Fhostif 开发指南

一、虚拟网卡模式

1. 硬件设备连接说明

```
              USB/SDIO              WIFI            WIFI/网线
Linux-host1 <----------> AIC-MCU <-------> route <----------> Linux-host2
   ^ ^                     ^                                    ^
   | | <-----------------> |                                    |
   v         COM-DEBUG                                          v
   <-------------------------------------------------------------->
                           ping/tcp/udp
```

在此模式下，AIC-MCU相当于Linux-host1有线连接的以太网卡，实际在Linux-host1中访问到的设备是一个虚拟的以太网卡设备，可通过ifconfig查看

2. AIC-MCU低功耗-target描述

以 AIC8800MC 为例，在 config/aic8800mc 目录下存在多个 fhostif 相关子目录，

```
aic8800-sdk/config/aic8800mc/
├── target_wifi_fhostif              a---> wifi功能虚拟网卡
├── target_ble_wifi_fhostif          b---> 相较a，增加ble功能
├── target_wifi_fhostif_ramopt       c---> 相较a，将wifi协议栈运行于flash，提供更多内存
└── target_ble_wifi_fhostif_ramopt   d---> 相较b，将wifi协议栈运行于flash，提供更多内存
```

3. AIC-MCU低功耗-编译启动

AIC-MCU

AIC-MCU 代指 ***AIC8800x***，当前 ***AIC8800x*** 芯片系列包括 ***AIC8800M***、***AIC8800MC***、***AIC8800M40***

以 target_wifi_fhostif 为例，

1. 打开文件 **/config/aic8800x/target_wifi_fhostif/tgt_cfg/tgt_cfg_wifi.h**
2. 定义虚拟网卡模式

```
/**
 * Hostif mode selection, match with host driver
 *  Current support:
 *   1) HOST_VNET_MODE
 *   2) HOST_RAWDATA_MODE
 */
#define CONFIG_HOSTIF_MODE  HOST_VNET_MODE
```

3. 进入目录 **/config/aic8800x/target_wifi_fhostif**，若需要使用蓝牙功能，则进入 **/config/aic8800x/target_ble_wifi_fhostif**
4. 使用脚本 **build_fhostif_wifi_case.sh** 编译

```
# 编译usb接口版本，默认usb接口
./build_fhostif_wifi_case.sh HOSTIF=usb -j8
# 编译sdio接口版本
./build_fhostif_wifi_case.sh HOSTIF=sdio -j8
```

5. 将 **/build/host-wifi-aic8800/host_wb.bin** 烧写到 **x 8000000**

   （或者**/build/host-wifi-aic8800mc/host_wb_aic8800mc.bin** 烧写到 **x 8000000**）
6. 使用 **g 8000000** 执行MCU程序

Linux主机

*Linux*主机运行两部分程序，**虚拟网卡驱动 + 应用程序**

*Linux*主机加载驱动过程中将会注册网络设备，该网络设备的*mac*地址需与*MCU*保持一致。加载驱动时获取*mac*地址并直接注册网络设备，称为**非快速启动**；加载驱动时使用随机*mac*地址，加载完驱动才进行同步，称为**快速启动**。

a. 加载网卡驱动

1. 进入 **/wifi/LinuxDriver/aic8800_netdrv** 目录
2. 打开 **Makefile** 文件，根据主机的实际环境修改配置

```
# 主机平台默认为UBUNTU
CONFIG_PLATFORM_NANOPI_M4 ?= n
CONFIG_PLATFORM_ALLWINNER ?= n
CONFIG_PLATFORM_INGENIC_T31 ?= n
CONFIG_PLATFORM_INGENIC_T40 ?= n
CONFIG_PLATFORM_UBUNTU ?= y

# Driver mode support list
# 默认使用虚拟网卡模式
CONFIG_VNET_MODE ?= y
CONFIG_RAWDATA_MODE ?= n

# 默认关闭快速启动
CONFIG_FAST_INSMOD ?= n

# 接口默认使用USB
CONFIG_SDIO_SUPPORT = n
CONFIG_USB_SUPPORT = y
```

3. 编译加载驱动

```
make -j8
sudo insmod aic8800_netdrv.ko
ifconfig vnet0 up （T31/T40/T41需要）

# 安装驱动后，在Linux-host1终端中输入ifconfig，即可观察到已经注册了虚拟网卡vnet0
```

