RALINK TECHNOLOGY, CORP.

RALINK WIRELESS DEVICE DRIVER PORTING GUIDE

Proprietary Notice and Liability Disclaimer

The confidential Information, technology or any Intellectual Property embodied therein, including without limitation, specifications, product features, data, source code, object code, computer programs, drawings, schematics, know-how, notes, models, reports, contracts, schedules and samples, constitute the Proprietary Information of Ralink (hereinafter "Proprietary Information")

All the Proprietary Information is provided "AS IS". No Warranty of any kind, whether express or implied, is given hereunder with regards to any Proprietary Information or the use, performance or function thereof. Ralink hereby disclaims any warranties, including but not limited warranties of non-infringement, merchantability, completeness, accuracy, fitness for any particular purpose, functionality and any warranty related to course of performance or dealing of Proprietary Information. In no event shall Ralink be liable for any special, indirect or consequential damages associated with or arising from use of the Proprietary Information in any way, including any loss of use, data or profits.

Ralink retains all right, title or interest in any Proprietary Information or any Intellectual Property embodied therein. The Proprietary Information shall not in whole or in part be reversed, decompiled or disassembled, nor reproduced or sublicensed or disclosed to any third party without Ralink's prior written consent.

Ralink reserves the right, at its own discretion, to update or revise the Proprietary Information from time to time, of which Ralink is not obligated to inform or send notice. Please check back if you have any question. Information or items marked as "not yet supported" shall not be relied on, nor taken as any warranty or permission of use.

Ralink Technology Corporation (Taiwan)

5F, No.36, Tai-Yuen Street,

ChuPei City

HsinChu Hsien 302, Taiwan, ROC

Tel +886-3-560-0868

Fax +886-3-560-0818

Sales Taiwan: Sales@ralinktech.com.tw

Technical Support Taiwan: FAE@ralinktech.com.tw

http://www.ralinktech.com/

Revision History

Date	Revision	Author	Description
05/07/2009	Ver. 0.10	Shiang TU	Initial draft for Ralink WIFI Device Driver Porting guide. (Based on RT_WIFI V2.2.0.0)
05/15/2009	Ver. 0.11	Jay Hung	Add USB interface porting guide

Contents

1 INTRODUCTION

The Ralink WIFI Device Driver (RT_WIFI) is a proprietary implementation that drives Ralink's IEEE 802.11n based chipsets. It offers a rich feature set as specified in IEEE 802.11 series standard specifications, depending on the RT_WIFI and the capabilities of the specified chipset.

This document describes the basic design, architecture, and porting guide of RT_WIFI. Specified code patches for different chipsets or modules are not included here. They are described in other related design specifications or release notes.

1.1 System Architecture

The design goal of RT_WIFI is to implement a multiple-OS (Operating System), multiple-chipset compatible device driver. To achieve this requirement, RT_WIFI is implemented as a layered and modulated structure. RT_WIFI has three separate layers depending on the relationship between the OS, the hardware and the device driver. Each layer can be divided into several sub-systems depending on the functions, protocols, and hardware/software attributions. Figure 1-1 shows the RT_WIFI architecture, components and the relationship with the operating system.

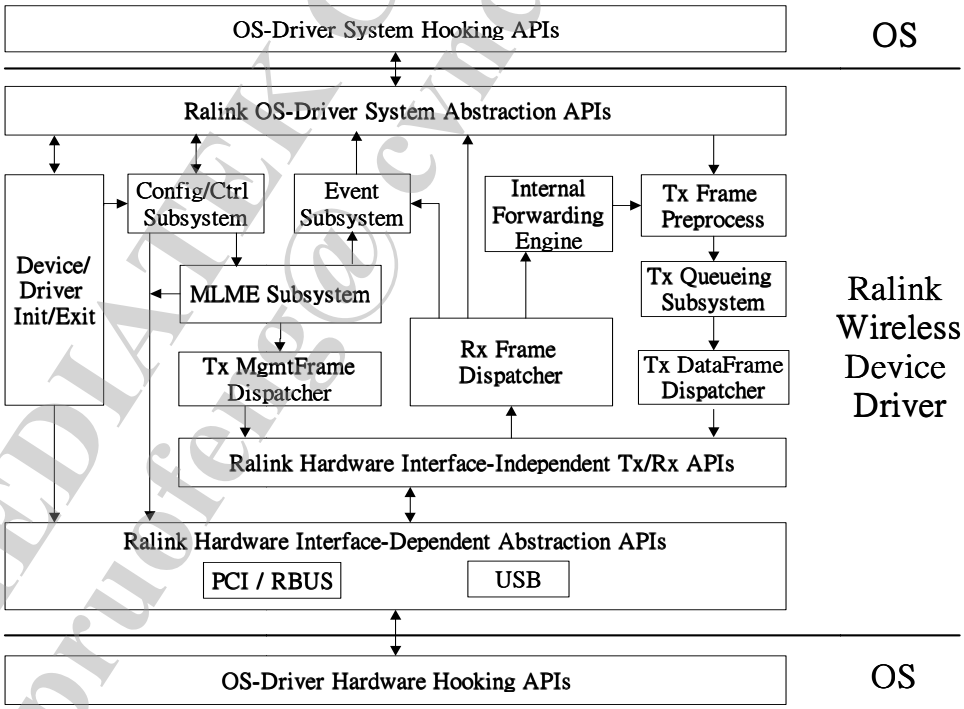

Figure 1-1 Ralink Wireless Device Driver Architecture

The classification of layers depends on three types. There are OS dependent/independent functions, IEEE 802.11 protocol stack functions, and hardware related functions. The three layers are called the OS-Driver System Layer (ODSL), the IEEE 802.11 Protocol Stack Layer (PSL), and the OS-Driver Hardware Layer (ODHL). It is easy to port to different operating systems because of the layered design, and the lower cost of maintenance. A short introduction for each layer is given in the subsequent sections.

1.1.1 OS-Driver System Layer (ODSL)

The main functions of a network device driver are usually packet transmission/reception, protocol stack management, and configuration/control of the device. Each of these functions must have the cooperation of OS system calls, the device driver, and the hardware device. There are various operating systems on various network capable devices. The operation performance between the OS and network device drivers is still equivalent. Based on this common operation model, a set of OS abstraction APIs (named as ODSL APIs) which work as wrapper functions can be used to connect the various OS-specified hooking functions and the corresponding RT_WIFI implementation which are in line with the requirements.

As a result of the ODSL APIs, for each specified OS, only those wrapper APIs need to be implemented to satisfy the requirements of the OS. They don't need to be rewritten to other parts of the device driver.

The ODSL layer is made of the OS-Driver System Hooking APIs, the Ralink OS-Driver System Abstraction APIs, and part of the OS-Driver Hardware Hooking APIs.

1.1.2 IEEE 802.11 Protocol Stack Layer (PSL)

The IEEE 802.11 Protocol Stack Layer is a pure, stand-alone layer. It is independent of the OS and it mainly focuses on the basic 802.11 MAC functions (i.e. MLME, Security, WMM, Frame encapsulation/parsing, Fragmentation/Defragmentation, etc.) It is also independent on other 802.11 standard specification related features (i.e. WPS, WMM-ACM, WDS, Roaming, etc.) In most cases, the functions in this part should all be the same in each OS. There is no need for any modifications.

One exception is the need to use a different IEEE 802.11 protocol stack instead of the built-in RT_WIFI protocol stack. In this case, personalized functions must be used to go around the built-in protocol stack and redirect the related information to the different protocol stacks. The use of this application is not in the scope of this document.

1.1.3 OS-Driver Hardware Layer (ODHL)

The network device usually connects to the main board/CPU through a specified hardware interface (i.e. the PCI interface, the USB interface, or Ralink's proprietary internal bus) to accomplish the data communication process. Chipsets may be designed with different hardware interfaces to satisfy various applications or equipment. Because of this, a multiple-chipset compatible device driver is the other design goal of RT_WIFI. RT_WIFI creates another abstraction layer called OS-Driver Hardware Layer (ODHL). This layer isolates different hardware interfaces and helps with the integration of software and hardware, to support variant interfaces.

The ODHL layer includes the Ralink Hardware Interface-Independent Tx/Rx APIs, Hardware Interface-Dependent Tx/Rx Abstraction APIs, and part of the OS-Driver Hardware Hooking APIs.

1.2 Flow Chart Notation

The symbols in Figure 1-2 are used to show the functions, decisions, conditions, states, data path, etc. in each flowchart. The flowcharts show the function or data processing procedures in this document.

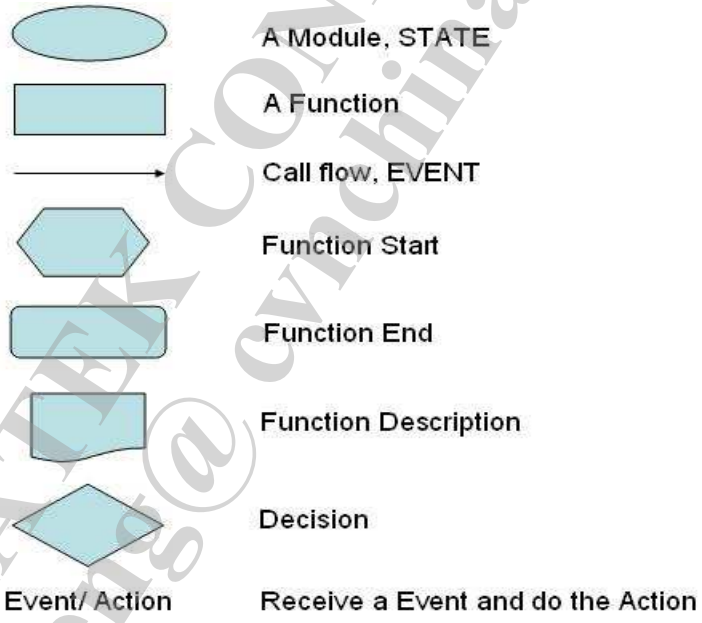

Figure 1-2 Flowchart notation

The following names or acronyms are used throughout this document: AP (WLAN Access Point), STA (WLAN Station), FSM (Finite State Machine), API (Application Programming Interface), IOCTL (Input Output Control), MISC (Miscellaneous), MGMT (Management), CTRL (Control), MBSS (Multiple BSS), WDS (Wireless Distribution System), DLS (Direct Link Setup), BBP (Base Band Processor), TX (Transmission), RX (Receive), MAT (Mac Address Translation), WSC (Wi-Fi Simple Config), INIT (Initialize), MLME (802.11 MAC Layer Management Entity), CONF (Configuration), CHAN (Channel), MCU (Microcontroller Unit), DESCP (Descriptor), AC (Access

Category), INT (Interrupt), ISR (Interrupt Service Routine), CMD (Command), RF(Radio Frequency), MCU (Microcontrollers), SW(Software).

