MAC 80211 Integration for MeshDynamics OpenWRT MD6000

TABLE OF CONTENTS

TABLE OF FIGURES

1 Introduction

MeshDynamics delivers third-generation wireless mesh networking solutions for high-performance outdoor data, voice, and video networking. Based on sophisticated dynamic channel-agile networking algorithms, MeshDynamics MD4000 family of Structured Mesh™ wireless nodes deliver very low-latency and low-jitter performance, even over multi-hop topologies where many earlier generation wireless mesh networking products fail.

Software development began in 2001 with United States Defence contracts. Prototypes were rigorously tested by the United States military through 2003-2005. Production shipments began in 2005, providing scalable wireless mesh networking solutions for Defence, Homeland Security, surface and underground mining.

Today, MeshDynamics products are used worldwide in mining and industrial, video surveillance, defence, and outdoor sport event.

MeshDynamics Structured Mesh™ multi-radio mesh network MESH Algorithm provides features which makes it different from other mesh networking implementations. These features include:-

1. Dynamic RF channel management
2. Dynamic scanning for mobility
3. Structured Mesh™ heartbeat transmission and processing
4. Mesh routing table management
5. Self-forming/Self-healing mesh networking

It also includes security components like Wi-Fi-Protected Access (WPA) version 1 and 2, IEEE 802.11i, FIPS 140-2, IEEE 802.1x which makes the MeshDynamics Structured Mesh™ a better choice for users across the Defence and Homeland Security Agencies in US, UK and Canada to provide mission critical video surveillance and perimeter security.

Currently the mesh algorithm implementation is tightly coupled with the underlying Atheros drivers and is based on closed sourced HAL implementation. This architecture is however robust, but to provide more flexibility to the customers, an approach is needed to make the mesh algorithm independent of

the HAL and the device drivers. With this approach, more options open to customers to easily choose between the underlying hardware and gain benefits of open source approach.

1.1 Objective

The objective of this document is to describe the software level changes which are required to port the existing meshap software to the standard Linux. This document shall describe the areas in the Linux Kernel, mac80211 and meshap which will require modifications to integrate meshap with mac80211.

2 Scope

This document describes the details of software level changes required for the integration of MeshDynamics' software with mac80211. It also explains the high level architecture of Mesh Networking and captures the terminology and definitions of MeshDynamics product line. It describes the complete architectural modification which is required for integration of mesh dynamics with mac80211. This document also captures the list of APIs which would be impacted and would require modification to support the new integration architecture.

3 Acronyms

IMCP	Infrastructure Mesh Control Protocol
AP	Access Point
BSS	Basic Service Set
CTS	Clear To Send
MAC	Media Access Control
MLME	Media Access Control Sub layer Management Entity
POE	Power Over Ethernet
RADIUS	Remote Authentication Dial In User Service
RTS	Request to Send
Rx	Reception
Tx	Transmission
SSID	Service Set Identifier
STA	Station

4 References

- "MD4000 Node Deployment and Trouble Shooting Guide" - MD4000_HWMANUAL.pdf
- "Architecture Overview" - MD-OEM-HARDWARE-INTEGRATION-VOL1-4.pdf
- http://www.campsmur.cat/files/mac80211_intro.pdf
- http://wireless.kernel.org/en/developers/Documentation/mac80211

5 Mesh Nodes

Functionally device in the mesh network can be classified as Root node, Relay node or Station.

Root Node

For root node the Ethernet connection of mesh node is connected to POE and POE to a Switch. This connection acts as the uplink for root nodes as shown in the figure. Root node backhaul is the wired network. For increased bandwidth all 4 radios in root node can be configured as downlinks radios.

Relay Node

Relays have wireless uplinks to an upstream downlink radio. Downlink radios act likes Access Points (AP), they send out a beacon. Uplink radios act like clients, they do not send out a beacon. The difference in the physical setup of root nodes and relay nodes is the connection from a root node's POE to a switch. Whenever either of the physical setups is altered, the nodes must be rebooted in order to assume their new role. The uplink and downlink radios form a wireless backhaul path.

Station

Mobile station is a device that connects to AP radio link of the mesh node. Ex: laptop, mobile phone etc.

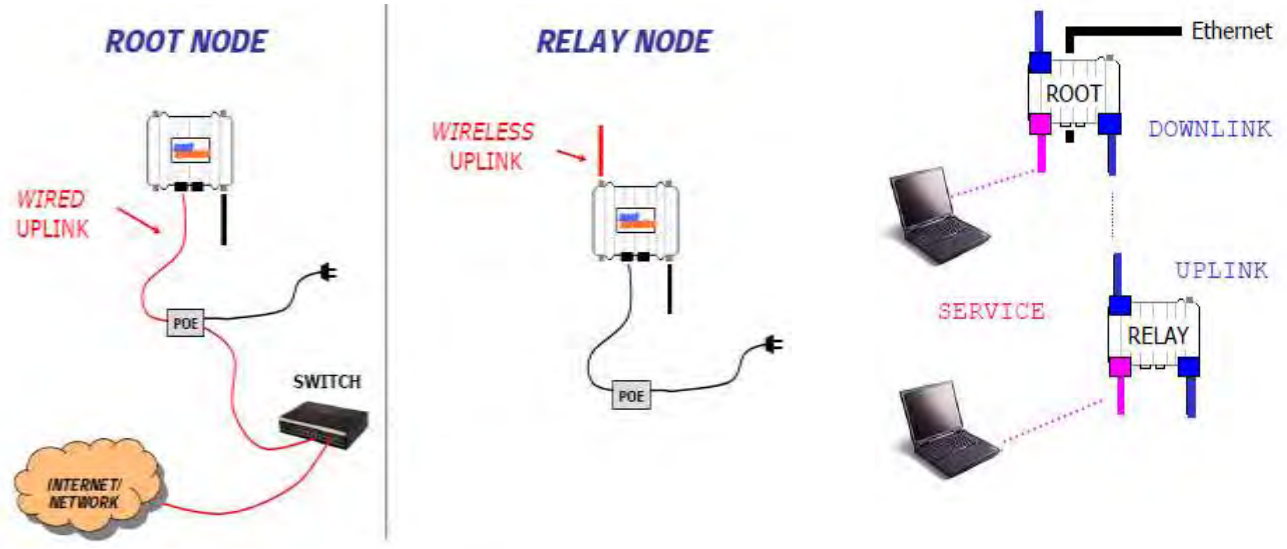


Figure 1 Different Mesh Nodes

A wireless radio card in a laptop can inform the presence of downlinks but not uplinks. Downlinks beacon but uplinks do not. AP radios operate in the 2.4GHz band service 802.11b/g clients. 802.11a wireless devices may be serviced by the 5.8GHz downlink. Backhaul radios operate in 5.8GHz band to avoid interference with the 802.11b/g 2.4GHz AP radio as shown pink in the figure.

MeshDynamics 3rd generation mesh technology surpasses 1st and 2nd generation mesh technology.3rd generation mesh products support up to 4 radios in a single enclosure. Radio slots 0, 1 house one uplink and one downlink (radio

backhaul) operates on non-interfering channels but in the same frequency. Slot 2 can be used for client (laptops etc.) connectivity, generally a 2.4GHz AP radio that supports 802.11b, g, b & g modes. Slot 3 can house a 2nd downlink, a 2nd AP or a scanning radio for mobile mesh module – that forms part of the meshed backhaul in dynamic infrastructure/high speed mobile mesh networks. There are two Ethernet ports on each module for wired connectivity.

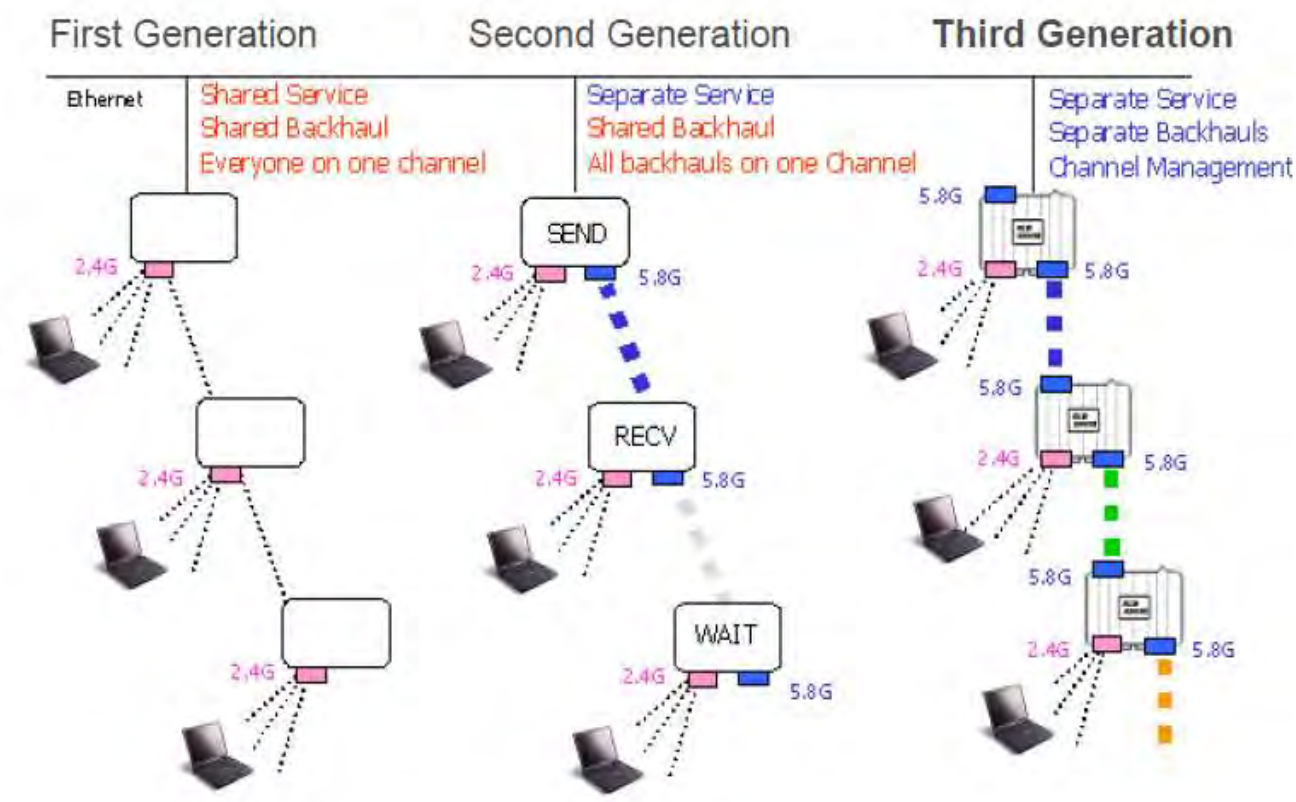


Figure 2 Generations of Mesh Technology

First and second generation mesh nodes use only one channel of a frequency spectrum across all links of a backhaul during operation (fig:). A node in the mesh cannot send and receive at the same time since the same frequency is used for both functions. This makes for a very inefficient process that severely affects bandwidth as the number of hops increases.

The third generation mesh nodes uses multiple channels (fig:) simultaneously within the utilized spectrum in order to ensure minimal bandwidth loss as the number of hops increases. Typically, the 5GHz spectrum is used for the backhaul. Since different 5GHz channels are used by adjacent links in the mesh, there is no interference along the backhaul. This allows each node to send and receive at the same time, therefore, conserving bandwidth over many hops.

6 Mesh Networking

Various mesh nodes explained in previous section forms the mesh network.

Network formation:

Upon boot up, a root node will beacon a default ESSID of "StructuredMesh" on its downlink and AP radios. When a relay node boots up, it will scan on its uplink radio. When the uplink radio of a relay node hears the beacon from a root node, it will associate. This same relay node will then start to beacon the default ESSID of "StructuredMesh" on its downlink and AP radios. Any scanning relay nodes that hear this beacon will associate, thus growing the network.

Until a relay node associates to a parent node, it will beacon an ESSID starting with the words "MESH-INIT" on its downlink and AP radios. This is to indicate that it has no association to the mesh network. If a root node continually beacons an ESSID of "MESH-INIT-… ", this indicates that it is not physically connected the switch, and is therefore attempting to function as a relay node.

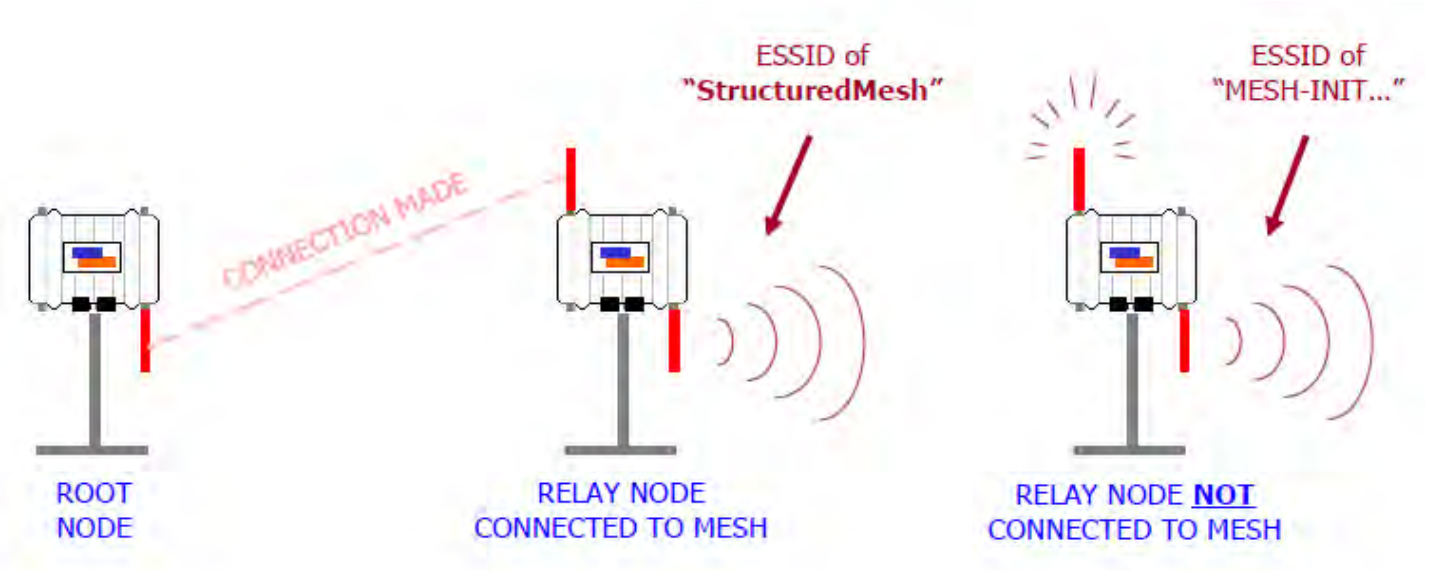


Figure 3 Formations of Mesh Networks

Joining Criteria and Switching:

The initial child-to-parent link is formed based on the signal strength the child sees from the parent. After joining, the child node pro-actively samples neighbour links. The connectivity rate then becomes the main criteria, and the "global" connectivity rate is given the higher priority.

Link switching decisions for stationary nodes are made every heartbeat interval. Mobile nodes make switching decisions much more quickly. For a child node to switch parent nodes, the new 'best' parent must provide the best link qualities for 3 consecutive heartbeats.