4. **驱动加载注意事项**

```
### sdio
加载sdio驱动之前必须先完成sdio识卡过程
1. sdio识卡（clk、cmd、GND）
   AIC8800x连接主控sdio，主控内核加载后将会启动识卡过程。识卡过程跟驱动无关，若sdio接口使用飞线或者卡槽连接，可能由于接线长度不等，导致识卡失败。调试过程中可以在clk线上接一个小电容并连接到GND，电容大小需实际测试。若识卡成功，AIC8800x将从主控分配得到一个sdio设备地址
2. 驱动加载
   同样，若是采用飞线或者卡槽连接，sdio时钟不能设置过高，设置过高很容易出现sdio数据包crc-err，推荐卡槽20M左右，飞线10M以内。此外，降低时钟后加载sdio驱动还有可能出现主控发包成功，收包失败的现象，此时调整sdio相位即可


### 修改说明
// V2: AIC8800M/MC, V3: AIC8800M40

// V2 直接修改以下宏即可修改时钟与相位
# aicwf_sdio.h 文件
#define SDIOWIFI_CLOCK_V2               20000000 // default 20MHz
#define SDIOWIFI_CLOCK_V3               20000000 // default 20MHz
#define SDIOWIFI_PHASE_V2               0        // 0: default, 2: 180°

// V3 需要打开以下代码
# 在 aicwf_sdiov3_func_init 接口中
    u8 val = 0;  // 去掉注释
    #if 1        // 打开条件编译
    if (host->ios.timing == MMC_TIMING_UHS_DDR50) {
        val = 0x21;//0x1D;//0x5;
    } else {
        val = 0x01;//0x19;//0x1;
    }
    val |= SDIOCLK_FREE_RUNNING_BIT;
    sdio_f0_writeb(sdiodev->func, val, 0xF0, &ret);
    if (ret) {
        sdio_err("set iopad ctrl fail %d\n", ret);
        sdio_release_host(sdiodev->func);
        return ret;
    }
    sdio_f0_writeb(sdiodev->func, 0x0, 0xF8, &ret);
    if (ret) {
        sdio_err("set iopad delay2 fail %d\n", ret);
        sdio_release_host(sdiodev->func);
        return ret;
    }
    # 修改寄存器0xF1值，可设置0x10/0x20/0x40/0x80/0xF0五个相位
    sdio_f0_writeb(sdiodev->func, 0x20, 0xF1, &ret);
    if (ret) {
        sdio_err("set iopad delay1 fail %d\n", ret);
```

```
        sdio_release_host(sdiodev->func);
        return ret;
    }
    msleep(1);
    #if 1//SDIO CLOCK SETTING
    if (host->ios.timing != MMC_TIMING_UHS_DDR50) {
        host->ios.clock = SDIOWIFI_CLOCK_V3;  # aicwf_sdio.h中修改
        host->ops->set_ios(host, &host->ios);
    }
    #endif
    #endif
```

b. 启动虚拟网卡

非快速启动

1. 进入 **/wifi/LinuxDriver/app/custom_msg** 目录直接编译

   (若嵌入式平台，则需使用相应的交叉编译工具进行编译)

2. 连接路由器 AP

```
./custom_msg vnet0 1 ssid password
例如，
./custom_msg vnet0 1 XIAOMI_HHH 1234567
```

3. 通过串口观察 **MCU-DEBUG** 信息，接受连接指令后，将连接上路由，并分配得到 **mcu-ip** + **mcu-gw** 通过MCU的串口即可观察到
   以下设置均跟 **mcu-ip** + **mcu-gw** 相关

4. 设置虚拟网卡 ip 与 gw

```
sudo ifconfig vnet0 <mcu-ip> (ie: 192.168.3.36)
sudo route add default gw <mcu-gw> (ie: 192.168.3.1)
```

5. 设置网络 DNS

```
# 打开resolv.conf文件
sudo chmod 777 /etc/resolv.conf
vim /etc/resolv.conf

# 添加如下内容，注意，Linux-host1系统重启之后该文件会复原
namespace mcu-gw (ie: 192.168.3.1)
```

6. 测试网络联通

```
# 实际中，依据不同平台，若通过步骤4即可ping通网络，则步骤5无需操作

# 路由
ping 192.168.12.1

# 局域网
ping 192.168.12.10

# 外网
ping 202.108.22.5
```

快速启动

再次说明：指系统加载驱动过程中，不获取*MCU*返回的*mac*地址，使用软件随机*mac*地址注册*vnet_dev*，驱动加载完成之后，再利用应用程序获取*MCU*的*mac*地址，并同步到*vnet_dev*网络设备

