RPU Host Driver

Copyright © Imagination Technologies Limited. All Rights Reserved.

‑This document is confidential. Neither the whole nor any part of the information contained in, nor the product described in, this document may be adapted or reproduced in any material form except with the written permission of Imagination Technologies Limited. Imagination Technologies, the Imagination logo, PowerVR, MIPS, Meta, Ensigma and Codescape are trademarks or registered trademarks of Imagination Technologies Limited. All other logos, products, trademarks and registered trademarks are the property of their respective owners. This document can only be distributed subject to the terms of a Non‑Disclosure Agreement or Licence with Imagination Technologies Limited.‑

Filename : RPU Host Driver  
Version : 1.0.36 Not Issued - Live Document  
Issue Date : 20 Sep 2016  
Author : Imagination Technologies Limited

Contents

[1. Introduction 3](#_Toc462135621)

[2. Software architecture 4](#_Toc462135622)

[2.1. MAC firmware functions 5](#_Toc462135623)

[2.2. Linux wireless Software Architecture 5](#_Toc462135624)

[2.2.1. Linux 5](#_Toc462135625)

[2.2.2. Linux wireless Components 7](#_Toc462135626)

[2.2.3. Host Driver Design 7](#_Toc462135627)

[2.2.4. Driver running sequence 8](#_Toc462135628)

[2.2.5. Driver Code 8](#_Toc462135629)

[3. Hardware Abstraction Layer 9](#_Toc462135630)

[3.1. Message format 9](#_Toc462135631)

[3.2. HAL API 10](#_Toc462135632)

[3.3. HAL for RPU’s HostPort Interface 12](#_Toc462135633)

[3.3.1. HAL on HOST 13](#_Toc462135634)

[3.3.2. HAL on RPU 13](#_Toc462135635)

[4. Driver evaluation 15](#_Toc462135636)

[4.1. Build procedure 15](#_Toc462135637)

[4.2. Loading and Running the Driver 15](#_Toc462135638)

[4.3. Scanning 15](#_Toc462135639)

[4.4. Connecting in Infrastructure Mode 15](#_Toc462135640)

List of Figures

[Figure 1 - Software Architecture – Functional View 4](#_Toc462135641)

[Figure 2 - Software Architecture in Linux 6](#_Toc462135642)

IF{{Internal}}

Document History

| Issue | Date | Changes/Comments |
| --- | --- | --- |
| 1.0.30 | 18 Aug 2016 | External Issue |
| 1.0.33 | 18 Aug 2016 | External Issue |
| 1.0.26 | 20 Sep 2016 | External Issue |
| 1.0.28 | 20 Sep 2016 | External Issue |
| 1.0.31 | 20 Sep 2016 | External Issue |
| 1.0.36 | 20 Sep 2016 | External Issue |

END{{Internal}}

# Introduction

This document describes the functionality and architecture of the RPU reference software driver provided as part of the RPU Platform IP core.

# Software architecture

| MAC  M  A  C  A  P  I  HOST  -  MAC Comms  UCC  Driver  PHY  MAC  configuration  Radio  Driver  HOST  -  MAC Comms  Host Driver  Network Stack  MIPS  UCC  **HOST**  IMG WLAN IP  Licensee developed  /  open source code  PHY API  Comms I  /  F  Comms I  /  F  WLAN API  Operating System  App  Supplicant  RADIO API  IMG  Developed  open source  **RPU**  IMG WLAN IP  Licensee customization  required |
| --- |
| Figure 1 - Software Architecture – Functional View |

Figure 1 shows the functional breakdown of the RPU Software and the layered approach to the design. The processing is split between the MCP, MIPS and Host Processor. The MCP is responsible for all the PHY processing, while the MAC processing is split between the MIPS and the Host Processor. The MIPS is implementing the MAC firmware which encompasses the timing sensitive aspects of the MAC processing as well some additional MAC functions. The Host Processor is implementing the host driver processing primarily responsible for presenting this as a network device to the OS network stack. Additionally host driver also performs some non-time-critical tasks and also the MAC functions which require buffering of packets, For example, when acting as an AP, the host driver is responsible for storing packets for a station which is in power save mode. See the next section for the detailed split of processing between MAC firmware and the host.

