## Linux Kernel and Modules

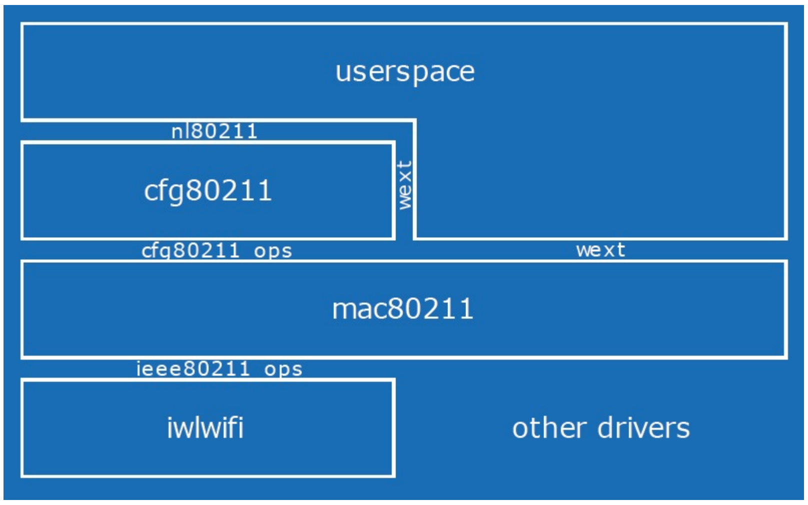
A kernel is the core of an operating system. The operating system is all of the programs that manages the hardware and allows users to run applications on a computer. The kernel controls the hardware and applications. Applications do not communicate with the hardware directly, instead they go to the kernel. In summary, software runs on the kernel and the kernel operates the hardware.

Modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system. For example, one type of module is the device driver, which allows the kernel to access hardware connected to the system. For our implementation we will be dealing with ath9k and mac80211 modules.Any Changes made to the source code will reflect in the module after compiling.

## Linux Networking Interface and Device Drivers

Each network interface device must have a driver, and the driver provides initialization functions that are called at kernel start up or module load time. The driver also includes tables that allow the device to work with the Linux kernel. All Linux network interface devices can be associated with the kernel in two different ways, either at kernel build time or later by loading once the kernel is up and operating. In addition, a Linux network interface driver can be implemented as a module so it is not necessary to statically link it into the kernel, and generally, it is preferable to implement the driver as a module. Whether modules are statically linked into the kernel is specified at kernel configuration time Driver modules can be loaded dynamically at boot time or later using the insmod command. Whether the driver is written as a module or is statically linked, the driver must provide an initialization function that is called by the kernel before the network interface device is ready to receive and transmit packets.

The Kernel Stack According to Driver



## Important structure of a Network Kernel

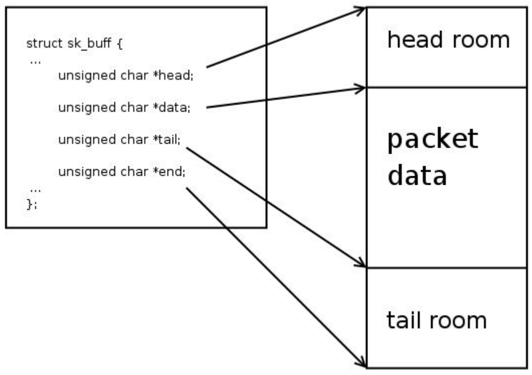
### sk_buff :

The socket buffer, or "SKB", is the most fundamental data structure in the Linux networking code. Every packet sent or received is handled using this data structure. Its a Doubly Linked List Structure its content is very large. Here we just include the used variable of the sk_buff. sk_buff is in linux header directory in skbuff.h.

**unsigned int len** ,

The total number of bytes in the packet is 'len'

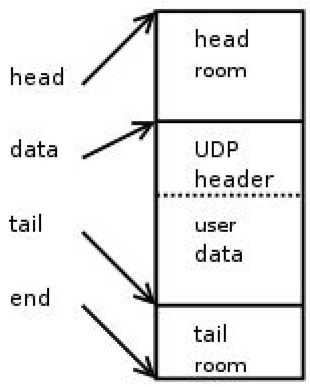
**unsigned char *head, *data, *tail, *end;**



```
struct sk_buff {
...
    unsigned char *head;
    unsigned char *data;
    unsigned char *tail;
    unsigned char *end;
...
};
```

**Important functions**

**skb_push()** will decrement the 'skb->data' pointer by the specified number of bytes. It will also increment 'skb->len' by that number of bytes as well. The caller must make sure there is enough head room for the push being performed. eg: This is what a new SKB looks like after we push the UDP header to the front of the SKB



**skb_pull()** - remove data from the start of a buffer

**skb_trim()** - remove data from the end of a buffer

### net_device:

Although we used the structure but we didn't explore it too much. We passed this as it is and didn't manipulate any variable of the structure.