1. 修改 **Makefile** 设置

```
# 默认关闭快速启动
CONFIG_FAST_INSMOD ?= y
```

2. 加载完驱动后，同步MCU的mac地址

```
# 获取MCU的mac地址，通过dmesg查看信息
custom_msg vnet0 5

# 同步vnet_dev的mac地址
sudo custom_msg vnet0 ndev mac_address
ie: sudo custom_msg vnet0 ndev 00:11:22:33:44:55

# 或者，使用ifconfig命令修改mac地址
sudo ifconfig vnet0 down
sudo ifconfig vnet0 hw ether 00:11:22:33:44:55
sudo ifconfig vnet0 up
```

3. 其余操作同 **非快速启动**

c. 应用程序

在 **/wifi/LinuxDriver/app/** 目录下，有 **custom_msg**、**fasync_demo** 两个应用程序

custom_msg

用于往主控驱动中下发指令，并通过驱动中的SDIO/USB接口发送到 AIC-MCU，custom_msg 与 AIC-MCU 之间的交互指令如下，

```
# 编译完成应用程序之后，输入./custom_msg，即可看到指令说明
"usage: custom_msg vnet0 [mode] <arg1> <arg2> <arg3>"
"----------------------"
">>>Interact with MCU:"
"custom_msg vnet0 1 ssid password          - connect ap"
"custom_msg vnet0 2                        - disconnect ap"
"custom_msg vnet0 3                        - close ble before sleep if used\r\n"
"custom_msg vnet0 4                        - enter sleep"
"custom_msg vnet0 5                        - exit sleep"
"custom_msg vnet0 6                        - get mcu mac"
"custom_msg vnet0 7                        - get wlan status"
```

```
"custom_msg vnet0 8 ssid password band     - start ap"
"                   -- band = <2.4G/5G>"
"custom_msg vnet0 9                        - change ap mode\r\n"
"custom_msg vnet0 10                       - stop ap"
"custom_msg vnet0 11                       - scan wifi"
"custom_msg vnet0 12 /your-path/update.bin - host ctrl OTA"
"------------------------"
">>>Interact with kernel:"
"sudo custom_msg vnet0 ndev mac_address    - set vnet_dev mac"
```

注意要点

1. **消息阻塞** 驱动可以针对 **custom_msg** 消息 id 进行阻塞与否的配置

```
// handle_custom_msg 函数
case APP_CMD_CONNECT:
    /.../
    // 消息阻塞 AICWF_CMD_WAITCFM
    cust_app_cmd.cmd_cfm.waitcfm = AICWF_CMD_WAITCFM;
    // 等待回应 CUST_CMD_CONNECT_IND
    cust_app_cmd.cmd_cfm.cfm_id  = CUST_CMD_CONNECT_IND;
    ret = rwnx_tx_msg((u8 *)&cust_app_cmd.connect_req,
sizeof(cust_app_cmd.connect_req), &cust_app_cmd.cmd_cfm, command);
    break;

case APP_CMD_SCAN_WIFI:
    printk("APP_CMD_SCAN_WIFI\n");
    cust_app_cmd.common_req.cmd_id = CUST_CMD_SCAN_WIFI_REQ;
    // 不阻塞
    cust_app_cmd.cmd_cfm.waitcfm = AICWF_CMD_NOWAITCFM;
    // 不等待回应
    cust_app_cmd.cmd_cfm.cfm_id  = 0;
    ret = rwnx_tx_msg((u8 *)&cust_app_cmd.common_req,
sizeof(cust_app_cmd.common_req), &cust_app_cmd.cmd_cfm, command);
    break;
```

2. **超时时间** 驱动中对 **custom_msg** 消息阻塞时间做了设置，用户可自行更改

```
// rwnx_main.h 注意，若需返回联网结果，超时时间应设置大于联网所需时间
#define AICWF_CMDCFM_TIMEOUT      2000
```

3. **结果回显** 对于阻塞的 **custom_msg** 消息，返回时将带有反馈消息，若不需要直接注释即可

```
// custom_msg.c
// If don`t need return-result, it`s OK to comment out this printf.
printf(APP_NAME "%s\n", (char *)&priv_cmd.buf[0]);
```

fasync_demo

利用字符设备的异步消息机制，实现驱动主动向应用层发送消息的功能。**fasync** 功能默认关闭，启动该功能，需要修改驱动 Makefile ，之后在驱动加载过程中将会注册用于消息发送的字符设备，加载完驱动，需将 **fasync_user** 运行在后台实时接收消息

```
# Msg Callback setting  默认为n
CONFIG_APP_FASYNC ?= y
```