## MAC firmware functions

The MAC firmware is main component of the MAC and it runs on the RPU. It handles all the time critical functions of the MAC. The following MAC functions are handled in the MAC firmware:

* Scanning
* Background scan
* Connected Scan
* Connection Tracking in station mode
* Packet encryption and decryption
* EDCA – Channel contention and QoS
* Packet transmission and re-transmisson
* Protection using RTS and CTS-to-Self frames
* Address filtering
* Up to 48 multicast address filtering
* A-MPDU de-aggregation
* Station mode power save – Legacy, uAPSD
* Implicit Block ACK and Immediate Block ACK
* RIFS
* MIMO power save: SM Powersave

## Linux wireless Software Architecture

The linux wireless services two categories of external applications.

The first category of applications view WLAN as any other network interface (like Ethernet) and are only concerned with transferring data (typically TCP/IP traffic) over WLAN. Examples of such applications include ftp, telnet, internet browser, skype, ping, dhcp client etc. These applications can run without any changes once WLAN is brought up and ready for use.

The second category of applications is involved in WLAN interface bring-up and controls the WLAN specific operation of the device. Operating mode (Adhoc or Infrastructure), wireless channel, SSID of the network, security key related information are examples of some WLAN specific parameters.

### Linux

In a Linux OS environment, the host driver integrates with the mac80211 layer present in the Linux kernel as a SoftMAC device to leverage the functionality available in the mac80211 to achieve the functions listed in section 2.2

Linux provides the nl80211 API to develop the second category of applications, i.e those which perform WLAN specific operations. This API is a generic, hardware independent API provided by Linux.

cfg80211 layer in the kernel implements this API and on the other side, it provides APIs to the mac80211 layer.

‘wpa\_supplicant’ is the most widely used second category application for performing WLAN specific operations such as scan and connect to a specific AP, establish security keys and program them into the device. It can also be used to join or create an IBSS network and for launching a SoftAP.

iw, iwconfig, iwlist and iwpriv are examples of other second category applications.

The following links are useful to develop a custom second category application in Linux:

* wpa\_supplicant source and docs: <http://hostap.epitest.fi/wpa_supplicant/>
* nl80211 information: <http://wireless.kernel.org/en/developers/Documentation/nl80211>

Note: nl80211 replaces the older “wireless extensions” API. A complete set of ioclts and events that constitute the wireless extensions can be found in the linux kernel source in the file include/linux/wireless.h

| Kernel  WPA Supplicant  Open SSL  Libraries  Non QOS  Applications  QOS  Applications  cfg  80211  UDP  TCP  S  o  c  k  e  t  B  u  f  f  e  r  s  Host Driver  RPU interface  To MIPS  cfg  80211  \_  ops  WEXT  /  nl  80211  Interface  Non  -  QOS  Data  QOS  Data  Host Processor  IP  Socket API  mac80211 |
| --- |
| Figure 2 - Software Architecture in Linux |

#### 

### Linux wireless Components

The following components comprise the Linux wireless stack.

#### cfg80211 layer

The cfg80211 layer is the wireless LAN framework within the Linux kernel. The cfg80211 implements the API (wireless extensions, nl80211) used to interact with the user space applications. It presents a set of APIs for WLAN drivers which implement the 802.11 MAC functionality and also integrate with the networking layer (TCP/IP stack) of the OS. More information on this can be found at: <http://linuxwireless.org/>

**mac80211 layer**

The mac80211 layer is in the wireless LAN framework within the Linux kernel. It implements the functions listed in 2.2 and provides an API for low level drivers to implement based on the underlying hardware. It also allows certain MAC functions to be offloaded to the hardware. These APIs are utilized by the driver to offload some of functions listed in 2.1 to the hardware.

#### Host Driver

The host driver module integrates with the mac80211 layer and acts as glue layer interfacing mac8011 layer and the firmware.

#### RPU interface

The RPU interface handles transmission and reception of logical messages between host and MAC firmware. This includes host-MAC control messages as well as data traffic. This is also referred to as Hardware Abstraction Layer in the following section 3 of this document.