2 DEVICE DRIVER MODULES

RT_WIFI separates into several modules, organized by functionality (except for the classification of layers described in previous chapter.) These modules are introduced briefly in the subsequent sections.

2.1 Device/Driver Init/Exit Module

A network device must be initialized correctly and the hooking points must be predefined to connect to the operating system. The hooking points let the network packets be exchanged. RT_WIFI summarizes those as three pieces, as show in Figure 2-1.

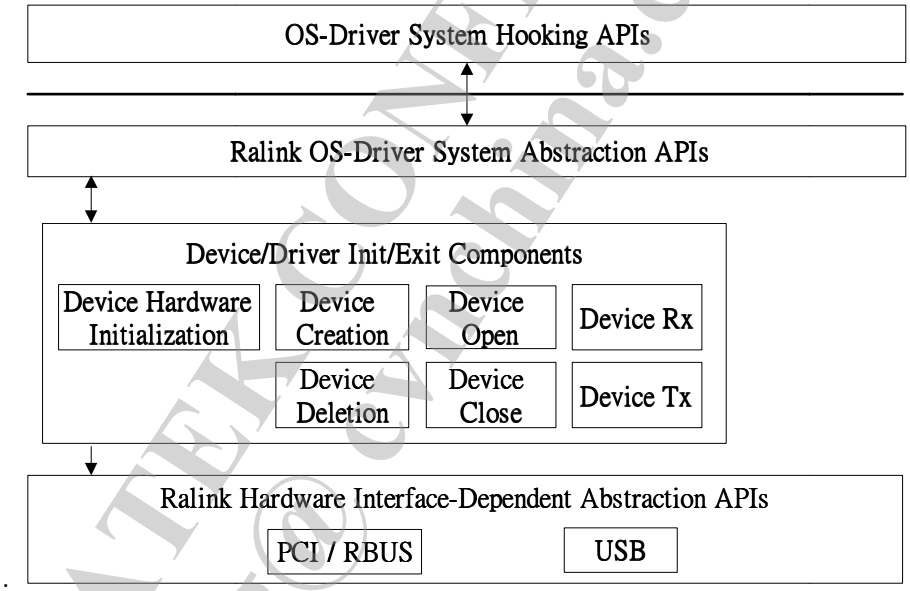

Figure 2-1 Main Components of Device/Driver Init/Exit Module

The “Device/Driver Init/Exit Components” is the primary part. It is necessary for hardware initializations, allocations for various necessary memories, pre-processing or post-processing for data exchanges between the OS and network device. It also makes sure the quality is satisfactory, and stops the device when told to by the OS.

The “OS-Driver system Hooking APIs” is a set of functions or entry points specified by the OS. Each OS has specified uses for those APIs. The device driver should follow those pre-defined interfaces and parameters to connect to the OS.

The “Ralink OS-Driver System Abstraction APIs” is a wrapper function set to pack the functions in “Device/Driver Init/Exit Components” to the correct format as specified in “OS-Driver System Hooking APIs.” This lets the two components connect.

When RT_WIFI must be ported to an OS, the “Ralink OS-Driver System Abstraction APIs” is one of the primary tasks. There are more detail about these APIs in Chapter 3. The specific uses and the procedures of “Device/Driver Init/Exit Components” are in the related programming guides and the source code supplies.

2.2 Network Data Exchange Module

The “Network Data Exchange” module is the primary function of a network device at the software level. It receives the network data from the OS and transfers it outside of the network device, or fetches the data received from network device and forwards it to the OS. Figure 2-2 shows the main components of this module implemented in RT_WIFI.

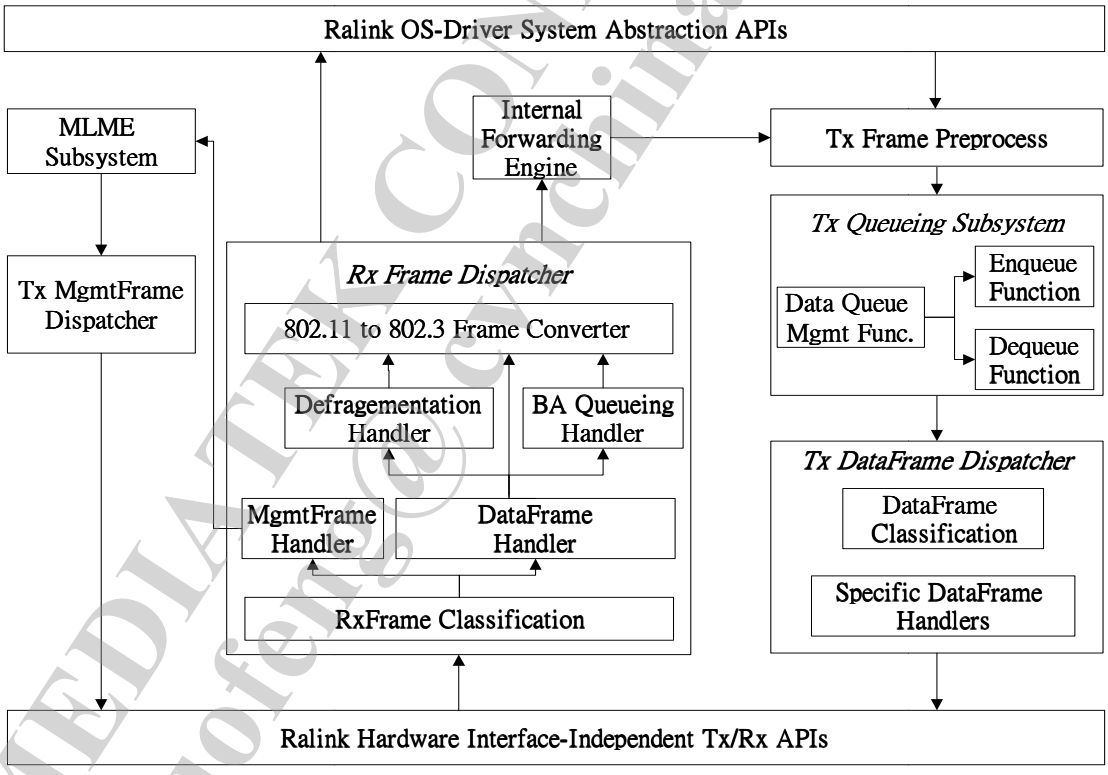

Figure 2-2 Components of Network Data Exchange Module

These components can be divided into three parts: transmission; reception; and related OS abstraction wrapper functions. A short description of each part is provided in the subsequent sections.

2.2.1 Flow of Packet Transmission

When the RT_WIFI receives a packet from the OS, the packet goes through the flow of transmission and is sent to the hardware. The whole transmission flow includes the subsequent procedures.

The “Tx Frame Preprocess” will first do some early checks and preparation for next step (i.e. it will check if this out-going packet is valid to send, the network protocol type of packet, and the status of QoS support, etc.) After this pre-processing step, it will be passed to the RT_WIFI’s internal “Tx Queuing Subsystem” and wait to be handled later.

The packet information received by early pre-processing is necessary after the “Dequeue Function” in “Tx Queuing Subsystem” pulls the packet out. The operation status of the device, and the connection status of the peer which this packet will be sent to, is used to construct the packet in the correct format, and send it to the hardware. There is a lot of information that must be sent to the hardware within this packet (e.g. sent in complete pieces or send as several fragmentations, as an A-MPDU aggregation format or an A-MSDU format, and the desired transmission rate, etc.) All of these procedures are done in the sub module named “Tx DataFrame Dispatcher”.

When these steps are completed, the packet can be sent to the hardware.

RT_WIFI must create and send some data out, in addition to the packet transmission requested from the OS. This is called the management frame, a.k.a., MgmtFrame. RT_WIFI uses another routine set called “Tx MgmtFrame Dispatcher” to handle them. The procedure of this routine set is almost the same as “Tx DataFrame Dispacther”. It removes some operations that are not necessary.

2.2.2 Flow of Packet Reception

When network data is received by the network device, RTP_WIFI gets the data directly from the hardware buffer or from the OS system calls. It then manages the data.

RT_WIFI must check to make sure the packet is valid for the upcoming processes after it has been successfully received. The receiving packet can be classified as two parts, data frames (DataFrame) and management frames (MgmtFrame). A specific routine manage it each part.

Packets classified as DataFrames are passed to the component named “DataFrame Handler”. This component can deliver the packet directly to the next stop called “802.11 to 802.3 Frame Converter” if the packet format is applicable. The packet can be sent to the “BA Queueing Handler” or “Defragmentation Handler” to do the pre-processing before it is passed to the next stop.

After the operation of “802.11 to 802.3 Frame converter” the original frame 802.11 header has changed format to 802.3. The destination address of this frame is read. The 802.3 frame may be delivered to the OS by the OS system calls or redirected to the component “Tx Frame Preprocess” through the “Internal Forwarding Engine” for further processing

Packets classified as MgmtFrames are passed to another module named “MLME Subsystem”. The “MLME Subsystem” uses various Finite State Machines (FSM) or handlers to handle different kinds of MgmtFrames. This module is explained in section 2.3.

