WLAN  
Host to RPU Interface

Copyright © Imagination Technologies Limited. All Rights Reserved.

‑‑This document is strictly 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 : WLAN.Host to RPU Interface.docx  
Version : 1.0.105 Not Issued - Live Document  
Issue Date : 08 Feb 2017  
Author : Imagination Technologies Limited

Contents

[1. Introduction 4](#_Toc474339892)

[2. Configurations / Split Models 4](#_Toc474339893)

[3. Host Packet Buffering 4](#_Toc474339894)

[4. Hardware Interface 5](#_Toc474339895)

[4.1. Bus Slave Access 5](#_Toc474339896)

[4.2. Bus Master Access 6](#_Toc474339897)

[4.3. Control/Status signals 6](#_Toc474339898)

[5. HAL Control Message Interface 6](#_Toc474339899)

[5.1. HAL API 6](#_Toc474339900)

[5.2. Hostport HAL 6](#_Toc474339901)

[5.2.1. Hostport register usage for commands (Host to RPU direction) 7](#_Toc474339902)

[5.2.2. Hostport register usage for events (RPU to Host direction) 7](#_Toc474339903)

[5.2.3. Message format 9](#_Toc474339904)

[5.2.4. HAL Initialization 10](#_Toc474339905)

[5.2.5. Sending HAL messages (commands) 10](#_Toc474339906)

[5.2.6. Receiving HAL messages (events) 11](#_Toc474339907)

[6. Other (non-HAL) Interfaces 11](#_Toc474339908)

[6.1. Cold Boot and Patch Downloading 11](#_Toc474339909)

[6.1.1. Patch Downloading (Step 6) 12](#_Toc474339910)

[6.1.2. Setting Boot Vector (Step 7) 12](#_Toc474339911)

[6.1.3. Boot MIPS CPU (Step 8) 12](#_Toc474339912)

[7. Driver Message Sequences 12](#_Toc474339913)

[7.1. Thin MAC Initialization Sequence 12](#_Toc474339914)

[7.1.2. uccp420wlan\_prog\_txpower (UMAC\_CMD\_TX\_POWER) 13](#_Toc474339915)

[7.1.3. uccp420wlan\_prog\_vif\_ctrl (UMAC\_CMD\_VIF\_CTRL) 13](#_Toc474339916)

[7.2. Basic Packet TX Sequence 13](#_Toc474339917)

[7.2.1. uccp420wlan\_prog\_tx (UMAC\_CMD\_TX) 14](#_Toc474339918)

[7.2.2. uccp420wlan\_tx\_complete (UMAC\_EVENT\_TX\_DONE) 14](#_Toc474339919)

[7.3. Basic Packet RX Sequence 14](#_Toc474339920)

[8. Power Management 14](#_Toc474339921)

[8.1. RPU Power States 14](#_Toc474339922)

[8.2. HP\_READY/RPU\_READY indication 15](#_Toc474339923)

[8.3. Host/HAL Implications 15](#_Toc474339924)

[8.3.1. Awakening RPU 15](#_Toc474339925)

[8.3.2. Handling RPU\_READY indication 16](#_Toc474339926)

[8.4. LMAC Operational States 16](#_Toc474339927)

[8.4.1. State Descriptions 16](#_Toc474339928)

[8.4.2. Sleep strategy for operational states 16](#_Toc474339929)

[8.5. Sleep Controller SOC Customization 17](#_Toc474339930)

[8.5.1. Calculation of wakeup time 17](#_Toc474339931)

[8.5.2. HAL Message Details 18](#_Toc474339932)

[8.5.3. General flow of firmware sleep operation 18](#_Toc474339933)

[9. References 19](#_Toc474339934)

[10. FAQ’s 19](#_Toc474339935)

List of Figures

**No table of figures entries found.**

IF{{Internal}}

Document History

| Issue | Date | Changes/Comments |
| --- | --- | --- |
| 1.0.39 | 13 Oct 2016 | External Issue. |
| 1.0.88 | 14 Dec 2016 | External Issue |
| 1.0.97 | 03 Feb 2017 | Updated with section on "Sleep Controller SOC Customization" |
| 1.0.102 | 07 Feb 2017 | Updated to make corrections as per PP# 103712 |
| 1.0.105 | 08 Feb 2017 | Removed API descriptions from the document and added a reference to the same |

END{{Internal}}

# Introduction

The Whisper Low-Power WiFi (LPW) System implements LPW 11n 1x1 as a convenient IP that consists of RF, AFE, baseband PHY and MAC.

The IP is an 802.11 b/g/n compliant Wi-Fi solution with a focus on lowering active power consumption, leakage power consumption, ensuring low duty cycle operation and providing the lowest area.

The key functions supported by the LPW System are:

* 802.11 b/g/n compliancy.
* SISO.
* 2.4 GHz Band.
* 802.11n bandwidth of 20 MHz.
* PHY data rates up to MCS7.