### Host Driver Design

Figure below shows the host driver architecture. It has three core modules mac80211\_if, umac\_if and hal\_if.

core

mac80211\_if

umac\_if

hal

* mac80211\_if module interfaces with mac80211 module in linux kernel and implements all mac80211 ops.
* Core module handles beacon programming, EDCA parms, reset command proc etc.
* umac\_if module implements framework to encap/decap messages to/from LMAC firmware
* hal module implements communication with RPU.

Host to firmware messages are communicated with CMD\_TX, CMD\_CHANNEL etc

Firmware to host messages are communicated with EVENT\_RX, EVENT\_TX\_DONE etc

### Driver running sequence

When Driver loads into the kernel, hal\_init() is called. Hal\_init() requests IO memory regions, initializes TX, RX tasklets and registers hal\_irq\_handler and finally registers mac80211\_ops.

Some of the driver config parms are read from proc/uccp420/parms and stored into wifi\_params struct.

When user creates virtual interface, add\_interface() op is called and programs firmware with vif\_index, mac\_addr and mode (STA/AP/ADHOC) with IF\_ADD command.

On removal of virtual interface, remove\_interface() op is called and sends IF\_REMOVE command to firmware.

When the interface is activated, start() op gets called and driver issues CMD\_RESET with value LMAC\_ENABLE to firmware and waits for EVENT\_RESET\_COMPLETE. Programs RX\_buffer cmds to hw, inits TxQs.

When interface is deactivated stop() op gets called and driver issues CMD\_RESET with value LMAC\_DISABLE to firmware and waits for EVENT\_RESET\_COMPLETE and releases all pending outstanding cmds.

Config() op takes care of channel info programming.

Tx() op supports transmission of frames. Mac80211 gives dot11 pkt skb (either mgmt/data) to driver through this op. Driver allocs Tx\_buf and submits to firmware through CMD\_TX. Completion of this pkt is handled through EVENT\_TX\_DONE.

Firmware indicates receive pkts through EVENT\_RX. Driver extracts skb from Rx\_buf and indicates to mac80211 through ieee80211\_rx().

### Driver Code

Driver global data stored in struct mac80211\_dev {}.

Config parms are stored in struct wifi\_params {}

Hal code is in **hal\_hostport.c**

umac\_if code is **umac\_if.c**

Mac80211\_if code is in **80211\_if.c**

Core code is in **core.c**

**hal\_umac\_if.h** is the common header file (contains struct defines) between Host/RPU HAL code.

Driver source tree:

* rpu\_host\_driver

|- src Source and Makefile

|- inc Header files

# Hardware Abstraction Layer

A hardware abstraction layer (HAL) is present on the Host and the RPU which abstracts the hardware interface between Host and RPU. A HAL implementation has to implement the APIs described in section 3.2. The message format for messages exchanged via the HAL is described in section 3.1

## Message format

The format of messages exchanged between Host and RPU is given in the diagram below:

7

0

31

23

15

Command/Event Header

|  |  |
| --- | --- |
| HAL\_PRIV\_DATA | |
| Queue Number | Descriptor Number |
| Payload length | |
| Command/Event Data | |

Messages from host to RPU are called commands while those from RPU to host are called events. Every message has a reserved area at the top for use by the HAL. The size of this area is decided at compile time by the HAL in use. For example, the Hostport HAL may require 8 bytes here while a different HAL may not require this at all. Following the HAL reserved area, every command/event has 8 bytes in which the descriptor number, queue number and payload length fields are encoded as shown in the figure above. These fields are valid only when the command/event has a payload associated with it. The payload pointer itself is passed to the HAL as one of the arguments of the hal\_send() API. (See section 3.2) This is indicated when the descriptor number is encoded as a value other than 0xFFFF. The queue number, descriptor number and payload length are related to the payload and this information can be used by the HAL to maintain a mapping between descriptors and actual payloads. Section 3.3 gives an example of how this information is used by the HAL. The actual command or event data follows the payload length field and its length can be variable. The little-endian format is assumed for all the commands and the events.