2.2.3 Network Data Buffer Handling

There must be some data structures and memory spaces to store and show the transmission and reception of a network packet. The memory space used to store the packet is treated as a continuous addressing space by RT_WIFI at this time. The packet is stored in several separated segments.

A data structure named Control Block (CB) is used to carry the necessary information for each data packet. It may used for further processes. This data structure is linked with the packet and is active during the RT_WIFI lifetime.

A detailed description is provided in section 3.

2.3 IEEE 802.11 MLME Module

The “IEEE 802.11 MLME” module is a combination of various management related tasks. It includes several subsets, shown in Figure 2-3.

The first subset is “MLME Queueing Subsystem”. The main job of this subset is to receive the MLME requests from various sources (e.g. internal requests by the configuration system or the MgmtFrame sent by other devices, are converted to a pre-defined MLME format, and stored in the queuing system for latter handling.)

The second subset is the management handler which includes “Device Mgmt Task”, “Timer Functions”, and “MLME Mgmt Task”. These components are usually related to each other and are not easy to separate. The “Device Mgmt Task” usually handles the device’s internal requests e.g. the device driver itself, the configuration program in the upper layers, or the requests generated during FSM processes. The “Timer Functions” is the set with periodic/non-periodic timers. The periodic timers are used to trigger routine jobs or event handling when the requested time-out occurs. The “MLME Mgmt Task” is the entry point to drive the

FSMs, which fetches the formatted MLME requests from the “MLME Queueing Subsystem”. The formatted MLME is sent to the corresponding FSM for further handling.

The third subset is the set of various Finite State Machines (FSMs). The FSMs are used based on a series of IEEE 802.11 standard specifications. The FSM is the final destination to deal with the MLME request.

This MLME Module may need to respond to the requester (depending on the results of each sub component.) Some functions are also used to make and send these packets out.

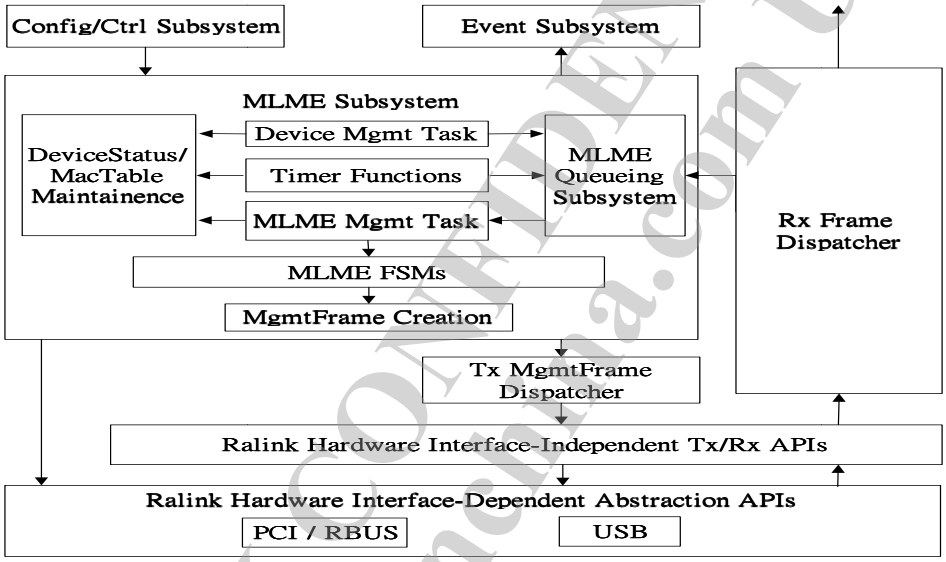

Figure 2-3 Components of MLME Module

The codes in this module should not be changed during the porting progress because they are already pre-defined as OS-independent.

2.4 Device Configuration and Control Module

The “Device Configuration and Control” module is the communication interface between the user and device driver. The user can query the device information, monitor the device operation status, and control the device using this interface.

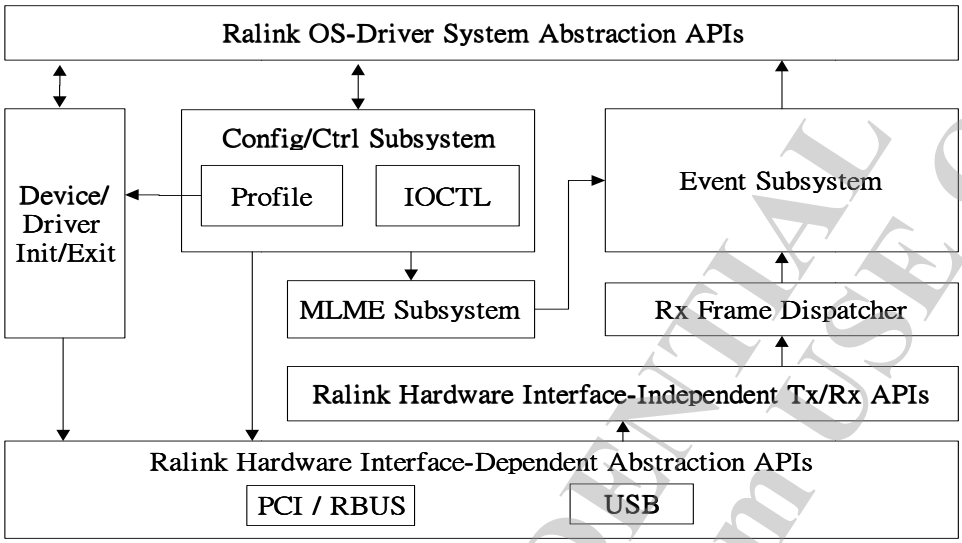

Figure 2-4 Components of Device Configuration and Control Module

As show in Figure 2-4, RT_WIFI can be used to configure and control the device in two ways.

1. "Profile" is a file based configuration mechanism. It only works during the device initialization stage. The user can disable this function or use a memory space to simulate these file operations in some operating systems that do not have file systems or do not support system calls for file operations.

2. The "IOCTL" interface is the main control interface when the device has been successfully initialized. This interface also exists in most modern operating systems.

RT_WIFI also uses another "Event Subsystem" which actively sends information to the OS. The detailed use of this event system is varied and the user can use any OS mechanism to achieve it (e.g. in Linux, the wireless event (a subset of NETLINK mechanism) can be used to implement this sub-system.

RT_WIFI can supply passive/active communication channels between the Device Driver and the OS/User if these mechanisms let it.

2.5 Integration

The main function of each module has been discussed in the previous sections. The last and most important thing is combining them and making them work correctly. All of these modules need to co-work with the OS system calls to perform their tasks. They also need to access the network data on the network device through the connected hardware interface. For this reason, RT_WIFI creates two abstracted interfaces to connect these three parts. These three parts include "Ralink OS-Driver System Abstraction APIs", and a combination of

"Ralink Hardware Interface-Independent Tx/Rx APIs" and "Ralink Hardware Interface-dependent APIs". Figure 1-1 shows the relationship between these modules and the abstraction APIs Set.

The "Ralink OS-Driver System Abstraction APIs" includes various OS-dependent functions and data structures. These functions and data structures are used to bridge the OS and RT_WIFI.

The "Ralink Hardware Interface-Independent Tx/Rx APIs" and "Ralink Hardware Interface-dependent APIs" give various Tx/Rx functions, device register access, device descriptor access, and all hardware related functions.

The next section contains a more detailed description of the OS dependent/independent functions and hardware dependent/independent functions, as well as the pre-defined wrapper functions.

3 PORTING RT_WIFI TO SPECIFIC OS

This chapter describes the RT_WIFI porting process in detail. Before beginning this process, make sure the development environment is set up and properly configured.

RT_WIFI gives various wrapper functions or data structures to create those abstraction layers and modules discussed in the previous sections. To make the explanation clearer and easier to read, one specified OS is used (i.e. Linux) as the target OS to show the uses. It is better if the device driver programming theory for Linux is understood. The user can also refer to the Linux kernel source to find the detail operation or definition of those Linux data structures or functions used in this document. The Linux kernel source is available at http://www.kernel.org/

All code segments shown in this chapter are used with the Linux OS. It will be noted if it is used for a different OS.

3.1 Hook the OS Network Device Structure and RT_WIFI

Most OSs usually create an instance for each attached device based on the pre-defined data structures to show the existence of various peripheral devices connected to the system. Through this instance, the OS and the device driver need the OS. The OS completes the operation/control to the device. e.g. for each attached network device, Linux creates an instance with the data type "net_device" to show this device. The parameters in the "net_device" or used. Linux cooperate with the device driver to drive the device properly. RT_WIFI defines many data types or wrapper functions to hook this OS-specified instance and internal data structure of RT_WIFI because of this.

The Hooking procedures of RT_WIFI and OS can separate into three stages: Initialization stage, operation stage, and exit stage. These stages are explained in subsequent sections.

3.1.1 Initialization of OS Specified Device Instance

In the RT_WIFI, the device initialization procedure is divided into three parts: Initialization of Device Hardware Interface, Initialization of Ralink Device Control Structure, and initialization of OS Network Device Instance. The Initialization of an OS Network Device Instance, is shown in this section. The Device Hardware Interface related initialization is shown in section 3.8.

To connect the RT_WIFI internal functions and the specified OS for the initialization progress, RT_WIFI defines the subsequent data structures and functions.

