# 配置线(好像只下发了配置没有修改硬件啊啊啊)

### kochab

int

dl\_wfradio\_set\_chanbw (struct db\_table\_hdr \*tbh, int m,

int chanbw)

{

dl\_iwpriv\_set(tb[m].name, "bhuChanBw", (char \*)&val, 1, sizeof(int))

}

int

dl\_iwpriv\_set(const char \*ifname, const char \*cmdname, const char \*args, int count, int sz)

{

skfd = if\_enter() //创建socket

dl\_iwpirv\_cmd(skfd, ifname, cmdname, &priv, &cmd, &subcmd)//获取正真的ioctl命令

ioctl(skfd, cmd->cmd, &wrq)//下发命令

}

## 获取ioctl命令列表 dl\_iwpirv\_cmd

### kochab

int

dl\_iwpirv\_cmd(int skfd, const char \*ifname, const char \*cmdname,

iwprivargs \*\*privp, iwprivargs \*\*cmd, iwprivargs \*\*subcmd)

{

num = iw\_get\_priv\_info(skfd, ifname, &priv);//获取该接口（ifname）支持的ioctl命令列表。

。。。。

匹配出真正的ioctl命令

}

int

iw\_get\_priv\_info(int skfd,

const char \* ifname,

iwprivargs \*\* ppriv)

{

Do

{

//每次读取16个命令。

iw\_get\_ext(skfd, ifname, SIOCGIWPRIV, &wrq)

}

}

static inline int

iw\_get\_ext(int skfd, /\* Socket to the kernel \*/

const char \* ifname, /\* Device name \*/

int request, /\* WE ID \*/

struct iwreq \* pwrq) /\* Fixed part of the request \*/

{

ioctl(skfd, request, pwrq) //下内核 (SIOCGIWPRIV)

}

### 内核：

static const struct file\_operations socket\_file\_ops = {

.unlocked\_ioctl = sock\_ioctl,

.compat\_ioctl = compat\_sock\_ioctl,

};

sock\_ioctl/ compat\_sock\_ioctl

---🡪 **dev\_ioctl**(){

if (cmd >= SIOCIWFIRST && cmd <= SIOCIWLAST) // SIOCIWFIRST:0x8B00 SIOCIWLAST:0x8BFF

return wext\_handle\_ioctl(net, &ifr, cmd, arg);

}

---🡪 wext\_handle\_ioctl(){}

---🡪 **wext\_ioctl\_dispatch**(){}

---🡪 wireless\_process\_ioctl(){

dev = \_\_dev\_get\_by\_name(net, ifr->ifr\_name))//获取接口对应的dev

if (cmd == SIOCGIWSTATS)

return standard(dev, iwr, cmd, info,

&iw\_handler\_get\_iwstats);

/\*\*\* **[dev->wireless\_handlers]** **这个wifi和vap使用不同的attach** \*\*\*/

if (cmd == SIOCGIWPRIV && dev->**wireless\_handlers**) //获取ioctl命令列表

return standard(dev, iwr, cmd, info,

&iw\_handler\_get\_private);

/\* New driver API : try to find the handler \*/

handler = get\_handler(dev, cmd); //获取iw ioctl命令的处理函数

if (handler) {

/\* Standard and private are not the same \*/

if (cmd < SIOCIWFIRSTPRIV) // SIOCIWFIRSTPRIV:0x8BE0

return standard(dev, iwr, cmd, info, handler);

else

return private(dev, iwr, cmd, info, handler);

}

/\* Old driver API : call driver ioctl handler \*/

if (dev->netdev\_ops->ndo\_do\_ioctl)

return dev->netdev\_ops->ndo\_do\_ioctl(dev, ifr, cmd);

}

----🡪 iw\_handler\_get\_private(){

memcpy(extra, dev->wireless\_handlers->private\_args,

sizeof(struct iw\_priv\_args) \* wrqu->data.length);

}

### 驱动Ath\_wlan

dev->wireless\_handlers是哪来的呢？

\_\_ath\_attach //

---🡪 ath\_iw\_attach(dev){

dev->wireless\_handlers = &ath\_iw\_handler\_def;

}

#ifdef ATH\_SUPPORT\_HTC

#else

static struct iw\_handler\_def ath\_iw\_handler\_def = {

.standard = (iw\_handler \*) NULL,

.num\_standard = 0,

.private = (iw\_handler \*) ath\_iw\_priv\_handlers,

.num\_private = TABLE\_SIZE(ath\_iw\_priv\_handlers),

.private\_args = (struct iw\_priv\_args \*) ath\_iw\_priv\_args,

.num\_private\_args = TABLE\_SIZE(ath\_iw\_priv\_args),

.get\_wireless\_stats = NULL,

};

#endif

ath\_iw\_priv\_args 即为ioctl参数列表。

## 下发真正的命令

### kochab ioctl(skfd, cmd->cmd, &wrq)

### 内核

同样先到---🡪 **wireless\_process\_ioctl**(){

dev = \_\_dev\_get\_by\_name(net, ifr->ifr\_name))//获取接口对应的dev

if (cmd == SIOCGIWSTATS)

return standard(dev, iwr, cmd, info,

&iw\_handler\_get\_iwstats);

if (cmd == SIOCGIWPRIV && dev->wireless\_handlers) //获取ioctl命令列表

return standard(dev, iwr, cmd, info,

&iw\_handler\_get\_private);

/\* New driver API : try to find the handler \*/

handler = get\_handler(dev, cmd); //获取iw ioctl命令的处理函数

if (handler) {

/\* Standard and private are not the same \*/

if (cmd < SIOCIWFIRSTPRIV) // SIOCIWFIRSTPRIV:0x8BE0

return standard(dev, iwr, cmd, info, handler);

else

return private(dev, iwr, cmd, info, handler);

}

/\* Old driver API : call driver ioctl handler \*/

if (dev->netdev\_ops->ndo\_do\_ioctl)

return dev->netdev\_ops->ndo\_do\_ioctl(dev, ifr, cmd);

}

----🡪 get\_handler (){

/\* Try as a private command \*/

index = cmd - SIOCIWFIRSTPRIV;

if (index < dev->wireless\_handlers->num\_private)

return dev->wireless\_handlers->private[index];

}

----🡪（private）//int ioctl\_private\_call()//见驱动

### 驱动Ath\_wlan

还是这个结构体

#ifdef ATH\_SUPPORT\_HTC

#else

static struct iw\_handler\_def ath\_iw\_handler\_def = {

.standard = (iw\_handler \*) NULL,

.num\_standard = 0,

.private = (iw\_handler \*) **ath\_iw\_priv\_handlers**,

.num\_private = TABLE\_SIZE(ath\_iw\_priv\_handlers),

.private\_args = (struct iw\_priv\_args \*) ath\_iw\_priv\_args,

.num\_private\_args = TABLE\_SIZE(ath\_iw\_priv\_args),

.get\_wireless\_stats = NULL,

};

#endif

static const iw\_handler **ath\_iw\_priv\_handlers**[] = {

(iw\_handler) ath\_iw\_setparam, /\* SIOCWFIRSTPRIV+0 \*/

(iw\_handler) ath\_iw\_getparam, /\* SIOCWFIRSTPRIV+1 \*/

(iw\_handler) ath\_iw\_setcountry, /\* SIOCWFIRSTPRIV+2 \*/

(iw\_handler) ath\_iw\_getcountry, /\* SIOCWFIRSTPRIV+3 \*/

(iw\_handler) ath\_iw\_sethwaddr, /\* SIOCWFIRSTPRIV+4 \*/

(iw\_handler) ath\_iw\_gethwaddr, /\* SIOCWFIRSTPRIV+5 \*/

/\* begin: liurq@BHU, 2013-3-4 \*/

(iw\_handler) ath\_iw\_setparam, /\* SIOCWFIRSTPRIV+6 \*/

(iw\_handler) ath\_iw\_getparam, /\* SIOCWFIRSTPRIV+7 \*/

/\* end: liurq@BHU, 2013-3-4 \*/

};

struct iwreq

{

union

{

char ifrn\_name[IFNAMSIZ]; /\* if name, e.g. "eth0" \*/

} ifr\_ifrn;

/\* Data part (defined just above) \*/

union iwreq\_data u;

};

union iwreq\_data

{

/\* Config - generic \*/

char name[IFNAMSIZ];

/\* Name : used to verify the presence of wireless extensions.

\* Name of the protocol/provider... \*/

struct iw\_point essid; /\* Extended network name \*/

struct iw\_param nwid; /\* network id (or domain - the cell)

struct iw\_freq freq; /\* frequency or channel :

\* 0-1000 = channel

\* > 1000 = frequency in Hz \*/

struct iw\_param sens; /\* signal level threshold \*/

struct iw\_param bitrate; /\* default bit rate \*/

struct iw\_param txpower; /\* default transmit power \*/

struct iw\_param rts; /\* RTS threshold threshold \*/

struct iw\_param frag; /\* Fragmentation threshold \*/

\_\_u32 mode; /\* Operation mode \*/

struct iw\_param retry; /\* Retry limits & lifetime \*/

struct iw\_point encoding; /\* Encoding stuff : tokens \*/

struct iw\_param power; /\* PM duration/timeout \*/

struct iw\_quality qual; /\* Quality part of statistics \*/

struct sockaddr ap\_addr; /\* Access point address \*/

struct sockaddr addr; /\* Destination address (hw/mac) \*/

struct iw\_param param; /\* Other small parameters \*/

struct iw\_point data; /\* Other large parameters \*/

};

()

static int ioctl\_private\_call(struct net\_device \*dev, struct iwreq \*iwr,

unsigned int cmd, struct iw\_request\_info \*info,

iw\_handler handler)

{

int extra\_size = 0, ret = -EINVAL;

const struct iw\_priv\_args \*descr;

extra\_size = get\_priv\_descr\_and\_size(dev, cmd, &descr);

/\* Check if we have a pointer to user space data or not. \*/

if (extra\_size == 0) {

/\* No extra arguments. Trivial to handle \*/

ret = handler(dev, info, &(iwr->u), (char \*) &(iwr->u));

} else {

ret = ioctl\_private\_iw\_point(&iwr->u.data, cmd, descr,

handler, dev, info, extra\_size);

}

/\* Call commit handler if needed and defined \*/

if (ret == -EIWCOMMIT)

ret = call\_commit\_handler(dev);

return ret;

}

int

dl\_iwpriv\_set(const char \*ifname, const char \*cmdname, const char \*args, int count, int sz)

{

…

u8 buff[1024];

if(offset) {

((u32 \*) buff)[0] = subcmd->cmd;

}

memcpy(buff+offset, args, sz\*count<1024 ? sz\*count : 1024);

wrq.u.data.pointer = (caddr\_t) buff;

wrq.u.data.flags = cmd->cmd;

…

}

(handler(dev, info, &(iwr->u), (char \*) &(iwr->u));)

static int **ath\_iw\_setparam**(struct net\_device \*dev,

struct iw\_request\_info \*info,

void \*w,

char \*extra)

{

int \*i = (int \*) extra;

if (param & ATH\_PARAM\_SHIFT){

param -= ATH\_PARAM\_SHIFT;

retval = scn->sc\_ops->ath\_set\_config\_param(scn->sc\_dev,

(ath\_param\_ID\_t)param,

&(i[1]));//&value); //liurq@BHU, 2013-3-4

}

}

{ **ATH\_PARAM\_TXPOWER\_LIMIT2G** | ATH\_PARAM\_SHIFT,

IW\_PRIV\_TYPE\_INT | IW\_PRIV\_SIZE\_FIXED | 1, 0, "TXPowLim2G" },

static const struct ath\_ops ath\_ar\_ops = {

…

ath\_set\_config // sc\_ops->ath\_set\_config\_param

…

}

int

ath\_set\_config(ath\_dev\_t dev, ath\_param\_ID\_t ID, void \*buff)

