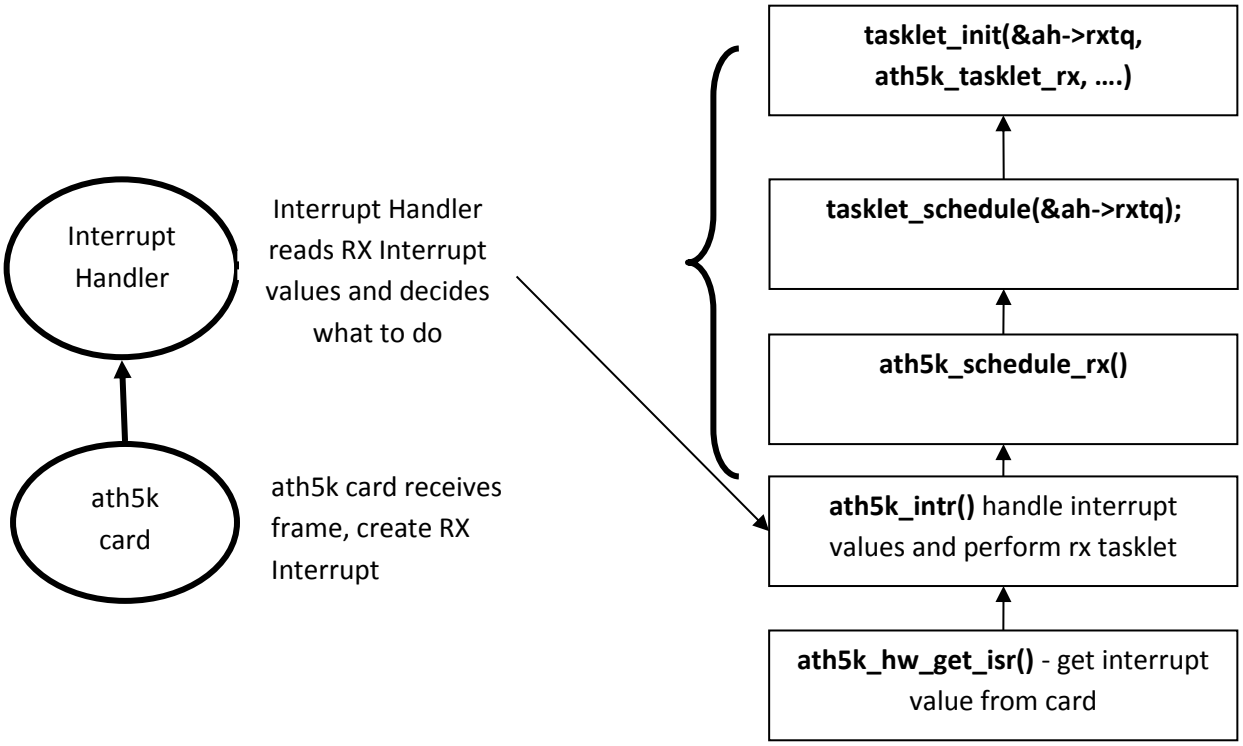


Figure 9: RX Interrupt Status is triggered, driver decides what to do

- Function `ath5k_intr()` in file base.c has the responsibility to handle the interrupt values (RXOK, RXERR, RXEOL) when they are triggered by the card.
- `ath5k_intr()` calls `ath5k_hw_get_isr()` to read the Interrupt value from Interrupt Register in the card.
- If RXEOL is caused, `ath5k_intr()` just counts the packet as a lost one.

```
if (status & AR5K_INT_RXEOL)
    ah->stats.rxeol_intr++;
```

- If RXERR/RXOK is caused, `ath5k_intr()` calls `ath5k_schedule_rx()` to schedule the rx tasklet.

```
/* RX -> Schedule rx tasklet */
if (status & (AR5K_INT_RXOK | AR5K_INT_RXERR))
    ath5k_schedule_rx(ah);
```

How `ath5k_schedule_rx()` work?

It performs RX Tasklet by calling ***`tasklet_schedule()`*** function.

```
static void ath5k_schedule_rx(struct ath5k_hw *ah)
{
```

```
        ah->rx_pending = true;
        tasklet_schedule(&ah->rxtq);
}
```

"With ***tasklet_schedule(&ah->rxtq)*** : schedule the tasklet **rxtq** (`struct tasklet_struct` ***rxtq;***) for execution. If a tasklet is scheduled again before it has a chance to run, it runs only once. However, if it is scheduled while it runs, it runs again after it completes; this ensures that events occurring while other events are being processed receive due attention. This behavior also allows a tasklet to reschedule itself". [8]

- Next, `ath5k_tasklet_rx()` (in file *base.c*) is called through a complicated functions calling.

Inside `ath5k_tasklet_rx()`

- `tasklet_init(&ah->`***rxtq, ath5k_tasklet_rx,*** `(unsigned long)ah);`

Function `ath5k_tasklet_rx()` has been initialized with the tasklet **`rxtq`** in the calling of `tasklet_init()` inside `ath5k_init()`. Every time, **`rxtq`** is scheduled in `tasklet_schedule(&ah->`***rxtq***`)`, `ath5k_tasklet_rx()` will be called.

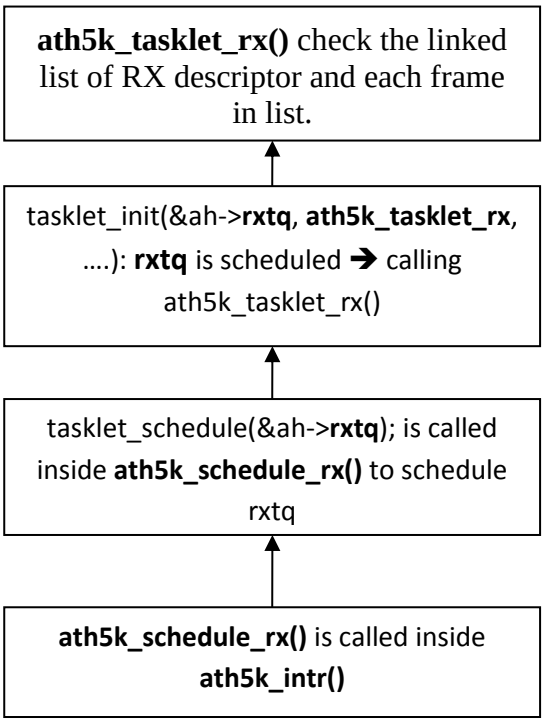


Figure 10: Tasklet RX operation

- `ath5k_tasklet_rx()` crawls the linked list of rx descriptors (**`struct ath5k_desc *desc`**). It checks the list of RX descriptors. In case a list is self-linked descriptor, it is skipped.

```
do {
            bf = list_first_entry(&ah->rxbuf, struct ath5k_buf,
list);
            BUG_ON(bf->skb == NULL);
            skb = bf->skb;
            ds = bf->desc;

            /* bail if HW is still using self-linked descriptor */
            if (ath5k_hw_get_rxdp(ah) == bf->daddr)
                  break;

            ret = ah->ah_proc_rx_desc(ah, ds, &rs);
            if (unlikely(ret == -EINPROGRESS))
                  break;
            else if (unlikely(ret)) {
                  ATH5K_ERR(ah, "error in processing rx
descriptor\n");
                  ah->stats.rxerr_proc++;
                  break;
            }
```

- If there is nothing wrong with the list of rx descriptors, it moves to check if this is a frame we want to receive or not by calling `ath5k_receive_frame_ok()`.

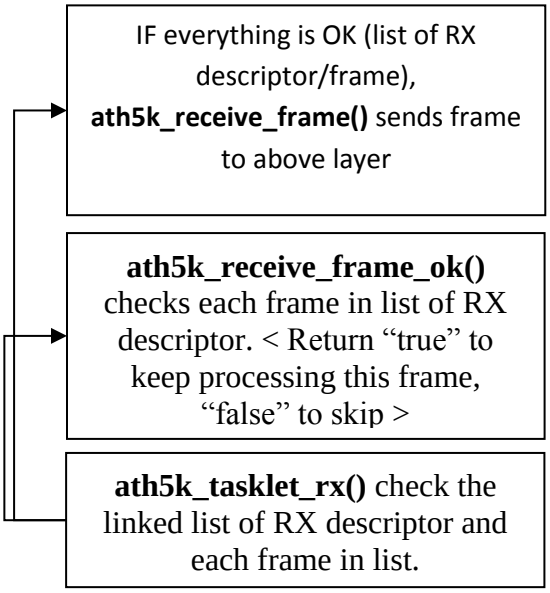


Figure 11: Check and receive frame

What do `ath5k_receive_frame_ok()` do?

- It checks the descriptor for flags that indicate an RX error; it can be an encryption/decryption error, a PHY decoding error etc.
- Update error counters.
- Return true or false. True means “this frame is OK, we are going to proceed it” and false means “Skip this frame”.

```
static bool ath5k_receive_frame_ok(struct ath5k_hw *ah, struct
ath5k_rx_status *rs)
{
      ah->stats.rx_all_count++;
      ah->stats.rx_bytes_count += rs->rs_datalen;
// Check the descriptor for flags that indicate an rx error, it
can be an encryption/decryption error, a PHY decoding error etc
and update error counters
      if (unlikely(rs->rs_status)) {
            if (rs->rs_status & AR5K_RXERR_CRC)
                  ah->stats.rxerr_crc++;
            if (rs->rs_status & AR5K_RXERR_FIFO)
                  ah->stats.rxerr_fifo++;
            if (rs->rs_status & AR5K_RXERR_PHY) {
                  ah->stats.rxerr_phy++;
                  if (rs->rs_phyerr > 0 && rs->rs_phyerr < 32)
                        ah->stats.rxerr_phy_code[rs->rs_phyerr]++;
                  return false;
            }
            if (rs->rs_status & AR5K_RXERR_DECRYPT) {
                  /*
                   * Decrypt error.  If the error occurred
                   * because there was no hardware key, then
                   * let the frame through so the upper layers
                   * can process it.  This is necessary for 5210
                   * parts which have no way to setup a ``clear''
                   * key cache entry.
                   *
                   * XXX do key cache faulting
                   */
                  ah->stats.rxerr_decrypt++;
                  if (rs->rs_keyix == AR5K_RXKEYIX_INVALID &&
                      !(rs->rs_status & AR5K_RXERR_CRC))
                        return true;
            }
            if (rs->rs_status & AR5K_RXERR_MIC) {
                  ah->stats.rxerr_mic++;
                  return true;
            }

            /* reject any frames with non-crypto errors */
```

```
            if (rs->rs_status & ~(AR5K_RXERR_DECRYPT))
                  return false;
      }

      if (unlikely(rs->rs_more)) {
            ah->stats.rxerr_jumbo++;
            return false;
      }
      return true;
}
```

Get back in `ath5k_tasklet_rx()`

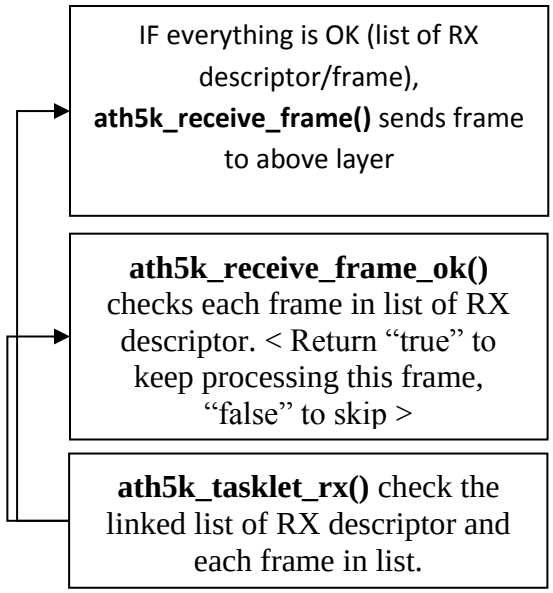


Figure 12: Check and receive frame

```
if (ath5k_receive_frame_ok(ah, &rs)) {
                  next_skb = ath5k_rx_skb_alloc(ah,
&next_skb_addr);

                  /*
                   * If we can't replace bf->skb with a new skb
under
                   * memory pressure, just skip this packet
                   */
                  if (!next_skb)
                        goto next;

                  dma_unmap_single(ah->dev, bf->skbaddr,
                              common->rx_bufsize,
                              DMA_FROM_DEVICE);

                  skb_put(skb, rs.rs_datalen);
```

```
            ath5k_receive_frame(ah, skb, &rs);

            bf->skb = next_skb;
            bf->skbaddr = next_skb_addr;
        }
```

- If the frame is damaged and driver doesn't want to send it to above layers, it will be skipped.
- If the frame is OK, driver will allocate a new buffer (for next incoming frame). It updates the RX descriptor to point there (driver doesn't reinitialize the descriptor list, it just update the pointers so they point to fresh memory and clean them up) by calling

```
next_skb = ath5k_rx_skb_alloc(ah, &next_skb_addr);
```

- There is a calling of

```
dma_unmap_single(ah->dev, bf->skbaddr,
                                common->rx_bufsize,
                                DMA_FROM_DEVICE);
```

So the memory that includes the packet is not owned by the device anymore.

- The calling of `skb_put(skb, rs.rs_datalen)` is to extend the buffer length to the received length.
- From the hardware, a frame is passed from ath5k driver to mac80211 by

```
ath5k_receive_frame(ah, skb, &rs);
```

Inside ath5k_receive_frame()

- Remove padding from the skb.

```
ath5k_remove_padding(skb);
```

- Create a ieee80211_rx_status

```
struct ieee80211_rx_status *rxs;
```

and copy info from the RX status descriptor to the `ieee80211_rx_status`. By this way the RX status will be understood by the protocol stack above.

```
rxs = IEEE80211_SKB_RXCB(skb);

rxs->flag = 0;
if (unlikely(rs->rs_status & AR5K_RXERR_MIC))
```

```
            rxs->flag |= RX_FLAG_MMIC_ERROR;

      rxs->mactime = ath5k_extend_tsf(ah, rs->rs_tstamp);
      rxs->flag |= RX_FLAG_MACTIME_MPDU;

      rxs->freq = ah->curchan->center_freq;
      rxs->band = ah->curchan->band;

      rxs->signal = ah->ah_noise_floor + rs->rs_rssi;

      rxs->antenna = rs->rs_antenna;

      if (rs->rs_antenna > 0 && rs->rs_antenna < 5)
            ah->stats.antenna_rx[rs->rs_antenna]++;
      else
            ah->stats.antenna_rx[0]++; /* invalid */

      rxs->rate_idx = ath5k_hw_to_driver_rix(ah, rs->rs_rate);
      rxs->flag |= ath5k_rx_decrypted(ah, skb, rs);

      if (rxs->rate_idx >= 0 && rs->rs_rate ==
          ah->sbands[ah->curchan->band].bitrates[rxs-
>rate_idx].hw_value_short)
            rxs->flag |= RX_FLAG_SHORTPRE;

      trace_ath5k_rx(ah, skb);

      ath5k_update_beacon_rssi(ah, skb, rs->rs_rssi);
```

- Call ieee80211_rx to send the packet above.

```
      ieee80211_rx(ah->hw, skb);
```

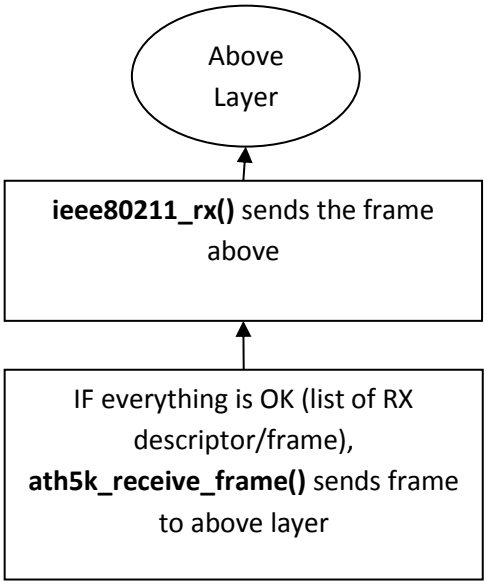


Figure 13: Above layer receives frame

- The calling of `ieee80211_rx()` to hand received frames to mac80211. The receive buffer in **skb** must start with an IEEE 802.11 header. This is why driver has to copy info from the RX status descriptor to the `ieee80211_rx_status`

Get back inside ath5k_tasklet_rx()