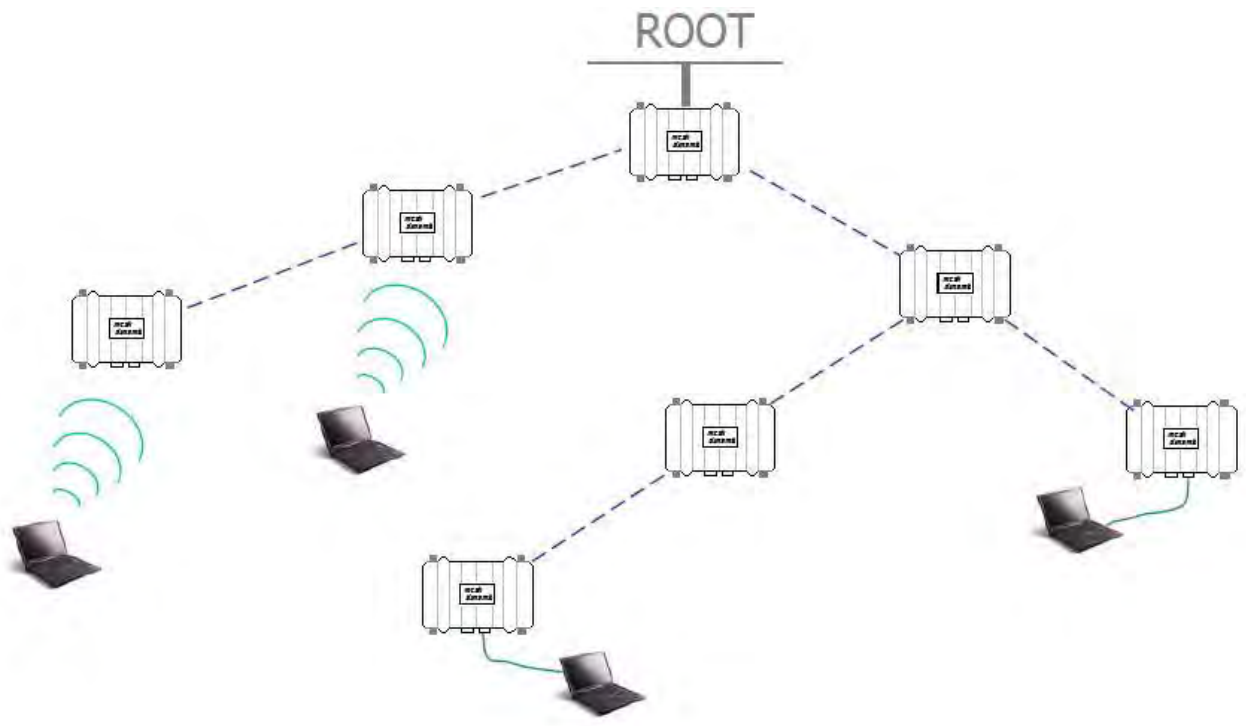


Figure 4 Mesh network with Root, Relay and station

The Persistent Third-Generation Mesh (P3M) technology of the Meshdynamic's product lines enables a node or set of nodes to remain functional without the presence of a wired root node where as When a standard mesh network loses contact with its root node, all connections within the network are broken. Clients within the mesh can no longer transfer data to and from other clients in the same mesh

7 Product Models

2.4GHz Backhaul Products

1. MD4220-IIxx: 2-Radio module 2.4GHz uplink and downlink Backhaul (BH).
2. MD4320-IIIx: 3-Radio module 2.4GHz sectored BH slots 0,1 and 2.4GHz AP radio in slot 2.
3. MD4325-IIxI: 3-Radio module 2.4GHz BH, Downlink also acts as AP. A 2.4GHz Mobility Scanner in slot3.
4. MD4424-IIII: 4-Radio module 2.4GHz service radios (AP) in all slots. Use with 4 panel antennas.

5GHz Backhaul Products

1. MD4250-AAxx: 2-Radio module 5GHz uplink and downlink Backhaul (BH).
2. MD4350-AAIx: 3-Radio module 5GHz BH and 2.4GHz AP radio in slot 2. AP modes may be b, g, or b & g.
3. MD4452-AAIA: 4-Radio module 5GHz BH and 2.4GHz AP radio. Second sectored 5.8GHz downlink in slot 3.
4. MD4454-AAAA: 4-Radio module 5GHz with radios as downlinks. Intended as root with four 90 deg panels.
5. MD4458-AAII: 4-Radio module 5GHz BH and two 2.4GHz AP radios in slots 2, 3 for sectored service.
6. MD4455-AAIA: 4-Radio module 5GHz BH and 2.4GHz AP radio in slot 2. 5GHz mobility scanner in slot 3.

8 Hardware Boards

MeshDynamics has provided different kind of hardware boards where each has different number of wireless cards and different processors as described below.

8.1 Gateworks Laguna

Gateworks Laguna GW2388-4 is designed for a wide range of outdoor applications such as WISP customer premise equipment, Mesh repeaters, WiMAX pico base stations, 3G-routers, wireless point to multipoint bridges and 3G to Wi-Fi routers and gateways.

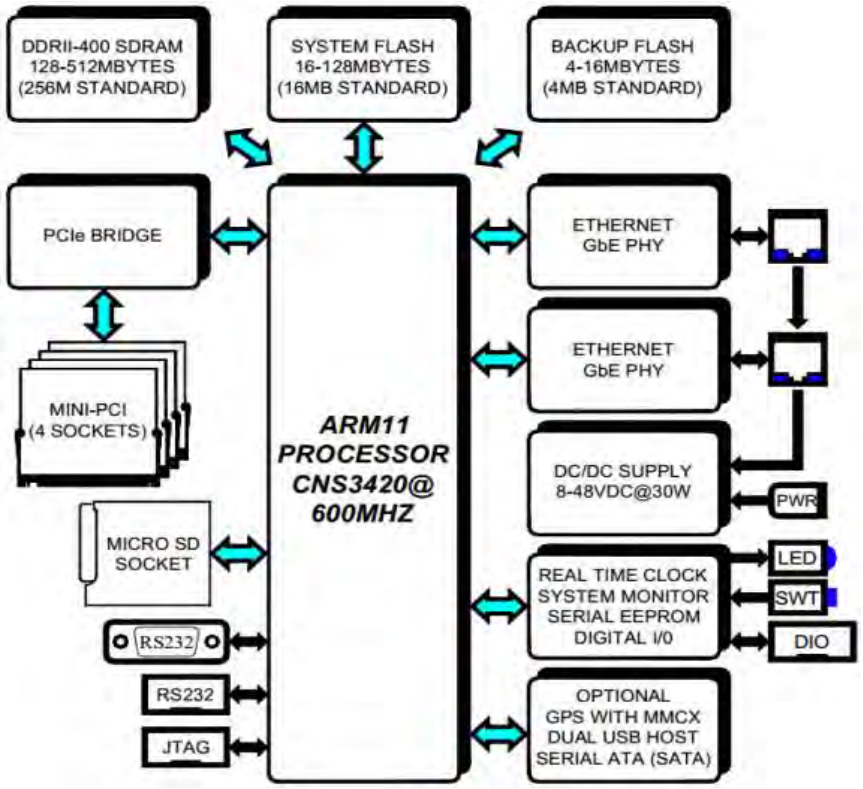


Figure 5 Block Diagram: Gateworks Laguna

As shown in **Figure 5** Laguna GW2388-4 board features the Cavium® ECONA™ CNS3420 Dual Core ARM11 SoC processor operating at 600MHz, 256Mbytes of DDRII-400 DRAM, 16Mbytes of System Flash, 4Mbytes of Backup and Restore Flash, and two Gigabit Ethernet ports. The board includes four high-power Type III Mini-PCI sockets capable of supporting any combination of 802.11abgn radios, WiMAX radios, and other Mini-PCI peripherals. MeshDynamics end product with Laguna GW2388-4 as shown in

Figure 6

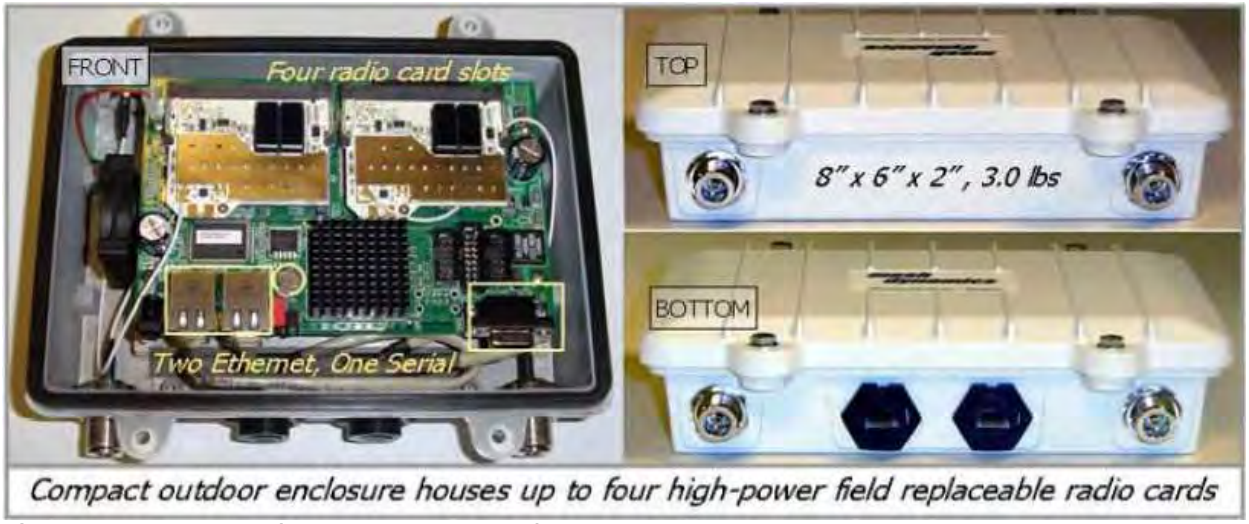


Figure 6 Gateworks Laguna Board

8.2 Gateworks Cambria

Gateworks Cambria GW2358-4 is designed for enterprise and residential network applications.

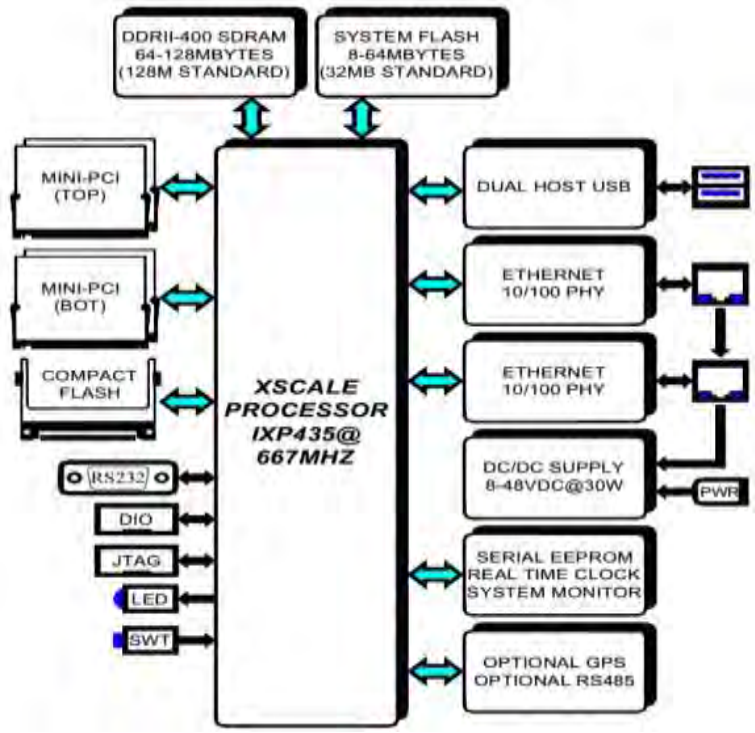


Figure 7 Block Diagram: Gateworks Cambria

As shown in Figure 7 this board consists of an Intel® IXP435 XScale® operating at 667MHz, 128Mbytes of DDRII-400 DRAM, and 32Mbytes of Flash. Peripherals include four Type III Mini-PCI sockets, two 10/100 Base-TX Ethernet ports with IEC-6100-4 ESD and EFT protection, two USB Host ports, and Compact Flash socket. Additional features include digital I/O, serial EEPROM, and real time clock with battery backup, system monitor to track operating temperature and input voltage, RS232 serial port for management and debug, and watchdog timer. The GW2358 also supports GPS and RS485 serial port as ordering options. Power is applied through a dedicated connector or through either Ethernet connector with the unused signal pairs in a passive power over Ethernet architecture.

8.3 Gateworks Avila

Gateworks Alvia GW2348-2 is designed for enterprise and residential applications.

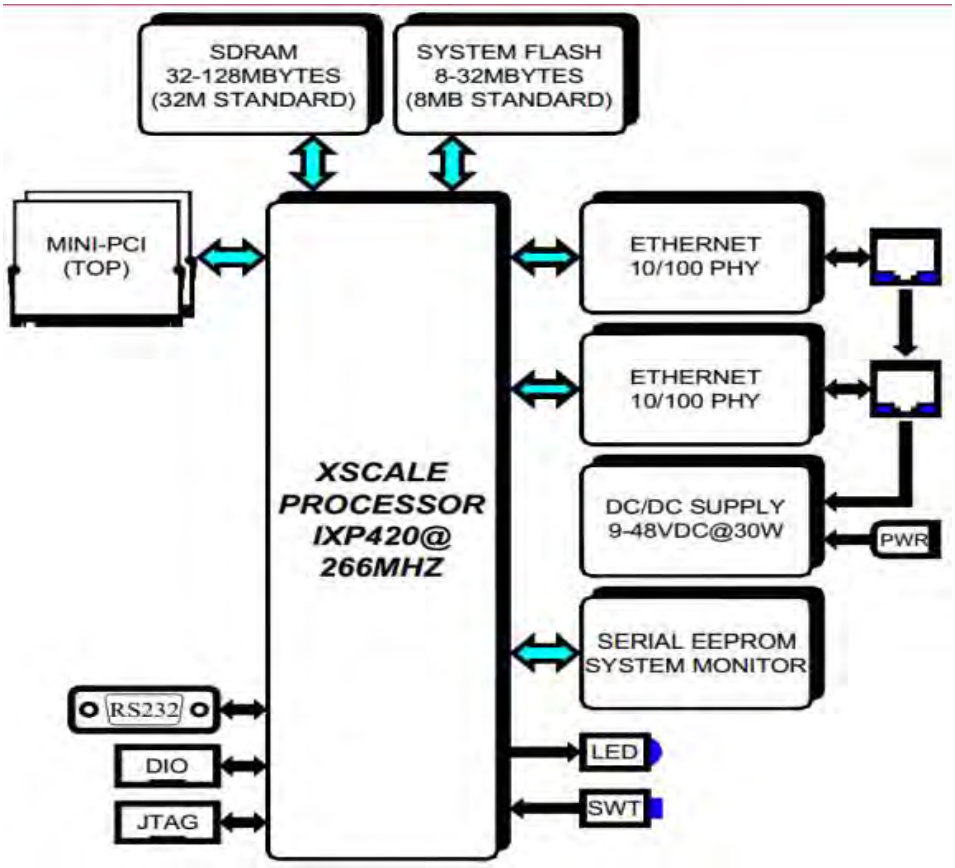


Figure 8 Block Diagram: Gateworks Avila

As shown in Figure 8 this network processor consists of an Intel® IXP420 XScale® operating at 266MHz, 32Mbytes of SDRAM, and 8Mbytes of Flash. Peripherals include two Type III Mini-PCI sockets and two 10/100 Base-TX Ethernet ports with IEC-6100-4 ESD and EFT protection. Additional features includes digital I/O, serial EEPROM, system monitor to track operating temperature and input voltage, RS-232 serial port for management and debug, and watchdog timer. Power is applied through a dedicated power connector or through any Ethernet connector with the unused signal pairs in a passive power over Ethernet architecture.