|  |  |  |
| --- | --- | --- |
| Field Name | Length (in Bytes) | Description |
| HAL\_PRIV\_DATA | HAL Dependent | For use by the HAL |
| Descriptor Number | 2 | Descriptor ID – If not 0xFFFF, indicates the descriptor ID of a command which has a payload |
| Queue Number | 2 | Queue Num: Indicates which queue this packet belongs to. Valid only when Descriptor Number is not equal to 0xFFFF |
| Payload length | 4 | The length of payload. Valid only when Descriptor Number is not equal to 0xFFFF |
| Command/Event Data | Command/Event dependent | Contains the actual Host<->RPU message content |

## HAL API

The HAL layer on both RPU and Host has to implement the following APIs. On the host, this corresponds to ‘RPU Interface’ in Figure 2.

None of these APIs can put the caller to sleep unless stated explicitly in the description

|  |  |
| --- | --- |
| **Name** | **hal\_init** |
| **Arguments** | void |
| **Returns** | void |
| **Description** | This function is called at startup before any messages are sent/recieved. The HAL can perform any local initializations when this function is called. This API can sleep. |

|  |  |
| --- | --- |
| **Name** | **hal\_deinit** |
| **Arguments** | void |
| **Returns** | void |
| **Description** | This function is called when the services of HAL are no longer needed. The HAL can free up resources in this call. This API can sleep. |

|  |  |
| --- | --- |
| **Name** | **hal\_init\_buffers** |
| **Arguments** | int num\_tx\_buffers, rx\_buffers\_2k, rx\_buffers\_12k |
| **Returns** | int |
| **Description** | This function is called to tell the HAL that it has to prepare for sending and receiving WLAN data packets to/from the firmware. This is required only on the host side. Depending on the actual physical interface, some HALs may do nothing in this API. This API can sleep.  Returns 0 on success and non-zero on failure |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Name** | **hal\_send** | | | | |
| **Arguments** | **Name** | **I/P** | **O/P** | **I/O** | **Description** |
| nw\_buff\_t \*msg | Y | N | N | This contains the network buffer to be sent to the destination module. This can be NULL when invoked in the RPU HAL, but can never be NULL when invoked in the host side HAL |
| unsigned char dst\_module\_id | Y | N | N | Module ID of the destination module.  On the host, this can be set to UMAC or LMAC  On the RPU, this can be set to LMAC or UMAC or HOST. |
|  | unsigned char src\_module\_id | Y | N | N | Module ID of the sending module.  On the host, this is set to HOST  On the RPU, this can be set to LMAC or UMAC |
|  | nw\_buff\_t \*payload | Y | N | N | On the Host, if a payload is associated with this command, this contains a pointer to the network buffer containing the payload.  On the RPU, when the ‘msg’ argument is NULL, this argument contains a pointer to the network buffer containing the payload + RX control information. |
| **Returns** | int | | | | |
| **Description** | This function is to send a message to the destination module. Returns 0 on success and 1 on failure.  On the host, nw\_buff\_t depends on the native OS. For example, this can be simply struct sk\_buff in case of Linux.(include/skbuff.h)  On the RPU, nw\_buff\_t is defined withint the memory management(MM) layer an MM layer provides the APIs to manipulate this buffer. (Refer to mm.h in the package) | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Name** | **hal\_register\_handler** | | | | |
| **Arguments** | **Name** | **I/P** | **O/P** | **I/O** | **Description** |
| msg\_handler | Y | N | N | This specifies the handler to be called for messages received for this module |
|  | unsigned char module id | Y | N | N | ID of the module which is invoking this API to register its handler. Can be HOST, UMAC or LMAC |
| **Returns** | void. | | | | |
| **Description** | This function is used by a module to register its message handler with the communications layer. The prototype of the function pointer is as follows:  typedef unsigned long (\*msg\_handler) (nw\_buff\_t \*msg, unsigned char module\_id);  The handler should return 0 in all cases other than fatal/ non recoverable exceptions | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Name** | hal\_map\_tx | | | | |
| **Arguments** | **Name** | **I/P** | **O/P** | **I/O** | **Description** |
|  | descriptor\_id | **Y** | **N** | **N** | Descriptor Identification of the TX buffer. |
|  | **Pkts** | **Y** | **N** | **N** | Pointer to the head of linked list of packets to be sent to RPU |
| **Returns** | void. | | | | |
| **Description** | This API is invoked prior to invoking the ‘send’ API for messages which have payloads associated with them. The HAL performs a ‘dma\_map\_single’ for each skb in ‘pkts’ and updates the physical address of each ‘skb’ in a predefined location of GRAM. The predefined location can be decided at compile time and is based on descriptor ID and packet count within the descriptor ID. This information is used by the UCCP HAL to fetch the packets when requested by the MAC layer. | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Name** | hal\_unmap\_tx | | | | |
| **Arguments** | **Name** | **I/P** | **O/P** | **I/O** | **Description** |
|  | descriptor\_id | **Y** | **N** | **N** | Descriptor Identification of the TX buffer. |
| **Returns** | void. | | | | |
| **Description** | This API is invoked when driver receives confirmation from the UCCP that a packet has been transmitted. In this API, the HAL can perform ‘dma\_unmap\_single’ for each packet in that descriptor ID. | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Name** | **hal\_stream** | | | | |
| **Arguments** | **Name** | **I/P** | **O/P** | **I/O** | **Description** |
| unsigned char queue\_num | Y | N | N | This specifies the queue number from which the packets are to be fetched |
|  | unsigned char num\_pkts | Y | N | N | This specifies the number of packets to be fetched |
|  | unsigned int desc\_list[] | Y | N | N | Array of descriptors whose size is equal to ‘num\_pkts’ |
| **Returns** | void. | | | | |
| **Description** | This is required only on RPU. This function is invoked by the RPU firmware to request to HAL to fetch/DMA the buffers corresponding to the descriptor list provided in the argument. | | | | |