{

case **ATH\_PARAM\_TXPOWER\_LIMIT2G**:

if (\*(int \*)buff > ATH\_TXPOWER\_MAX\_2G) {

retval = -1;

} else {

**sc->sc\_config.txpowlimit2G = \*(int \*)buff;**

}

}

struct ifreq \*ifr, 🡸🡺 struct iwreq \*iwr

struct iw\_point

{

**void \_\_user \*pointer; /\* Pointer to the data (in user space) \*/**

\_\_u16 length; /\* number of fields or size in bytes \*/

\_\_u16 flags; /\* Optional params \*/

};

# 创建VAP

## Kochab

char \*

dl\_wfvap\_create\_vap (char \*radio, const char \*name, char mode)

{

struct ieee80211\_clone\_params cp;

struct ifreq ifr;

…

switch(mode) {

case 1:/\* **AP** \*/

cp.icp\_opmode = IEEE80211\_M\_HOSTAP;

cp.icp\_flags = IEEE80211\_CLONE\_BSSID;

break;

case 2:/\* **STA** \*/

cp.icp\_opmode = IEEE80211\_M\_STA;

cp.icp\_flags = IEEE80211\_CLONE\_BSSID;

break;

…

strncpy(ifr.ifr\_name, radio, IFNAMSIZ);

strncpy(cp.icp\_name, name, IFNAMSIZ);

ifr.**ifr\_data** = (void \*) &cp;

ioctl(skfd, SIOC80211IFCREATE, &ifr)

}

## 内核

先到---🡪 **wireless\_process\_ioctl**()

{

struct net\_device \*dev;

dev = \_\_dev\_get\_by\_name(net, ifr->**ifr\_name**)) /\* namePCI设备名： wifi0 \*/

if (dev->netdev\_ops->ndo\_do\_ioctl)

return dev->netdev\_ops->ndo\_do\_ioctl(dev, ifr, cmd);

}

static const struct net\_device\_ops athdev\_net\_ops = {

.ndo\_open = ath\_netdev\_open,

.ndo\_stop = ath\_netdev\_stop,

.ndo\_start\_xmit = ath\_netdev\_hardstart,

.ndo\_set\_mac\_address = ath\_netdev\_set\_macaddr,

.ndo\_tx\_timeout = ath\_netdev\_tx\_timeout,

.ndo\_get\_stats = ath\_getstats,

.ndo\_change\_mtu = ath\_change\_mtu,

.ndo\_set\_multicast\_list = ath\_netdev\_set\_mcast\_list,

**.ndo\_do\_ioctl = ath\_ioctl,**

};

## 驱动

static int

ath\_ioctl(struct net\_device \*dev, struct ifreq \*ifr, int cmd)

{ switch (cmd){

。。。

case SIOC80211IFCREATE:

error = osif\_ioctl\_create\_vap(dev, ifr, scn->sc\_osdev);

。。。

}

}

---🡪

int

osif\_ioctl\_create\_vap(struct net\_device \*comdev, struct ifreq \*ifr, osdev\_t os\_handle)

{

struct ieee80211\_clone\_params cp;

wlan\_dev\_t devhandle = ath\_netdev\_priv(comdev);

ifc\_name2unit(cp.icp\_name, &unit); /\* wlan1 🡺 1 \*/

**\_copy\_from\_user(&cp, ifr->ifr\_data, sizeof(cp))**

strncpy(name, cp.icp\_name, sizeof(name));

// Allocate net device for this network interface （for vap like : **wlan0**）

dev = alloc\_netdev(sizeof(osif\_dev), name, ether\_setup);

if (cp.icp\_opmode == IEEE80211\_M\_HOSTAP) {

scan\_priority\_mapping\_base = DEF\_VAP\_SCAN\_PRI\_MAP\_OPMODE\_AP\_BASE;

}

vap = **wlan\_vap\_create**(**devhandle**, cp.icp\_opmode, scan\_priority\_mapping\_base, cp.icp\_flags, cp.icp\_bssid);

**osifp->****os\_if = vap;**

osifp->os\_handle = os\_handle;

osifp->os\_devhandle = devhandle;

osifp->os\_comdev = comdev;

osifp->os\_opmode = cp.icp\_opmode;

osifp->os\_unit = unit;

osif\_vap\_setup(vap, dev, cp.icp\_opmode); /\* ? 各种event注册，状态机初始化 ? \*/

#if LINUX\_VERSION\_CODE > KERNEL\_VERSION(2,6,30)

dev->netdev\_ops = &osif\_dev\_ops;

register\_netdevice(dev);

}

------🡪

wlan\_if\_t

**wlan\_vap\_create**(wlan\_dev\_t **devhandle**,

enum ieee80211\_opmode opmode,

int scan\_priority\_base,

int flags,

u\_int8\_t \*bssid)

{

**struct ieee80211vap \*vap;**

struct ieee80211com \*ic = devhandle;

vap = ic->ic\_vap\_create(ic, opmode, scan\_priority\_base, flags, bssid);

//ath\_vap\_create

TAILQ\_INSERT\_TAIL(&ic->ic\_vaps, vap, iv\_next);

}

---🡪

static struct ieee80211vap \*

ath\_vap\_create(struct ieee80211com \*ic,

int opmode,

int scan\_priority\_base,

int flags,

const u\_int8\_t bssid[IEEE80211\_ADDR\_LEN])

{

struct ath\_vap\_net80211 \*avn;

ic\_opmode = IEEE80211\_M\_HOSTAP;

avn = (struct ath\_vap\_net80211 \*)OS\_ALLOC\_VAP(scn->sc\_osdev,

sizeof(struct ath\_vap\_net80211));

avn->av\_sc = scn;

vap = &avn->av\_vap;

scn->**sc\_ops**->add\_interface(scn->sc\_dev, id, vap, ic\_opmode, ath\_opmode, nostabeacons) // ath\_vap\_attach

//配置VAP，

ieee80211\_vap\_setup(ic, vap, opmode, scan\_priority\_base, flags, bssid);

}

static int

ath\_vap\_attach(ath\_dev\_t dev, int if\_id, ieee80211\_if\_t if\_data, HAL\_OPMODE opmode, HAL\_OPMODE iv\_opmode, int nostabeacons)

{

struct ath\_softc \*sc;

avp = (struct ath\_vap \*)OS\_MALLOC(sc->sc\_osdev, sizeof(struct ath\_vap), GFP\_KERNEL);

avp->av\_if\_data = if\_data;

sc->sc\_vaps[if\_id] = avp;

}

int

ieee80211\_vap\_setup(struct ieee80211com \*ic, struct ieee80211vap \*vap,

int opmode, int scan\_priority\_base, int flags,

const u\_int8\_t bssid[IEEE80211\_ADDR\_LEN])

{

vap->iv\_ic = ic;

**vap->iv\_opmode = opmode;**

}

# 启动VAP（BSS）

## Kochab

const struct db\_table g\_dbtb\_if = {

.name = "interface",

. . .

.load = dl\_if\_load,

. . .

};

dl\_if\_enable(tbh, m, tc\_str2enable(s), 0)

{

if\_modify\_flags(iftb->name, IFF\_UP | IFF\_RUNNING, enable);

}

static int

if\_modify\_flags (char \*if\_name, short int flags, int fill)

{

ioctl(if\_ctl\_fd, **SIOCSIFFLAGS**, &ifr);

}

## 内核

-----🡪ioctl()

->sock\_ioctl()

->dev\_ioctl(){

->SIOCSIFFLAGS

}

->dev\_ifsioc()->dev\_change\_flags(){

((old\_flags & IFF\_UP) ? dev\_close : dev\_open)(dev); /\* 哎呀还可以这样写 \*/

}

->dev\_open()

{

const struct net\_device\_ops \*ops = dev->netdev\_ops;

ops->ndo\_open(dev);

}

// osif\_ioctl\_create\_vap ()🡪dev->netdev\_ops = &osif\_dev\_ops;

static const struct net\_device\_ops **osif\_dev\_ops** = {

.ndo\_open = **osif\_vap\_open**,

};

## 驱动

int

osif\_vap\_open(struct net\_device \*dev)

{

return osif\_vap\_init(dev, 0);

}

int

osif\_vap\_init(struct net\_device \*dev, int forcescan)

{

struct net\_device \*parent = osifp->**os\_comdev**;

if (osif\_get\_num\_active\_vaps(comhandle) == 0 &&

(parent->flags & IFF\_RUNNING) == 0){

return\_val=**dev\_open(parent)**;

}

dev->flags |= IFF\_RUNNING; /\* mark us running \*/

opmode = wlan\_vap\_get\_opmode(vap);

if (opmode == BSS){

//Stop the bss if it is already running

wlan\_mlme\_stop\_bss(vap, 0);

//Delay for BSS to stop

OS\_DELAY(1000);

chan = wlan\_get\_current\_channel(vap, false);

//kochab: dl\_wfradio\_set\_channel应该会先设置下

//先假设没有HT20/HT40这个好像是传说中的扩展信道便宜。

**wlan\_mlme\_start\_bss(vap);**

//Initiate BSS start processing in UMAC

wlan\_mlme\_connection\_up(vap);

}

osifp->is\_up = 1;

}

int wlan\_mlme\_start\_bss(wlan\_if\_t vaphandle)

{

struct ieee80211vap \*vap = vaphandle;

case IEEE80211\_M\_HOSTAP:

mlme\_create\_infra\_bss(vap);

}

int

mlme\_create\_infra\_bss(struct ieee80211vap \*vap)

{

/\* create BSS node for infra network \*/

ieee80211\_create\_infra\_bss(vap,ssid->ssid, ssid->len);

/? ? ?/

ieee80211\_resmgr\_vap\_start(ic->ic\_resmgr,vap,chan,MLME\_REQ\_ID,0);

/? ? ?/

ieee80211\_vap\_join(vap);

ieee80211\_mlme\_create\_infra\_continue(vap);

}

int

ieee80211\_create\_infra\_bss(struct ieee80211vap \*vap,

const u\_int8\_t \*essid,

const u\_int16\_t esslen)

{

struct ieee80211com \*ic = vap->iv\_ic;

**struct ieee80211\_node \*ni;**

ni = ieee80211\_alloc\_node(&ic->ic\_sta, vap, vap->iv\_myaddr);

ieee80211\_copy\_bss(ni, vap->iv\_bss);

return ieee80211\_sta\_join\_bss(ni);

}

# VAP 状态机

# 数据流（Tx）

static int

ath\_netdev\_hardstart(struct sk\_buff \*skb, struct net\_device \*dev)

{

do\_ath\_netdev\_hardstart(skb,dev);

}

----🡪

int

ath\_netdev\_hardstart\_aponly(struct sk\_buff \*skb, struct net\_device \*dev)

{

ath\_tx\_send\_aponly(skb);

}.

static inline int

ath\_tx\_send\_aponly(wbuf\_t wbuf)

{

struct ieee80211\_node \*ni = wbuf\_get\_node(wbuf);

struct ieee80211com \*ic = ni->ni\_ic;

struct ath\_softc\_net80211 \*scn = ATH\_SOFTC\_NET80211(ic);

wbuf\_t next\_wbuf;

**//.3🡪.11**

wbuf = ieee80211\_encap\_8023\_aponly(ni, wbuf);

ieee80211\_tx\_control\_t ltxctl; \

ieee80211\_tx\_control\_t \* **txctl** = &ltxctl

//准备该报文txctl相关的信息

ath\_tx\_prepare\_aponly(scn, wbuf, nextfraglen, **txctl**);

ath\_tx\_start\_aponly(scn->sc\_dev, wbuf, txctl)

}

static inline int

ath\_tx\_start\_aponly(ath\_dev\_t dev, wbuf\_t wbuf, ieee80211\_tx\_control\_t \*txctl)

