# RW-WLAN-nX LMAC SW

Functional Specifications

RW-WLAN-nX-LMAC-SW-FS/1.07

Version 1.07

2018-08-03

RivieraWaves
http://www.rivierawaves.com/

# Revision History

| Version | Date | Revision Description | Author |
|---|---|---|---|
| 0.01 | 2012-07-12 | Initial Release | Steven L'Her |
| 0.02 | 2012-07-27 | Added RX path description | Steven L'Her |
| 0.03 | 2012-09-17 | Added A-MPDU formatting and TX buffer management | Steven L'Her |
| 1.0 | 2012-12-21 | Minor edits | Steven L'Her |
| 1.01 | 2013-10-18 | Updates of RX path | Steven L'Her |
| 1.02 | 2015-09-23 | Added P2P, TD, Channel Context, PS blocks description | Laurent Trarieux |
| 1.03 | 2015-09-24 | Added error detection and recovery mechanism description | Steven L'Her |
| 1.04 | 2016-04-21 | Added chapter for the SW profiling signals description | Steven L'Her |
| 1.05 | 2016-09-21 | Updated TX path description<br>Updated MM description | Steven L'Her |
| 1.06 | 2016-09-28 | Added A-MSDU transmission | Steven L'Her |
| 1.07 | 2018-08-03 | Updated Channel Context | Cédric Izoard |

Changes between a version and the previous one is reflected by the addition of **change bars**, like for the line below:

| TBD | Date entered | Description | Status |
|---|---|---|---|

**Items to be determined in the future versions of this document**

# Table of Contents

# List of Figures

# List of Tables

# 1 Overview

## 1.1 Document overview

### 1.1.1 Purpose

The document deals with the high level overview of the RW-WLAN-nX Lower MAC SW (Hence forth referred as LMAC SW). The purpose of the document is to give high level design of the LMAC SW to MAC development and verification engineers.

### 1.1.2 Scope

The scope of this document is to provide a high level description of the LMAC SW blocks and the functionality handled by each of the blocks. The exact "C"/assembly language functions/sub-routines, local variables etc. are not defined here.

### 1.1.3 Overview

This document describes the design in two levels in the ascending order of detail provided.

Level 1: *LMAC SW design overview*

Modules of LMAC-SW and their interfaces are defined at a high level. The operation of LMAC-SW is illustrated by explaining the interaction of these modules.

Level 2: *Functional blocks*

Sub-modules/functional blocks of the main modules defined in *LMAC SW design overview* are identified. A description of the operation of each functional block is given.

The design of each functional block is illustrated by explaining state machines, logic/algorithm at a higher level

High level data structures created/used by functional blocks are specified. Messages and message flow sequences between functional blocks are illustrated.

### 1.1.4 Abbreviations and Acronyms

| | |
|---|---|
| AC | Access Category |
| ACK | Acknowledgment |
| ACM | Access Category Mandatory |
| AID | Association Identifier |
| AP | Access Point |
| APSD | Automatic Power Save Delivery |
| A-MPDU | Aggregate MAC Protocol Data Unit |
| A-MSDU | Aggregate MAC Service Data Unit |
| BA | Block Acknowledgement |
| BAR | Block Acknowledgement Request |
| BSSID | Basic Service Set Identifier |
| CDE | Channel Distribution Event |

| | |
|---|---|
| CF | Contention Free |
| CSI | Carrier State Information |
| CTS | Clear To Send |
| CW | Contention Window |
| DCF | Distributed Coordination Function |
| EDCA | Enhanced Distributed Channel Access |
| FIFO | First-In-First-Out |
| FC | Frame Control |
| FCS | Frame Check Sequence |
| HCCA | HCF Controlled Channel Access |
| HT | High Throughput |
| IBSS | Independent Basic Service Set |
| ICV | Integrity Check Value |
| IV | Initialization Vector |
| LLC | Logical Link Control |
| L-SIG | Legacy (Non-HT) Signal Field |
| MAC | Medium Access Control |
| MEM-BAR | Memory Base Address Register |
| MIB | Management Information Base |
| MIMO | Multiple Input Multiple Output |
| MLME | MAC Sub Layer Management Entity |
| MMPDU | MAC Management Protocol Data Unit |
| MPDU | MAC Protocol Data Unit |
| MSDU | MAC service data unit |
| NAV | Network Allocation Vector |
| NOA | Notice of Absence |
| OppPS | Opportunistic PS |
| OS | Operating System |
| P2P | Peer to Peer |
| PC | Point Coordinator |
| PHY | Physical layer |
| PLME | PHY Sub Layer Management Entity |
| PS | Power Save |
| PSMP | Power Save Multi-Poll |
| QAP | QoS Access Point |
| QoS | Quality of Service |

| | |
|---|---|
| QSTA | QoS Station |
| RD | Reverse Direction |
| RDG | Reverse Direction Grant |
| ROC | Remain on Channel |
| RSNA | Robust Security Network Association |
| RTS | Request to Send |
| RX | Receiver |
| SSID | Service Set Identifier |
| STA | Station |
| STBC | Space-Time Block Coding |
| TBTT | Target Beacon Transmission Time |
| TID | Traffic Identifier |
| TIM | Traffic Indication Map |
| TS | Traffic Stream |
| TU | Time Unit |
| TX | Transmitter |
| TX_REQ | Transmit Request message received from UMAC-SW |
| U-APSD | Unscheduled Automatic Power Save Delivery |
| WEP | Wired Equivalent Privacy |
| WLAN | Wireless LAN |
| WM | Wireless Medium |
| WMM | Wi-Fi Multimedia |

**Table 1: Abbreviations and Acronyms**

# 2 High level design overview

## 2.1 Design objectives

The LMAC SW design objectives are to:

Perform the time critical functionalities that comply with IEEE 802.11 protocol standards for the STA, AP and IBSS operating modes.

Provide the interfaces to UMAC SW and MAC HW parts of MAC component.

Provide the interfaces to the PHY component.

## 2.2 Relationship to external environment

Refer to [3] for the description of the interface between the UMAC SW and the LMAC SW.

Refer to *[4]* for the description of the interface between the LMAC SW and the MAC HW.

Refer to [5] for the description of the interface between the LMAC SW and the platform SW.

## 2.3 LMAC SW design overview

Figure 1 below shows the high level blocks of the LMAC SW as well as the platform dependent SW layers the LMAC is interfacing with.