# Configurations / Split Models

LPW has two distinct hardware configurations (1 or 2 embedded MCUs) which support several distinct MAC split models. The split models affect the allocation of function between the CPUs and therefore also the software interaction between the host and RPU.

|  |  |  |  |
| --- | --- | --- | --- |
| Functional Block | Thin MAC Model | Full MAC Model | Full+S MAC Model |
| Number of MCUs in RPU | 1 | 2 | 2 |
| TCP/IP Stack | Host | Host | Host |
| WPA Supplicant | Host | Host | MCU2 |
| UMAC | Host | MCU2 | MCU2 |
| LMAC | MCU1 | MCU1 | MCU1 |

# Host Packet Buffering

The RPU follows an “inline model” with respect to packet buffering, i.e. packet buffers are stored in host memory, not in RPU memory.

In the transmit direction, packets will remain in host memory until the wireless channel becomes clear, at which point the buffer will be **immediately** transferred via DMA through the crypto block in the RPU and onto the air.

In the receive direction, packets are only briefly buffered within the RPU before getting transferred to an available host buffer.

The host can determine the amount of TX and RX buffering to use at run-time based on the system use case. This gives the host the ability to control the trade-off between packet memory footprint and performance. The table below provides recommended buffer configurations for several different use cases.

| Description/target | 20mhz 1x1 11n certification (agg & wmm) | Basic agg (mpdu=8), wmm | Basic agg (mpdu=8), no wmm | no agg, wmm | no agg, no wmm |
| --- | --- | --- | --- | --- | --- |
| Theoretical UDP/TCP tput (Mbps) | 60.7 / 54.3 | 56.9 / 48.5 | 56.9 / 48.5 | 30.4 / 19.5 | 30.4 / 19.5 |
| Number aggregation sessions | 1 | 1 | 1 | 0 | 0 |
| Max MPDUs within AMPDU | 16 | 8 | 8 | 1 | 1 |
| WMM (QoS) supported | yes | yes | no | yes | No |
| TX buffers | 78 | 46 | 36 | 14 | 4 |
| Small (1.5KB) RX buffers | 64 | 32 | 32 | 16 | 16 |
| Large (8 KB) RX buffers | 4 | 4 | 4 | 0 | 0 |
| TX memory (KB) | 117 | 69 | 54 | 21 | 6 |
| RX memory (KB) | 128 | 80 | 80 | 24 | 24 |
| Total memory (KB) | 245 | 149 | 134 | 45 | 30 |

The LMAC firmware is compiled with a maximum number of TX and RX descriptors which the host cannot exceed. The following outlines the dimensions and the corresponding sizes.

|  |  |
| --- | --- |
| Dimension | Size |
| Maximum Number of TX Requests | 12 |
| Maximum number of MPDUs per TX AMPDU | 16 |
| Maximum number of small RX buffers (for MPDUs) | 256 |
| Maximum number of large RX buffers (for AMSDUs) | 16 |

# Hardware Interface

The hardware interface between RPU and host is a system bus and control signals. Both the host and the RPU are capable of acting as bus masters. The RPU is capable of asserting an IRQ to interrupt the host.

Further hardware details of this bus interface can be found in [1] and [2].

## Bus Slave Access

The host is capable of accessing registers and memories within the RPU in many different ways. Some of this access is for diagnostics and is not strictly required for normal operation. The core areas which must be accessed by the host are:

* RPU SysBus (and Peripheral Bus) Registers: These registers allow the host to control interrupts from the RPU, to use the Host-MCU Communications Interface, and to access the Sleep Controller.
* GRAM (packed view): This contains the shared memory area used by the HAL message interface.

These two areas are not contiguous in the RPU host address map, so the driver maintains a separate base address for each of these areas.

The full host address map can be found in [1] in the section titled “Host/Meta MCU and Meta JTAG Address Map”. (Note: this host map applies to both MIPS and Meta RPU configurations. The SysBus registers are typically located at address 0x02000000 and the packed view of GRAM is typically located at 0xB7000000.)

As an example, the IMG FPGA implements the following mapping function in RTL outside the IP which results in a 24-bit addressable slave space.

|  |  |  |
| --- | --- | --- |
| RPU Address Region | RPU Base Address | Maps to  Host Base Address |
| GRAM Type A : double, left aligned | 0xB4000000 | 0x00000000 |
| GRAM Type B : sign extended, right aligned | 0xB5000000 | 0x00400000 |
| GRAM Type C : complex, left aligned | 0xB6000000 | 0x00800000 |
| GRAM Type D : packed, left aligned | 0xB7000000 | 0x00C00000 |
| SysBus / PBus | 0x02000000 | 0x00E00000 |

It should be noted that the Core Memory for the MCUs are not directly mapped to the host. MCU core memory is accessed indirectly via SysBus registers.

## Bus Master Access

The RPU is capable of acting as a bus master and has 26-bits (64 MB) of addressable range. The RPU uses master access to transfer packets to and from host memory.

## Control/Status signals

The RPU has several control/status signals which are used by the host software. Typically, these signals are mapped to an SOC register for the host software to access. Further details of these signals can be found in [2].