8.4 Ubiquity Bullets

The Bullet M2 HP is a revolutionary outdoor radio device that features a signal strength LED meter for antenna alignment, a low-loss integrated N-type RF connector, and a strong and robust weatherproof design. This inline wireless access point can instantly transform any antenna into a carrier class radio system. This unique inline access point is perfect for all your 802.11b/g WLAN applications.

Figure 9 Ubiquity Bullets

This consists of an Atheros MIPS 4KC operating at 180MHz, 16Mbytes of SDRAM, 4Mbytes of Flash and 10/100 Base-TX Ethernet interface.

9 Mac80211

Mac80211 is a subsystem to the Linux kernel, implements shared code for SoftMAC wireless devices. SoftMAC devices allow for a finer control of the hardware, allowing for 802.11 frame management to be done in software for them, for both parsing and generation of 802.11 wireless frames. The

Figure 10shows mac80211 architecture overview.

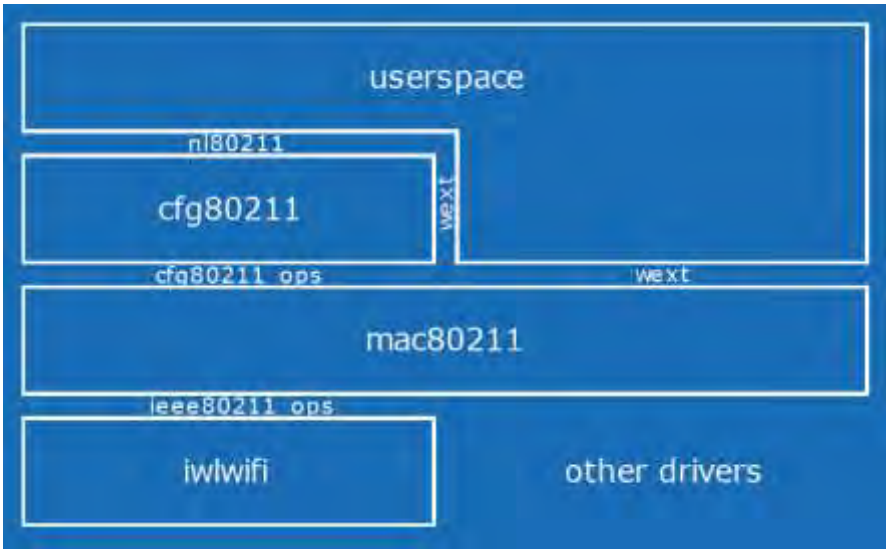


Figure 10 mac80211 architecture overview

9.1 Mac80211 components

9.1.1 Hostapd

Hostapd is a user space daemon for access point and authentication servers. It implements IEEE 802.11 AP management, IEEE 802.1X/WPA/WPA2/EAP Authenticators, RADIUS client, EAP server and RADIUS authenticator server.

The hostapd also handles generation of beacons and other wireless packets, as well as wpa-psk, wpa2 etc encryptions. The current version supports Linux (Host AP, madwifi, mac80211-based drivers) and FreeBSD (net80211).

Hostapd also includes following functions,

1. Implements (almost) the entire AP MLME
2. Works with mac80211 through nl80211
3. Requires working radio tap packet injection
4. Requires many of the nl80211 callbacks
5. Requires 'cooked ' monitor interfaces

9.1.2 cfg80211

cfg80211 is the Linux 802.11 configuration API. cfg80211 replaces Wireless-Extensions. nl80211 is used to configure a cfg80211 device and is used for kernel <–> user space communication. The functions of cfg80211 are as follows:

1. drivers register a struct wiphy with cfg80211, this includes hardware capabilities like

- Bands and channels
- Bitrates per band
- HT capabilities
- Supported interface modes.

These parameters need to be set before registering netdevs,

2. The netdev ieee80211_ptr links to registered wiphy, cfg80211 will also update the list of registered channels and (optionally) notify driver.
3. Create/remove the virtual interfaces
4. Change type of virtual interfaces (provides wext handler)
5. Change 'monitor flags'
6. Keeps track of interfaces associated with wireless device
7. Will set all interfaces down on rfkill
8. Allow multiple interfaces combining e.g. WDS and AP for wireless backhaul
9. Supports multiple SSID's, channel specification, IE insertion

9.1.3 mac80211

mac80211 implements the cfg80211 call backs for SoftMAC devices, mac80211 then depends on cfg80211 for both registration to the networking subsystem

and for configuration. Configuration is handled by cfg80211 both through nl80211 and wireless extensions.

In mac80211 the MLME is done in the kernel for station mode (STA) and in user space for AP mode (hostapd).

Supported features:

- IEEE 802.11abgn
- IEEE 802.11abgn
- Integration of work for the emerging 802.11s standard
- Different types of interfaces
- Vendor specific rate support
- QOS
- All mac80211 drivers get monitor mode support

9.1.4 Drivers

Drivers such as ath5k, ath9k are supported with mac80211 framework. In the transmission path when mac80211 layer has packet to send out, the driver will send the packet out to the hardware connected to it, and in the reception path it hands over the incoming packets coming from hardware to the mac80211 layer.

9.2 MAC 80211 architecture

9.2.1 Transmission Path

In the transmission path the kernel hands over the packet to the virtual interface and then the 80211 header is added, initialization of transmission time is done, headroom is created for encryption,

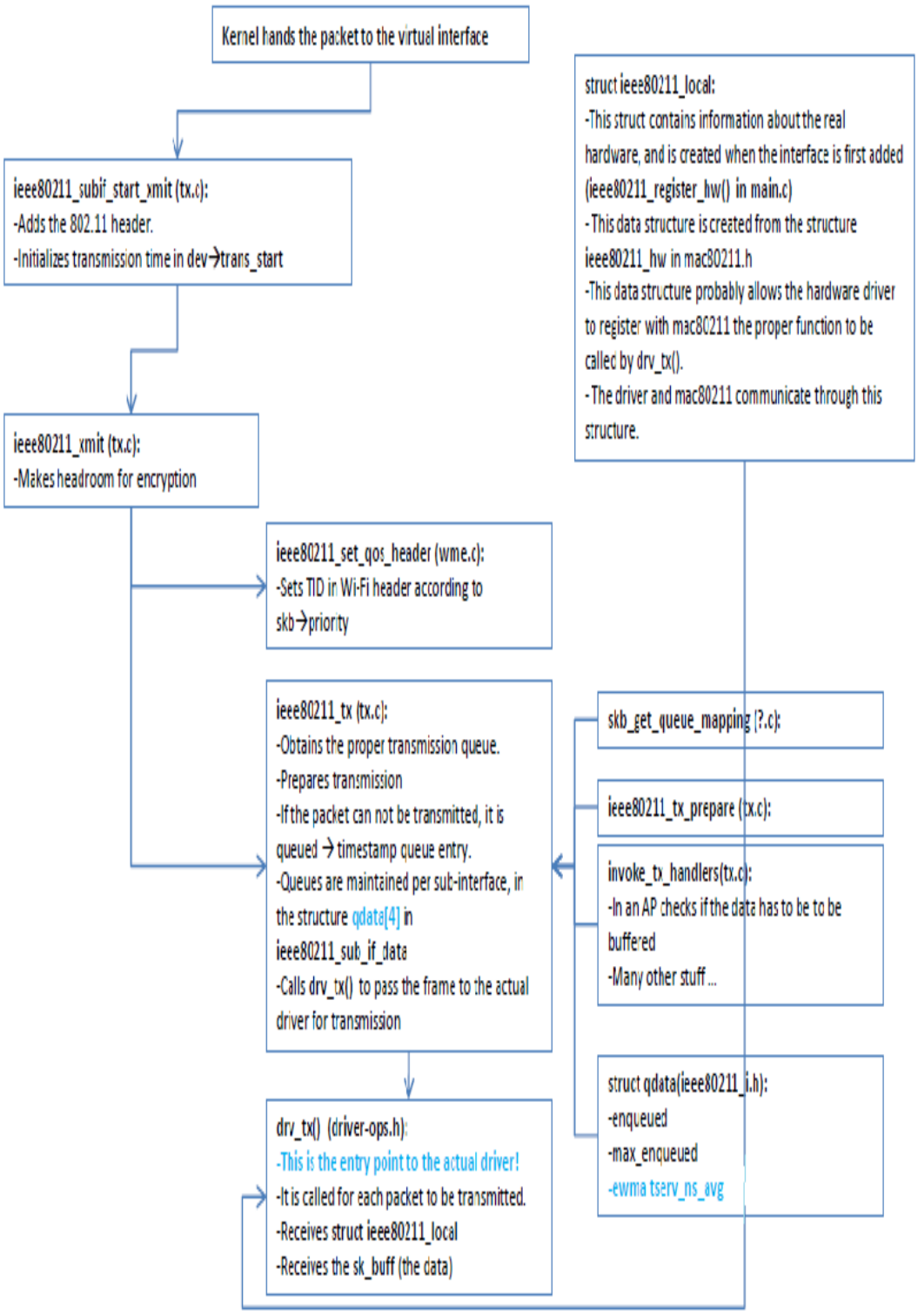


Figure 11 Transmission Path of Mac80211

The below flowchart shows the transmission path of ath5k driver to the hardware.

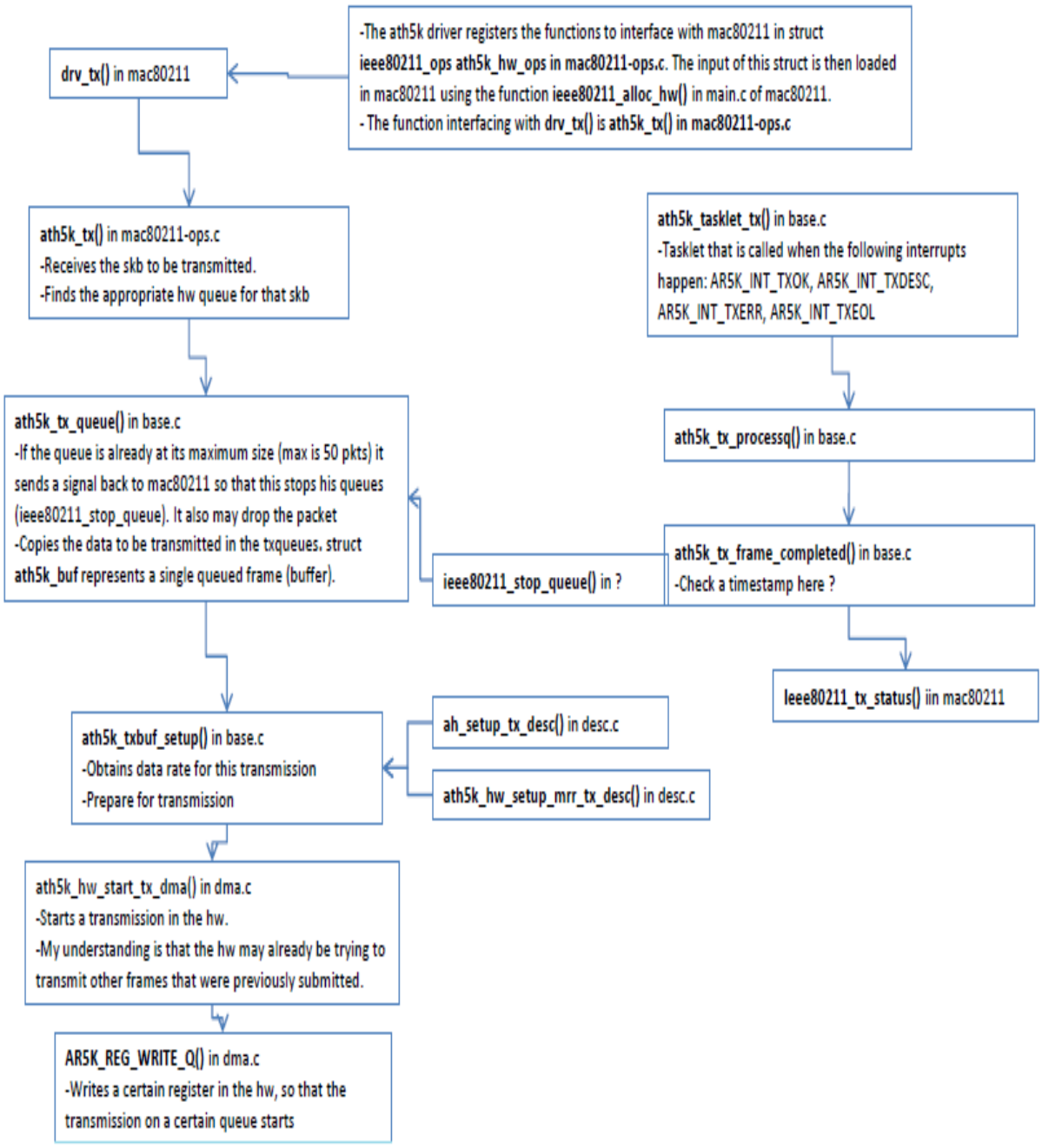


Figure 12 Transmission Path of ath5k drivers

9.2.2 Reception path

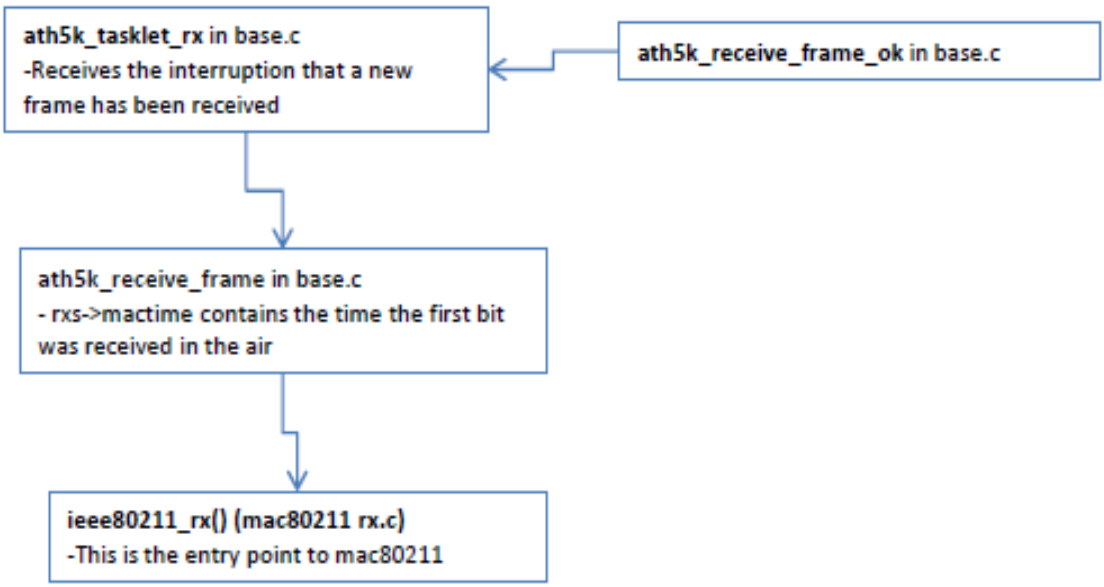


Figure 13 Reception Path of ath5k

The below figure shows the flowchart of reception path from mac80211 to kernel.