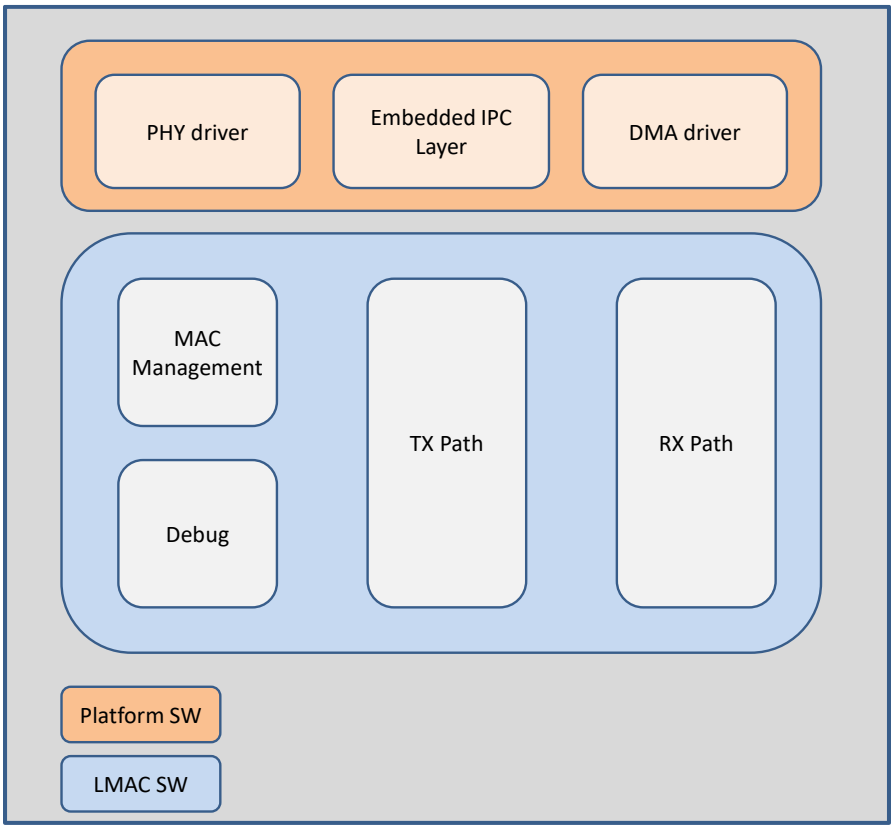



**Figure 1: LMAC SW and platform dependent SW overview**

The figure above applies to the usual partitioning of the LMAC, i.e. when it is running on a dedicated CPU while the upper MAC is running on the application CPU. Other partitioning (e.g. LMAC and upper MAC running on the application CPU) might be possible.

### 2.3.1 Embedded IPC Layer

This layer facilitates the communication between the application system and the embedded system along with its peer component namely, Application IPC layer in the application system. The unit of communication used is IPC message descriptor. Each of these message descriptors is associated with a payload. The payload contains the parameters required to handle the message either by UMAC SW or LMAC SW in the embedded system depending on MAC SW partitioning mode or by the layers above the Application IPC Layer in the application system. The Application IPC Layer and the Embedded IPC Layer maintain FIFOs to hold message descriptors. The messages are categorized as:

- ✓ Data transmission
- ✓ Data reception
- ✓ Control or debug.
- ✓ The IPC message descriptors are different for type of message category. Refer to [5] for more details.

### 2.3.2 LMAC SW

The LMAC SW is responsible for the MAC HW configuration and the chaining of descriptors for the transmission and reception. The LMAC SW modules are grouped into four blocks.

- ✓ MAC Management Block
- ✓ TX Path Block
- ✓ RX Path Block
- ✓ Debug Block.

#### 2.3.2.1 MAC Management Block

This block is mainly responsible for the static MAC HW and PHY configuration, as well as the management of the overall system state:

- ✓ Management of the virtual interfaces
- ✓ Configuration of the MAC HW static parameters (i.e. MAC address, Basic Rates, EDCA parameters, etc.)
- ✓ Management of the peer STA database (STA adding/removal, Key Adding/Removal, BlockAck agreement)

#### 2.3.2.2 TX Path Block

This block is responsible for the transmission of the packets indicated via the IPC layer. The following operations are performed by this block:

- ✓ Creation of the A-MPDUs
- ✓ Allocation/Deallocation of the Data Buffers and HW descriptors per Access Category
- ✓ Payload download from Upper MAC memory prior to transmission (via the DMA driver)
- ✓ Chaining of the descriptors to the MAC HW
- ✓ Processing of the received BA after A-MPDU transmission
- ✓ Generation of the TX confirmations to the upper MAC SW

#### 2.3.2.3 RX Path Block

This block is responsible for the reception of the packets from the MAC HW and of their indication to the Upper MAC via the IPC layer. The following operations are performed by this block:

- ✓ Management of the MAC HW RX descriptors lists
- ✓ Payload upload to the Upper MAC memory (via the DMA driver)
- ✓ Frame indication to the Upper MAC

#### 2.3.2.4 Debug Block

The debug block provides an API for managing the debugging functionalities of the LMAC:

- ✓ Accessing the LMAC memory (Read/Write)
- ✓ Enabling/Disabling the traces

## 2.4 Functional blocks

### 2.4.1 Introduction

This section describes the functionality of individual modules of LMAC SW blocks. The Figure 2 shows the LMAC SW modules.