* RESETN – this asserts a hard reset to the RPU
* WAKEUP\_NOW – this triggers the RPU to awaken
* HP\_READY (sleep\_hp\_rpu\_ready) – this indicates when the RPU is awake (powered and firmware has booted)

# HAL Control Message Interface

Most of the interaction between the RPU MAC firmware and the host occurs using HAL control messages. (The acronym HAL stands for Hardware Abstraction Layer). The HAL layer is a module within the host driver which is responsible for transporting control messages between the host and the RPU MAC firmware. Messages flowing from host to RPU are known as “Commands”. Messages flowing from RPU to host are known as “Events”. These control messages also form the basis for signalling data packet transfer.

## HAL API

The HAL module within the reference driver provides APIs which allows the upper parts of the host driver to control the HAL and the RPU.

The API’s and associated data structures are detailed in [3].

## Hostport HAL

The transport method used by the LPW HAL is known as the “hostport” method. This method uses an area of GRAM as a shared memory mailbox for storing messages as they are transferred between host and firmware. The reference driver software provides an implementation of the HAL.

The RPU hardware provides registers for facilitating communication between host and firmware. In particular, they provide a method for the sending CPU to interrupt the destination CPU. These registers are also known as the “host port” registers. The RPU hardware supports multiple groups/channels of these registers to support communications between multiple pairs of CPUs.

The table below describes the registers for Hostport #1.

|  |  |
| --- | --- |
| RPU Register Name | Name in driver code |
| UCCP\_CORE\_HOST\_TO\_MTX\_CMD | HOST\_TO\_UCCP\_CORE\_CMD |
| UCCP\_CORE\_MTX\_TO\_HOST\_CMD | UCCP\_CORE\_TO\_HOST\_CMD |
| UCCP\_CORE\_HOST\_TO\_MTX\_ACK | HOST\_TO\_UCCP\_CORE\_ACK |
| UCCP\_CORE\_MTX\_TO\_HOST\_ACK | UCCP\_CORE\_TO\_HOST\_ACK |

See the “Host-MCU Communications” section of [1] for further general information about these registers.

In addition to generating an interrupt, the registers can convey 31 bits of information along with the interrupt.

Below is a description of how the hostport registers used within RPU and HAL. Counters are used to ascertain the interrupt source.

### Hostport register usage for commands (Host to RPU direction)

UCCP\_CORE\_HOST\_TO\_MTX\_CMD: The current command count (a sequence number) is written to this register once the CMD content is copied in to the GRAM. Any write to this register triggers an interrupt to the MTX in RPU.

UCCP\_CORE\_MTX\_TO\_HOST\_ACK: Writing to this register clears the interrupt. Once MTX processes the CMD, it allocated another buffer for next CMD and places it in the CMD\_START address and then writes to this register.

### Hostport register usage for events (RPU to Host direction)

UCCP\_CORE\_MTX\_TO\_HOST\_CMD: The current event\_count (a sequence number) in RPU\_HAL is written to this register once the EVENT is ready. Any write to this register triggers an interrupt to the HOST.

UCCP\_CORE\_HOST\_TO\_MTX\_ACK: Writing to this register clears the interrupt. Once HOST processes the EVENT, copies the event information, it writes to this register to clear the interrupt.

The location of the shared memory mailbox within GRAM must be known to both host and firmware at compile time, using the definitions in the table below.

#### Shared Memory Layout

The layout should be same for both Host and RPU.

|  |  |
| --- | --- |
| Shared Mem Range | Region |
| SH\_MEM to SH\_MEM + 60 | CMD\_START |
| CMD\_START to CMD\_START + 60 | EVENT\_START |

**HAL Compile-time Parameters**

#### These parameters are used by both host and RPU and hence they must be same.

|  |  |  |
| --- | --- | --- |
| Name | Value | Description |
| HAL\_SHARED\_MEM\_OFFSET | 0x0FFC | offset of shared memory region within GRAM (packed view) |
| HOSTPORT NUMBER | 0-3 | This number decides the offsets to set of hostport registers. |

### 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 | |
| Identifier | |
| Length | |
| More commands | |
| 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 internal use by the HAL. Following the HAL reserved area, every command/event has 8 bytes in which the descriptor number, queue number and payload length, ID, length and more\_cmd\_data 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. 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. 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 | 8 | For internal 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 |
| Length | 4 | Size of Command/Event Payload |
| More\_cmd\_data | 4 | TX: Indicates fragmented command and presence of more fragments.  RX: 0 🡺 Non-QoS 1🡺 Qos. |
| Command/Event Data | Command/Event dependent | Contains the actual Host<->RPU message content |

### HAL Initialization

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\_bufs() 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 0xffffffff to indicate this).