注意要点

1. **后台运行** **fasync_user** 应该在 **aic8800_netdrv.ko** 驱动加载后，**custom_msg** 下发指令之前启动，并且一直保持在后台运行
2. **消息回写** 在驱动连接多次发送消息的情况下，应用层来不及处理消息，可能导致出现消息遗漏的问题。为此，字符设备消息的发送过程，要求应用层收到消息之后，进行状态位的设置，之后驱动才能够进行下一次消息的发送。用户在移植时应注意。

```
static void signal_handler(int signum)
{
    int ret = 0;
    char data_buf[sizeof(struct rwnx_fasync_info)];
    struct rwnx_fasync_info *fsy_info = (struct rwnx_fasync_info *)data_buf;
    ret = read(fd, data_buf, sizeof(struct rwnx_fasync_info));
    if(ret < 0) {
        printf(APP_NAME "Read kernel-data fail\n");
    } else {
        printf(APP_NAME "%s\n", fsy_info->mem);
        // 设置状态 0，表示已经收到了消息
        fsy_info->mem_status = 0;
        ret = write(fd, &fsy_info->mem_status, sizeof(fsy_info-
>mem_status));
        // 回写设备 buff
        if (ret < 0)
            printf(APP_NAME "Write kernel-data fail\n");
        // 解析内核消息
        analy_signal_msg(fsy_info->mem);
    }
}
```

若没有启动 **fasync_user** 程序，或者该程序中没有设置 **mem_status**，则驱动往应用层发送消息将会等待超时，才能发送下一条消息。

```
# rwnx_main.h 可调整超时时间
#define FASYNC_APP_TIMEOUT          100
```

3. **自动配网** **fasync_user** 解析到联网信息时，将自动提取 **ip** 和 **gw**，通过 **system** 接口自动配置主控网络状态，用户可针对具体需求解析消息并作相应配置

```
void analy_signal_msg(char *mem)
{
    char buff[64];
    struct wlan_settings wlan;
    // auto wlan-settings
    char *ptr = strstr(mem, "3003");
    if (ptr) {
        // set vnet0 ip
        ptr = strstr(mem, "ip");
        memcpy(buff, ptr+4, WLAN_SET_LEN);
        wlan.ip = str2uint(buff);
        sprintf(buff, "ifconfig vnet0 %d.%d.%d.%d",
```

```
                (unsigned int)((wlan.ip >> 0 )&0xFF), (unsigned int)
((wlan.ip >> 8 )&0xFF),
                (unsigned int)((wlan.ip >> 16)&0xFF), (unsigned int)
((wlan.ip >> 24)&0xFF));
        system(buff);

        // set vnet0 gw
        memcpy(buff, ptr+18, WLAN_SET_LEN);
        wlan.gw = str2uint(buff);
        sprintf(buff, "route add default gw %d.%d.%d.%d",
                (unsigned int)((wlan.gw >> 0 )&0xFF), (unsigned int)
((wlan.gw >> 8 )&0xFF),
                (unsigned int)((wlan.gw >> 16)&0xFF), (unsigned int)
((wlan.gw >> 24)&0xFF));
        system(buff);
        system("ifconfig");
    }
}
```

4. **设备号冲突** 若因为开启了CONFIG_APP_FASYNC，导致加载驱动失败，可以先确认下是否因为字符设备号冲突了，按照如下修改即可

```
# 相关指令
ls -l /dev          查看设备文件号
cat /proc/devices 查看设备号
ls /sys/class       查看字符设备class
```

```
int rwnx_aic_cdev_driver_init(void)
{
    int ret = 0;
    struct device *devices;
    if (alloc_chrdev_region(&chardev.dev, 0, 1, "aic_cdev_ioctl")) {
        printk("%s: alloc_chrdev_region failure\n", __FUNCTION__);
        goto exit;
    }
    chardev.major = MAJOR(chardev.dev);

    // add cdev
    chardev.c_cdev = kzalloc(sizeof(struct cdev), GFP_KERNEL);
    if(IS_ERR(chardev.c_cdev)) {
        printk("%s: kmalloc failure\n", __FUNCTION__);
        ret = PTR_ERR(chardev.c_cdev);
        goto free_chrdev_region;
    }
    cdev_init(chardev.c_cdev, &aic_cdev_driver_fops);
    ret = cdev_add(chardev.c_cdev, chardev.dev, 1);
    if(ret < 0) {
        printk("%s: cdev_add failure\n", __FUNCTION__);
        goto free_chrdev_region;
    }

    // create device_class
    chardev.cdev_class = class_create(THIS_MODULE, "aic_cdev_class");
    if(IS_ERR(chardev.cdev_class)) {
        printk("%s: class_create failure\n", __FUNCTION__);
        ret = PTR_ERR(chardev.cdev_class);
        goto free_cdev;
    }

    // create device
    devices = device_create(chardev.cdev_class, NULL, MKDEV(chardev.major,0), NULL, "aic_cdev");
    if(IS_ERR(devices)) {
        printk("%s: device_create failure\n", __FUNCTION__);
        ret = PTR_ERR(devices);
        goto free_device_class;
    }

    printk("Create device: /dev/aic_cdev_class\n");
    return 0;
```