{

\_\_ath\_tx\_prepare\_aponly(sc, wbuf, txctl);//?不知道干了啥

// Start DMA mapping

error = \_\_wbuf\_map\_sg\_aponly(sc->sc\_osdev, **wbuf**,

OS\_GET\_DMA\_MEM\_CONTEXT(txctl, dmacontext),

txctl);

}

static inline int

\_\_wbuf\_map\_sg\_aponly(osdev\_t osdev, adf\_nbuf\_t **nbf**, dma\_addr\_t \*pa, void \*arg)

{

struct scatterlist sg;

\*pa = bus\_map\_single(osdev, nbf->data, UNI\_SKB\_END\_POINTER(nbf) - nbf->data, **BUS\_DMA\_TODEVICE**);

//吧放到一个scatterlist里

sg\_dma\_address(&sg) // (sg)->dma\_address

sg\_dma\_address(&sg) = \*pa; // (sg)->length = \*pa

ath\_tx\_start\_dma\_aponly(nbf, &sg, 1, arg);

}

static inline int

**ath\_tx\_start\_dma\_aponly**(wbuf\_t wbuf, sg\_t \*sg, u\_int32\_t n\_sg, void \*arg)

{

struct ath\_txq \*txq = &sc->sc\_txq[txctl->qnum];

struct ath\_node \*an = txctl->an;

/\* For each sglist entry, allocate an ath\_buf for DMA \*/

for (i = 0; i < n\_sg; i++, sg++) {

/\* 从bf上摘一个节点放到bf\_head上 \*/

retval = ath\_tx\_get\_buf\_aponly(sc, sg, &bf, &bf\_head, buf\_used);

}

if (likely(txctl->**ht** && sc->**sc\_txaggr**))

{

if (likely(ath\_aggr\_query(**tid**))) {

/\*

\* Try aggregation if it's a unicast data frame

\* and the destination is HT capable.

\*/

ath\_tx\_send\_ampdu(sc, txq, tid, &bf\_head, txctl);

} else {

/\*

\* Send this frame as regular when ADDBA exchange

\* is neither complete nor pending.

\*/

ath\_tx\_send\_normal(sc, txq, tid, &bf\_head, txctl);

}

}

int

ath\_tx\_send\_normal(struct ath\_softc \***sc**, struct ath\_txq \***txq**, struct ath\_atx\_tid \***tid**, ath\_bufhead \***bf\_head**, ieee80211\_tx\_control\_t \***txctl**)

{

if (!**TAILQ\_EMPTY**(&tid->**buf\_q**) || tid->paused || txq->**axq\_depth**)

{

//Add this frame to software queue for scheduling later.

\_\_11nstats(sc, tx\_queue);

TAILQ\_CONCAT(&tid->buf\_q, bf\_head, bf\_list);

//各种链表挂接

ath\_tx\_queue\_tid(txq, tid);

if (!txq->axq\_depth && !tid->paused)

{

ath\_txq\_schedule(sc, txq);

}

/\* Queue to h/w directly \*/

return 0;

}

/? ? ?/

bf->bf\_nframes = 1;

bf->bf\_lastbf = bf->bf\_lastfrm; /\* one single frame \*/

ath\_buf\_set\_rate(sc, bf);

return ath\_tx\_txqaddbuf(sc, txq, bf\_head);

}

ath\_tx\_queue\_tid(struct ath\_txq \*txq, struct ath\_atx\_tid \*tid)

{

tid->sched = AH\_TRUE;

TAILQ\_INSERT\_TAIL(&ac->tid\_q, tid, tid\_qelem);

ac->sched = AH\_TRUE;

TAILQ\_INSERT\_TAIL(&txq->axq\_acq, ac, ac\_qelem);

}

void

**ath\_txq\_schedule**(struct ath\_softc \*sc, struct ath\_txq \*txq)

{

struct ath\_atx\_ac \*ac;

struct ath\_atx\_tid \*tid = NULL;

TAILQ\_HEAD(,ath\_atx\_tid)paused\_tid\_q;

do{

ac = TAILQ\_FIRST(&txq->axq\_acq);

TAILQ\_REMOVE(&txq->axq\_acq, ac, ac\_qelem);

tid = TAILQ\_FIRST(&ac->tid\_q);

TAILQ\_REMOVE(&ac->tid\_q, tid, tid\_qelem);

if (tid->paused)

{ /\* check next tid to keep h/w busy \*/

TAILQ\_INSERT\_TAIL(&paused\_tid\_q, tid, tid\_qelem);

tid->sched = AH\_TRUE;

continue;

}

ath\_tx\_sched\_normal(sc, txq, tid);

// \* add tid to round-robin queue if more frames are pending for the tid

}

}

// process pending HT single or legacy frames

INLINE void

ath\_tx\_sched\_normal(struct ath\_softc \*sc, struct ath\_txq \*txq, ath\_atx\_tid\_t \*tid)

{

struct ath\_buf \*bf, \*tbf, \*bf\_last;

do{

if ((sc->sc\_enhanceddmasupport) && (txq->axq\_depth >= HAL\_TXFIFO\_DEPTH)) {

/\* Reached the MAX FIFO DEPTH - do not add any more buffer to the HW \*/

break;

}

ath\_bufhead **bf\_q**;

**bf** = TAILQ\_FIRST(&tid->buf\_q);

ath\_hal\_setdesclink(sc->sc\_ah, bf->bf\_lastfrm->bf\_desc, 0);

/\*

\* this packet will be send as a single

\* - remove all descriptors belonging to this frame from software queue

\* 从TID上摘除【bf 到 bf->lastfrm（last buf of this frame）】的bf挂到bf\_q,

\*/

TAILQ\_REMOVE\_HEAD\_UNTIL(&tid->buf\_q, &**bf\_q**, bf->bf\_lastfrm, bf\_list);

ath\_tx\_txqaddbuf(sc, txq, &**bf\_q**)

}

}

int

ath\_tx\_txqaddbuf(struct ath\_softc \*sc, struct ath\_txq \*txq, ath\_bufhead \*head)

{

//内存初始化

TAILQ\_FOREACH(tbf, head, bf\_list) {

OS\_SYNC\_SINGLE(sc->sc\_osdev, tbf->**bf\_daddr**,

sc->sc\_txdesclen, **BUS\_DMA\_TODEVICE**, NULL);

}

//吧一帧的bf挂到txq的axq\_fifo[x]上

ATH\_EDMA\_TXQ\_CONCAT(txq, head);

/\* Write the tx descriptor address into the hw fifo \*/

//把bf->bf\_daddr 写到txq寄存器

//把准备发送的帧写入硬件队列的寄存器，硬件QCU则读出//内存🡪DMA🡪QCU

ath\_hal\_puttxbuf(ah, txq->axq\_qnum, bf->bf\_daddr); // ar9300SetTxDP

ath\_hal\_txstart(ah, txq->axq\_qnum);

}

static ath\_get\_buf\_status\_t

ath\_tx\_get\_buf\_aponly(struct ath\_softc \*sc, sg\_t \*sg, struct ath\_buf \*\*pbf,

ath\_bufhead \*bf\_head, u\_int32\_t \*buf\_used)

{

**bf = TAILQ\_FIRST(&sc->sc\_txbuf);**

\*pbf = bf;

**TAILQ\_REMOVE(&sc->sc\_txbuf, bf, bf\_list);**

sc->sc\_txbuf\_free--;

(\*buf\_used)++;

**TAILQ\_INSERT\_TAIL(bf\_head, bf, bf\_list);**

**bf->bf\_buf\_addr[sc->sc\_num\_txmaps - bf->bf\_avail\_buf] = sg\_dma\_address(sg);**

**bf->bf\_buf\_len[sc->sc\_num\_txmaps - bf->bf\_avail\_buf] = sg\_dma\_len(sg);**

}

# RX流

bf[512]

|

|[10]

|

V

PHY—>FIFO->DMA->rxfifo[10]

|

|

V

后半部

|

|

V

内核（netif\_rx(skb);）

# Tasklet 中断后半部(RX)

static void

ath\_tasklet(TQUEUE\_ARG data)

{

struct net\_device \*dev = (struct net\_device \*)data;

struct ath\_softc\_net80211 \*scn = ath\_netdev\_priv(dev);

do\_ath\_handle\_intr(scn->sc\_dev);

}

#define do\_ath\_handle\_intr(\_dev) do{ \

**ath\_handle\_intr\_aponly**(\_dev);\

}while(0)

void

ath\_handle\_intr\_aponly(ath\_dev\_t dev)

{

if (status & (HAL\_INT\_RXHP | HAL\_INT\_RXEOL | HAL\_INT\_RXORN)) {

**ath\_rx\_handler\_aponly**(dev, 0, **HAL\_RX\_QUEUE\_HP**);

}

if (status & HAL\_INT\_RXLP) {

**ath\_rx\_handler\_aponly**(dev, 0, **HAL\_RX\_QUEUE\_LP**);

}

}

static int

ath\_rx\_handler\_aponly(ath\_dev\_t dev, int flush, HAL\_RX\_QUEUE qtype)

{

struct ath\_softc \*sc = ATH\_DEV\_TO\_SC(dev);

struct ath\_rx\_edma \*rxedma;

wbuf\_t wbuf = NULL;

wbuf = bf->bf\_mpdu;

struct ath\_buf \*bf;

do

{

**bf = TAILQ\_FIRST(&rxedma->rxqueue);**

ds = (void \*)wbuf\_raw\_data(wbuf);

/\* point to the beginning of actual frame \*/

bf->bf\_vdata = (void \*)((u\_int8\_t \*)ds + sc->sc\_rxstatuslen);

// Init wbuf with recv data length

wbuf\_init(wbuf, (rxs->rs\_datalen + sc->sc\_rxstatuslen));

/\* 这个好像很高端的样子 \*/

ath\_rx\_bf\_handler(dev, wbuf, rxs, bf);

ath\_rx\_process\_aponly(sc, bf, rxs, wh->i\_fc[0], &rx\_status, &chainreset);

}while();

}

static inline void

ath\_rx\_process\_aponly(struct ath\_softc \*sc, struct ath\_buf \*bf, struct ath\_rx\_status \*rxs, u\_int8\_t frame\_fc0,

ieee80211\_rx\_status\_t \*rx\_status, u\_int8\_t \*chainreset)

{

type = ath\_rx\_indicate\_aponly(sc, wbuf, rx\_status, rxs->rs\_keyix);

}

static inline int ath\_rx\_indicate\_aponly(struct ath\_softc \*sc, wbuf\_t wbuf, ieee80211\_rx\_status\_t \*status, u\_int16\_t keyix)

{

struct ath\_buf \*bf = ATH\_GET\_RX\_CONTEXT\_BUF(wbuf);

wbuf\_t nwbuf;

/\* allocate a new wbuf and queue it to for H/W processing \*/

nwbuf = ath\_rxbuf\_alloc(sc, sc->sc\_rxbufsize);

type = ath\_net80211\_rx\_aponly(sc->**sc\_ieee**, wbuf, status, keyix);// sc->**sc\_ieee**??