The mapping table contains information as shown below:

**Descriptor ID: DDR Physical Address : DDR Virtual Address**

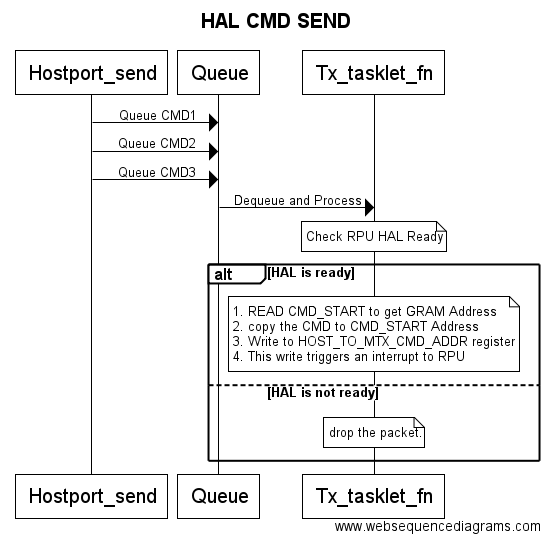
1 0xAABBCCDD 0xAABBCCDD

2 0XBBBBCCDD 0XBBBBCCDD

### Sending HAL messages (commands)

In the hal\_send() function, the HAL first sets the first four bytes of the HAL\_PRIV\_DATA field to 0 to indicate a regular message. It then checks for the interface ready status bit and after it is ready, it reads CMD\_START to get the address of the buffer to copy the CMD and it finally copies this message.

The figure below illustrates the flow of hal\_send operation in the reference HAL module.



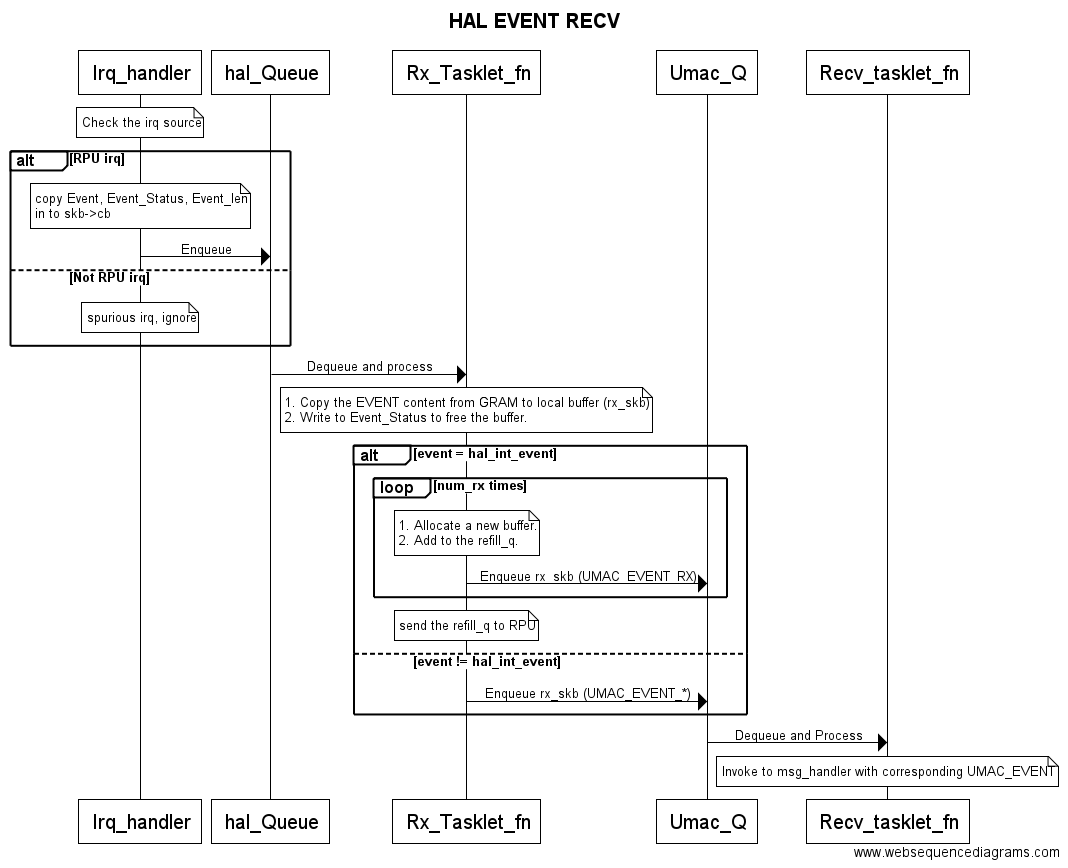
### Receiving HAL messages (events)

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.

The figure below illustrates the flow of event reception in the reference HAL module.



# Other (non-HAL) Interfaces

## Cold Boot and Patch Downloading

The initial cold boot of the RPU is initiated by the host. The general sequence is outlined as 9 steps in the “System Start” section of [2]. The first 5 steps in the sequence are hardware-related and are not covered here. After the first 5 steps have been completed, the RPU MIPS CPUs will be powered and waiting in a state ready to be programmed. The default POWER\_ENABLES settings in the Sleep Controller have the BOOT\_IN flag set to zero. The effect of BOOT\_IN==0 is the CPUs will be blocked on WAIT instructions and not executing any code at the end of step 5.

### Patch Downloading (Step 6)

The firmware ROM has the capability to be “patched” with an additional RAM patch image. Performing this procedure will allow the ROM-based firmware to detect and incorporate the patch image. The patch image will be in the proprietary BIMG format, which contains relocation information.