4. AIC-MCU低功耗-验证流程

1. Linux-Host1 在没有执行任务时自动关闭，并在需要执行任务时进行唤醒，恢复工作状态
2. 在AIC-MCU端进入休眠之后，可通过对EVB板的电流进行实时监控，测量应用中AIC-MCU休眠功耗
3. 用于测量功耗的EVB板必须**断开外部器件**，使用vbat供电，测量核心供电以免功耗数据偏高

验证流程可参考如下

请结合*SDK*中，***aic8800_netdrv***驱动与 ***fhostif_cmd.c***应用代码进行分析

1. 上电运行 AIC-MCU 与 Linux-Host1，其中 AIC-MCU 可切换成正常模式，上电将自动执行
2. Linux-Host1 加载驱动

```
# 主控操作
sudo insmod aic8800_netdrv.ko
```

3. AIC-MCU 配网，当前支持三种模式，

   1）通过 **AIC-MCU 串口**交互指令进行配网

   2）通过 Linux 端**应用层交互程序 wifi/LinuxDriver/app/custom_msg** 向驱动发送指令，控制 MCU 配网

```
# 串口指令
connect 0 ssid password
# 主控指令
./custom_msg vnet0 1 ssid password
```

   3）通过蓝牙配网（编译带有蓝牙的target）

```
# fhostif_example.c do_blewifi_config 蓝牙配网例程
1. 通过串口启动，ble_wifi_cfg
2. 修改例程，在适当时机启动配网
```

4. Linux-Host1 获取 MCU 配网信息并配置网络，执行相关任务
5. Linux-Host1 空闲，通过应用程序下发指令告知MCU进入休眠，同时主控驱动将停止接收上层应用数据包

```
# 关闭蓝牙，蓝牙功能会影响wifi休眠
如果编译了蓝牙功能，进入休眠之前必须先关闭蓝牙
./custom_msg vnet0 3
# 主控指令
./custom_msg vnet0 4
```

6. Linux-Host1 卸载驱动，并掉电关闭

```
# 主控操作
sudo rmmod aic8800_netdrv
```

7. AIC-MCU 收到进入休眠指令，将设置低功耗唤醒源

```
# MCU操作
# 体现于 custom_msg_enter_sleep_handler
sleep_level_set(...);          // 设置休眠等级
user_sleep_wakesrc_set(...);   // 设置唤醒源
user_sleep_allow(1);           // 允许休眠
```

8. AIC-MCU 将执行 **host_if_poweroff** 函数，关闭 SDIO/USB 接口

```
host_if_poweroff不能在custom_msg_enter_sleep_handler中调用，这将可能导致buff出问题
例程中，host_if_poweroff在timer_handler中被调用

# 串口指令
# 注意：测试过程中可在MCU串口上关闭接口，但这种方式无法停止主控驱动继续往MCU接口
(SDIO/USB)发送数据包
hpof
```

9. AIC-MCU 进入低功耗，并进行其他相关保活操作

```
# 低功耗状态下应用开发说明
1. MCU休眠一段时间后，定时唤醒执行相关任务
2. 若有用户应用需要执行，MCU将在执行完相关任务才进入休眠
```

10. AIC-MCU 被相关事件唤醒，退出低功耗，并通过指定 IO 给主控上电

```
# MCU操作
user_sleep_allow(0);                 // 不允许休眠
sleep_level_set(PM_LEVEL_ACTIVE);    // 设置活跃等级
# IO上电，demo中未体现，客户自行添加
```

