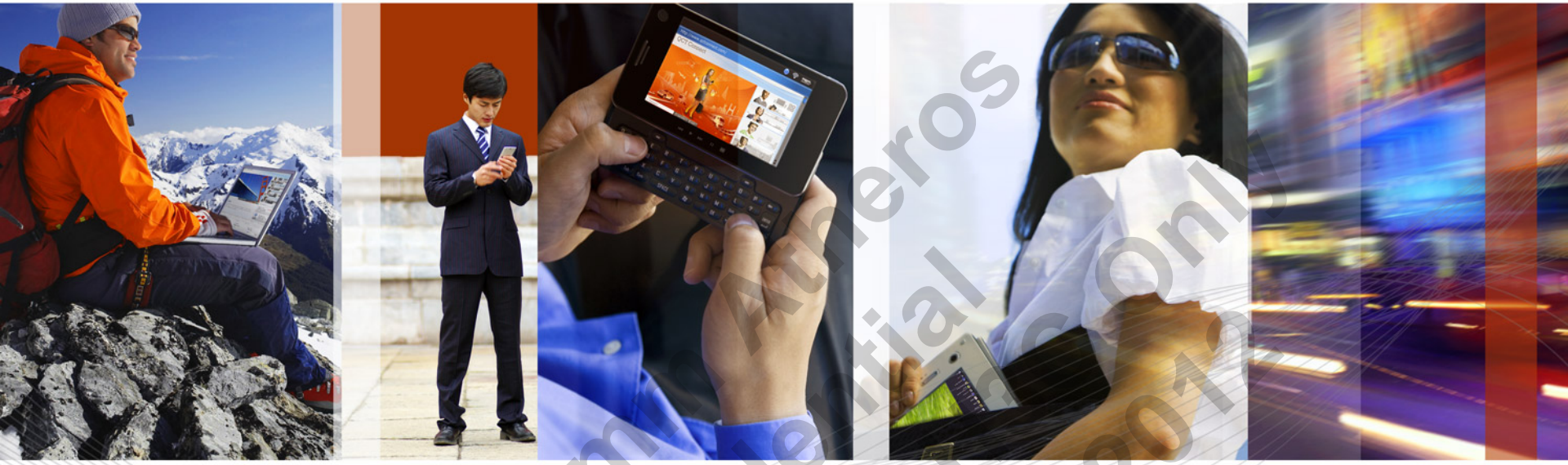


Qualcomm Atheros

Peregrine Software

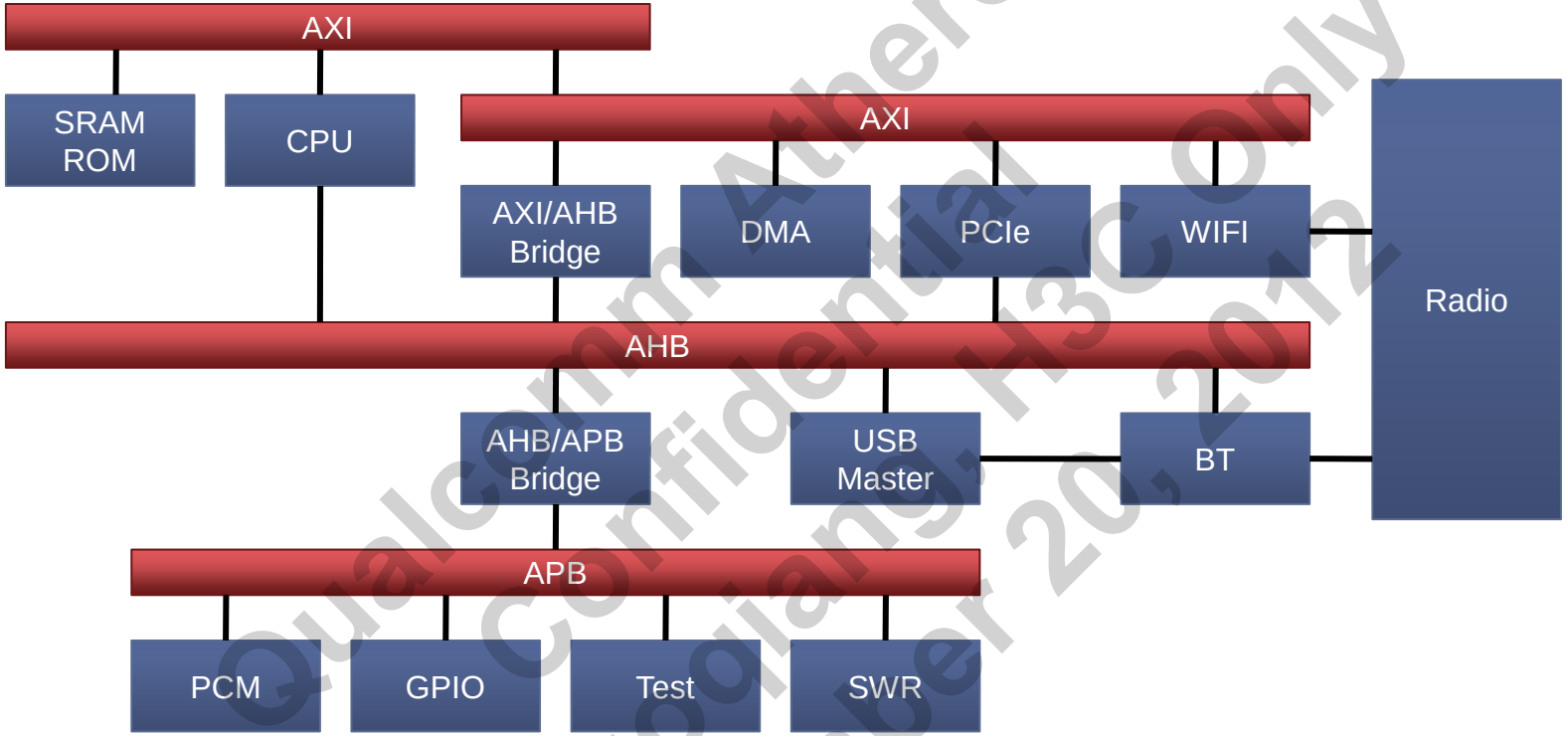
.

Qualcomm Atheros, Inc.
1700 Technology Drive
San Jose, CA 95110-1383
U.S.A.

Content

1. Peregrine HW Overview

1.1 SoC Blocks

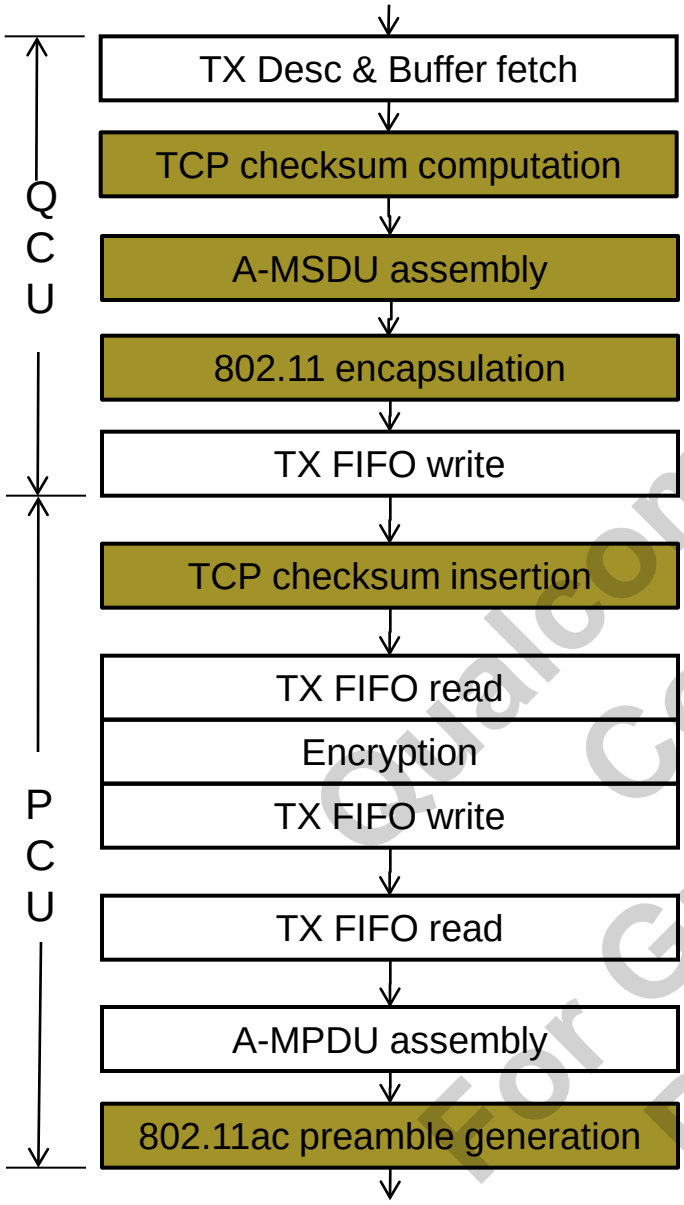


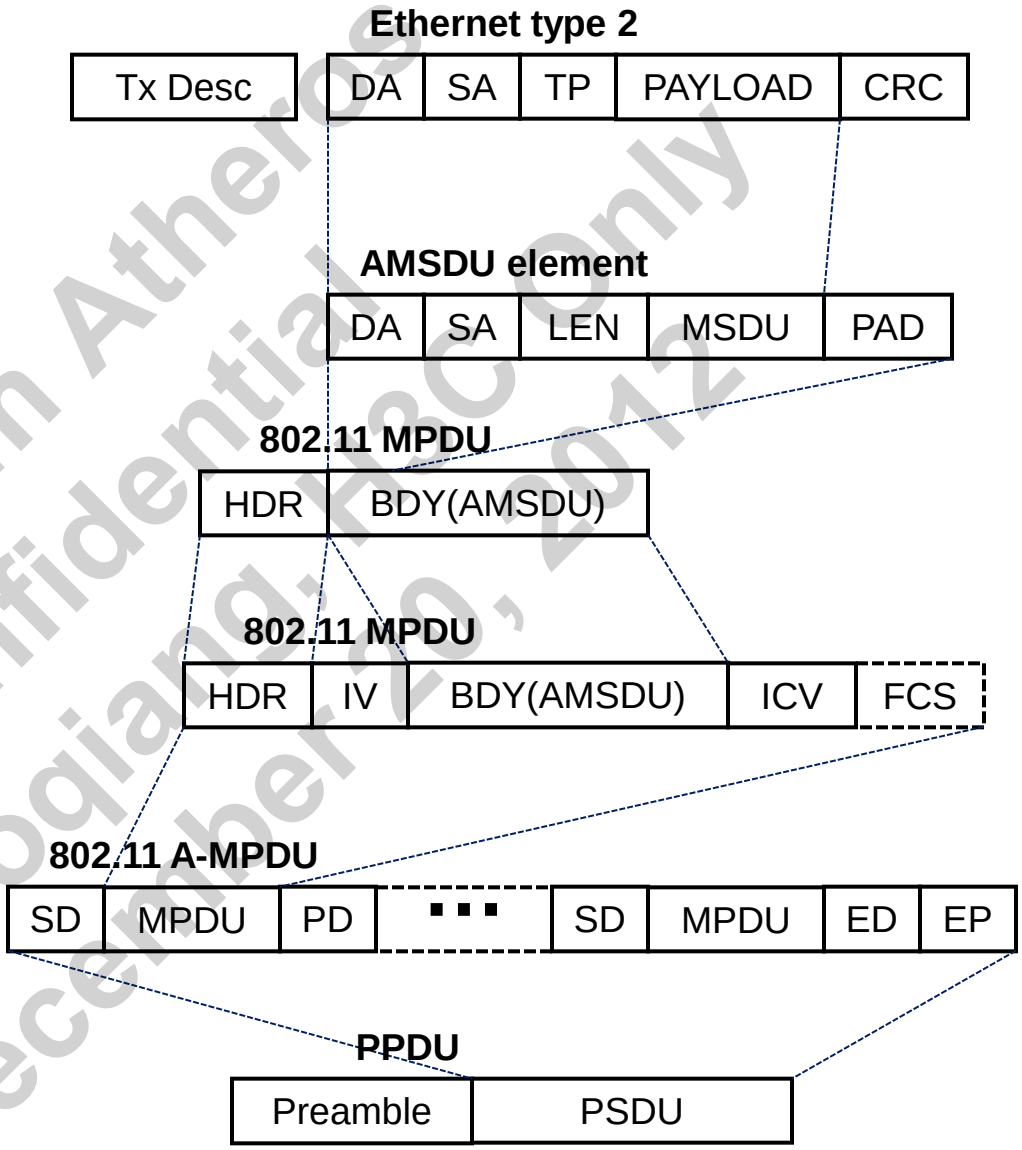
- Tensilica CPU @ 240 MHz
- Single cycle access to ROM and RAM
- One thread of execution

- Tight loop using a run-to-completion model
- > 2x code density compared to MIPS