First, the patch image must be loaded into MIPS core memory, specifically the retention memory (i.e. RAM A) The core memory is not directly mapped to the host address space, so it must be written indirectly through System Bus registers: MIPS\_MCU\_SYS\_CORE\_MEM\_CTRL and MIPS\_MCU\_SYS\_CORE\_MEM\_WDATA which are further described in [1].

Second, the address of the patch image within core memory must be written into the UCC\_SLEEP\_CTRL\_DATA\_0 register of the Sleep Controller. The firmware boot loader reads this register to detect the presence of the patch image (non-zero value indicates patch is present).

### Setting Boot Vector (Step 7)

For the cold boot procedure, the MIPS boot vector for the ROM image must be set by the host. This due to the fact that Sleep Controller has powered the MIPS CPU with BOOT\_IN set to zero, and host must now prepare to trigger the boot. Normally, this is programmed with a jump to the boot vector of the ROM, so this is a fixed sequence of register writes.

|  |  |
| --- | --- |
| Register | Value |
| MIPS\_MCU\_BOOT\_EXCP\_INSTR\_0 | 0x3c1a8000 |
| MIPS\_MCU\_BOOT\_EXCP\_INSTR\_1 | 0x275a0000 |
| MIPS\_MCU\_BOOT\_EXCP\_INSTR\_2 | 0x03400008 |
| MIPS\_MCU\_BOOT\_EXCP\_INSTR\_3 | 0x00000000 |

### Boot MIPS CPU (Step 8)

A register write is now needed to trigger the MIPS to boot.

The register MIPS\_ECU\_CONTROL should be written with value 0.

# Driver Message Sequences

## Thin MAC Initialization Sequence

The primary purpose of the initial sequence of HAL commands is to configure key parameters and bring the RPU into an operational state.

* + 1. **uccp420wlan\_core\_init**

This is the very first function that is called after loading the Firmware and in turn does the following initializations:

1. ***hal\_ops.start***

This enables the interrupt line from RPU to Host.

1. ***hal\_ops.init\_bufs***

This allocates the RX buffers in the host and programs the buffers to the RPU

1. ***uccp420wlan\_prog\_reset (UMAC\_CMD\_RESET)***

This will primarily:

* Initialize the MAC, BB and RF modules in RPU.
* Calibrate sample channels across 2.4 and 5 GHz bands for scanning.

1. ***uccp420wlan\_prog\_btinfo(UMAC\_CMD\_BT\_INFO)***

This programs the Bluetooth state (enabled/disabled) to the RPU.

### uccp420wlan\_prog\_txpower (UMAC\_CMD\_TX\_POWER)

This function configures the default TX power to the RPU.

### uccp420wlan\_prog\_vif\_ctrl (UMAC\_CMD\_VIF\_CTRL)

This function configures the VIF index, mac address, interface type (for e.g. AP/STA/IBSS etc.) and operation (add/remove) to the RPU.

## Basic Packet TX Sequence

The RPU is capable of queueing up to a certain number of TX descriptors. Each TX descriptor can accommodate one MPDU frame or AMPDU frame with maximum of 16 AMPDU sub frames. The Host driver has flexibility to use these TX descriptors for any WiFi functionality.

The TX descriptors are of 2 types:

*Reserved*: These are a set of (M) descriptors assigned per WMM AC (access category) and also for a separate Beacon/Broadcast queue.

*Spare*: These are a global set of (N) descriptors which can be used across any queue.

A token bucket algorithm is used for assigning a descriptor for a TX MPDU. If a packet is unable to find a descriptor, the packet is queued in the driver’s corresponding pending queue and if the drivers pending queue gets filled up, the TCP/IP stack is informed to stop sending further packets. Once enough buffers are freed up the TCP/IP is informed to resume the packet transmission.

To support WMM/WiFi certification, it is recommended to use 12 TX descriptors (reserved + spare), split as below.

BK: Background access category: Descriptor 0 and 5

BE: Best effort access category: Descriptor 1 and 6

VI: Video access category: Descriptor 2 and 7

VO : Voice access category : Descriptor 3 and 8

Beacon: Beacon/Broadcast category: Descriptor 4 and 9.

Spare: These descriptors can be shared across all above categories: Descriptor 10 and 11.

Each descriptor has control structure which can contain information of all ampdu sub frames. Each descriptor can accommodate maximum of 16 ampdu frames. Following rules have to apply while aggregating frames to tx descriptors:

1. A1, A2 and A3 address fields of AMPDU sub frames has to be same.
2. All AMPDU sub frames have to be same access category.
3. First AMPDU sub frame and last ampdu sub frame sequence numbers difference cannot be more than 64.

RPU processing: Following are sequence of steps involved in TX processing.

1. Queue tx commands depending on Queue number (queue number is different than descriptor number).
2. Process TX command and arm edca engine if it is not already armed for this access category.
3. Schedule transmission as soon as you get event from EDCA. EDCA is a hardware module which is used to for Channel sensing.
4. Wait for TX Complete for broadcast fames or ACK/BA for unicast frames.
5. Generate TX done event after successful transmission.
6. Host can reuse this descriptor after receiving tx done event from RPU.