3.1.1.1 Data Structures

PNET_DEV

This structure is the main wrapper data type used in RT_WIFI to identify the instance of a network device. It should be defined for the OS-specified network device data type. The definition of this data type in Linux is shown as:

```
typedef struct net_device  * PNET_DEV;
```

RTMP_OS_NETDEV_OP_HOOK

This structure is the main function set for a network device to do a transmission, reception, and control. When the RT_WIFI initializes the device, it first fills the data structure then sends it as an argument to the corresponding function call to do hooking progress with the OS system calls. The definition of this data structures and example code of setup this data structure in Linux is shown as:

```
typedef struct _RTMP_OS_NETDEV_OP_HOOK_ {
    void            *open; /* The operation when device is opened */
    void            *stop; /* The function call when device is required to stop */
    void            *xmit; /* Handler when a packet send from OS to device */
    void            *ioctl; /* The entry point to do the control to the device */
    void            *get_stats; /* Callback used to retrieve device statistics */
    void            *priv; /* Pointer refer to the RTMP_ADAPTER structure */
    int             priv_flags; /* Indicate the operation type of the device */
    unsigned char   devAddr[6]; /* The MAC address binding to the device */
    unsigned char   devName[16]; /* The device name presented in the OS */
    unsigned char   needProtcted; /* Protect or not for the initialization */
}RTMP_OS_NETDEV_OP_HOOK, *PRTMP_OS_NETDEV_OP_HOOK;

pNetDevHook->open = MainVirtualIF_open;
pNetDevHook->stop = MainVirtualIF_close;
pNetDevHook->xmit = rt28xx_send_packets;
pNetDevHook->ioctl = rt28xx_ioctl;
pNetDevHook->priv_flags = INT_MAIN;
pNetDevHook->get_stats = RT28xx_get_ether_stats;
pNetDevHook->needProtcted = FALSE;
```

RtmpOSNetDevAttach(net_dev, pNetDevHook);

3.1.1.2 Function Definitions

RtmpOSNetDevAlloc()

This is the wrapper function used to do the allocation procedure for the OS-specified device instance. This function should be used if the target OS has a specified mechanism for the allocation of the network device instance.

RtmpOSNetDevRequestName()

This is the wrapper function used to get/assign the device name, which shows the device in the OS. If the target OS has the specified mechanism for the assignment of the device, this function should be used for the target OS.

RtmpOSNetDevCreate()

This is the wrapper function used to create an OS network instance. When this function call is completed, the OS network instance should be correctly created. This function may be used with the previous two functions, and maybe some extra OS-specified function calls to achieve the functionality.

RtmpOSNetDevAddrSet()

This is the wrapper function used to assign the MAC address of the network device to the OS device instance. Specified OS-specified function calls can be used in this function to complete this assignment task for the target OS.

RtmpPhyNetDevInit()

This is the entry point for OS device instance initialization. It gives the memory space for the instance; fill the "RTMP_OS_NETDEV_OP_HOOK" data structure; and other necessary OS-specified initializations which must be done before attaching this instance to the OS. The OS network instance should be ready to attach to the OS when this function is completed. This function may include some or all functions in the upper paragraph and any necessary extra OS-specified functions calls used to accomplish the initialization for the network instance of target OS.

3.1.2 Operation of OS Specified Device Instance

The RT_WIFI go into operation stage after the function "**RtmpPhyNetDevInit()**" successfully creates an OS-specified network device instance. In this stage, the main functions are attached to the OS device instance to OS. The device is controlled and the job is cleaned after the device is required to stop.

3.1.2.1 Function Definitions

RtmpOSNetDevAttach()

This is a wrapper function and should be used referring to the OS. It uses the OS device instance created in function "**RtmpPhyNetDevInit()**" and attaches it to the OS through this function. After the

device has successfully attached to the OS, the OS can control, i.e. open, close, transmission, reception, query, etc., to the device through the OS network instance.

RtmpOSNetDevDetach()

This wrapper function is used to detach the OS network instance from the OS when the device is requested to stop. The use of this function refers to the target OS.

rt28xx_open()

This wrapper function is used to do necessary operations before the network device can do transmission and reception. This function should be used referring to the target OS and obey the subsequent sequence:

Step 1: OS-specified configurations must be done before starting the operation to the device. e.g., initializations of hardware IRQ for PCI interface, etc.
Step 2: Function "***rt28xx_init()***" for RT_WIFI to do device/configuration initialization.
Step 3: OS-specified operations need to be done after the device is ready to start. E.g., enable the hardware interrupt or change the status of OS device instance, etc.

rt28xx_close()

This wrapper function used to do necessary operations after the device is stops the service.

rt28xx_send_packets()

This wrapper function is called by the OS to send network packets to the RT_WIFI. The use should obey the subsequent sequence:

Step 1: Do necessary check for packet send to RT_WIFI and OS-specified device status.
Step 2: call ***"rt28xx_packet_xmit()"*** for the next stage of Transmission process.

rt28xx_packet_xmit()

This function mainly handles the OS-dependent sanity check and both OS and RT_WIFI related operations before the next function call "**STASendPackets()**" or "**APSendPackets()**", all OS-dependent operations about the packet should be finish in this function.

announce_802_3_packet()

This function is the wrapper function used to send the 802.3 packet from RT_WIFI to OS. At end of this function, the user should call the receive function of the target OS to deliver the packet to the OS.

3.1.2.2 Macro Definitions

Except for the functions explained in section 3.1.2.1, there are some important Macros defined to do the operation to OS network instance. Here list these Macros.

RTMP_OS_NETDEV_GET_PRIV()
RTMP_OS_NETDEV_SET_PRIV()

Usually, there is a hooking pointer specified in the OS network instance to link to the internal control structure allocated and maintained by the device driver itself. These two macros used to get/set the RT_WIFI internal control structure (i.e., the RTMP_ADAPTER) from/to the OS network instance.

```
#define RTMP_OS_NETDEV_GET_PRIV(_pNetDev)          ((_pNetDev)->priv)
#define RTMP_OS_NETDEV_SET_PRIV(_pNetDev, _pPriv) ((_pNetDev)->priv = (_pPriv))
```

RTMP_OS_NETDEV_GET_DEVNAME()
RTMP_OS_NETDEV_GET_PHYADDR()

These two macros are used to retrieve information from the OS network instance. The definitions of these macros are used to hide differences of declaration and manipulation mechanism of different OSs.

```
#define RTMP_OS_NETDEV_GET_DEVNAME(_pNetDev) ((_pNetDev)->name)
#define RTMP_OS_NETDEV_GET_PHYADDR(_PNETDEV) ((_PNETDEV)->dev_addr)
```

RTMP_OS_NETDEV_START_QUEUE()
RTMP_OS_NETDEV_STOP_QUEUE()
RTMP_OS_NETDEV_WAKE_QUEUE()
RTMP_OS_NETDEV_CARRIER_OFF()

Some OSs support some software-based flow control mechanisms for network devices. The mechanism can stop the transmission from the upper layer until the device can handle the network data again. The subsequent macros are used to do this.

```
#define RTMP_OS_NETDEV_START_QUEUE(_pNetDev)   netif_start_queue((_pNetDev))
#define RTMP_OS_NETDEV_STOP_QUEUE(_pNetDev)    netif_stop_queue((_pNetDev))
#define RTMP_OS_NETDEV_WAKE_QUEUE(_pNetDev) netif_wake_queue((_pNetDev))
#define RTMP_OS_NETDEV_CARRIER_OFF(_pNetDev)   netif_carrier_off((_pNetDev))
```

3.1.3 Exit of OS Specified Device Instance

When the RT_WIFI must stop the functionality, it goes into the exit stage. The subsequent section shows two function calls that must be used in this stage.

3.1.3.1 Function Definitions

RtmpOSNetDevClose()
RtmpOSNetDevFree()

These two wrapper functions are used to close and free the memory space of the OS device. These two functions should be used based on the procedures required by the target OS (e.g. in Linux, it must call "**dev_close()**" in function "**RtmpOSNetDevClose()**" to stop this OS device instance and call "**free_netdev()**" in function "**RtmpOSNetDevFree()**" to free the OS device instance.)

3.2 Memory Management

RT_WIFI uses different kinds of memory that must be given. The memory can be put into three groups. The subsequent sections show these memory types.

3.2.1 Network Packet Buffer

The Network Packet Buffer is the key memory type used to save the payload of a network packet in the OS. Most of operating systems have pre-defined data structures to show this Network Packet Buffer (e.g. the

"sk_buff" data structure in Linux or the "mBlk" in vxworks.) The given mechanisms and operations of the Network Buffer may be different too.

As described in previous section 2.2.3, RT_WIFI treats the payload of those network packets as a continuous space, regardless of use for the OS. Operating systems which use the scatter-gatter mechanism to make up their packets must consider the porting process.

RT_WIFI use several wrapper data structures to indicate the Network buffer which encapsulated in the OS specified data structure with most of the packet handling procedures. In the subsequent sections, these data types are shown and Linux is used as an example.

3.2.1.1 Data Structures

PNDIS_PACKET
PPNDIS_PACKET
NDIS_PACKET
PNDIS_BUFFER

The "**PNDIS_PACKET**" is the main data type used to show the network buffer in RT_WIFI. It is a wrapper data type which is specified as a pointer with type "void". It is used to refer to the OS-specified network buffer. When you need to operate the network buffer, you must still convert it to corresponding OS-specified data type. RT_WIFI defines a lot of macros to handle network buffer related operations. The definition of these macros is shown in section 3.2.1.2.

```
typedef void                * PNDIS_PACKET;
typedef PNDIS_PACKET        * PPNDIS_PACKET;
```

The "**PNDIS_BUFFER**" is specified as a pointer to indicate the payload of the network buffer. It is used to manipulate the payload of network buffer.

```
typedef char                NDIS_PACKET;
typedef char                * PNDIS_BUFFER;
```

3.2.1.2 Macro Definitions

In the RT_WIFI, most of the operations to the network buffer are specified as macros. The macros are shown in the subsequent section.