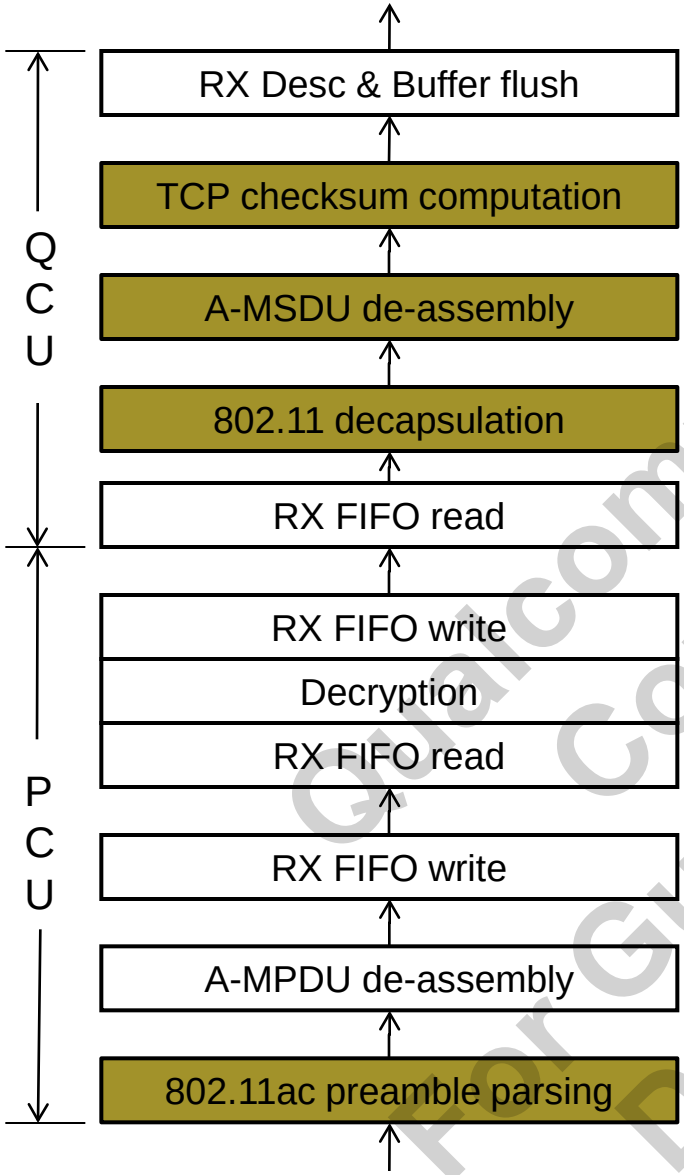
1.2 MAC Tx/Rx Data Flow

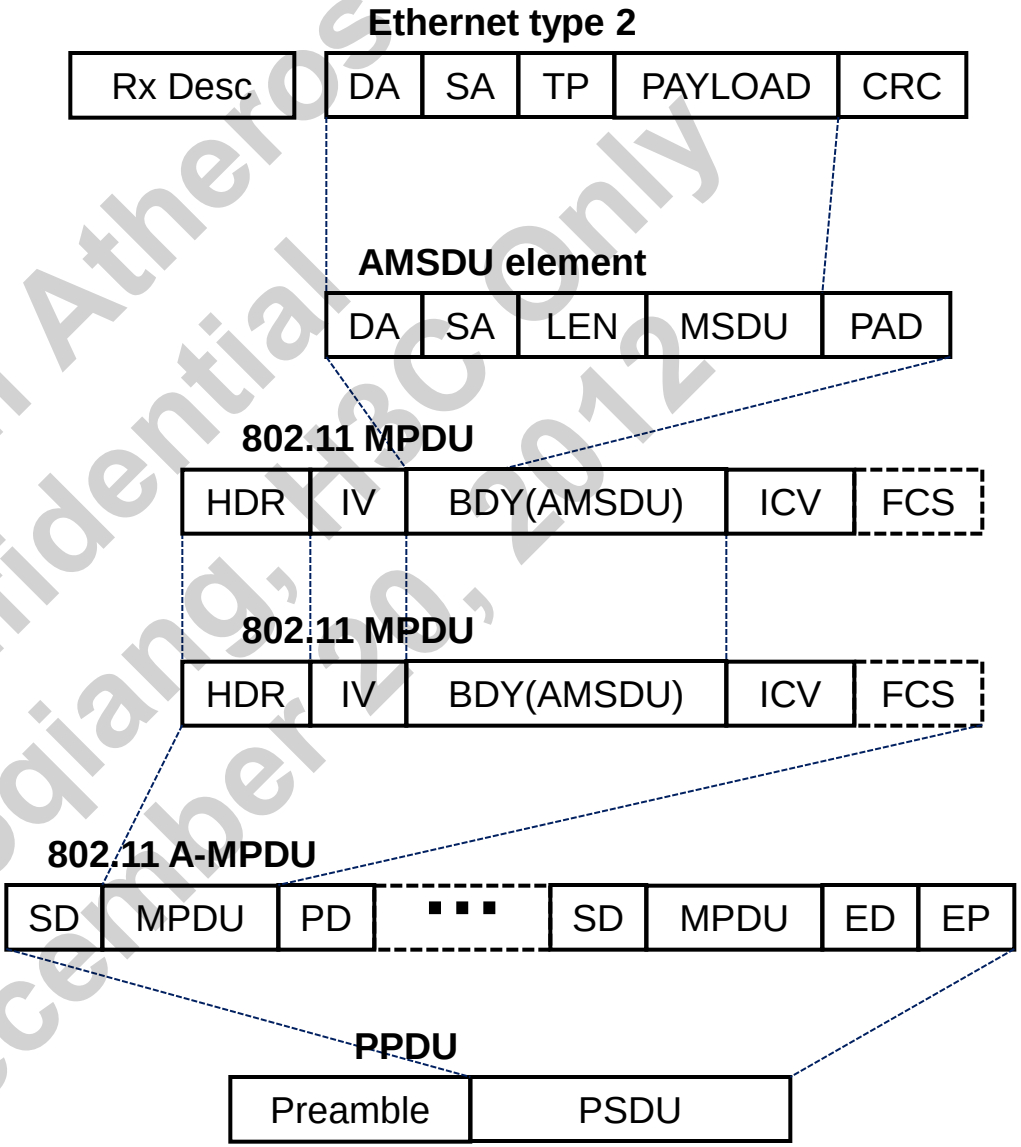
11ac MAC Tx Data Flow



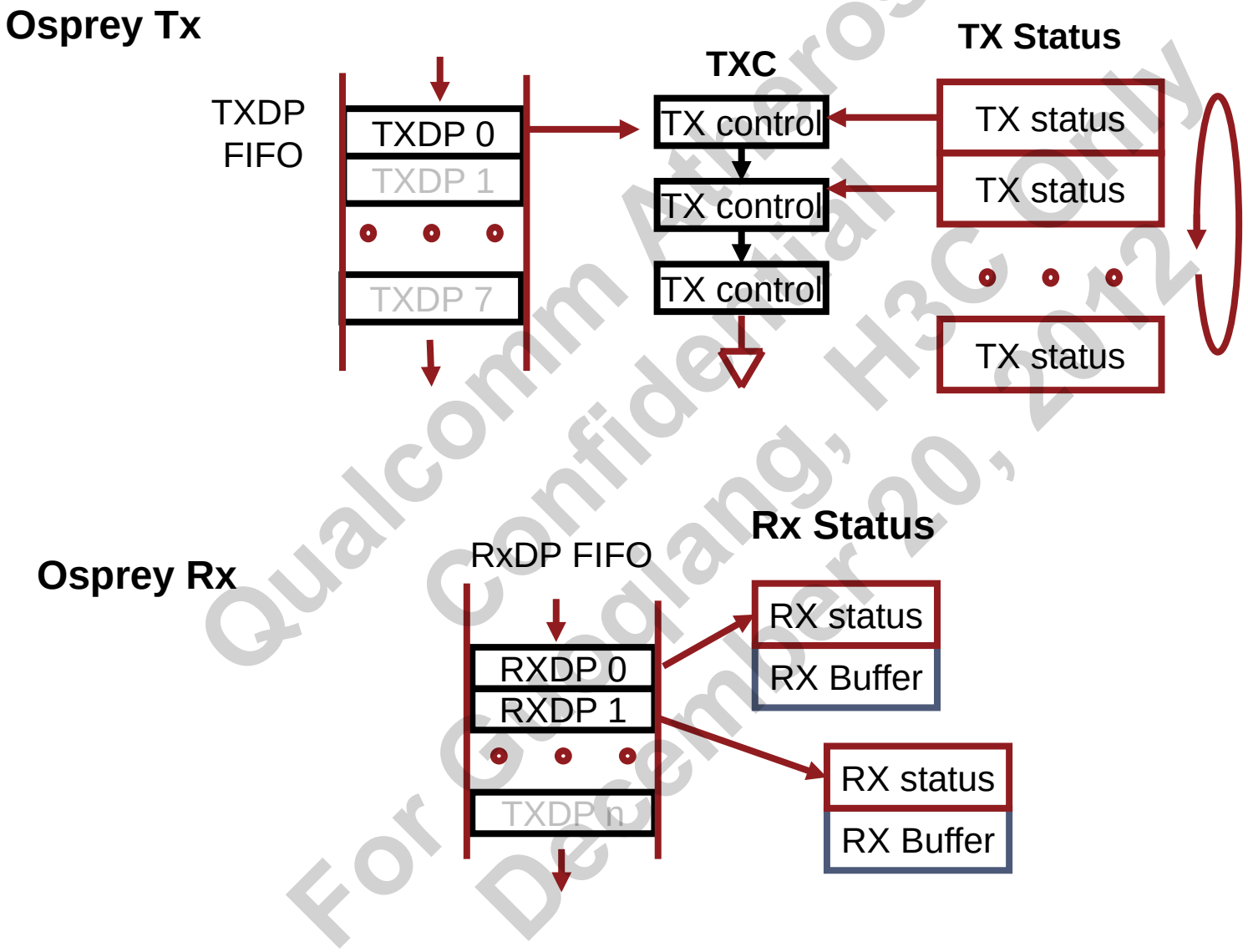


11ac MAC Rx Data Flow





1.3 Tx/Rx Descriptors

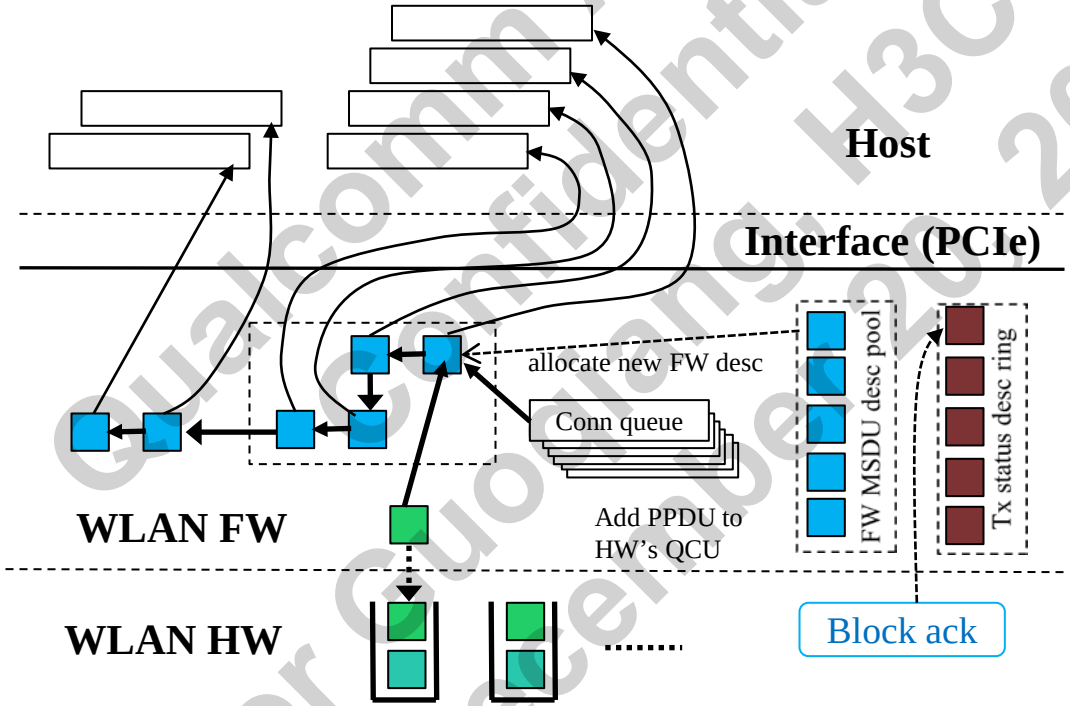


Peregrine Descriptors

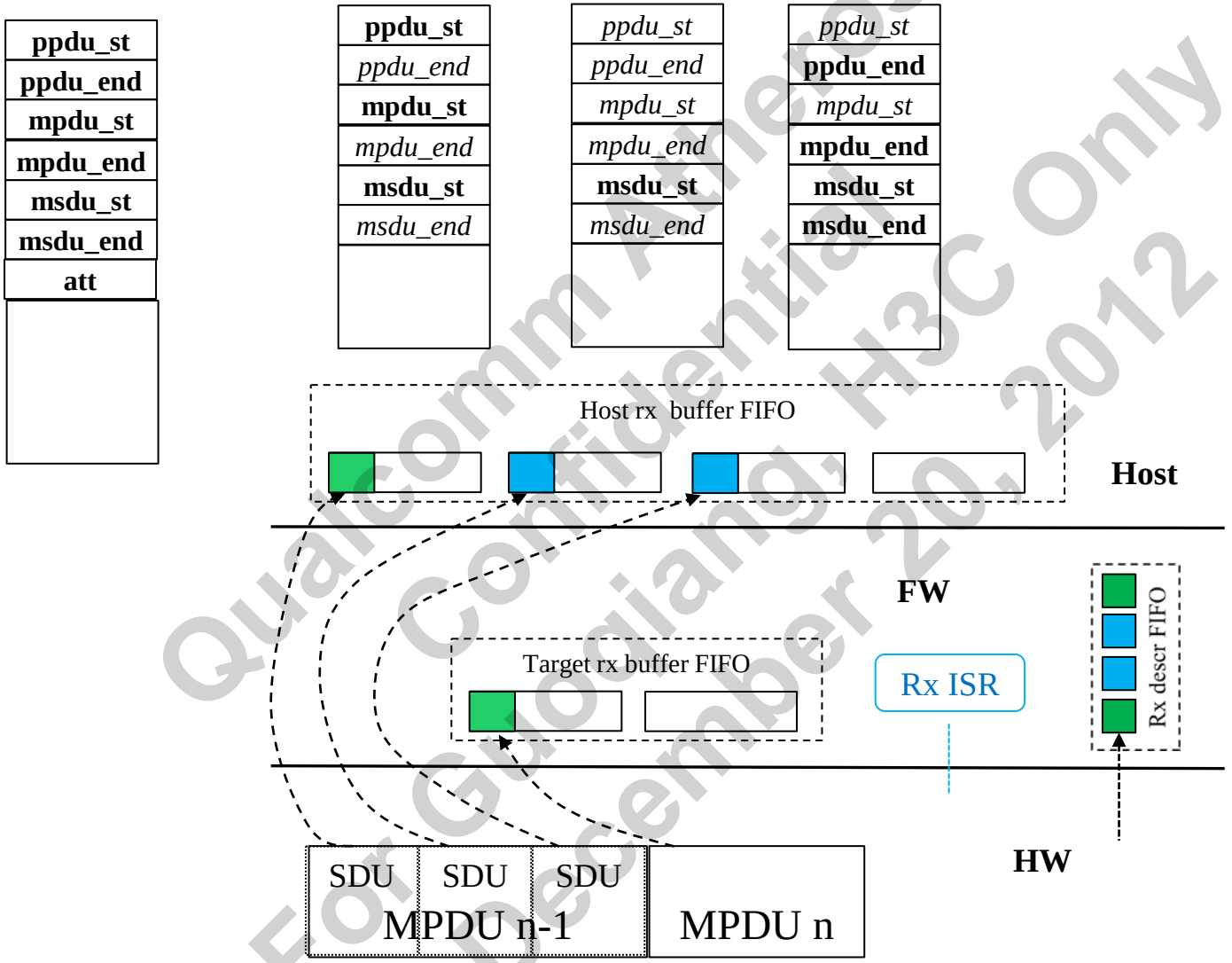
Description	Tag Value	FW visible	Created by	Modified by	Consumed by
tx_ppdu_start	0x00	Y	FW		DMA, PCU
tx_ppdu_end	0x01	Y	PCU TXSM		FW
tx_peer_table	0x02	Y	FW		OLE, PCU
tx_txbf_cv	0x03	Y	FW		BB
tx_msdu_start	0x04	Y	FW	TX OLE	OLE, PCU
tx_msdu_end	0x05	N	TX OLE		PCU
tx_packet	0x06	N	TX DMA	TX OLE	PCU
rx_ppdu_start	0x08	Y	PCU rxsm		FW
rx_ppdu_end	0x09	Y	PCU rxsm		FW
rx_mpdu_start	0x0A	Y	PCU rxsm		FW
rx_mpdu_end	0x0B	Y	PCU rxsm		FW
rx_msdu_start	0x0C	Y	RX OLE		FW
rx_msdu_end	0x0D	Y	RX OLE		FW
rx_attention	0x0E	Y	RX OLE		FW
rx_header	0x0F	Y	RX OLE		FW
rx_packet	0x10	Y	PCU rxsm		FW

Transmit Descriptors

TX PPDU start descriptor
TX MSDU start descriptor
TX MSDU end descriptor
TX PPDU end status descriptor

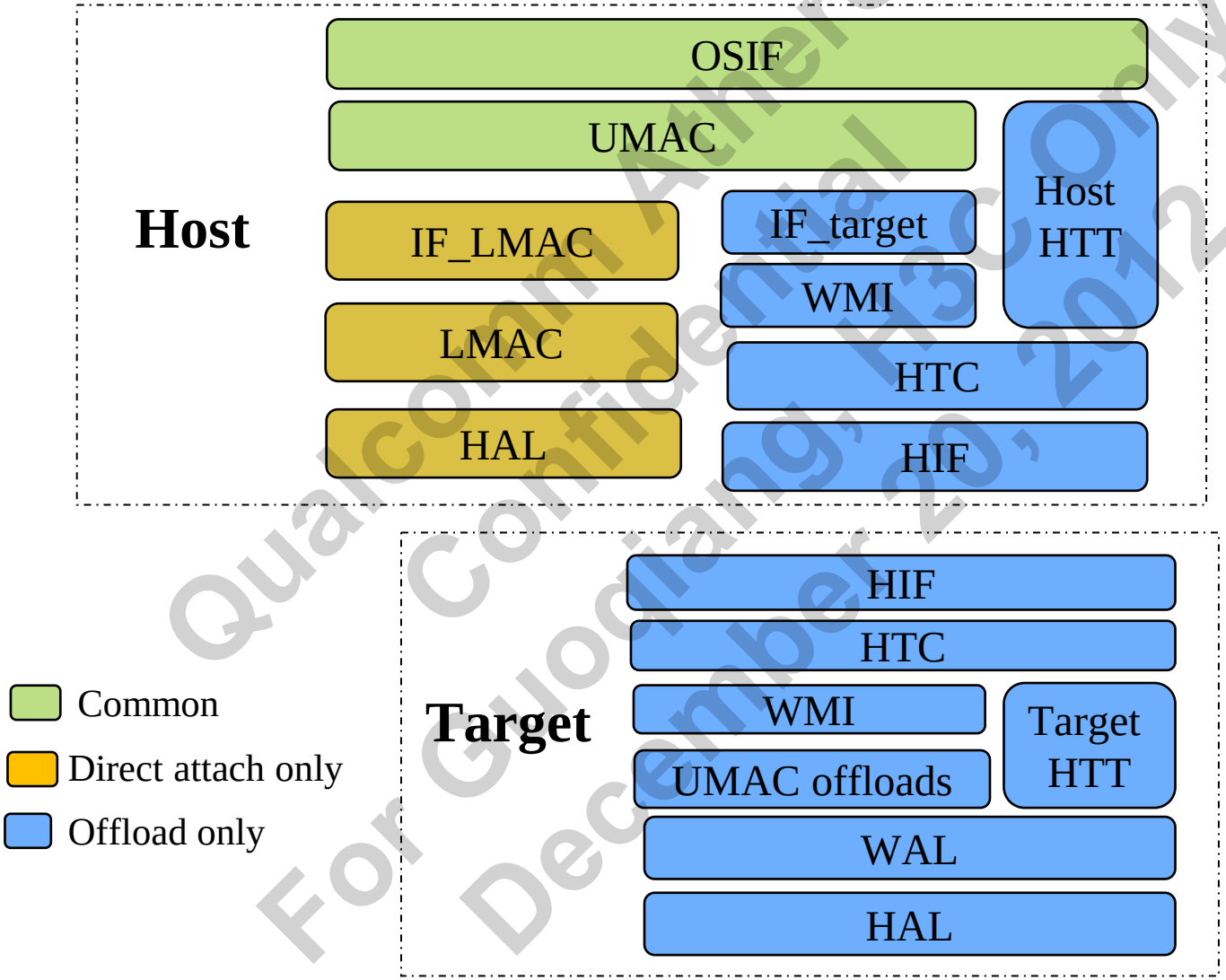


Receive Descriptors



2. Software Architecture

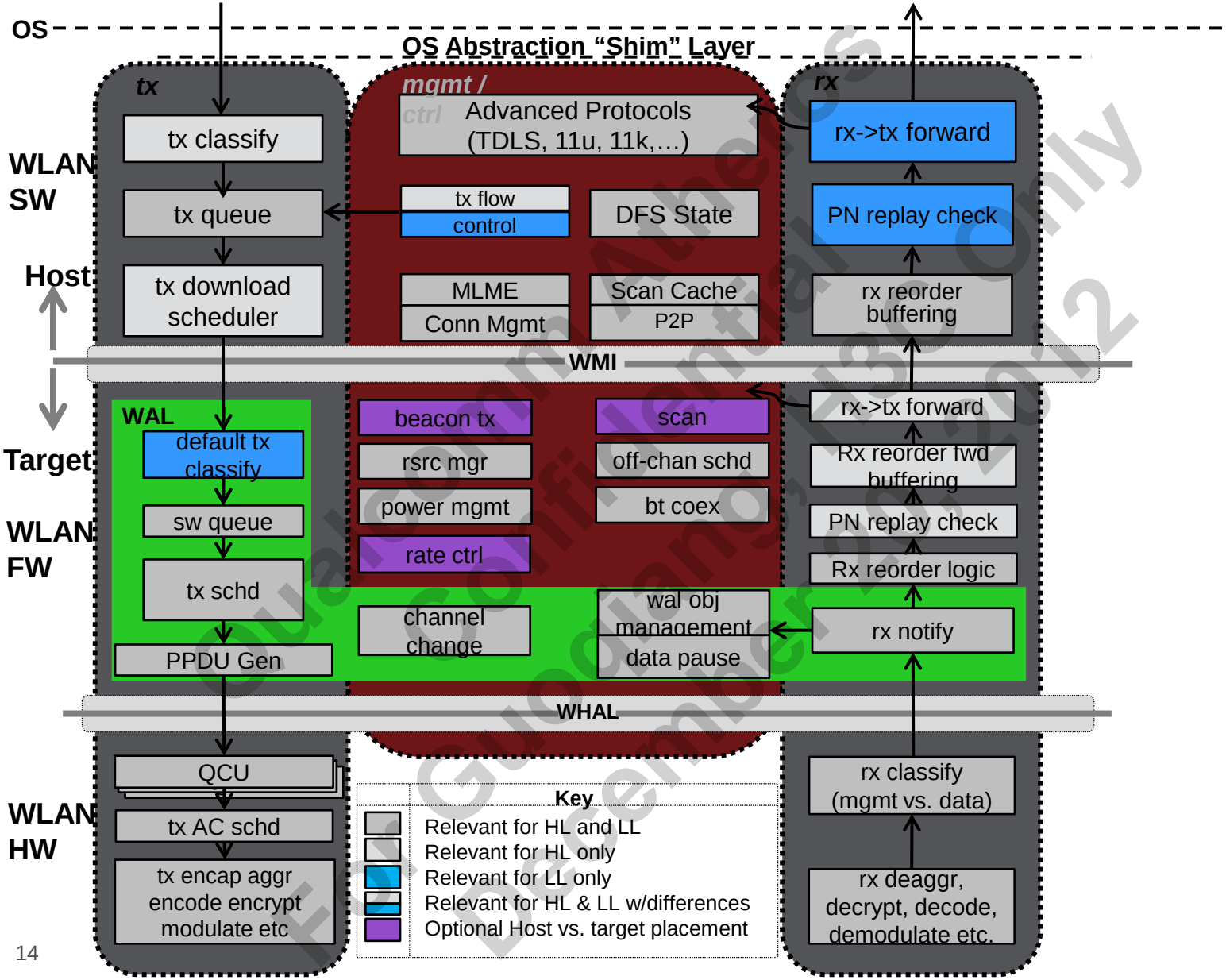
2.1 QCAmain Driver Architecture



- Unified architecture across all QCA products
- Fundamental considerations for firmware offload
  - Offload for power
  - Offload for performance
- Provides additional flexibility for custom offloads
  - Example: Application specific power offloads (ARP, mDNS …etc).
- Common UMAC (newma) protocol stack for existing 11n Direct attach and new 11ac fw based solutions.
- Support multiple Devices (both new 11ac off load and existing 11n direct attach solutions) .
  - DBDC AP with 11n direct attach (scorpion) and 11ac offload (peregrine).
- Support third party UMAC protocol stacks (bsd net80211, mac80211)
- Support Multiple Bus interfaces .
  - Low Latency interfaces (PCIE)
  - High Latency interfaces (USB,SDIO …)

- Provides rich configuration options, managed from host platform
  - Allows several resources on firmware/host to be configured to match the product /platform requirements .
  - the set of resources on firmware include vdevs, peers, HW descriptors ..etc.
  - example 1: retail AP product might choose 128 peers and 4 vdevs to support maximum of 128 client and 4 Virtual Aps.
  - example 2: windows client product might choose 8 peers and 3 vdevs to support a station and p2p functionality at the same time.
- Common OS Shim layer and configuration and tools.
- Reuse newma OS Shim layer interface and configuration and tools.
  - The existing Os shim layers on all supported OS are reused.
  - The existing configuration tools on the supported platforms are reused.
- Reuse most of the OS abstraction interfaces (wbuf,adf,asf).
- Common DFS detection module .