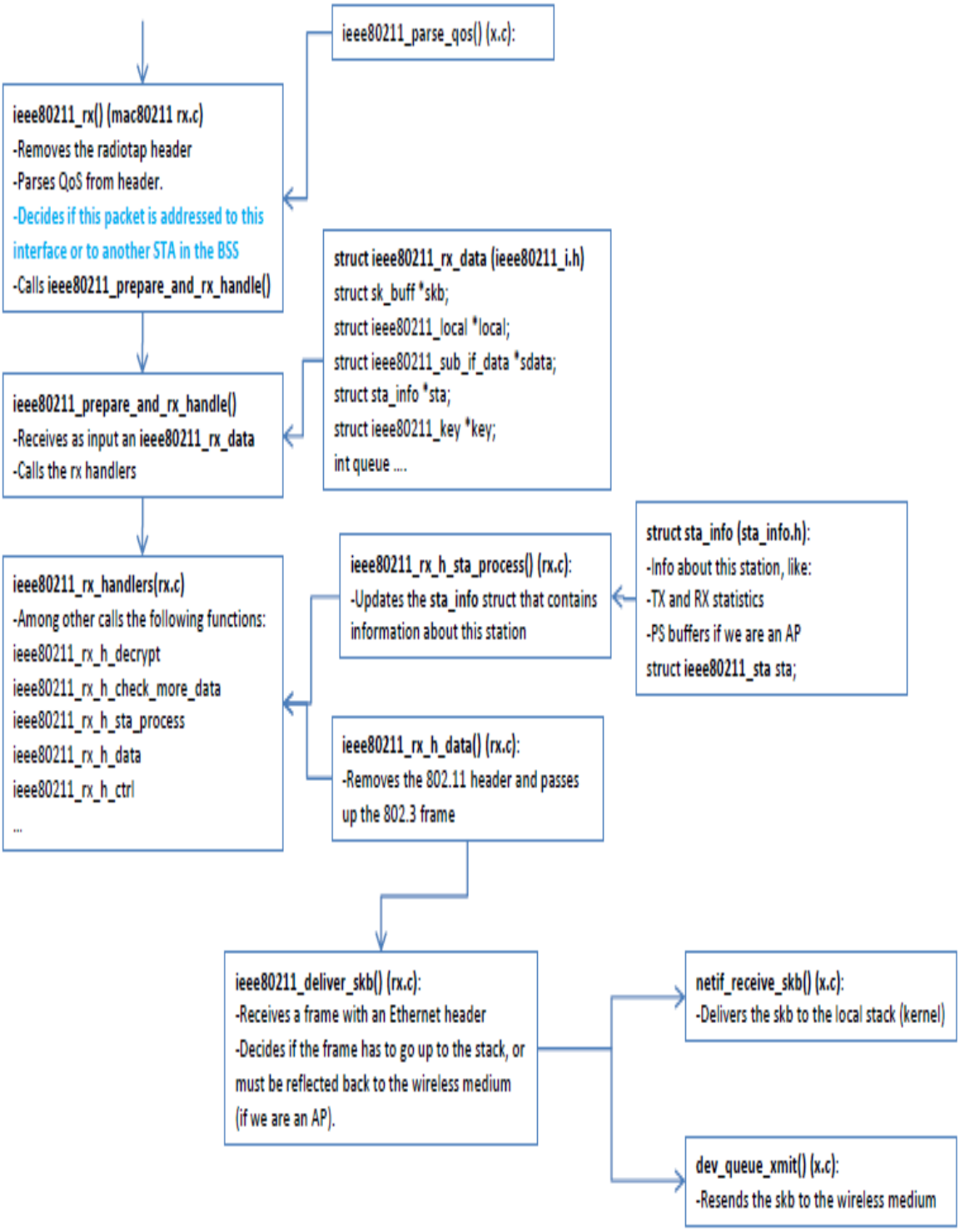


Figure 14 Reception Path of mac80211

10 MAC 80211 Frame formats

IEEE 802.11 is set of specification for implementation of wireless local area network (WLAN). The main components of wire local area network are media access control (MAC) and physical layer (PHY). MAC is set of rules to determine how to access the medium and send the data, but the details of transmission and reception are left to the PHY. This operates on 2.4, 3.6, 5 and 6 Ghz frequency band.

Frame format

Following diagram represent the generic 802.11 MAC frame. All type of frames do not use all address fields. Contents of address field may change depending on type of frame being transmitted. Fields are transmitted from left to right.

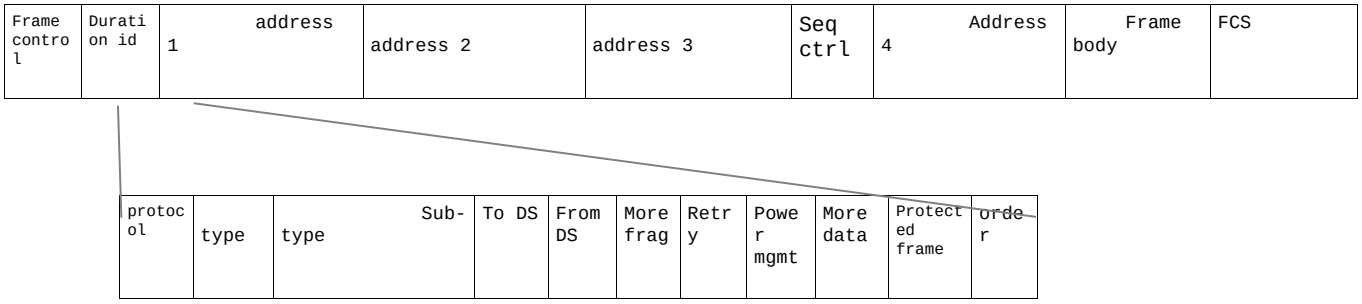


Figure 15Max80211 Frane Fornat

Frame control

Frame control sub field is of 2 bytes. Following are the components of frame component subfield.

Protocol version

This field is of 2 bits representing the protocol version of 802.11. For 802.11, the value of protocol version is zero.

Type and subtype

Type and subtype field identify the type of frame used. For example type can be management and subtype can be association etc.

To DS and from DS

To DS = 1 indicates frame is for distribution system and from DS = 1 represent frame is received from the distribution system. Whereas To DS = 0 and From DS = 0 indicates frame is destined to/received from within the BSS.

More fragment bit

If frame is fragmented then more fragment bit is set to 1. Large frame which is beyond the scope of one frame can set the more fragment bit to 1, whereas last frame set more fragment bit to 0 i.e. No more fragmented frame of large frame.

Retry bit

Some frame requires retransmission. For this if frame is retransmitted retry bit is set to 1.

Power management bit

This field is of one bit. When this field is set to 1 means station is in power save mode whereas 0 means station is active. Access point cannot set power management bit. So this bit always remains 0 for access point.

More data bit

More data bit is used to buffer the frame received from the distribution system. An access point sets this bit to 1 to indicate that at least one frame is available and is for the sleeping station.

Protected frame bit

If the frame is protected by the link layer then protected frame bit is set to 1. When frame is decrypted this bit is toggled.

Order bit

When strict ordering method is employed this bit is set to 1.

Duration/ID field

Duration/ID has basically three types of usages

PS-POLL frame

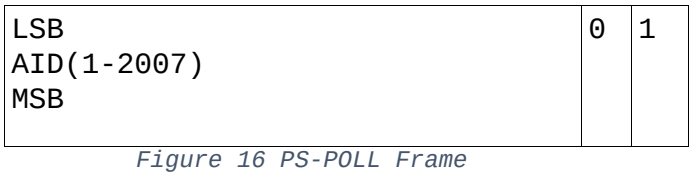


Figure 16 PS-POLL Frame

Mobile station may sleep in order to save the battery power. In PS-POLL frame both bits 14 and 15 are set to 1. Sleeping station must wake up periodically in order to retrieve the buffered frame from AP. For this station sends PS-POLL frame to AP. AID is also inserted in PS-POLL frame to indicate which BSS it belongs to.

Address field

An 802.11 frame may contain four address fields. All addresses are 48 bit long. Address 1 is used for the receiver; an address 2 is used for the transmitter and address 3 field for filtering by the receiver. If first bit of these addresses is 1 this means address is unicast address. If first bit

is 1 this means address represents a group of station called multicast address. If all bits are set to 1 then frame is broadcast address.

These addresses are used for the following purposes.

Destination address

This address represents the address of the final recipient. If this matches with the host address then frame will be handover to higher protocol layer for processing.

Source address

This address identifies the originator of the frame. Only single host can be the originator of the frame so this address will be the unicast address which always starts from 0.

Receiver address

This 48 bit address indicates that which station will process the frame. Receiver address may or may not be the destination address.

Transmission address

This 48 bit address identifies that which station has transmitted the frame on wireless medium. Transmission address may or may not be the source address.

Basic service set ID (BSSID)

BSSID address is the MAC address used by the wireless interface in the access point. If AP is receiving the frame then its receiver address will be the MAC address of AP.

Case 1: If from DS = 0 and To DS = 0

address 1: Destination

address 2: Source

address 3: BSSID

Case 2: If from DS = 0 and To DS = 1

address 1: BSSID

address 2: Source

address 3: Destination

Case 3: If from DS = 1 and To DS = 0

address 1: Destination

address 2: BSSID

address 3: Source

Case 4: If from DS = 1 and To DS = 1

address 1: Receiver

address 2: Transmitter

address 3: Destination

address 4: Source

Sequence control

This 16 bit field is used for defragmentation and discarding the duplicate frame. It consists of 4 bit fragment number and 12 bit sequence number.

Fragment number	Sequence number

Frame Body

This is also called data field. 802.11 can transmit maximum of 2304 bytes of higher layer data.

Frame check sequence (FCS)

When frame is sent over wireless medium, before transmission FCS is calculated. At the receiver end, FCS is calculated from the frame received and compared with the FCS of the frame received. If both are same frame is not corrupted otherwise corrupted.

11 Meshap Architecture

11.1 Meshap components

11.1.1 Access Point Thread

11.1.2 Mesh Table

11.1.3 Mesh Heart Beat Processing Hash Table

11.1.4 Mesh name Hash Table

11.1.5 Station Hash Table

11.1.6 Access Point Vlan Hash Table

11.1.7 Access Point Indirect Vlan Hash Table

11.1.8 Parent Hash Table

11.1.9 DS MAC Hash Table

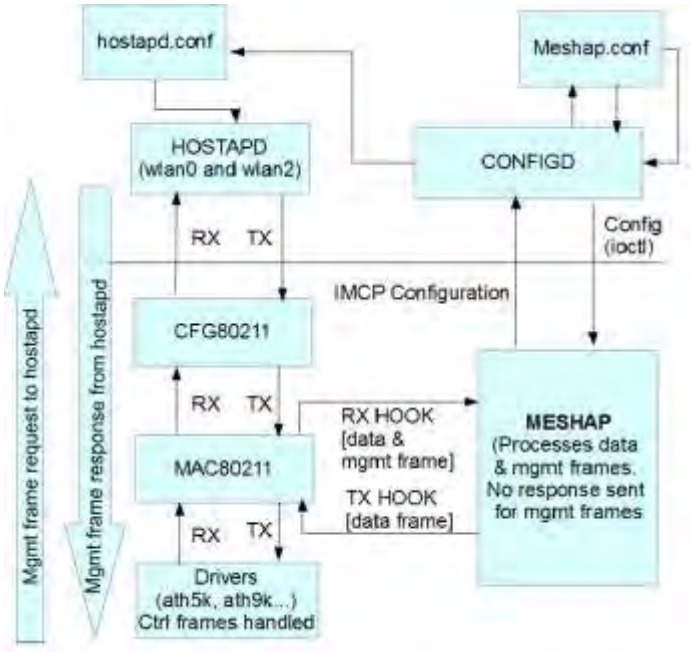
11.2 Mesh Init Sequence

12 Software Architecture

The MeshDynamics Mac80211 based Mesh Networking architecture defines a design approach where the proprietary Mesh Networking algorithm provided by MeshDynamics will be fully integrated with the Linux based MAC80211 architecture.

It provides the complete abstraction of the proprietary Mesh Networking algorithm (meshap) from the underlying device drivers and therefore the dependency of meshap on any of the underlying device drivers will be removed.

If the underlying device driver changes, the meshap continues to provide services without any impact and no modification is required in the code. The goal is to support all the functionalities provided by the existing MeshDynamics algorithm and there shall be no interface level impact on the existing meshap algorithm.



The block diagram of the Mesh Dynamics' Mac 80211 based Mesh Networking architecture is above, Fig 17.

The **Error! Reference source not found.** describes the existing Linux based Mac80211 architecture which is integrated with the Meshap. The diagram shows the packet flow between the blocks via MAC80211 to the Meshap.

11.1 Device Drivers

It is the underlying WLAN device drivers like atheros 5k, atheros 9k in the linux kernel. The packets received by the atheros devices are processed by the device driver layer and handed over to MAC80211 block. These are the standard drivers of linux and are not impacted with the integration of meshap and MAC80211..

11.2 MAC80211

The mac80211 is a framework which driver developers can use to write drivers for SoftMAC wireless devices. mac80211 implements the cfg80211 callbacks for SoftMAC devices. Device drivers call the routines of MAC80211 to hand the received packet. The Mac80211 block processes this packet and at one point calls the Rx hook which will redirect the data and management frames to the meshap block for processing. The response from meshap is passed back to mac80211 by calling the Tx hook function. For management frames, one copy is

sent to meshap and the other copy is sent to hostapd. The response for these management frames is only sent back from hostapd while meshap only processes the frames and updates its database. For more details on MAC80211 architecture, refer to section 9.

11.3 cfg80211

cfg80211 is the Linux 802.11 configuration API. The netlink 'nl80211' driver is used to configure a cfg80211 device and is used for communication between kernel and userspace. This layer is responsible to configure MAC80211 based on callbacks from hostapd in userspace. There is no change in this block with integration of meshap with mac80211.

11.4 Hostapd

Hostapd is a user space daemon for access point and authentication servers. It configures the wireless interfaces (using netlink sockets) for MAC80211 based system. It implements IEEE 802.11 access point management, IEEE 802.1X/WPA/WPA2/EAP Authenticators, RADIUS client, EAP server, and RADIUS authentication server It depends on hostapd to handle authenticating clients, setting encryption keys, establishing key rotation policy, and other aspects of the wireless infrastructure. Hostapd is run on the interfaces configured as AP (wlan2) and on the downlink interface (wlan0). Hostapd handles the management frames and sends the response for the management frames it receives.

11.5 Configd

It is a user space daemon used by Meshap to process configuration requests from Meshap in the user space. The configd daemon is used for the boot-time as well as the runtime configuration of meshap. When meshap boots up, configd daemon configures and start the meshap. On receiving a configuration change parameter configd initiates IOCTL command to configure that parameter to meshap and also applies this change to mac80211 via hostapd by modifying the hostapd.conf configuration file and issuing a SIGHUP to the hostapd daemon.

11.6 Meshap

The meshap block is the complete mesh networking core algorithm provided by Mesh Dynamics.

13 Functional Description

13.1 Overview

The mesh nodes have four interfaces whose usage is configuration dependent i.e. they can act as an Uplink, Downlink, Access point or scanning interfaces. The generally accepted usage is - wlan0 (Downlink interface), wlan1 (Uplink interface), wlan2 (Access Point Interface) and wlan3 (Scanning interface). The Downlink Interface and the Access point interfaces runs hostapd daemon over themselves. Hostapd running on them will be responsible to configure the underlying mac80211 block and the device drivers.