RTPKT_TO_OSPKT()
OSPKT_TO_RTPKT()

These two macros are used to do the conversion between the OS network data buffer data structure and corresponding RT_WIFI wrapper data structure

```
#define RTPKT_TO_OSPKT(_p)          ((struct sk_buff *)(_p))
#define OSPKT_TO_RTPKT(_p)          ((PNDIS_PACKET)(_p))
```

GET_OS_PKT_xxxx()
SET_OS_PKT_xxxx()

Even if we define the "**PNDIS_PACKET**" to indicate the OS network buffer in the RT_WIFI, we must still base it on the data structure of the OS network buffer to do the manipulation. The "**GET_OS_PKT_xxxx()**" and "**SET_OS_PKT_xxxx()**" is a series of macros with "**xxxx**" and shows different parameters of the network buffer. The subsequent text shows the definitions. All of these macros are used for the target OS.

```
#define GET_OS_PKT_DATAPTR(_pkt) (RTPKT_TO_OSPKT(_pkt)->data)
#define SET_OS_PKT_DATAPTR(_pkt,_dPtr)(RTPKT_TO_OSPKT(_pkt)->data) = (_dPtr)
#define GET_OS_PKT_LEN(_pkt) (RTPKT_TO_OSPKT(_pkt)->len)
#define SET_OS_PKT_LEN(_pkt, _len) (RTPKT_TO_OSPKT(_pkt)->len) = (_len)
#define GET_OS_PKT_DATATAIL(_pkt) (RTPKT_TO_OSPKT(_pkt)->tail)
#define SET_OS_PKT_DATATAIL(_pkt, _start, _len) \
                        ((RTPKT_TO_OSPKT(_pkt))->tail) = (PUCHAR)((_start) + (_len))
#define GET_OS_PKT_HEAD(_pkt) (RTPKT_TO_OSPKT(_pkt)->head)
#define GET_OS_PKT_END(_pkt) (RTPKT_TO_OSPKT(_pkt)->end)
#define GET_OS_PKT_NETDEV(_pkt) (RTPKT_TO_OSPKT(_pkt)->dev)
#define SET_OS_PKT_NETDEV(_pkt, _pDev)(RTPKT_TO_OSPKT(_pkt)->dev) = (_pDev)
#define GET_OS_PKT_TYPE(_pkt) (RTPKT_TO_OSPKT(_pkt))
#define GET_OS_PKT_NEXT(_pkt) (RTPKT_TO_OSPKT(_pkt)->next)
```

skb_put()
skb_push()
skb_reserve()

These Macros are used to do the data insertion, data removal, and space reservation operations of the network buffer space. These macros are used referring to the target OS. The subsequent text shows the implementation for vxworks platform.

```
#define skb_put(skb, n)         m_data_put(skb, n)
#define skb_push(skb, n)        m_data_push(skb, n)
#define skb_reserve(skb,n)RTPKT_TO_OSPKT(skb)->m_data += n;
```

OS_NTOHS()
OS_HTONS()
OS_NTOHL()
OS_HTONL()

Parsing the content of the network data depends on the content of the corresponding operation. It is very common procedure in RT_WIFI. One issue needs to be considered; the endian format of the data. These macros are used to do the endian conversion only between network endian (Big-endian) format and host endian format.

```
#define OS_NTOHS(_Val) (ntohs(_Val)) /* big-endian to host endian with 2 bytes */
#define OS_HTONS(_Val) (htons(_Val) /* host endian to big-endian with 2 bytes */
#define OS_NTOHL(_Val) (ntohl(_Val)) /* big-endian to host endian with 4 bytes */
#define OS_HTONL(_Val) (htonl(_Val)) /* host endian to big-endian with 4 bytes */
```

Another important part related to network buffer is the CB (control block) used for RT_WIFI. For each packet, RT_WIFI uses a control block to store the related information for this specified packet. When you are porting RT_WIFI to your target OS, find a small memory space for each packet to store the CB info. e.g. in Linux, use the "**cb[]**" parameter built-in the "**sk_buff**" structure for Linux; for vxworks, reserve a block of memory in the

head of the “pClBlk->clNode.pClBuf” as the CB. The related macros and definitions you shall port to target OS are shown in the subsequent section. The subsequent sample codes are used for vxworks.

cb

The “**cb**” is a macro point to the real start of control block. For Linux, it already has a parameter named “**cb**” in the “**sk_buff**”.

```
#define cb      pClBlk->clNode.pClBuf
```

CB_OFF
CB_MAX_OFFSET

The “**CB_OFF**” and “**CB_MAX_OFFSET**” is the start offset of the control block and maximum bytes which can be used to store the information in this control block. The “**CB_OFF**” may be different in different OSs, but the “**CB_MAX_OFFSET**” is the same for each OS.

```
#define CB_OFF              10
#define CB_MAX_OFFSET       32
```

RTMP_SET_PACKET_XXXX()
RTMP_GET_PACKET_XXXX()

This Macro set are used to manipulate the CB to set/get various information which used for the packet handling in RT_WIFI. The operation is base on the “**cb**”. Subsequent show us some sub set of this Macros set. You should refer to the RT_WIFI source code for the complete list of all macros need to be implemented.

```
#define RTMP_SET_PACKET_UP(_p, _pri) (RTPKT_TO_OSPKT(_p)->cb[CB_OFF+0] = _pri)
#define RTMP_GET_PACKET_UP(_p) (RTPKT_TO_OSPKT(_p)->cb[CB_OFF+0])
#define RTMP_SET_PACKET_FRAGMENTS(p,n)(RTPKT_TO_OSPKT(p)->cb[CB_OFF+1]=n)
#define RTMP_GET_PACKET_FRAGMENTS(p) (RTPKT_TO_OSPKT(p)->cb[CB_OFF+1])
#define RTMP_SET_PACKET_WCID(p, idx) (RTPKT_TO_OSPKT(_p)->cb[CB_OFF+2] = idx)
#define RTMP_GET_PACKET_WCID(p) ((UCHAR)(RTPKT_TO_OSPKT(p)->cb[CB_OFF+2]))
```

This section gives information about some macros related to the enqueue/dequeue operations for network buffer. It does not include information about the macros talked about in previous sections.

QUEUE_ENTRY_TO_PACKET()
PACKET_TO_QUEUE_ENTRY()

These two macros are used to do the data type conversion between the Queue entry format to RT_WIFI network buffer format. The the sample codes are as follows:

```
#define QUEUE_ENTRY_TO_PACKET(pEntry)    (PNDIS_PACKET)(pEntry)
#define PACKET_TO_QUEUE_ENTRY(pPacket)    (PQUEUE_ENTRY)(pPacket)
```

3.2.1.3 Function Definitions

RT_WIFI also defines some functions used to manipulate the allocation, copy, and query for the network buffers. The functions used for the target OS are explained in the subsequent section.

skb_copy()

This function was originally used in Linux. It is used to duplicate another new network buffer data structure and its payload from the original one, and does not change the original at all. When this function is used for your target OS, you should obey this requirement.

skb_clone()

This function was originally implemented in Linux. It only copies the contents of data structure of original network buffer and shares the same memory space as the payload owned by original one (i.e. this two net buffer data structure shows one payload space.) There are two key features of this function.

1. The payload is only made available after the last network buffer data structure, which means it is available.
2. Any modification to the payload will simultaneously effect other network buffer data structures. The benefit of this function is it saves memory space. It would be better if it was implemented for the target OS, based on these features. The user can also use a "skb_copy()" to replace it.

ClonePacket()

This function is an appliance of "**skb_clone()**" used for A-MSUD frame handling in RT_WIFI. This function use "**skb_clone()**" to create a new network buffer data structure and modify the data pointer of new network buffer to a specified offset which indicated in the argument of this function. The function is used for target OSs referring to this requirement.

duplicate_pkt()

This function is used to create a new network buffer data structure and payload referring to the input arguments of this function. If the creation is successfully, the payload is copied and the corresponding fields of the network buffer data structure are filled. This function needs this requirement.

DuplicatePacket()

This function is the appliance of "**skb_clone()**", except it has one more parameter used to indicate which network device instance this packet sent from.

duplicate_pkt_with_TKIP_MIC()

This function checks if the room space of the payload of the input network buffer is big enough to add extra data to the tail of the payload. If there is enough space, it adds the data directly. If there is not enough space, a new network buffer is made to save the original data and the new data is added to the tail. This function is used referring to this requirement.

duplicate_pkt_with_VLAN()

This function is similar to "**duplicate_pkt()"** but with different input parameters.

RTMP_QueryPacketInfo()

This function is used to query information from the network buffer data structure and fill the argument as specified by the data type "**PACKET_INFO**". The definition of data type "**PACKET_INFO**" is referred to. Refer to the RT_WIFI source code.

update_os_packet_info()

This function is used to change some specified internal fields of the OS's network buffer data structure. Refer to the RT_WIFI source code for more details.

wlan_802_11_to_802_3_packet()

This function is used to convert the 802.11 header of a network buffer to the 802.3 header, and to do necessary modifications to the network buffer fields. This network buffer should delivery to OS for future handling.

RTMPFreeNdisPacket()

This function is used to free the network buffer and corresponding payload. This function refers to the rule of target OS. Another macro named "**RELEASE_NDIS_PACKET()**" is used to wrap this function. An example of this macro is shown below.

```
#define RELEASE_NDIS_PACKET(_pAd, _pPacket, _Status) \
        RTMPFreeNdisPacket(_pAd, _pPacket);
```

3.2.2 Hardware Buffer

There are some memory space requirements for the hardware in RT_WIFI. There are some restrictions for the memory required by the hardware (e.g. the memory space may need to be physically continuous, or the memory space should be able to be accessed by the DMA.) The data structures and functions related to the hardware buffer are explained in the subsequent section.

3.2.2.1 Data Structures

dma_addr_t

This data type is the build-in data type of Linux and used to indicate that this is the DMA address. This data type should be defined to a proper data type referring to the target OS. The subsequent example shows the definition for vxworks.

```
typedef unsigned int        dma_addr_t;
```

NDIS_PHYSICAL_ADDRESS
PNDIS_PHYSICAL_ADDRESS

These two data types are used to indicate the physical address of the memory. The subsequent example shows the definition for vxworks.

```
typedef  dma_addr_t                 NDIS_PHYSICAL_ADDRESS;
typedef  dma_addr_t                 * PNDIS_PHYSICAL_ADDRESS;
```

3.2.2.2 Function Definitions

RTMP_GetPhysicalAddressLow()
RTMP_GetPhysicalAddressHigh()

These two functions are used to get the high or low part (32 bits) of the input physical address.