- Some jobs have to be done for receiving next incoming frames. `ath5k_tasklet_rx()` updates the pointer of the **`skb`** to the fresh memory we allocated before, moves on to the next packet while calling `ath5k_rxbuf_setup()` to setup a new descriptor in the place of the old one.
- We are done when the whole list gets crawled.

```
next:
            list_move_tail(&bf->list, &ah->rxbuf);
      } while (ath5k_rxbuf_setup(ah, bf) == 0);
unlock:
      spin_unlock(&ah->rxbuflock);
      ah->rx_pending = false;
      ath5k_set_current_imask(ah);
```

c. Transmission

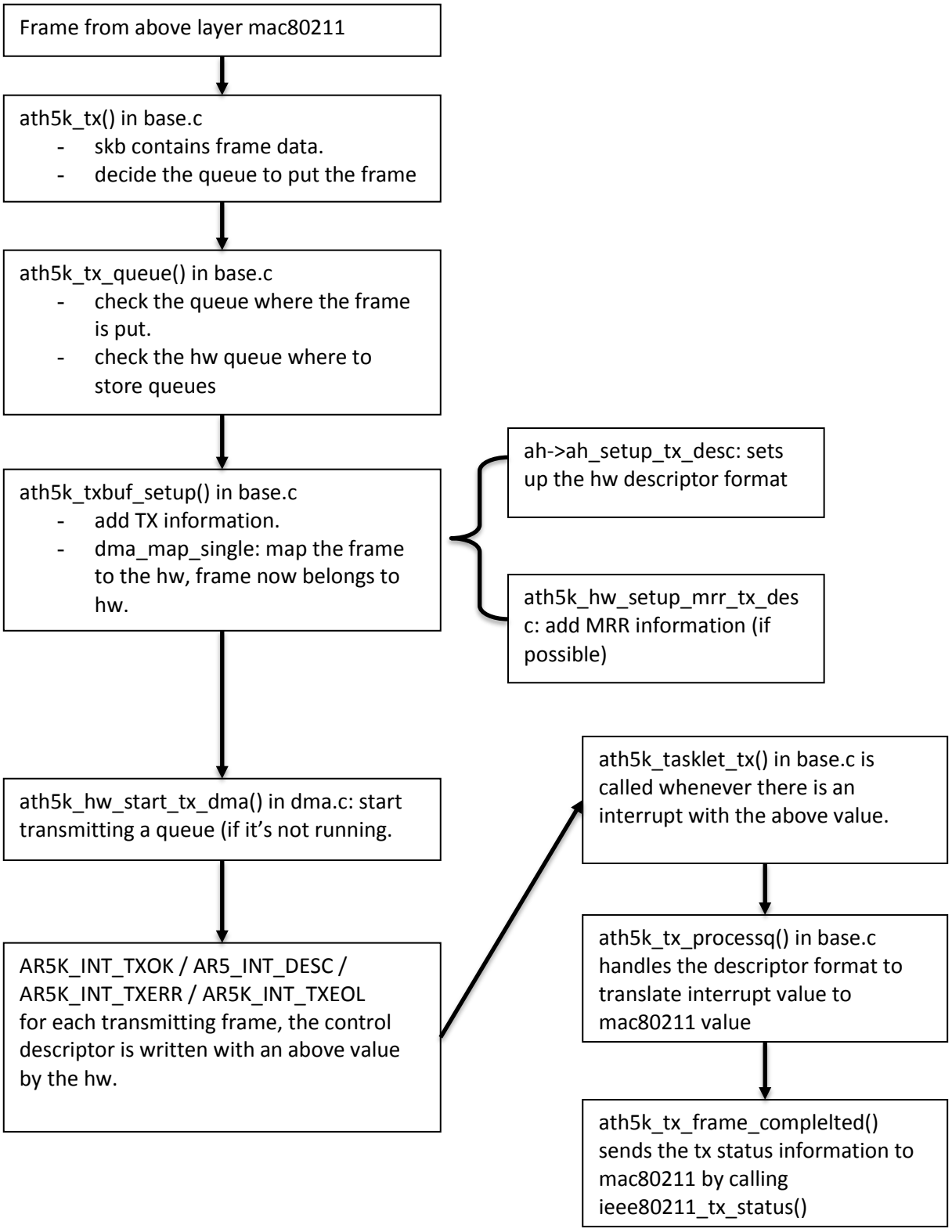

Frame from above layer mac80211
ath5k_tx() in base.c
- skb contains frame data.
- decide the queue to put the frame
ath5k_tx_queue() in base.c
- check the queue where the frame is put.
- check the hw queue where to store queues
ath5k_txbuf_setup() in base.c
- add TX information.
- dma_map_single: map the frame to the hw, frame now belongs to hw.
ah->ah_setup_tx_desc: sets up the hw descriptor format
ath5k_hw_setup_mrr_tx_desc: add MRR information (if possible)
ath5k_hw_start_tx_dma() in dma.c: start transmitting a queue (if it's not running.
AR5K_INT_TXOK / AR5_INT_DESC / AR5K_INT_TXERR / AR5K_INT_TXEOL
for each transmitting frame, the control descriptor is written with an above value by the hw.
ath5k_tasklet_tx() in base.c is called whenever there is an interrupt with the above value.
ath5k_tx_processq() in base.c handles the descriptor format to translate interrupt value to mac80211 value
ath5k_tx_frame_complelted() sends the tx status information to mac80211 by calling ieee80211_tx_status()

Figure 14: Transmission Path from mac80211 to ath5k driver

i. Initialize the buffer

- When a frame is sent from mac80211 to driver, `ath5k_tx()` in file mac80211-ops.c is called. `ath5k_tx()` calls function `skb_get_queue_mapping()` to find an appropriate queue for the frame which is stored in **`struct sk_buff *skb`**.

```
u16 qnum = skb_get_queue_mapping(skb);
```

- In here, ***skb*** contains the frame data. ath5k_tx() decides on which queue should it put the frame. After finding a queue where the frame can be stored, `ath5k_tx_queue()` is called.

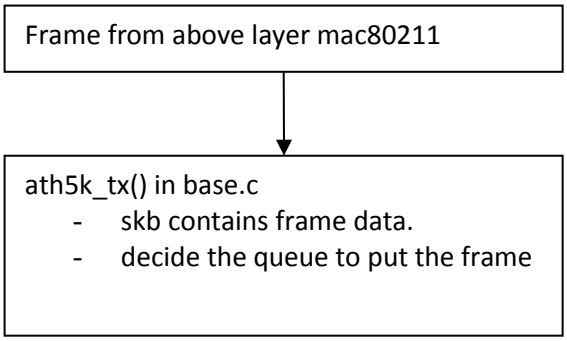


Figure 15: Above layer transmits frame to driver

```
static void ath5k_tx(struct ieee80211_hw *hw, struct sk_buff
*skb)
{
      struct ath5k_hw *ah = hw->priv;//priv: pointer to private
area that was allocated for driver use along with this structure
      u16 qnum = skb_get_queue_mapping(skb);//return skb->queue_mapping  -
Finds the appropriate hw queue for that skb

      if (WARN_ON(qnum >= ah-
>ah_capabilities.cap_queues.q_tx_num)) {
            dev_kfree_skb_any(skb);
            return;
      }

      ath5k_tx_queue(hw, skb, &ah->txqs[qnum]);//This function is call to receives
the skb, which is going to be transmitted, from mac80211
}
```

ii. Assign frame to a proper queue

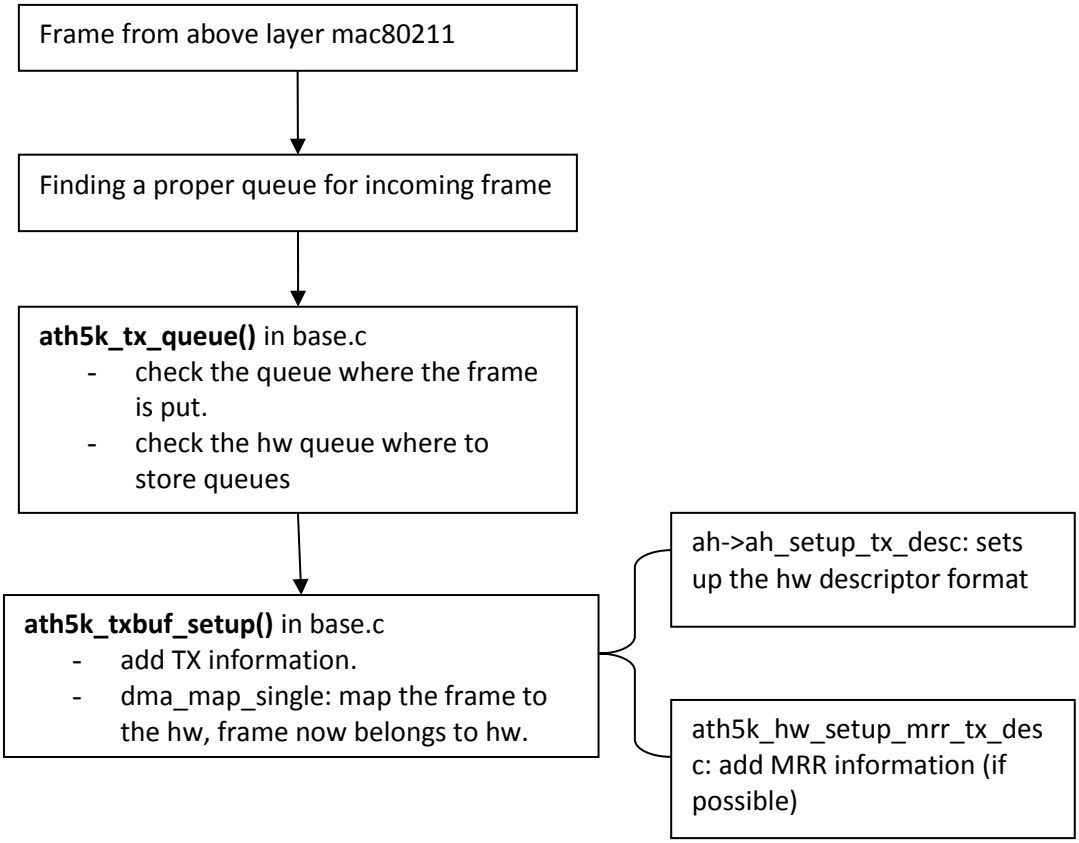


Figure 16: Assign frame to a queue, prepare information for transmission

- After the process of finding a hardware queue, the driver now receives the packet from mac80211 (by the calling of `ath5k_tx_queue()` in file base.c). This is how `ath5k_tx_queue()` works.

- It fixes the frame to be 4byte aligned by adding some padding (because hw expects it this way) by calling:

```
   /*
    * The hardware expects the header padded to 4 byte
boundaries.
    * If this is not the case, we add the padding after
the header.
    */
   padsize = ath5k_add_padding(skb);
```

- It checks the frame. If there is something wrong with the padding. Drop this frame.

```
if (padsize < 0) {
          ATH5K_ERR(ah, "tx hdrlen not %%4: not enough headroom
to pad");//Check if adding 4bytes padding is failed -> drop
frame
          goto drop_packet;
     }
```

- It also check queue where the driver intends to copy the frame and the tx buffer. If the specified queue is full it tells mac80211 to stop sending more packets on that queue. If we are out of tx buffers it tells mac80211 to stop sending frames on all queues and drops the frame by the calling of `ieee80211_stop_queue()`.

```
/*Check the specified hardware queue. If it is already at
its maximum size (50 packets), it will call
ieee80211_stop_queue() to notify mac80211 to stop its queue
and not receiving packets anymore or also drop incoming
packet (with txq->txq_len is the number of queued buffer,
txq->txq_max is the maximum allowed number of queued
buffer, txq->qnum is the number of hardware queue)*/

     if (txq->txq_len >= txq->txq_max &&
         txq->qnum <= AR5K_TX_QUEUE_ID_DATA_MAX)
           ieee80211_stop_queue(hw, txq->qnum);
```

```
It also checks txbuf (transmit buffer - struct list_head
txbuf; ) list (or list of tx queue). If we are out of
txbuf, it will call ieee80211_stop_queues() or also drop
frame.

     spin_lock_irqsave(&ah->txbuflock, flags);
     if (list_empty(&ah->txbuf)) {
          ATH5K_ERR(ah, "no further txbuf available, dropping
packet\n");
          spin_unlock_irqrestore(&ah->txbuflock, flags);
          ieee80211_stop_queues(hw);
          goto drop_packet;
     }
     bf = list_first_entry(&ah->txbuf, struct ath5k_buf, list);
     list_del(&bf->list);
     ah->txbuf_len--;
     if (list_empty(&ah->txbuf))
          ieee80211_stop_queues(hw);
     spin_unlock_irqrestore(&ah->txbuflock, flags);
```

- If everything is OK, the frame is copied to a **`struct ath5k_buf *bf`**; which is a single queued frame and then `ath5k_txbuf_setup()` is called.

```
      bf->skb = skb; //Copy a skb to a struct ath5k_buf *buf
which is a single queued frame.
With good packets, ath5k_txbuf_setup() is called to prepare the
requirements for transmission.
      if (ath5k_txbuf_setup(ah, bf, txq, padsize)) {
            bf->skb = NULL;
            spin_lock_irqsave(&ah->txbuflock, flags);
            list_add_tail(&bf->list, &ah->txbuf);
            ah->txbuf_len++;
            spin_unlock_irqrestore(&ah->txbuflock, flags);
            goto drop_packet;
      }
      return;

drop_packet:
      dev_kfree_skb_any(skb);
}
```

iii. Set transmission flags

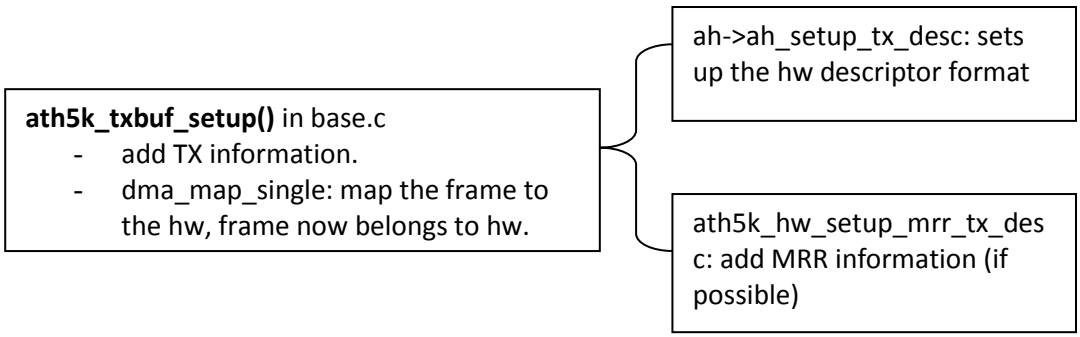


Figure 17: Set wireless card's transmission flags

- `ath5k_txbuf_setup()` is called inside `ath5k_tx_queue()`.
- `ath5k_txbuf_setup()` has responsibility to prepares the descriptor information, telling the hw how it should send the frame, on which rate etc.

```
rate = ieee80211_get_tx_rate(ah->hw, info);
      if (!rate) {
            ret = -EINVAL;
            goto err_unmap;
```

```
     }

     if (info->flags & IEEE80211_TX_CTL_NO_ACK)
          flags |= AR5K_TXDESC_NOACK;

     rc_flags = info->control.rates[0].flags;
     hw_rate = (rc_flags & IEEE80211_TX_RC_USE_SHORT_PREAMBLE) ?
          rate->hw_value_short : rate->hw_value;

     pktlen = skb->len;

     /* FIXME: If we are in g mode and rate is a CCK rate
      * subtract ah->ah_txpower.txp_cck_ofdm_pwr_delta
      * from tx power (value is in dB units already) */
     if (info->control.hw_key) {
          keyidx = info->control.hw_key->hw_key_idx;
          pktlen += info->control.hw_key->icv_len;
     }
     if (rc_flags & IEEE80211_TX_RC_USE_RTS_CTS) {
          flags |= AR5K_TXDESC_RTSENA;
          cts_rate = ieee80211_get_rts_cts_rate(ah->hw, info)-
>hw_value;
          duration = le16_to_cpu(ieee80211_rts_duration(ah->hw,
               info->control.vif, pktlen, info));
     }
     if (rc_flags & IEEE80211_TX_RC_USE_CTS_PROTECT) {
          flags |= AR5K_TXDESC_CTSENA;
          cts_rate = ieee80211_get_rts_cts_rate(ah->hw, info)-
>hw_value;
          duration =
le16_to_cpu(ieee80211_ctstoself_duration(ah->hw,
               info->control.vif, pktlen, info));
     }
```

- It collects and sets information from mac80211 that the hardware requires such as:

```
/**
* enum ieee80211_rate_flags - rate flags
*
* Hardware/specification flags for rates. These are structured
* in a way that allows using the same bitrate structure for
* different bands/PHY modes.
*
* @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.
*/
```

- These informations are set in transmit flags of the hardware. They have the values depending on the type of the packet, physical layer parameters and control information. All these informations are then set to the hardware by calling `ah_setup_tx_desc()`. `dma_map_single()` is called for the buffer so that it's owned by our device.

```
bf->skbaddr = dma_map_single(ah->dev, skb->data, skb->len,
DMA_TO_DEVICE)
```

- If we are ok it calls `ah->ah_setup_tx_desc` (that's a function pointer (located in **desc.c**) that points to the chip revision specific function that handles the hw descriptor format. `ah->ah_setup_tx_desc` sets up the hw descriptor format. Hw expects a linked list of descriptors, each descriptor contains a pointer to the buffer and some fields with information on how to send it (these fields are mostly set in `ath5k_txbuf_setup`). Hw starts from the first descriptor (**txdp**) and follows the list when the queue is started.

```
     ret = ah->ah_setup_tx_desc(ah, ds, pktlen,
           ieee80211_get_hdrlen_from_skb(skb), padsize,
           get_hw_packet_type(skb),
           (ah->power_level * 2),
           hw_rate,
           info->control.rates[0].count, keyidx, ah->ah_tx_ant,
flags,
           cts_rate, duration);
     if (ret)
           goto err_unmap;
```

- If possible it also adds MRR information (multi rate retry)

```
     /* Set up MRR descriptor */
     if (ah->ah_capabilities.cap_has_mrr_support) {
           memset(mrr_rate, 0, sizeof(mrr_rate));
           memset(mrr_tries, 0, sizeof(mrr_tries));
           for (i = 0; i < 3; i++) {
```

```
                rate = ieee80211_get_alt_retry_rate(ah->hw,
info, i);
                if (!rate)
                     break;

                mrr_rate[i] = rate->hw_value;
                mrr_tries[i] = info->control.rates[i + 1].count;
           }
```

- There is also a small calling of `ath5k_hw_setup_mrr_tx_desc()` to prepare a Multi Rate Retry (MRR) descriptor which support for MRR algorithm.

```
//Adding MRR (Multi Rate Retry) information
           ath5k_hw_setup_mrr_tx_desc(ah, ds,
                mrr_rate[0], mrr_tries[0],
                mrr_rate[1], mrr_tries[1],
                mrr_rate[2], mrr_tries[2]);
     }

     ds->ds_link = 0;
     ds->ds_data = bf->skbaddr;

     spin_lock_bh(&txq->lock);
     list_add_tail(&bf->list, &txq->q);
     txq->txq_len++;
     if (txq->link == NULL) /* is this first packet? */
```

- Now that the hw descriptor struct is ready (with the right format and everything) we check out if it's the first descriptor on the list (so we call `ath5k_hw_set_txdp` to put it's address on a hw register so that hw knows where to start reading) or if it's yet another one so it only adds it to the list.