## Debugging the Source

For debugging the source we use "printk()" function to display value of any variable.

eg: if inside a module we need to know the value of variable "int *a" we will write

printk(KERN_INFO "The Value of a is – %d, a);

This will print value of a. But the question is where we will get the value. A simple command called "dmesg" which displays the messages running inside a Kernel. So as we include the module and if any code hit the printk instance the dmesg command will display the value. The "KERN_INFO" part is the priority of the task its used in kernel.

### Supporting software and Packages

#### Editing tool:

We use "vim" editing tool for editing the code. For navigating the the source code we use a very beautiful tool "ctags" . Ctags allows us to navigate any function and structure in programs. Steps to embed ctags in code

1. Installing vim : sudo apt-get install vim

2. Installing ctags : sudo apt-get install ctags

Enter the following command next. First find the kernel version using "uname -r" command. For our Implementation we have linux-headers-2.6.32-46.

3. find /home/wifidit/compat-wireless-3.6.8-1 /usr/src/linux-headers-2.6.32-46-generic\( -name "*.h*" -o -name "*.c*" \) -follow > /home/wifidit/compat-wireless-3.6.8-1/allFiles.txt

4. ctags -L /home/wifidit/compat-wireless-3.6.8-1/allFiles.txt -f /home/wifidit/compat-wireless-3.6.8-1/tagFiles.txt -R

5. (Only for one time) Add this /etc/vim/vmrc

set tags=/home/wifidit/compat-wireless-3.6.8-1/tagFiles.txt

**Monitoring tool:**

Till now we are using **Wireshark** and **Iperf** as monitoring tool.

Wireshark shows the tx and rx packet details

Iperf can be used to know the link bandwidth,delay,and jitter

Uses of iperf

1. Need to install using apt-get install iperf

2. First need to make a server in a system " iperf -s"

3. On other system use "iperf -c 10.42.43.1"

10.42.43.1 is the ip of the server.

**Application:**

We use ekiga to do video calling to generate voip traffic.

## Atheros Drivers

Initially atheros provides open source driver madwifi for linux kernel. The newest version of atheros driver is Ath9k. Its has capability to run 802.11 n based hardware. We are using AR9285 chipset in our desktops which also a 802.11n based chip and uses Ath9k as driver. The linux Kernel version 2.6.32-38-generic that we are using loads the required modules on startup. The source code for ath9k can be found in http://wireless.kernel.org/en/users/Download/stable . compat-wireless-3.6.8-1.tar.bz2 is the main source file for all kernel version and also other wireless drivers. We need to unzip the source and need to run a shell script to get our required driver.

Inside the compat-wireless directory we have to select our required driver using “./**scripts/driver-selec**
**Ath9k**”. So if we compile the code using “make” command . We will be dealing with the mac80211 module first for changing access default 802.11. The directory which includes the mac80211 functionality is “/compat-wireless-3.6.9-1/net/mac80211”. If we compile the source a object file called mac80211.ko (Kernel Object or Kernel Module) will generated. Which can be include in the Kernel using “insmod” command. A command called “lsmod” will display all the modules that are currently running inside the kernel.

Some useful command to manipulate functionality

### iwconfig :

iwconfig is used to configure a wireless network interface. Many parameters can be set using this command. The following that we are using

**iwconfig wlan0 mode monitor** : Will set the wlan0 card in monitor mode

**iwconfig wlan0 channel 1** : Will set the card to operate in channel 1

Some other parameters are also available . You can see the man page for this

**man iwconfig**

### ifconfig:

ifconfig is used to configure any network interface. The following command we using

**ifconfig wlan0 down** : Will down the wlan0 card

**ifconfig wlan0 up** : Will up the wlan0 card

**ifconfig wlan0 10.42.43.1** : Will set the ip address for wlan0

Some other parameters are also available . You can see the man page for this

**man ifconfig**

## Inside Compat Wireless Source:

The main Directory for us is

**/compat-wireless-3.6.8-1/net/mac80211** : For MAC functionality

**/compat-wireless-3.6.8-1/drivers/net/wireless/ath/ath9k** : For Hardware specific functionality

The card operating modes are

1. Ad-Hoc

2. Managed

3. Master

4. Monitor

All other mode except monitor uses default CSMA/CA functionality.