RTMP_SetPhysicalAddressLow()
RTMP_SetPhysicalAddressHigh()

These two functions are used to set the high or low part (32 bits) of the input physical address to the destination.

3.2.3 Normal Memory

"Normal memory" refers to any other memory that is not used as a network buffer or a hardware buffer. This kind of memory is usually used in the internal of device driver (e.g. the database system, state machine, etc.)

The macros and functions related to this kind of memory are explained in the subsequent section.

3.2.3.1 Macro Definitions

MEM_ALLOC_FLAG

This macro is used to show the flags used for memory allocation. If the target OS doesn't use any flag as the parameter to do the memory allocation, it can set as 0. This flag should be defined to make the allocation work in the any condition (e.g. even in the interrupt can do the allocation and properly return.)

#define MEM_ALLOC_FLAG (GFP_ATOMIC)

3.2.3.2 Function Definitions

The subsequent functions are used to control the memory. These functions should be used referring to the descriptions in this section.

Kmalloc()
Kfree()

These two functions used to allocate memory and free memory. Usually, the memory space allocated by this function is continuous in physical level.

Vmalloc()
Vfree()

These two functions is a pair used to allocate/free memory. Usually, the memory space allocated by this function may not continuous in physical level. So it may not be able to be used for hardware related access.

NdisMoveMemory()
NdisCopyMemory()
NdisZeroMemory()

NdisFillMemory()
NdisCmpMemory()
NdisEqualMemory()
RTMPEqualMemory()
MlmeAllocateMemory()
MlmeFreeMemory()
COPY_MAC_ADDR()

These functions used to copy, move, clean, and fill the memory indicated by the input parameters. You should implement these function base on the implementation in Linux.

copy_to_user()
copy_from_user()

These two functions are the system call of Linux used to transmit memory between user space and kernel space. For the target Os which only has one memory space, you can set it as normal memory copy functions. Subsequent sample code show the definition in vxworks.

```
static inline int copy_to_user(void *pDst, void *pSrc, int len)
{
    memcpy(pDst,pSrc,len);
    return 0;
}

static inline int copy_from_user(void *pDst, void *pSrc, int len)
{
    memcpy(pDst, pSrc, len);
    return 0;
}
```

3.2.4 Memory Access

In some hardware platform or OS, they are some restriction for memory access, e.g., you can not directly access a 4 bytes integer which not starts in a 4 byte aligned address. In the RT_WIFI, there are some memory access may not aligned, so you need implement subsequent functions if the target Os you porting to is not support unaligned access.

get_unaligned()
put_unaligned()

The other function you may need to implement is the memory barrier. If the target OS and the compiler may change the execution sequence of source code, you should implement subsequent function to prevent this problem occurred.

wmb()

The implementations of these functions are CPU-specified and OS dependent and out of the scope of this document.

3.3 Synchronization mechanism

In the RT_WIFI, there are many tasks and procedures share the same data information, properly mechanism to protect the synchronization of data is necessary. In this section, we introduce two synchronization mechanisms used in RT_WIFI.

3.3.1 Lock Protection

The Lock Protection is used to prevent other tasks break the currently operation and take the control of the CPU.

NDIS_SPIN_LOCK

This data type is a wrapper used to indicate the lock data type of the target OS. Subsequent show the definition in Linux.

```
typedef spinlock_t              NDIS_SPIN_LOCK;
```

os_lock

This data type is another wrapper used to indicate the lock data type of the target OS. Usually this data type is used the linking the data structure "**NDIS_SPIN_LOCK**", and the OS-specified lock data type. Subsequent show the definition and usage for vxworks.

```
struct os_lock    {
    spinlock_t          lock;
    unsigned long       flags;
};
typedef struct os_lock          NDIS_SPIN_LOCK;
```

Subsequent introduce the functions or macros used to allocate, and operate to the lock parameters.

NdisAllocateSpinLock()
NdisFreeSpinLock()

These two functions used to allocate/free the lock data structures.

```
#define NdisAllocateSpinLock(__lock)    spin_lock_init((spinlock_t *)(__lock));
#define NdisFreeSpinLock(lock)          do{}while(0)
```

RTMP_SEM_LOCK()/NdisAcquireSpinLock()
RTMP_SEM_UNLOCK()/NdisReleaseSpinLock()

These four functions used to acquire/release the lock. If the task is protected by this lock, only the hardware interrupt can preempt currently operations.

```
#define RTMP_SEM_LOCK(__lock)       spin_lock_bh((spinlock_t *)(__lock));
#define RTMP_SEM_UNLOCK(__lock)     spin_unlock_bh((spinlock_t *)(__lock));
```

RTMP_IRQ_LOCK()

RTMP_IRQ_UNLOCK()

These two functions used to acquire/release the lock. These two functions are very similar with upper two functions except these two will set/clear a flag (pAd->irq_disabled). Also, if the task is protected by this lock, only the hardware interrupt can preempt this task.

```
#define RTMP_IRQ_LOCK(__lock, __irqflags)          \
{                                                  \
    __irqflags = 0;                                \
    spin_lock_bh((spinlock_t *)(__lock));          \
    pAd->irq_disabled |= 1;                        \
}

#define RTMP_IRQ_UNLOCK(__lock, __irqflag)         \
{                                                  \
    pAd->irq_disabled &= 0;                        \
    spin_unlock_bh((spinlock_t *)(__lock));        \
}
```

RTMP_INT_LOCK()
RTMP_INT_UNLOCK()

These two functions used to acquire/release the lock. If the task is protected by this lock function, neither the hardware interrupt nor any task can preempt this task.

```
#define RTMP_INT_LOCK(_lock, _flags) spin_lock_irqsave((spinlock_t *)_lock, _flags);
#define RTMP_INT_UNLOCK(_lock, _flag) \
  spin_unlock_irqrestore((spinlock_t *)(_lock), ((unsigned long)_flag));
```

3.3.2 Semaphore

The system of signaling in RT_WIFI is mainly used as an event to start a waiting task. The structures and functions specified in RT_WIFI section data are shown in this section.

RTMP_OS_SEM

This data type is a wrapper used to indicate the OS-specified semaphore data structure. An example of the definition for Linux is shown in the subsequent text.

```
typedef struct semaphore  RTMP_OS_SEM;
```

wait_queue_head_t

This data type specified in Linux is used to create a waiting queue. If the target OS has another mechanism to implement the waiting mechanism, you can simply set it as a dummy type. A definition in vxworks is shown in the subsequent text.

```
typedef int      wait_queue_head_t;
```

completion

This data type specified in Linux is used to create an event queue to queue threads that must wait for the “completion” event. If the target OS has another mechanism to implement this “completion”

event, you can simply set it as a dummy type. A definition in vxworks is shown in the subsequent text.

```
struct completion
{
    unsigned int done;
    SEM_ID          wait;
};
```

RTMP_SEM_EVENT_INIT()
RTMP_SEM_EVENT_INIT_LOCKED()
RTMP_SEM_EVENT_WAIT()
RTMP_SEM_EVENT_UP()

These four macros are used to initialize the event data structure with lock or unlocked state, waiting for some event, and trigger some event to wake up. The use of these four macros in vxworks is shown in the subsequent text.

```
#define RTMP_SEM_EVENT_INIT_LOCKED(_pSema) \
    do{
        SEM_ID      __pSemaID;\
        __pSemaID = semCCreate((SEM_Q_FIFO | SEM_INTERRUPTIBLE), 0);    \
        *(_pSema) = __pSemaID;\
    }while(0)

#define RTMP_SEM_EVENT_INIT(_pSema)    \
    do{                                \
        (*(_pSema)) = semCCreate((SEM_Q_FIFO | SEM_INTERRUPTIBLE), 1);   \
    }while(0)

#define RTMP_SEM_EVENT_WAIT(_m, _s) ((_s)=semTake((*(_m)), WAIT_FOREVER))
#define RTMP_SEM_EVENT_UP(_pSema)     semGive((*(_pSema)))
```

3.4 Task

The tasks done by RT_WIFI can divided into two groups. One group of tasks are related to network transmission and reception. The other group of tasks are related to device and MLME management. These two kinds of task are called “Network Data Processing Task” and “MLME Task”, respectively. The priority of “Network Data Processing Task” must be higher than “MLME Task” when porting job to the target OS (i.e. the network task is faster than the execution of MLME task). The data structures and operations related to these tasks are explained in the subsequent section.

3.4.1 Network Data Processing Task

RTMP_NET_TASK_STRUCT

PRTMP_NET_TASK_STRUCT

The "**RTMP_NET_TASK_STRUCT**" is a wrapper of the OS-specified task data structures. The Linux definition is shown in the subsequent text.. The "**PRTMP_NET_TASK_STRUCT**" is a pointer of "**RTMP_NET_TASK_STRUCT**".

```
typedef struct tasklet_struct    RTMP_NET_TASK_STRUCT;
typedef struct tasklet_struct    *PRTMP_NET_TASK_STRUCT;
```

tasklet_hi_schedule()

This function is used to schedule a Network Data Processing Task to execute the routine. This function is a system call of Linux. For the target OS, a proper function should be used to integrate with the task scheduling of the target OS. The subsequent sample code show the use of this function in vxworks.

```
#define tasklet_hi_schedule(_pTask) \
do{\
    STATUS _ret;\
    _ret = netJobAdd((FUNCPTR)((_pTask)->funcPtr), (int)((_pTask)->data), 0, 0, 0, 0);\
    if (_ret == OK)      \
        (_pTask)->taskStatus = RTMP_NET_TASK_PENDING;\
    else\
        logMsg("NetTask(%s) add failed!\n", (int)((_pTask)->taskName),0,0,0,0,0);\
}while(0)
```

3.4.2 MLME Task

The MLME task is designed as other kinds of task with lower priority than net task. In Linux, these task implemented by the kernel thread. In vxworks, these tasks implemented by "**taskSpawn()**" with specified priority. For the target OS, you should use proper way to create these MLME tasks.