The Uplink and the scanning interface will not run hostapd and therefore will be configured by the 'iwconfig' user space utility.

Whenever a management frame is received by the mac80211 block, a copy of this frame is sent to hostapd as well as to the meshap using hook functions. On receiving this frame, both hostapd and meshap will process the frame. The hostapd updates its data structures and also configures the mac80211 block and sends the response (if any). Meshap only updates its data structure and does not send any response at all.

The control frames are handled by the device drivers only. They never reach meshap or hostapd.

The data frames are only processed by the meshap.

13.2 Boot time Initializations

13.2.1 Meshap Data structure Initializations

At the start-up, the MAC80211 and the device drivers are loaded. The initialization of meshap is not done in kernel space. It is delayed until the system is completely up and is triggered from the user space. This is required to have a controlled way to configure both meshap and hostapd, mac80211 & the device drivers. In order to configure the mac80211 block, hostapd daemon is required. It uses the hostapd.conf configuration file with which it can configure the devices and mac80211 block.

The hostapd daemon is run on the access point interface (generally wlan2) and the downlink interface (generally wlan0) on the mesh nodes. The other two interfaces, i.e. the uplink (generally wlan1) and scanning interface (wlan3) will not run hostapd.conf. These two interfaces are configured using the 'iwconfig' user space too.

12.2.2 Meshap hook for diverting packets

The purpose of the hook functions in the RX and TX path is to serve the packet redirection to and from the meshap. Until the hook functions are not registered by the meshap, the MAC80211 block will not handle any packet which is received. It keeps dropping those packets. Once Meshap registers the hooks, meshap can immediately start handling the packets and process them accordingly. The hook function acts as a bridge that takes care of translating the packets in the format as needed by the existing Meshap block, thus keeping meshap abstracted after its integration with MAC80211. When a packet is to be handed over to MAC80211 by meshap, the hook function converts the packet in the format which MAC80211 can process.

12.2.3 Meshap Runtime configuration

The interface configurations can change at run time through the Network management Viewer. These configuration change messages are received by meshap as an IMCP packet. The following steps are followed on receiving a configuration change message:

1. Meshap receives the "IMCP_SNIP_PACKET_TYPE_CONFIGURATION_INFO" configuration IMCP message and sends the configuration parameters buffer to configd
2. Configd updates the meshap.conf file

3. Configd creates the hostapd.conf file based upon the interface's usage type.

4. Configd sends ioctl to meshap. Meshap configures its data structures.

5. Configd signals the running hostapd daemons with a SIGHUP. On receiving the SIGHUP signal, hostapd reloads its configuration.

6. Configd configures the uplink and scanning interfaces with iwconfig utility.

12.3 Packet Handling

12.3.1 Management Packets

Management packets are processed by hostapd/mac80211 as well as by meshap. Meshap will receive a copy of the management frame. The response for the management frame will be sent by hostapd/mac80211. The responses from meshap will be blocked.

12.3.2 Control Packets

Control packets will be handled by mac80211. Meshap will not process any of these frames.

12.3.3 Data Packets

Data Packets are processed only by meshap. Meshap transmits the packet out using the mac80211 hooks. Mac80211 will take care of sending the packets out of the appropriate interface and the corresponding driver.

12.3.4 Packets to mip interface

The Meshap IP device (MIP) is a virtual IP interface which is initialized to handle L3 packets like ARP Packets and IP packets. The IMCP packets which are handled by the configd daemon are also received on the mip interface on which the configd daemon binds the socket to receive packets. A MIP interface processes data packets.

Mip device creation:

On start-up, in the Init phase of meshap initialization, when the INIT_MESH_CMD ioctl is invoked, the meshap_ip_device_initialize routine is invoked which creates and registers an interface with a name mip0. The mip interface is associated with its mip_priv_data private data structure. The _mip_open routine initializes the IFF_RUNNING flag and the mip interface is ready to process packets.

A packet is sent to mip interface by the routine al_send_packet_up_stack. This function passes the packet received via interface al_net_if to the OS networking stack using the following rules with the given order:

1. For IP packets (type 0x800), if the destination MAC address is broadcast or multicast, the packet is sent up the OS networking stack via the MIP interface.

2. For IP packets (type 0x800), if the destination IP address matches the IP address of the MIP interface, the packet is sent up the OS networking stack via the MIP interface.

3. For ARP packets (type 0x806), if the target IP address matches the IP address of the MIP interface, the packet is sent up the OS networking stack via the MIP interface.

4. The packet is sent up the OS networking stack via the received network interface in all other cases

12.3.5 Packets from mip interface

When the network stack transmits a packet via the MIP interface it calls the on_before_transmit routine which en-queues the packet into the fifo queue. The access points thread de-queue's packets from this queue and further processes the packet to make decision where to send the packet i.e. on WM or DS.

The configd daemon sends responses of IMCP packets via the mip interface

12.4 Packet handling for virtual interfaces

Virtual operation modes are used when one physical radio interface is used as two or more logical radio interfaces. The driver shall process calls to this function only if the device type has been set to MESHAP_DEV_MODE_MIXED in a call to the set_device_type function.

On reception of a frame, meshap checks if it is running in a mixed mode by checking if the device_mode is set to ATHEROS_DEV_MODE_MIXED. If yes, then each frame is handled based on the sub-type. For example, in case of management frames, if the frame sub-type is PROBE_REQ, the meshap is assumed as a master mode and handled by atheros_sta_fsm_process_mgmt_frame. Similarly if a frame sub-type is ASSOC_RESP or REASSOC_RESP, meshap is assumed to be in infra mode and frames are handled by meshap_process_mgmt_frame.

For data frames also, meshap checks that if it is in a mixed mode or not. If yes then based on the Torna header flags, sets it current mode as IW_MODE_INFRA or IW_MODE_MASTER and handles the frames accordingly.

The control frames in the existing MeshDynamics architecture are handled by the device driver. After integration, the control frames will be handled by the mac80211.

13 IMCP message handling

IMCP stands for Infrastructure Mesh Control Protocol, used and managed in Mesh networks for configuring network parameters and through IMCP the mesh nodes will also form the hash table of underlying nodes with which it can communicate, this is being done by heartbeat messages received by IMCP.

Some of the IMCP messages are processed in kernel and others in userspace by configd daemon.

Meshap receives IMCP messages from Network viewer and configures the specified parameters and in some cases responses back to the network.

The below figure shows format of IMCP packet structure

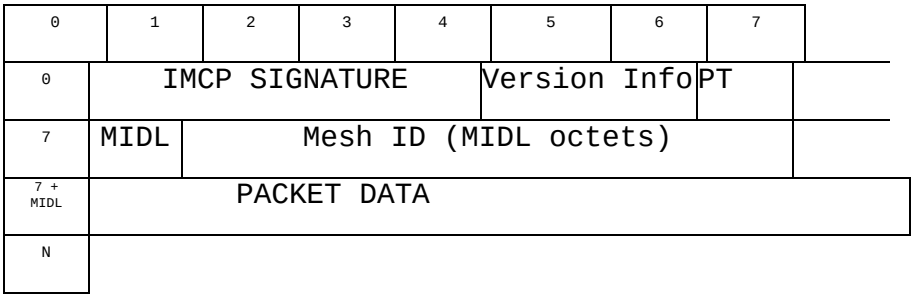


Figure 17 IMCP packet format

PT: PT stands for packet type Maximum value of this is 127, where Most Significant Bit used as flag for encryption.

MSB = 0(encryption disabled)

MSB =1(encryption enabled)

Packet Data is encrypted depending on encryption flag. E.g. of different Packet Types are

- Handshake request
- AP Hardware Info

Version Info: Version Info includes 1 byte major version followed by 1 byte minor version.

MIDL: MIDL stands for Mesh ID

IMCP SIGNATURE: IMCP SIGNATURE is 4 byte field, when the IMCP messages are received the signature part version and packet type is checked before processing the packet. it usually contains 'I', 'M', 'C','P' as signature.

PACKET DATA: It contains the Packet data e.g. If it is a Heartbeat packet then it contains information about the Heartbeat packet.

Examples of IMCP Messages:

- IMCP_SNIP_PACKET_TYPE_HEARTBEAT: This message processes Heartbeat packet.
- IMCP_SNIP_PACKET_TYPE_STA_ASSOC_NOTIFICATION: This message gives notification of station association.
- IMCP_SNIP_PACKET_TYPE_STA_DISASSOC_NOTIFICATION: This message gives notification of station disassociation.

There is no change in the design part with respect to IMCP; it is used the same way as in MeshDynamics.

14 Design Details

13.1 Boot time Initializations

The meshap is initialized and started up in three phases as described below:

- **Init Phase –** Meshap maintains a 'meshap_core_net_if_t' global structure for keeping information of all the devices which are created (like wlan0, wlan1...). At startup, the configd daemon initiates an INIT_MESH_CMD ioctl command which is handled by meshap and meshap ipopulates the meshap_core_net_if_t data structure. This structure is used by meshap for managing net devices. Figure 18 describes the init Phase.

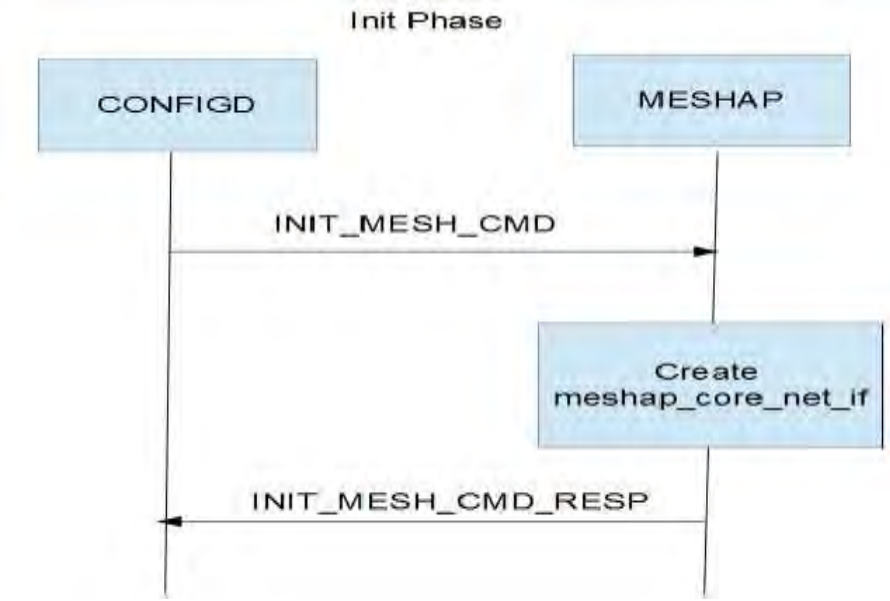


Figure 18 Meshap Init Phase

- **Configuration Phase –** Meshap uses a configuration file (meshap.conf) which contains the complete information which is required by meshap to configure each device present on the system as well as the meshap's implementation specific parameters like "heartbeat interval", "preferred parent", "model" etc that are required by meshap. This configuration file is placed at path /etc/meshap.conf on the file system. Meshap generates the hostapd.conf file which will be used by hostapd as its input configuration file. The configuration read by from the meshap.conf files is also used to configure the Uplink and the scanning interfaces. The following steps are followed in the configuration phase:

1. The configd daemon reads the meshap.conf configuration file and populates its global meshap_device_conf_t data structure with all the information present in the meshap.conf file.

2. Configd generates the hostapd.conf file for interfaces whose usage type is "wm" in the meshap.conf file

3. Configd configures the meshap by passing configurations using SET_CONFIG_<TYPE>_CMD ioctl, where TYPE is the type of configuration parameter which is to be configured. For example, if configd needs to set the RTS threshold parameter, it issues a SET_CONFIG_RTS_TH_CMD to meshap which updates this parameter to its data structure.

The Figure 19shows configuration phase.

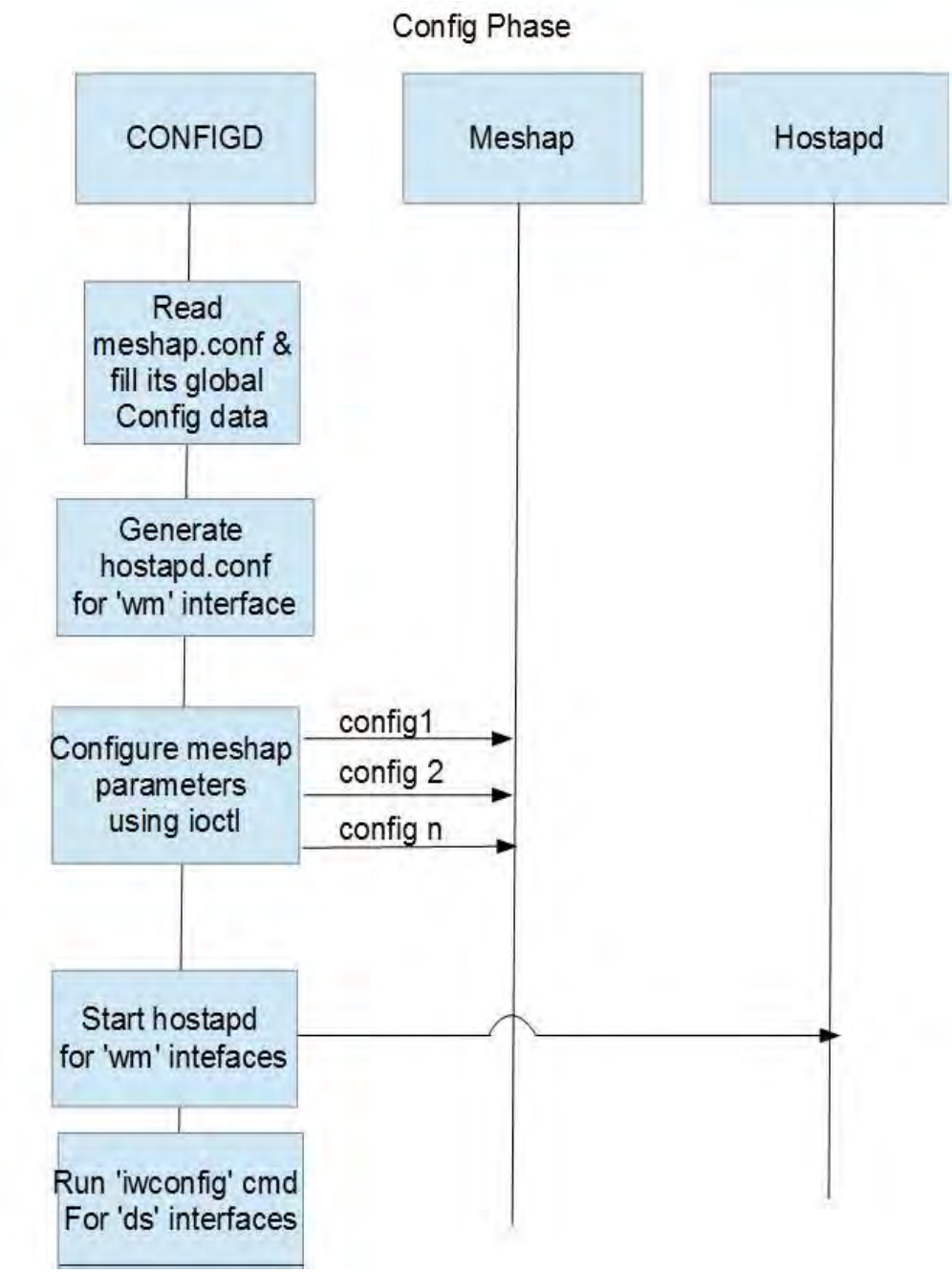


Figure 19 Config Phase for Meshap Init

Upon successfully configuring the parameters to meshap, the configd daemon will issue the 'iwconfig' command for the interfaces whose usage type is 'ds' in the meshap.conf configuration file. Configd prepares a command type like 'iwconfig <interface name> rts <rts value>' and this command will be executed using the 'system' API call.