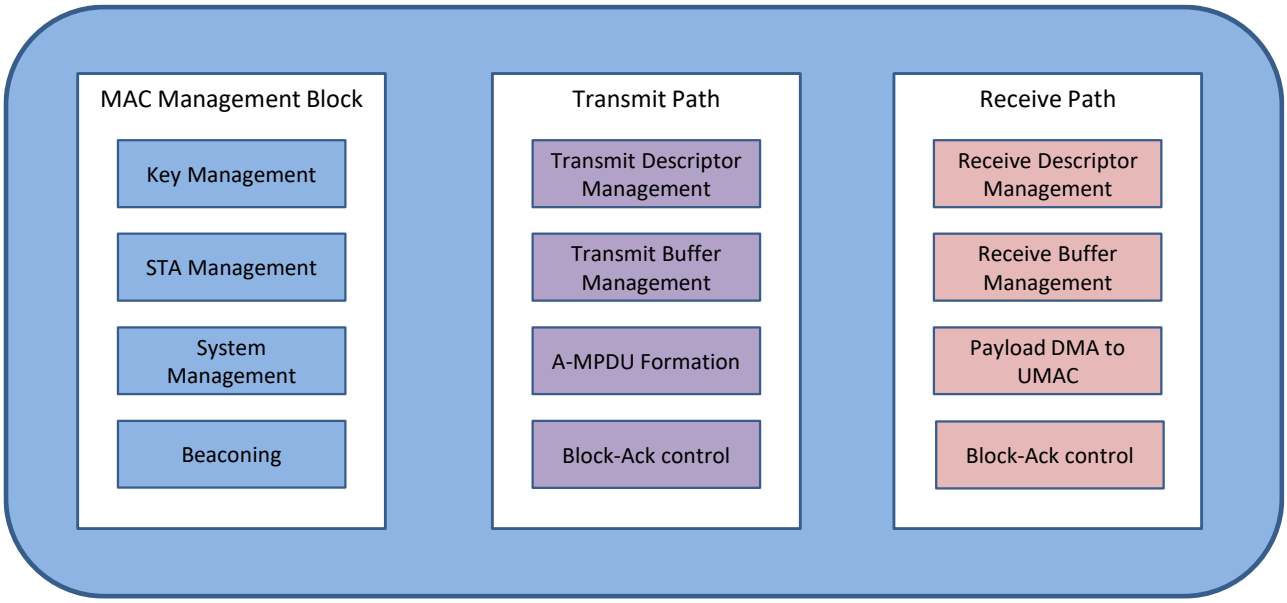



**Figure 2: Functional block diagram of LMAC SW**

The following sections give a detailed description of the constituent modules of each of these blocks.

### 2.4.2 MAC Management Block

This block is responsible to handle the configuration and control of the LMAC SW and the MAC HW. It is implemented in the form of a kernel task receiving and sending control messages. The MM is the main SW block interacting with the UMAC for the configuration of the system.

The MM is also responsible of various procedures offloading the operation of the Upper MAC, such as the STA connection monitoring, the synchronization with the AP, and the automatic transmission of the beacons when in AP mode.

#### 2.4.2.1 MM Interface to Upper Layers

As a kernel task the MAC Management block has an API composed of kernel messages received and sent. The API of the block can be found in [3]. The type of messages that can be sent/received is shown in the figure below:

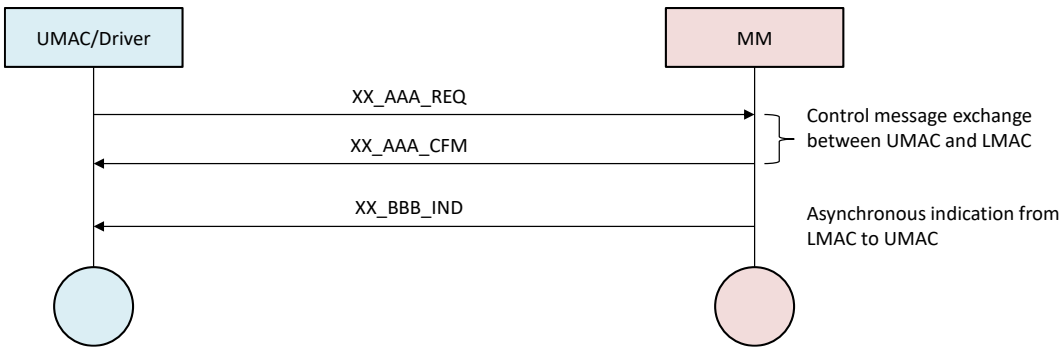



**Figure 3: MM Interaction with Upper Layers**

#### 2.4.2.2 MM States

The MM task has four possible states: MM_IDLE, MM_ACTIVE, MM_GOING_TO_IDLE and MM_HOST_BYPASSED.

The first two states match the possible HW states: IDLE, ACTIVE. In IDLE the HW is clocked with fast clocks but it is able neither to receive nor transmit. In ACTIVE state the HW is able to receive or transmit. The DOZE state of the HW is not reflected in the MM because when the system is in the DOZE state the FW is not running and therefore cannot receive any request.

When a configuration request coming from Upper Layers requires changing the HW to IDLE state, the IDLE request is sent to the HW and the received message is saved. Once the HW indicates that its state has changed to IDLE (via an interrupt), the message is restored and handled by the MM task. The MM_GOING_TO_IDLE state is used for this purpose.

The MM_HOST_BYPASSED is a state in which the MM_SET_IDLE_REQ request allowing the Upper Layers to control state of the HW is bypassed, because some internal procedures (e.g. channel context scheduling) require a full control of the HW state.

##### 2.4.2.2.1 Case of a MM request not requiring HW in IDLE state

The figure shows the reception of a request that is not requiring the HW to be IDLE state. The MM will in this case handle the request immediately:

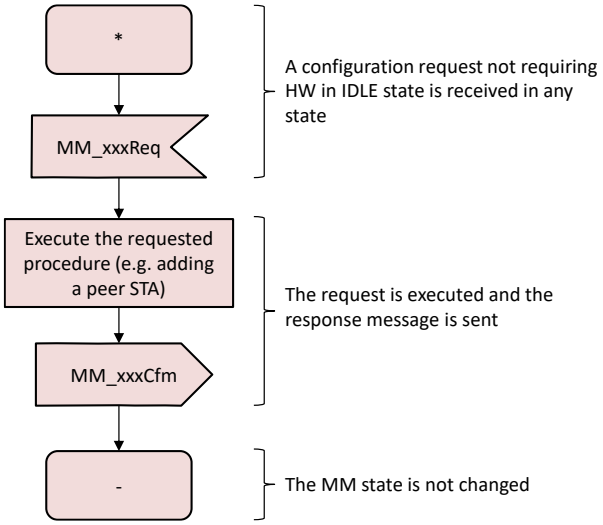



**Figure 4: MM Request not requiring HW in IDLE state**

#### 2.4.2.2.2 Case of a MM request requiring HW in IDLE state

In case such a request is received, the behavior of the MM is different depending of the HW state is. The figure below shows the behavior of the MM when the system is already in IDLE state:

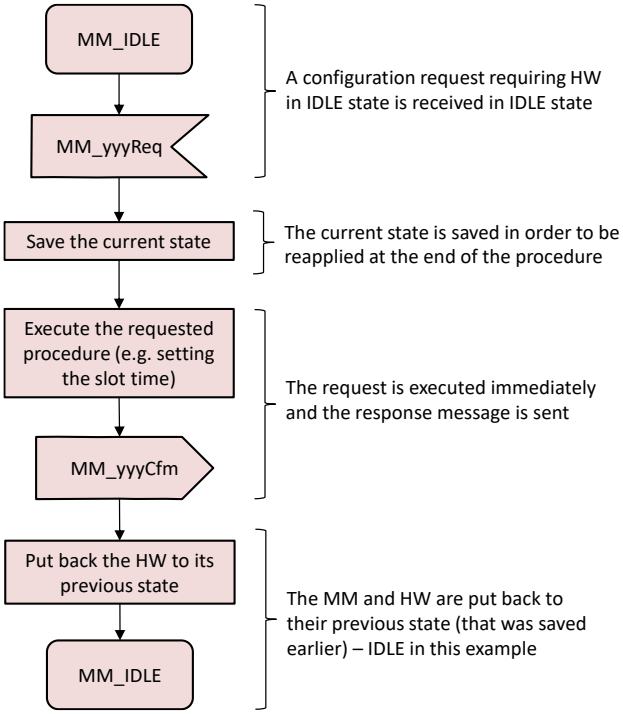



**Figure 5: Procedure requiring IDLE state - HW already IDLE**

In case the system is in ACTIVE state the procedure cannot be handled immediately, because the HW needs to be first put to IDLE state. As the transition to IDLE may not be immediate (e.g. in case the HW is currently receiving or transmitting a packet), the MM needs to be able to request the transition to IDLE and then waits for the indication from the HW before proceeding to the request.

In that case the MM first requests the HW to move to IDLE and saves the request by making use of the message saving feature of the kernel (see [6] for details about this feature):

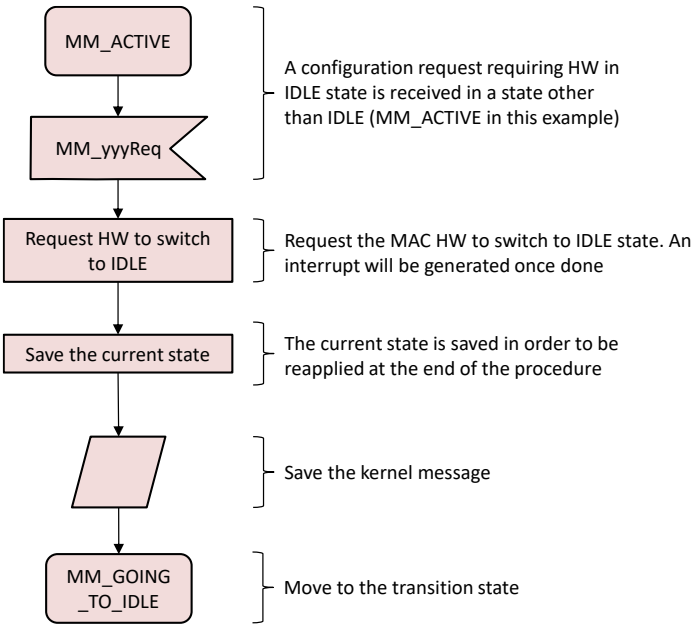



**Figure 6: MM requesting HW to move to IDLE**

The HW will then indicate the transition to IDLE using an interrupt. Upon this interrupt a corresponding kernel event is triggered in order to handle the event in background context:

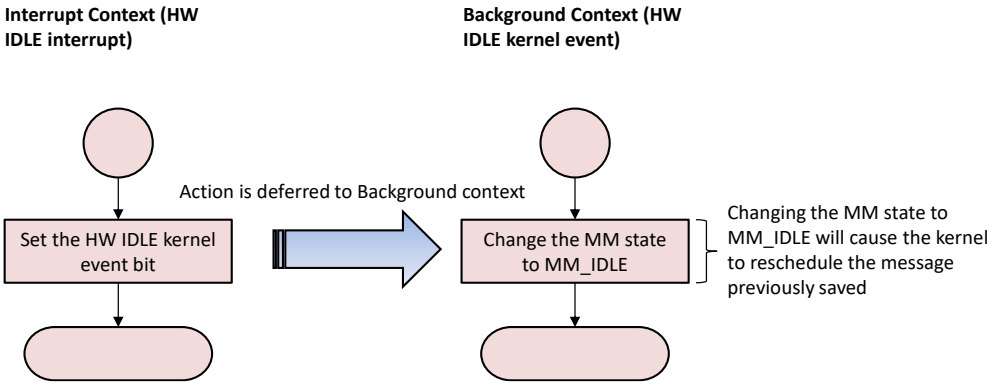



**Figure 7: HW IDLE interrupt handling**

The MM state change (from MM_GOING_TO_IDLE to MM_IDLE) will cause the kernel to reschedule the message that was saved. The message can now be handled because the MM is in IDLE state:

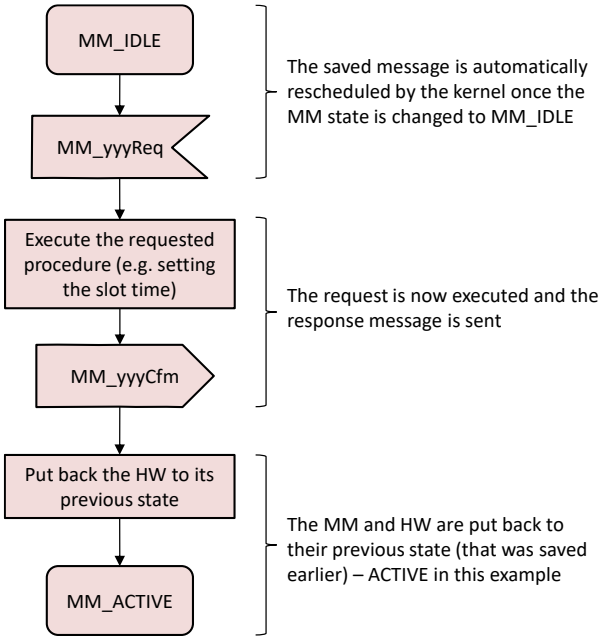



**Figure 8: Message handling and back to ACTIVE**

#### 2.4.2.3 Connection Monitoring

While acting as a STA connected to an AP, the MM is able to do the monitoring of the connection, i.e. it ensures that the AP is still up and running and sends periodically a keep-alive frame. It is also not waking up the Upper Layers upon all the beacon receptions, thus reducing the system power consumption.

The connection monitoring module is also checking the RSSI of received beacons in order to detect if we will go out of range.

This feature can be disabled at compile time by putting the CMON option to off. In that case the UMAC is responsible to perform the connection monitoring. Note that in FullMAC partitioning the feature is always included.

##### 2.4.2.3.1 Beacon Reception monitoring

One of the goals of the connection monitoring feature is to ensure that the AP is still up and running and in a range allowing operation. For this the MM is checking that the beacons are received. If this is not the case, the MM will try sending a NULL frame to check if the AP is still present or not.