RTMP_OS_PID
THREAD_PID

These two data types are wrapper of the OS-specified task identifier or instance. Subsequent show the definition in Linux.

```
typedef pid_t                RTMP_OS_PID;
typedef  pid_t THREAD_PID;
```

RTMP_OS_MGMT_TASK_FLAGS

This macro is used to define the flag used to create the task. This flag is currently only used for Linux. For the target OS, you may need to specify it, or set it to 0.

```
#define RTMP_OS_MGMT_TASK_FLAGS      CLONE_VM
```

RtmpOSTaskInit()

This function is used to initialize a new task.

RtmpOSTaskCustomize()

This function is used to to customize the created task (e.g. change the name of the task, or change the attribution of the task, etc). The use of this function is variant and refers to the target OS.

RtmpOSTaskAttach()

This function is used to start the created task.

RtmpOSTaskNotifyToExit()

This function is used by a task to notify others that it has stopped. The use of this function is variant and refers to how the task is created in the target OS.

RtmpOSTaskKill()

This function used to notify an existing task to stop. The use of this function is variant and refers to how the task is created in target OS.

3.5 Time and Timer

RT_WIFI uses many software timers to handle various timing related procedures. For this requirement,

RT_WIFI defines many data structures and functions to show the time and timers.

RTMP_OS_TIMER
NDIS_MINIPORT_TIMER

These two data types are wrappers of the OS-specified timer structures. It refers to the target OS to set this data type. Linux definitions are shown in the subsequent section.

```
typedef struct timer_list    RTMP_OS_TIMER;
typedef struct timer_list    NDIS_MINIPORT_TIMER;
```

TIMER_FUNCTION

This is the data type of callback function when a timeout occurred. The definition for vxworks is shown in the subsequent section.

```
typedef void (*TIMER_FUNCTION)(unsigned long);
```

OS_HZ

This macro is used to identify how many ticks per seconds occurred for OS. The subsequent text shows the definition for vxworks.

```
#define OS_HZ                sysClkRateGet() /*HZ */
```

RTMP_TIME_AFTER (a,b)
RTMP_TIME_AFTER_EQ (a,b)
RTMP_TIME_BEFORE(a,b)

These three macros are used to check the relationship between time a and b. These macros should be used to avoid the turn-around issue. The subsequent text shows the use of these macros in Linux.

```
#define RTMP_TIME_AFTER(a,b)        \
    (typecheck(unsigned long, (unsigned long)a) && \
    typecheck(unsigned long, (unsigned long)b) && ((long)(b) - (long)(a) < 0))
#define RTMP_TIME_AFTER_EQ(a,b)   \
    (typecheck(unsigned long, (unsigned long)a) && \
    typecheck(unsigned long, (unsigned long)b) && ((long)(a) - (long)(b) >= 0))
#define RTMP_TIME_BEFORE(a,b)      RTMP_TIME_AFTER_EQ(b,a)
```

OS_WAIT()

This macro is used to wait for some time. The use refers to the OS and there should be no wait. The subsequent text shows the definition in vxworks.

```
#define OS_WAIT(_time)                  taskDelay(_time)
```

RTMP_OS_Init_Timer()

This function is used to initialize a timer, but not start it immediately. The use of this function is variant and refers to the OS.

RTMP_OS_Add_Timer()

This function is used to end a created timer and start the timeout counting. If the timer is not yet made, this function makes it automatically. If the timer has started, this function automatically resets the timeout to the value of the input parameter.

RTMP_OS_Mod_Timer()

This function used to change the timeout of a fired timer. If the timer does not exist, it is made and then set the timeout value referring to the input parameter.

RTMP_OS_Del_Timer()

This function is used to delete a fired timer and free the timer instance. If the timer has exited, then directly return.

RTMP_SetPeriodicTimer()

This function is used to create a timer which will execute periodically.

RTMPusecDelay()

This function is used to make a delay in the time in µsec units.

NdisGetSystemUpTime()
RTMP_GetCurrentSystemTime()

These two functions are used to get the system time when the OS is booting up.

3.6 File operation

RT_WIFI has a profile based configuration mechanism. The configuration can read from a file and set to the related data parameters for further use. If the target OS supports the file system and must use the file as the configuration mechanism, the functions shown in this section should be used.

3.6.1 Data Structures

RTMP_OS_FD

This data type is the wrapper of the data structure of a file descriptor in the target OS. The subsequent text shows the definition in Linux.

typedef struct file RTMP_OS_FD;*

RTMP_OS_FS_INFO

This data structure is used for Linux to save the uid, gid, and information of memory segment. If the target OS is monolithic and all task run in the same memory space, you can set this data structure as a dummy one. The subsequent text shows the definition in Linux.

```
typedef struct _RTMP_OS_FS_INFO_
{
    int                 fsuid;
    int                 fsgid;
    mm_segment_t  fs;
}RTMP_OS_FS_INFO;
```

3.6.2 Function Definitions

RtmpOSFileOpen()

This function opens a file and returns the file descriptor in data type "**RTMP_OS_FD**".

RtmpOSFileClose()

This function closes the file indicated by the input file descriptor.

RtmpOSFileSeek()

This function changes the file data pointer to the specified offset.

RtmpOSFileRead()

This function reads data from the file indicated by the file descriptor.

RtmpOSFileWrite()

This function writes data to the file indicated by the file descriptor.

RtmpOSFSInfoChange()

This function changes the uid, guid and memory segment in Linux. For those OSs, this function can be set as a dummy.

IS_FILE_OPEN_ERR()

This macro checks if the file descriptor is valid or not. The subsequent text shows the sample code for Linux.

```
#define IS_FILE_OPEN_ERR(_fd)        IS_ERR((_fd))
```

3.7 IOCTL and Event Subsystem

RT_WIFI uses two mechanisms for device configuration and control. The first is the profile, and the the other is the IOCTL. In this section, some data structures used in IOCTL subsystem are shown.

3.7.1 IOCTL

The subsequent data types are native to Linux and are used by IOCTL to carry information. When RT_WIFI is ported to the target OS, these data types should be set to make the IOCTL work. The subsequent text shows the definitions for vxworks.

iw_point
iwreq_data
iwreq

```
struct iw_point {
    PVOID           pointer;
    USHORT          length;
    USHORT          flags;
};
union iwreq_data {
    struct iw_point data;
};
struct iwreq {
    union     iwreq_data          u;
};
```

IW_SCAN_MAX_DATA
IWEVCUSTOM
IW_PRIV_SIZE_MASK

```
#define IW_SCAN_MAX_DATA        4096    /* in bytes */
#define IWEVCUSTOM              0x8C02 /* Driver specified ascii string */
#define IW_PRIV_SIZE_MASK       0x07FF /* Max number of those args */
```

3.7.2 Wireless Event

RT_WIFI uses a mechanism to send the event to the upper application. In Linux, RT_WIFI uses the netlink based wireless event to do that. To target other OSs, the subsequent functions should be used.

RtmpOSWrielessEventSend()

This function sends the event from RT_WIFI to the OS. This function is based on the mechanism supplied by the target OS. If this feature is not supported in the target OS, it can be set as dummy function.

3.8 Hardware Interface-Dependent APIs

RT_WIFI is designed to support multiple chipsets. It must be compatible with various hardware interfaces. In this section, three hardware interfaces supported by RT_WIFI are shown.

3.8.1 PCI

RT_WIFI and the device use the PCI interface for communication. There is a lot of memory and register access issues that must be referred to. In this section, the DMA memory allocation, cache coherency, IRQ request/release, and register access related operations are shown.

PCI_ALLOC_CONSISTENT()
PCI_FREE_CONSISTENT()

These two macros allocate/free memory for PCI device. The allocated memory is usable for the DMA and the cache coherent issues should be considered.

PCI_MAP_SINGLE()
PCI_UNMAP_SINGLE()

These two macros clear the cache and write back to memory or the invalid the cache memory.

RtmpOSIRQRequest()
RtmpOSIRQRelease()

These two functions get IRQ or release the IRQ from/to the OS. The assignment of IRQ refers to the target OS. These functions should be used referring to the requirement of the target OS.

readl()
readb()
writel()
writew()
writeb()

These five functions access the register in the PCI device. These functions should be used referring to the requirement of the target OS.

RTMP_AllocateTxDescMemory()
RTMP_AllocateMgmtDescMemory()
RTMP_AllocateRxDescMemory()
RTMP_FreeDescMemory()

These four functions give out memory for the hardware descriptor. The memory is given out by the "**PCI_ALLOC_CONSISTENT()**", or the DMA access may fail.

RTMP_AllocateFirstTxBuffer()
RTMP_FreeFirstTxBuffer()

These two functions give out/free the memory space for first network data buffer. This memory space is used for DMA. Cache coherency issues should be considered.

3.8.2 USB

3.8.2.1 Register RT_WIFI USB Driver Porting

USB is a Plug & Play system. Most modern OSs support callback functions made by the USB core when a USB event occurs. The RT_WIFI driver now supports the four kinds of callback functions described below. These callback functions refer to the OS, so when porting to another OS, you must located them in the specified "os\[os_name]\" directory and named as usb_main_xxx.c, such as

"os\linux\usb_main_dev.c" (Linux)

"os\vxworks\usb_main_end.c" (VxWorks)

rtusb_probe
The probe function is called by the USB core when the RT USB device is plugged into the USB hub port. The RT_WIFI USB device driver is bound to the RT USB dongle when both Vendor ID and Product ID match. One is in the USB host driver registered before and the other is from the USB dongle.

rtusb_disconnet
The disconnect function is called by USB core when RT USB dongle is removed from USB hub port.

rtusb_suspend
When there is no TX, RX packets are being transmitted and received, most USB host controllers will suspend the USB device to conserve power. The USB core calls this API.

rtusb_resume
If there are more TX, RX packets to be transmitted and received in suspend mode, the device will be resumed by USB core. The USB core calls this API to make RT_WIFI driver continue transferring packets.