11. AIC-MCU 恢复与主控接口，执行 **host_if_repower** 函数，开启 SDIO/USB 接口

```
# 串口指令
hpon
```

12. Linux-Host1 上电启动，重新加载驱动

```
# 主控操作
sudo insmod aic8800_netdrv
```

13. Linux-Host1 获取 AIC-MCU 配网信息并配置网络，重新执行任务

---

5. AIC-MCU低功耗-状态说明

依据*AIC-MCU*是否连接主控，*msg*是否连通，*data*是否连通，定义*AIC-MCU*与主控接口的四种状态

```
# 定义 AIC-MCU 与 Linux-Host1 接口的状态
typedef enum hostif_status
{
    HOSTIF_ST_INIT      = 0, // host power off ->  null  msg, null  data
    HOSTIF_ST_IDLE      = 1, // host power on  ->  tx/rx msg, null  data
    HOSTIF_ST_AWAKE     = 2, // host power on  ->  tx/rx msg, tx/rx data
    HOSTIF_ST_DEEPSLEEP = 3, // host power off ->  null  msg, null  data
} hostif_status_e;
```

状态	描述
HOSTIF_ST_INIT	MCU-AIC刚上电，未连接主控，data与msg均不通
HOSTIF_ST_IDLE	连接主控，未配网，只有msg通
	set_hostif_wlan_status(HOSTIF_ST_IDLE);
HOSTIF_ST_AWAKE	连接主控，已配网，data与msg均通
	set_hostif_wlan_status(HOSTIF_ST_AWAKE);
HOSTIF_ST_DEEPSLEEP	未连接主控，已配网，data与msg均不通
	set_hostif_wlan_status(HOSTIF_ST_DEEPSLEEP);

STA状态转移

```
# INIT  -> HOSTIF_ST_INIT
# IDLE  -> HOSTIF_ST_IDLE
# AWAKE -> HOSTIF_ST_AWAKE
# SLEEP -> HOSTIF_ST_DEEPSLEEP

# MCU-AIC状态转移1: 不启动编译softAP功能
                  ┌ ----------3----------- ┐  --> Forbid this change!!
      ┌ --------1------------ ┐            |
      |           |           V            V
    INIT --1--> IDLE --2--> AWAKE --3--> SLEEP
                  ^           |  ^           |
                  └ ---2---- ┘  └ ---1---- ┤
                  ^                        |
                  └ ----------1----------- ┘

# 转移条件说明
1. custom_msg_get_wlan_status_handler
主控加载驱动时必须调用该handler，以获取AIC-MCU配网状态

2. custom_msg_connect_status_ind_handler
   以及custom_msg_disconnect_status_ind_handler
在连接主控之后，AIC-MCU连网与断网将会调用这两个handler

3. custom_msg_enter_sleep_handler
主控下发休眠指令后，调用该handler进入休眠
```

AP状态转移

fhostif 提供两种模式的 AP

模式	描述	HOSTIF_ST
AIC_AP_MODE_CONFIG	softAP 数据流只在 MCU 本地处理，用于AP配网	HOSTIF_ST_IDLE
AIC_AP_MODE_DIRECT	softAP 数据流与主控联通，可用于数据收发	HOSTIF_ST_AWAKE

```
# INIT  -> HOSTIF_ST_INIT
# IDLE  -> HOSTIF_ST_IDLE
# AWAKE -> HOSTIF_ST_AWAKE
# SLEEP -> HOSTIF_ST_DEEPSLEEP

# MCU-AIC状态转移2: 启动编译softAP功能-AIC_AP_MODE_CONFIG
                 ┌ -----------3----------- ┐  --> Forbid this change!!
      ┌ ---1---- | ---------- ┐            |
      |       ┌ 4/5 ┐         |            |
      |       |  |  V         V            V
    INIT --1--> IDLE --2--> AWAKE --3--> SLEEP
                ^ ^         | | ^          |
                | └ ---2--- ┘ | └ ---1---- ┤
                | ^           |            |
                └ -----4----- ┘            |
                  └ ----------1---------- ┘
# 转移条件说明
1~3 同上
4. custom_msg_start_ap_handler
主控下发启动AP指令，如果此刻AIC-MCU正连接路由器，将会先断开路由器，返回IDLE状态

5. custom_msg_stop_ap_handler
主控下发停止AP指令

# MCU-AIC状态转移3: 启动编译softAP功能-AIC_AP_MODE_DIRECT
                 ┌ -------------3----------- ┐  --> Forbid this change!!
      ┌ ---1---- | ------------ ┐            |
      |       ┌ 4/5 ┐           |            |
      |       |  |  V           V            V
    INIT --1--> IDLE --2/6--> AWAKE --3--> SLEEP
                ^ ^           | | ^          |
                | └ ---2/6--- ┘ | └ ---1---- ┤
                | ^             |            |
                └ ------4------ ┘            |
                  └ ------------1---------- ┘
# 转移条件说明
1~3 同上
4. custom_msg_start_ap_handler
主控下发启动AP指令，如果此刻AIC-MCU正连接路由器，将会先断开路由器，返回IDLE状态

5. custom_msg_stop_ap_handler
主控下发停止AP指令

6. custom_msg_change_ap_mode_handler
主控下发切换AP模式指令
```