}

ath\_net80211\_rx\_aponly(ieee80211\_handle\_t ieee, wbuf\_t wbuf, ieee80211\_rx\_status\_t \*rx\_status, u\_int16\_t keyix)

{

struct ieee80211com \*ic = NET80211\_HANDLE(ieee);

struct ath\_softc\_net80211 \*scn = ATH\_SOFTC\_NET80211(ic);

struct ieee80211\_node \*ni;

struct ieee80211\_frame \*wh;

ni = ieee80211\_find\_rxnode(ic, (struct ieee80211\_frame\_min \*)

wbuf\_header(wbuf));

if (ni == NULL) {

struct **ieee80211\_rx\_status** rs;

return ieee80211\_input\_all(ic, wbuf, &rs);

}

}

ieee80211\_input\_all(struct ieee80211com \*ic,

wbuf\_t wbuf, struct ieee80211\_rx\_status \*rs)

{

struct ieee80211\_iter\_input\_all\_arg params;

u\_int32\_t num\_vaps;

params.wbuf = wbuf;

params.rs = rs;

params.type = -1;

//把数据发给所有的VAP

ieee80211\_iterate\_vap\_list\_internal(ic,**ieee80211\_iter\_input\_all**,(void \*)&params,num\_vaps);

}

--🡪

for (idx=0; idx<vaps\_count; ++idx) {

**ieee\_80211\_iter\_input\_all**(arg, bss\_node[idx]->ni\_vap, (idx == (vaps\_count -1)));

ieee80211\_free\_node(bss\_node[idx]);

}

static INLINE void

ieee80211\_iter\_input\_all(void \*arg, struct ieee80211vap \*vap, bool is\_last\_vap)

{

struct ieee80211\_iter\_input\_all\_arg \*params = (struct ieee80211\_iter\_input\_all\_arg \*) arg;

struct ieee80211\_node \*ni;

wbuf\_t wbuf1;

ni = vap->iv\_bss;

wbuf1 = params->wbuf;

params->type = **ieee80211\_input**(ni, wbuf1, params->rs);

}

int

ieee80211\_input(struct ieee80211\_node \*ni, wbuf\_t wbuf, struct ieee80211\_rx\_status \*rs)

{

wbuf\_set\_node(wbuf, ni);

if (type == IEEE80211\_FC0\_TYPE\_DATA) {

ieee80211\_input\_data(ni, wbuf, rs, subtype, dir);

}

}

/\*

\* processes data frames.

\* ieee80211\_input\_data consumes the wbuf .

\*/

static void

ieee80211\_input\_data(struct ieee80211\_node \*ni, wbuf\_t wbuf, struct ieee80211\_rx\_status \*rs, int subtype, int dir)

{

key = ieee80211\_crypto\_decap(ni, wbuf, hdrspace, rs);

ieee80211\_deliver\_data(vap, wbuf, ni, rs, hdrspace, is\_mcast, is\_bcast, subtype);

}

static void

ieee80211\_deliver\_data(struct ieee80211vap \*vap, wbuf\_t wbuf, struct ieee80211\_node \*ni, struct ieee80211\_rx\_status \*rs,

u\_int32\_t hdrspace, int is\_mcast, int is\_bcast, u\_int8\_t subtype)

{

ieee80211\_deliver\_data\_part1(vap, wbuf, ni, rs, hdrspace);

ieee80211\_deliver\_data\_part2(vap, wbuf, ni, is\_bcast, is\_mcast, subtype);

}

static int

ieee80211\_deliver\_data\_part1(struct ieee80211vap \*vap, wbuf\_t wbuf, struct ieee80211\_node \*ni, struct ieee80211\_rx\_status \*rs,

u\_int32\_t hdrspace)

{

if (!IEEE80211\_VAP\_IS\_DELIVER\_80211\_ENABLED(vap)) {

**//.11🡪.3**

wbuf = **ieee80211\_decap**(vap, wbuf, hdrspace, rs);

}

}

void

**ieee80211\_deliver\_data\_part2**(struct ieee80211vap \*vap, wbuf\_t wbuf, struct ieee80211\_node \*ni, int is\_bcast, int is\_mcast, u\_int8\_t subtype)

{

//**deliver the data frame to the os**. the handler cosumes the wbuf.

vap->iv\_evtable->**wlan\_receive**(vap->iv\_ifp, wbuf, IEEE80211\_FC0\_TYPE\_DATA, subtype, NULL);

}

static void **osif\_receive** (os\_if\_t osif, wbuf\_t wbuf,

u\_int16\_t type, u\_int16\_t subtype,

ieee80211\_recv\_status \*rs)

{

**struct sk\_buff \*skb = (struct sk\_buff \*)wbuf;**

\_\_osif\_deliver\_data(\_osif, \_skb);

}

#define \_\_osif\_deliver\_data(\_osif, \_skb) osif\_deliver\_data(\_osif, \_skb)

void

osif\_deliver\_data(os\_if\_t osif, struct sk\_buff \*skb)

**osif\_deliver\_data\_ol**

{

/\* 交给内核协议 \*/

netif\_rx(skb);

}

## OFFLOAD

adf\_nbuf\_t buf

eth\_hdr = (struct ether\_header \*)(adf\_nbuf\_data(msdu));

[osif\_deliver\_data\_ol:1017]

Call Trace:

[<8028e390>] dump\_stack+0x8/0x34

[<c07bb868>] **osif\_deliver\_data\_ol**+0x68/0x170 [umac]

[<c081a3ec>] ol\_rx\_deliver+0x308/0x380 [umac]

[<c081aac4>] ol\_rx\_indication\_handler+0x3a0/0x9bc [umac]

[<c082d7c4>] htt\_t2h\_msg\_handler\_fast+0x1b4/0x1ec [umac]

[<c08142e8>] CE\_per\_engine\_service\_each+0x590/0x5dc [umac]

[<c08183ec>] ath\_tasklet+0x5c/0x10c [umac]

[<80028bb0>] tasklet\_action+0x88/0xdc

[<80029310>] \_\_do\_softirq+0xb0/0x148

[<800293f0>] do\_softirq+0x48/0x6c

[<80006204>] ath\_dispatch\_wlan\_intr+0x4c/0xa8

[<8000674c>] ret\_from\_irq+0x0/0x4

[<80006940>] r4k\_wait+0x20/0x40

[<8000857c>] cpu\_idle+0x24/0x44

[<803369e0>] start\_kernel+0x2f8/0x310

#if UMAC\_SUPPORT\_PROXY\_ARP

#else

#endif

**ol\_rx\_reorder\_flush**(

vdev, peer, tid, seq\_num\_start,

seq\_num\_end, htt\_rx\_flush\_release);

ol\_ath\_vap\_create()

{

}

vap->**iv\_wrap\_mat\_rx** = **ol\_if\_wrap\_mat\_rx**;

}

ol\_txrx\_pdev\_attach()

{

pdev->**rx\_opt\_proc** = ol\_rx\_fwd\_check;

**pdev->rx\_opt\_proc = ol\_rx\_deliver;**

}

ol\_rx\_deliver()

{

**OL\_RX\_DELIVER\_RAW();**

}

#define **OL\_RX\_DELIVER\_RAW**(\_vdev, \_peer, \_tid, \_msdu\_list) \

ol\_rx\_deliver\_raw((\_vdev), (\_peer), (\_tid), (\_msdu\_list))

ol\_txrx\_osif\_vdev\_register()

{

vdev->osif\_rx = txrx\_ops->rx.std;

// txrx\_ops->rx.std = **osif\_deliver\_data\_ol**

}

**configs/UrouterAC/drivers.makefile: 275:export QCA\_OL\_SUPPORT\_RAWMODE\_TXRX=1**

ol\_rx\_deliver\_raw()

{

}

## Offload RX流

**ol\_txrx\_pdev\_attach**

{

pdev->**rx\_opt\_proc** = **ol\_rx\_deliver**;

}

void

**ol\_rx\_deliver**(

struct ol\_txrx\_vdev\_t \*vdev,

struct ol\_txrx\_peer\_t \*peer,

unsigned tid,

adf\_nbuf\_t msdu\_list)

{

vdev->**osif\_rx**(vdev->osif\_vdev, deliver\_list\_head);

}

ol\_txrx\_osif\_vdev\_register()

{

vdev->**osif\_rx** = txrx\_ops->rx.std;

// txrx\_ops->rx.std = **osif\_deliver\_data\_ol**

}

static void **osif\_vap\_setup**(wlan\_if\_t vap, struct net\_device \*dev,

enum ieee80211\_opmode opmode)

{

ops.rx.std = (ol\_txrx\_rx\_fp) **osif\_deliver\_data\_ol**;

}

**osif\_deliver\_data\_ol**(os\_if\_t osif, struct sk\_buff \***skb\_list**)

{

netif\_rx(skb);

}

# 中断线

ath\_isr(int irq, void \*dev\_id)

{

do\_ath\_isr(irq,dev\_id);

}

**configs/BXO2000n\_2S\_v2/drivers.makefile:#export UMAC\_SUPPORT\_APONLY=1**

#define do\_ath\_isr(\_irq,\_dev\_id) do{\

return ath\_isr\_aponly(\_irq,\_dev\_id); \

}while(0)

irqreturn\_t

ath\_isr\_aponly(int irq, void \*dev\_id)

{

struct net\_device \*dev = dev\_id;

struct ath\_softc\_net80211 \*scn = ath\_netdev\_priv(dev);

sched = ath\_intr\_aponly(scn->sc\_dev);

/\* 激活调度后半部 \*/

**ATH\_SCHEDULE\_TQUEUE**(&scn->sc\_osdev->intr\_tq, &needmark);

// tasklet\_schedule(bh); --🡪 **ath\_tasklet**

}

int

ath\_intr\_aponly(ath\_dev\_t dev)

{

ath\_hal\_getisr(ah, &**status**, HAL\_INT\_LINE, 0);//获取触发中断的原因

sc->sc\_intrstatus |= status;

isr\_status = ath\_common\_intr\_aponly(dev, status);

}

static inline int

ath\_common\_intr\_aponly(ath\_dev\_t dev, HAL\_INT status)

{

struct ath\_softc \*sc = ATH\_DEV\_TO\_SC(dev);

struct ath\_hal \*ah = sc->sc\_ah;

int sched = ATH\_ISR\_NOSCHED;

ath\_rx\_edma\_intr\_aponly(sc, status, &sched);

}

ath\_rx\_edma\_intr\_aponly(ath\_dev\_t dev, HAL\_INT status, int \*sched)

{

struct ath\_softc \*sc = ATH\_DEV\_TO\_SC(dev);

struct ath\_hal \*ah = sc->sc\_ah;

// Atheros 新的芯片系列支持两个收报队列，采用不同的优先级，以保证高优先级队列能够优先处理

if (likely(status & (HAL\_INT\_RXHP | HAL\_INT\_RXEOL | HAL\_INT\_RXORN))) {

**ath\_rx\_intr\_aponly**(dev, **HAL\_RX\_QUEUE\_HP**);

\*sched = ATH\_ISR\_SCHED;

}

if (likely(status & (HAL\_INT\_RXLP | HAL\_INT\_RXEOL | HAL\_INT\_RXORN))) {

**ath\_rx\_intr\_aponly**(dev, **HAL\_RX\_QUEUE\_LP**);

\*sched = ATH\_ISR\_SCHED;

}

}

void

ath\_rx\_intr\_aponly(ath\_dev\_t dev, HAL\_RX\_QUEUE qtype)

{

struct **ath\_rx\_edma** \*rxedma;

rxedma = &sc->sc\_rxedma[qtype];

do

{

**wbuf = rxedma->rxfifo[rxedma->rxfifoheadindex];**

**bf = ATH\_GET\_RX\_CONTEXT\_BUF(wbuf);**

rxs = bf->bf\_desc;

/\***访问流式DMA缓冲区（RAM）**

**刷新D-Cache**

\*/

OS\_SYNC\_SINGLE(sc->sc\_osdev,

bf->bf\_buf\_addr[0], sc->sc\_rxstatuslen, **BUS\_****DMA\_FROMDEVICE**,

OS\_GET\_DMA\_MEM\_CONTEXT(bf, bf\_dmacontext));

/\* 配置些状态 **rxs** \*/

retval = ath\_hal\_rxprocdescfast(ah, NULL, 0, NULL, rxs, wbuf\_raw\_data(wbuf));