2.2 Offload Structure



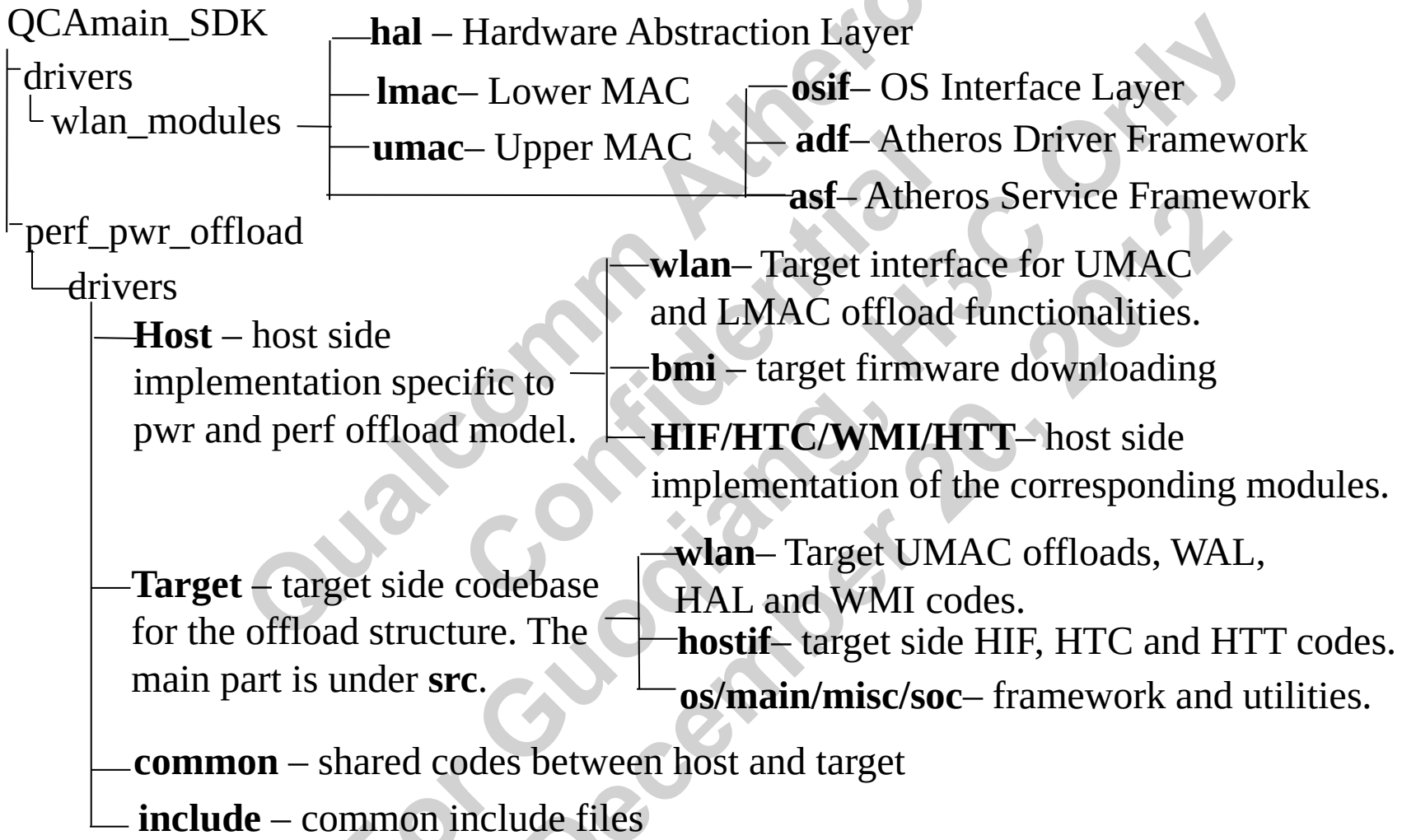
Firmware power offloads

- link management offloads
  - Examples : Beacons / probe response offloads, scan offload
- Application specific power offloads
  - Enables host CPU to go to sleep for applications that have fixed, but periodic, response requirements (Examples : ARP, mDNS etc)
- Chatter offloads
  - Enables low power operation by batching upload of several MSDUs
  - Discard unwanted Multicast frames.
- flexible / innovative power offloads
  - Examples : Green AP, STA <->STA intra-BSS or WDS routing without waking up the host
- Very likely reduces active power at high throughputs
  - Packet processing done more efficiently by firmware/MAC rather than high performance/cached host CPU

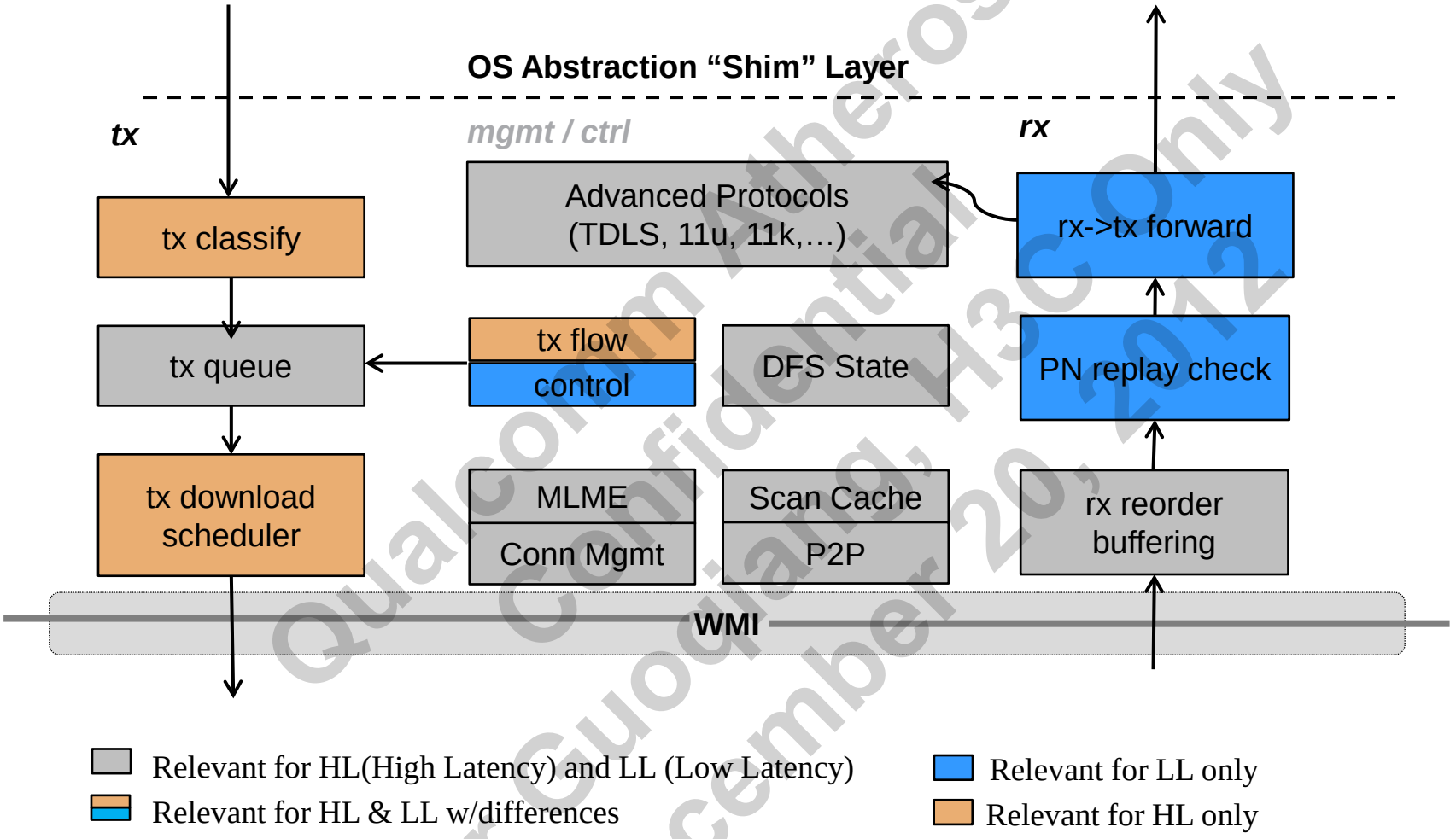
Firmware performance offloads

- Per MSDU offloads:
  - Flexible packet inspection and classification
- Tx Scheduling / per MPDU operations
  - Flexible tx scheduling options.
- Better support of DBDC configurations
  - Reduces host CPU requirements significantly for a dual-and dual concurrent scenario, as the aggregate throughput reaches in excess of multi-Gbps
- Flexible L2 routing decisions without host involvement
  - Provides reduced host CPU load and power for intra-BSS and WDS routing.
- False detect filtering and event coalescing for DFS/Spectral Scan
  - Ability to filter false detects in firmware
- Better performance for latency sensitive operations triggered by hardware events
  - Example : access point locationing support where a scan is triggered based on tx completion of CTS to self, off-channel switching triggered by tx completion or TSF interrupts, UAPSD trigger response (if done by software), response to STA sleep indications (if done by software)
  - AMPDU retries are completely handled in Firmware.

2.3 Source Tree



3. Host Driver



3.1 Boot Sequence

3.1.1 Steps to start wlan

1) Initialize AHB/PCI at “insmod”, init_ahb( ) os\linux\src\if_ath_ahb.c
2) Dev attachment at “insmod”, ath_dev_attach( ), wlan\lmac\ath_dev\ath_main.c
3) VAP creation at "wlanconfig ath create wlandev wifi0 wlanmode ap“, ath_vap_create() wlan\umac\if_lmac\if_ath.c
4) Attach interface at "wlanconfig ath create wlandev wifi0 wlanmode ap“, scn->sc_ops->add_interface() = ath_vap_attach(), lmac\ath_dev\ath_main.c
5) VAP open at "ifconfig ath0 192.168.2.20 up”, ath_open() at lmac/ath_dev/ath_main.c
6) Set channel at "ifconfig ath0 192.168.2.20 up”, ath_set_channel() at lmac/ath_dev/ath_main.c
7) VAP up at "ifconfig ath0 192.168.2.20 up”, ath_vap_up(), lmac/ath_dev/ath_main.c

3.1.2 PCI /PCIe Probe and Attachment

ret_fast_syscall( ) →sys_init_module( ), kernel\module.c, This is where the real work happens

└init_ath_pci(), wlan_modules\os\linux\src\if_ath_pci.c

└pci_register_driver(), drivers\pci\pci-driver.c, register a new pci driver: Adds the driver structure to the list of registered drivers → driver_register(), drivers\base\driver.c, register driver with bus →bus_add_driver(), drivers\base\bus.c, Add a driver to the bus →driver_attach(), drivers\base\dd.c, try to bind driver to devices →bus_for_each_dev( ), drivers\base\bus.c, Iterate over @bus's list of devices, and call @fn for each, passing it @data. If @start is not NULL, we use that device to begin iterating from. →__driver_attach(), dd.c, Lock device and try to bind to it. →driver_probe_device(), dd.c, attempt to bind device & driver →pci_device_probe(), driverpci\pci-driver.c

└ath_pci_probe(), wlan_modules\os\linux\src\if_ath_pci.c

- **For Peregrine, ol_ath_pci_probe(), perf_pwr_offload\drivers\host\hif\pci\linux\ath_pci.c**
- **Else, for existing chipsets.**
  - dev = alloc_netdev(sizeof(struct ath_ahb_softc), "wifi%d", ether_setup);
  - _ath_attach(), os\linux\src\ath_linux.c → **Create wifi0**

- **For the chips before Peregrine, attach the radio**

ath_pci_probe(), wlan_modules\os\linux\src\if_ath_pci.c

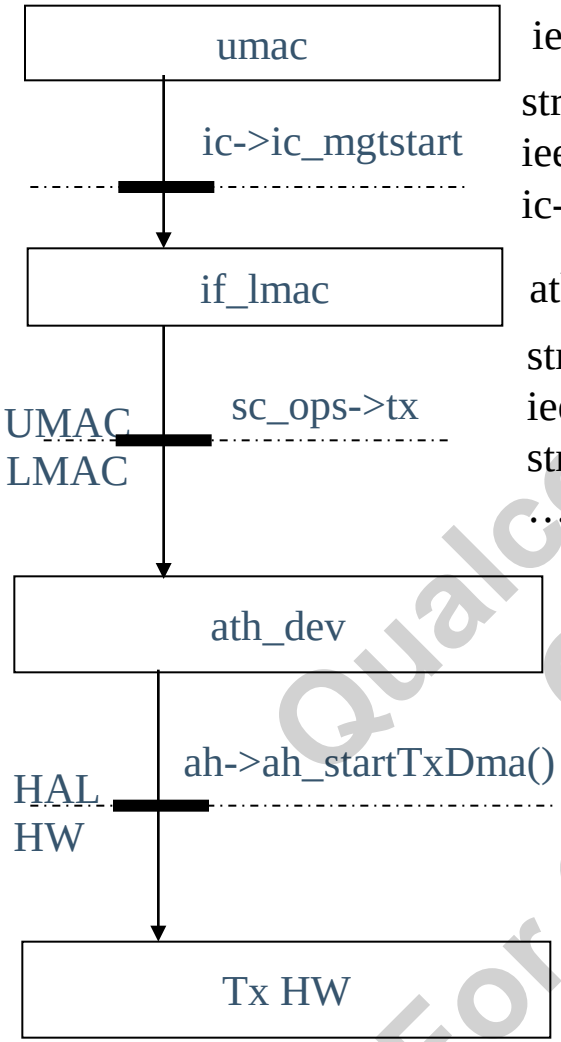
- **For Peregrine, ol_ath_pci_probe(), perf_pwr_offload\drivers\host\hif\pci\linux\ath_pci.c**
- **Else, for existing chipsets.**
  - dev = alloc_netdev(sizeof(struct ath_ahb_softc), "wifi%d", ether_setup); → **Create wifi0**
  - __ath_attach(), os\linux\src\ath_linux.c
    - hal_conf_parm.calInFlash = bus_context->bc_info.cal_in_flash;
    - ath_attach(devid, bus_context, scn, osdev, &ath_params, &hal_conf_parm, &wlan_reg_params);
      - ath_dev_attach(…, hal_conf_parm), wlan\lmac\ath_dev\ath_main.c, Create an Atheros Device object
        - _ath_hal_attach(…, hal_conf_parm,…), common\lmac\hal\linux\ah_osdep.c
          - ath_hal_attach(…, hal_conf_parm,…), hal\ah.c
            - ar9300Attach(…, hal_conf_parm,…), hal\ar9300\ar9300_attach.c
              - ar9300NewState(…, hal_conf_parm,…), hal\ar9300\ar9300_attach.c
                - ath_hal_factory_defaults(…, hal_conf_parm), hal\ar9300\ar9300_eeprom.c

For Peregrine

- Initialize BMI, HTC , HIF and WMI.
- Download firmware and complete BMI Done phase.
- Download configuration (To be completed)
  - Configuration of FW resources which include number of vdevs, peers, tids, descriptors …etc .
- Connect HTC Services(HTT Data and WMI).
- Wait for WMI Ready.
- Attach ol txrx/htt .
- Attach UMAC (ic) .
- Attach OL ath modules .

ol_ath_pci_probe(), perf_pwr_offload\drivers\host\hif\pci\linux\ath_pci.c
└_ol_ath_attach(), perf_pwr_offload\drivers\host\os\linux\ol_ath_linux.c
├dev = alloc_netdev(sizeof(struct ath_ahb_softc), "wifi%d", ether_setup); → **Create wifi0**
├scn = ath_netdev_priv(dev); wlan\umac\if_lmac\if_ath.c
└scn = dev->priv
└ol_ath_attach(), perf_pwr_offload\drivers\host\wlan\lmac_offload_if\ol_if_ath.c
├BMIInit(); 1. Initialize BMI
├BMIGetTargetInfo(); 2. Get target information
├ol_ath_configure_target(); 3. Configure targe
└ol_ath_download_firmware(), ol_if_ath.c
└ol_transfer_bin_file(), perf_pwr_offload\drivers\host\os\linux\ol_ath_linux.c
└ BMIWriteMemory(), perf_pwr_offload\drivers\host\bmi\bmi.c
└HIFExchangeBMIMsg(), perf_pwr_offload\drivers\host\hif\pci\hif_pci.c

3.2 Dynamic Attachment
3.2.1 Newma Attachment



ieee80211_send_mgmt(), umac\mlme\ieee80211_mgmt.c

struct ieee80211com {… int (*ic_mgtstart) (struct ieee80211com *, wbuf_t); …}, ieee80211_var.h
ic->ic_mgtstart = ath_tx_mgt_send, umac\if_lmac\if_ath.c

ath_tx_mgt_send(), if_ath.c

struct ath_ops { … int (*tx)(ath_dev_t, wbuf_t, ieee80211_tx_control_t *);…}, wlan\include\ath_dev.h
struct ath_ops ath_ar_ops = { … ath_tx_start, /* tx */ …} ath_dev\ath_main.c

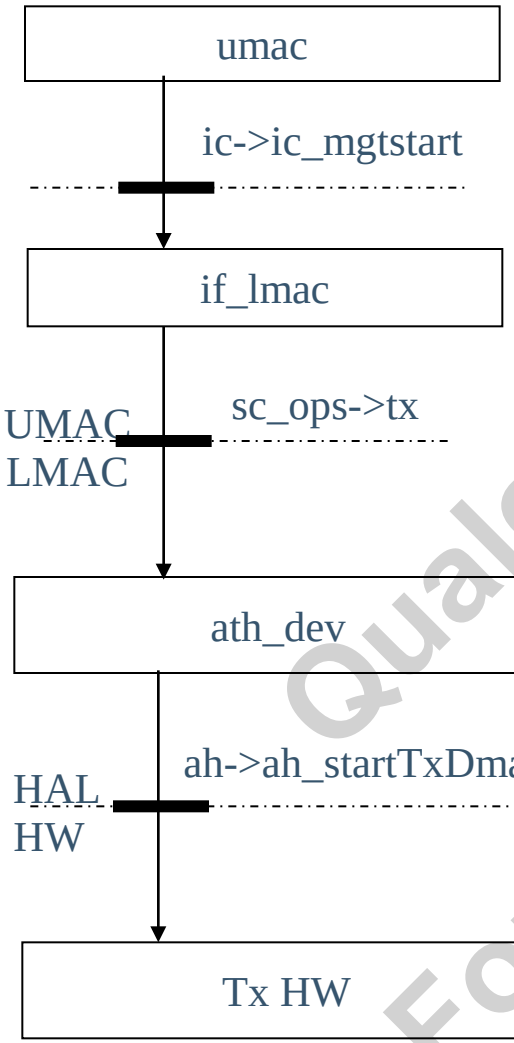
ath_tx_start(ath_dev_t, wbuf_t, ieee80211_tx_control_t *); ath_dev\ath_xmit.c

struct ath_hal {… HAL_BOOL __ahdecl(*ah_startTxDma) (struct ath_hal*, u_int);…}, hal\ah.h
static const struct ath_hal_private ar9300hal = {… ar9300StartTxDma, …} hal\ar9300\ar9300_attach.c

ar9300StartTxDma(), hal\ar9300\ar9300_xmit.c

3.2.2 QCA main 11n Attachment

It's the same even though the stack is under wlan_modules



ieee80211_send_mgmt(), wlan_modules\umac\mlme\ieee80211_mgmt.c

struct ieee80211com {… int (*ic_mgtstart) (struct ieee80211com *, wbuf_t); …}, ieee80211_var.h

ic->ic_mgtstart = ath_tx_mgt_send, umac\if_lmac\if_ath.c

ath_tx_mgt_send(), if_ath.c

struct ath_ops { … int (*tx)(ath_dev_t, wbuf_t, ieee80211_tx_control_t *);…}, wlan\include\ath_dev.h

struct ath_ops ath_ar_ops = { … ath_tx_start, /* tx */ …} ath_dev\ath_main.c

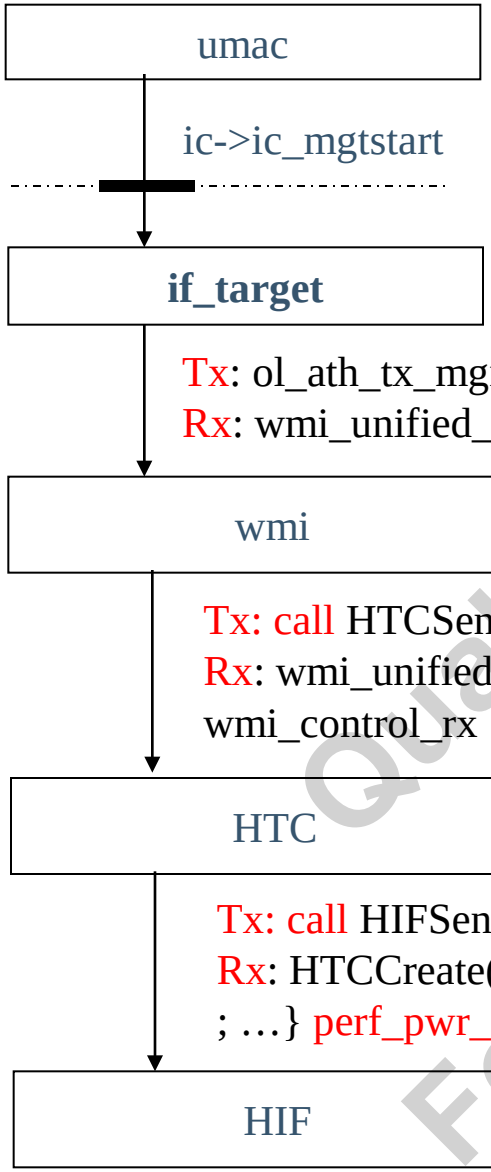
ath_tx_start(ath_dev_t, wbuf_t, ieee80211_tx_control_t *); ath_dev\ath_xmit.c

struct ath_hal {… HAL_BOOL __ahdecl(*ah_startTxDma)(struct ath_hal*, u_int);…}, hal\ah.h

static const struct ath_hal_private ar9300hal = {… ar9300StartTxDma, …} hal\ar9300\ar9300_attach.c

ar9300StartTxDma(), hal\ar9300\ar9300_xmit.c

3.2.3 QCA main 11ac Attachment



ieee80211_send_mgmt(), wlan_modules\umac\mlme\ieee80211_mgmt.c

struct ieee80211com {… int (*ic_mgtstart) (struct ieee80211com *, wbuf_t); …}, ieee80211_var.h

ic->ic_mgtstart = ol_ath_tx_mgmt_wmi_send, perf_pwr_offload\drivers\host\wlan\umac_offload_if\ ol_if_mgmt.c

ol_ath_tx_mgmt_wmi_send(), ol_if_mgmt.c

Tx: ol_ath_tx_mgmt_wmi_send() call wmi_unified_cmd_send()

Rx: wmi_unified_register_event_handler(…wmi_unified_mgmt_rx_event_handler,..);

wmi_unified_cmd_send() perf_pwr_offload\drivers\host\wmi\wmi_unified.c

Tx: call HTCSendPkt(), drivers\wlan_modules\htc\htc\htc_send.c

Rx: wmi_unified_connect_htc_service() { … connect.EpCallbacks.EpRecv = wmi_control_rx ; …} perf_pwr_offload\drivers\host\wmi\wmi_unified.c

HTCSendPkt(), drivers\wlan_modules\htc\htc\htc_send.c

Tx: call HIFSend(), drivers\wlan_modules\htc\htc\hif\pci\hif_pci.c

Rx: HTCCreate() { … htcCallbacks.rxCompletionHandler=HTCRxCompletionHandler; ; …} perf_pwr_offload\drivers\host\htc\htc.c