In real case if any packet is coming to the network layer passes sk_buff and net_device to mac80211 with a ethernet header attach in it.

| DA | SA | Type | Data |
|---|---|---|---|

DA- Destination Address -The destination MAC ID of 6 bytes

SA- Source Address - The Source MAC ID of 6 bytes

Type- Type of packet in IP or ICMP 2 Byte

So a total of 14 byte is attached to it. Now what first three modes do, they encapsulate the first 12 byte and modify it to 802.11frame format.

| FC | DI | Add1 | Add2 | Add3 | SC | Add4 | Data |
|---|---|---|---|---|---|---|---|

FC - Frame Control of 2 Byte

DI - Duration ID 2 Byte

Add1 - is Always DA 6 Bytes

Other Address Field changes according to mode. All are of 6 Byte

SC- Sequence Control Field of 2 Byte

Note: The Hardware than adds 4 Byte Frame Check Sequence for Error Correction in Tail

All 3 modes uses RTS/CTS and Link Layer ACK's and the above frame format. So We can't use this mode.

So by setting the card in monitor mode we found

1. No RTS/CTS

2. No 802.11 Frame format

3. Transmission of Custom Frame Format.

Monitor mode allows us to send the packet without manipulating the packet. But still there are some problems with the code in monitor mode transmission and reception in mac80211. So Lets first follow the packet tx path in monitor mode.

## Tx Path Monitor Mode

1. The Kernel First calls **ieee80211_monitor_start_xmit(sk_buff,net_device) in tx.c** by passing a skb and dev. This function uses the structure and do some check and create a sdata from dev and calls the ieee80211_xmit().
2. **ieee80211_xmit(skb,sdata) in tx.c** it takes the skb and sdata as input and calls ieee80211_tx()
3. **ieee80211_tx(sdata,skb,txpending) in tx.c** it takes sdata ,skb and a bool flag and if all went right it will call __ieee80211_tx()
4. **__ieee80211_tx(local,skb,led_len,sta,txpending) in tx.c** if all went right it will calls ieee80211_tx_frags()
5. **ieee80211_tx_frags(local,vif,sta,skbs,txpending) in tx.c** calls drv_tx()
6. **drv_tx(local,skb) in driver-ops.h** is the entry point to driver.
7. **ath9k_tx() (main.c)** -Wakes up the hardware if its sleeping -Uses data struct ath_tx_control to track the transmission status
8. **ath_tx_start() (xmit.c)** receives a pointer to an struct ath_tx_control containing a pointer to the queue that contains the Frame.
9. **ath_tx_send_normal() (xmit.c)** -Adds the data to be transmitted in the txq ath_tx_txqaddbuf
10. **ath_tx_txqaddbuf() (xmit.c)**
11. **ath9k_hw_txstart() (mac.c)**

For our implmentation we will be dealing with first function i.e **ieee80211_monitor_start_xmit()** .The other function doesn't have any work to do with the packet except stamping the Duration ID and Sequence number in the hardware level. This is all about tx path. Now lets see the receive path

**Rx Path Monitor Mode**

1. When receiving a packet, and also for other reasons, the hardware sends an interruption that ath9k has previously registered. The function in charge of handling the interruption seems to be **irqreturn_tath_isrin (main.c)**. This function discovers the type of interruption and asks the kernel to schedule the execution of a tasklet, ath9k_tasklet in main.c. This function in turn calls the the receive tasklet **ath_rx_tasklet()**.
2. **ath_rx_tasklet(recv.c)**-Obtains the frame header-Obtains the current tsf value-Records info about received packet in struct ieee80211_rx_status- Insert received data in the receive buffer-Creates a new skb to contain the received data-Passes the skb to **ieee80211_rx()**
3. **ieee80211_rx() (rx.c)**-This is the entry point to mac80211
4. **__ieee80211_rx_handle_packet() in rx.c** Calls the ieee80211_rx_monitor() to remove FCS and Radiotap if any
5. **ieee80211_rx_monitor() in rx.c** Returns a cleand up skb to the called function or if the interface is in monitor mode it calls **netif_recive_skb()**
6. **netif_receive_skb()** - Entry Point to the Kernel

**Problems in Monitor mode**

1. One cannot communicate between two devices running in monitor mode
2. We cannot ping one machine from another running in monitor mode
3. The machines running in monitor mode were not able to resolve ARP request. The reason is that monitor mode functionality is coded in such a way that it attaches a Radio Tap header to all incoming packets, to be processed by sniffer software's which is not a valid packet for network Layer .

Note : Radio Tap header contains the information like the time at which driver received the packet, the channel on which the packet was received, the signal strength and the noise level etc.

## Completed Tasks

Firstly to make a prototype we use Dell Vostro 360 Desktop With Atheros AR9285 Chipset for Wireless.We are using Linux Ubuntu 10.04 with Kernel Version 2.6.32-48-generic. We extract the compat-wireless-3.6.8-1 in home directory. We include two shell scripts do compilation .

for installation we use install.sh which contains the following lines

```
rmmod ath9k
rmmod mac80211
make
insmod /compat-wireless-3.6.8-1/drivers/net/wireless/ath/ath9k/ath9k.ko
insmod /compat-wireless-3.6.8-1/net/mac80211/mac80211.ko
```

For Keeping the card in monitor mode and giving ip to the interface we use monitor.sh scripts. Which contains the following lines

```
configure wlan0 down
iwconfig wlan0 mode monitor
ifconfig wlan0 up
ifconfig wlan0 10.42.43.1
```

Note: For different machine we use different IP.