### uccp420wlan\_prog\_tx (UMAC\_CMD\_TX)

This function programs the control information for 1 AMPDU, DDR Addresses to LMAC for transmission. This descriptor is enqueued within the LMAC. When the wireless channel is ready for transmission, the RPU will immediately read the packet from host memory via DMA and transmit the packet onto the air.

### uccp420wlan\_tx\_complete (UMAC\_EVENT\_TX\_DONE)

This event indicates that the transmission of the MPDU’s as a part of UMAC\_CMD\_TX in completed, this returns the control information like the status of the transmission, the rate and retries at which each MPDU is transmitted.

## Basic Packet RX Sequence

RPU uses HAL\_INTERNAL\_CMD’s to indicate the presence of incoming packets to host.

Each HAL\_INTERNAL\_CMD can transport up to 16 RX packets.

When an packet is received on the air, the RPU will write the packet into host memory via DMA. The RPU will then generate the HAL\_INTERNAL\_CMD message. Upon receiving the HAL\_INTERNAL\_CMD, using the mapping table, the host gets the virtual address pointer of the payload. Now this payload has control info + actual payload (802.11 Header + Data). Control info is of type UMAC\_EVENT\_RX, which will then be handed over to message handler for further processing.

For each consumed buffer, the host refills a new one to RPU. This refilling is done using HAL\_INTERNAL\_CMD, each command can populate 16 RX buffers to RPU.

# Power Management

## RPU Power States

The RPU transitions to different power states. These states influence whether RAMs and functional blocks are powered (and therefore power consumption of the RPU).

This section contains a high-level description of the model implemented by the sleep controller hardware. A more detailed description of the hardware can be found in [1] and [2].

|  |  |
| --- | --- |
| State | Description |
| RPU Off | RPU firmware has not yet been cold booted. Sleep controller registers are populated with RTL defaults. |
| Sleeping | RPU has been booted and is in a state where it is minimizing power consumption. GRAM is not accessible. The LMAC CPU is not running (not powered). |
| RPU On / Awake | LMAC CPU is running. GRAM is powered and accessible. |

| RPU On  RPU  Off  Awake  *Ready to*  *power down*  *(transitional*  *)*  Sleeping  Can  sleep?  Host prevents sleep  Host requests wake up  Sleep Controller requests wake up  Power down  Hot “boot”  Warm boot  Yes  No  Sleep opportunity  Cold  boot |
| --- |
| **Top Level Sleep State Diagram** |

## HP\_READY/RPU\_READY indication

The firmware will generate the RPU\_READY indication to indicate to the host that the firmware is up and running after awakening. The RPU\_READY indication is given to the host as a single 32-bit word via mtx\_to\_host\_cmd hostport register. This hostport indication will in turn assert the host interrupt, which will in turn assert the HP\_READY signal.

The firmware generates the RPU\_READY indication for the following situations:

* Cold boot
* Host-requested warm boot

Note: the firmware does NOT generate the RPU\_READY indication for the sleep controller (i.e. timer) requested warm boot. Sleep timers are used as part of the internal operation of the RPU (e.g. beacon wakeup) and this indication would disturb the host unnecessarily.

## Host/HAL Implications

Primary impact to host: the host must awaken the RPU before sending a HAL message to the RPU. In particular, the GRAM must be powered and accessible such that the host can write the HAL message.

### Awakening RPU

The host awakens the RPU prior to sending a group of HAL messages using the following procedure.

Assert WAKEUP\_NOW signal

If ( HP\_READY == FALSE ) {

Wait for RPU\_READY indication/interrupt

}

Foreach command in HAL\_commands\_to\_be\_sent {

Send HAL command (and verify ack)

}

De-assert WAKEUP\_NOW signal

### Handling RPU\_READY indication

The RPU\_READY indication is given to the host via mtx\_to\_host\_cmd hostport register. This is the same register as is used for HAL event messages. For normal HAL event messages, the hostport register is populated with an event sequence number. For the RPU\_READY indication the hostport register is instead populated with a specific data pattern, which is reserved for RPU\_READY indication.

|  |  |
| --- | --- |
| Hostport register value | Meaning |
| 0x7DEADBAD | RPU\_READY indication. No message is in GRAM. |
| 0x7FFFxxxx | HAL event message. The low order 16 bits of the hostport register value correspond to the event sequence number. |

## LMAC Operational States

The following section is background information regarding how the LMAC firmware manages its sleep state based on the operation state of the system.

The LMAC transitions to different operational states. Most operational state changes are initiated by the host.