- **Starts Phase –** Upon successful completion of the configuration phase, the configd daemon initiates the next step of starting the meshap and the hostapd daemons. The registration of hook functions for the redirection of packets to and from meshap is also done in this phase. The following steps are followed to bring the entire system up and running:

1. Configd issues the START_MESH_CMD ioctl to meshap and meshap registers its TX and RX hook functions with mac80211 block.
2. Meshap starts its access point thread and marks its state as _MESH_STATE_RUNNING .

Figure 20 shows the start phase.

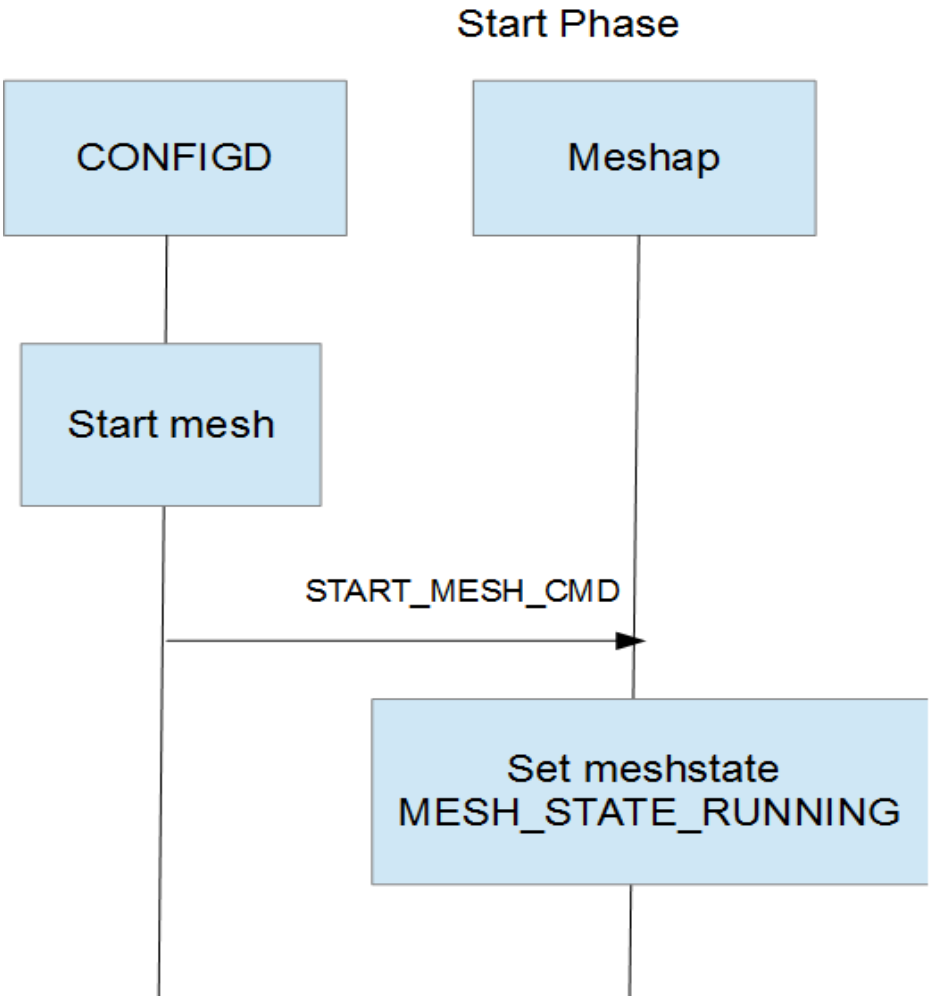


Figure 20Start Phase of Meshap

After the start phase, the system is considered to be completely up and it can start handling any packet it receives.

13.2 Packet Path handling

During initialization of ath5k driver, various tasklets are initialized. ath5k_tasklet_rx and ath5k_tasklet_tx are the tasklets initialized for processing for received frame at the interface and processing of TX frame which is already transmitted respectively.

When frame is received at wireless interface card and copied to driver memory, interrupt is generated to handle the frame. In interrupt context decision is made on the basis of status of the frame. If interrupt received is fatal error, then work queue is scheduled to reset the interface card in order to prevent the error. Since fatal errors are irrecoverable so the only

option is to reset the card. If for previously transmitted frame, if interrupt received is TX_ERR or TX_OK then ath5k_tasklet_tx tasklet will be scheduled. If frame is transmitted successfully then it will be freed and in case of error same frame will be reused. If the received interrupt is for frame received successfully then tasklet for Rx will be scheduled and ath5k_tasklet_rx will called as soon as interrupt will be returned. If the received descriptor is not pending for the processing from wireless interface card then the memory buffer containing the frame will be unmapped from preventing further DMA operation and new memory buffer will be allocated and mapped to device.

To process the frame received at interface, tasklet function ath5k_tasklet_rx is called in interrupt context. This tasklet function calls the ath5k_receive_frame_ok to verify that receive frame is not invalid frame. If the frame received is valid then it returns TRUE otherwise false. If the frame received is valid then it calls ath5k_receive_frame. From the received frame status it finds the current channel centre frequency, frequency band, signal and antenna and the update the received signal strength index (rssi) if the received frame is beacon. During frame processing another function ieee80211_invoke_rx_handlers is called. This function checks the retry bit and sequence number of the received frame. If the frame is duplicate frame then it is dropped. Also it filters the frame based on the station auth/assoc status. It drops the frame from non-associated station.

For further processing of the received frame receive path handler of mac80211 is called. This is the function called by low level driver when 802.11 MPDU is received from the hardware. This function internally calls the actual Rx frame handler __ieee80211_rx_handle_packet. This must be called with rcu_read_lock protection. Based on the type of frame received and virtual interface type of the interface which received the frame various function is called to process the frame further which is discussed below.

TX frame handling:

When meshap needs to sends the TX frame through the device interface, it calls dev_queue_xmit. The function dev_queue_xmit calls the function __dev_queue_xmit internally. This function queue a buffer for transmission to a network device. This function can be called from interrupt context. It calls netdev_pick_tx to choose the TX queue for transmission. If there is queuing discipline for the network device then it calls the __dev_xmit_skb with skb, txq, and dev as input parameters. After the en-queuing the frame to queue, this function calls the __qdisc_run to transmit the frame. __qdisc_run internally invokes the qdisc_restart and pass the queue as input parameter. This de-queue the one frame from queue and calls the sch_direct_xmit function. sch_direct_xmit calls the function dev_hard_start_xmit to transmit the frame with input parameters frame pointer skb and net-device pointer dev. The function dev_hard_start_xmit calls the driver defined transmit function ops->ndo_start_xmit which is registered with kernel. This is the actual function which is use to add the frame directly to dma queue.

If the device has no queue (loopback, tunnels) then in this case it calls dev_hard_start_xmit and called interface internally calls the driver's Tx function to transmit function to transmit the frame.

13.2.1 Torna Header Handing

Torna header is an important header for frame processing. Torna header updating is valid only when frame is processed by meshap. Whenever frame is received from the interface, Torna header is updated from the available control information from the frame and sends to meshap for further processing. Similarly whenever frame is transmitted from meshap, Torna header is stripped and sends to the wireless driver for the transmission. Torna header updating and stripping is done in mac80211 in linux kernel.

In struct sk_buff, the structure element mac_header is used for the processing of Torna header. For the reception of frame, along with frame size of memory, Torna header memory is also allocated.

```
skb->mac_header = (torna_mac_hdr_t*)kmalloc(sizof(torna_mac_hdr_t), GFP_KERNEL);
```

when frame is received at the interface, it is processed by the driver. It calls the update_meshap_torna_header function exported by mac80211 for the updating of Torna header. First it updates signature and type of the Torna header. Without the Torna header signature meshap will not process the frame.

```
hdr = (torna_mac_hdr_t*)skb->mac_header;

hdr->signature[0] = TORNA_MAC_HDR_SIGNATURE_1;

hdr->signature[1] = TORNA_MAC_HDR_SIGNATURE_2;

hdr->type = TORNA_MAC_HDR_TYPE_SK_BUFF;
```

After updating the signature and type, frame control field of the frame is traversed and parameters like various addresses of frame, rssi, tx_rate and various meshap header flags are maintained.

13.2.2 Management Packets

To send the management frame to meshap in case of AP, a hook process_meshap_mgmt_frame is called in function ieee80211_rx_h_userspace_mgmt defined in mac80211. These frames are also sent to the hostapd in case if interface works as AP. For sending the frame to user space, cfg80211_rx_mgmt is called. It further calls the nl80211_send_mgmt function. This function further allocates the page size or 8kb of memory and update the netlink header and copy frame content to allocated memory. The allocated frame is then added to the end of the receiving socket queue.

The entire frame belonging to management frame will be transmitted to hostapd and meshap. The hostapd daemon will process the frame and send the response to the station while meshap process the frame and maintains it data structure. Meshap will not transmit the response frame as response is already sent by hostapd.

13.2.3 Control Packets

The entire control frame is handled by Linux kernel. This is handled by the function ieee80211_rx_h_uapsd_and_pspoll and ieee80211_rx_h_ctrl. If mobile station has no data to send to distribution system, it sends null frame with power management bit set in the frame control field. This indicates the change in power status of the station. So null frame sending station is

marked as sleep and set the station status WLAN_STA_PS_STA. Station will wake up periodically to check if there any available data to receive. For this station sends PS-POLL frame to AP. If there is any data corresponding to the station which sends PS-POLL frame, AP will release the buffered data to the wake up station.

Another example of control frame is RTS, CTS and ACK frame. If station wants to transmit bulky data, before transmission it send request to send (RTS) frame to the AP. Upon receiving the control to send (CTS) frame station will transmit the required data.

13.2.4 Data Packets

The entire data frame should be processed by meshap. Meshap is the controlling entity which will decide the fate of the frame. If it is consumed by the host then frame will be handover to the network stack otherwise frame will be forwarded to other node based on mesh routing

13.3 Meshap APIs with the mac80211

Meshap has exported quite a few symbols, which are currently used by the ath5K drivers. The sections below list down the various symbols exported by meshap and how they will be used w.r.t. mac80211 code

13.3.1 meshap_get_board_temp

This function is unimplemented in meshap code. It returns 0 by default. Hence, this API will not be called from mac80211 code.

13.3.2 meshap_get_board_voltage

This function is unimplemented in meshap code. It returns 0 by default. Hence, this API will not be called from mac80211 code

13.3.3 meshap_set_led_on

This function calls the led_brightness_set api of linux stack with argument LED_FULL. Meshap will continue to use this API. In all the code paths where LED is set to ON, the same function will be called in the mac80211 code path also.

13.3.4 meshap_set_led_off

This function calls the led_brightness_set api of linux stack with argument LED_OFF. Meshap will continue to use this API. In all the code paths where LED is set to OFF, the same function will be called in the mac80211 code path also.

13.3.5 meshap_set_led_blink

This function calls the led_blink_set api of linux stack with the 'delay' argument specifying the frequency of blink as 1000. Meshap will continue to use this API. In all the code paths where LED is set to BLINK, the same function will be called in the mac80211 code path also.

13.3.6 meshap_set_led_blink_fast

This function calls the led_blink_set api of linux stack with the 'delay' argument specifying the frequency of blink as 200. Meshap will continue to use this API. In all the code paths where LED is set to BLINK, the same function will be called in the mac80211 code path also.

13.3.7 meshap_set_led_blink_once

This function calls the led_brightness_set api of linux stacks with argument as LED_OFF and modifies the timer to expire after 1000 hz. Meshap will continue to use this API. In all the code paths where LED is set to BLINK, the same function will be called in the mac80211 code path also.

13.3.8 meshap_enable_reset_generator

This function is not implemented in the meshap code and hence won't be called from mac80211 code as well.

13.3.9 meshap_strobe_reset_generator

This function is not implemented in meshap code and hence won't be called from mac80211 code as well.

13.3.10 meshap_get_gpio

This function is unimplemented in meshap code. It returns 0 by default. Hence, this API will not be called from mac80211 code.

13.3.11 meshap_set_gpio

This function is unimplemented in meshap code. It returns 0 by default. Hence, this API will not be called from mac80211 code.

13.3.12 meshap_get_gps_info

This function gets the gps location of meshap. Meshap will continue to use this API.

13.3.13 meshap_set_gps_info

This function sets the gps location of meshap. Meshap will continue to use this API.

13.3.14 meshap_process_mgmt_frame

This api is called by the atheros driver when it receives a management frame and is used to pass the frame to meshap for processing. With the mac80211 code, a hook will be added which will generate the copy of the frame and give the frame to meshap for processing. Meshap will then call meshap_core_process_mgmt_frame and process the management frames.

13.3.15 meshap_process_data_frame

This api is called by the atheros driver when it receives a data frame and is used to pass the frame to meshap for processing. With the mac80211 code, a hook will be added give the frame to meshap for processing. Meshap will then call meshap_core_process_data_frame and process the data frames.

13.3.16 meshap_on_link_notify

The api is called from meshap code from the following code paths

a. When a station gets associated / disassociated, then this meshap hook gets called in case of relay node processing for the uplink interface. This is because the uplink interface of the mesh node acts like a station.

b. Meshap registers with the netdev notifier to get updates about the state of the link. Any change in the state of the link is notified via meshap callback handler which ends up calling the above API.

c. In the previous code, the driver has a callback registered for on_phy_link_notify_watchdog. <TBD: need to check where it happens now>

13.3.17 meshap_on_net_device_create

This api is called when a mip0 device creation notification is received from linux. It creates the core_net_if_t structure when a mip0 device is registered (on receiving NETDEV_REGISTER event) or on receiving NETDEV_CHANGE event.

For wireless interfaces, this api is currently called by the atheros_attach api.

For integration with mac80211, meshap_on_net_device_create will be called for wireless interfaces on trigger from configd.

13.3.18 meshap_on_net_device_destroy

This api is called when a mip0 device is un-registered (on receiving NETDEV_UNREGISTER event). Meshap removes the entry for this device from core_net_if_t structure.

13.3.19 meshap_get_sta_info

Wireless Network drivers shall call this function to obtain information about a destination mac-address from the meshap LKM. The most common use of this function is in the "**hard_header**" handler for the **net_device**. The Kernel calls the **hard_header** handler, when sending packets from the stack directly through the network device.

With mac80211 integration, it is not required to call this API as the drivers get this information directly from mac80211.

13.3.20 meshap_reboot_machine

This function is called by the atheros_tx routine of meshap when the TX buffer pool overflows. In this case meshap needs to be restarted. For integration with mac80211, this would not be required.

13.3.21 torna_hw_id_get_address

This api is unimplemented in meshap. It is used to set the mac address of the device. The setting of mac addresses of the devices will be handled by standard wireless drivers.

13.3.22 torna_get_product_oui_id

This api is unimplemented in meshap and not called by meshap. . Hence, this API will not be called from mac80211 code.