We create three extra header files to put our code so that we don't have to modify too much in main code in /net/mac80211

The included files are

2C-tx.h - For all TX and header related tasks

2C-rx.h - For all RX related tasks

2C-main.h - Timer related tasks

**Task 1 : Transmission in monitor mode**

Problem: As per as the code there is no provision for transmission in monitor mode. Some application like aireplay-ng can use monitor interface for tx. For our case we are not able to do transmission .

Solution: In Function **ieee80211_monitor_start_xmit() in tx.c** We found some condition which failing our transmission. So As per our data we remove some check and setting some required value. And allow to fulfill all condition and so that it can call **ieee80211_xmit().**

After doing so we found some unrecognized packet in other system running in monitor mode. We ping as our test application. By debugging the skb data value we found packet is generated and transmitted in air. And by running another system in monitor mode and using wireshark we collect the packet.

**Task 2 : Ping in Monitor mode**

**Problem 1** : Though Raw packet from network layer (for us ICMP packets) is transmitted in air without manipulating it but in reviving side a 26 byte radiotap header is attached to every packet to pass to capturing tools. This type of packet is unrecognized for network layer .

Solution : We edit the code in **rx.c file in mac80211** directory in function **ieee80211_rx_monitor().**

1. We create a header file call **2C-rx.h** in the same directory and include the required function to perform receive operation in monitor mode. First of all we passed the given skb to **recvData()** in **2C-rx.h** .

2. In recvData() the we remove the 26 byte radiotap header and 24 Byte TDMA Header and 2 byte protocol id from data and remove 4 byte CRC from the tail and return the clear skb to ieee80211_rx_monitor() and in turn it will pass it to **netif_receive_skb()** ;

3. We use some skb_pull() to remove header from start and skb_trim() to remove data from tail.

**Problem 2** : Though the packet is transferd to network layer but the layer is not replying for any ping packet. The reason is that the packet get corrupted while sending as it doesn't have any 802.11 header. The hardware accept all packet as it have 30 bytes of 802.11 header.

Solution: We made a header format for our implementation look like a 802.11 header Keeping reserve some fields that the hardware changes. Our TDMA data header format

| FC | RES1 | BRD | DA | SA | RES2 | Data |
|---|---|---|---|---|---|---|

FC- Frame Control of 2 Byte (Will Keep 0xFF value for data packet later we may change it for our requirement)

RES1- Reserve 2 byte for Duration ID

BRD- a 6 Byte Broadcast Address

DA- Destination Address of 6 Byte

SA- Source Address of 6 Byte

RES2- For Sequence Control

and Rest is Data Part.

The 4 byte CRC will be added to the tail by Hardware itself

We Encapsulate the Ethernet Header of 12 Byte having DA and SA and embed it to respective DA and SA Address in our TDMA header.

After this we able to ping in monitor mode.

Though atheros hardware doesn't send ack in monitor mode but we found link layer is generating ack for every packet. So to overcome the issue we put broadcast id in BRD field all are 0XFF. If we follow the 802.11 format the first address bye contains always destination ID and hardware don't send ack for broadcast packet. So if we put BRD in that address location the hardware thinks packet as broadcast and doesn't send any ack. We overcome the issue by this but. still we are trying to find other way so that we can reduce our header format to reduce overhead.

The reserved field are also like that only . The hardware manipulate this data so we leave reserve and so we don't loose any corrupt packet at receiver end.

We use skb_push() to insert header in packet.

**Task 3: Queuing in MAC layer**

For queuing we maintain a software queue using linked list data structure. The packet from the network layer is handed to **ieee80211_monitor_start_xmit()** passing skb and network device information . We queue the both information in **tdma_queue**. The a function called sendData() will extract from the queue and passes it to **ieee80211_xmit()** which transmit the packet in air. We maintain a timer which will call the **sendData()** function periodically.

**Task 4 : Linux Timer**

In the Linux kernel, time is measured by a global variable named jiffies. The kernel 250 MHZ clock to increment the jiffies value. But we found kernel provide a tick in 4ms granularity which is not suitable for time sensitive applications. So we used another timer called High Resolution Timer to implement our tdma system. We made a TDMA timer which will periodically calls **sendData()**. In High Resolution timer we can give a granularity of nano seconds.

**Task 5 : Generating Control Packet in MAC layer**

For this we first keep the instance of skb and sdata as the interface start in a global variable called **globalskb** and **globalsdata**. Then when the timer calls **sendCtrl()** function will add modify the globale variable and will put our control header inside it and transmit in air calling **ieee80211_xmit()**.