HIFSend(), drivers\wlan_modules\htc\htc\hif\pci\hif_pci.c

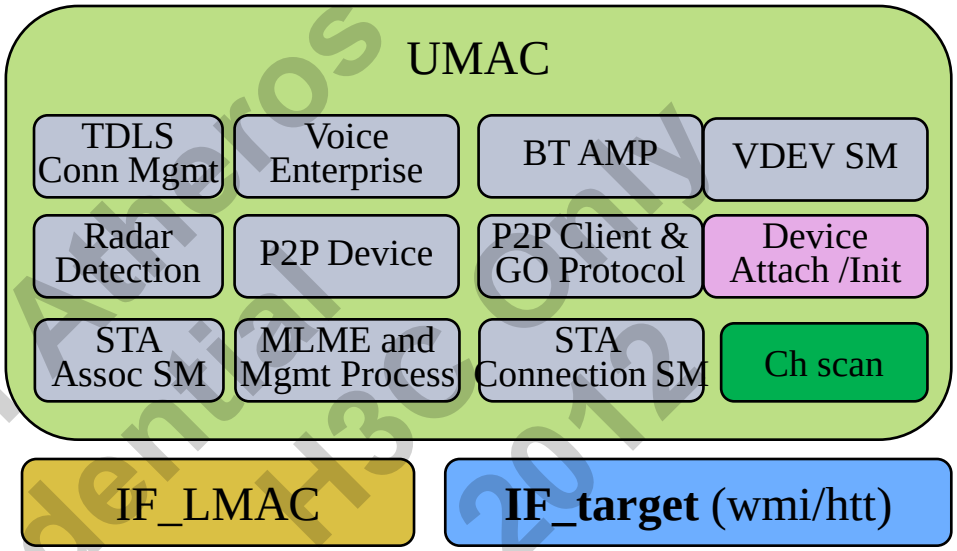
3.2.4 HTC/HIF Layers

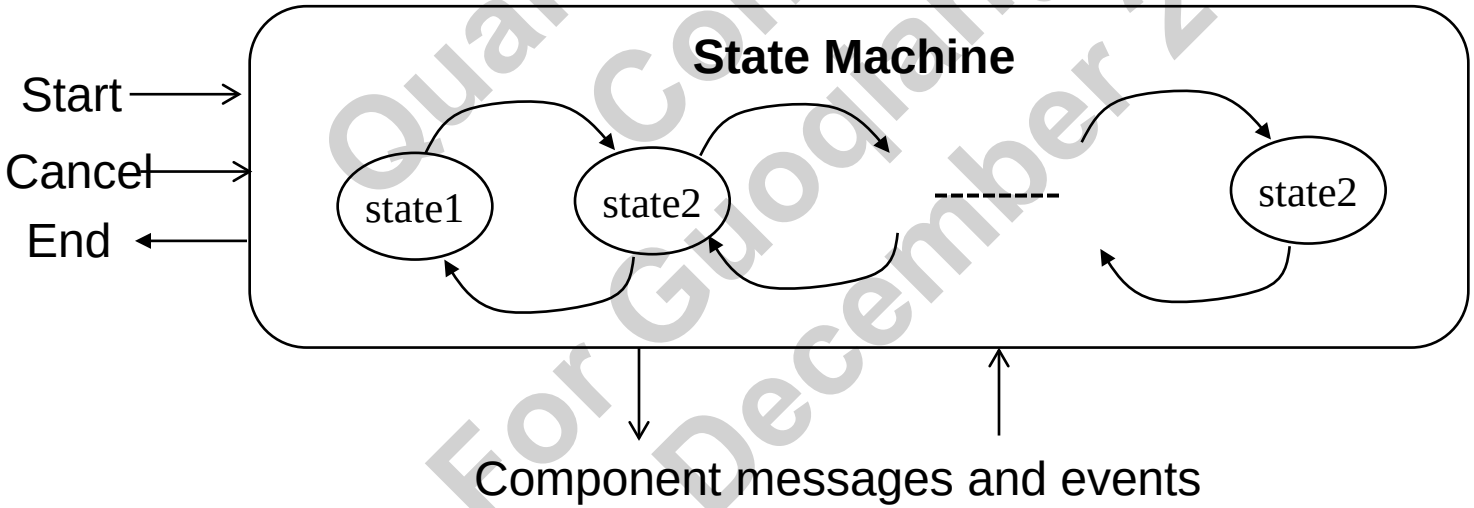
- HTC (Host Target Communication layer)
  - Mainly used for HL (high latency)
  - Provides virtual pipe/endpoint abstraction.

- HIF(Host Interface Layer)
  - Provides an abstraction for Host/Target bus (PCI,USB, SDIO …)
  - A separate implementation for each bus/platform combination.

3.3 Host UMAC Modules

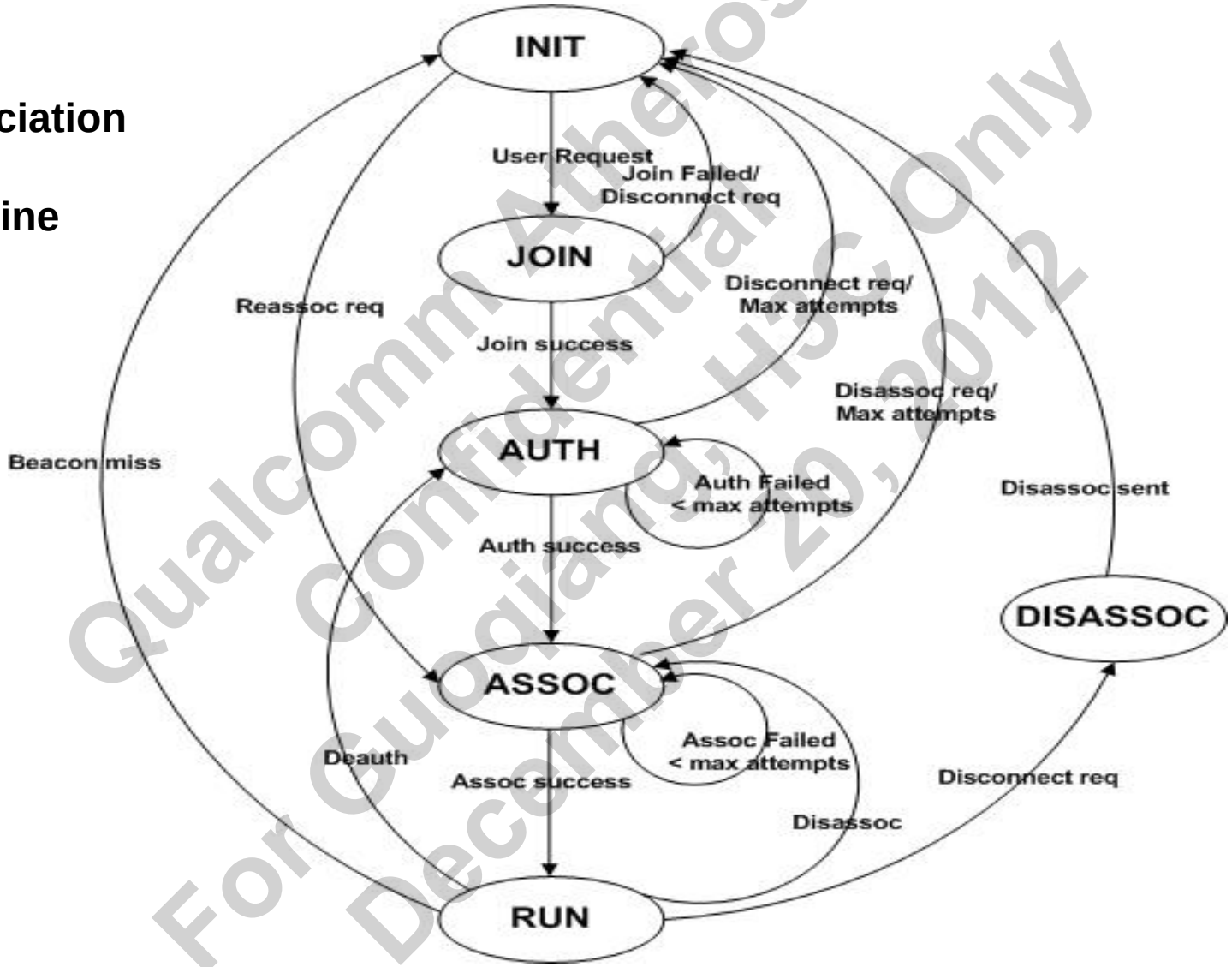
- A number of UMAC modules are running on host with only LMAC interactions with the target. Examples: MLME, PS, DFS..etc.
- By running on host means the main part, such as state machine, is on the host.





3.3.1 MLME Messages

Association State Machine



1) Tx Management Frame in 11n (QCAmain)

ieee80211_recv_probereq(ni, ,,), wlan_modules\umac\mlme\ieee80211_mgmt_bss.c,
the node pointer is from ieee80211_recv_probereq, which is vap->iv_bss

└ieee80211_send_proberesp(), ieee80211_mgmt_ap.c, Send probe response
 └ieee80211_send_mgmt(), wlan_modules\umac\mlme\ieee80211_mgmt.c
  └**ic->ic_mgtstart = ath_tx_mgt_send**, umac\if_lmac\if_ath.c
   └ath_tx_mgt_send(), wlan_modules\umac\if_lmac\if_ath.c

UMAC MLME processing

scn->sc_ops->tx= ath_tx_start, wlan_modules\lmac\ath_dev\ath_main.c
 └ath_tx_start(), ath_xmit.c → wbuf_map_sg() → __wbuf_map_sg( ), ath_wbuf.c
  └wbuf_start_dma(), ath_wbuf.c → ath_tx_start_dma(), ath_xmit.c
   └ath_tx_send_normal(), Send this frame as regular frame.
    └ath_rate_findrate(), ath_buf_set_rate(), ath_tx_txqaddbuf(), Queue it to h/w directly

Lmac Tx processing

ath_tx_txqaddbuf(), lmac\ath_dev\ath_xmit.c, Insert a chain of ath_buf (descriptors) on a txq and assume the descriptors are already chained together by caller
└ ath_hal_puttxbuf(), ath_internal.h, Write the tx descriptor address into the hw fifo.
 └ath_hal_puttxbuf=*(_ah)->ah_setTxDP()
  └ar9300SetTxD(), hal\ar9300\ar9300_xmit.c, Set the TxDP for the specified queue.
   └OS_REG_WRITE(ah, AR_QTXDP(q), txdp)

Queue data to AR9300 type hw

2) Tx Management Frames in Peregrine

•The UMAC on host constructs all management (except beacons and probe response frames) frames with 802.11 header and sends the frame to the WMI (HTT) module on the host.
•The WMI (HTT) then sends the frame down to target at appropriate time.
•The WAL part of FW will queue the frame to the HW for transmission. The FW (WAL) will setup the sequence number part of the descriptor for HW to update the sequence number of the frame Target maintains per node/per tid sequence number for management frames.
•All the Block ack agreement related frames (transmit and receive) like addba req, addba resp and delba a are completely handled in the firmware for performance offload reasons.
•HTC Layer defines a separate virtual pipe for sending management frames. HTT/Host Data Path uses the HTC virtual pipe to send down the management frame.
•The HTC virtual pipe will be mapped to a separate physical pipe (like a copy engine for peregrine) if available else it will use the same physical pipe used for TX data .

Send Management Frames Via WMI

osif_vap_stop(), wlan_modules\os\linux\src\osif_umac.c
└osif_vap_down(), osif_umac.c
  └wlan_mlme_stop_bss(), wlan_modules\umac\mlme\ieee80211_mlme.c
    └wlan_iterate_station_list(), wlan_modules\umac\base\ieee80211_node.c
      └ieee80211_iterate_node_list(), ieee80211_node.c

sta_disassoc(); ieee80211_mlme.c
└ieee80211_send_disassoc(); wlan_modules\umac\mlme\ieee80211_mgmt.c
  └ieee80211_send_disassoc_with_callback(), ieee80211_mgmt.c
    └ieee80211_send_mgmt(), wlan_modules\umac\mlme\ ieee80211_mgmt.c
      └ ic->ic_mgtstart(ic, wbuf)

ol_ath_tx_mgmt_wmi_send(), perf_pwr_offload\drivers\host\wlan\umac_offload_if**ol_if_mgmt.c**, Send Mgmt frames via WMI
  └wmi_unified_mgmt_send(), umac_offload_if\ ol_if_mgmt.c ,
    └wmi_unified_cmd_send(); Send the management frame buffer to the target

perf_pwr_offload\drivers\host\wlan\umac_offload_if\ol_if_mgmt.c
void ol_ath_mgmt_attach(struct ieee80211com *ic) { …
ic->ic_mgtstart = **ol_ath_tx_mgmt_wmi_send**;
ic->ic_newassoc = ol_ath_net80211_newassoc; …..}

3) Rx Management Frame in 11n (QCAmain)

Create the node for the station in Auth Rx

ath_handle_intr(), lmac\ath_dev\ath_main.c →ath_handle_rx_intr(), os\linux\src\ath_osdep.c
└sc_ops->rx_proc()=ath_rx_tasklet(), ath_recv.c →ath_rx_indicate(), os\linux\src\ath_osdep.c
└ath_net80211_rx(), umac\if_umac\if_ath.c →ieee80211_input_all().
├Spam the packet to all VAPs. For each VAP, make a copy of the skb, and do the following.
├ieee80211_ref_node(); net80211\ieee80211_node.h, add a reference to the ni.
└ni = vap->iv_bss; ieee80211_input().
└ieee80211_recv_mgmt, umac\mlme\ieee80211_mgmt.c
└ieee80211_recv_auth(), umac\mlme\ieee80211_mgmt.c
└ieee80211_mlme_recv_auth, umac\mlme\ieee80211_mlme.c
└ mlme_recv_auth_ap(), umac\mlme\ieee80211_mlme_ap.c
└ ieee80211_dup_bss(), ieee80211_node.c
└ieee80211_alloc_node(), Create an entry in the specified node table. The node is setup with the mac address, an initial reference count, and some basic parameters obtained from global state. This interface is not intended for general use, it is used by the routines below to create entries with a specific purpose.

4) Reception of Management Frames (11ac)

•All the management frames are received by the FW.

•FW will get a chance to process them before sending them up to the host via WMI command.

•FW completely handles all the addba frame exchange during block ack windows negotiation and all the addba action frame exchange terminates in the FW.

Create the node for the station in Auth Rx

HIF_PCI_CE_recv_data(), perf_pwr_offload\drivers\host\hif\pci\hif_pci.c, Called by lower (CE) layer when data is received from the Target.

└hif_completion_thread(), perf_pwr_offload\drivers\host\hif\pci\hif_pci.c, This thread provides a context in which send/recv completions are handled.

└msg_callbacks->rxCompletionHandler()

└HTCRxCompletionHandler(), drivers\wlan_modules\htc\htc\htc_recv.c

└RecvPacketCompletion(), drivers\wlan_modules\htc\htc\htc_recv.c

└DoRecvCompletion(), drivers\wlan_modules\htc\htc\htc_recv.c

└pEndpoint->EpCallBacks.EpRecv() = wmi_control_rx(), host_drv\perf_pwr_offload\drivers\host\wmi\wmi_unified.c

pEndpoint->EpCallBacks.EpRecv() = wmi_control_rx(),
host_drv\perf_pwr_offload\drivers\host\wmi\wmi_unified.c
└wmi_unified_event_rx(), wmi\wmi_unified.c
 └wmi_handle->event_handler(), Call the WMI registered event handler.
  └wmi_unified_mgmt_rx_event_handler(), perf_pwr_offload\drivers
   \host\wlan\umac_offload_if\ol_if_mgmt.c
    └**ieee80211_input_all**(), drivers**wlan_modules**\umac\txrx\ieee80211_input.c

ieee80211_input_all(), at this time, the node isn't set up. Spam the frame.
└ni = vap->iv_bss; ieee80211_input().
 └ieee80211_recv_mgmt, umac\mlme\ieee80211_mgmt.c
  └ieee80211_recv_auth(), umac\mlme\ieee80211_mgmt.c
   └ieee80211_mlme_recv_auth, umac\mlme\ieee80211_mlme.c
    └ mlme_recv_auth_ap(), umac\mlme\ieee80211_mlme_ap.c
     └ ieee80211_dup_bss(), ieee80211_node.c
      └ieee80211_alloc_node(), Create an entry in the specified node table. The node is setup with the mac address, an initial reference count, and some basic parameters obtained from global state. This interface is not intended for general use, it is used by the routines below to create entries with a specific purpose.

3.4 Target UMAC Modules

- Some UMAC modules are made optional/dynamic and made them to run either host (for direct attach) or Target (for offload).

  Examples: Scan, Resource Manager, Station Power save ..etc.

- These modules changed to be attached dynamically.

  For each module

  - API interface exposed by the module remains the same.
  - API is changed from direct/static linking to indirect, Function pointer based dynamic linking.
  - Existing newma solution attaches the existing umac module .
  - The 11ac solution attaches a different module part of OL WMI layer implementing the same interface. This modules translate the API to WMI command and sends the command down to target.

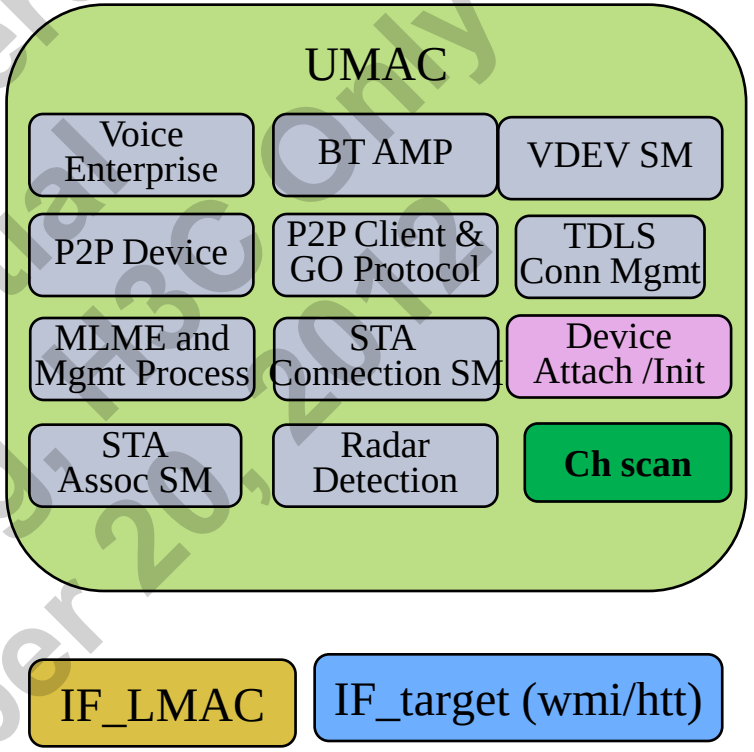
- The changes are minimal as umac implementation itself is modular and granularity of the changes is at module level.