The beacon reception monitoring is started by issuing the MM_SET_VIF_STATE_REQ message to the MM indicating that the operation is now started. The MM programs a “fake” TBTT event (because it has currently no knowledge on when the next beacon arrives) and then waits for a first beacon.

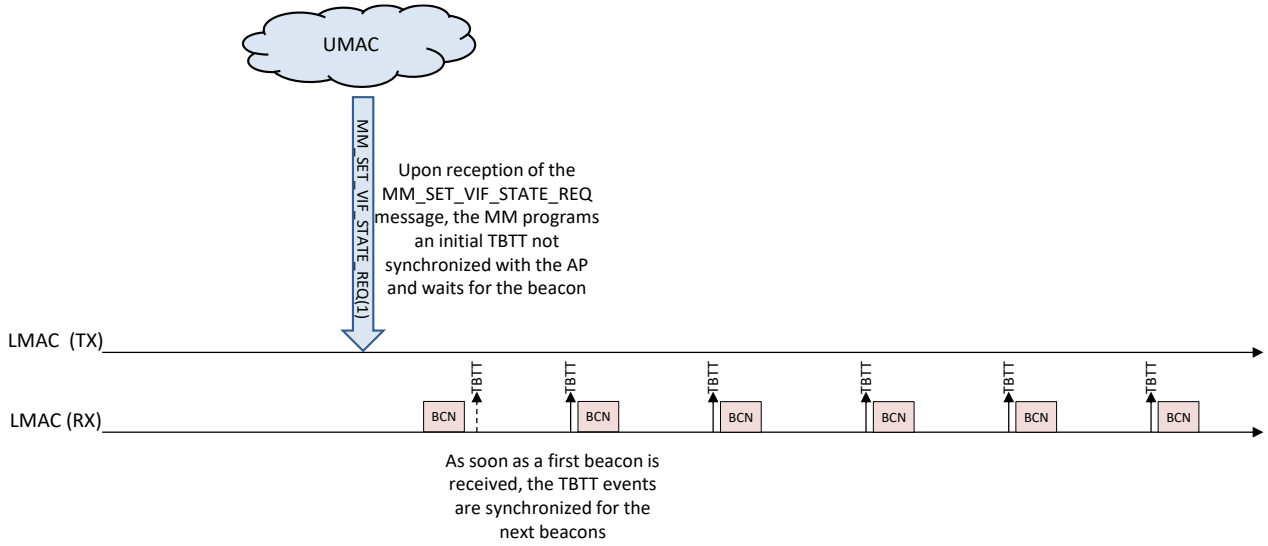



**Figure 9: MM - Starting the connection monitoring**

The figure below shows a case where for some reason the beacons are not received for some time. After 10 consecutive TBTTs not followed by a beacon reception, the MM programs the transmission of a NULL frame to check if the AP is still operating or in the range. AP replies with an Ack. In such case the connection is not considered as lost and the operation continues.

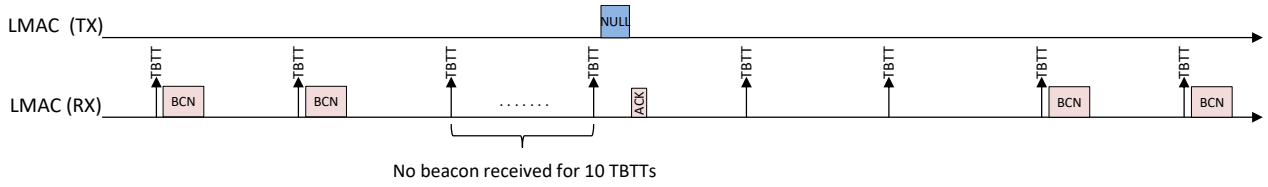



**Figure 10: MM - Temporary beacon reception failures**

In case the NULL frame is not acknowledged, then the MM sends a MM_CONNECTION_LOSS_IND message to the UMAC, but it does not stop scheduling the TBTT and monitoring the beacons. It will stop this operation upon the reception of a MM_SET_VIF_STATE_REQ requesting to deactivate the VIF operation.

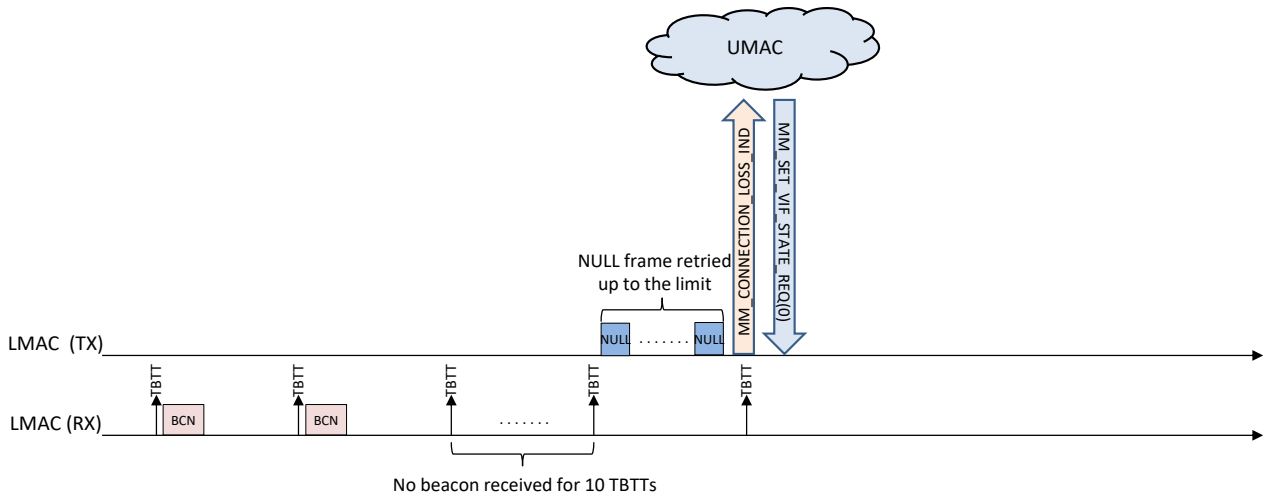



**Figure 11: MM - Connection loss detection**

#### 2.4.2.3.2 Beacon Reception offload

When a beacon from the AP is received, the MM checks if it is required to forward it to the Upper Layers. To do this the MM computes a CRC over the fields of the beacon that are not handled internally to the LMAC (e.g. timestamp, TIM). If the computed CRC is the same as previous beacon received, then the beacon is not forwarded to the Upper Layers, otherwise it is forwarded.