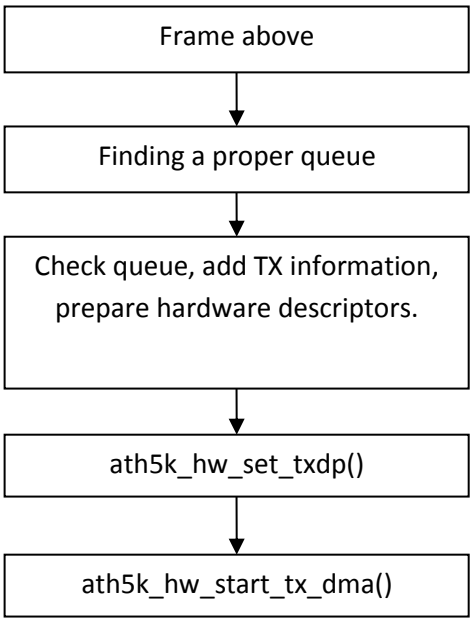


Figure 18: Driver transmits frame

```
            ath5k_hw_set_txdp(ah, txq->qnum, bf->daddr);
      else /* no, so only link it */
            *txq->link = bf->daddr;

      txq->link = &ds->ds_link;
      ath5k_hw_start_tx_dma()
```

```
      ath5k_hw_start_tx_dma(ah, txq->qnum);
      mmiowb();
      spin_unlock_bh(&txq->lock);

      return 0;
err_unmap:
      dma_unmap_single(ah->dev, bf->skbaddr, skb->len,
DMA_TO_DEVICE);
      return ret;
}
```

iv. Transfer the packet

- After setting all transmit flags, `ath5k_hw_start_tx_dma()`, which is located in dma.c, is called to transmit the packet using DMA engine. ***AR5K_REG_WRITE_Q*** is used to write a certain register in the hardware, so that the transmission on a certain queue starts.

```
int ath5k_hw_start_tx_dma(struct ath5k_hw *ah, unsigned int
queue)
```

```
{
      u32 tx_queue;

      AR5K_ASSERT_ENTRY(queue, ah-
>ah_capabilities.cap_queues.q_tx_num);

      /* Return if queue is declared inactive */
      if (ah->ah_txq[queue].tqi_type == AR5K_TX_QUEUE_INACTIVE)
            return -EINVAL;
      //Queue for 5210 cards
      if (ah->ah_version == AR5K_AR5210) {
            tx_queue = ath5k_hw_reg_read(ah, AR5K_CR);

            /*
             * Set the queue by type on 5210
             */
            switch (ah->ah_txq[queue].tqi_type) {
            case AR5K_TX_QUEUE_DATA:
                  tx_queue |= AR5K_CR_TXE0 & ~AR5K_CR_TXD0;
                  break;
            case AR5K_TX_QUEUE_BEACON:
                  tx_queue |= AR5K_CR_TXE1 & ~AR5K_CR_TXD1;
                  ath5k_hw_reg_write(ah, AR5K_BCR_TQ1V |
AR5K_BCR_BDMAE,
                              AR5K_BSR);
                  break;
            case AR5K_TX_QUEUE_CAB:
                  tx_queue |= AR5K_CR_TXE1 & ~AR5K_CR_TXD1;
                  ath5k_hw_reg_write(ah, AR5K_BCR_TQ1FV |
AR5K_BCR_TQ1V |
                        AR5K_BCR_BDMAE, AR5K_BSR);
                  break;
            default:
                  return -EINVAL;
            }
            /* Start queue */
            ath5k_hw_reg_write(ah, tx_queue, AR5K_CR);
            ath5k_hw_reg_read(ah, AR5K_CR);
      } else {
            /* Return if queue is disabled */
            if (AR5K_REG_READ_Q(ah, AR5K_QCU_TXD, queue))
                  return -EIO;

            /* Start queue */
            AR5K_REG_WRITE_Q(ah, AR5K_QCU_TXE, queue);
      }

      return 0;
}
```

v. Calling transmission interrupt to check the frame's status

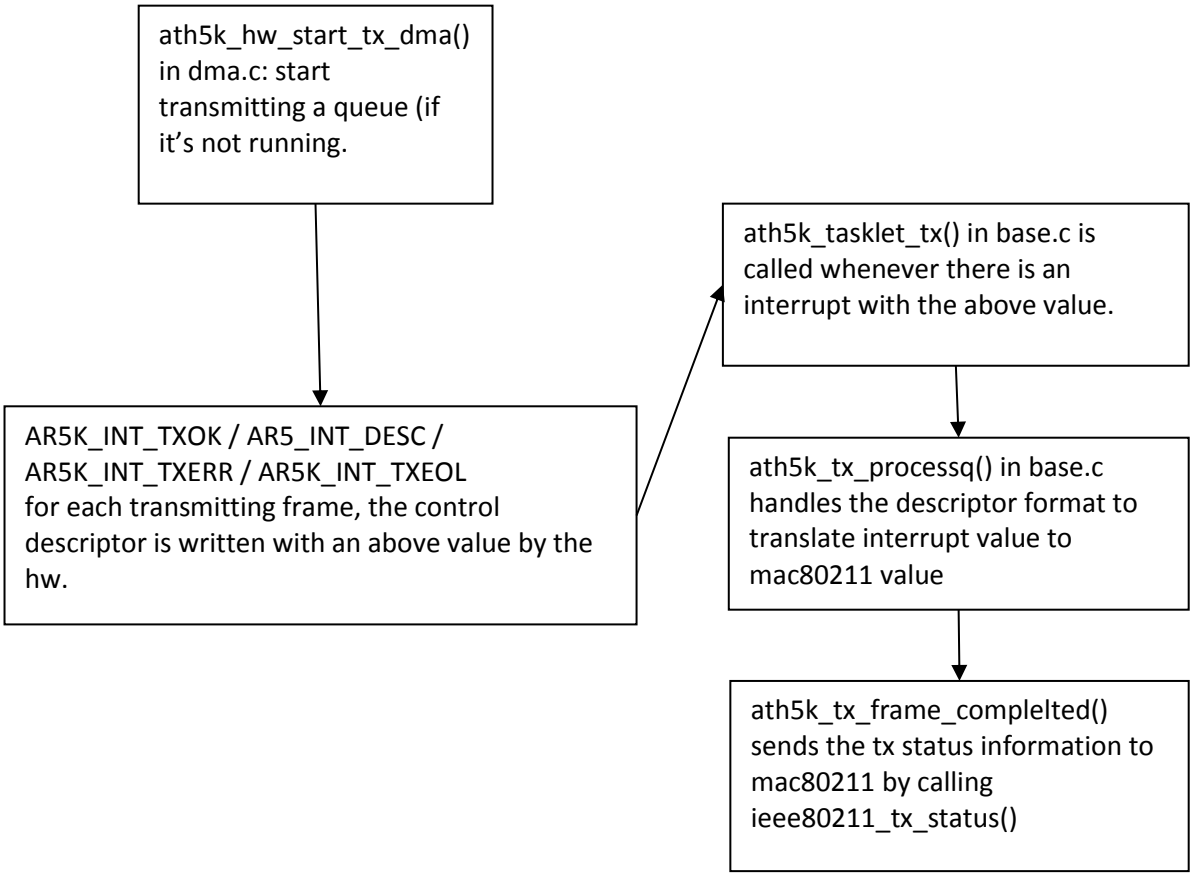


Figure 19: TX Interrupt Status is triggered, driver reports to above layer

Ath5k card crawls the linked list and tries to send the packet, for each packet it overwrites the descriptor (control descriptor) which the card set up previously in Initialization process with a new one containing status information (status descriptor).

If a frame was transmitted ok, the card sends the **AR5K_INT_TXOK** interrupt.

If we didn't enable the **AR5K_INT_TXOK** interrupt but instead asked the card to notify us when a specific frame is sent (e.g. the 4th frame on the list) using a field on the control descriptor, it sends the **AR5K_INT_TXDESC** interrupt.

If a frame was not transmitted due to an error we get the **AR5K_INT_TXERR** interrupt and if the card ran out of descriptors it sends the **AR5K_INT_TXEOL** interrupt.

On all the above cases we just want to process the status descriptors and notify mac80211 about the status of each transmitted frame, a tx tasklet to do it via `ath5k_schedule_tx()`. The calling of ath5k_schedule_tx() is the same with ath5k_shedule_rx() previously.

```
static void ath5k_tasklet_tx(unsigned long data)
{
      int i;
      struct ath5k_hw *ah = (void *)data;

      for (i = 0; i < AR5K_NUM_TX_QUEUES; i++)
            if (ah->txqs[i].setup && (ah->ah_txq_isr_txok_all &
BIT(i)))
                  ath5k_tx_processq(ah, &ah->txqs[i]);

      ah->tx_pending = false;
      ath5k_set_current_imask(ah);
}
```

`ath5k_tasklet_tx()` is called and for each queue that's enabled -and we got an interrupt and runs the `ath5k_tx_processq()`.

For each queue that has been sent, `ath5k_tx_processq()` is called. It calls for each descriptor on the queue `ah->ah_proc_tx_desc()` (a function pointer located in desc.c that points to the chip revision specific function that handles the descriptor format "translation") and then `ath5k_tx_frame_completed()`.

For each descriptor on the queue `ath5k_tx_frame_completed()` is being called - It sends the tx status information to mac80211 by calling **ieee80211_tx_status(ah->hw, skb);** (to update various counters and the rate control algorithm) and returns the buffer (after removing the padding that driver has added) back to mac80211 (so in case of error mac80211 can re-send the packet, or in case everything went ok it can free it).

4.4/ Configuration Path

ath5k is a mac80211-based driver. mac80211 [9] implements cfg80211 [10] callbacks for Soft-MAC devices, mac80211 then depends on cfg80211 for both registration to the networking subsystem and for configuration device. Configuration is handled by cfg80211 (through nl80211 [11]) and wireless extensions [12].

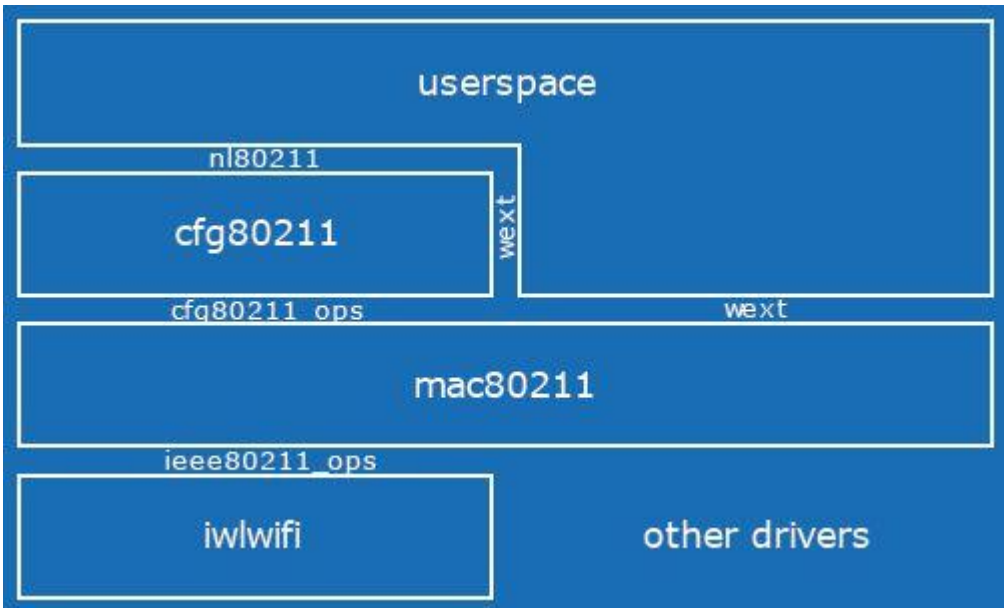


Figure 20: Architecture of mac80211 [13]

The different between Wireless Extension and cfg80211 is that: Wireless Extension uses **ioctl** system call [14] while cfg80211 replaces it to use "callbacks". What are callbacks? User space gives command to configure driver. Kernel analyzes command, takes command's values and passes values to mac80211 (through cfg80211). So, callbacks are data structures in cfg80211 and mac80211 that store user space's command. In above figure, callbacks are `cfg80211_ops` and `ieee80211_ops`.

Here are some mac80211 callbacks; they are implemented in `struct ieee80211_ops` in file \...\net\mac80211.h:

```
struct ieee80211_ops {
      void (*tx)(struct ieee80211_hw *hw, struct sk_buff *skb);
      void (*tx_frags)(struct ieee80211_hw *hw, struct
ieee80211_vif *vif,
                  struct ieee80211_sta *sta, struct sk_buff_head
*skbs);
      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);
#endif
      int (*add_interface)(struct ieee80211_hw *hw,
                     struct ieee80211_vif *vif);
```

```
……etc……
};
```

How is a driver configured by callbacks? For example: in below figure, user space gives command to change operation mode of ath5k card.

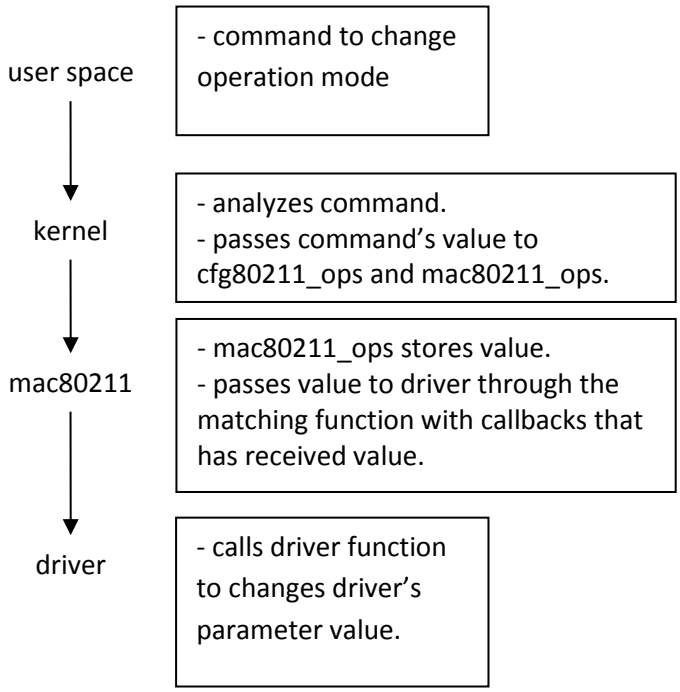


Figure 21: Configure driver parameter by callbacks

Each callback is pointed by a matching function which is implemented in driver. This function uses callbacks' values to configure wireless card.