/\* bf挂到rxqueue \*/

**TAILQ\_INSERT\_TAIL(&rxedma->rxqueue, bf, bf\_list);**

}while();

**// remove ath\_bufs from free list and add it to fifo**

**ath\_rx\_addbuffer\_aponly(sc, qtype, frames);**

}

static void

ath\_rx\_addbuffer\_aponly(struct ath\_softc \*sc, HAL\_RX\_QUEUE qtype, int size)

{

/\* Add free buffers to rx queue \*/

i = 0;

TAILQ\_FOREACH\_SAFE(bf, &sc->sc\_rxbuf, bf\_list, tbf) {

**if (i == size)**

break;

TAILQ\_REMOVE(&sc->sc\_rxbuf, bf, bf\_list);

if (bf == NULL) {

break;

}

i++;

**ath\_rx\_buf\_link\_aponly(sc, bf, qtype);**

}

}

static inline void

ath\_rx\_buf\_link\_aponly(struct ath\_softc \*sc, struct ath\_buf \*bf, HAL\_RX\_QUEUE qtype)

{

/\*

\*\* Since the descriptor header (48 bytes, which is 64 bytes, 2-3 cache lines

\*\* depending on alignment) is cached, we need to sync to ensure harware sees

\*\* the proper information, and we don't get inconsistent cache data. So sync

\*/

//啥啊？先打扰下设备，但我现在cache里面有数据，不能去rx新的数据

OS\_SYNC\_SINGLE(sc->sc\_osdev, bf->bf\_buf\_addr[0], sc->sc\_rxstatuslen,

**BUS\_DMA\_TODEVICE**, OS\_GET\_DMA\_MEM\_CONTEXT(bf, bf\_dmacontext));

//-🡪// dma\_cache\_sync //**局部映射**

**rxedma->rxfifo[rxedma->rxfifotailindex] = bf->bf\_mpdu;**

/\* push this buffer in the MAC Rx fifo 告知DMA控制器新的总线地址 \*/

ath\_hal\_putrxbuf(ah, bf->**bf\_buf\_addr[0]**, qtype);

}

# 创建AHB设备

int \_\_init

init\_ath\_ahb(void)

{

init\_ahb();

}

int init\_ahb(void)

{

devid = AR9300\_DEVID\_AR9340;

init\_ath\_wmac(devid, 0);

}

static int

init\_ath\_wmac(u\_int16\_t devid, u\_int16\_t wlanNum)

{

struct **net\_device** \*dev;

struct ath\_pci\_softc \*sc；

dev = alloc\_netdev(sizeof(struct ath\_pci\_softc), "**wifi%d**", ether\_setup);

sc = ath\_netdev\_priv(dev);

sc->aps\_osdev.netdev = dev;

**dev->irq** = get\_wmac\_irq(wlanNum); //=2

\_\_ath\_attach(devid, dev, &bc, &sc->**aps\_osdev**)

}

\_\_ath\_attach(u\_int16\_t devid, struct net\_device \*dev, HAL\_BUS\_CONTEXT \*bus\_context, **osdev\_t osdev**)

{

ath\_attach(devid, bus\_context, scn, **osdev**, &ath\_params, &hal\_conf\_parm, &wlan\_reg\_params);

// initialize tx/rx engine

//configs/BXO5000n\_2S\_v2/drivers.makefile:export **ATH\_TXBUF=512**

error = scn->sc\_ops->**tx\_init**(scn->sc\_dev, ATH\_TXBUF);//🡪 ath\_tx\_init

// configs/**BXO5000n\_2S\_v2**/drivers.makefile:export **ATH\_RXBUF=512**

error = scn->sc\_ops->**rx\_init**(scn->sc\_dev, ATH\_RXBUF);// 🡪**ath\_rx\_edma\_init;**

dev->netdev\_ops = &**athdev\_net\_ops**; /\* 在kernel里面用到了呀 \*/

request\_irq(dev->irq, **ath\_isr**, IRQF\_SHARED|IRQF\_SAMPLE\_RANDOM, dev->name, dev);

/\* 创建tasklet处理中断下半部 \*/

ATH\_INIT\_TQUEUE(&osdev->intr\_tq, **ath\_tasklet**, dev);

//🡪 **tasklet\_init**(bh, (adf\_os\_bh\_fn\_t)func, (unsigned long)arg);

}

static const struct net\_device\_ops athdev\_net\_ops = {

.ndo\_open = ath\_netdev\_open,

.ndo\_stop = ath\_netdev\_stop,

**.ndo\_start\_xmit = ath\_netdev\_hardstart,**

.ndo\_set\_mac\_address = ath\_netdev\_set\_macaddr,

.ndo\_tx\_timeout = ath\_netdev\_tx\_timeout,

.ndo\_get\_stats = ath\_getstats,

.ndo\_change\_mtu = ath\_change\_mtu,

.ndo\_set\_multicast\_list = ath\_netdev\_set\_mcast\_list,

**.ndo\_do\_ioctl = ath\_ioctl,**

};

int

ath\_attach(u\_int16\_t devid, void \*base\_addr,

**struct ath\_softc\_net80211 \*scn**,

**osdev\_t osdev**, struct ath\_reg\_parm \*ath\_conf\_parm,

struct hal\_reg\_parm \*hal\_conf\_parm, IEEE80211\_REG\_PARAMETERS \*ieee80211\_conf\_parm)

{

**struct ieee80211com \*ic;**

scn->sc\_osdev = osdev;

ic = &scn->sc\_ic;

ic->ic\_osdev = osdev;

**ic->ic\_vap\_create = ath\_vap\_create;**

ic->ic\_vap\_delete = ath\_vap\_delete;

/\*

\* Create an Atheros Device object

\*/

ath\_dev\_attach(devid, base\_addr,

ic, &**net80211\_ops**, **osdev**,

&**scn->sc\_dev**, **&scn->sc\_ops**,

scn->amem.handle,

ath\_conf\_parm, hal\_conf\_parm);

/\* attach channel width management \*/

error = ath\_cwm\_attach(scn, ath\_conf\_parm);

/\*

\* Set the Atheros Advanced Capabilities from station config before

\* starting 802.11 state machine.

\* 这个东西好像很厉害的样子

\*/

ieee80211com\_set\_athcap(ic, (ops->have\_capability(dev, ATH\_CAP\_BURST) ? IEEE80211\_ATHC\_BURST : 0));

}

int ath\_dev\_attach(u\_int16\_t devid,

void \*bus\_context,

ieee80211\_handle\_t ieee,

struct ieee80211\_ops \*ieee\_ops,

osdev\_t **osdev**,

**ath\_dev\_t \*dev**,

**struct ath\_ops \*\*ops,**

asf\_amem\_instance\_handle amem\_handle,

struct ath\_reg\_parm \*ath\_conf\_parm,

struct hal\_reg\_parm \*hal\_conf\_parm)

{

struct ath\_softc \*sc = NULL;

sc = (struct ath\_softc \*)OS\_MALLOC(osdev, sizeof(struct ath\_softc), GFP\_KERNEL);

**\*dev = (ath\_dev\_t)sc;**

sc->sc\_ath\_ops = ath\_ar\_ops;

**\*ops = &sc->sc\_ath\_ops;**

**sc->sc\_osdev = osdev;**

sc->sc\_ieee = ieee;

sc->sc\_ieee\_ops = ieee\_ops;

ah = \_ath\_hal\_attach(devid, osdev, sc, bus\_context, hal\_conf\_parm,

amem\_handle, &halCallbackTable, &status);

sc->sc\_ah = ah;

// Attach the enhanced DMA module.

ath\_edma\_attach(sc, ops);

}

\_ath\_hal\_attach(u\_int16\_t devid,

HAL\_ADAPTER\_HANDLE osdev,

HAL\_SOFTC sc,

HAL\_BUS\_CONTEXT \*bus\_context,

struct hal\_reg\_parm \*hal\_conf\_parm,

asf\_amem\_instance\_handle amem\_handle,

const struct ath\_hal\_callback \*callbackTable,

void \*s)

{

ah = ath\_hal\_attach(

devid, osdev, sc,

bus\_context->bc\_info.bc\_tag, bus\_context->bc\_handle, bus\_context->bc\_bustype,

amem\_handle, hal\_conf\_parm, &status);

}

struct ath\_hal\* \_\_ahdecl

ath\_hal\_attach(u\_int16\_t devid, HAL\_ADAPTER\_HANDLE osdev, HAL\_SOFTC sc,

HAL\_BUS\_TAG st, HAL\_BUS\_HANDLE sh, HAL\_BUS\_TYPE bustype,

asf\_amem\_instance\_handle amem\_handle,

struct hal\_reg\_parm \*hal\_conf\_parm, HAL\_STATUS \*error)

{

case AR9300\_DEVID\_AR9340:

ah = ar9300Attach(devid, osdev, sc, st, sh, bustype, amem\_handle,

hal\_conf\_parm, error);

}

ar9300Attach(u\_int16\_t devid, HAL\_ADAPTER\_HANDLE osdev, HAL\_SOFTC sc,

HAL\_BUS\_TAG st, HAL\_BUS\_HANDLE sh, HAL\_BUS\_TYPE bustype,

asf\_amem\_instance\_handle amem\_handle,

struct hal\_reg\_parm \*hal\_conf\_parm, HAL\_STATUS \*status)

{

struct ath\_hal\_9300 \*ahp;

struct ath\_hal \*ah;

struct ath\_hal\_private \*ahpriv;

ahp = ar9300NewState(

devid, osdev, sc, st, sh, bustype, amem\_handle, hal\_conf\_parm, status);

ah = &ahp->ah\_priv.priv.h;

/\* 下面大概是在初始化wifi芯片 和各种结构体 \*/

if (!ar9300SetResetReg(ah, **HAL\_RESET\_POWER\_ON**)) { /\* reset chip \*/

if (!ar9300SetPowerMode(ah, HAL\_PM\_AWAKE, AH\_TRUE)) {

}

struct ath\_hal\_9300 \*

ar9300NewState(u\_int16\_t devid, HAL\_ADAPTER\_HANDLE osdev, HAL\_SOFTC sc,

HAL\_BUS\_TAG st, HAL\_BUS\_HANDLE sh, HAL\_BUS\_TYPE bustype,

asf\_amem\_instance\_handle amem\_handle,

struct hal\_reg\_parm \*hal\_conf\_parm, HAL\_STATUS \*status)

{

struct ath\_hal\_9300 \*ahp;

struct ath\_hal \*ah;

ahp = amalloc\_adv(

amem\_handle, sizeof(struct ath\_hal\_9300), adf\_os\_mem\_zero\_outline);

**ah = &ahp->****ah\_priv.priv.h;**

OS\_MEMCPY(&ahp->ah\_priv.priv, &ar9300hal, sizeof(ahp->ah\_priv.priv));

ah->ah\_osdev = osdev;

ah->ah\_sc = sc;

return ahp;

}

void

ath\_edma\_attach(struct ath\_softc \*sc, struct ath\_ops \*\*ops)

{

if (ath\_hal\_hasenhanceddmasupport(sc->sc\_ah)) {

**sc->sc\_enhanceddmasupport = 1;**

**(\*ops)->rx\_init = ath\_rx\_edma\_init;**

**(\*ops)->rx\_proc = ath\_rx\_edma\_tasklet;**

**(\*ops)->rx\_requeue = ath\_rx\_edma\_requeue;**

**(\*ops)->rx\_cleanup = ath\_rx\_edma\_cleanup;**

**(\*ops)->tx\_proc = ath\_tx\_edma\_tasklet;**

} else {

sc->sc\_enhanceddmasupport = 0;

}

}

int

ath\_rx\_edma\_init(ath\_dev\_t dev, int **nbufs**) // **BXO5000n\_2S\_v2 : nbufs=512**