数据解析基本过程

结合上述AIC-MCU低功耗-验证流程，不考虑被过滤数据包，一般数据包解析过程

```
                          主控下发指令
|-主控,MCU初次上电-| <-------------> |-主控掉电,MCU进行休眠-| -->> 主控解析 WIFI 数据包
                          MCU被事件唤醒
|---MCU进入休眠---| <--------------> |--MCU唤醒退出低功耗--| -->> MCU解析 WIFI 数据包
                           主控下发指令
|---主控重新上电---| <--------------> |-主控掉电,MCU进行休眠-| -->> 主控解析 WIFI 数据包
```

以上过程持续循环，直到系统关闭，详细可见demo中对 **set_hostif_wlan_status** 接口调用设置

---

6. AIC-MCU低功耗-数据过滤

fhostif 提供两种数据包过滤模式，*VNET_FILTER_DIRECT* 和 *VNET_FILTER_SHARED*，如上*AIC-MCU*低功耗-状态说明，*HOSTIF_ST_STATUS* 设置为 *HOSTIF_ST_AWAKE*，*AIC-MCU*才会将数据包透传给主控解析，设置过滤规则可以将满足条件的数据包保留在 *AIC-MCU* 中进行处理

设置过滤模式

```
// 在每个 fhostif-target 目录下均有文件 target_xxx_fhostif/tgt_cfg/tgt_cfg_wifi.h
/**
 * Hostif rx pkt filter mode
 *  Current suppoer:
 *   1) VNET_FILTER_DIRECT
 *   2) VNET_FILTER_SHARED
 */
#define CONFIG_HOSTIF_FILTER_MODE VNET_FILTER_DIRECT
```

VNET_FILTER_DIRECT

这种模式只根据数据包中的 dst_port 和 protocol 进行判断，区分数据包发往主控，还是本地处理

```
// set ip pkt filter
filter.used = 1;
filter.protocol = 17;  // UDP
filter.dst_port = 68;  // DHCP client
ret = set_hostif_user_filter(&filter);
if (ret) {
    dbg("failed to set ip pkt filter\n");
}
```

VNET_FILTER_SHARED

这种模式提供了多种的过滤规则，在设置过滤器时，通过**filter_mask**设置规则

过滤规则	mask
src_ipaddr	PACKET_FILTER_MASK_SRC_IP
src_ipaddr && src_port	PACKET_FILTER_MASK_SRC_IP | PACKET_FILTER_MASK_SRC_PORT
protocol && src_port	PACKET_FILTER_MASK_PROTOCOL | PACKET_FILTER_MASK_SRC_PORT
protocol && dst_port	PACKET_FILTER_MASK_PROTOCOL | PACKET_FILTER_MASK_DST_PORT
src_ipaddr, protocol && src_port	PACKET_FILTER_MASK_SRC_IP | PACKET_FILTER_MASK_PROTOCOL | PACKET_FILTER_MASK_SRC_PORT

```
// set ip pkt filter
filter.used = 1;
filter.protocol = 17;  // UDP
filter.dst_port = 68;  // DHCP client
filter.filter_mask = PACKET_FILTER_MASK_PROTOCOL | PACKET_FILTER_MASK_DST_PORT;
ret = set_hostif_user_filter(&filter);
if (ret) {
    dbg("failed to set ip pkt filter\n");
}
```

此外，这种模式针对 **ping包** 过了共享处理，当 AIC-MCU 联网之后，无论主控还是MCU都可以 ping 通外部网络，而外部网络发起 ping 请求时，将由 AIC-MCU 进行回应