3.4.1 Channel Scan

3.4.1.1 Dynamic attach ‘Scan’ module

- API interface drivers\wlan_modules \umac\include\ieee80211_scan.h

```
struct ieee80211_scanner_common { …
int (* scan_start)    (…. )
int (* scan_cancel) ( … )
        ……..   };
static INLINE int ieee80211_scan_start(….) {
return ( (struct ieee80211_scanner_common
*)ss)->scan_start(….) }
static INLINE int ieee80211_scan_cancel(….) {
return ( (struct ieee80211_scanner_common
*)ss)->scan_cancel(…) }
        ……
```



- 2 modules implement the same scan API interface. One is the existing scan module(ieee80211_scan.c) under umac , the othe WMI translation module implemented in if wmi layer.
  - umac scan attach

```
int   _ieee80211_scan_attach(ieee80211_scanner_t      *ss, …)   {
        (*ss)->ss_common.scan_start = _ieee80211_scan_start;
       (*ss)->ss_common.scan_cancel = _ieee80211_scan_cancel;
}
```

  - if wmi scan attach

```
int   ol_scan_attach(ieee80211_scanner_t      *ss, …)   {
        (*ss)->ss_common.scan_start =   ol_scan_start;
       (*ss)->ss_common.scan_cancel =  ol_scan_cancel;
}
```

•Start Scan

osif_vap_init() → wlan_autoselect_register_event_handler() → osif_vap_init()
→ wlan_autoselect_register_event_handler()
└wlan_set_channel(), wlan_modules\umac\base\ieee80211_channel.c
└wlan_autoselect_find_infra_bss_channel() wlan_modules\umac\acs\ieee80211_acs.c
└ieee80211_autoselect_infra_bss_channel(), ieee80211_acs.c
└wlan_scan_start(), drivers\wlan_modules\umac\scan\ieee80211_scan_api.c
└ieee80211_scan_start(), host_drv\drivers\wlan_modules\umac\include\ieee80211_scan.h
└ ss->scan_start(ss,vaphandle,params, requestor,priority,scan_id);

Before Peregrine

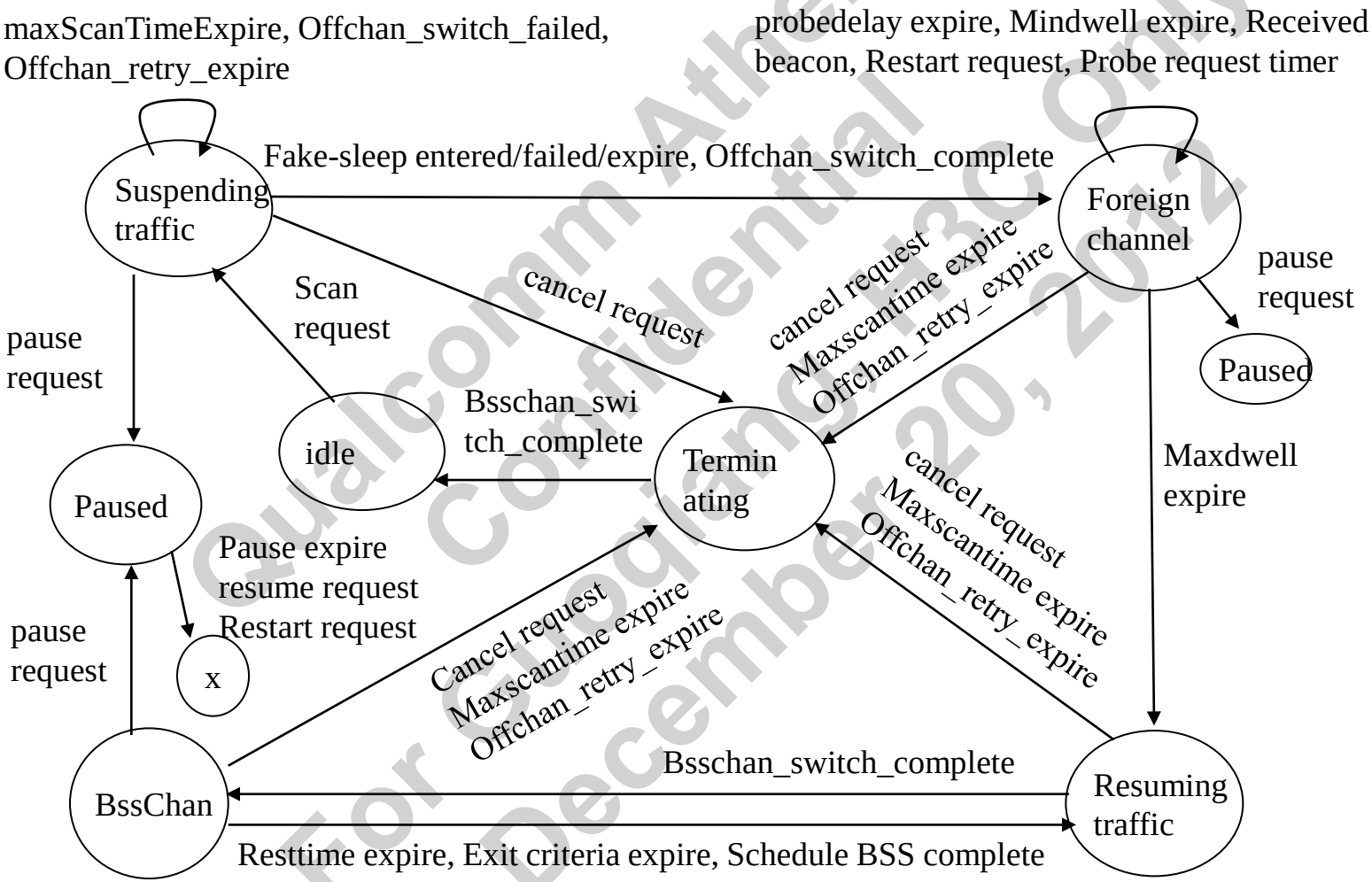
_ieee80211_scan_start(), drivers\wlan_modules\umac\scan\ieee80211_scan.c
└ieee80211_scan_run(), drivers\wlan_modules\umac\scan\ieee80211_scan.c
└ieee80211_sm_dispatch(ss->ss_hsm_handle, request_data->preemption_data.preempted ?
SCANNER_EVENT_RESTART_REQUEST : SCANNER_EVENT_SCAN_REQUEST, 0,
NULL); dispatch event into the HSM. drivers\wlan_modules\umac\sme\ieee80211_sm.c
└OS_MESGQ_SEND()

After Peregrine

ol_scan_start(), perf_pwr_offload\drivers\host\wlan\umac_offload_if\ ol_if_scan.c
└wmi_unified_scan_start_send(), drivers\wlan_modules\umac\scan\ieee80211_scan.c
└wmi_unified_cmd_send(wmi_handle, buf, len, WMI_UNIFIED_START_SCAN_CMDID);

3.4.1.2 Scan State Machine

- For the chips before Peregrine (11n chipsets), the scan state machine is on the host.



Foreign Channel State: Check dwell time (MINDWELL_EXPIRE)

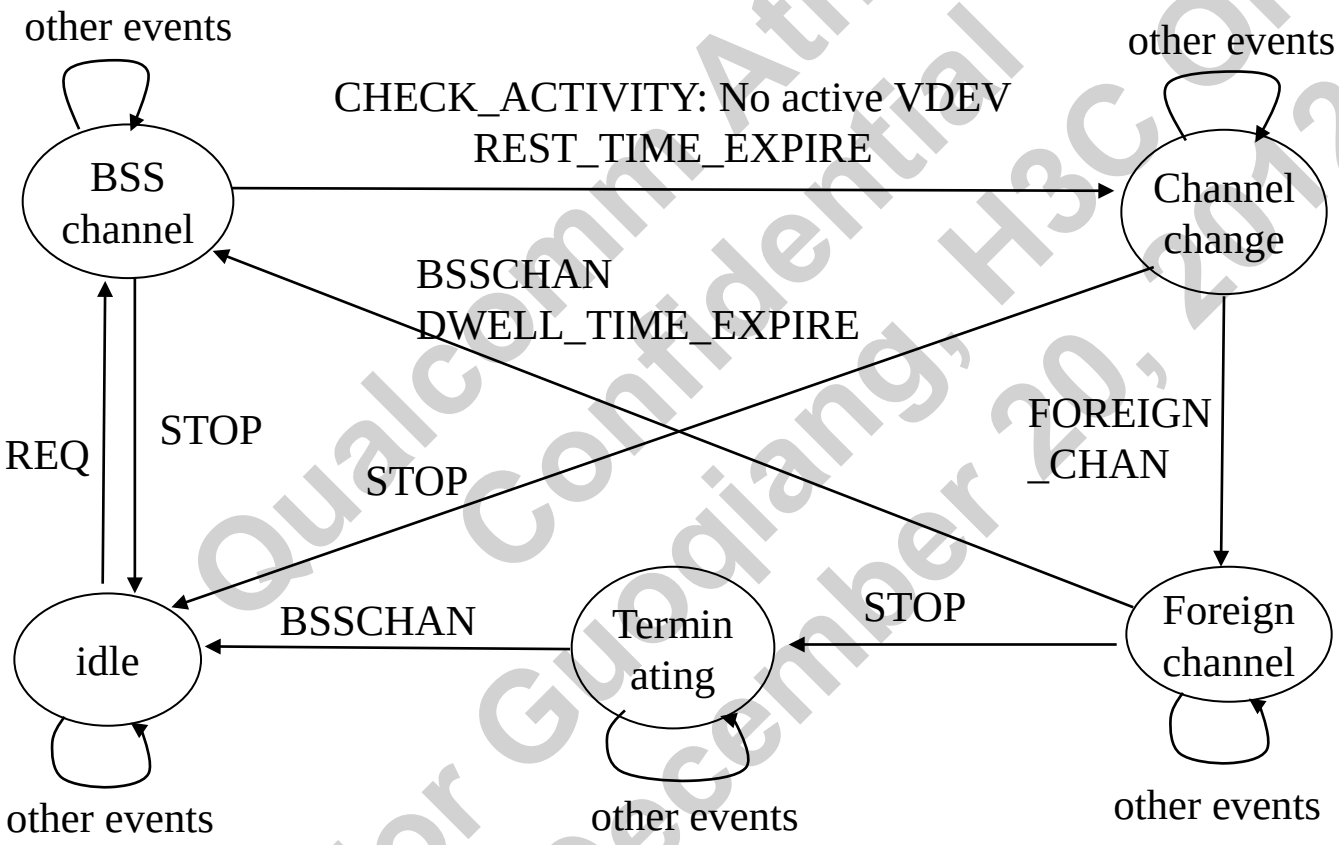
start_kernel() →cpu_idle() →r4k_wait_irqoff() → ret_from_irq()
└plat_irq_dispatch() → do_softirq() →__do_softirq() →run_timer_softirq()
└os_mesgq_handler(), drivers**wlan_modules**\os\linux\include\osdep_adf.h, API to deliver messages(events) asynchronously.
└queue->**handler**(queue->ctx,mesg->mesg_type,mesg->mesg_len, msg);
└ieee80211_sm_dispatch_sync_internal(), drivers\wlan_modules\umac\sme\ieee80211_sm.c
└event_handled = (*hsm->state_info[state].**ieee80211_hsm_event**) (hsm->ctx, event, event_data_len, event_data);

└scanner_state_foreign_channel_event(), drivers\wlan_modules\umac\scan\ieee80211_scan.c
└scanner_leave_foreign_channel(), ieee80211_scan.c
└ieee80211_sm_transition_to(), ieee80211_sm.c
└scanner_state_foreign_channel_entry(), ieee80211_scan.c
└ieee80211_set_channel(), wlan_modules\umac\base\ieee80211_channel.c
└ath_net80211_set_channel(), wlan_modules\umac\if_lmac\if_ath.c
└ath_set_channel(), wlan_modules\lmac\ath_dev\ath_main.c

Mindwell expire

For Peregrine, the Scan State Machine is on target by default

Scan SM : Handles a single scan request at a time . Implements a simple scan state machine which executes a single scan request by hopping between channels and sending probe request with all the behavior configurable via the set of parameters specified in the scan request,



Foreign channel to BSS channel due to WLAN_SCAN_EV_DWELL_TIME_EXPIRE

main () perf_pwr_offload\drivers\target\src\os\athos\athos_main.c
- athos_main (), athos_main.c
  - check_idle () athos_main.c, If there are pending DSRS or background calls, process them and return FALSE. If there's nothing to do, then sleep until an interrupt wakes us up and return TRUE.
    - cmnos_intr_handle_pending_dsrs () target\src\os\common\cmnos_intrinf.c
      - _wal_intr_scheduler_intr_handler(), target\src\wlan\wal\ar\wal_intr_schd\wal_intr_schd.c, WAL main DSR. It can service multiple WAL pending interrupts
        - wal_tsf_timer_hdlr(), target\src\wlan\wal\ar\tsf_timer\wal_tsf_timer.c, TSF timer handler routine for the hw timer expiry event
          - cmnos_custom_timer_handler(), target\src\os\common cmnos_timer.c
            - tsf_timer_hdlr(), target\src\wlan\resmgr\resmgr_ocs.c
              - schedule(), resmgr_ocs.c
                - resmgr_ocs_is_cat_window_done(), resmgr_ocs.c
                  - wlan_scan_chreq_cb(), target\src\wlan\scan\wlan_scan_sm.c
                    - sm_dispatch(), perf_pwr_offload\drivers\common\src\sm\sm.c
                      - sm_dispatch_internal(), sm.c
        - wlan_scan_sm_state_foreign_chan_event(), wlan_scan_sm.c
          - sm_transition_to(), sm.c
            - wlan_scan_sm_state_bsschan_entry(), wlan_scan_sm.c:
              - wlan_scan_deliver_event(), wlan_scan_sm.c
                - _wlan_scan_internal_event_handler(), target\src\wlan\scan\wlan_scan.c

3.5 WMI Layer

3.5.1 IF WMI Layer

- Adaptation layer between umac and WMI.
  - Binds umac and wmi. Similar to if lmac that binds umac and lmac.
  - Attaches all the required components (htc,hif, wmi, txrx/htt ..etc) specific to 11ac offload chipsets (peregrine, hera ..etc).
  - Provides required services for txrx/htt layer.
  - Translates the function calls from UMAC to WMI messages to target.
  - Translates the WMI events from target to callbacks into UMAC.
  - Implements a WMI translation component for each module offloaded to FW.
    - Scan .
    - Vdev,pdev,peer .
    - Beacon/mgmt .
    - Pktlog and more .

ieee80211_send_mgmt(),
wlan_modules\umac\mlme\ieee80211_mgmt.c
└ ic->ic_mgtstart(ic, wbuf)
└**ol_ath_tx_mgmt_wmi_send**(), perf_pwr_offload\drivers \host\wlan\umac_offload_if\ ol_if_mgmt.c,
└wmi_unified_mgmt_send(), umac_offload_if\ ol_if_mgmt.c ,

3.5.2 WMI Layer

- Defines WMI commands/events.
- WMI Layer hides the endianness difference between host and target using the following rules.
  - Fields in WMI commands/events are always defined as 32bit integers. Entire message is integer (32-bit) aligned.
  - If a parameter for commands/events requires < 32bits then define a 32-bit integer field and , allocate different portions of the fields for parameters and use macros to get/set the individual parameters. But do not use bit fields in structures.
  - The WMI Layer (either SW or HW) simply performs byte swaps at the 32-bit integers level of the entire WMI message/event.
  - The Mac addresses are converted into 2 integers and stored inside the message. Use wmi_mac_addr structure and use WMI_CHAR_ARRAY_TO_MAC_ADDR and WMI_MAC_ADDR_TO_CHAR_ARRY.
  - perf_pwr_offload/include/wmi_unified.h has all of wmi command/event definitions.

- Implements API to allocate and send WMI commands.
  - Defines an abstract type called wmi buffer for storing sending wmi commands.
  - The wmi buffer is mapped to wbuf (network buffer) for now.
  - Allows us to change it to any data type at a latter time.
- Implements API to register/unregister for wmi events.
  - The event buffers also use wmi buffer abstraction.
  - Delivers events to the registered event handlers.
- Communicates with HTC to send and receive commands/events.
  - Adds WMI header to the WMI commands and invokes HTC to send wmi command. Releases wmi command buffer when the HTC completes transmitting the buffer .
  - Registers with HTC for WMI events and strips WMI header from events and sends up the actual event to the registered handler. Releases the emi event buffer after invoking callback.
  - Establishes a HTC service (virtual pipe) for WMI.

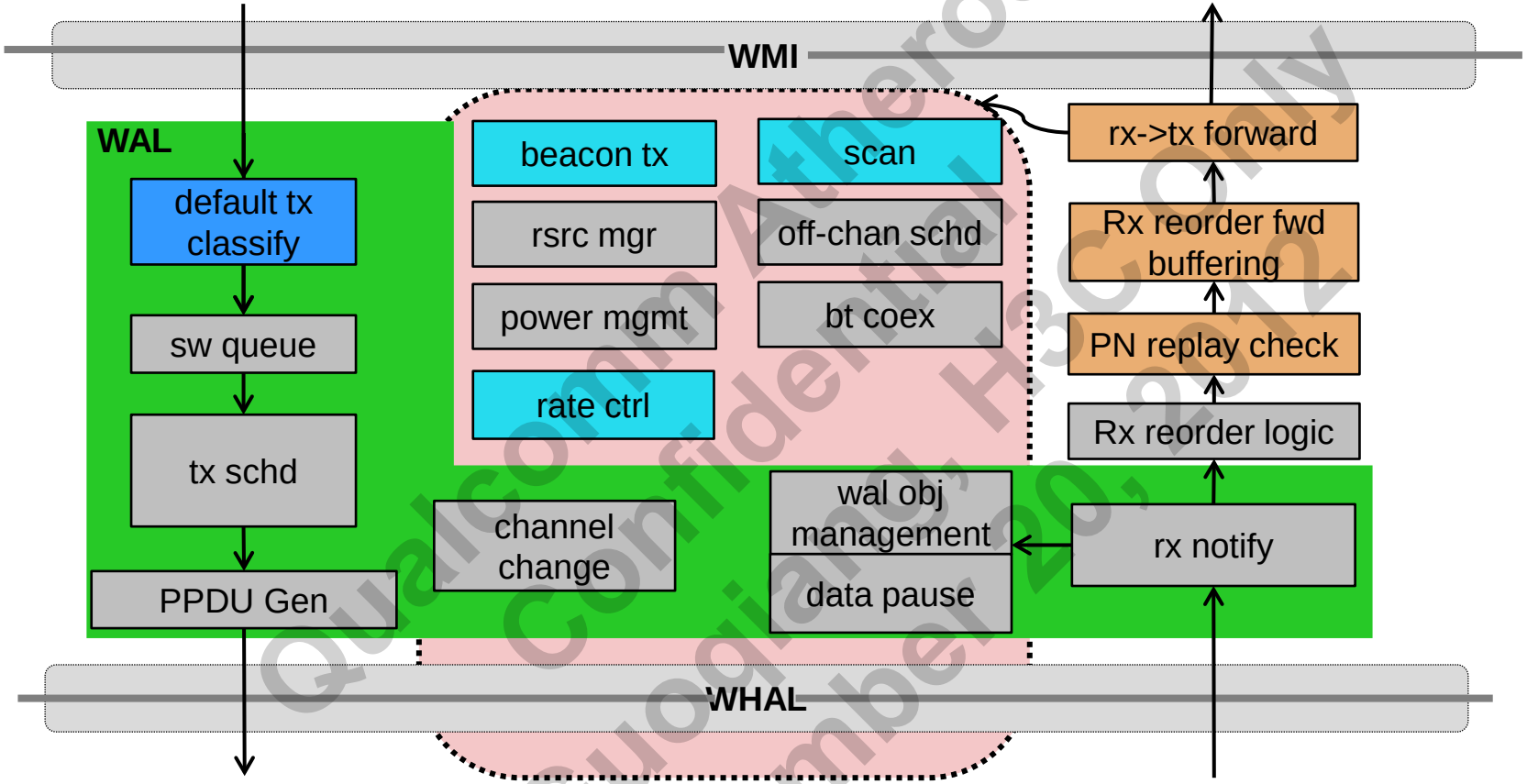
3.5.3 QCA Main WMI Commands/events

- Commands/events set are different from olca.
- Commands/events set are close to UMAC/LMAC interaction in newma.
- Commands/events are non blocking (Required for some Oss).
- Commands/events are grouped into several groups
  - Basic PDEV (IC in newma)
  - Basic VDEV (VAP in newma)
  - Basic peer (node in newma)
  - Mgmt (genreal tx/rx management and beacon)
  - Station Powersave.
  - More Groups to Come….

Example WMI Commands/events

- VDEV Commands
  - Create/Delete Vdev(vap) commands. The Vdev is identified by id. Both id and mac address are allocated on host.
  - Set multicast key.
  - Start, join, up, down commands.
- Peer Commands
  - Create/delete commands. Peer is identified by the mac address. No id is used.
  - Flush Tids command.
  - Set Key command.
- Events
  - SWBA event.
  - Management Frame RX.
  - Channel list updated (due to country code changed).

4. Target Firmware



4.1 FW download and Startup

4.1.1 BMI (Bootloader Module Interface)

- Bootcode runs from ROM
- With bootloader, BMI commands from host are accepted

_ce_pcie_host_if() target\src\os\athos\pcie.c
└ce_bmi_recv_cb() target\src\os\athos\pcie.c
 └A_BMI_DISPATCH()=cmnos_bmi_dispatch(),
 target\romexport\ar9888\hw.1\include\os\cmnos_api.h
 └cmnos_bmi_dispatch() target\src\os\common\cmnos_bmi.c, Bootloader Messaging Interface Dispatcher. Call the appropriate function to handle a BMI Command received from the Host.
  ├─cmnos_bmi_read_memory_cmd(), BMI_READ_MEMORY
  ├─cmnos_bmi_write_memory_cmd(), BMI_WRITE_MEMORY
  ├─cmnos_bmi_execute_cmd(), BMI_EXECUTE
  ├─cmnos_bmi_set_app_start_cmd(), BMI_SET_APP_START
  ├─cmnos_bmi_read_soc_register_cmd(), BMI_READ_SOC_REGISTER
  ├─cmnos_bmi_write_soc_register_cmd(), BMI_WRITE_SOC_REGISTER
  ├─cmnos_bmi_get_target_info_cmd(), BMI_GET_TARGET_INFO
  ├─cmnos_bmi_lz_stream_start_cmd(), BMI_LZ_STREAM_START
  └─cmnos_bmi_lz_data_cmd(), BMI_LZ_DATA

4.1.2 RAM Image Initialization

_stext () at crt1-tiny.S
└ main () target\src\os\athos\athos_main.c
  └ athos_main() athos_main.c

- Register various CPU exceptions to be handled by our exception handers
  (void)_xtos_set_exception_handler(…, misaligned_load_handler); …..
  (void)_xtos_set_exception_handler(…, fatal_exception_handler);
- athos_indirection_table_install(), athos_main.c, sets up the indirection table. clears the table and then installs function pointers for some of the most basic software modules.
- athos_init_part1(), athos_main.c, does a large number of miscellaneous hardware initializations for various modules, including 1) waits for hardware to be ready, 2) Various hardware inits, 3) A data set list is set up, 4) the interrupt system is initialized.
- EFUSE_MODULE_INSTALL(); OTPSTREAM_READ_MODULE_INSTALL(); otpstream_init(); install One Time Programmable (OTP) APIs
- nvram_module_init(); nvram_autoload(); install nvram APIs
- A_APPLY_OTP_PATCHES(OTPSTREAM_ID_PATCH2); contact_host(); A_APPLY_OTP_PATCHES(OTPSTREAM_ID_PATCH3); is intended to give the chance to modify our memory, including especially "host interest".
- athos_init_part2(), athos_main.c, do initializations that are supposed to be done after the host has had a chance to intervene.
- for (;;) { (void)check_idle(); }, athos_main.c, dead loop calling check_idle()

Populate Function Table

Structure _A_athos_indirection_table_t defines the layout of the indirection table, which is used to access exported APIs of various modules. The layout is shared across ROM and RAM code. RAM code may call into ROM and ROM code may call into RAM. Because of the latter, existing offsets must not change for the lifetime of a revision of ROM; but new members may be added at the end.