{

struct ath\_softc \*sc = ATH\_DEV\_TO\_SC(dev);

wbuf\_t wbuf;

struct ath\_rx\_status \*rxs;

struct ath\_buf \*bf;

sc->sc\_rxbufsize = OS\_MAX\_RXBUF\_SIZE(sc->sc\_rxstatuslen);

// Sets receive buffer size in the hardware.

ath\_hal\_setrxbufsize(sc->sc\_ah, sc->sc\_rxbufsize - sc->sc\_rxstatuslen);

ath\_rxfifo\_alloc(sc, HAL\_RX\_QUEUE\_HP);

ath\_rxfifo\_alloc(sc, HAL\_RX\_QUEUE\_LP);

/\* allocate ath\_buf pool \*/

bsize = sizeof(struct ath\_buf) \* **nbufs**;

bf = (struct ath\_buf \*)OS\_MALLOC(sc->sc\_osdev, bsize, GFP\_KERNEL);

/\* allocate ath\_rx\_status pool \*/

bsize = sizeof(struct ath\_rx\_status) \* nbufs;

rxs = (struct ath\_rx\_status \*)OS\_MALLOC(sc->sc\_osdev, bsize, GFP\_KERNEL);

for (i = 0; i < nbufs; i++, **bf++**, rxs++) {

**wbuf** = ath\_rxbuf\_alloc(sc, sc->sc\_rxbufsize);

bf->bf\_mpdu = **wbuf**;

/\*传说中的流式DMA映射。

\***一旦缓冲区被映射，它将属于设备，而不是处理器**

**\* FIFO一满就会DMA到这里来吧**

\*/

bf->bf\_buf\_addr[0] = wbuf\_map\_single(sc->sc\_osdev, **wbuf**, **BUS\_DMA\_FROMDEVICE**,

OS\_GET\_DMA\_MEM\_CONTEXT(bf, bf\_dmacontext));

ATH\_SET\_RX\_CONTEXT\_BUF(wbuf, bf);

bf->bf\_desc = (void \*)rxs;

/\* 把 bf 串起来 \*/

TAILQ\_INSERT\_TAIL(&sc->sc\_rxbuf, bf, bf\_list);

}

}

/\* 这个fifo是为了要保存硬件FIFO的数据，先申请的RAM \*/

static int

ath\_rxfifo\_alloc(struct ath\_softc \*sc, HAL\_RX\_QUEUE qtype)

{

struct ath\_rx\_edma \*rxedma;

int bsize, error;

rxedma = &sc->sc\_rxedma[qtype];

error = ath\_hal\_getrxfifodepth(sc->sc\_ah, qtype, **&rxedma->rxfifohwsize**);

bsize = sizeof(**wbuf\_t**) \* rxedma->rxfifohwsize;

rxedma->**rxfifo** = (wbuf\_t \*)OS\_MALLOC(sc->sc\_osdev, **bsize**, GFP\_KERNEL);

}

//但是不知道这个函数在哪里调用的。

int

ath\_edma\_startrecv(struct ath\_softc \*sc)

{

//设置了DMA的映射

OS\_EXEC\_INTSAFE(sc->sc\_osdev, ath\_rx\_addbuffer\_intsafe, sc);

}

/\*

\* Initialize TX queue and h/w

\*/

int

ath\_tx\_init(ath\_dev\_t dev, int nbufs)

{

/\* Setup tx descriptors \*/

error = ath\_descdma\_setup(sc, &sc->sc\_txdma, &**sc->sc\_txbuf**,

"tx", nbufs, ATH\_TXDESC, 1, ATH\_FRAG\_PER\_MSDU);

}

int

ath\_descdma\_setup(

struct ath\_softc \*sc,

struct ath\_descdma \*dd, ath\_bufhead \*head,

const char \*name, int nbuf, int ndesc, int is\_tx, int frag\_per\_msdu)

{

u\_int8\_t \*ds;

struct ath\_buf \*\*bf\_arr;

/\* allocate **descriptors** \*/

// **dd->dd\_desc\_paddr:物理地址，总线地址**

**// 返回虚拟地址**

dd->dd\_desc = (void \*)OS\_MALLOC\_CONSISTENT(**sc->sc\_osdev**,

**dd->dd\_desc\_len**, &**dd->dd\_desc\_paddr**,

OS\_GET\_DMA\_MEM\_CONTEXT(dd, dd\_dmacontext),

sc->sc\_reg\_parm.shMemAllocRetry);

**ds = dd->dd\_desc;**

/\* allocate buffers \*/

//一次只能分配128K连续的内存，分成buf\_arr\_size次alloc

buf\_arr\_size = ((nbuf \* sizeof(struct ath\_buf))/MAX\_BUF\_MEM\_ALLOC\_SIZE + 2) ;

/\* one extra element is reserved at end to indicate the end of array \*/

bf\_arr = (struct ath\_buf \*\*)OS\_MALLOC(sc->sc\_osdev,

(sizeof(struct ath\_buf \*) \* buf\_arr\_size), GFP\_KERNEL);

//每次alloc的bf的个数

nbuf\_alloc = **(MAX\_BUF\_MEM\_ALLOC\_SIZE / (sizeof(struct ath\_buf)))**;

for(j = 0; (nbuf\_left && (j < (buf\_arr\_size-1))); j++)

{

nbuf\_alloc = MIN(nbuf\_left, nbuf\_alloc);

alloc\_size = (nbuf\_alloc \* sizeof(struct ath\_buf));

**bf\_arr[j] = (struct ath\_buf \*)OS\_MALLOC(sc->sc\_osdev, alloc\_size, GFP\_KERNEL);**

for (i = 0; i < nbuf\_alloc; i++, bf++, ds += (desc\_len \* ndesc)) {

bf->bf\_desc = ds; **/\* virtual** addr of desc **\*/**

bf->bf\_daddr = DS2PHYS(dd, ds); /\* **physical** addr of desc \*/

//bf挂到head()上

TAILQ\_INSERT\_TAIL(head, bf, bf\_list);

} }

}

# ACS

Kochab中channel为0时会下发auto select channel

umac/scan/ieee80211\_scan.c implements the scanning task, which is started by the connection manager using the API ***wlan\_scan\_start*.** The client scans to find the infrastructure APs available in all the allowed channels. The list of channels to be scanned is created by ***ieee80211\_scan\_ update\_channel\_list*** during the initialization of the generic scanner task (*ieee80211\_scan\_attach*). This list is actually derived from a list of allowed channels prepared by the HAL module based on the device capabilities (bands of operation supported) and regulatory domain configured. The client scans by switching to each channel and collects the AP information from the beacons received. The available infrastructure networks (APs) are stored in the scan table. From this scan table, a candidate AP list is built (umac/scan/ieee80211\_aplist.c) that consists of those APs with SSIDs matching the desired SSIDs chosen by the user.

//配置信道

static int

**ieee80211\_ioctl\_siwfreq**(struct net\_device \*dev,

struct iw\_request\_info \*info,

struct iw\_freq \*freq, char \*extra)

{

retval = **wlan\_set\_channel**(vap, **i**); //i = **IEEE80211\_CHAN\_ANY** (i = **-1**)

}

int

**wlan\_set\_channel**(wlan\_if\_t vaphandle, int **chan**)

{

if (**chan == IEEE80211\_CHAN\_ANY**) { //自动扫描

if (**vap->iv\_opmode == IEEE80211\_M\_HOSTAP**) {

/\* allow IEEE80211\_CHAN\_ANYC for auto channel select in AP mode\*/

vap->iv\_des\_chan[vap->iv\_des\_mode] = IEEE80211\_CHAN\_ANYC;

TAILQ\_FOREACH(tmpvap, &vap->iv\_ic->ic\_vaps, iv\_next){

if(tmpvap->iv\_des\_chan[tmpvap->iv\_des\_mode])

tmpvap->iv\_des\_chan[tmpvap->iv\_des\_mode] = IEEE80211\_CHAN\_ANYC;

}/\* zhangjj@BHU: sync all vaps of same wifi to ANYC \*/

/\* Trigger EACS only when vap is ready or channel change issued by DCS module \*/

if (ieee80211\_vap\_ready\_is\_set(vap) || (ic->cw\_inter\_found)) {

**//扫描并自动选取完信道后的处理函数的注册**

**wlan\_autoselect\_register\_event\_handler(vap, &****spectral\_eacs\_event\_handler, (void \*)vap);**

**//注册并调用扫描**

**wlan\_autoselect\_find\_infra\_bss\_channel(vap);**

} else {

IEEE80211\_DPRINTF(vap, IEEE80211\_MSG\_ANY,"%s: failed due to vap not ready",\_\_func\_\_);

}

return 0;

} else {

}

} else {

}

}

**wlan\_autoselect\_register\_event\_handler ---🡪**

acs->**acs\_event\_handlers**[acs->acs\_num\_handlers] = evhandler;

//🡪 **spectral\_eacs\_event\_handler (在ieee80211\_acs\_post\_event()调用)**

acs->**acs\_event\_handler\_arg**[acs->acs\_num\_handlers++] = arg;

//🡪 **vap**

/\*

\* Auto Channel Select handler used for interface up.

\*/

static void spectral\_eacs\_event\_handler(void \*arg, wlan\_chan\_t channel)

{

error = **wlan\_set\_channel**(vap, chan);

}

//启动扫描

int **wlan\_autoselect\_find\_infra\_bss\_channel**(wlan\_if\_t vaphandle)

{

return **ieee80211\_autoselect\_infra\_bss\_channel**(**vaphandle**, **false** /\* is\_scan\_report \*/);

}

static int **ieee80211\_autoselect\_infra\_bss\_channel**(struct ieee80211vap \*vap, bool is\_scan\_report)

{

acs->acs\_scan\_req\_param.acs\_scan\_report\_active = is\_scan\_report;

if(acs->acs\_scan\_req\_param.acs\_scan\_report\_active) {

ieee80211\_acs\_construct\_chan\_list(acs, IEEE80211\_MODE\_AUTO);

} else {

//构建自动选择信道列表。不是扫描信道列表

**ieee80211\_acs\_construct\_chan\_list(acs,acs->acs\_vap->iv\_des\_mode);**

}

//挂载扫描信道信息读取函数

**rc = wlan\_scan\_register\_event\_handler(vap,** **ieee80211\_acs\_scan\_evhandler, (void \*) acs);**

/\*Flush scan table before starting scan \*/

wlan\_scan\_table\_flush(vap);

ieee80211\_acs\_flush\_olddata(acs);

if (**wlan\_scan\_start**(vap, //--🡪 **ath\_scan\_start**

&scan\_params,

acs->acs\_scan\_requestor,

IEEE80211\_SCAN\_PRIORITY\_HIGH,

&(acs->acs\_scan\_id)) != EOK) {}

}

ss->**ss\_event\_handlers**[ss->ss\_num\_handlers] = evhandler; //ieee80211\_acs\_scan\_evhandler

ss->ss\_event\_handler\_arg[ss->ss\_num\_handlers++] = arg;

/\*

\* 构建选择信道列表 \***construct the available channel list**

\*/

static void **ieee80211\_acs\_construct\_chan\_list**(ieee80211\_acs\_t acs, enum ieee80211\_phymode mode)

{

} else if (mode == IEEE80211\_MODE\_11AC\_VHT40) {

eacs\_trace(EACS\_DBG\_LVL0,( " get channels with phy mode VHT40 plus and minuse"));

/\* if PHY mode is not AUTO, get channel list by PHY mode directly \*/

**ieee80211\_acs\_get\_phymode\_channels(acs, IEEE80211\_MODE\_11AC\_VHT40PLUS);**

**ieee80211\_acs\_get\_phymode\_channels(acs, IEEE80211\_MODE\_11AC\_VHT40MINUS);**

} else if (mode == IEEE80211\_MODE\_11AC\_VHT80) {

}

static INLINE void **ieee80211\_acs\_get\_phymode\_channels**(ieee80211\_acs\_t acs, enum ieee80211\_phymode mode)