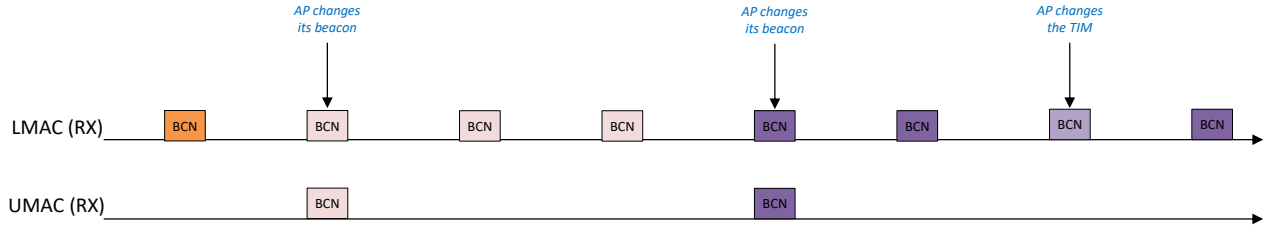



**Figure 12: Connection Monitoring - Beacon reception offload**

#### 2.4.2.3.3 Keep-alive Frame transmission

When the connection monitoring is used, the MM will ensure that the connection is maintained at the AP side by periodically sending a keep-alive frame to the AP. The periodicity of transmission is around 30s.

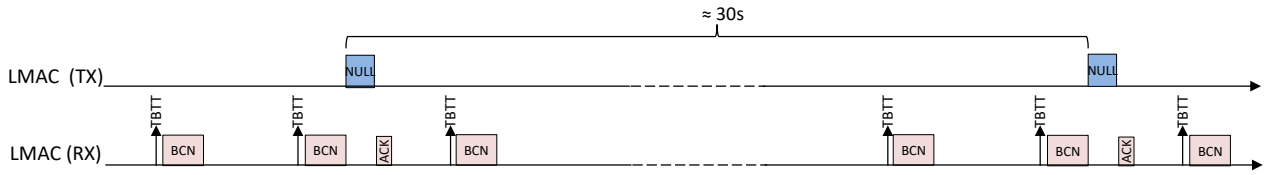



**Figure 13: MM - Keep-alive frame transmission**

#### 2.4.2.3.4 RSSI monitoring

After the host has programmed some thresholds using the MM_CFG_RSSI_REQ message, the MM starts monitoring the RSSI of the received beacons. When the RSSI is getting lower or greater than the programmed threshold (taking into account the hysteresis value), the MM_RSSI_STATUS_IND message is sent to the UMAC to indicate the event.

This information may then be used by the UMAC/Supplicant to help for the roaming decisions.

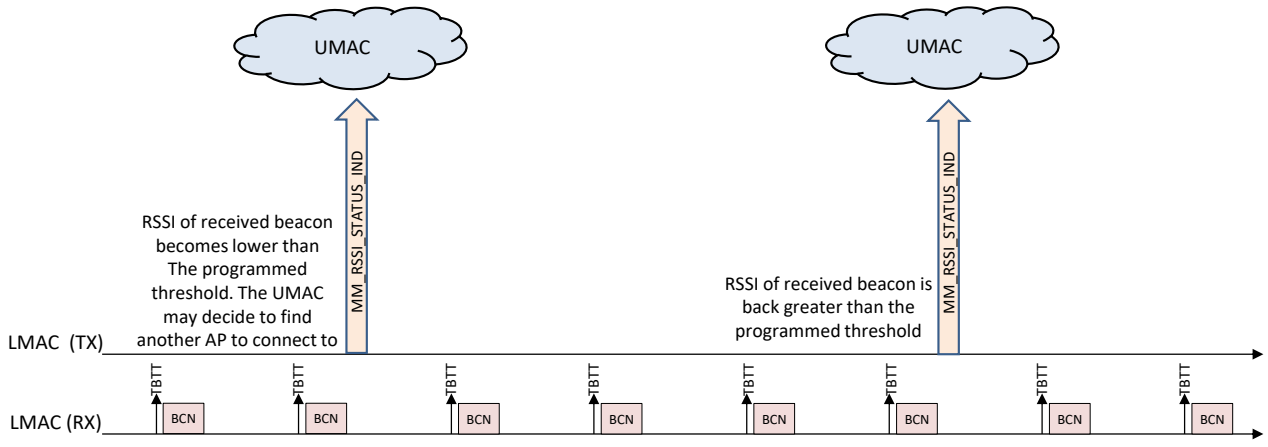



**Figure 14: MM - RSSI monitoring**

#### 2.4.2.4 Autonomous Beacon Transmission

While acting as an AP, the MM is able to automatically transmit the beacons at TBTT. It allows not waking up the Upper Layers for all the beacon transmissions, thus reducing the system power consumption.

This feature can be disabled at compile time by putting the AUTOBCN option to off. In that case the UMAC is responsible to chain the next beacon to transmit each time a TBTT event is asserted to the Upper Layers. Note that in FullMAC partitioning the feature is always included.

The MM considers the beacon as several parts that can be updated independently:

1. The complete beacon frame except the TIM IE using the MM_BCN_CHANGE_REQ message
2. The TIM IE using the MM_TIM_UPDATE_REQ

The MM therefore keeps two independent buffers in the shared memory for each VAP:

1. The beacon template buffer
2. The traffic-indication virtual bitmap that is used to build the TIM partial virtual bitmap

A small additional buffer is used to contain the first bytes of the TIM IE.

The format of the beacon template buffer is the following:

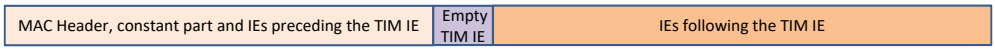



**Figure 15: Beacon template for autonomous beacon transmission**

The traffic-indication virtual bitmap is composed of a 256 byte buffer, in which 2008 bits are valid:

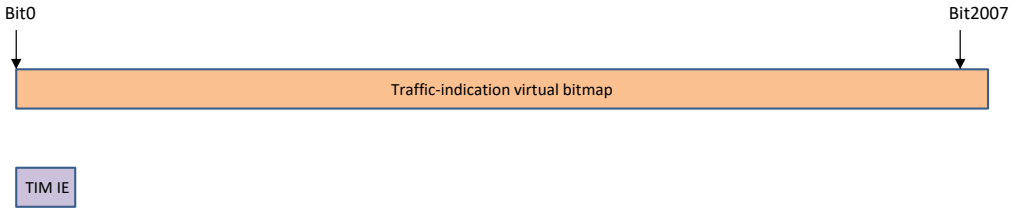



**Figure 16: Traffic-indication virtual bitmap and TIM IE**

For the transmission the beacon is split in four parts:

1. The MAC Header, constant parameters, and IEs before the TIM IE
2. The TIM IE
3. The IEs following the TIM IE

In order to allow this the MM uses 4 different HW descriptors linked together: 1 THD for the frame description and the first part of the beacon, 2 TBDs for the TIM IE (1 for the first bytes of the IE, and a second one for the partial virtual bitmap) and a last TBD for the rest of the beacon. This split allows updating independently the TIM IE or the other IEs without copying any data. The figure below shows the HW descriptors used for the transmission and the data buffers where they are pointing to:

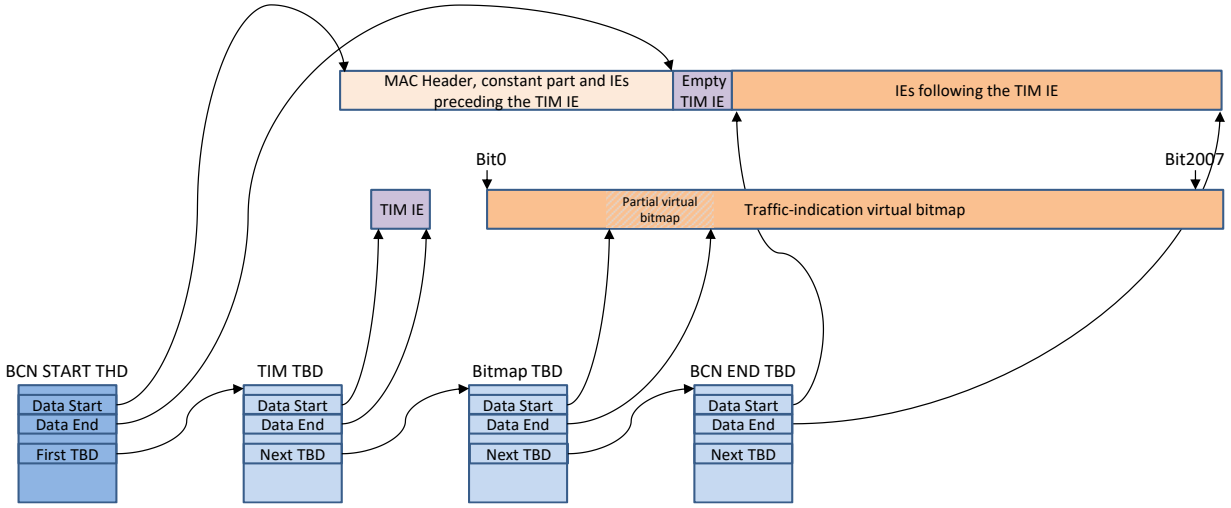



**Figure 17: Chaining of the beacon to the HW**

Additionally to the basic beacon portions shown above, some specific features of the FW add their own part to the beacon, using additional buffers and TBDs. Such features are the P2P GO (to add the NoA IE) as well as the Mesh (for Mesh PS).

#### 2.4.2.4.1 Updating the beacon content

In order to update the beacon content except the TIM IE the Upper Layers make use of the MM_BCN_CHANGE_REQ message. Upon the reception of this message the MM downloads the content of the beacon from the host memory to the shared memory, and then prepares the TX HW descriptors that will be used to chain the different parts of the beacon.

The figure below shows the initial beacon update operation followed by a second one:

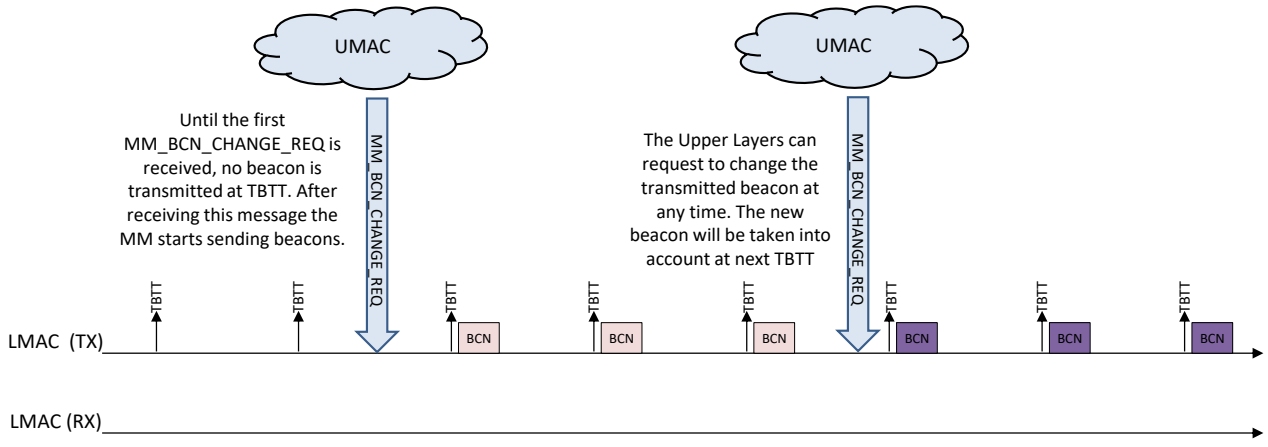



**Figure 18: MM - Updating the beacon content**

#### 2.4.2.4.2 Updating the TIM IE

The MM is maintaining in shared memory the traffic-indication virtual bitmap for each running VAP. The bits inside this bitmap are set/reset upon the reception of the MM_TIM_UPDATE_REQ message. The partial virtual bitmap transmitted inside the TIM Information Element is obtained by updating the TBD attached to the virtual bitmap to point to the correct start and end bytes to transmit.