These states influence:

* Logic used to transition between sleep states
* Whether or not RX packets are handled and forwarded

### State Descriptions

|  |  |
| --- | --- |
| State | Description |
| Disabled | The WiFi network interface has been shut down by the user. PHY/RF blocks are powered down. |
| Enabled, Idle | The WiFi network interface has been enabled, but there is currently no active scan and no active association. Typically, this is a pause between scans. PHY/RF blocks may be powered down. |
| Scanning | The LMAC is in the process of performing a scan for APs. PHY/RF blocks are powered up and RX packets are forwarded to the host. |
| Associated | The LMAC is forwarding traffic and monitoring beacons. PHY/RF blocks are powered up and RX packets are forwarded to the host. |

### Sleep strategy for operational states

|  |  |
| --- | --- |
| State | Sleep Strategy |
| Disabled | The LMAC will remain asleep in this state indefinitely until the host sends a command. |
| Enabled, Idle | The LMAC will go to sleep after inactivity of HAL commands. The LMAC will generally remain asleep in this state indefinitely until the host sends a command. |
| Scanning | The LMAC will remain awake in this state. (i.e. awake while scanning) |
| Associated | The LMAC will autonomously transition between awake and sleeping power states during the association to conserve power.  In general, the LMAC will remain awake when TX/RX data traffic is flowing. After traffic stops, the LMAC will begin long sleeps and only awaken for beacons. |

## Sleep Controller SOC Customization

Different customer applications may require different programming of the sleep controller.

One example of customization is the SOC may use some RPU peripherals/clocks for other blocks within the SOC, and therefore may require these peripherals/clocks to remain powered, even when the RPU is sleeping.

Another example is the choice of sleep clock frequency for the system. Nominally, a 32000 Hz clock will be used, but the system designer has the choice of using a 32768 Hz crystal. Furthermore, this clock may not be very accurate, so it is desirable to provide a way to compensate for this.

If the host CPU attempted to modify the relevant sleep controller registers directly to accomplish this, it would conflict with the operation of the firmware. Therefore, a method is provided for the host to specify these settings within a HAL command to the firmware, and the firmware will in turn use this information to program the sleep controller.

### Calculation of wakeup time

The figure below describes the time components of the sleep interval.

beacon

beacon

CPU run

CPU run

SC pwr up

D

W

S= D-W-P-B

P

B

Sleep

The following parameters feed into the calculation of wakeup time into the sleep timer:

A = DTIM beacon interval, e.g. 300 msec: This value is discovered from the 802.11 association.

W = protocol working time: This is the time that the CPU has remained awake since the TBTT of the previous beacon was received. The firmware inspects the beacon information and maintains a timer to measure how much time has elapsed since the TBTT of the previous beacon.

P = sleep controller managed power-up time: This is the period of time which begins when the sleep timer expires and ends when the LMAC CPU is triggered to boot. The firmware discovers this value by reading the UCC\_SLEEP\_CTRL\_PWR\_ON\_TIME[RESETN\_MCU0\_EVENT] register.

B = firmware boot time: This is the period of time which begins when the LMAC CPU is triggered to boot and ends when the system is fully initialized and ready to receive packets. The firmware uses an internal constant for this value.

Before the firmware transitions to sleep, it computes the sleep duration S, which must be programmed into the UCC\_SLEP\_CTRL\_WAKEUP\_TIME register. S can be computed as:

S = D – W – P – B

The value S is then scaled into units of sleep timer ticks. The firmware sets the UCC\_SLEEP\_CTRL\_SLEEP\_CLK\_DIV to zero (i.e. no hardware divider is used and WAKEUP\_TIME register is specified in units of sleep timer ticks.)

If S is smaller than a threshold, the firmware will not sleep and instead remain awake.

### HAL Message Details

The sleep controller customization settings are specified within the HAL command CMD\_CFG\_PWRMGMT. The appropriate time for the host to send the CMD\_CFG\_PWRMGMT command is as the first HAL command immediately after cold boot.

The CMD\_CFG\_PWRMGMT HAL command structure is shown in the figure below.

The config\_mask field is a bit-coded field indicating which parameters to configure. If a configure bit is set to 1, the corresponding parameter field within the HAL message must be populated and the firmware will use it. If a configure bit is set to 0, the firmware will use an internal default value and the corresponding parameter field will be ignored.

The firmware default values for the sleep controller registers are the RTL default values. The default value for sleep\_timer\_freq\_hz is 32000.

struct cmd\_cfg\_pwrmgmt {

struct host\_mac\_msg\_hdr hdr;

#define PMFLAG\_PWR\_ON\_VALUE 0x0001 /\* config UCC\_SLEEP\_CTRL\_PWR\_ON\_VALUE \*/

#define PMFLAG\_PWR\_OFF\_VALUE 0x0002 /\* config UCC\_SLEEP\_CTRL\_PWR\_OFF\_VALUE \*/

#define PMFLAG\_RAM\_ON\_STATE 0x0004 /\* config UCC\_SLEEP\_CTRL\_RAM\_ON\_STATE \*/

#define PMFLAG\_RAM\_OFF\_STATE 0x0008 /\* config UCC\_SLEEP\_CTRL\_RAM\_OFF\_STATE \*/

#define PMFLAG\_PWR\_ON\_TIME 0x0010 /\* config UCC\_SLEEP\_CTRL\_PWR\_ON\_TIME \*/