But at this moment, ath5k source code has just been implemented `ieee80211_ops`, `cfg80211_ops` has not been implemented yet. It now uses both nl80211 ("iw" command) and wireless extension ("iwconfig" command) to pass value to `ieee80211_ops`; but Wireless Extension is completely hidden from driver. Although we cannot find any line of ath5k source code that uses IOCTL system call, we can use "iwconfig" command normally. In ath5k driver, some matching functions are implemented in file **mac80211-ops.c** and they are combined in `struct` **`ath5k_hw_ops`**:

```
const struct ieee80211_ops ath5k_hw_ops = {
      .tx              = ath5k_tx,
      .start                 = ath5k_start,
      .stop            = ath5k_stop,
      .add_interface         = ath5k_add_interface,
```

```
	/* .change_interface  = not implemented */
	.remove_interface	= ath5k_remove_interface,
	.config			= ath5k_config,
	.bss_info_changed	= ath5k_bss_info_changed,
	.prepare_multicast	= ath5k_prepare_multicast,
	.configure_filter	= ath5k_configure_filter,
	/* .set_tim		= not implemented */
	.set_key		= ath5k_set_key,
	/* .update_tkip_key	= not implemented */
	/* .hw_scan		= not implemented */
	.sw_scan_start		= ath5k_sw_scan_start,
	.sw_scan_complete	= ath5k_sw_scan_complete,
	.get_stats		= ath5k_get_stats,
	/* .get_tkip_seq= not implemented */
	/* .set_frag_threshold     = not implemented */
	/* .set_rts_threshold = not implemented */
	/* .sta_add		= not implemented */
	/* .sta_remove		= not implemented */
	/* .sta_notify		= not implemented */
	.conf_tx		= ath5k_conf_tx,
	.get_tsf		= ath5k_get_tsf,
	.set_tsf		= ath5k_set_tsf,
	.reset_tsf		= ath5k_reset_tsf,
	/* .tx_last_beacon    = not implemented */
	/* .ampdu_action= not needed */
	.get_survey		= ath5k_get_survey,
	.set_coverage_class	= ath5k_set_coverage_class,
	/* .rfkill_poll       = not implemented */
	/* .flush		= not implemented */
	/* .channel_switch    = not implemented */
	/* .napi_poll         = not implemented */
	.set_antenna		= ath5k_set_antenna,
	.get_antenna		= ath5k_get_antenna,
	.set_ringparam		= ath5k_set_ringparam,
	.get_ringparam		= ath5k_get_ringparam,
};
```

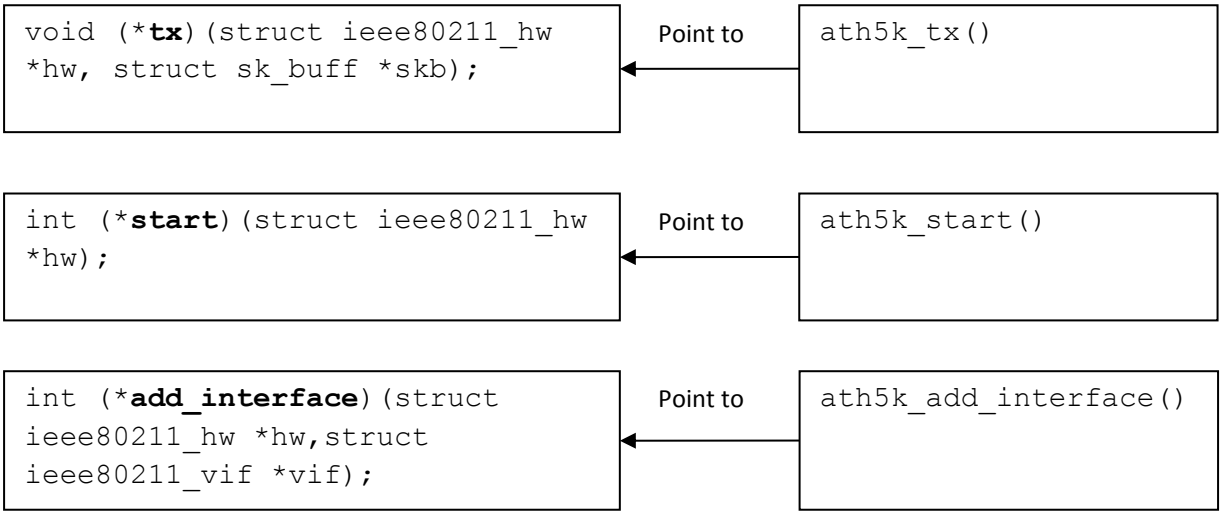


Figure 22: Relationship between driver functions and mac80211 callbacks

By this way, every time user space has commands to configure device's parameters. These commands will be analyzed by kernel; command's value is passed to mac80211 callbacks and then passed to driver's matching functions.

User space can configure card's parameter if and only if driver has implemented functions that match mac80211's callbacks. But not all callbacks of mac80211 are implemented in driver, so user space cannot change all the parameters that he wants. As we can see in the previous section code of **`struct ieee80211_ops ath5k_hw_ops`**, there are some functions that have been commented as "Not Implemented". It means that the driver not support to configure these parameters.

Each callback has its own calling path. In this section, I choose to analyze the most popular function that is "configuration working mode of ath5k wireless card".

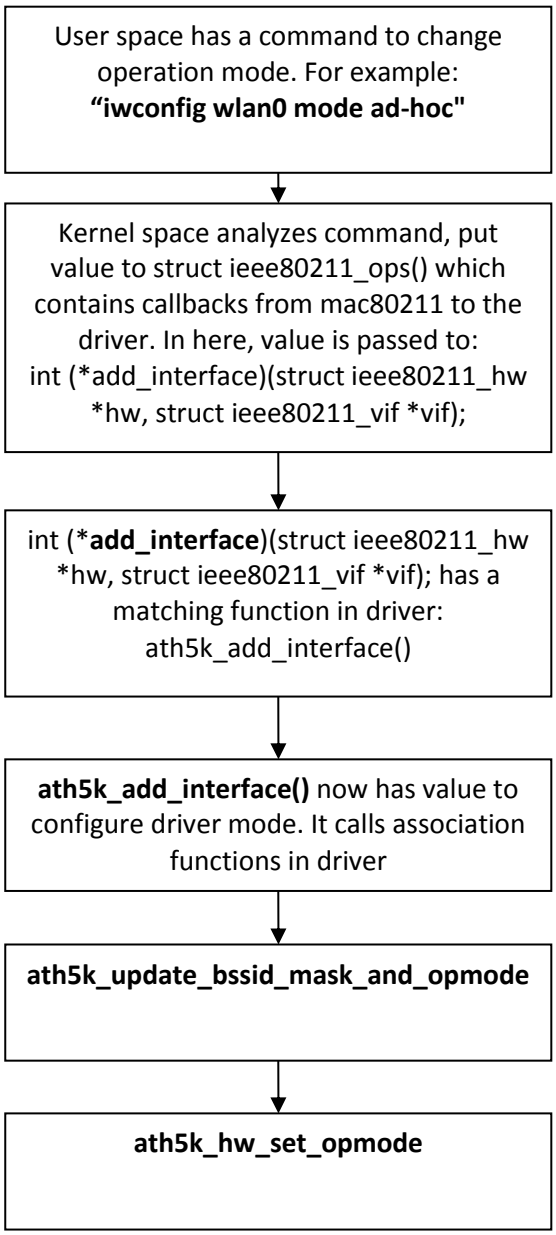


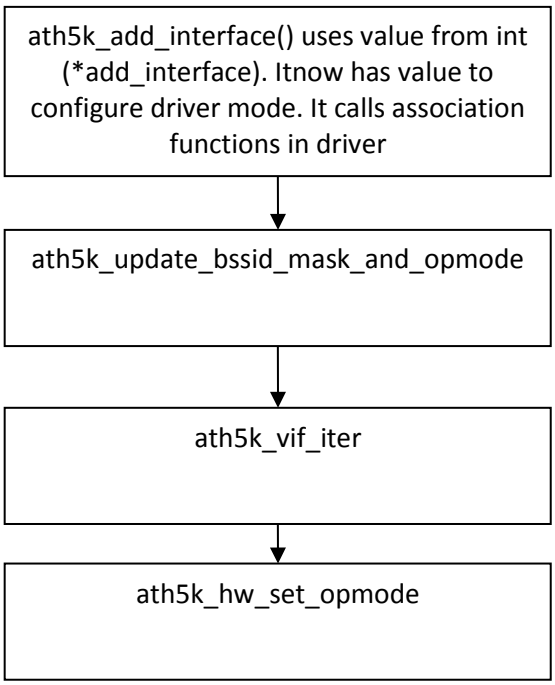
Figure 23: Configuration function calls of ath5k

Firstly, user space input command, for example: **“iwconfig wlan0 mode ad-hoc"** to change the mode of ath5k wireless card. A wireless network interface card always operates in one of the following operating modes: station infrastructure mode, access point infrastructure mode, monitor mode, ad-hoc (IBSS) mode, wireless distribution mode, mesh point mode (for details of operation mode, see in Appendix). The mode sets the main functionality of the wireless link. It is possible to run in two modes at the same time. So when user space changes operation mode, mac80211 will create a new interface. This new interface runs the modes that user space has chosen.

Secondly, the configuration command is analyzed by kernel. It is handled by cfg80211. So cfg80211 has responsibility to pass value to mac80211 callbacks which is suitable for configuration command's value. In here, configuration command is "adding a new interface that runs ad-hoc mode". So, value is passed to `int (*add_interface)(struct ieee80211_hw *hw, struct ieee80211_vif *vif)`.

Now, we get back to driver. mac80211-ops.c has implemented functions `ath5k_add_interface()` that match with `int (*add_interface)(struct ieee80211_hw *hw, struct ieee80211_vif *vif)`.

So, `ath5k_add_interface()` is called to handle value from mac80211.



Inside `ath5k_add_interface()`:

The value of operation mode is store in **`vif->type`** ("type" is in "enum nl80211_iftype" type). `ath5k_add_interface()` considers if the card can add more interface and update interface counter:

```
/* Don't allow other interfaces if one ad-hoc is configured.
* TODO: Fix the problems with ad-hoc and multiple other
interfaces.
* We would need to operate the HW in ad-hoc mode to allow TSF
updates
* for the IBSS, but this breaks with additional AP or STA
interfaces
```

```
* at the moment. */
     if (ah->num_adhoc_vifs ||
         (ah->nvifs && vif->type == NL80211_IFTYPE_ADHOC)) {
          ATH5K_ERR(ah, "Only one single ad-hoc interface is
allowed.\n");
          ret = -ELNRNG;
          goto end;
     }
```

****

```
     if (avf->opmode == NL80211_IFTYPE_AP)
               ah->num_ap_vifs++;
          else if (avf->opmode == NL80211_IFTYPE_ADHOC)
               ah->num_adhoc_vifs++;
          else if (avf->opmode == NL80211_IFTYPE_MESH_POINT)
               ah->num_mesh_vifs++;
```

`ath5k_add_interface()` then calls `ath5k_update_bssid_mask_and_opmode()` (is implemented in base.c) to set mode and BSSID for new interface.

`ath5k_vif_iter()` is called in side `ath5k_update_bssid_mask_and_opmode()` to pass the value of type and calculate combined mode (when APs are active, operate in AP mode only. Otherwise use the mode of the new interface. This can currently only deal with combinations of APs and STAs. An only one ad-hoc interface is allowed).

After calculating suitable mode, `ath5k_hw_set_opmode()` is called (is implemented in pcu.c). Base on the operation mode that has been set, `ath5k_hw_set_opmode()` write the value to card register to set operation mode:

```
switch (op_mode) {
     case NL80211_IFTYPE_ADHOC:
          pcu_reg |= AR5K_STA_ID1_ADHOC |
AR5K_STA_ID1_KEYSRCH_MODE;
          beacon_reg |= AR5K_BCR_ADHOC;
          if (ah->ah_version == AR5K_AR5210)
               pcu_reg |= AR5K_STA_ID1_NO_PSPOLL;
          else
               AR5K_REG_ENABLE_BITS(ah, AR5K_CFG,
AR5K_CFG_IBSS);
          break;

     case NL80211_IFTYPE_AP:
     case NL80211_IFTYPE_MESH_POINT:
          pcu_reg |= AR5K_STA_ID1_AP |
AR5K_STA_ID1_KEYSRCH_MODE;
          beacon_reg |= AR5K_BCR_AP;
          if (ah->ah_version == AR5K_AR5210)
               pcu_reg |= AR5K_STA_ID1_NO_PSPOLL;
          else
```

```
                AR5K_REG_DISABLE_BITS(ah, AR5K_CFG,
AR5K_CFG_IBSS);
          break;

     case NL80211_IFTYPE_STATION:
          pcu_reg |= AR5K_STA_ID1_KEYSRCH_MODE
               | (ah->ah_version == AR5K_AR5210 ?
                    AR5K_STA_ID1_PWR_SV : 0);
     case NL80211_IFTYPE_MONITOR:
          pcu_reg |= AR5K_STA_ID1_KEYSRCH_MODE
               | (ah->ah_version == AR5K_AR5210 ?
                    AR5K_STA_ID1_NO_PSPOLL : 0);
          break;

     default:
          return -EINVAL;
     }
```

By using values of mac80211 callbacks, ath5k driver can implement other functions and use them to configure more parameters such as: ath5k_remove_interface(), ath5k_bss_info_changed(), ath5k_set_antenna(),ath5k_set_ringparam() …etc.

IV. TESTING SCENARIOS

In this chapter, I am going to make some test cases about RX/TX path and configuration path of ath5k source code. With my test case, I hope this helps to clarify my theory and also prove that I traced successfully ath5k function call. After tracing ath5k source code, I realized that there are some trace points which has been set in order to record packet traffic. Recorded packet traffic is stored in trace buffer so that these data can be extracted with **trace-cmd** [15] and external plugin. With configuration path, **ath5k debug mode** can be used to trace function calls.

1. RX/TX Path

trace-cmd is a user-space front-end command-line tool for **Ftrace** [16]. It interacts with Ftrace Linux kernel internal tracer. In other way, it help user space easily sets the Ftrace. *"Ftrace is a tracing utility built directly into the Linux kernel. Many distributions already have various configurations of Ftrace enabled in their most recent releases. One of the benefits that Ftrace brings to Linux is the ability to see what is happening inside the kernel. As such, this makes finding problem areas or simply tracking down that strange bug more manageable"* [16]. After installing, user uses Terminal commands to run trace-cmd.

Structure of trace-cmd command:

```
trace-cmd COMMAND [OPTIONS]
```

COMMAND includes:

- record - record a live trace and write a trace.dat file to the local disk or to the network.
- report - reads a trace.dat file and converts the binary data to a ASCII text readable format.
- start - start the tracing without recording to a trace.dat file.
- stop - stop tracing (only disables recording, overhead of tracer is still in effect)
- extract - extract the data from the kernel buffer and create a trace.dat file.
- reset - disables all tracing and gives back the system performance. (clears all data from the kernel buffers)
- split - splits a trace.dat file into smaller files.
- list - list the available plugins or events that can be recorded.
- listen - open up a port to listen for remote tracing connections.

OPTIONS are chosen by user. They are optional.

A notice in here is that: in order to use trace-cmd and ath5k debug mode of ath5k driver, I have to turn on the ath5k debugging/tracer in kernel.

Enable ath5k debug and tracer

In order to nable ath5k debug/tracer, I have to change the configuration of kernel. In this document, the Linux Operating system that I use is Ubuntu 12.10. Furthermore by doing this, a developer can combined a modified ath5k source code.

Here are steps to do it:

i. Download the source code of Linux kernel by:

```
apt-get source linux-source-version_that_you_want
```

E.g: apt-get source linux-source-3.5.0

Or if you want to build with the source of the Ubuntu kernel, use this command. This command takes more time than first one.

```
git clone git://kernel.ubuntu.com/ubuntu/ubuntu-quantal.git
```

Note: I am using Ubuntu 12.10 so the source in my command is “ubuntu-quantal.git”. Change “ubuntu-quantal.git” to your Ubuntu version

ii. After downloading source code of kernel, I got a compressed file. Copy it to folder /usr/src by “cp” command with root privilege.

iii. Extract compressed file.

iv. Move to extracted folder and copy the kernel config file from existing system to the kernel tree:

```
cp /boot/config-`uname -r` .config
```

v. Bring the config file in source code folder up to date.

```
yes '' | make oldconfig
```

vi. Now I change the configuration of ath5k driver by running:

```
make menuconfig
```

To run above command, I have to install some tools:

```
sudo apt-get install git kernel-package fakeroot build-essential ncurses-dev
```

Following these steps:

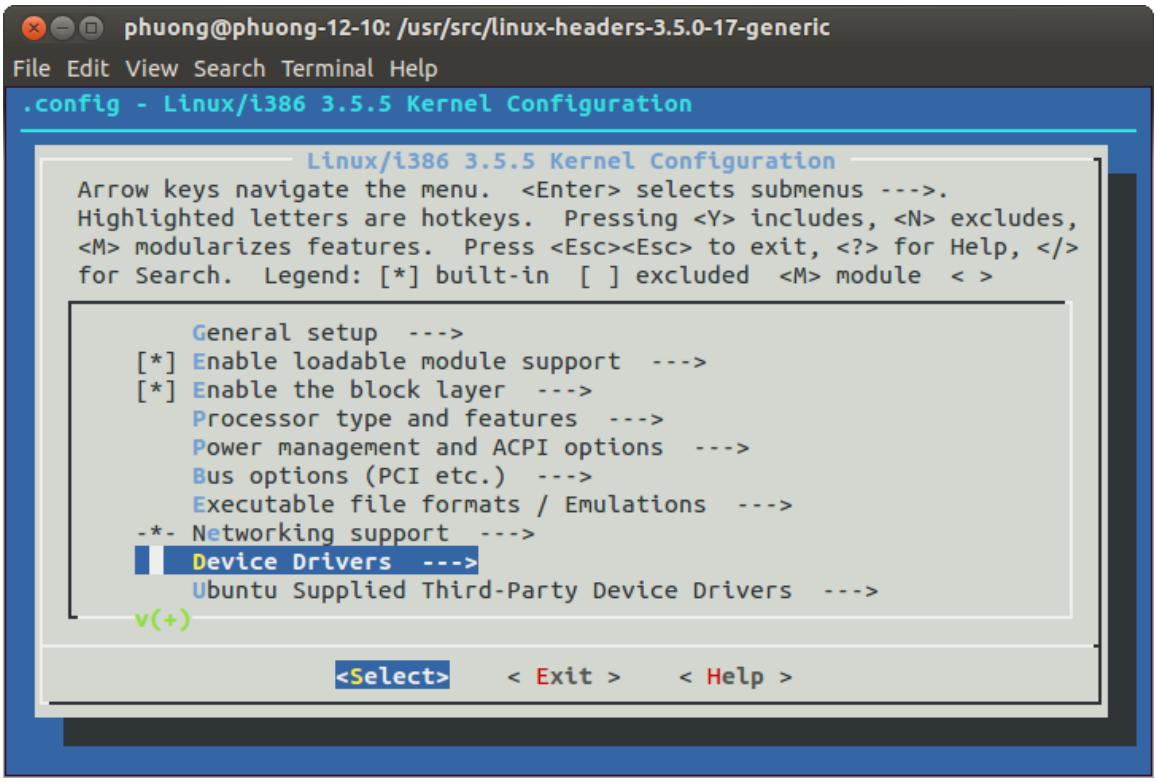


Figure 24: A menu appears after running "make menuconfig", choose "Device Driver"