{

struct ieee80211\_channel \*channel;

int i, extchan, skipchanextinvalid;

struct ieee80211\_channel\_list chan\_info;

//**for(i = 0, channel = acs->acs\_ic->****ic\_channels;**

**i < acs->acs\_ic->ic\_nchans;**

**i++, channel++)**

**ieee80211\_enumerate\_channels(channel, acs->acs\_ic, i)** {

**acs->****acs\_chans[acs->acs\_nchans++] = channel;**

**//这个列表只是要选择的列表，即将会根据扫描的信息在这个列表里面选择一个信道。这个列表不是要扫描的列表。**

}

}

/\*

\* Iterator for channel list

\*/

#define **ieee80211\_enumerate\_channels**(\_c, \_ic, \_index) \

for ((\_index) = 0, (\_c) = (\_ic)->ic\_channels; \

(\_index) < (\_ic)->ic\_nchans; \

(\_index)++, (\_c)++)

**vaphandle->****iv\_ic->ic\_scanner**

/\*

\* scan handler used for **scan complete**

\*/

static **void ieee80211\_acs\_scan\_evhandler**(struct ieee80211vap \*originator, ieee80211\_scan\_event \*event, void \*arg)

{

//获取扫描到的信道信息。

if( event->type == IEEE80211\_SCAN\_FOREIGN\_CHANNEL\_**GET\_NF** ) {

now = (u\_int32\_t) CONVERT\_SYSTEM\_TIME\_TO\_MS(OS\_GET\_TIMESTAMP());

flags = ACS\_CHAN\_STATS\_NF;

/\* Get the noise floor value \*/

acs->acs\_noisefloor[ieee80211\_chan2ieee(originator->iv\_ic,event->chan)] = ic->ic\_get\_cur\_chan\_nf(ic);

eacs\_trace(EACS\_DBG\_SCAN,("Requesting for CHAN STATS and NF from Target \n"));

eacs\_trace(EACS\_DBG\_SCAN,("%d.%03d | %s:request stats and nf \n", now / 1000, now % 1000, \_\_func\_\_));

ic->**ic\_hal\_get\_chan\_info**(ic, flags); //🡪**ath\_net80211\_get\_chan\_info**

}

/\* To prevent channel selection when acs report is active \*/

if(!acs->acs\_scan\_req\_param.acs\_scan\_report\_active) {

//传说中的信道选择。

channel = **ieee80211\_acs\_find\_best\_centerchan(acs)**;

}

**scan\_done: //扫描完成**

**ieee80211\_acs\_post\_event(acs, channel);**

}

static void

**ieee80211\_acs\_post\_event**(ieee80211\_acs\_t acs,

struct ieee80211\_channel \*channel)

{

for (i=0; i < num\_handlers; ++i) {

acs\_event\_handlers[i] = **acs->acs\_event\_handlers[i]**;

acs\_event\_handler\_arg[i] = **acs->acs\_event\_handler\_arg[i]**;

}

spin\_unlock(&acs->acs\_lock);

for (i = 0; i < num\_handlers; ++i) {

(**acs\_event\_handlers**[i]) (**acs\_event\_handler\_arg[i]**, **channel**);

}

}

ic->**ic\_hal\_get\_chan\_info** = **ath\_net80211\_get\_chan\_info**; (at : ath\_attach())

/\*

\* **Read NF and channel load registers and invoke ACS update API**

\*/

static void ath\_net80211\_get\_chan\_info(struct ieee80211com \*ic, u\_int8\_t flags)

{

struct ath\_softc\_net80211 \*scn = ATH\_SOFTC\_NET80211(ic);

u\_int ieee\_chan\_num;

struct ieee80211\_chan\_stats chan\_stats;

int16\_t acs\_noisefloor = 0;

**ieee80211\_acs\_stats\_update**(ic->ic\_acs, flags, ieee\_chan\_num,

acs\_noisefloor, &chan\_stats);

}

## 设置扫描信道集合1

**（BXO5000n\_2S\_v2/AR9344）**

/\*

\* **Set country code**

\*/

int

**ieee80211\_set\_country\_code**(struct ieee80211com \*ic, char \*isoName, u\_int16\_t cc, enum ieee80211\_clist\_cmd cmd)

{

error = ic->**ic\_set\_country**(ic, isoName, cc, cmd);

//🡪 ***ath\_net80211\_set\_country***

/\* update channel list \*/

ieee80211\_update\_channellist(ic, 1);

**ieee80211\_build\_countryie\_all(ic);**

}

**//设置设备支持的信道列表**

***ic->ic\_set\_country = ath\_net80211\_set\_country;***

ath\_net80211\_set\_country(struct ieee80211com \*ic, char \*isoName, u\_int16\_t cc, enum ieee80211\_clist\_cmd c

{

struct ath\_softc\_net80211 \*scn = ATH\_SOFTC\_NET80211(ic);

int retval = 0;

retval = **scn->sc\_ops->set\_country**(scn->sc\_dev, isoName, cc, cmd);

//--🡪 **ath\_set\_country**, /\* set\_country \*/

return retval;

}

/\*

\* Set the **802.11D** country

\* 设置设备支持的信道。

\*/

static int

**ath\_set\_country**(ath\_dev\_t dev, char \*isoName, u\_int16\_t cc, enum ieee80211\_clist\_cmd cmd)

{

//先到EEPROM读表

**ath\_hal\_init\_channels**(ah, **chans**, IEEE80211\_CHAN\_MAX, (u\_int \*)&nchan,

regclassids, ATH\_REGCLASSIDS\_MAX, &nregclass,

sc->sc\_config.ath\_countrycode, wMode, outdoor,

sc->sc\_config.ath\_xchanmode,

sc->sc\_is\_blockdfs\_set));

**//设置给VAP**

if (sc->sc\_ieee\_ops->setup\_channel\_list) {

sc->sc\_ieee\_ops->**setup\_channel\_list**(sc->sc\_ieee, CLIST\_UPDATE,

**chans**, nchan, regclassids, nregclass,

CTRY\_DEFAULT);//

}

//-🡪 **ath\_net80211\_channel\_setup**

}

/\*

\* **update the country ie in all vaps**.

\*/

static void

**ieee80211\_build\_countryie\_all**(struct ieee80211com \*ic)

{

u\_int8\_t num\_vaps;

ieee80211\_iterate\_vap\_list\_internal(ic,**ieee80211\_build\_countryie\_vap**,NULL,num\_vaps);

}

static void

**ieee80211\_build\_countryie\_vap**(void \*arg,struct ieee80211vap \*vap, bool is\_last\_vap)

{

**ieee80211\_build\_countryie**(vap);

}

/\*

\* Build the country information element.

\*/

void

ieee80211\_build\_countryie(struct ieee80211vap \*vap)

{

ieee80211\_enumerate\_channels(c, ic, i) {

}

}

net80211\_ops-> **setup\_channel\_list** // **ath\_net80211\_channel\_setup**

//创建channel列表

// Callbacks for **ath\_dev** module, which calls net80211 API's

//(ieee80211\_xxx) accordingly.

static void

**ath\_net80211\_channel\_setup**(ieee80211\_handle\_t ieee,

enum ieee80211\_clist\_cmd cmd,

const HAL\_CHANNEL \***chans**, int nchan,

const u\_int8\_t \*regclassids, u\_int nregclass,

int countrycode) {

if ((countrycode == CTRY\_DEFAULT) || (cmd == CLIST\_NEW\_COUNTRY)) {

/\*

\* Convert HAL channels to ieee80211 ones.

\*/

for (i = 0; i < nchan; i++) {

c = **&chans[i];**

**ichan = ieee80211\_get\_channel(ic, i);//** **ic ->ic\_channels[i]**

OS\_MEMZERO(ichan, sizeof(struct ieee80211\_channel));

**IEEE80211\_CHAN\_SETUP**(ichan,

scn->sc\_ops->mhz2ieee(scn->sc\_dev, c->channel, c->channel\_flags),

c->channel,

c->channel\_flags,

0, /\* ic\_flagext \*/

c->max\_reg\_tx\_power, /\* dBm \*/

c->max\_tx\_power / 4, /\* 1/4 dBm \*/

c->min\_tx\_power / 4, /\* 1/4 dBm \*/

c->regClassId

);

}

}

}

/\*

\* Setup the channel list based on the information in the EEPROM and

\* any supplied country code. Note that we also do a bunch of EEPROM

\* verification here and setup certain regulatory-related access

\* control data used later on.

\*/

bool \_\_ahdecl

ath\_hal\_init\_channels(struct ath\_hal \*ah,

HAL\_CHANNEL \*chans, u\_int maxchans, u\_int \*nchans,

u\_int8\_t \*regclassids, u\_int maxregids, u\_int \*nregids,

HAL\_CTRY\_CODE cc, u\_int32\_t modeSelect,

bool enableOutdoor, bool enableExtendedChannels,

bool block\_dfs\_enable)

{

}

static INLINE int ieee80211\_scan\_update\_channel\_list(ieee80211\_scanner\_t ss)

{

return ((struct ieee80211\_scanner\_common \*)ss)->**scan\_update\_channel\_list(ss)**;

}

(\*ss)->ss\_common.scan\_update\_channel\_list = **\_ieee80211\_scan\_update\_channel\_list**;

（ set at *\_ieee80211\_scan\_attach()*）

**\_ieee80211\_scan\_update\_channel\_list//****设置要扫描的集合**

**ol\_scan\_update\_channel\_list //ofload: 设置要扫描的集合**

## 设置扫描信道集合2

**（BXO5000ac\_2S\_lite/AR9344/*offload*/11ac）**

**//设置设备支持的信道列表。**

**ol\_ath\_set\_country**(){

**chans** = ic->**ic\_channels**;

if (!ol\_regdmn\_init\_channels(ol\_regdmn\_handle, **chans**, IEEE80211\_CHAN\_MAX, (u\_int \*)&nchan,

regclassids, ATH\_REGCLASSIDS\_MAX, &nregclass,

cc, wMode, outDoor, xchanMode,

scn->sc\_is\_blockdfs\_set)) {

}

}

**ic\_dfs\_clist\_update**

**ol\_regdmn\_init\_channels**(struct ol\_regdmn \*ol\_regdmn\_handle,

struct ieee80211\_channel \***chans**, u\_int maxchans, u\_int \*nchans,

u\_int8\_t \*regclassids, u\_int maxregids, u\_int \*nregids,

REGDMN\_CTRY\_CODE cc, u\_int32\_t modeSelect,

bool enableOutdoor, bool enableExtendedChannels,

bool block\_dfs\_enable){

… …

OS\_MEMCPY(&**chans**[next++], &**icv**, sizeof(struct **ieee80211\_channel**));

… …

\*nchans = next;

ieee80211\_set\_nchannels(ic, next);

}

## 扫描

**ieee80211\_scan.c**

**ieee80211\_find\_channel**

**ss->ss\_all\_chans**

ath\_scan\_start, /\* scan\_start \*/

**//设置所有要扫描的信道。**

**int ieee80211\_scan\_update\_channel\_list(ieee80211\_scanner\_t ss)**

**{**

**//设置这次扫描的信道**

**static void scanner\_construct\_chan\_list(ieee80211\_scanner\_t ss)**

**{**

/\*

\* **get all the channels to scan** . 这个应该就是设置要扫描的信道了吧。

\* can be called whenever the set of supported channels are changed.