13.3.23 torna_get_generic_id

This api is unimplemented in meshap and not called by meshap. Hence, this API will not be called from mac80211 code.

13.3.24 torna_put_reboot_info

This api is unimplemented in meshap. It is called from _meshap_panic_event and meshap_die_event APIs. . Hence, this API will not be called from mac80211 code

13.3.25 torna_get_reboot_info

This API populates the '"reboot" proc entry.

13.4 Meshap hook functions registered with the driver

In the current code, meshap registers various hook functions with the drivers, which are then called by the driver explicitly to notify meshap of the various events. The sections below list down the various hook functions, their functionality in the current code and how the same functionality will be achieved with the mac80211

13.4.1 round_robin_hook

The hook is registered by meshap to the driver, so that the driver can inform meshap when it receives beacons from the AP. This is set by the function _meshap_net_dev_set_round_robin_notify_hook().

13.4.2 probe_request_hook

The hook is registered by meshap to the driver, so that driver can inform meshap whenever the station sends the probe request to the AP. This will be required in the case of relay node on the uplink interface, where the interface acts like a station. It is set by the function

_meshap_net_dev_set_probe_request_notify_hook()

13.4.3 radar_hook

<TBD: Need some inputs>

13.5 Meshap netdev ops

13.5.1 set_hw_addr

In the current implementation a vector for set_hw_addr is registered in the meshap_net_dev_t structure. This vector calls the ath_hal_setmac api to set the mac address in the device.

With the mac80211 code, each device registers with net_dev a function to set the mac address.

set_hw_addr will be modified to call the ndo_set_mac_address from the net_dev directly.

13.5.2 associate

In the current implementation, when the meshap wants to send the associate request to the parent, it calls ath5K routine to send the associate request.

ret = atheros_sta_fsm_join(instance,essid,length,bssid,channel,ie_in,ie_in_length,ie_out);

However, now since the intention is to make the code independent of the ath5k specific drivers, the code will be changed to call the cfg_80211 routines to generate the association request. During the boot time, drivers call the ieee80211 specific initializations where they register the ops specific to mac80211. Hence, those ops will be called to handle send the association request from a particular interface.

err = rdev->ops->assoc(&rdev->wiphy, dev, &req);

13.5.3 dis_associate

In the current implementation, when the meshap wants to send the dis-associate request to the parent, it calls ath5K routine to send the dis-associate request

ret = atheros_sta_fsm_leave(instance,1,WLAN_REASON_UNSPECIFIED);

However, now since the intention is to make the code independent of the ath5k specific drivers, the code will be changed to call the cfg_80211 routines to generate the association request. During the boot time, drivers call the ieee80211 specific initializations where they register the ops specific to mac80211. Hence, those ops will be called to handle send the association request from a particular interface.

rdev->ops->disassoc(&rdev->wiphy, dev, &req, wdev);

13.5.4 get_bssid

In the current implementation, when the meshap wants to get the bssid, it copies same from the driver's instance structure associated with the wireless device.

The code used to do the following:-

```
memcpy(bssid,instance->current_bssid,ETH_ALEN);
```

Now the code, will be modified to get the value from the ieee80211_ptr associated with the netdev.

It will be fetched using the following:-

```
struct wireless_dev *wdev = dev->ieee80211_ptr;

memcpy(bssid, wdev->current_bss->pub.bssid, bssid, ETH_ALEN) ;
```

13.5.5 scan_access_points

In the current code, there is a function written for the scanning, the result of which is used for the determining the active parent for the relay node. In this code, the thread sleeps till the response for message sent arrives and the response is used for figuring out the results.

In mac80211, the scan code is quite different. Mac80211 provides various APIs to start the scanning and notify when the call-back gets completed. The scan code in Meshap needs to be changed to be able to get integrated with the mac80211 code.

13.5.6 scan_access_points_active

In the current code, there is a function written for the scanning, the result of which is used for the determining the active parent for the relay node. In this code, the thread sleeps till the response for message sent arrives and the response is used for figuring out the results.

In mac80211, the scan code is quite different. Mac80211 provides various APIs to start the scanning and notify when the call-back gets completed. The scan code in Meshap needs to be changed to be able to get integrated with the mac80211 code

13.5.7 scan_access_points_passive

In the current code, there is a function written for the scanning, the result of which is used for the determining the active parent for the relay node. In this code, the thread sleeps till the response for message sent arrives and the response is used for figuring out the results.

In mac80211, the scan code is quite different. Mac80211 provides various APIs to start the scanning and the notify them, when the call-back gets completed. The scan code in Meshap needs to be changed to be able to get integrated with the mac80211 code

13.5.8 get_last_beacon_time

In the existing code, this is implemented by fetching the value from the driver instance structure.

However, the "op" is not getting called from meshap. Hence, the implementation of this op will return 0.

13.5.9 set_mesh_downlink_round_robin_time

This is used to set the configuration parameter, for round_robin_time. This is being used by the driver to set the time interval of sending message to various stations connected to the Access point.

Now, since the messages to the various stations will be triggered by hostapd, meshap won't have any role to play for this. This will be stubbed out in the current meshap code.

13.5.10 add_downlink_round_robin_child

This is used by the meshap code, to add the station to the driver. This is done as part of the processing of the association messages received from the stations. In the current code, meshap is directly adding the child to the driver.

However, now informing about the child to the driver will be managed by hostapd and meshap doesn't need to worry about it.

Hence, this function will be stubbed out.

13.5.11 remove_downlink_round_robin_child

This is used by the meshap code, to remove the station from the driver. This is done as part of the processing of dissociate message, or for any other reason meshap was to disassociate the child.

However, now informing about the child to the driver will be managed by hostapd and meshap doesn't need to worry about it.

Hence, this function will be stubbed out.

13.5.12 virtual_associate

<TBD: Need some inputs >

13.5.13 get_duty_cycle_info

<TBD: Need some inputs >

13.5.14 set_rate_ctrl_parameters

<TBD: Need some inputs >

13.5.15 reset_rate_ctrl

<TBD: Need some inputs >

13.5.16 get_rate_ctrl_info

<TBD: Need some inputs >

13.5.17 set_round_robin_notify_hook

<TBD: Need some inputs >

13.5.18 enable_ds_verification_opertions

<TBD: Need some inputs >

13.5.19 dfs_scan

<TBD: Need some inputs >

13.5.20 set_radar_notify_hook

<TBD: Need some inputs >

13.5.21 get_mode

In the current implementation, this function is used to fetch the mode from the instance structure. This is used by meshap during the init time.

With the mac80211 code, the implementation of the API, will be changed to fetch the value from the meshap's private structure stored in the ieee80211_hw structure.

13.5.22 set_mode

In the current implementation, this function is used to set the mode for each of the interfaces in the instance structure associated with the driver. This is done during the init time.

With the mac80211 code, the mode will be computed and set in the driver's ieee80211_hw structure

13.5.23 get_essid

This function returns the stored essid in the atheros_instance_t structure.When an association frame is received by meshap, it compares the receive ssid information element with the value returned by get_essid. If the ssid matches, association is allowed. Meshap will continue using this API.

13.5.24 set_essid

This function sets the ssid for the interface. This information is obtained from the configuration file and stored in the atheros_instance_t structure. Meshap will continue using this API.

13.5.25 get_rts_threshold

This function returns the rts threshold value which is present in the atheros_instance_t structure. Meshap uses this value while transmitting a packet.

For integration, mac80211 will transmit the packets, and takes care of rts threshold handling.

13.5.26 set_rts_threshold

This function sets the rts threshold value in the atheros_instance_t structure. Meshap will store this information.

13.5.27 get_frag_threshold

This function returns the fragmentation threshold value which is present in the atheros_instance_t structure. In the current implementation the _atheros_setup_packet function gets the frag threshold from the api _atheros_get_fragment_size. Meshap then fragments the packet.

13.5.28 set_frag_threshold

This function stores the value of fragmentation threshold in the atheros_instance_t structure of meshap. This function is called while applying the configuration at startup or in the IMCP message handling.

In mac80211, If the device does the fragmentation itself then "set_frag_threshold" defined by the driver will be called by mac80211 to do fragmentation else mac80211 will itself fragment the packets.

For integration, meshap can send the packet to mac80211 and mac80211 will take care of fragmenting the packet if required.

13.5.29 get_beacon_interval

Meshap never handles the probe response and it never sends the beacon interval. This information is maintained by meshap in the core_net_if structure.

13.5.30 set_beacon_interval

Meshap stores this information in core_net_if structure. Meshap will not send any beacons.

13.5.31 get_default_capabilities

On startup, meshap sets the default capabilities in the "default_capability" field of atheros_instance_t structure. The capability information is used in beacon transmissions to advertise the network's capabilities. Capability Information is also used in Probe Request and Probe Response frames.

Since meshap is not sending beacons, capability information is not used by it.

13.5.32 get_capabilities

On startup, meshap sets the default capabilities in the "capability" field of atheros_instance_t structure. The capability information is used in beacon transmissions to advertise the network's capabilities. Capability Information is also used in Probe Request and Probe Response frames.

Since meshap is not sending beacons, capability information is not used by it.

13.5.33 get_slot_time_type

This field is the part of the capability field. This is only used for beacon frames.

13.5.34 set_slot_time_type

This field is the part of the capability field. This is only used for beacon frames.

13.5.35 get_erp_info

This api returns the erp value. This is not being called in the current code.

13.5.36 set_erp_info

This field is the Effective Radiated power. This field is set only for WLAN_PHY_MODE_802_11_G and WLAN_PHY_MODE_802_11_PURE_G. When meshap receives a beacon, it processes it and gets this information from received beacon. It then compares this information with the stored in the extended_rate_phy_info field of the instance structure. If the value is changed, meshap updates extended_rate_phy_info field the value received in the beacon frame.

After integration, the beacon frames will be received by hostapd and meshap and meshap will handle the erp info . Meshap will not respond to this beacon.

13.5.37 set_beacon_vendor_info

This parameter is set by meshap which is sent in beacon frame. If the node is a root node, then the vendor id used is the DS mac of the root node, if the node is relay node, the vendor id used is parent's bssid. Hostapd will configure this value in the parameter "vendor_elements"

13.5.38 enable_wep

This is used by meshap to set the flag field in instance structure with value ATHEROS_INSTANCE_FLAGS_WEP_ENABLE and if this value is set then meshap updates the 'capability' field of instance structure with WLAN_CAPABILITY_PRIVACY flag.

This field is applicable in the context of sending beacon frames, and since meshap will not send out beacons, it will not be required while integrating with mac80211.

13.5.39 disable_wep

This is used by meshap to clear the flag field in instance structure with value ATHEROS_INSTANCE_FLAGS_WEP_ENABLE.

13.5.40 set_rsn_ie

This is used by meshap to set the flag field in instance structure with value ATHEROS_INSTANCE_FLAGS_RSN_IE_ENABLE and if this value is set then meshap updates the 'capability' field of instance structure with WLAN_CAPABILITY_PRIVACY flag.

This field is application in the context of sending beacon frames, and since meshap will not send out beacons, it will not be required while integrating with mac80211.

13.5.41 set_security_key

This is configured at init time. It will be configured using hostapd/iwconfig. Meshap uses this to get the key during the init time.

Since, initializations will be taken over by the mac80211/hostaps, since meshap is not supposed to configure this.

However, when this routine is called meshap will store the key in its local structure to be able to use the same in processing later on.

13.5.42 release_security_key

In the current meshap code, it is called when the key needs to be released from the driver. Now, this functionality should be handled by mac80211 and/or hostapd. Meshap shouldn't have any role to play in this.

However, when this routine is called meshap will remove the key info from its local structure.

13.5.43 get_security_key_data

In the current meshap code, it is called in the processing of the auth frames.

Now, this functionality will be handled by mac80211 and/or hostapd. However, since meshap will also receive the auth frames, process it and send the auth response, it will do so using the key info stored in the local structure.

However, the auth frames, will be blocked in the end from being transmitted.

The implementation will fetch the values from the local structures.

13.5.44 set_ds_security_key

In the current meshap code, it is called on the relay node to set the encryption key based on the data received from the parent.

With the current code, the mac80211 APIs will be called directly to set the key in the driver.

13.5.45 get_supported_rates

Meshap initializes the supported rates during its initialization. Meshap uses this value to setup beacon and respond to probe response contents. Meshap also receives this value in probe req and response messages.

Since, the response of the messages will be sent by mac80211 and not meshap, meshap doesn't need this api.

At the same time, every driver sets up this value at the init time. Hence, the data can be fetched if required.

13.5.46 get_extended_rates

Meshap initialized the extended rates during initialization. It then puts this value in the association response. Meshap will not send the management response so no mac80211 api is required. Meshap will just not transmit the association response..

13.5.47 get_bit_rate

Meshap sends the bit rate information in the heartbeat IMCP message. Meshap needs to retrieve this information from its database

13.5.48 set_bit_rate

This is the txrate parameter of each WLAN interface defined by the meshap.conf file. This is a configuration parameter and set via hostapd or iwconfig.

13.5.49 get_rate_table

<TBD>

13.5.50 get_tx_power

Meshap is not calling this.

13.5.51 set_tx_power

On transmission, meshap sets the TX power in the driver. It can be configured using iwconfig when needed

13.5.52 get_channel_count

This api is not being called by meshap.

13.5.53 get_channel

This function gets the current operating channel on an interface. Meshap can call rdev->ops->get_channel which maps to ieee80211_wiphy_get_channel. This will return the current operating channel.

13.5.54 set_channel

This function sets the current operating channel on an interface. Meshap can call rdev->ops->set_channel which maps to ieee80211_set_channel. This will set the current operating channel.

13.5.55 get_phy_mode

This function returns the current phy mode of the device. When the phy mode is 80211G or 80211BG. Meshap uses this information to set the preamble time and erp info for this phy mode.

13.5.56 set_phy_mode

This is being set during the init time to configure the drivers appropriately. However, now the drivers should get set using hostapd/mac80211. Hence, meshap is not supposed to set these fields in the driver.

However, meshap will maintain a local structure and then store the information in the local structure for reference in various scenarios.

13.5.57 get_preamble_type

This is not getting called in the current meshap code. However, we will maintain the value in a local structure in cre_net_if. The value can be fetched using that.

13.5.58 set_preamble_type

In the current meshap code, this is being called during the init time and the value is being set directly into the Atheros structure. Since, this is the part of initialization, the mac80211/hostapd code takes care of setting the appropriate values in the driver. Hence, from the meshap perspective it doesn't require it. However, still the value will be stored in a local structure maintained in core_net_if.

13.5.59 set_dev_token

In the current meshap code, a token is being set into the instance structure of the driver. This structure is used to fetch the local representation of dev from the instance and get all the information quickly.

The token will be set in the ieee80211_hw structure.

13.5.60 set_essid_info

In the current implementation a vector for set_ssid_info is registered in the meshap_net_dev_t structure. The information is later used by the driver to respond to the beacon requests.

However, with the mac80211 code, the beacons will be handled by mac80211 code and meshap will not sending them out. Hence, this function doesn't have much used. However, we will store the information in the device specific structure, so that if any need arises, meshap can easily fetch the value.

13.5.61 get_ack_timeout

In the current implementation meshap does not call this api . Hence, this will be stubbed out.

13.5.62 set_ack_timeout

In the current implementation a vector for set_ack_timeout is registered in the meshap_net_dev_t structure. This vector calls the ath_hal_setacktimeout api to set the ack timeout in the device.

With the mac80211 code, set_ack_timeout will be modified to call the set_coverage_class to set the ack timeout value.