1) Table difinition

Root Function Table

target\include\os\athos_api.h

```
typedef struct
_A_athos_indirection_table {
_A_cmnos_indirection_table
_t   cmnos;
struct dset_api       dset;
struct gpio_api       gpio;
……
}
_A_athos_indirection_table_t
;
```

2nd Level Function Table

target\include\os\cmnos_api.h

```
typedef struct
_A_cmnos_indirection_table
{ …..
struct mem_api      mem;
struct misc_api     misc;
…. }
```

_A_cmnos_indirection_table_t;

3rd Level Function Table

target\include\os\misc_api.h

```
struct misc_api {
void (* _init)(void);
void (* _system_reset)(void);
   ….
};
```

2) Function Pointer Initialization

target\src\os\common\cmnos_power.c

```
void cmnos_power_module_install (struct power_api *tbl) {
tbl->_power_init = cmnos_power_init;
tbl->_power_hwinit = cmnos_power_hwinit;
tbl->_power_uart_control = cmnos_power_uart_control;
tbl->_power_syssleep_control= cmnos_power_syssleep_control; }
```

API name	indrt tbl field	Actual function
A_RESET()	**cmnos**.misc._system_reset	cmnos_system_reset()
A_SSCANF()	**cmnos**.sscanf._sscanf	sscanf()
A_POWER_INIT()	**cmnos**.power._power_init	cmnos_power_init()
….	….	….

3) API Calling

target\include\os\cmnos_api.h

```
#define A_CMN(sym)    _A_OS_INDIRECTION_TABLE->cmnos.sym
#define A_POWER_INIT()      A_CMN(power._power_init())
```

target\src\os\athos\athos_main.c

```
void  athos_init_part1(void) { ….
A_TIMER_INIT();  A_POWER_INIT();  …}
```

4.1.3 Working Loop

check_idle() athos_main.c

- A_INTR_HANDLE_PENDING_DSRS(), target\include\os\cmnos_api.h
  - cmnos_intr_handle_pending_dsrs(), target\src\os\common\cmnos_intrinf.c, It calls the pending DSRs (delayed service routines) which were pended by interrupt service routines
    - dsr_rv = intr_svc->dsr(intr_svc->dsr_arg); Loop to call all DSRs (Delayed Service Routine).
      - _wal_intr_scheduler_intr_handler (), target\src\wlan\wal\ar\wal_intr_schd\wal_intr_schd.c
        - wal_tsf_timer_hdlr, target\src\wlan\wal\ar\tsf_timer\wal_tsf_timer.c, TSF timer handler routine for the hw timer expiry event. Call a customer timer handler.
- athos_wait_for_activity(), athos_main.c, it sleeps the processor while waiting for interrupts.

cmnos_intr_handle_pending_dsrs() is the main polling loop for handling events that deal with wireless, and Host communications.

- It is the responsibility of each event/interrupt handler to avoid monopolizing the CPU.
- If a handler is taking "too long", it should return A_HANDLER_YIELD, which causes the handler to be called again later (after other handlers have had a chance to run).
- If a handler returns A_HANDLER_NOENABLE, it will not be called again until the associated event/interrupt is re-enabled.

WMI WLAN Init command Processing

A_INTR_HANDLE_PENDING_DSRS(), target\include\os\cmnos_api.h
└ cmnos_intr_handle_pending_dsrs(), target\src\os\common\cmnos_intrinf.c,
 └ dsr_rv = **intr_svc->dsr**(intr_svc->dsr_arg); Loop to call all DSRs (Delayed Service Routine).
  └ _CE_per_engine_handler(), target\src\hostif\copy_engine\copy_engine.c
   └ _HIF_CE_recv_cb(), target\src\hostif\hif\hif_ce\hif_ce.c, Receive Callback function, registered with the Copy Engine module. Called whenever a buffer is received (e.g. from Host).
    └HifLayerRecvCallback(), target\src\hostif\htc\htc.c
     └ _HTCPipeIndicateRecvMgs (), htc.c
      └WMIRecvMessageHandler(), target\src\hostif\wmisvc\wmi_svc.c
       └**pCmdHandler**(pContext,cmd, info1, pCmdBuffer,length), call dispatch function

WMI_DISPATCH_ENTRY wlan_pdev_dispatchentries[] = { {**dispatch_wlan_init_cmd**, WMI_UNIFIED_INIT_CMDID, sizeof(wmi_unified_init_cmd)}, …}, target\src\wlan\init\wlan_dev.c
└dispatch_wlan_init_cmd(), target\src\wlan\init\wlan_dev.c
 └wlan_main_init()= _wlan_main_init(), target\src\wlan\include\wlan_init.h
  _wlan_main_init(), target\src\wlan\init\wlan_init.c
  └wlan_dev_attach()= _ wlan_dev_attach(), target\src\wlan\include\wlan_dev.h
   _wlan_dev_attach(), target\src\wlan\init\wlan_dev.c
   └wal_dev_attach()= _ wal_dev_attach (), target\src\wlan\wal\include\wal_phy_dev.h
    _wal_dev_attach (), target\src\wlan\wal\ar\dev\ar_wal_phy_dev.c

_wal_dev_attach (), Allocate and initialize a device/radio

- whalSetMacAggrDelim();
  whalSetMacDmaBurstSize();
- whalSetRegulatoryDomain()
- wal_intr_scheduler_init(), Attach Intr module
- wal_tsf_timer_module_init(), init TSF timer
- tx_send_attach(), Attach TX Send
- ar_wal_tx_q_init(), Attach TX Q
- tx_sched_init(), Attach TX Scheduler
- rx_attach(), Attach RX
- ar_wal_ast_attach(), attach AST module
- ar_wal_peer_attach(), attach peer module
- ar_wal_vdev_attach(), attach vdev handlers
- local_frame_attach(), attach local module
- ar_wal_connection_pause_attach(), attach the connection pause
- wal_rc_enable_dyn_bw(), enable dynamic bw by default
- wal_coex_complete_init()
- wal_intr_scheduler_event_register() , resgister various event handlers
- Init the wdog reset timer

Attach the master handler with the OS
_wal_intr_scheduler_intr_handler()

Setup the HW TXQs

```
……
/* UAPSD / PSPOLL */
tx_send_txq_setup(tx_ctxt, WAL_TXQ_ID_PSPOLL,
WHAL_TX_QUEUE_PSPOLL, WHAL_WMM_BEACON,
WMM_AC_PSPOLL);
/* Data Queues */
tx_send_txq_setup(tx_ctxt, WAL_TXQ_ID_WMM_BE,
WHAL_TX_QUEUE_DATA, WHAL_WMM_BE, WMM_AC_BE);
……
tx_send_txq_setup(tx_ctxt, WAL_TXQ_ID_CAB,
WHAL_TX_QUEUE_CAB, WHAL_WMM_VO, WMM_AC_VO);
…..
```

Register event handler

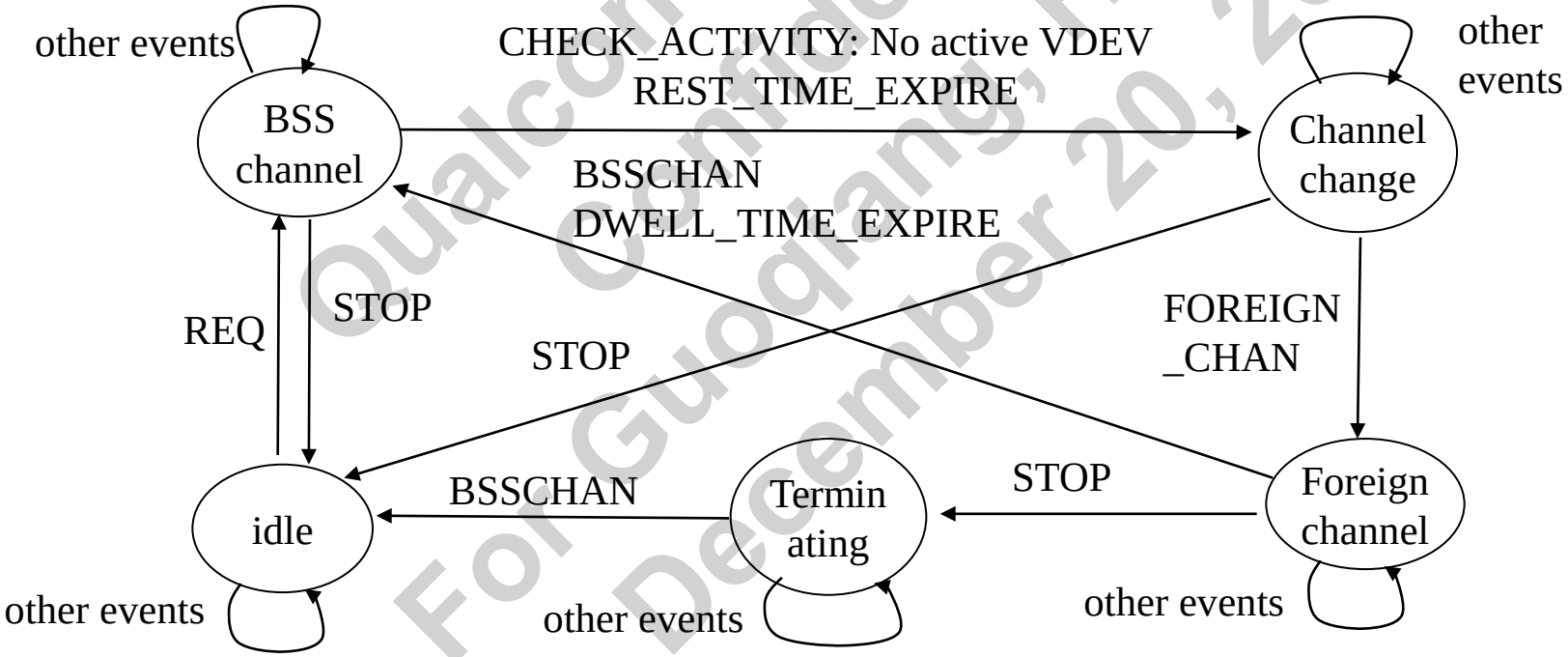
FATAL	O	wal_dev_fatal_int_handler()
MIB	O	wal_dev_mib_int_handler()
BB_PANIC	O	wal_dev_baseband_panic_handler()
BB	O	wal_dev_baseband_timeout_handler()

4.2 Key OLF modules

4.2.1 Scan

- Implements a scan state machine. Exposes very flexible API to support several types of scans.
  - ✓User requested scans directed and undirected.
  - ✓P2P Device scans; CCX Scan.
  - ✓All passive scan; Back ground scans.

- Scan Scheduler
  - ✓Accepts scan requests from multiple modules.
  - ✓Queues , Prioritizes scan requests and issues them one at a time to the scan module.
  - ✓Allows prioritization to be configurable.



Foreign channel → BSS channel

check_idle() athos_main.c
- cmnos_intr_handle_pending_dsrs(), target\src\os\common\cmnos_intrinf.c
  - dsr_rv = intr_svc->dsr(intr_svc->dsr_arg); Loop to call all DSRs (Delayed Service Routine).
    - _wal_intr_scheduler_intr_handler (), target\src\wlan\wal\ar\wal_intr_schd\wal_intr_schd.c
      - wal_tsf_timer_hdlr, target\src\wlan\wal\ar\tsf_timer\wal_tsf_timer.c, TSF timer handler routine for the hw timer expiry event. Call a customer timer handler.

wal_tsf_timer_hdlr, target\src\wlan\wal\ar\tsf_timer\wal_tsf_timer.c
- cmnos_custom_timer_handler(), target\src\os\common\cmnos_timer.c, handler for TSF timer.
  - tsf_timer_hdlr(), target\src\wlan\resmgr\resmgr_ocs.c
    - schedule (), resmgr_ocs.c
      - resmgr_ocs_is_cat_window_done(), resmgr_ocs.c
        - wlan_scan_chreq_cb(), target\src\wlan\scan\wlan_scan_sm.c
          - sm_dispatch(), **perf_pwr_offload\drivers\common\src\sm\sm.c**

sm_dispatch_internal (), sm.c
- wlan_scan_sm_state_foreign_chan_event(), wlan_scan_sm.c
  - sm_transition_to(), sm.c
    - wlan_scan_sm_state_bsschan_entry(), wlan_scan_sm.c
      - wlan_scan_deliver_event(), wlan_scan_sm.c
        - _wlan_scan_internal_event_handler(), wlan_scan.c

4.2.2 Management and Beacon TX/RX

- Most of the Management frames are generated on host.
- The management frames can be sent to the target either by reference or by value. Send by reference is only valid for LL(low latency) interface like pcie.
- Supports 2 modes for beacon and probe response transmission.
  - ***Firmware-heavy/firmware mode***: This is the preferred mode for most solutions where performance is the predominant requirement. In this mode Beacons, TX/RX of Probe responses, ADDBA, DELBA frames are handled by the firmware while all other management frames are generated (on TX) and processed (on RX) by the host. Note that the maximum number of VDEVs that needs to be supported in this mode is 16.
  - ***Host-heavy /Host mode***: This mode is desired for solutions where flexibility is important. In this mode the host generates and processes all management frames with the exception of ADDBA, DELBA frames.
  - In both the modes the power save specific IEs (like TIM, p2p GO NOA ..) in beacons/probe responses are generated by FW(firmware).in Host mode FW sends these IEs along with the SWBA event.
- Received beacons are sent up to the host based on beacon filter criteria set up by the host.

4.2.3 Resource Manager

- Manages channel.
- Arbitrates channel between multiple requestors (VDEV,scan ..etc).
- Resolves conflicting channel requests from multiple requestors .
- Switches channel from one requestor to another by coordinating between them.

  - example, when resmgr moves channel from vdev(s) to scanner, it asks each vdev to perform pause operation(a STA vdev enters network sleep part of pause) before taking the channel away from vdev(s).

- Invokes off channel scheduler for multi channel operation.

4.2.4 Roam

- Performs Back Ground(BG) scanning. The conditions to trigger BG scan are configurable from host(rssi threshold, periodicity ..etc)
- Invokes the host MLME via a WMI event to roam to a new AP only if there is a better AP is found. (called Soft Roam).
- If beacon miss, invokes the host MLME with the candidate AP list.
- Supports two modes for indicating bg scan results to host.
  - Mode 1 : during the back ground scan , if a beacon/probe response received from a better AP sends the whole beacon/probe response to host.
  - Mode 2: collects all the candidate Aps and sends up the list at the end of BG scan.

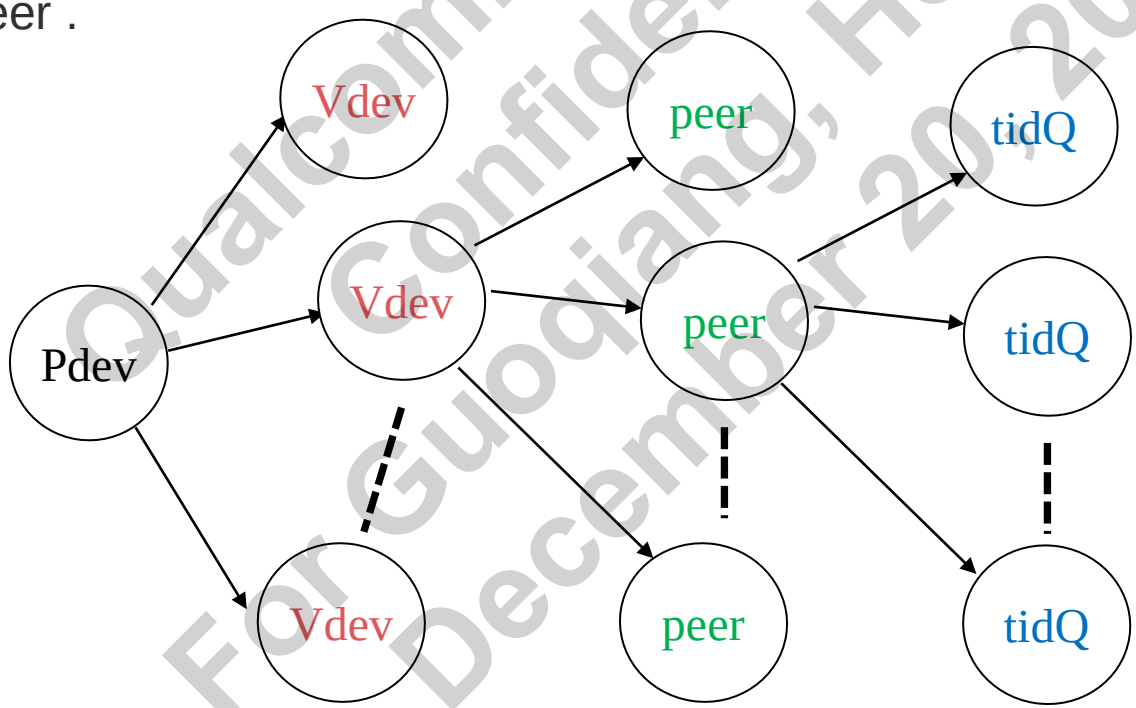
4.3 WAL

- WLAN Abstraction Layer
- Abstracts the HW
- Abstracts more higher level HW functionality than WHAL/HAL.
- Abstracts the higher level functionalities like tx AMPDU, AMSDU and 802.11 encapsulation, frame queuing and tid que scheduling …etc.
- Allows legacy and future chipset support without changing the upper layers.
- supports functionality required for NART.