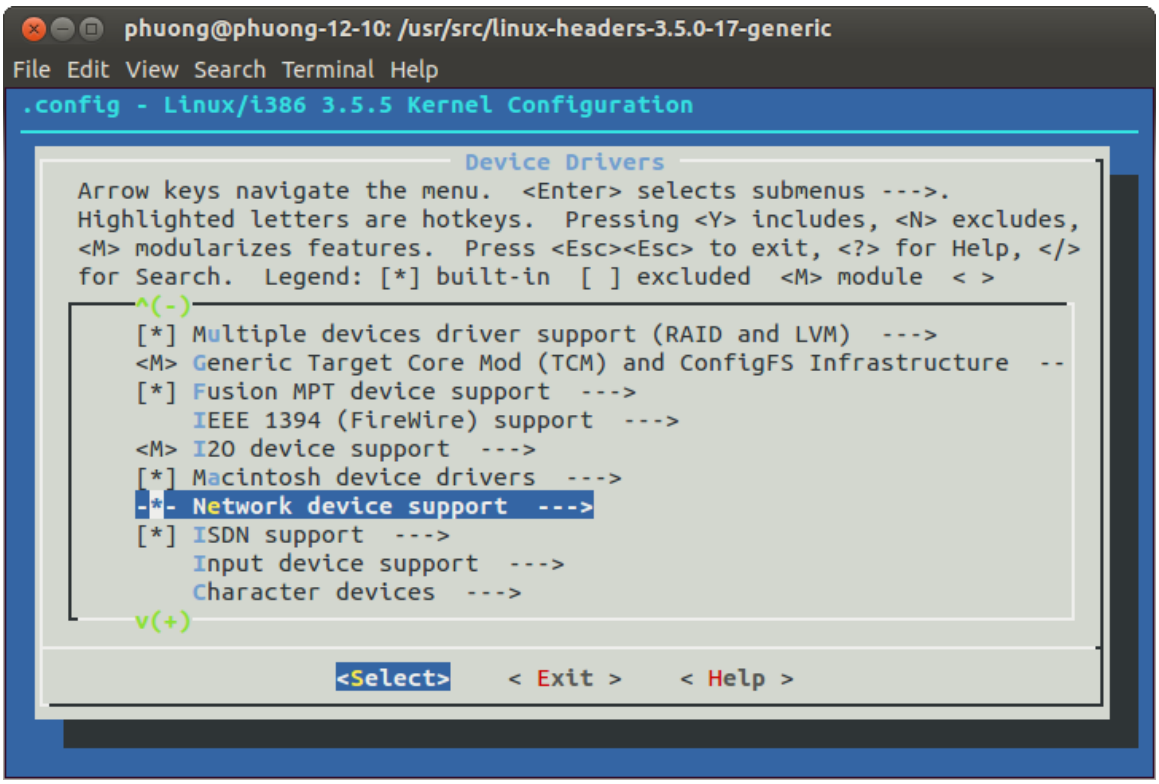


Figure 25: Choose "Network device support"

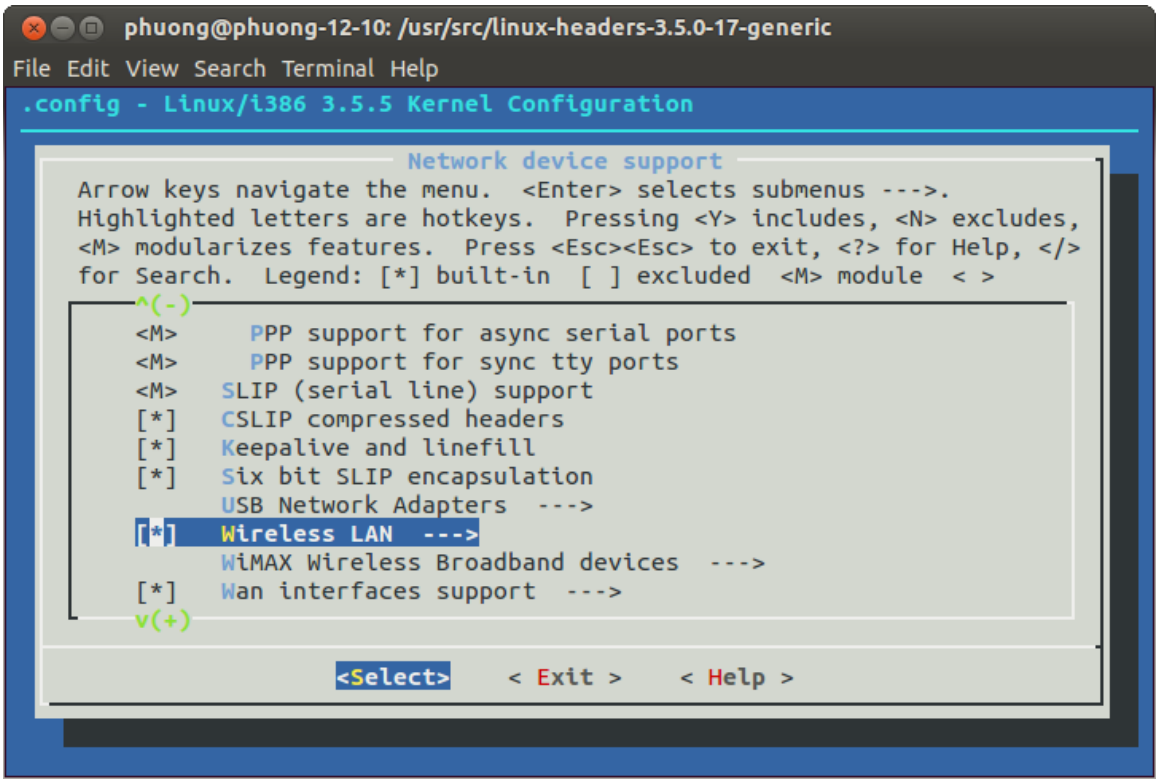


Figure 26: Choose "Wireless LAN"

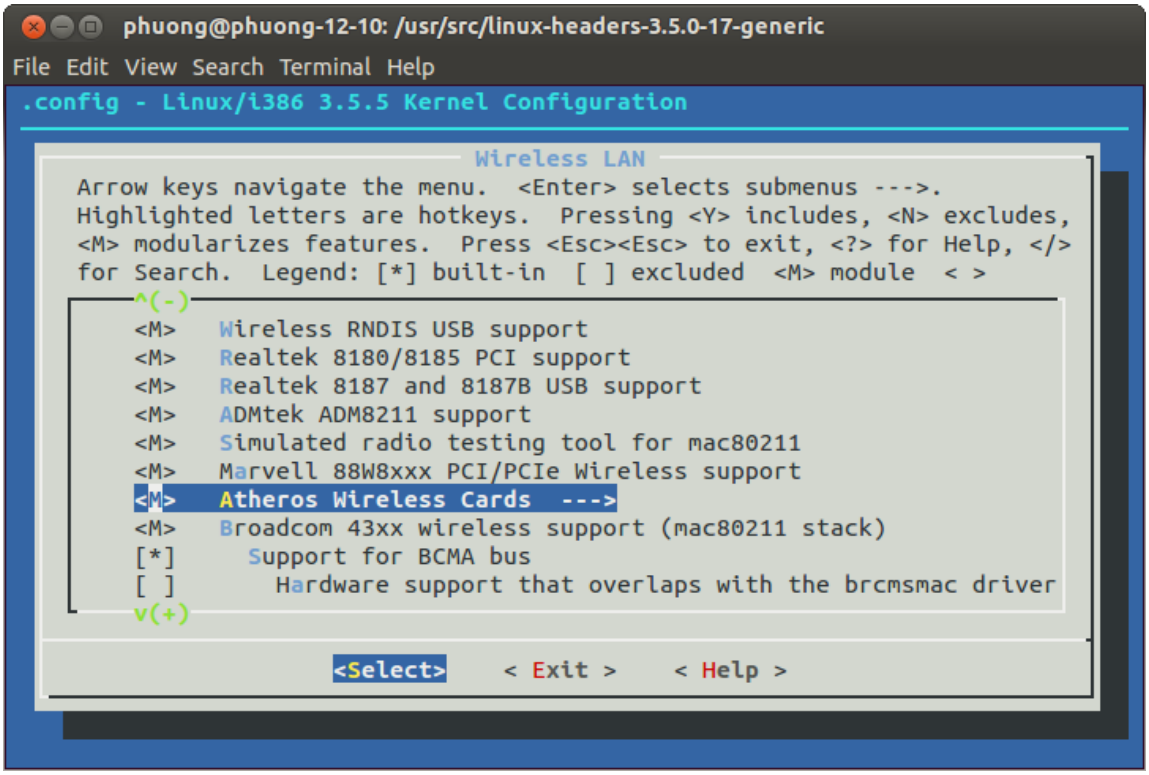


Figure 27: Choose "Atheros Wireless Card"

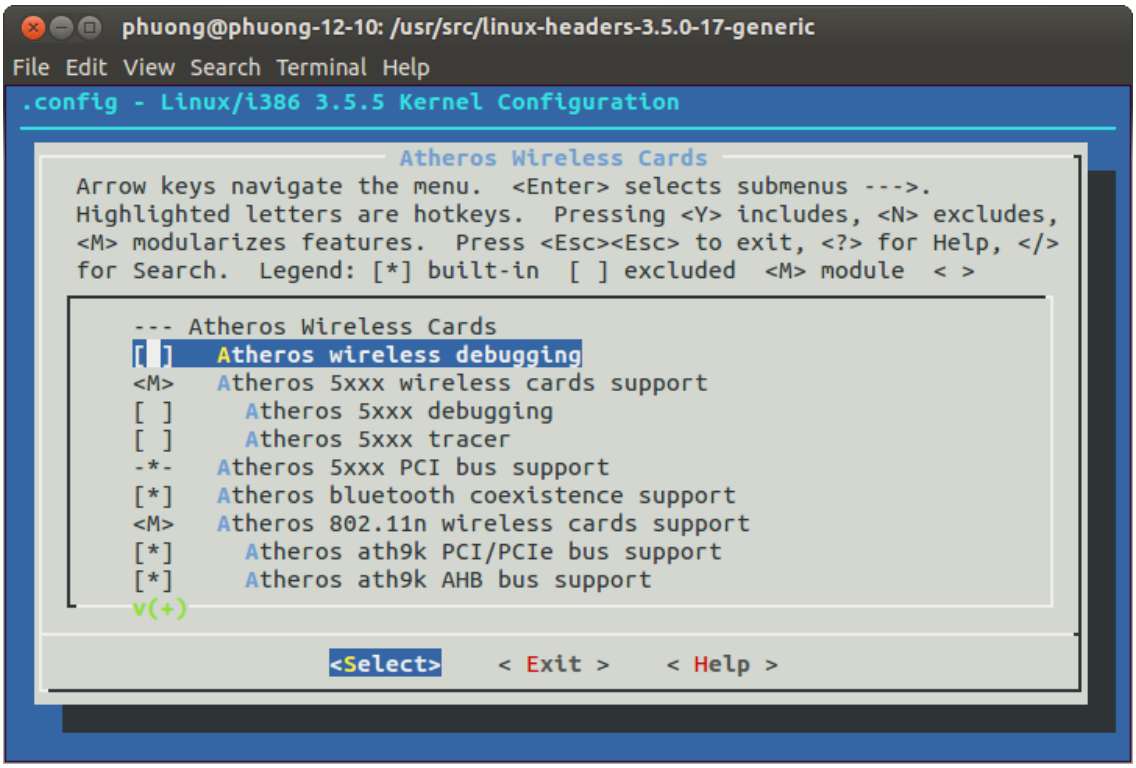


Figure 28: Enable "Atheros wireless debugging", "Atheros 5xxx debugging" and "Atheros 5xxx tracer" by pressing Space Bar

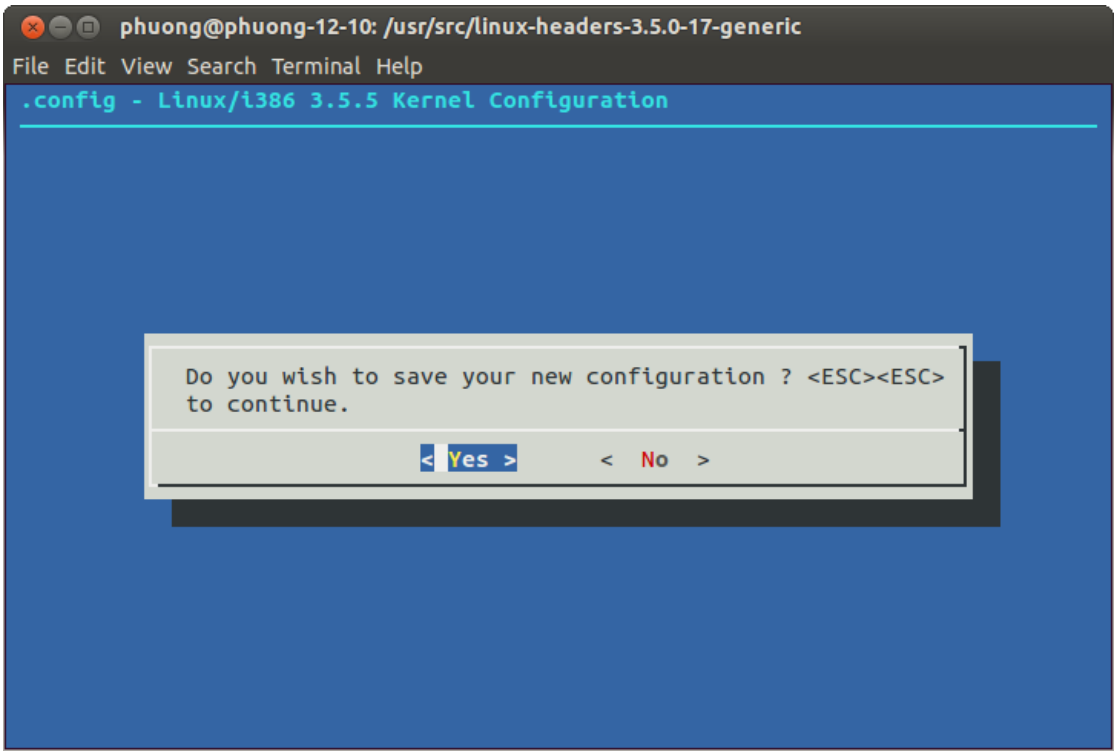


Figure 29: Press ESC several times until this menu appear, choose "SAVE"

vii. Clean the kernel source directory.

```
make-kpkg clean
```

viii. Build the **linux-image.deb** and **linux-header.deb** files (CONCURRENCY_LEVEL can also be set manually to how many CPUs/cores to use to build the kernel). This process takes a lot of time.

```
CONCURRENCY_LEVEL=`getconf _NPROCESSORS_ONLN` fakeroot make-kpkg --initrd --append-to-version=-custom kernel_image kernel_headers
```

The name of compiled kernel can be changed to something else by changing value of “custom” of “--append-to-version” option to something. In my case, I changed “custom” to something like “ath5k+debug+”. This is optional. The compiling step should take a while of time.

ix. After finish building the .deb file, Change to one directory level up (this is where the linux-image and linux-header .deb files were put)

```
cd ..
```

x. Run the command “ls”, two .deb files are there. Now install the .deb files.

```
sudo dpkg -i linux-image-something_here_.Custom.deb
```

```
sudo dpkg -i linux-headers-something_here_.Custom.deb
```

xi. Make the new kernel bootable.

```
sudo update-initramfs -ck module_name_for_new_kernel
```

```
sudo update-grub
```

xii. New kernel with ath5k debug/tracer is now ready. Just make sure to select the new kernel when reboot.

After enabling ath5k debug/tracer, now I can use trace-cmd to trace ath5k function calls. In my test case, I just need trace-cmd to record a trace and then view its report. So, I am going to use the command trace-cmd record. The full option of trace-cmd record is:

```
trace-cmd record [OPTIONS] [command]
```

The trace-cmd record command will set up the Ftrace Linux kernel tracer to record the specified plugins or events that happen while the command executes. If no command is given, then it will record until the user hits Ctrl-C.

The record command of trace-cmd will set up the Ftrace tracer to start tracing the various events or plugins that are given on the command line. It will then create a number of tracing processes (one per CPU) that will start recording from the kernel ring buffer straight into temporary files. When the command is complete (or Ctrl-C is hit) all the files will be combined into a trace.dat file that can later be read by command trace-cmd report

All things to handle with my test case are writing a suitable trace-cmd command, reading the trace-cmd output and comparing output with my theory. I am going to explain each step by step.