## HAL for RPU’s HostPort Interface

This part of the document describes a possible way to implement the HAL layer assuming a host port communication mechanism between HOST and RPU

The physical address of the start of shared memory region in GRAM is known to the HALs on both RPU and HOST at compile time. The size of HAL\_RPIV\_DATA in case of HostPort comms is 8 bytes. The first 4 bytes are used to mark a message as a regular command/event or HAL internal message to be interpreted by the HAL only.

### HAL on HOST

On the host, in the hal\_init() function, the HAL takes care of mapping the shared memory regions, requesting and registering the interrupt handler for RPU and allocating memory for any internal data structures.

In the hal\_init\_rx\_buffs() function, the HAL allocates the requested number of RX buffers and maintains a table of RX buffer vs descriptor number. It then sends this mapping information to the RPU HAL by forming a HAL internal message (The first four bytes of HAL\_PRIV\_DATA are set to 0 to indicate this).

In the hal\_send() function, the HAL first sets the first four bytes of the HAL\_PRIV\_DATA field to 0xffffffff to indicate a regular message. Next, the HAL checks if the payload is not NULL. If so, includes the pointer to the payload in the 2nd four bytes of the HAL\_PRIV\_DATA field. It then checks for the interface ready status bit and after it is ready, it finally copies this message to a location contained in a fixed shared memory offset. It then kicks the appropriate register to raise an interrupt to RPU.

In the interrupt handler, the HAL first checks the HAL\_PRIV\_DATA to determine whether the message is a HAL internal message or a regular message.

If it’s a regular message, it allocates a buffer large enough to hold this message, copies the contents of message and then invokes the host’s message handler function. It also marks the buffer as free in the shared memory.

If it’s a HAL internal message, it looks at the RX descriptor numbers present in that message. For each descriptor, it looks up the corresponding buffer in the mapping table and dispatches it to the host’s message handler. Further, it allocates a new buffer and sends a HAL internal message to the RPU HAL to refresh this RX descriptor.

### HAL on RPU

On the RPU, in the hal\_init() function, the HAL takes care of initializing host port related registers, registering interrupt handler with MeOS and allocating memory for any internal data structures. It then requests a buffer from ‘MM’ layer by setting pool ID to MEM\_HAL.