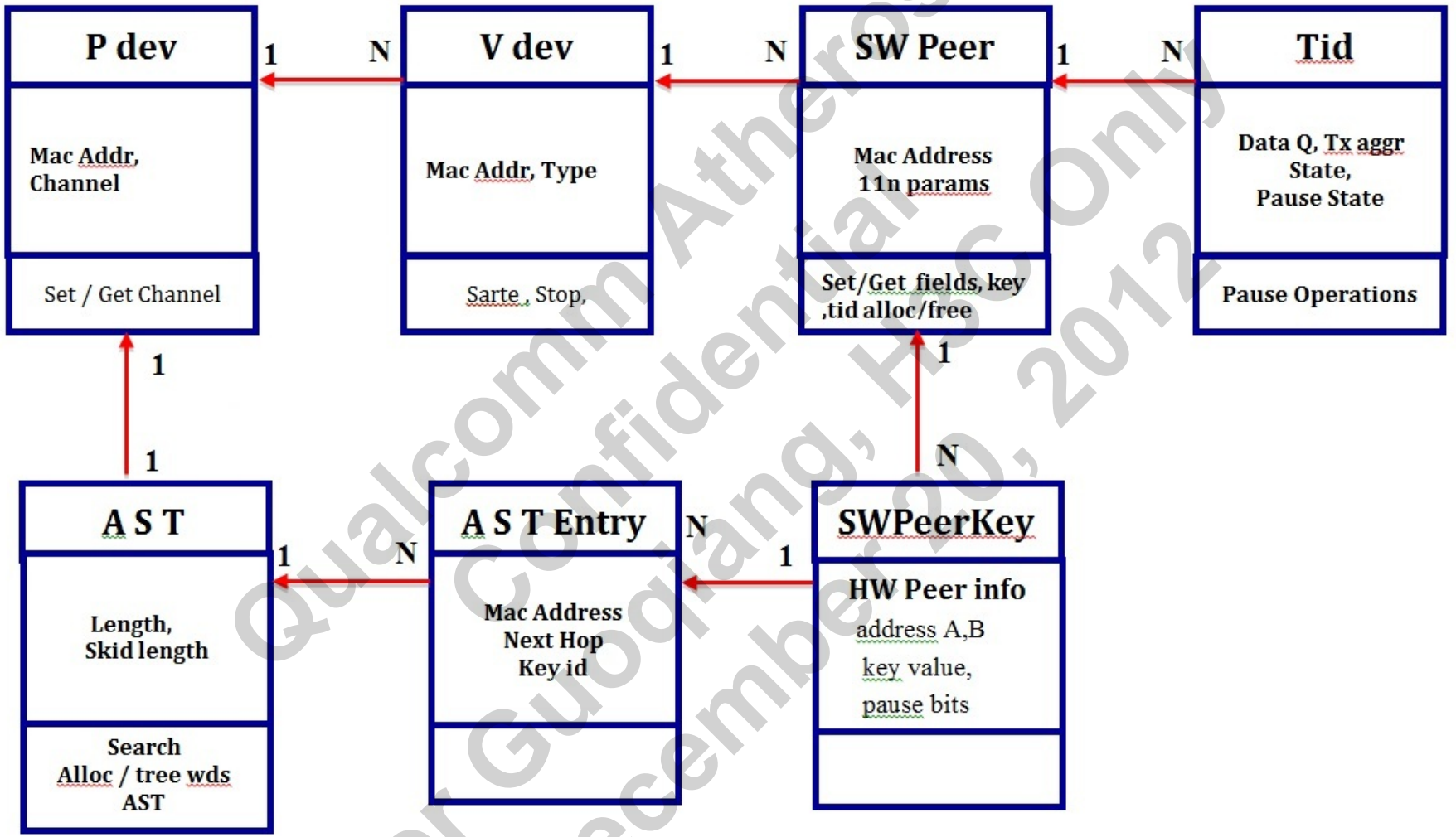
4.3.1 WAL Objects

Provides a set of services through following WAL objects

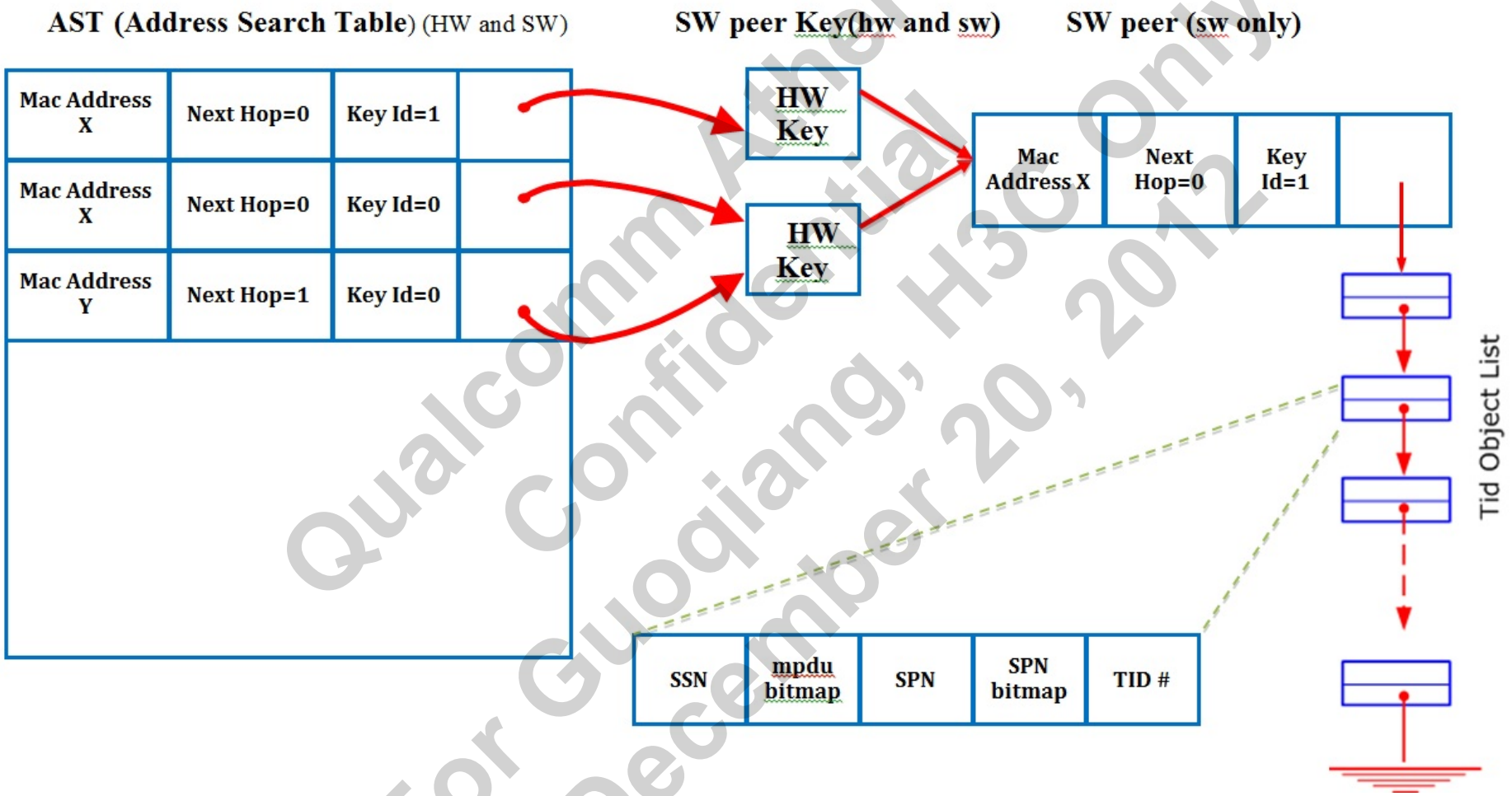
- Pdev : Represents and provides services for a physical radio .
- Vdev : Represents and provides services for a virtual 802.11 mac (called vap on host).
- Peer : Represents and provides services for a connected peer device (analogous to conn_t on olca and node on newma).
- TID Queue: Represents and provides a data queue for a given qos tid with in a peer .



WAL objects



AST (Address search Table)



WAL Pdev Services

- Start/stop the device.
- Set/Get Rxfilter, Chain mask.
- Channel change/reset.
- Create/delete Virtual device.
- Set/Get chip power state (network sleep, active, full sleep)
- Set/get country code and regulatory domain .
- Query HW capabilities(11ac/11n, number of chains/streams ...etc)
- Get channel list.
- Set/get mac address, bssid mask.
- Set quiet ie, slot time.
- Enable/disable radio.
- Get channel property (noise, extension channel busy,
- Set/get ac queue (BE,BK ...) properties(cwmin,cwmax,txop ...etc).
- Packet log related operations (enable/disable, fetch packet log data)
- Set/get misc properties related interrupt mitigation, connection scheduler …etc.

target\src\wlan\wal\include\wal_phy_dev.h

```
wal_dev_attach();
wal_dev_detach();
wal_dev_init();
wal_phy_dev_set_param();
wal_phy_dev_get_param();
wal_dev_detach();
wal_dev_attach();
wal_dev_detach();
wal_phy_dev_set_power_state();
wal_phy_dev_get_power_state();
….
```

WAL Vdev Services

- Up/listen/down operations.
- Beacon setup, enable, disable, pause.
- ATIM,TIM,DTIM,BMISS notifications.
- SWBA, beacon tx complete, tbtt, tsf jump notifications.
- Reconfigure beacon timers operation.
- Create/delete peer.
- Get mac address, 32 bit tsf,64 bit tsf .
- Tsf timer support (alloc, free, set, cancel ..etc)
- Add/remove multicast keys with keyids.
- Set/get aid, bssid, multicast rate.
- Enable/disable/flush cab queue data.
- Set/get MIMO power save (Static/Dynamic/disable)..

target\src\wlan\wal\include\wal_virtual_dev.h

wal_vdev_attach(),
wal_vdev_detach(),
wal_vdev_start(),
wal_vdev_stop(),
wal_vdev_up(),
wal_vdev_down()
wal_vdev_set_param(),
wal_vdev_get_param(),
….

_wal_vdev_attach(),
_wal_vdev_detach(),
_wal_vdev_start(),
_wal_vdev_stop(),
_wal_vdev_up(),
_wal_vdev_down()
_wal_vdev_set_param(),
_wal_vdev_get_param()
, ….

WAL Peer Services

- Add/remove unicast key(key + mic + type) with keyids.
- Set/get properties (tx chain mask, power, tx chain mask, qos capability,11ac capability, PS bit, amsdu length, ...etc).
- Set/get upper auth related properties (auth state, frame type allowed when not authed).
- Set/update negotiated rates.
- Get number of frames queued on HW/SW and notification when the number reaches 0 .
- Flush queues.
- Data pause related operations (support AP/STA UAPSD, AP pspoll, fake sleep, multi channel, Wifi Direct ...etc).
- Tid allocationi/delete notifications.
- Send/Receive Data/Control/Management frames
- Generic TX frame completion notification for requested frames.
- RX MIC error notifications.
- Fixed rate.
- query tx/rx stats including last data (unicast/multicast/rx/tx) time stamp.

target\src\wlan\wal\include\wal_peer.h

wal_peer_alloc(),
wal_peer_free(),
wal_peer_set_key(),
wal_peer_get_key(),
wal_peer_set_param(),
wal_peer_get_param()
....

WAL Tid Queue Services

- Pause/unpause data on a tid.
- Allow n frames from a tid while data is paused (to support UAPSD, pspoll response)
- Enable/disable aggregation and set/get aggr related params.

4.3.2 WAL SW Components

- DE: descriptor engine. Performs packet classification and packet queuing to the individual tid queues. Handle multicast,unicast frames for different VDEV types.
- Dev: pdev,vdev object services.
- Peer: peer and tid objectservices. address search table manipulation.
- Tx_sched: scheduling multiple tid queues assigned to a given HW TXQ.one instance for each HW queue. In addition to 4 AC queues, beacon queue, cabq and UAPSD queues are all managed by the scheduler.
- Pause: handle pause of tx frames at tid,peer,vdev,pdev level.
- Rx: rx frame processing . Both local ring and remote rings. Invoke separate callbacks for local and remote frames (remote is for LL only).
- Tx: create and queue AMPDU/AMSDU/MPDU frame to HW queue. Handle frame completion , block ack window and MPDU retry.

- Local_frames: local frame transmission . Frame transmission from firmware memory (management , null frames ..etc).
- beacon: beacon transmission for multiple. SWBA event generation. SWBA/DBA/tbtt timer manipulation
- Channel change: handles channel change functionality.
- Ratectrl : per peer rate control .also handles management rates.
- Waltest: set of wal level tests to verify the functionality of all the WAL components.
- Concurrency: handles all the concurrency related aspects. Chip mode, tsf clock allocation among multiple vdevs and the Rx filter manipulation.

Connection Pause

- Centralize/consolidate all mechanisms to pause data.
- Supports all the following upper protocol requirements.
  - STA Power Save (UAPSD , Pspoll)
  - STA Fake Sleep
  - AP Power Save mechanisms(UAPSD, Pspoll , normal)
  - P2P Power Save Mechanisms(NOA, CTWINDOW)
  - Multi channel/off channel support.
  - IBSS Power Save.
  - BT COEX.
- pause/unpause on a peer or tid queue.
- support a different pause called hard pause
- Allow a set of frames from a paused tid queue

  - UAPSD/pspoll response.

  - send null frame for fake sleep.

4.4 WHAL

- WLAN HW Abstraction Layer
- Same as before , abstracts the low level HW as before
- one difference is that the 11ac descriptors are directly manipulated by WAL as opposed to WHAL.

target\src\wlan\wal\whal\recv\
ar600P_recv.c

```
static const WHAL_RECV_API
ar6000RecvApi = {
    ar6000ProcRxDesc,
    ar6000SetupRxDesc,
    ar6000GetRxDP,
    ar6000SetRxDP,
    ar6000SetRxFilter,
    ar6000GetRxFilter,
    ar6000StopDmaReceive,
    ar6000StartDmaReceive,
…..};
```

target\src\wlan\wal
\whal\phy*.ini

target\src\wlan\wal\whal\xmit\
ar600P_xmit.c

```
static const WHAL_XMIT_API
ar6000XmitApi = {
    ar6000StopTxDma,
    ar6000GetTxQueueProps,
    ar6000SetTxQueueProps,
    ar6000SetRateControlCallback,
    ar6000AllocateTxQueue,
    ar6000ReleaseTxQueue,
    ar6000GetTxDP,
    ar6000SetTxDP,
    ar6000KeepBeaconTimers,
    ar6000NumTxPending,
    ar6000XmitReset,
    ar6000TxopProtectEnable,
    ar6000GetAggrTxDelim,
    ar6000GetActiveTxQueues,
    ar6000SetupTxStatusRing,
    ar6000ResetTxStatusRing,
    ar6000ProcTxStatusDesc,
    ar6000StopTxq,
    ar6000SetPostFrameBackoff,
    ar6000ProcTxDesc,
    ar6000SetupPpduCommon,
    ar6000SetupPpduRateSeries,
    ar6000SetupPpduSeqno,
    ar6000GetDescInfo };
```

4.5 Frameworks/utilities

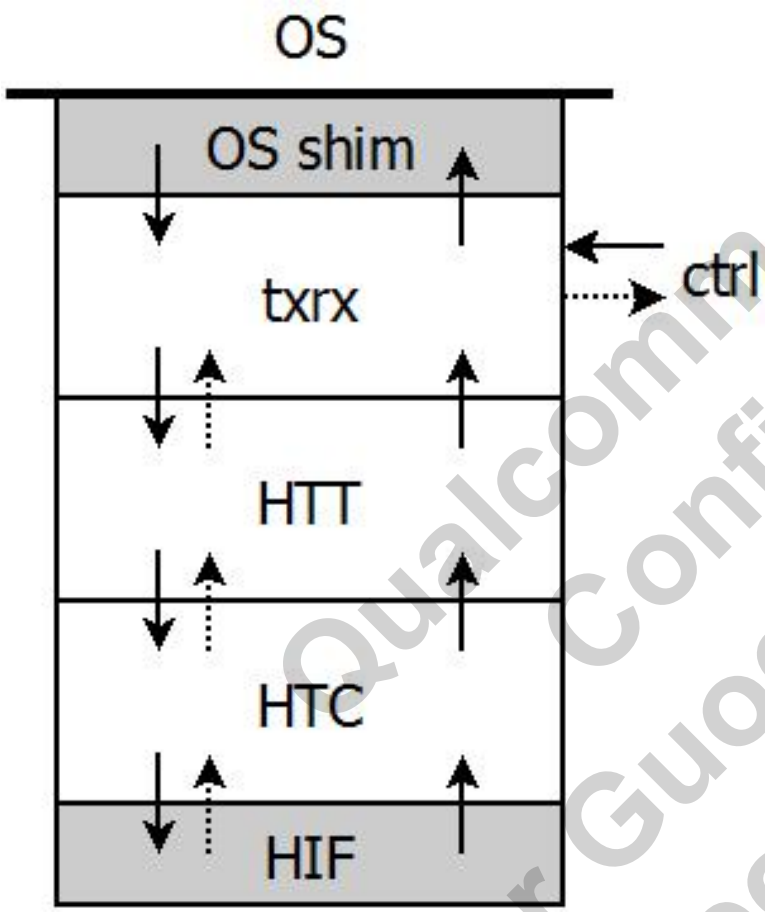
- Athos
  - Supports basic os services.
  - Unchanged from olca.
- Pool Manager
  - Allows a pre allocation of a pool of objects of same type.
  - Provides api for allocation/ deallocation of the objects from pool.
- SM frame work
  - Provides a framework for creating state machines.
  - Imported from newma.

```
_stext () at crt1-tiny.S
└main () target\src\os\athos\athos_main.c
 └athos_main() athos_main.c
  └check_idle() athos_main.c
   └A_INTR_HANDLE_PENDING_DSRS(),
    target\include\os\cmnos_api.h
     └cmnos_intr_handle_pending_dsrs(),
      target\src\os\common\cmnos_intrinf.c
      └ dsr_rv = intr_svc->dsr(intr_svc->dsr_arg);
       Loop to call all DSRs.
```

```
POOL_MGR_API_FN
_pool_mgr_fn = {
    _pool_init,
    _pool_alloc,
    _pool_free,
    _pool_destroy,
    _pool_node_to_id,
    _pool_node_from_id,
};
```

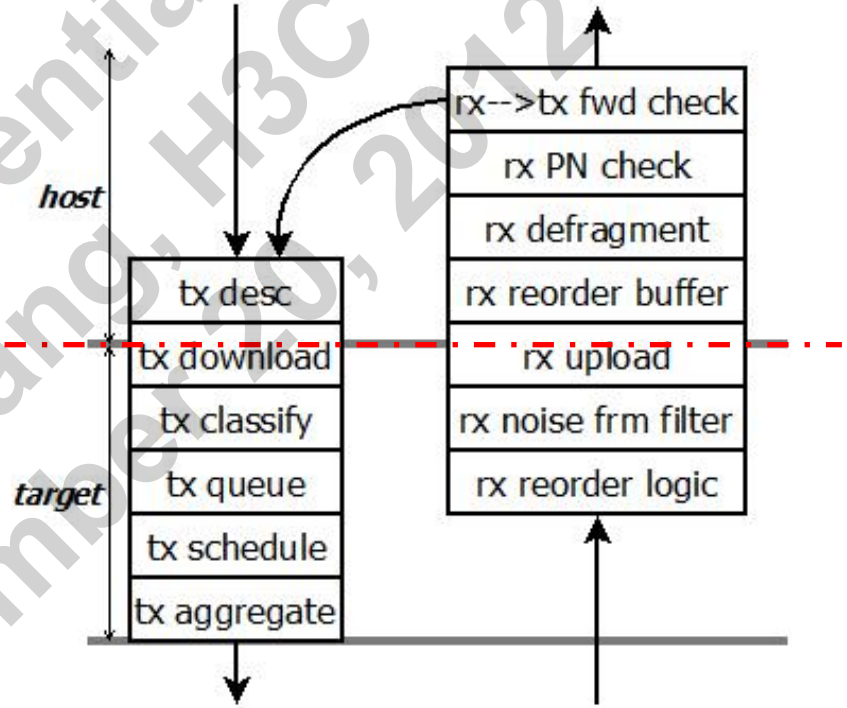
5. Tx/Rx Data Path

5.1 Host Tx/Rx Data Path



Low-Latency

- Tx: host only creates descriptor
- Rx: host checks PN and Rx → Tx forwarding



5.1.1 Layer Overviews – OS Shim

Purpose: provide abstract interaction with OS

- API: tx, rx, monitor
- Data types: adf_nbuf
- Do DMA mapping for tx frames
- If required, provide a pool of pre-mapped rx and management netbufs
- Invoke TSO segmentation

Newma OS Shim

wlan\os\linux\src\osif_umac.c
intosif_ioctl_create_vap(){ …
dev->open = osif_vap_open;
dev->stop = osif_vap_stop;
dev->**hard_start_xmit** =
    osif_vap_hardstart;
dev->set_multicast_list =
    osif_set_multicast_list;
….}

wlan\os\linux\src\osif_umac.c
static wlan_event_handler_table
common_evt_handler = {
  osif_receive,
  osif_receive_filter_80211,
  osif_receive_monitor_80211,
  osif_dev_xmit_queue,
  osif_vap_xmit_queue,
 osif_xmit_update_status, };

dev_hard_start_xmit() dev.c
└**dev->hard_start_xmit** = **osif_vap_hardstart**,
 wlan\os\linux\src\osif_umac.c
 └**osif_vap_hardstart**(), os\linux\src\osif_umac.c
  └**wlan_vap_send**(), umac\txrx\ieee80211_output.c

ieee80211_input(), umac\txrx\ieee80211_input.c
└**vap->iv_evtable->wlan_receive**(),
 wlan\include\ieee80211_defines.h
  └**osif_receive**(), wlan\os\linux\src\osif_umac.c
   └ netif_rx(), net/core/dev.c

QCAmain OS Shim

The rx portion is filled in by OSIF SW before calling ol_txrx_osif_vdev_register; inside the ol_txrx_osif_vdev_register the txrx SW stores a copy of these rx function pointers, to use as it delivers rx data frames to the OSIF SW.

wlan_modules\os\linux\src\osif_umac.c

```
static void osif_vap_setup()
{ …..
#ifndef QCA_PARTNER_PLATFORM
ops.rx.std = (ol_txrx_rx_fp) osif_deliver_data_ol;
#else
ops.rx.std = (ol_txrx_rx_fp) osif_pltfrm_deliver_data_ol;
#endif
ops.rx.mon = (ol_txrx_rx_mon_fp) osif_receive_monitor_80211_base;
ol_txrx_osif_vdev_register( osifp->iv_txrx_handle, (ol_osif_vdev_handle) osifp, &ops);
osifp->iv_vap_send = ops.tx.std;
….. }
```

ol_rx_deliver(), host\wlan\txrx\ol_rx.c
└ vdev->osif_rx()=**osif_deliver_data_ol**()
wlan_modules\os\linux\src\osif_umac.c
└netif_rx(), net/core/dev.c

The tx portion is filled in by the txrx SW inside ol_txrx_osif_vdev_register; when the function call returns, the OSIF SW stores a copy of these tx functions to use as it delivers tx data frames to the txrx SW.