Write a suitable trace-cmd command:

I want to trace functions call, so the command is:

```
trace-cmd record
```

When recording a record, plugins can be used to help user analyzes easily output later. The "-p" is used to add a plugin. Plugins are special Ftrace tracers that usually do more than just trace an event. Common plugins are function, function_graph, preemptirqsoff, irqsoff, preemptoff, and wakeup. A plugin must be supported by the running kernel. There are multiple plugins but only two of them are used usually: function (tracing function, list it in every line) and function_graph (tracing function, list to a group of function call). I am going to use function_graph. My command now is:

```
trace-cmd record –p function_graph
```

I now set trace-cmd to trace with function_graph plugin. The only thing that need to be set is what to trace. The functions calls which I want to trace are ath5k's functions. They are started by key-word "ath5k", so I set up trace-cmd to trace function that start with "ath5k". The "-l" OPTIONS will limit the function and function_graph tracers to only trace the given function name. The command now is:

```
trace-cmd record –p function_graph –l ath5k*
```

(with ath5k* to only filter functions that start with ath5k. *ath5k to only filter functions that end with ath5k. *ath5k* to only filter on functions that contain ath5k)

More than one -l may be specified on the command line to trace more than one function. In here, I can add one more “-l” to view mac80211 function calls. The command to trace will be:

```
trace-cmd record –p function_graph –l ath5k* -l ieee80211_rx -l ieee80211_tx*
```

Another note that: from kernel 3.x, ath5k source code has implemented some trace-points when TX/RX occurs. The functions that RX/TX frame are not traced by the trace-cmd but they are set as an event trace. An event trace can be seen in trace-cmd list. So if I want to see when RX/TX happens, I have to add “-e ath5k” which means that also trace RX/TX events. The final trace-cmd command is:

```
trace-cmd record –p function_graph –l ath5k* -l ieee80211_rx*-l ieee80211_tx* -e ath5k
```

After record for a while, press Ctrl+C to stop. In current directory, there should be a **trace.dat** file. Run this command to view the trace:

```
trace-cmd report | less
```

Here are results after executing above command:

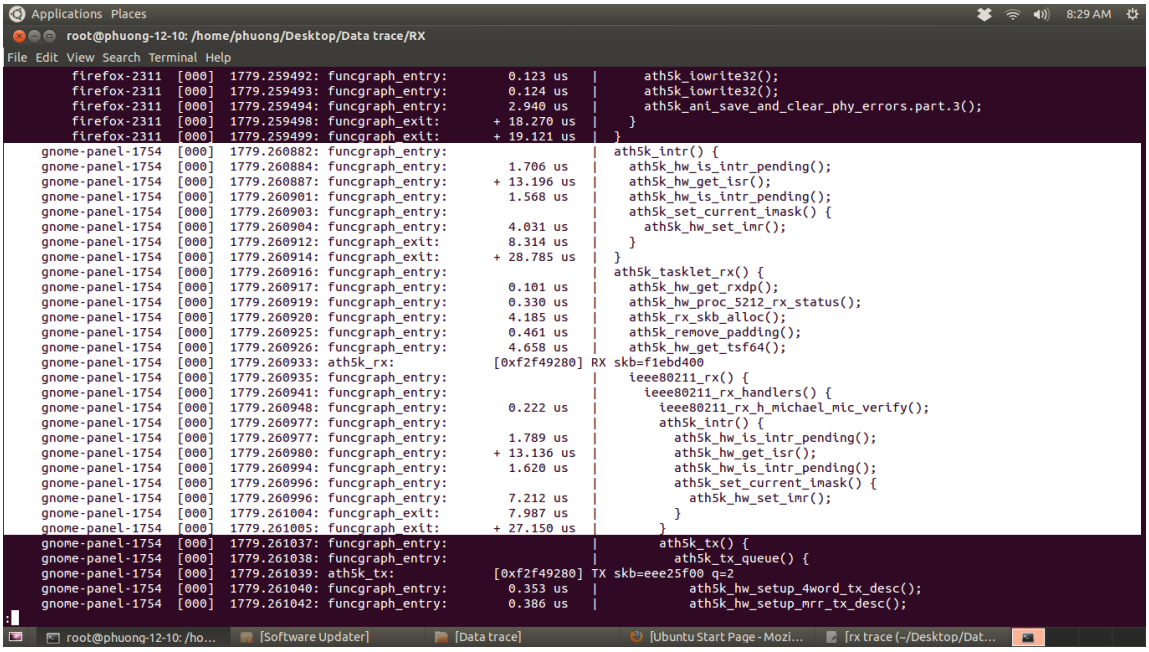


Figure 30: "trace-cmd report" output – RX Path

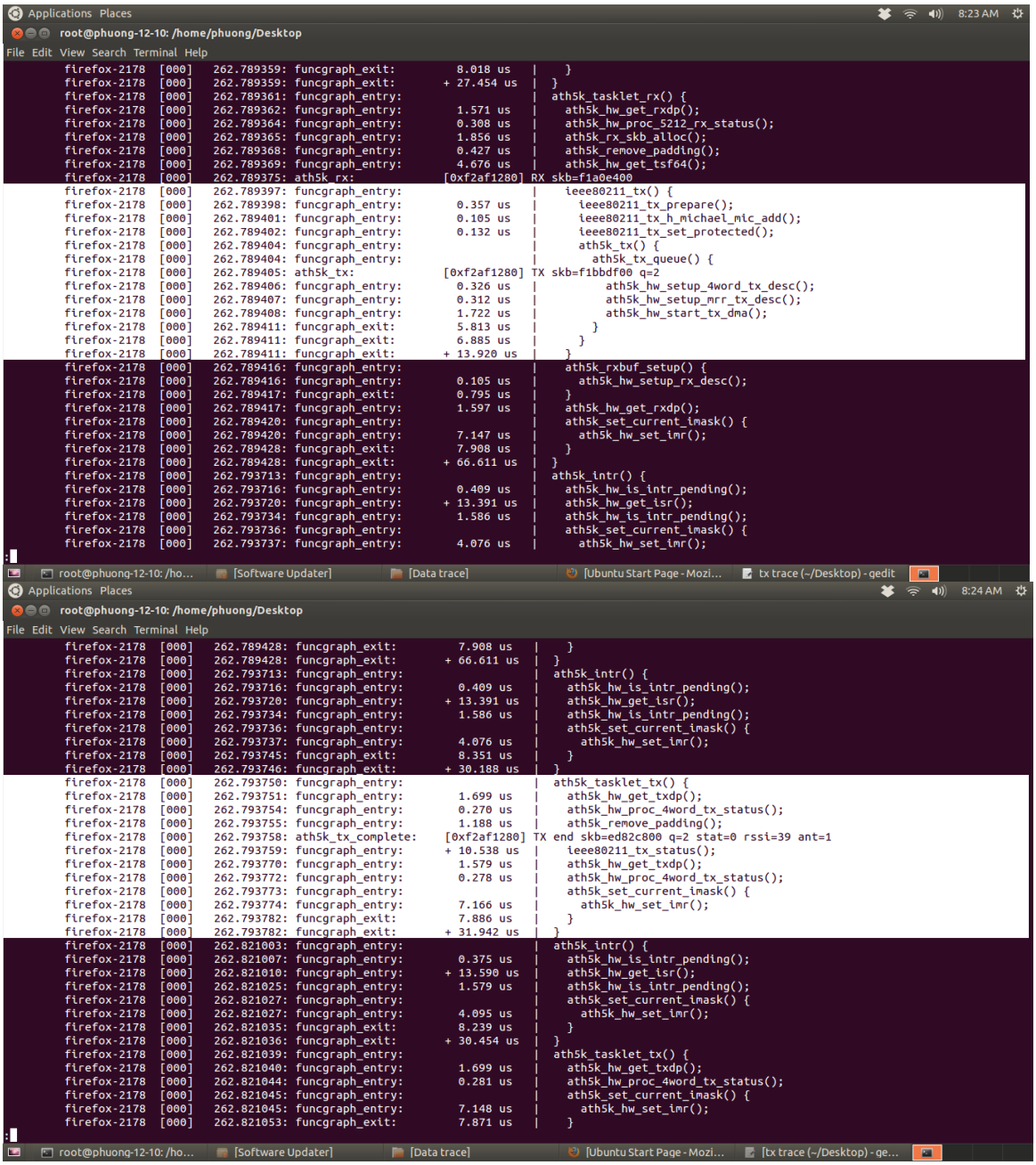


Figure 31: "trace-cmd report" output – TX Path

Read and compare results with my theory:

Now I have result with trace-cmd. Next step, I am going to compare this result with my theory to see that my theory is right or wrong.

With RX Path:

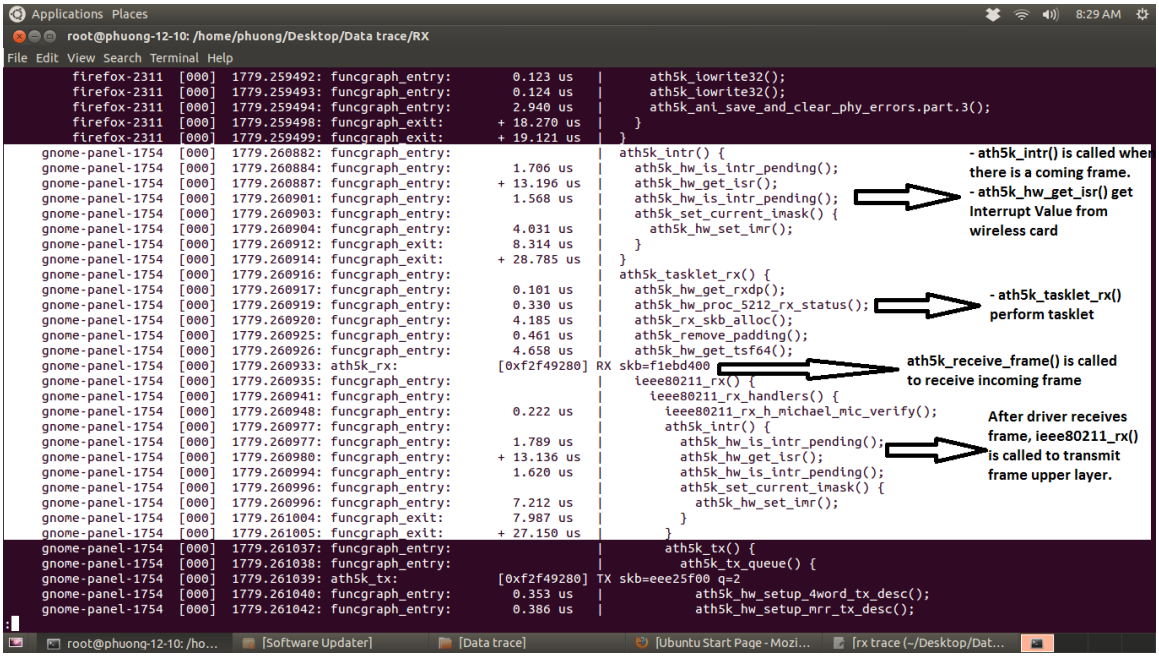


Figure 32: Analyzing RX Path

I extract function call from my testing – a RX Path and compare with my theory. In the righ of below figure, the RX Path has been summaried. In the left, I extract function call from my testing. I can see clearly that there are matching points between testing and my theory

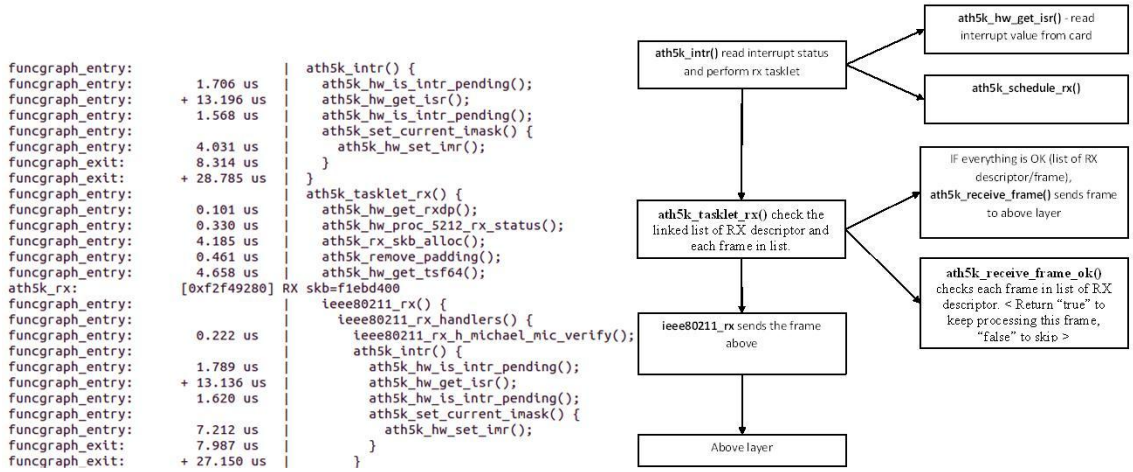


Figure 33: Compare testing result and theory with RX Path

With TX Path:

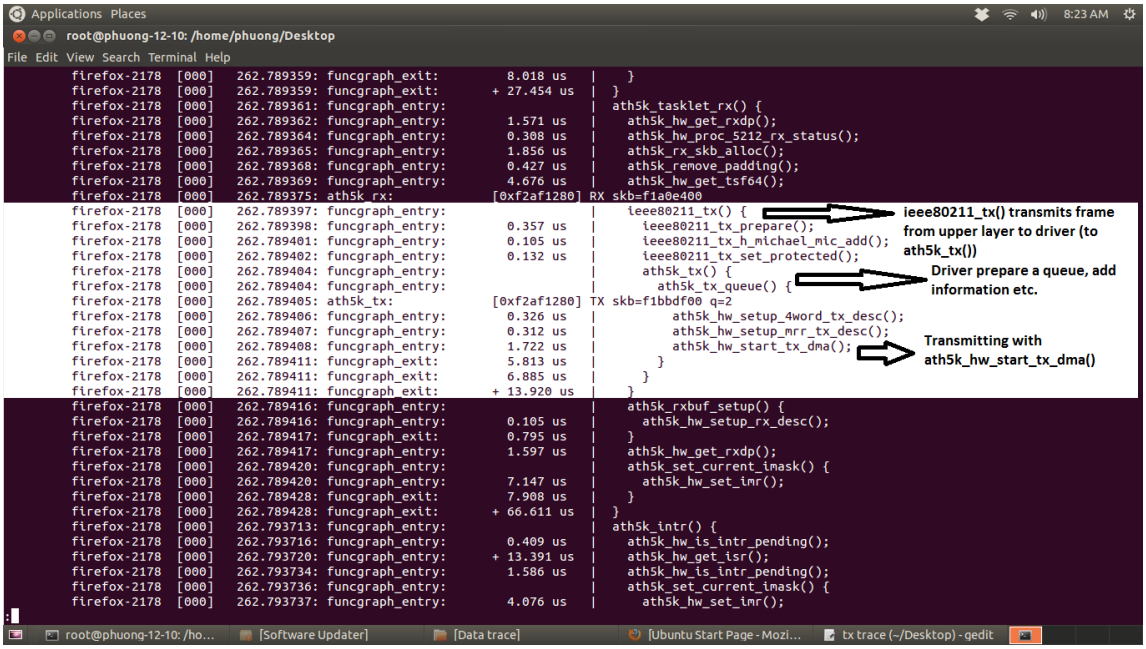


Figure 34: Analyzing TX Path

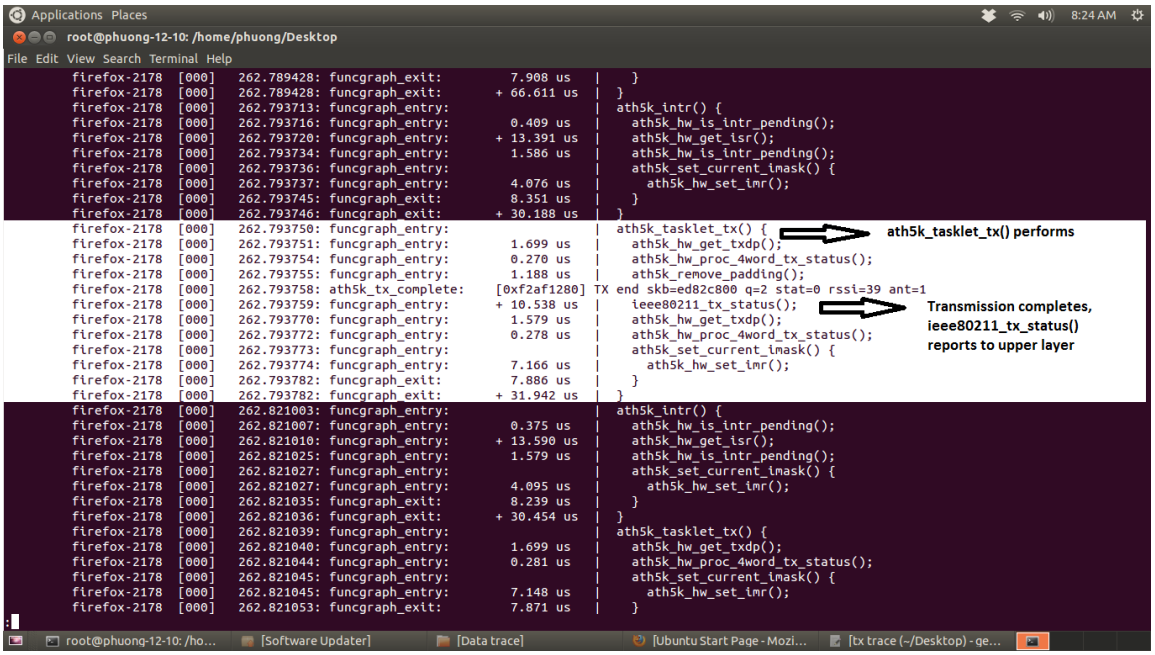


Figure 35: Analyzing TX Path (cont.)

I also compare the testing TX Path with my theory too. They are the same between the testing and my theory.

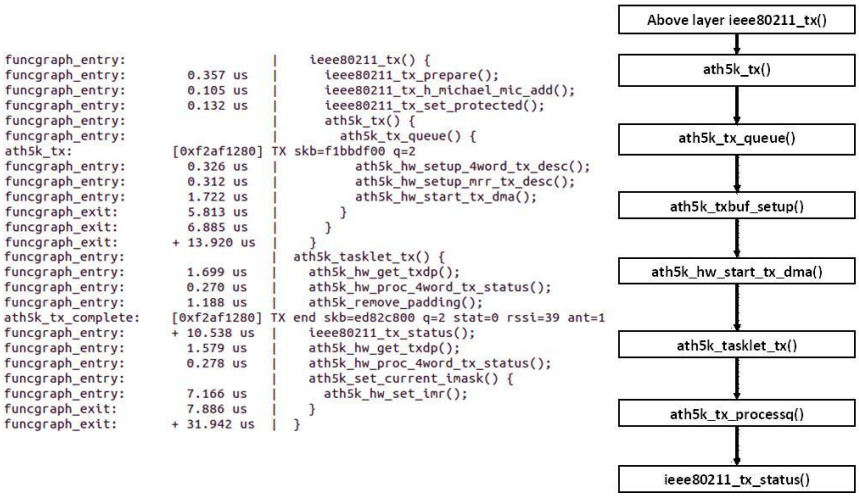


Figure 36: Compare testing TX Path and my theory

2. Configuration Path

With configuration path, ath5k debug mode can help users to trace function calls. After enabling ath5k debug mode in kernel, a folder naming ath5k appears inside **/sys/kernel/debug/ieee80211/phy0**.