3.8.2.2 RT_WIFI USB MAC/BBP Control and TX/RX Transfer Porting

RT USB dongle is made of one control endpoint which controls MAC or BBP register through, 4 bulk-out endpoints (EP5 and EP6 not used), and 1 bulk-in endpoints which RT_WIFI driver uses for TX or receiving RX packers. In "common\rtusb_bulk.c" file, there are many APIs for wrapping Wi-Fi frames into USB specified structures (e.g. Linux uses the URB structure) to send to or receive from these endpoints. If you want to port to another OS, you must know how to wrap 802.11 packets to OS specified USB request block (URB).

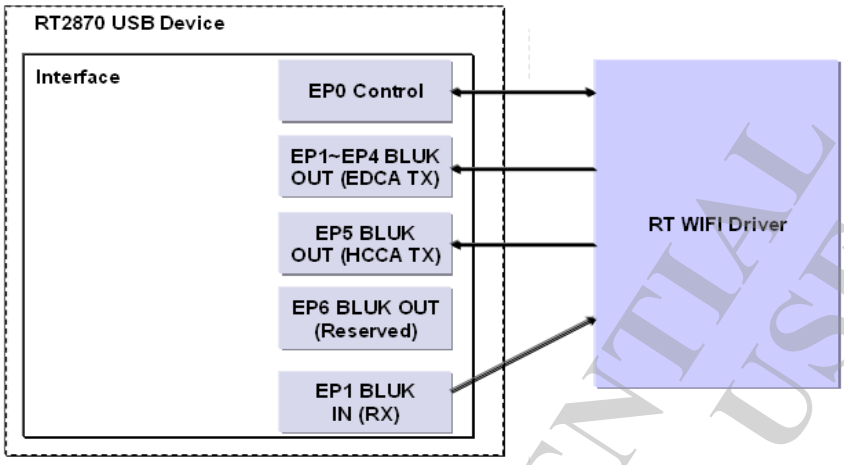

Figure 3-1 RT USB Dongle Internal Structure

In “common\rtusb_bulk.c”, there are a lot of APIs for these common wrapper behaviors. These APIs call the subsequent macros in “include\[os_name]\rtmp_[os_name].h (e.g. rtmp_linux.h, or rtmp_vxworks.h)” to allocate, initialize TX, RX URB, and submit URB to USB core.

RTUSB_ALLOC_URB

Allocate specified OS USB request block structure.

RTUSB_FREE_URB

Free specified OS USB request block structure.

RTUSB_URB_ALLOC_BUFFER

Allocate memory buffer for transferring TX packets or receiving RX packets.

RTUSB_URB_FREE_BUFFER

Free memory already allocated for transferring or receiving packets.

RTUSB_FILL_TX_BULK_URB

Fill TX packets address, packets size, completion callback function, USB device info, private context, and etc, into URB structure

RTUSB_FILL_HTTX_BULK_URB

Fill high throughput transfer packers address, packets size, completion callback function, USB device info, private context and etc, into URB structure

RTUSB_FILL_RX_BULK_URB

Fill RX packets address, packets size, completion callback function, USB device info, private context and etc, into URB structure

RTUSB_URB_DMA_MAPPING

Notify USB Core to do DMA mapping for this URB not done by RT_WIFI driver

RTUSB_SUBMIT_URB

Submit URB to USB core and hand off URB control to USB core.

RTUSB_UNLINK_URB

Stop URB transfer and hand off URB control from USB core to RT WIFI driver

In “common\rtusb_io.c”, it has a lot APIs to control MAC and BBP register or downloading firmware through endpoint zero. RT_WIFI driver uses below macro in “include\[os_name]\rtmp_[os_name].h” to wrap these control transfers.

RTUSB_CONTROL_MSG

Fill USB device info, endpoint address, request, request type, value, index, control buffer, control buffer length, and time-out value to USB control URB structure, and hand off control to USB core.

3.8.2.3 Specified USB APIs (For different OS) implementation

In “os\[os_name]\rt_usb.c” file, we implement OS-specified APIs called by “RTUSB_XX Macro” described above and management thread (MIme, cmdQ, and timer threads) OS-Specified initialization and termination APIs.

3.8.3 RBUS

The RBUS interface is like the PCI bus and does not need any special handling for hardware related configurations.

4 SOURCE CODE MANAMEMENT FOR RT_WIFI

There are many source files in the RT_WIFI source package. These files separate into several sub-directories. Figure 4-1 shows t the directory list of RT_WIFI source tree.

Figure 4-1 RT_WIFI Source File Tree Layout

As show in Figure 4-1, we can separate the sub directories into three groups, referring to the porting method. The first group has the directories that should not be touched (with a few exceptions). The second group has the compiling related tool folders that are needed to make changes. The third group is what is needed to create new files or folders. In this section we only discuss the sub directories which are related to and needed by the target OS. For a detailed explanation of each file, please refer to other design specifications.

4.1 RT_WIFI Source Tree Layout

The first group has the directories that should not be touched, as shown below.

ap

Files in this sub directory are used for RT_WIFI work in AP mode

sta

Files in this sub directory are used for RT_WIFI work in STA mode

chips

Files in this sub directory are used for RT_WIFI to support different wireless chipset. The files in this directory should be named a "chipname_or_family.c" file. E.g., the source file "***rt30xx.c***" means some command settings for all rt30 series chipsets; the source file "***rt3070.c***" means specified functions only work for RT3070.

common

Files in this sub directory refer to the MLME module, network data exchange module, and various functions used to implement related IEEE 802.11 specifications.

CVS

Files used for CVS version control, it does not have any source file.

Most of the files in these directories are OS-independent and there is no need to port them. It is possible you need to work around some issue for specified OS. When the OS-specified patch in these directories is added, please add a proper compile flag to cover the patch segments and make sure it will not effect other OSs.

The second group has the compiling related tool folders that are needed to make changes

striptool

Files used to remove un-necessary source code. You may need to add some compile flags or files in the source code to strip the source code correctly.

tools

Utilities used for generation of "***firmware.h***", you need to port the bin2h.c to work properly in the compile environment of the target OS, or the "***firmware.h***" may not correct, this will make serious problem when bring the RT_WIFI up.

The third group is the most important for porting to the target OS. You must add some of the included files for the target OS. You must also create new source files and functions to implement those functions or macros discussed in the previous section. The sub directories include:.

include

All header files implemented for RT_WIFI is located in this directory. There are several sub directories in this folder.

"***chip***" – All files in this sub directory are used to define chip-related data structures or macros.

"***iface***" – All files in this sub directory are used to define hardware interface related data structures or macros.

"***os***" – All files in this sub directory are used to define OS related data structures or macros. For each target OS, it should define a header file to include all definitions, data structures, or macros which related to the target OS, and this OS- specified header file should only included by the header file "***rtmp_os.h***" which located in the "***include***" folder. You can refer to the existing OS-specified header files (e.g., the "***rt_linux.h***" file for Linux platform) for the content of the header file you should implement for the target OS. For the naming of the OS-specified header file, you should subsequent the template "rt_OSNAME.h", here the "OSNAME" is the significant name which can used to show the target OS.

os

All OS-specified implementations for RT_WIFI shall be put in this directory. For each target OS, there is a sub folder for it. E.g., the sub folder "***linux***" is implemented for Linux platform; the sub folder "***vxworks***" is implemented for vxworks platform. For your target OS, you also need to create a folder named "TRAGET_OS" and put related source files in this folder, here the "TARGET_OS" is a OS name which can significant to identify the target OS.

For the "TARGET_OS" sub directory, you can put all OS-related files in this folder. The Figure 4-2 shows the sample source files that a specified OS should implemented.

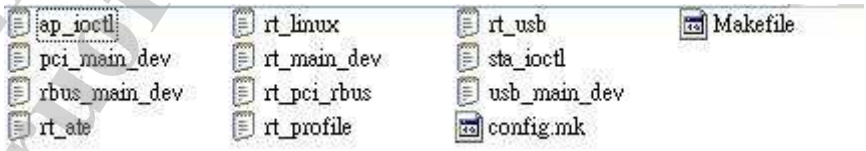

Figure 4-2 Example of target OS Source Layout

The "***Makefile***" and "***config.mk***" are necessary for configuration for the compile flags and RT_WIFI functionalities. For your target OS, you should have this configuration and these compile files.

The "***ap_ioctl***", "***sta_ioctl***", and the "***rt_profile.c***" are the main entry point for Device Configuration and Control Module used to connect between the RT_WIFI and OS. As we mentioned earlier, you should implement a proper mechanism to hook the OS specified mechanism and convert to the format what RT_WIFI can handle.

The "***rt_main_dev.c***" is the main entry point for the Device/Driver Init/Exit Module used to connect the R_WIFI and the network layer of OS.

The "***rt_pci_rbus.c***" and "***rt_usb.c***" are the hardware related files, many network data tasks and receive handler are implemented in these two files. These two file are implemented for PCI/RUBS, and USB, respectively. You shall implement the Ralink-Hardware Interface-dependent APIs depends on the hardware interface type.

The"***pci_main_dev.c***", "***rbus_main_dev.c***" and "***usb_main_dev.c***" are the Hardware-OS dependent files which mainly implemented as the first function entry point of the OS will call to bring the RT_WIFI up. Each one source file is created for one interface. You should implemented the "INTERFACE_main_dev.c" for the target OS.

The "***rt_ate.c***" is a mini driver which takes care of the hardware calibration and manufacture related jobs. Each target OS may use a different mechanism to make the ATE function work. This may need to be implemented for the target OS.

Referring to the description in this section, all OS related files should be put into the "TARGET_OS" folder. The TARGET_OS folder should have these files implemented. After the porting is done, "***Makefile***" and "***config.mk***" can be changed to compile the RT_WIFI for the target OS.