In the hal\_send() function, if the ‘msg’ argument is **not equal to** NULL, the HAL first sets the first four bytes of the HAL\_PRIV\_DATA field to 0xffffffff to indicate a regular message. It then copies the pointer to this message into a fixed shared memory offset. It then checks for interface ready status bit and after it is ready, it kicks the appropriate register to raise the interrupt to the host.

In the hal\_send() function, if the ‘msg’ argument is **equal to** NULL, for each buffer in the payload list, it uses a free RX descriptor and kicks off a DMA job to transfer the payload buffer to the address corresponding to the descriptor. Next, the HAL creates a new HAL internal message by setting first four bytes of the HAL\_PRIV\_DATA field to 0x0 and populating the message with the RX descriptor list which was used for transfering the payload. It then dispatches this command to the HAL on the host, like any other command.

In the interrupt handler, the HAL first checks the HAL\_PRIV\_DATA to determine whether the message is a HAL internal message or a regular message. If it’s a regular message, it further looks at the descriptor number field of the header. If it’s not equal to 0xFFFF, it updates its mapping table to store the tuple <queue, descriptor, payload\_address>. (This information is used later when the hal\_stream() API is invoked by the RPU firmware.) After this step, it dispatches the message by calling the appropriate message handler

If it’s a HAL internal message, it looks at the RX descriptor information and updates it’s RX descriptor table.

In the hal\_stream() function, it submits a job to the DMA engine based on the descriptor list and the mapping table it had created when processing the commands with a valid descriptor (number not equal 0xFFFF).

# Driver evaluation

The driver source code can be evaluated on IMG MIPS reference platform running Linux OS.

## Build procedure

Please extract rpu driver source on a Ubuntu 14.04 Linux 64 bit OS x86 machine.

Take either mips GNU cross compiler or ARM cross compiler based on Host processor architecture.

1. Issue make command as below  
   **#<rpu\_host\_driver>$ make CC=<The compiler/cross compiler for the host>**
2. uccp420wlan.ko will be generated.

## Loading and Running the Driver

The WLAN driver is loaded and configured using the following steps –

1. Load the Linux device driver, specifying the MAC address for the unit, e.g. 0x001122334455

**#insmod uccp420wlan.ko mac\_addr=”001122334455”**

1. Optionally, adjust any driver specific parameters via the /proc pseudo filesystem interface. Initially, the current parameters can be inspected via  
   **#cat /proc/uccp420/params**
2. Bring the driver “up”  
   **#ifconfig wlan0 up**

The WLAN driver may be unloaded via the following steps –

1. Unload the driver from the kernel

**#rmmod uccp420wlan**

## Scanning

Having completed the steps in Section 4.2, available networks can be found using the Wireless Tool iw as follows

**#iw dev wlan0 scan**

## Connecting in Infrastructure Mode

Connecting to an Access Point (AP) is best achieved using wpa\_supplicant. This tool requires a configuration script for each AP you wish to connect to. Section **Error! Reference source not found.** details some simple wpa\_supplicant configuration scripts for different encryption schemes. Firstly, load and run the driver as described in Section 4.2.

1. Associate with an AP using an appropriate configuration file. Configuration files for connecting in plain mode, wep and wpa psk are available in /usr/img (e.g. plaintext.conf)

**#wpa\_supplicant –iwlan0 –Dnl80211 –B –c <*conf\_file*>**

1. Observe the connection status via the console output using dmesg  
   **#dmesg –c**

*wlan0: authenticate with 00:0c:43:44:06:fa (try 1/3)  
wlan0: authenticated  
wlan0: associate with 00:0c:43:44:06:fa (try 2/3)*

*wlan0: RX AssocResp from 00:0c:43:44:06:fa (capab=0x401 status=0 aid=1)  
wlan0: associated  
  
ADDRCONF(NETDEV\_CHANGE): wlan0: link becomes ready*

1. Obtain an IP address from a DHCP server (if present)  
   **#killall dhclient**

**#dhclient wlan0**

1. Alternatively, specify an IP address explicitly, e.g. 192.168.1.100  
   **#ifconfig wlan0 192.168.1.100**
2. Test the connection by pinging another station or the AP itself, e.g. 192.168.1.1  
   **#ping 192.168.1.1**