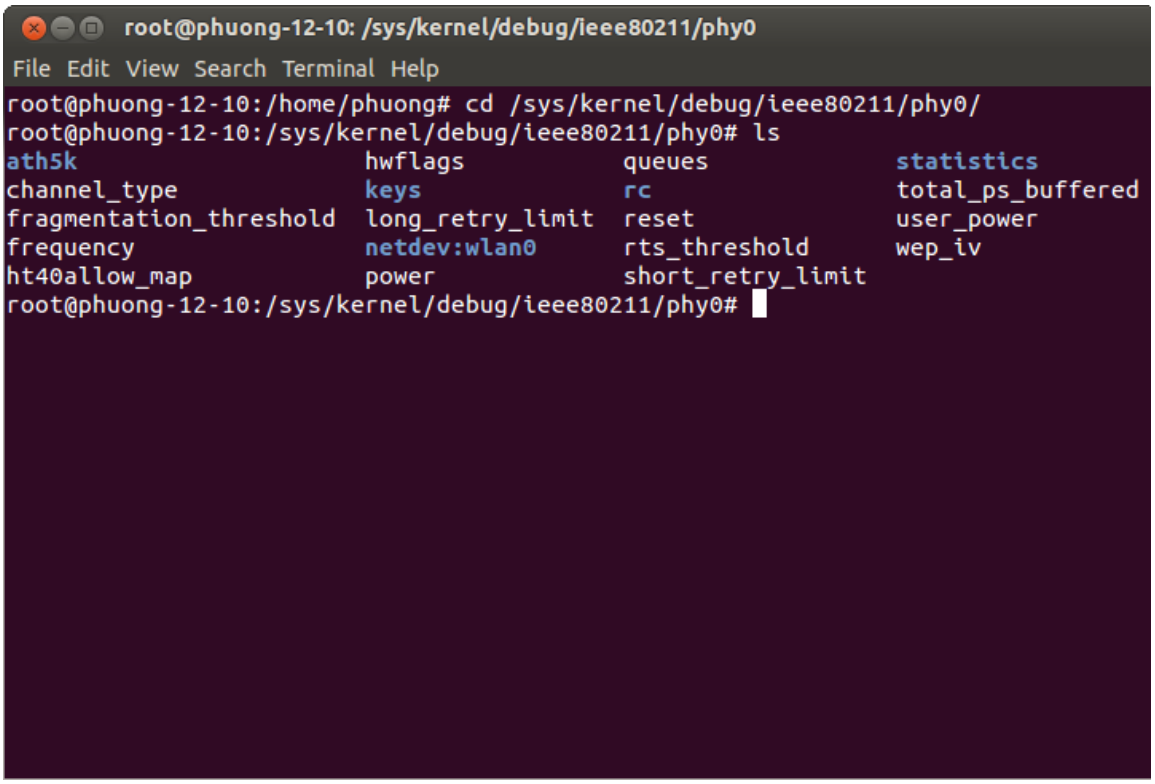


Figure 37: ath5k debug folder location

This ath5k folder contains information of ath5k debug mode. Users can set the type of debug that he wants by changing value of “debug” inside ath5k folder. These are all the types of debug:

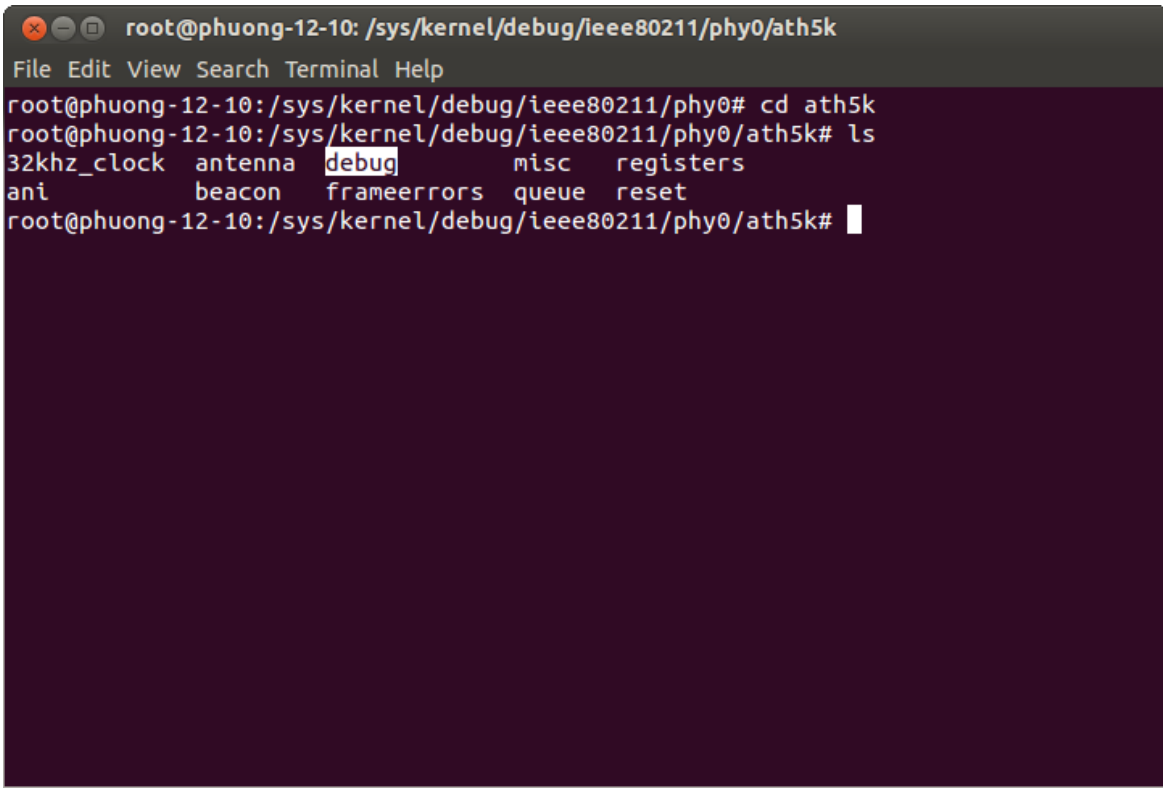


Figure 38: Inside ath5k debug folder – "debug" stores current information of ath5k debug

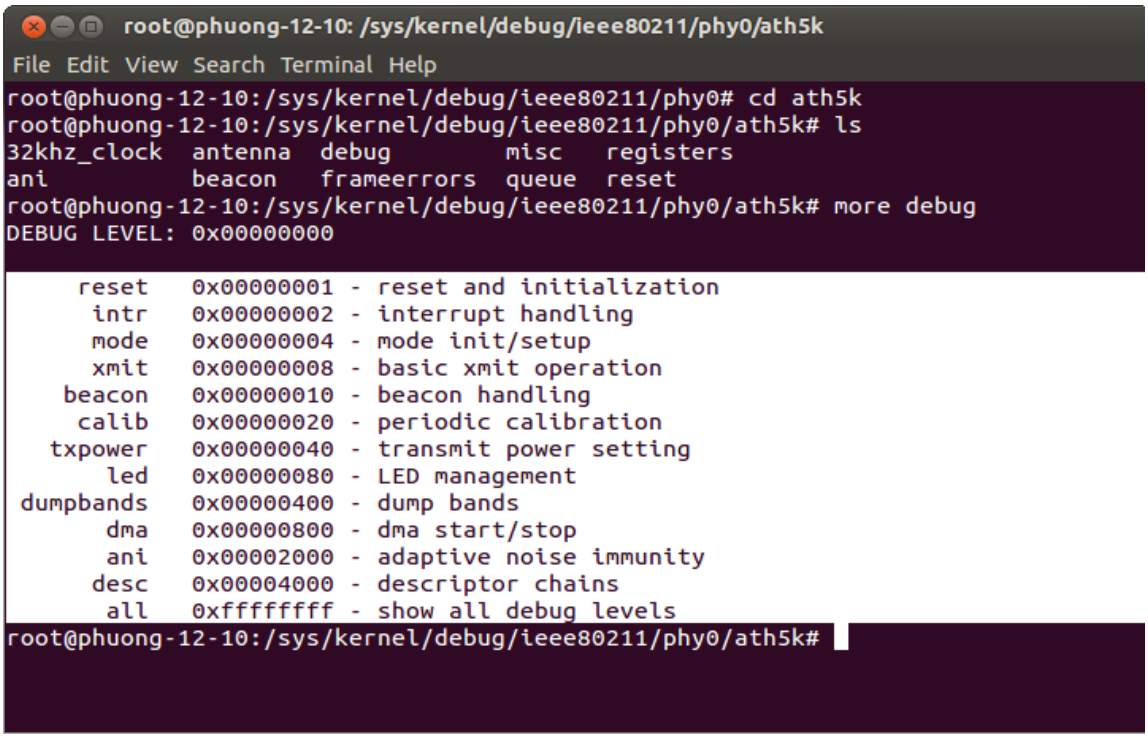


Figure 39: Supported modes of debug

Users choose the type of debug by this command:

echo “type_of_debug_mode” > /sys/kernel/debug/ieee80211/phy0/ath5k/debug

In my test case, I want to use “mode” debug. So my command is:

echo “mode” > /sys/kernel/debug/ieee80211/phy0/ath5k/debug

After running above command, “mode” debug is enabled.

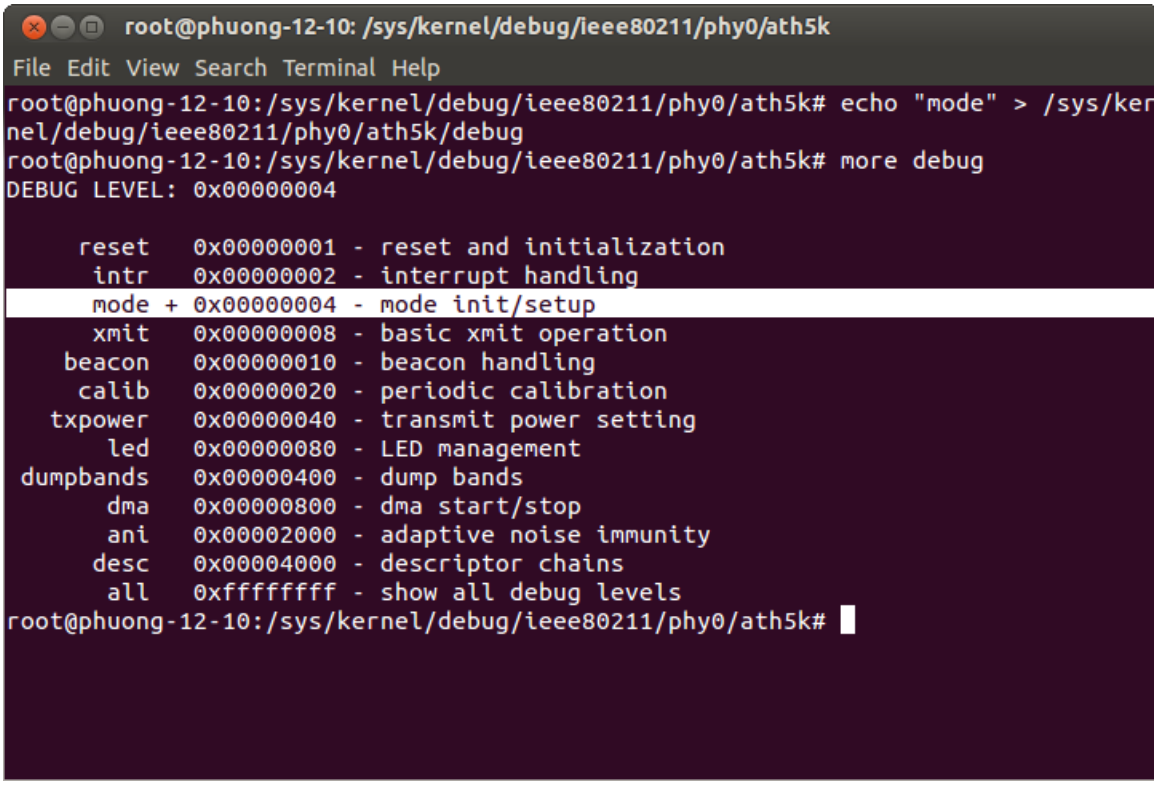


Figure 40: "mode" in debug mode is enabled

Next, I try to change operation mode of the wireless card. Since “mode” debug has been enabled, all function calls of operation mode will be recorded to “dmesg” [17] (Linux command to show driver message). I am going to do something like:

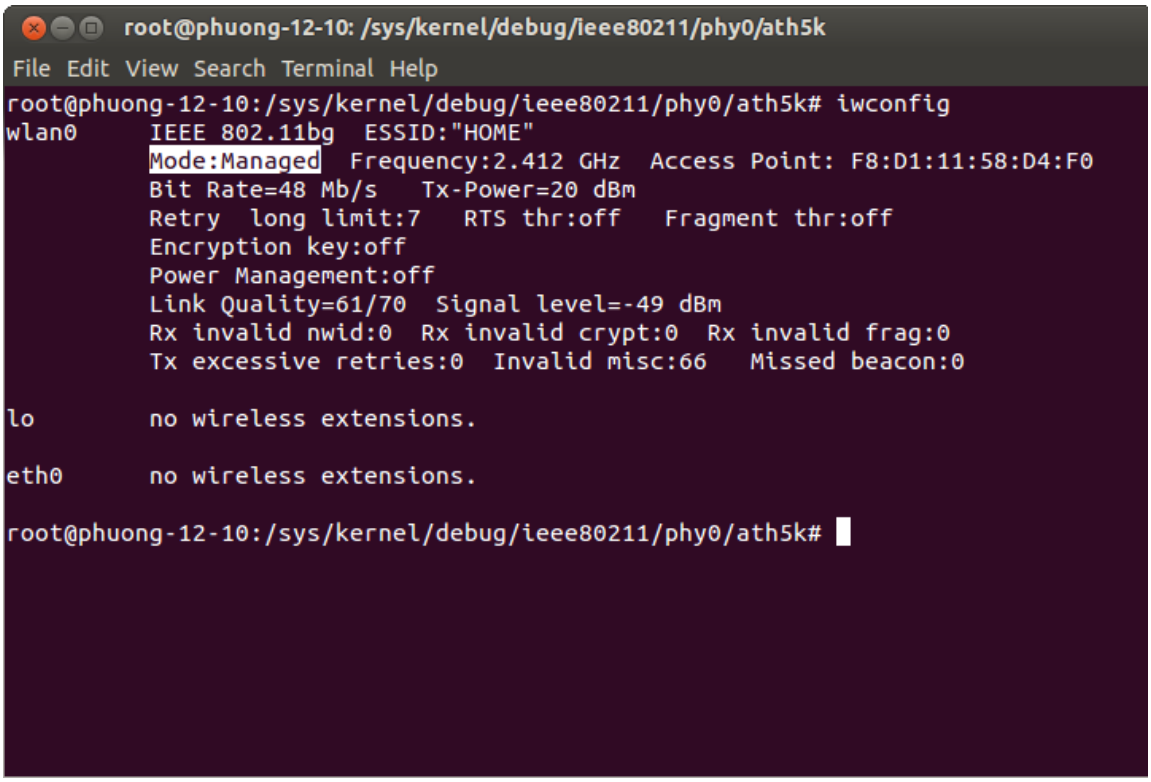


Figure 41: The operation mode is now in the Managed mode.

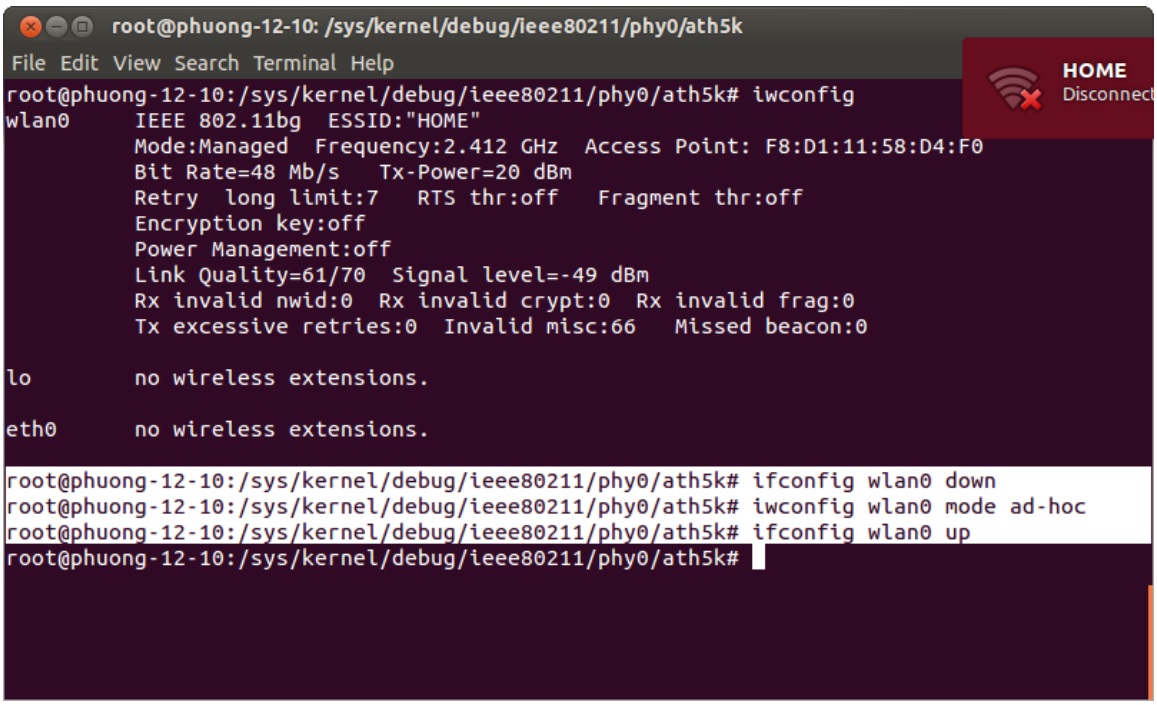


Figure 42: Changing the operation mode to Ad-hoc.