#define PMFLAG\_PWR\_OFF\_TIME 0x0020 /\* config UCC\_SLEEP\_CTRL\_PWR\_OFF\_TIME \*/

#define PMFLAG\_RAM\_ON\_TIME 0x0040 /\* config UCC\_SLEEP\_CTRL\_RAM\_ON\_TIME \*/

#define PMFLAG\_RAM\_OFF\_TIME 0x0080 /\* config UCC\_SLEEP\_CTRL\_RAM\_OFF\_TIME \*/

#define PMFLAG\_SLEEP\_FREQ 0x0100 /\* config sleep\_timer\_freq\_hz \*/

uint32 config\_mask; /\* bit mask indicating which values to program \*/

uint32 pwr\_on\_value[2]; /\* UCC\_SLEEP\_CTRL\_PWR\_ON\_VALUE registers \*/

uint32 pwr\_off\_value[2]; /\* UCC\_SLEEP\_CTRL\_PWR\_OFF\_VALUE registers \*/

uint32 ram\_on\_state[2]; /\* UCC\_SLEEP\_CTRL\_RAM\_ON\_STATE registers \*/

uint32 ram\_off\_state[2]; /\* UCC\_SLEEP\_CTRL\_RAM\_OFF\_STATE registers \*/

uint32 pwr\_on\_time[32]; /\* UCC\_SLEEP\_CTRL\_PWR\_ON\_TIME registers \*/

uint32 pwr\_off\_time[32]; /\* UCC\_SLEEP\_CTRL\_PWR\_OFF\_TIME registers \*/

uint32 ram\_on\_time[4]; /\* UCC\_SLEEP\_CTRL\_RAM\_ON\_TIME registers \*/

uint32 ram\_off\_time[4]; /\* UCC\_SLEEP\_CTRL\_RAM\_OFF\_TIME registers \*/

uint32 sleep\_timer\_freq\_hz; /\* sleep timer frequency in Hz \*/

};

### General flow of firmware sleep operation

1. At the very beginning, the host will release reset on the RPU. The RTL default values within the sleep controller are used as the RPU comes out of reset. Releasing reset will always use the RTL defaults because sleep controller registers cannot be modified prior to reset. Specifically, the following registers are used by the sleep controller to control the power up sequence and result:
   1. UCC\_SLEEP\_CTRL\_PWR\_ON\_VALUE[n] selects which peripherals will be powered (i.e. which signals will be triggered)
   2. Similarly, UCC\_SLEEP\_CTRL\_RAM\_ON\_STATE[n] selects the RAM power state for awake operation
   3. UCC\_SLEEP\_CTRL\_PWR\_ON\_TIME[n] determines the timing sequencing used to power on the selected peripherals/signals
2. The host may download a patch and trigger the LMAC firmware to cold boot. The firmware is then running.
3. The host will provide the sleep controller customization values to the firmware within a HAL message. The firmware will store these customization values in its memory.
4. When there is a sleep opportunity, the firmware will prepare to initiate sleep.
5. If the firmware is maintaining a wifi association, the firmware computes the duration of the sleep based on several parameters (see previous section).
6. The firmware will program the sleep controller registers, based on the stored customization settings as well as its internal state. Both the power up registers and the power down registers are configured.
7. When transitioning from awake to sleep, these registers are used by the sleep controller to control the power down sequence and result:
   1. UCC\_SLEEP\_CTRL\_PWR\_OFF\_VALUE[n] selects which peripherals will now be powered down (i.e. which signals will be triggered)
   2. Similarly, UCC\_SLEEP\_CTRL\_RAM\_OFF\_STATE[n] selects the new RAM power state for sleeping.
   3. UCC\_SLEEP\_CTRL\_PWR\_OFF\_TIME[n] determines the timing sequencing used to power on the selected peripherals/signals
8. The RPU will remain in sleep state until the sleep timer expires (or the host requests wake up), and then it will initiate the transition from sleep to awake. These registers are used by the sleep controller to control the power up sequence and result:
   1. UCC\_SLEEP\_CTRL\_PWR\_ON\_VALUE[n] selects which peripherals will now be powered (i.e. which signals will be triggered)
   2. Similarly, UCC\_SLEEP\_CTRL\_RAM\_ON\_STATE[n] selects the new RAM power state for awake operation
   3. UCC\_SLEEP\_CTRL\_PWR\_ON\_TIME[n] determines the timing sequencing used to power on the selected peripherals/signals
9. The firmware will warm boot and begin running. The process repeats with Step 4.

# References

1. C4201 (Echo510) RPU Technical Reference Manual
2. LPW System Technical Reference Manual
3. Data Structures for Host RPU Communication

# FAQ’s

1. Why can’t we generate an event to host for every AMPDU sub frame instead of coalescing up to 16 frames?

Interrupt coalescing in event generation logic is added to reduce host ISR load from RPU. This ISR load on host will be huge for the small sized WiFi packets. This will be more applicable for TCP TX cases. TCP ACK frame will have a size of 82 bytes. We will be generating events to host for every 5us in 11n 40 MHz MCS7 data rate. So, coalescing RX events will improve host performance at cost of more DDR pointers.