---

7. AIC-MCU低功耗-休眠说明

AIC-MCU的低功耗休眠有不同的等级可供选择（常用等级）

芯片型号	休眠等级
AIC8800M40	PM_LEVEL_ACTIVE（不休眠）
	PM_LEVEL_LIGHT_SLEEP
	PM_LEVEL_DEEP_SLEEP
AIC8800MC	PM_LEVEL_ACTIVE（不休眠）
	PM_LEVEL_LIGHT_SLEEP
	PM_LEVEL_DEEP_SLEEP
AIC8800M	PM_LEVEL_ACTIVE（不休眠）
	PM_LEVEL_LIGHT_SLEEP
	PM_LEVEL_DEEP_SLEEP
	PM_LEVEL_HIBERNATE

支持唤醒IO请查看《AIC8800低功耗》文档

---

8. AIC-MCU低功耗-KeepAlive

在主控加载驱动之后，AIC-MCU 将与主控之间的接口状态设置为 **HOSTIF_ST_IDLE**，进一步依据 <5. AIC-MCU低功耗-状态说明> 更改接口状态。然而当主控 SDIO/USB 接口发生异常中断时，通常在这种意外情况下，AIC-MCU 无法自动识别，只能依靠看门狗进行复位。

为此，在主控与 AIC-MCU 之间增加 **Keep-Alive** 机制，主控定时发送指定的 msg 数据包，AIC-MCU 收到之后进行回复并更新心跳时间，同时 AIC-MCU 会启动定时器定时检查心跳时间是否及时进行更新，否则认为与主控断开了连接，之后将设置接口状态为 **HOSTIF_ST_INIT**

```
# fhostif_cmd.c

1. keep_alive 默认关闭
#define HOSTIF_KEEP_ALIVE 0

2. 心跳包时间设置（ms）
#define KEEP_ALIVE_PERIOD 250
```

---

9. AIC-MCU低功耗-主控DHCP

通常情况下，AIC-MCU 连接AP之后会自动进行DHCP获取IP，若需要由主控进行DHCP过程，则需要先给AIC-MCU配置虚拟的IP信息，等待主控获取到IP之后，主控再下发给AIC-MCU同步IP信息。

```
# fhostif_cmd.c

主控DHCP 默认关闭
#define HOSTIF_CNTRL_DHCP 0
```

10. AIC-MCU低功耗-OTA自动重启

通常情况下，AIC-MCU OTA升级之后不会自动重启，重启之后才会运行新的软件版本。

```
# fhostif_cmd.c

OTA自动重启 默认关闭
#define HOST_OTA_REBOOT 0
```

注意事项

1. 系统正常关闭流程，

   ```
   # a. Linux-host1 卸载驱动
   sudo rmmod aic8800_netdrv
   # b. Linux-host1 断电
   # c. MCU 断电
   ```

2. 系统正常运行之后，确保 **AIC-MCU** 供电稳定，若中途 **AIC-MCU** 意外断电，将导致 **SDIO/USB** 连接异常， **Linux-host1** 可能无法正常卸载驱动，可能需要强制关机

二、rawdata模式

硬件设备连接说明

```
              USB/SDIO              WIFI            WIFI/网线
Linux-host1 <----------> AIC-MCU <-------> route <----------> Linux-host2
     ^                    ^ ^                                    ^
     |      COM-DEBUG     | |              ping/tcp/udp           |

     v <-----------------> v v <---------------------------------> v
```

在此模式下，AIC-MCU相当于Linux-host1的USB/SDIO设备，可用于大量数据传输。

编译与启动

AIC-MCU

1. 更改**tgt_cfg_wifi.h**模式配置

```
/**
 * Hostif mode selection, match with host driver
 *  Current support:
 *   1) HOST_VNET_MODE
 *   2) HOST_RAWDATA_MODE
 */
#define CONFIG_HOSTIF_MODE  HOST_RAWDATA_MODE
```

2. 其余步骤同上述虚拟网卡

Linux主机

a. Linux驱动

1. 进入 **/wifi/LinuxDriver/aic8800_netdrv** 目录
2. 配置 **Makefile** 相关宏

```
# Driver mode support list
CONFIG_VNET_MODE ?= n
CONFIG_RAWDATA_MODE ?= y
```

3. 编译加载驱动

```
make -j8
sudo insmod aic8800_netdrv.ko
```

b. 应用程序

1. 进入 **/wifi/LinuxDriver/app/nlaic_demo** 目录
2. 编译运行

```
make -j8
# 运行应用程序必须添加权限
# demo-1
sudo ./nlaic_user 0 <char msg> <uint data> <bytes byte_ptr>
# demo-2
sudo ./nlaic_user 1
# demo-3
sudo ./nlaic_user 2 <uint block_size> <uint block_count>
```