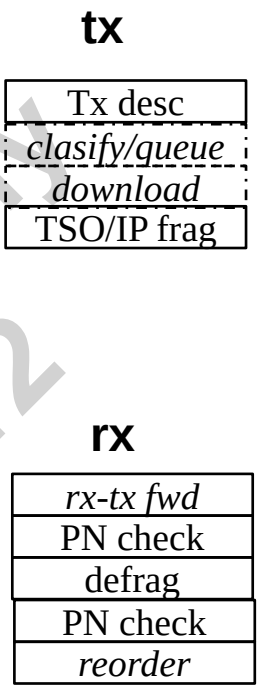
host\wlan\txrx\ol_txrx.c

```
void ol_txrx_osif_vdev_register()
{ …
   vdev->osif_rx = txrx_ops->rx.std;
   vdev->osif_rx_mon = txrx_ops->rx.mon;
….
   txrx_ops->tx.std = vdev->tx = ol_tx_ll;
   txrx_ops->tx.non_std = ol_tx_non_std_ll;
}
```

dev_hard_start_xmit()
└osif_vap_hardstart_generic(),
wlan_modules\os\linux\src\osif_umac.c
└ol_tx_ll(), host\wlan\txrx\ol_tx.c
└ ol_tx_send(), host\wlan\txrx\ol_tx_send.c

5.1.2 txrx: perform protocol data processing

- tx: store meta-data in SW tx descriptor
- HL tx: classify + queue tx frames
- HL tx: choose which tx frames to download next
- tx: TSO segmentation + IP fragmentation
- rx: reorder buffering
- rx: PN check (if applicable)
- rx: rx → tx forwarding
- rx: defragmentation
- rx: LRO (batch delivery of frames from the same “flow”)
- tx: publish tx status information for packetlog (or other tools)
- rx: publish rx descriptor information for packetlog (or other tools)



perf_pwr_offload\drivers\host\htt\htt_t2h.c

void **htt_t2h_msg_handler**(void *context, HTC_PACKET *pkt)
{ …..
ol_rx_indication_handler(pdev->txrx_pdev, htt_t2h_msg, peer_id, tid, num_mpdu_ranges);
…….}

Possibilities in Rx processing in txrx module.

1. Nothing

rx->tx forwarding and rx PN entirely within target. (This is unlikely; even if the target is doing rx->tx forwarding, the host should be doing rx->tx forwarding too, as a back up for the target's rx->tx forwarding, in case the target runs short on memory, and can't store rx->tx frames that are waiting for missing prior rx frames to arrive.)

2. Just rx -> tx forwarding.

This is the typical configuration for HL, and a likely configuration for LL STA or small APs.

3. Both PN check and rx -> tx forwarding.

This is the typical configuration for large LL APs. Host-side PN check without rx->tx forwarding is not a valid configuration, since the PN check needs to be done prior to the rx->tx forwarding.

host\ wlan\txrx\ol_tx_send.c

void ol_tx_send(struct ol_txrx_vdev_t *vdev, struct ol_tx_desc_t *tx_desc, adf_nbuf_t msdu)
{ …
htt_tx_send_std(pdev->htt_pdev, tx_desc->htt_tx_desc, msdu, id)) …}

When the tx frame is downloaded to the target, there are two outstanding references:

1. The host download SW (HTT, HTC, HIF). This reference is cleared by the ol_tx_send_done callback functions.
2. The target FW. This reference is cleared by the ol_tx_completion_handler function.

5.1.3 HTT/HTC/HIF

HTT: abstraction of HL vs. LL host/target data transfer

- tx: add tx meta-data needed by target (HTT tx desc)
- rx: descriptor abstraction
- LL rx: MAC DMA rx ring
- Host ←→ target data-related messages

HTC: virtualization of host / target communication

- Store frames until the underlying HIF layer can accept them
- Map physical “pipes” to virtual “endpoints” (*necessary for HL, useful for LL*)
- Provide flow control between endpoints

HIF: abstraction of physical host / target communication

- Send data to target
- Invoke callback when the send is completed
- Invoke callback when data is received from the target

```
void hif_completion_thread_startup(struct HIF_CE_state *hif_state)
{ …
 if (attr.src_nentries) { /* pipe used to send to target */
        CE_send_cb_register(pipe_info->ce_hdl, HIF_PCI_CE_send_done, pipe_info, attr.flags &
CE_ATTR_DISABLE_INTR); ….}
      if (attr.dest_nentries) { /* pipe used to receive from target */
        CE_recv_cb_register(pipe_info->ce_hdl, HIF_PCI_CE_recv_data, pipe_info); …}
….}
```

drivers\wlan_modules\htc\htc\htc_host.c

```
/* registered target arrival callback from the HIF layer */
static hif_status_t HTCTargetInsertedHandler(hif_handle_t hHIF, adf_os_handle_t os_hdl)
{ ….  /* setup HIF layer callbacks */
      htcCallbacks.rxCompletionHandler = HTCRxCompletionHandler;
      htcCallbacks.txCompletionHandler = HTCTxCompletionHandler;
      htcCallbacks.usbStopHandler     = HTCUsbStopHandler;
      htcCallbacks.usbDeviceRemovedHandler = HTCUsbDeviceRemovedCheck;   /* Current, only Linux
paltform used. */ ….}
```

perf_pwr_offload\drivers\host\htt\htt.c

```
int htt_htc_attach(struct htt_pdev_t *pdev)
{ …
   connect.EpCallbacks.EpTxComplete = htt_h2t_send_complete;
   connect.EpCallbacks.EpRecv = htt_t2h_msg_handler;
   connect.EpCallbacks.EpSendFull = htt_h2t_full;
…}
```

5.1.4 The whole Function Flows

Linux Rx

tasklet_action+0x55/0xe0, linux
└ath_tasklet(), perf_pwr_offload\drivers\host\hif\pci\linux\ath_pci.c
 └CE_per_engine_service_any(), perf_pwr_offload\drivers\host\hif\pci\copy_engine.c Handler for per-engine interrupts on ALL active CEs. This is used in cases where the system is sharing a single interrput for all CEs
  └CE_per_engine_service(), copy_engine.c; Guts of interrupt handler for per-engine interrupts on a particular CE.

HIF_PCI_CE_recv_data(), perf_pwr_offload\drivers\host\hif\pci\hif_pci.c, Called by lower (CE) layer when data is received from the Target.
└hif_completion_thread(), perf_pwr_offload\drivers\host\hif\pci\hif_pci.c, This thread provides a context in which send/recv completions are handled.
 └msg_callbacks->rxCompletionHandler()
  └HTCRxCompletionHandler(), drivers\wlan_modules\htc\htc\htc_recv.c
   └RecvPacketCompletion(), drivers\wlan_modules\htc\htc\htc_recv.c
    └DoRecvCompletion(), drivers\wlan_modules\htc\htc\htc_recv.c
     └pEndpoint->EpCallBacks.EpRecv()
      └htt_t2h_msg_handler(), perf_pwr_offload\drivers\host\htt\htt_t2h.c

ol_rx_indication_handler(), perf_pwr_offload\drivers\host\wlan\txrx\ol_rx.c
└ol_rx_reorder_release (), perf_pwr_offload\drivers\host\wlan\txrx\ol_rx_reorder.c
 └peer->rx_opt_proc() = ol_rx_deliver(), host\wlan\txrx\ol_rx_pn.c
  └ol_rx_fwd_check(), host\wlan\txrx\ol_rx_fwd.c
   └ol_rx_deliver(), host\wlan\txrx\ol_rx.c
    └vdev->osif_rx()= osif_deliver_data_ol() wlan_modules\os\linux\src\osif_umac.c
     └netif_rx(), net/core/dev.c

Linux Tx

do_IRQ()→ irq_exit()→ do_softirq()→ __do_softirq()→ net_rx_action()→ gfar_poll() linux

└gfar_clean_rx_ring()→ netif_receive_skb()→ br_handle_frame()→ br_handle_frame_finish()

└br_dev_queue_push_xmit()→ dev_queue_xmit()→dev_hard_start_xmit()

└osif_vap_hardstart_generic(), drivers\wlan_modules\os\linux\src\osif_umac.c

└ol_tx_ll(), perf_pwr_offload\drivers\host\wlan\txrx\ol_tx.c

└ ol_tx_send(), perf_pwr_offload\drivers\host\wlan\txrx\ol_tx_send.c

└htt_tx_send_std(), perf_pwr_offload\drivers\host\htt\htt_tx.c

└HTCSendDataPkt(), perf_pwr_offload\drivers\host\htc\htc_send.c

HIFSendCompleteCheck(), perf_pwr_offload\drivers\host\hif\pci\hif_pci.c

└CE_per_engine_service(), perf_pwr_offload\drivers\host\hif\pci\copy_engine.c

└HIF_PCI_CE_send_done(), perf_pwr_offload\drivers\host\hif\pci\hif_pci.c

└hif_completion_thread(), hif_pci.c, This thread provides a context in which send/recv completions are handled

└HTCTxCompletionHandler(), perf_pwr_offload\drivers\host\htc\htc_send.c

└SendPacketCompletion(), perf_pwr_offload\drivers\host\htc\htc_send.c

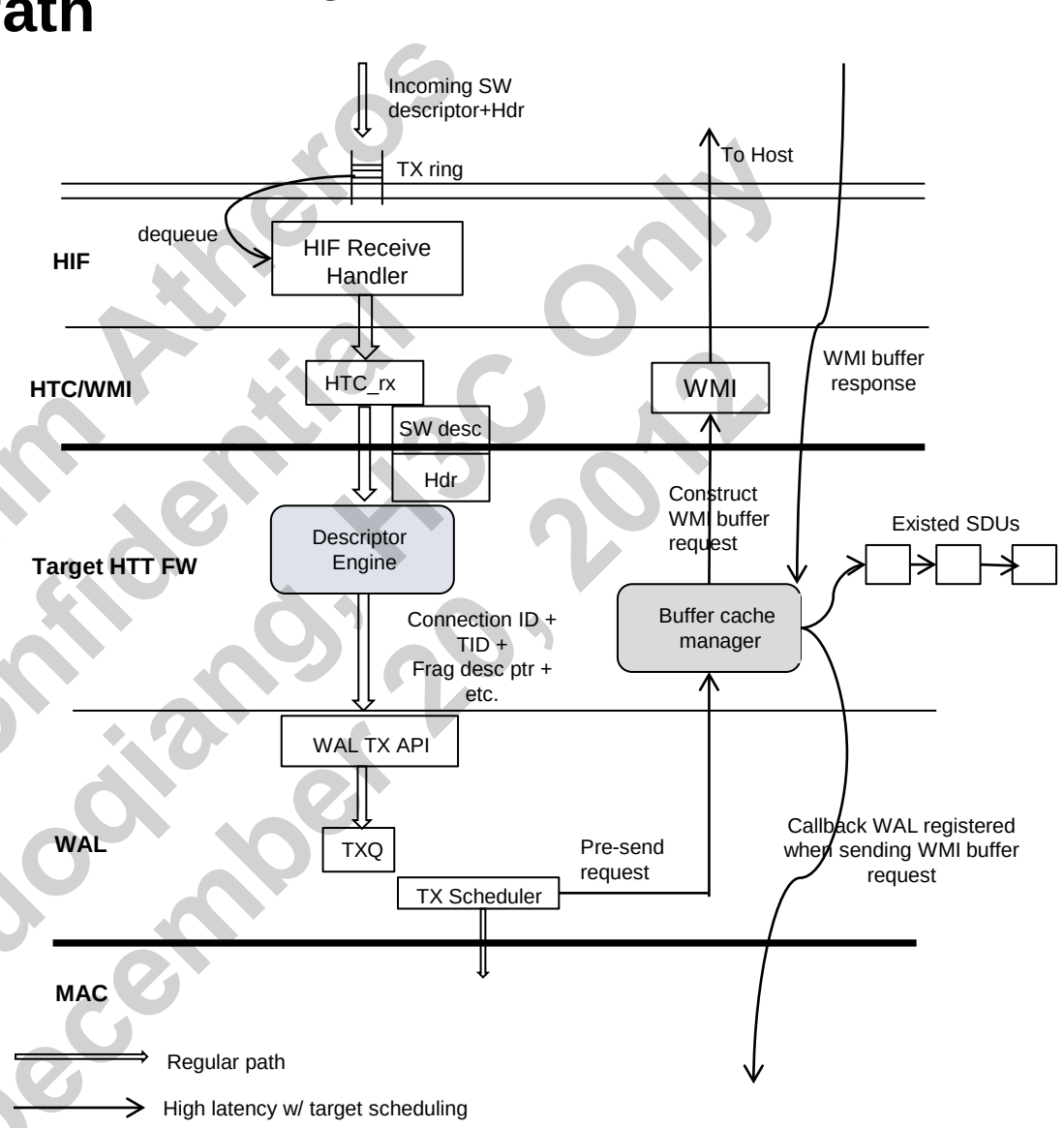
└ DoSendCompletion(), perf_pwr_offload\drivers\host\htc\htc_send.c

└ pEndpoint->EpCallBacks.EpTxComplete();

5.2 Target Tx/Rx Data Path

5.2.1 Tx Data Path

- HTC/WMI/HIF FW
  - receive host → target comm
- HTT FW
  - Invoke DE module to do packet inspection
  - Use WAL API to pass the TX information to transmit
- WAL FW
  - Allocate MSDU descriptor
  - Queue MSDU descriptor to connection queue
  - Perform aggregation and queue PPDU descriptor to MAC HW
- MAC HW
  - Create PPDU to transmit



Queue a frame to the TID queue

check_idle() athos_main.c
└cmnos_intr_handle_pending_dsrs(), cmnos_intrinf.c
  └_CE_per_engine_handler(), copy_engine.c
    └_HIF_CE_recv_cb(), hif_ce.c
      └HifLayerRecvCallback(), htc.c
        └WMIRecvMessageHandler(), wmi_svc.c

_htt_tgt_hif_svc_h2t_input(), tx_sched.c
└_htt_tgt_tx_input (), ar_wal_tx_send.c
  └ar_wal_tx_de_input() = _tx_de_input(), target\src\wlan\wal\ar\include\ar_wal_tx_de.h
  └_tx_de_input(), target\src\wlan\wal\ar\de\ar_wal_tx_de.c, This is the entry function HTT will call to send out a SDU. The function will try to find the peer entry and tidq for this SDU and enque this frame.
    └ar_wal_tx_q_enque() =_tx_q_enque(), target\src\wlan\wal\ar\include\ar_wal_tx_q_enq.h
      └_tx_q_enque_fn(), target\src\wlan\wal\ar\tx\tx_q_enque.c, The entry point for all frames (local/ remote or mgmt or data) in the system. All frames come here with a fragmentation descriptor.
        1) SDU descriptor are allocated, populated and then they are posted to connection tid q.
        2) Based on tid_q status, if it is non-paused, scheduler is tickled on this q.
        3) For a paused connection, once a frame arrives for that tid, this api will enque and trigger the necessary associated callbacks for appropriate notification and potentially, subsequent action.

Tx Complete → shedule another PPDU

check_idle() athos_main.c
└cmnos_intr_handle_pending_dsrs(), cmnos_intrinf.c
└dsr_rv = intr_svc->dsr(intr_svc->dsr_arg);
└_wal_intr_scheduler_intr_handler(), target\src\
wlan\wal\ar\wal_intr_schd\wal_intr_schd.c
└_tx_send_completion_hdlr(), target\src
\wlan\wal\ar\tx\ar_wal_tx_send.c,

_tx_sched_trigger_conn(),
target\src\wlan\wal\ar\tx_sched\tx_sched.c
└tx_sch_evt_proc_tickle(), tx_sched.c
└tx_dispatch_tid(), tx_sched.c
└_tx_send_post_ppdu(), ar_wal_tx_send.c

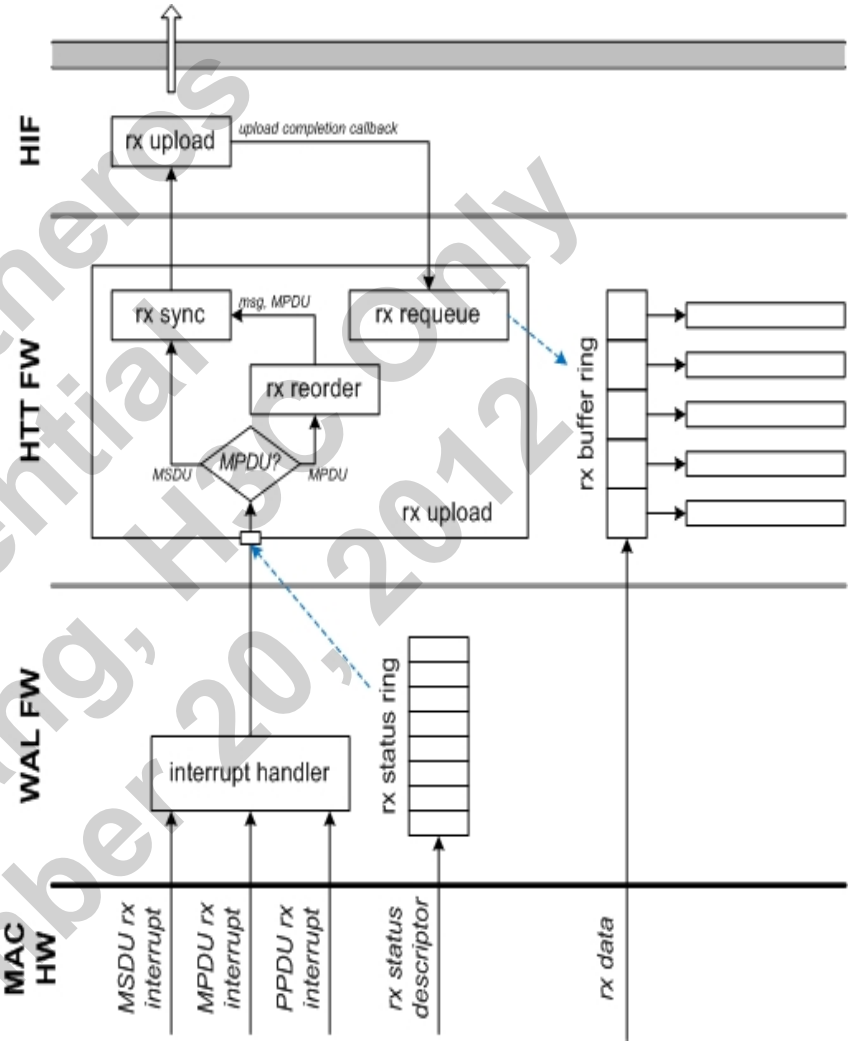
wal_rc_query_rate_info(), ar_wal_rc.
└_RATE_GetTxRetrySchedule(), ratectrl.c
└_GetRateTblAndIndex(), ratectrl.c
└_rcRateFind(), ratectrl.c
└_GetBestRate(), ratectrl.c

└tx_send_queue_to_hw(), ar_wal_tx_send.c
└whalSetTxDP(), target\src\wlan
\wal\include\whal_api.h

whalSetTxDP(), whal_api.h
└whalSetTxDP() = xmitApi.SetTxDP()
└ar6000SetTxDP(), target\src\wlan
\wal\whal\xmit\ar600P_xmit.c

5.2.2 Rx Target Data-path

1. **MAC HW** – receive rx PPDUs from PHY and do MAC processing to break them into MSDUs
2. **WAL FW** – register for rx interrupt, invoke local rx (mgmt) processing, invoke rx HTT processing
3. **HTT FW**:
   - Check if MSDU completes a MPDU. If so…
     Check if the FCS is okay.
     Invoke Rx Reorder.
     Invoke Rx Sync with FCS and Rx Reorder results to generate a message telling the host how to process the MPDU.
   - Upload the Rx Sync message.
     (high-latency only) Upload the MSDU.
4. **HTC/WMI/HIF FW** – send target → host comm



Receive a frame from HW

check_idle() athos_main.c
└cmnos_intr_handle_pending_dsrs(), cmnos_intrinf.c
└_wal_intr_scheduler_intr_handler(), copy_engine.c
└**_rx_completion_hdlr**(), target\src\wlan\wal\ar\rx\ar_wal_rx.c
└rx_process_recv_status(), wal\ar\rx\ar_wal_rx_internal.h
└#define rx_process_recv_status(pdev) RX_INTERNAL _INDIR_FN(process_recv_status ((pdev))),
└**_rx_process_recv_status**(), rx\ar_wal_rx.c
└ieee80211_data_rx_ind()=_ieee80211_data_rx_ind(), target\src\wlan\data_txrx\ieee80211_data_rx_internal.h
└_ieee80211_data_rx_ind(), target\src\wlan\data_txrx\ieee80211_data_rx.c
└#define ieee80211_data_rx_reorder IEEE80211_DATA_RX_INTERNAL_INDIR_FN(reorder) target\src\wlan\data_txrx\ieee80211_data_rx_internal.h
└_ieee80211_data_rx_reorder(), target\src\wlan\data_txrx\ieee80211_data_rx.c
└#define ieee80211_data_rx_flush IEEE80211_DATA_RX_INTERNAL_INDIR_FN(flush), ieee80211_data_rx_internal.h
└_ieee80211_data_rx_flush (), target\src\wlan\data_txrx\ieee80211_data_rx.c
└#define htt_tgt_rx_ind_flush_ie_add HTT_TGT_RX_EVENT_API_INDIR_FN(ind_flush_ie_add)
└_htt_tgt_rx_ind_flush_ie_add(), target\src\hostif\htt\htt_tgt_rx_event.c

```
#define RX_INTERNAL_INDIR_FN(fn) _rx_##fn

RX_INTERNAL_FN
rx_internal_indir_tbl = {
_rx_ring_fill_bufchain,
_rx_process_recv_status,
_rx_completion_hdlr,
_rx_init_status_ring,
_rx_save_oversize_amsdu,
_rx_patch_oversize_amsdu,
_rx_get_ppdu_rate_and_phy
_mode };
```

Thank you!

Qualcomm Atheros Confidential and Proprietary