13.5.63 get_reg_domain_info

In the current meshap implementation this is not being called.

13.5.64 set_reg_domain_info

It sets various parameters in the meshap, it is being set during init time.

Need to check whether these parameters can be configured with hostapd and iwconfig n it is being set during init time.

13.5.65 get_supported_channels

In the current implementation get_supported_channels is used to get the supported channel based on phy_mode, It is being called for mesh_imcp_send_packet_supported_channels_info to send the supported channel information.

13.5.66 get_hide_ssid

This is not being called in meshap design.

13.5.67 set_hide_ssid

In the current implementation set_hide_ssid is used to set the hide_ssid is enable or disable into the device. Now the parameter will be set through hostapd and this is moved out from meshap.

In the mac code set_hide_ssid can be done through calling **ieee80211_start_ap(struct wiphy *wiphy, struct net_device *dev, struct cfg80211_ap_settings *params)**

13.5.68 set_dot11e_category_info

In the current implementation set_dot11e_category_info is used to set various parameters such as category, acwmin, acwmax, aifsn, disable_backoff and burst_time.

Now this is being moved out from mesh and set it through hostapd.

13.5.69 set_tx_antenna

In the current implementation set_tx_antenna is set during init time and this is now moved out of meshap and set it through hostapd.

Not able to get the exact parameter in hostapd <TBD>.

13.5.70 set_radio_data

Not being called in meshap

13.5.71 radio_diagnostic_command

<TBD: Need more inputs>

13.5.72 set_probe_request_notify_hook

This is a hook function which meshap will register with mac80211 and mac80211 code will be modified to call this hook whenever probe request will generate.

13.5.73 set_virtual_mode

This is meshap specific function used to set Master and Infra mode for the virtual interface.
Now, we will create virtual interfaces explicitly and enable hostapd/mac80211 to run on the virtual interface itself. Hence, from the mac80211 perspective, it doesn't require the virtual mode.

However, from meshap perspective the current processing will remain. The mode for the interface will be set in the core_net_if structure locally.

13.5.74 set_device_type

In the current meshap code, this is used to set the device type as virtual and the device_mode as MIXED. Since, for the mac80211 code, there will be separate interfaces created and it handles the virtual interfaces, there is no need to set the device type.

However, for the meshap purposes, device_type is required to be set and will be used to determine the interface on which the packet arrived. The device-type and mode will be stored directly in the core_net_if structure, instead of the instance structure as happening currently.

13.5.75 get_device_type

In the current implementation, this is used to get the device type associated with the device on which the packet arrived. Based on the type of device, the physical or the virtual core_net_if is used for the packet processing.

The new implementation will fetch the value from the core_net_if structure and let the callers of the functions take appropriate decisions.

13.5.76 initialize_mixed_mode

In the current implementation, this is used to set the mode in the Atheros driver and reinitialise the driver.

However, with mac80211, this will be managed via hostapd/mac80211 directly and meshap
doesn't need to do anything about this. Hence, the implementation of this function will be stubbed out.

13.5.77 enable_beaconing_uplink

In the current meshap code, through the imcp messages, the uplinks can be enabled to send beacons. This is something which is special to meshap and

will be handled specifically for meshap. Hence, in the mac80211 code, the changes will be done to allow the beacons coming from uplink to be forwarded to meshap for processing. However, mac80211 won't process those beacons.

13.5.78 disable_beaconing_uplink

In the current meshap code, through the imcp messages, the uplinks can be disabled to send beacons. This is something which is special to meshap and will be handled specifically for meshap.

13.5.79 set_action_hook

The set_action_hook is used to process action frames in meshap, so will register set_action_hook with mac80211 to handle these frames.

13.5.80 send_action

The meshap will call Send_action hook to transmit the action frames. The current implementation of the send_action hook calls the atheros driver routine to transmit the frame. Now the implementation will be modified to call the mac80211 transmit routines.

13.6 al_802_11 ops

13.6.1 Associate

This function invokes the corresponding associate callback of meshap_net_dev_ops and is described in the section associate.

13.6.2 dis_associate

This function invokes the corresponding dis_associate callback of meshap_net_dev_ops and is described in the section dis_associate.

13.6.3 get_bssid

This function invokes the corresponding get_bssid callback of meshap_net_dev_ops and is described in the section get_bssid.

13.6.4 send_management_frame

This api sends the management frames out from meshap. Since meshap will only process the management frames, this API will be as it is but will not transmit any management response. For this the transmit calls will be stubbed out in this function.

13.6.5 scan_access_points

This function invokes the corresponding set_access_points callback of meshap_net_dev_ops and is described in the section scan_access_points.

13.6.6 scan_access_points_active

This function invokes the corresponding set_access_points_active callback of meshap_net_dev_ops and is described in the section scan_access_points_active.

13.6.7 scan_access_points_passive

This function invokes the corresponding set_access_points_passive callback of meshap_net_dev_ops and is described in the section scan_access_points_passive.

13.6.8 set_management_frame_hook

This api sets the process_management_frame callback function .Meshap will continue to use this function to set the hook.

13.6.9 set_beacon_hook

Meshap does not implement this function.

13.6.10 set_error_hook

Meshap does not implement this function.

13.6.11 get_last_beacon_time

This function invokes the corresponding get_last_beacon_time callback of meshap_net_dev_ops and is described in the section get_last_beacon_time.

13.6.12 set_mesh_downlink_round_robin_time

This function invokes the corresponding set_mesh_downlink_round_robin callback of meshap_net_dev_ops and is described in the section set_mesh_downlink_round_robin_time.

13.6.13 add_downlink_round_robin_child

This function invokes the corresponding add_downlink_round_robin_child callback of meshap_net_dev_ops and is described in the section add_downlink_round_robin_child.

13.6.14 remove_downlink_round_robin_child

This function invokes the corresponding remove_downlink_round_robin_child callback of meshap_net_dev_ops and is described in the section remove_downlink_round_robin_child.

13.6.15 virtual_associate

This function invokes the corresponding virtual_associate callback of meshap_net_dev_ops and is described in the section virtual_associate.

13.6.16 get_duty_cycle_info

This function invokes the corresponding get_duty_cycle_info callback of meshap_net_dev_ops and is described in the section get_duty_cycle_info.

13.6.17 set_rate_ctrl_parameters

This function invokes the corresponding set_rate_ctrl_parameters callback of meshap_net_dev_ops and is described in the section set_rate_ctrl_parameters.

13.6.18 reset_rate_ctrl

This function invokes the corresponding reset_rate_ctrl callback of meshap_net_dev_ops and is described in the section reset_rate_ctrl.

13.6.19 get_rate_ctrl_info

This function invokes the corresponding get_rate_ctrl_info callback of meshap_net_dev_ops and is described in the section get_rate_ctrl_info.

13.6.20 set_round_robin_notify_hook

This function invokes the corresponding set_round_robin_notify_hook callback of meshap_net_dev_ops and is described in the section set_round_robin_notify_hook.

13.6.21 enable_ds_verification_opertions

This function invokes the corresponding enable_ds_verification_operations callback of meshap_net_dev_ops and is described in the section enable_ds_verification_opertions.

13.6.22 dfs_scan

This function invokes the corresponding dfs_scan callback of meshap_net_dev_ops and is described in the section dfs_scan.

13.6.23 set_radar_notify_hook

This function invokes the corresponding set_radar_notify_hook callback of meshap_net_dev_ops and is described in the section set_radar_notify_hook.

13.6.24 set_probe_request_hook

This is called during meshap initialization and it registers a hook function to process probe requests. Meshap will continue to use this function to set the hook.

13.6.25 get_mode

This function invokes the corresponding get_mode callback of meshap_net_dev_ops and is described in the section get_mode.

13.6.26 set_mode

This function invokes the corresponding set_mode callback of meshap_net_dev_ops and is described in the section set_mode.

13.6.27 get_essid

This function invokes the corresponding get_essid callback of meshap_net_dev_ops and is described in the section get_essid.

13.6.28 set_essid

This function invokes the corresponding set_essid callback of meshap_net_dev_ops and is described in the section set_essid.

13.6.29 get_rts_threshold

This function invokes the corresponding get_rts_threshold callback of meshap_net_dev_ops and is described in the section get_rts_threshold.

13.6.30 set_rts_threshold

This function invokes the corresponding set_rts_threshold callback of meshap_net_dev_ops and is described in the section set_rts_threshold.

13.6.31 get_frag_threshold

This function invokes the corresponding get_frag_threshold callback of meshap_net_dev_ops and is described in the section get_frag_threshold.

13.6.32 set_frag_threshold

This function invokes the corresponding set_frag_threshold callback of meshap_net_dev_ops and is described in the section set_frag_threshold.

13.6.33 get_beacon_interval

This function invokes the corresponding get_beacon_interval callback of meshap_net_dev_ops and is described in the section get_beacon_interval.

13.6.34 set_beacon_interval

This function invokes the corresponding set_beacon_interval callback of meshap_net_dev_ops and is described in the section set_beacon_interval.

13.6.35 set_security_info

This function sets the flags ATHEROS_INSTANCE_FLAGS_WEP_ENABLE or ATHEROS_INSTANCE_FLAGS_RSN_IE_ENABLE flag based on which security configuration is enabled.

This information is later used by driver to setup beacons. Since beacon frames will be handled by mac80211, meshap will stub this api.

13.6.36 set_security_key

This function invokes the corresponding set_security_key callback of meshap_net_dev_ops and is described in the section set_security_key.

13.6.37 release_security_key

This function invokes the corresponding release_security_key callback of meshap_net_dev_ops and is described in the section release_security_key.

13.6.38 get_security_key_data

This function invokes the corresponding get_security_key_data callback of meshap_net_dev_ops and is described in the section get_security_key_data.

13.6.39 set_ds_security_key

This function invokes the corresponding set_ds_security_key of meshap_net_dev_ops and is described in the section set_ds_security_key.

13.6.40 get_default_capabilities

This function invokes the corresponding get_default_capabilities callback of meshap_net_dev_ops and is described in the section get_default_capabilities.

13.6.41 get_capabilities

This function invokes the corresponding get_capabilities callback of meshap_net_dev_ops and is described in the section get_capabilities.

13.6.42 get_slot_time_type

This function invokes the corresponding get_slot_time_type callback of meshap_net_dev_ops and is described in the section get_slot_time_type.

13.6.43 set_slot_time_type

This function invokes the corresponding set_slot_time_type callback of meshap_net_dev_ops and is described in the section set_slot_time_type.

13.6.44 get_erp_info

This function invokes the corresponding get_erp_info callback of meshap_net_dev_ops and is described in the section get_erp_info.

13.6.45 set_erp_info

This function invokes the corresponding set_erp_info callback of meshap_net_dev_ops and is described in the section set_erp_info.

13.6.46 set_beacon_vendor_info

This function invokes the corresponding set_beacon_vendor_info callback of meshap_net_dev_ops and is described in the section set_beacon_vendor_info.

13.6.47 get_ack_timeout

This function invokes the corresponding get_ack_timeout callback of meshap_net_dev_ops and is described in the section get_ack_timeout.

13.6.48 set_ack_timeout

This function invokes the corresponding set_ack_timeout callback of meshap_net_dev_ops and is described in the section set_ack_timeout.

13.6.49 get_hide_ssid

This function invokes the corresponding get_hide_ssid callback of meshap_net_dev_ops and is described in the section get_hide_ssid.

13.6.50 set_hide_ssid

This function invokes the corresponding set_hide_ssid callback of meshap_net_dev_ops and is described in the section set_hide_ssid.

13.6.51 enable_beaconing_uplink

This function invokes the corresponding enable_beaconing_uplink callback of meshap_net_dev_ops and is described in the section enable_beaconing_uplink.

13.6.52 disable_beaconing_uplink

This function invokes the corresponding disable_beaconing_uplink callback of meshap_net_dev_ops and is described in the section disable_beaconing_uplink.

13.6.53 get_supported_rates

This function invokes the corresponding get_supported_rate callback of meshap_net_dev_ops and is described in the section get_supported_rates.

13.6.54 get_bit_rate

This function invokes the corresponding get_bit_rate callback of meshap_net_dev_ops and is described in the section get_bit_rate.

13.6.55 set_bit_rate

This function invokes the corresponding set_bit_rate callback of meshap_net_dev_ops and is described in the section set_bit_rate.

13.6.56 get_rate_table

This function invokes the corresponding get_rate_table callback of meshap_net_dev_ops and is described in the section get_rate_table.

13.6.57 get_tx_power

This function invokes the corresponding get_tx_power callback of meshap_net_dev_ops and is described in the section get_tx_power.

13.6.58 set_tx_power

This function invokes the corresponding set_tx_power callback of meshap_net_dev_ops and is described in the section set_tx_power.

13.6.59 get_channel_count

This function invokes the corresponding get_channel_count callback of meshap_net_dev_ops and is described in the section get_channel_count.

13.6.60 get_channel

This function invokes the corresponding get_channel callback of meshap_net_dev_ops and is described in the section get_channel.

13.6.61 set_channel

This function invokes the corresponding set_channel callback of meshap_net_dev_ops and is described in the section set_channel.

13.6.62 get_phy_mode

This function invokes the corresponding get_phy_mode callback of meshap_net_dev_ops and is described in the section get_phy_mode.

13.6.63 set_phy_mode

This function invokes the corresponding set_phy_mode callback of meshap_net_dev_ops and is described in the section set_phy_mode.

13.6.64 get_preamble_type

This function invokes the corresponding get_preamble_type callback of meshap_net_dev_ops and is described in the section get_preamble_type.

13.6.65 set_preamble_type

This function invokes the corresponding set_preamble_type callback of meshap_net_dev_ops and is described in the section set_preamble_type.

13.6.66 get_reg_domain_info

This function invokes the corresponding get_reg_domain_info callback of meshap_net_dev_ops and is described in the section get_reg_domain_info.

13.6.67 set_reg_domain_info

This function invokes the corresponding set_reg_domain_info callback of meshap_net_dev_ops and is described in the section set_reg_domain_info.

13.6.68 get_supported_channels

This function invokes the corresponding get_supported_channels callback of meshap_net_dev_ops and is described in the section get_supported_channels.

13.6.69 set_dot11e_category_info

This function invokes the corresponding set_dot11e_category_info callback of meshap_net_dev_ops and is described in the section set_dot11e_category_info.

13.6.70 set_tx_antenna

This function invokes the corresponding set_tx_antenna callback meshap_net_dev_ops and is described in the section set_tx_antenna.

13.6.71 set_auth_hook

This is called during meshap initialization and it registers a hook function to process auth frames. Meshap will continue to use this function to set the hook.

13.6.72 set_assoc_hook

This is called during meshap initialization and it registers a hook function to process association frames. Meshap will continue to use this function to set the hook.

13.6.73 set_essid_info

This function invokes the corresponding set_essid callback of meshap_net_dev_ops and is described in the section set_essid_info.

13.6.74 set_radio_data

This function invokes the corresponding set_radio_data callback of meshap_net_dev_ops and is described in the section set_radio_data.

13.6.75 radio_diagnostic_command

This function invokes the corresponding radio_diagnostic_command callback of meshap_net_dev_ops and is described in the section radio_diagnostic_command.

13.6.76 set_action_hook

This function invokes the corresponding set_action_hook callback of meshap_net_dev_ops and is described in the section set_action_hook.

13.6.77 send_action

This function invokes the corresponding send_action callback of meshap_net_dev_ops and is described in the section send_action.

Version	Date	Change Log	Author	Approved By
0.1	02/28/14	Base Draft		