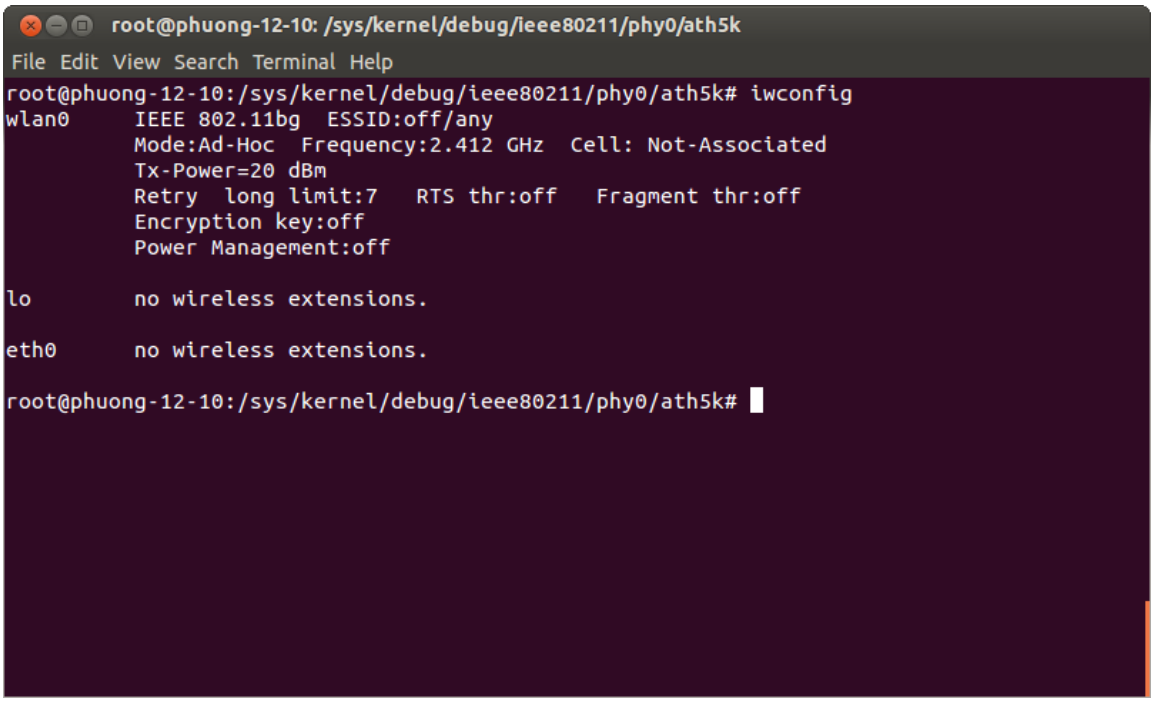


Figure 43: Operation mode is now Ad-hoc

After changing operation mode of the wireless card to AD-HOC, I try to view driver messages by command:

```
dmesg | grep ath5k
```

(“grep ath5k” means that show only driver message of ath5k driver)

And here is the result:

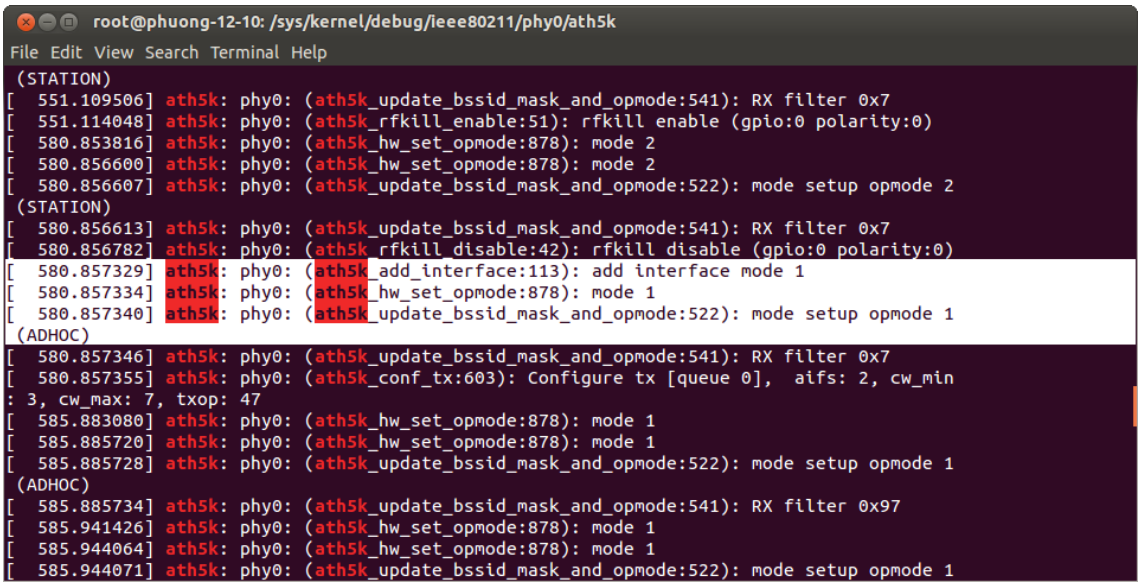
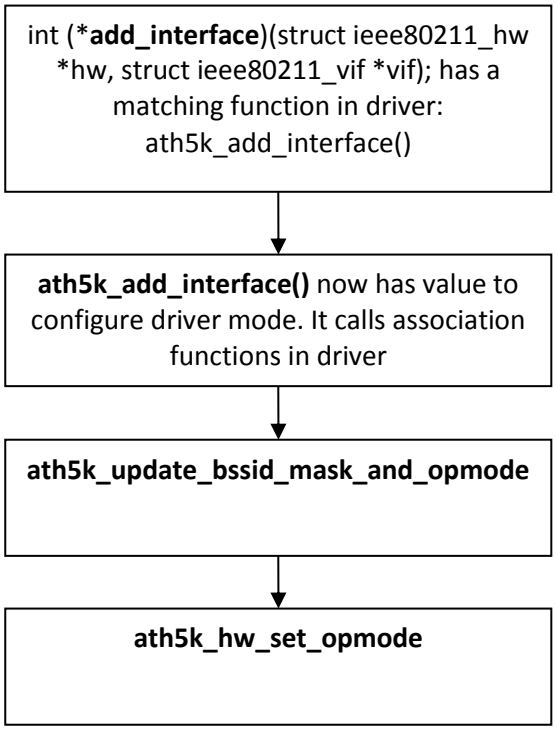


Figure 44: "mode" debug result

The function calls when ath5k try to set operation mode of wireless card in my test are the same with my theory.



In conclusion, all test cases above have showed that the function calls when RX/TX and Configuration in my theory are the same.

CONCLUSION

The purpose of this study is building a brief overview about an open source project of Linux platform that is ath5k driver for WLAN card using Atheros chipsets. Within the thesis's objective I have two primary goals. The first goal is to understand the basic operation of the ath5k driver by analyzing source code. The second is to test my theory and show the way how to debug/compile ath5k source code.

I propose the basic operation of ath5k such as: how a frame is sent or received by ath5k source code, how user space can configure a WLAN card using ath5k driver. I hope this helps further researches about ath5k can save time of researching.

With my test cases, I propose the way how to compile ath5k driver that has been modified and use debug to view or test ath5k driver operation.

My research was still limited. It only focused on theory. However some sections such as "Configuration Path" can help to solve further topics, e.g. writing function to adjust some parameters of a WLAN card using ath5k driver.

REFERENCES

[1]. “A bit more about the technologies involved” - Jean Tourrilhes - Hewlett Packard Laboratories, Palo Alto - 3 August 2000.
[2]. “Medium Access Protocol” definition in Wikipedia http://en.wikipedia.org/wiki/Media_access_control
[3]. “Dynamic source routing in ad hoc wireless networks” - *D. Johnson* and *D. Maltz* - in *Mobile Computing, Kluwer Academic Publishers, 1996*.
[4]. Linux Wireless – About ath5k - http://wireless.kernel.org/en/users/Drivers/ath5k
[5]. MadWifi project - http://madwifi-project.org/
[6]. Madwifi project – About OpenHal - http://madwifi-project.org/wiki/About/OpenHAL
[7]. The Linux Kernel Archieves - http://www.kernel.org/
[8].Tasklet mechanism - http://www.makelinux.net/ldd3/chp-7-sect-5
[9]. Linux Wireless – mac80211 - http://wireless.kernel.org/en/developers/Documentation/mac80211
[10]. Linux Wireless – cfg80211 - http://wireless.kernel.org/en/developers/Documentation/cfg80211
[11]. Linux Wireless – nl80211 - http://wireless.kernel.org/en/developers/Documentation/nl80211
[12]. “Wireless Extensions for Linux” – Jean Tourrilhes - 23 January 1997
[13]. “mac80211 Overview” – Johannes Martin Berg – 25 February 2009
[14]. Kernel doc – ioctl - http://www.kernel.org/doc/man-pages/online/pages/man2/ioctl.2.html
[15]. trace-cmd: A front-end for Ftrace - https://lwn.net/Articles/410200/
[16]. Debugging the kernel using Ftrace- http://lwn.net/Articles/365835/ and http://lwn.net/Articles/366796/
[17] “dmesg” command - http://en.wikipedia.org/wiki/Dmesg

Appendix A: Operation modes of a wireless card

A Wireless Network Interface Card always operates in one of the following operating modes. The mode sets the main functionality of the wireless link. It is possible to run in two modes at the same time.

Station (STA) infrastructure mode: Any wireless driver is capable of running this mode. Thus it could be called the default mode. Two WNICs in STA mode cannot connect to one another. They require a third WNIC in AP mode to manage the wireless network! A WNIC in STA mode connects to a WNIC in AP mode, by sending certain management frames to it. This process is called the authentication and association. After the AP sent the successful association- reply, the STA is part of the wireless network. This mode is also called managed in the fully depreciated WEXT tools (e.g. iwconfig)

Access Point (AP) infrastructure mode: In a managed wireless network the Access Point acts as the Master device. It holds the network together by managing and maintaining lists of associated STAs. It also manages security policies. The network is named after the MAC-Address (BSSID) of the AP. The human readable name for the network, the SSID, is also set by the AP. To use AP mode in Linux you need to use hostapd, at least a current 0.6 release, preferably from git. Cf. http://wireless.erley.org

Monitor (MON) mode: Monitor mode is a passive-only mode, no frames are transmitted. All incoming packets are handed over to the host computer completely unfiltered. This mode is useful to see what's going on the network. With mac80211, it is possible to have a network device in monitor mode in addition to a regular device; this is useful to observe the network whilst using it. However, not all hardware fully supports this as not all hardware can be configured to show all packets while in one of the other operating modes. Monitor mode interfaces always work on a "best effort" basis. With mac80211, it's also possible to transmit packets in monitor mode, which is known as packet injection. This is useful for applications that wish to implement MLME work in user space, for example to support nonstandard MAC extensions of IEEE 802.11.

Ad-Hoc (IBSS) mode: The Ad-Hoc mode aka IBBS (Independent Basic Service Set) mode, is used to create a wireless network without the need of having an AP in the network. Each station in an IBSS network is managing the network itself. Ad-Hoc is useful for connecting two or more computers to each other when no (useful) AP is around for this purpose.

Wireless Distribution System (WDS) mode: The Distribution System is the wired uplink connection to an AP. The Wireless Distribution System is the wireless equivalent to it.

WDS serves as a wireless communication path between cooperating APs (usually in a single ESS), it can be used instead of cabling. Read iw WDS documentation for details on how to enable this, but also review and consider using 4-address mode.

Mesh: Mesh interfaces are used to allow multiple devices to communication with each other by establishing intelligent routes between each other dynamically.

Appendix B: DMA engine

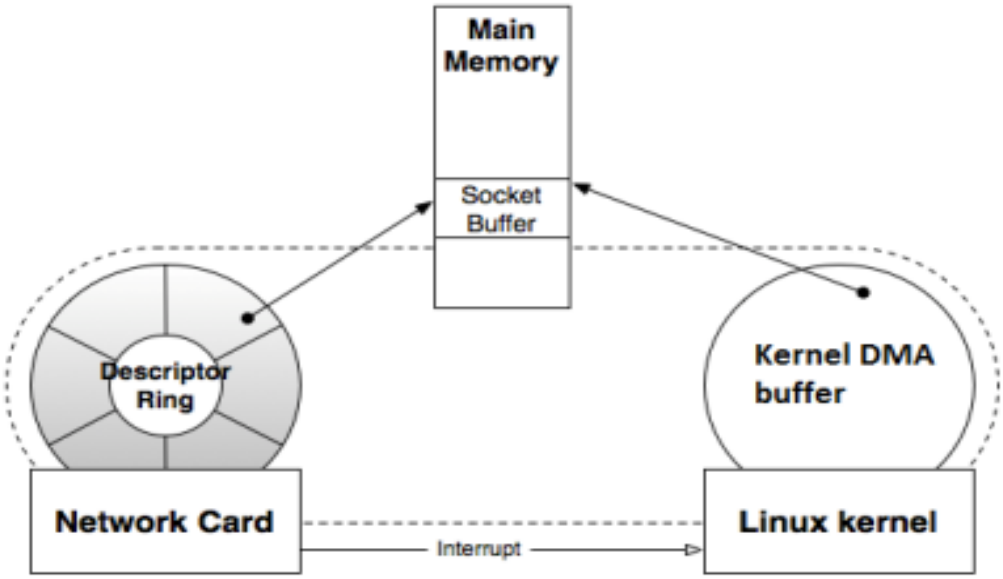


Figure: main components involved in the low level packet: main memory, the network card, and the Linux kernel

In above figure, we can see the main components involved in the low level packet. Wireless network card holds a descriptor ring (it may be called hardware descriptor), a pointer in the descriptor ring points to a socket buffer which is located in main memory. The packets will be prepared and created in socket buffer. Socket buffer also contains the state of the network too. In Linux kernel, a Kernel DMA buffer is maintained. Its functionality is to keep the temporary buffer packet so the network card can go back to grab a new one. Interrupt can happen between the direct communications from network card to the kernel. This figure has showed the operation of DMA engine.

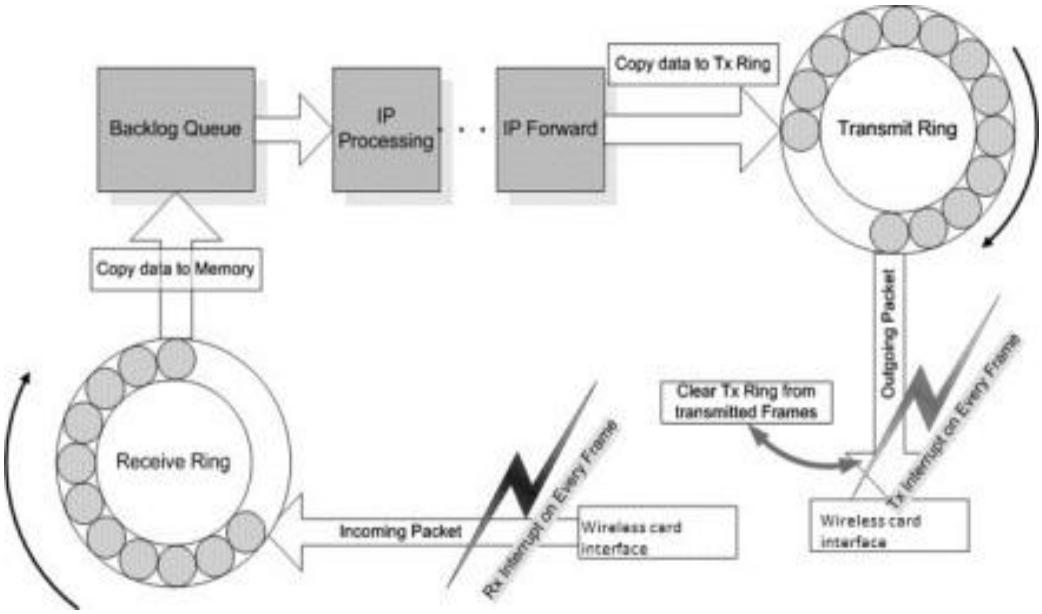


Figure: process of transmitting and receiving data

In above figure, the operation of sending and receiving packets are displayed. There are Descriptor Rings, which are Receive Ring (or RX Descriptor) and Transmit Ring (or TX Descriptor), to keep the sending and receiving packets. These rings are implemented in DMA engine that is used to send and receive packets of ath5k driver. DMA is the hardware mechanism that allows peripheral components to transfer their I/O data directly to and from main memory without the need for the system processor to be involved in the transfer. Use of this mechanism can greatly increase throughput to and from a device, because a great deal of computational overhead is eliminated.