\* (ex: when a beacon with valid country code received )

\*/

static int **\_ieee80211\_scan\_update\_channel\_list**(ieee80211\_scanner\_t ss)

{

}

int **ieee80211\_scan\_run**(ieee80211\_scanner\_t ss,

wlan\_if\_t vaphandle,

scan\_request\_data\_t \*request\_data)

{

**scanner\_construct\_chan\_list**(ss, vaphandle);

if (ss->ss\_max\_scan\_time != 0) {

u\_int32\_t elapsed\_scan\_time;

u\_int32\_t remaining\_time;

elapsed\_scan\_time = CONVERT\_SYSTEM\_TIME\_TO\_MS(current\_system\_time - ss->ss\_scan\_start\_time);

if (ss->ss\_max\_scan\_time > elapsed\_scan\_time) {

remaining\_time = ss->ss\_max\_scan\_time - elapsed\_scan\_time;

}

else {

remaining\_time = 0;

}

//设置扫描定时器

**OS\_SET\_TIMER**(&ss->**ss\_maxscan\_timer**, **remaining\_time**);

}

}

OS\_INIT\_TIMER(osdev, &((\*ss)->**ss\_maxscan\_timer**), scanner\_timer\_maxtime\_handler, (void \*) (\*ss));

#define OS\_TIMER\_FUNC(\_fn) \

void \_fn(void \*timer\_arg)

static **OS\_TIMER\_FUNC**(**scanner\_timer\_maxtime\_handler**)

{ //--🡪 scanner\_timer\_maxtime\_handler\_fn

ieee80211\_scanner\_t ss;

OS\_GET\_TIMER\_ARG(ss, ieee80211\_scanner\_t);

**ieee80211\_sm\_dispatch**(ss->ss\_hsm\_handle,

**SCANNER\_EVENT\_MAXSCANTIME\_EXPIRE**,

0,

NULL);

}

static bool **is\_common\_event**(ieee80211\_scanner\_t ss,

enum scanner\_event event,

u\_int16\_t event\_data\_len,

void \*event\_data)

{

switch (event) {

case **SCANNER\_EVENT\_MAXSCANTIME\_EXPIRE**:

termination\_reason = **IEEE80211\_REASON\_TIMEDOUT**;

break;

}

}

**scanner\_post\_event**(ieee80211\_scanner\_t ss,

wlan\_if\_t vaphandle,

ieee80211\_scan\_event\_type type,

ieee80211\_scan\_completion\_reason reason,

IEEE80211\_SCAN\_ID scan\_id,

IEEE80211\_SCAN\_REQUESTOR requestor,

wlan\_chan\_t chan)

{

}

(\*ss)->ss\_common.scan\_register\_event\_handler = **\_ieee80211\_scan\_register\_event\_handler**;

# 自动选择扫描信道跳过：

1. ath\_net80211\_channel\_setup：设备支持的信道，去掉信新到后，将不支持去掉的信道，**页面**

**offload: ol\_regdmn\_init\_channels**

1. **\_ieee80211\_scan\_update\_channel\_list：设置所有要扫描的集合，需要在选择的集合中去掉**

**//offload ol\_scan\_update\_channel\_list**

1. **scanner\_construct\_chan\_list：设置这次扫描的信道**
2. **ieee80211\_acs\_construct\_chan\_list：设置要选择的信道集合列表。**
3. **2,4来自于1；3来自于2；**

# 问题：wlan\_mlme\_stop\_bss()

**IEEE80211\_DPRINTF**

wlan\_set\_debug\_flags

asf\_print\_mask\_set

ieee80211\_ioctl\_setparam

//offload打印debug的地方

**ol\_ath\_log\_text**(struct ieee80211com \*ic, char \*text)

osif\_acs\_start\_bss

wlan\_mlme\_start\_bss

**osif\_vap\_init**(struct net\_device \*dev, int forcescan){

if()

{

/\* Wait for previous vdev\_stop\_command to complete \*/

}

}

ol\_vdev\_wmi\_event\_handler

Call Trace:

[<8028f444>] dump\_stack+0x8/0x34

[<c0d87dac>] wlan\_autoselect\_register\_event\_handler+0x50/0x130 [umac]

[<c0db74f0>] osif\_vap\_init+0x6d8/0xcc8 [umac]

[<c0dab498>] ieee80211\_ioctl\_siwfreq+0x250/0x5ac [umac]

//直接调用ACS是失败在osif\_vap\_init启动了ACS

[<80289580>] ioctl\_standard\_call+0x98/0x3d8

[<80289a20>] wext\_handle\_ioctl+0xbc/0x218

[<801bb2ac>] dev\_ioctl+0x734/0x78c

[<8007fa4c>] vfs\_ioctl+0x2c/0x78

[<80080150>] do\_vfs\_ioctl+0x5bc/0x610

[<800801f4>] sys\_ioctl+0x50/0x8c

[<8000ea84>] stack\_done+0x20/0x3c

int wlan\_mlme\_stop\_bss(wlan\_if\_t vaphandle, int flags)

{

while (**vap->init\_in\_progress**) {

if (waitcnt >= OSIF\_MAX\_STOP\_VAP\_TIMEOUT\_CNT) {

ieee80211\_note (vap,"%s(): Timeout for start response event!!!\n", \_\_func\_\_);

error = -1;

goto out;

}

schedule\_timeout\_interruptible(OSIF\_STOP\_VAP\_TIMEOUT);

waitcnt++;

**ieee80211\_note (vap,"%s(): Init in progress. Delay vap\_stop\n", \_\_func\_\_);**

}

}

**ol\_vdev\_wmi\_event\_handler**（）{

vaphandle->init\_in\_progress = **false**;

}

3.4.5.22 umac/**resmgr**

The **Resource Manager** directory contains some functions to handle the driver global resources.

These functions are called from multiple locations within the driver to allocate and free these

resources.

ieee80211\_resmgr\_t ol\_resmgr\_create（）

{

/\* Register WMI event handlers \*/

**wmi\_unified\_register\_event\_handler**(scn->wmi\_handle, WMI\_VDEV\_START\_RESP\_EVENTID, **ol\_vdev\_wmi\_event\_handler**, resmgr);

}

**wmi\_unified\_event\_rx**

# WMI

ieee80211\_resmgr\_t **ol\_resmgr\_create**（）

{

/\* Register WMI event handlers \*/

**wmi\_unified\_register\_event\_handler**(scn->wmi\_handle, WMI\_VDEV\_START\_RESP\_EVENTID, **ol\_vdev\_wmi\_event\_handler**, resmgr);

}

int wmi\_unified\_register\_event\_handler(wmi\_unified\_t wmi\_handle,

WMI\_EVT\_ID event\_id,

wmi\_unified\_event\_handler handler\_func,

void\* cookie)

{

wmi\_handle->**event\_handler**[handler\_id] = handler\_func;

wmi\_handle->event\_handler\_cookie[handler\_id] = cookie;

}

static int **wmi\_unified\_event\_rx**(struct wmi\_unified \*wmi\_handle, wmi\_buf\_t evt\_buf)

{

/\* Call the WMI registered event handler \*/

status = wmi\_handle->**event\_handler**[handler\_id](wmi\_handle->scn\_handle, event, len,

wmi\_handle->event\_handler\_cookie[handler\_id]);

}

**wmi\_control\_rx**()

{

**wmi\_unified\_event\_rx**

}

wmi\_unified\_connect\_htc\_service()

{

connect.EpCallbacks.EpRecv = **wmi\_control\_rx** /\* Control path rx \*/;

if ((status = **HTCConnectService**(htc\_handle, &connect, &response)) != EOK)

}

**htc\_service ????**

## OL vap\_start +

[<8028f444>] dump\_stack+0x8/0x34

[<c0de2544>] **wmi\_unified\_vdev\_start\_send**+0x74/0x180 [umac]

[<c0df370c>] \_ieee80211\_resmgr\_vap\_start+0xa8/0x174 [umac]

[<c0d59158>] mlme\_create\_infra\_bss+0x19c/0x3a0 [umac]

[<c0db3c7c>] osif\_acs\_start\_bss+0x28/0xf8 [umac]

[<c0db7e34>] osif\_acs\_event\_handler+0xf0/0x104 [umac]

[<c0d79768>] ieee80211\_acs\_post\_event+0xa0/0x10c [umac]

[<c0d7eac4>] ieee80211\_autoselect\_infra\_bss\_channel+0x60c/0x940 [umac]

[<c0db8530>] osif\_vap\_init+0x6e8/0xcc8 [umac]

[<801ba3c0>] **dev\_open**+0xe8/0x15c

[<801b9780>] dev\_change\_flags+0xd0/0x1c8

[<802236e8>] devinet\_ioctl+0x2d4/0x80c

[<801a5f04>] sock\_ioctl+0x29c/0x2f4

[<8007fa4c>] vfs\_ioctl+0x2c/0x78

[<80080150>] do\_vfs\_ioctl+0x5bc/0x610

[<800801f4>] sys\_ioctl+0x50/0x8c

[<8000ea84>] stack\_done+0x20/0x3c

\_ieee80211\_resmgr\_vap\_start(

avn->**av\_ol\_resmgr\_wait** = TRUE;

**wmi\_unified\_vdev\_start\_send**();

vap->**init\_in\_progress** = true;

)

wmi\_unified\_vdev\_start\_send(wmi\_unified\_t wmi\_handle, u\_int8\_t if\_id, struct ieee80211\_channel \*chan,

u\_int32\_t freq, bool disable\_hw\_ack, void \*nl\_handle)

{

return **wmi\_unified\_cmd\_send**(wmi\_handle, buf, len, WMI\_VDEV\_START\_REQUEST\_CMDID);

}

/\* WMI command API \*/

int

**wmi\_unified\_cmd\_send**(wmi\_unified\_t wmi\_handle, wmi\_buf\_t buf, int len, WMI\_CMD\_ID cmd\_id)

{

status = **HTCSendPkt**(wmi\_handle->htc\_handle, &cookie->HtcPkt);

}

/\* HTC API - HTCSendPkt \*/

A\_STATUS  **HTCSendPkt**(HTC\_HANDLE HTCHandle, HTC\_PACKET \*pPacket)

{

HTC\_PACKET\_QUEUE queue;

**INIT\_HTC\_PACKET\_QUEUE\_AND\_ADD**(&queue,pPacket);

return **HTCSendPktsMultiple**(HTCHandle, &queue);

}

A\_STATUS HTCSendPktsMultiple()

{

**HTCTrySend**(target,pEndpoint,pPktQueue);

}

# DEBUG

**export DEBUG\_EACS=1**

# WORD

ACS :Auto Channel Selection (ACS)

CW :Continuous Wave (CW) Interference is caused by Video Bridges (VB) and Baby Monitors. They  
use the Frequency Modulation technique to transmit signals in some of the ISM bands. They are  
persistent and completely block the WLAN channel in which they operate

Acronyms

Table 1-1 Acronyms Used in This Document

Acronym Definition

ACL Access Control List

ACS Auto Channel Selection

ADF Atheros Driver Framework

ASF Atheros Service Framework

DBDC Dual Band Dual Concurrent

DFS Dynamic Frequency Selection

EAP Extensible Authentication Protocol

HAL Hardware Abstraction Layer

HIF Host Interface

HT High Throughout

HTC Host Target Communications

LAN Local Area Network

LMAC Lower MAC

MAC Medium Access Control

P2P Peer to Peer

RIFS Reduced Inter Frame Spacing

STBC Space-Time Block Codes

TDLS Tunneled Direct Link Setup

TxBF Transmit Beamforming

UAPSD Unscheduled Automatic Power Save Delivery

UMAC Upper MAC

WDS Wireless Distribution System

WLAN Wireless LAN

WMI Wireless Message Interface

WMM Wi-Fi Multimedia

WMM-AC WMM admission control

WPA Wi-Fi Protected Access

# D

RX FIFIO depth 16

TX 128

ol\_ath\_vap\_set\_param

# HAND

IEEE80211\_IS\_CHAN\_11AC(\_c)