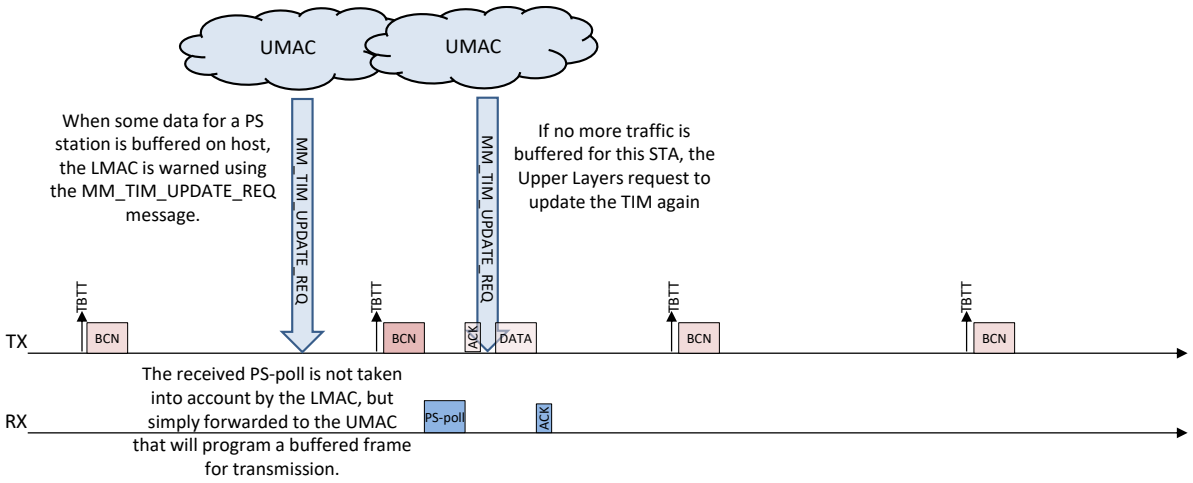



**Figure 19: Updating the TIM IE and delivering buffered traffic**

#### 2.4.2.5 STA Management

This module manages the adding/removal of peer STAs to the LMAC SW. The STA info table contains information such as aggregation parameters of the peer STAs.

#### 2.4.2.6 Key Management

This module manages the adding/removal of security keys to the LMAC SW.

#### 2.4.2.7 System Management

This module manages the LMAC SW system states and parameters. It is responsible for:

- ✓ Initializing the full system (TX and RX paths, MAC HW, PHY)
- ✓ Managing the different states of the MAC HW
- ✓ Adding virtual interfaces
- ✓ Configuring the system parameters (PHY channel, RX filters, QoS parameters, BSSID, etc.)

### 2.4.3 Transmit Path

The Transmit Path is responsible for transmitting the frames using the MAC HW. The information needed for transmission to MAC HW is communicated using the Transmit Header Descriptor and the Policy Table. MAC HW registers are also used to control transmission properties.

The TX Path has been designed to minimize the buffer memory used during a packet transmission. In order to allow this, the payload data is downloaded from the Upper MAC memory in the last steps of the transmission

The frame transmission is handled in the following phases:

- ✓ Get the transmit descriptors from the upper MAC, classified on an Access Category (AC) basis
- ✓ Form A-MPDUs when applicable
- ✓ Allocate the transmission buffers and download the payloads for the 4 Access Categories and the broadcast/multicast queue
- ✓ Update certain MAC Header and Transmit Header Descriptor fields
- ✓ Queue Transmit Header Descriptor to MAC HW for transmission
- ✓ Confirm the transmitted packets to the upper MAC

The figure below gives an overview of the major steps of a transmission.

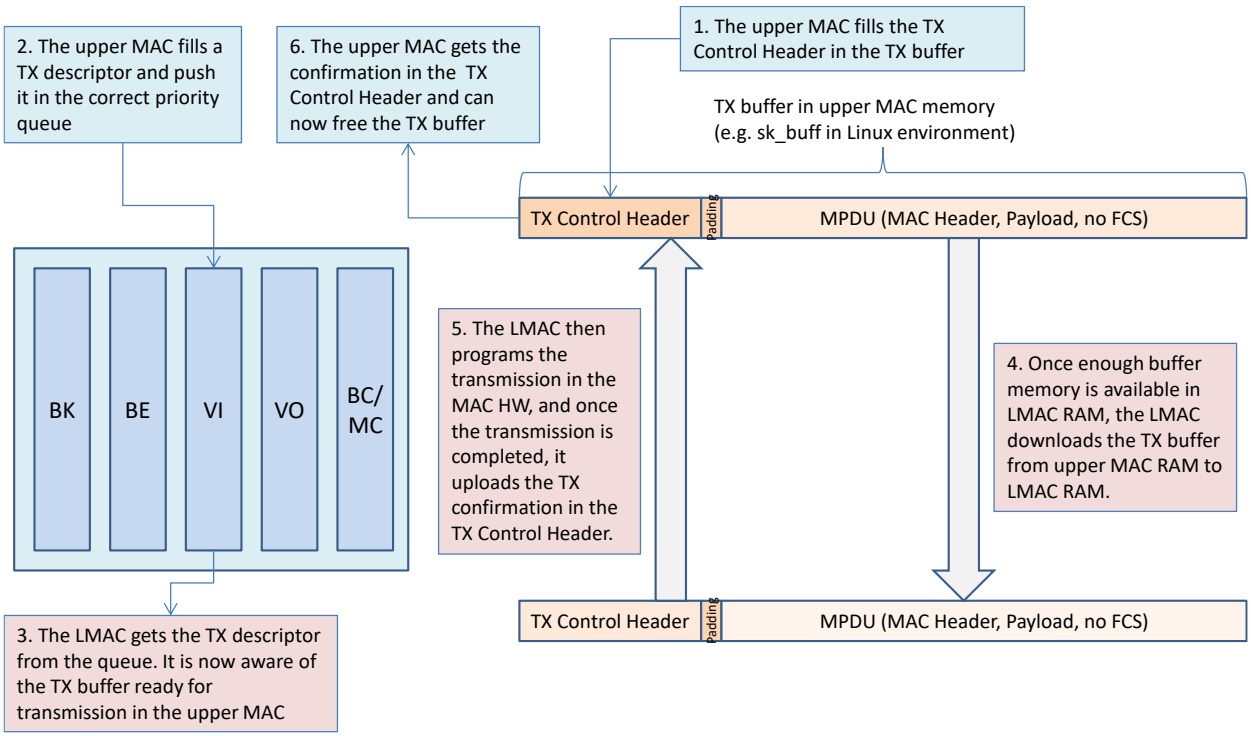



**Figure 20: Overview of a transmission**

#### 2.4.3.1 TX Data Structures

The TX path manipulates various data structures that are used at the different steps of the transmission. The following sections describe these structures and how they are used.

#### 2.4.3.1.1 TX Descriptor

The TX Descriptor is the data structure describing an MPDU to be transmitted. It is pushed to the LMAC by the Upper MAC, and it is then used by the LMAC for the whole duration of the transmission. It contains the following information:

- ✓ Information coming from the Upper MAC
  - ➢ The physical address of the MPDU in Upper MAC memory
  - ➢ The length of the MPDU
  - ➢ The destination station index
  - ➢ The priority
  - ➢ The MAC and PHY transmission flags (e.g. if MPDU has to be aggregated, at which rate it has to be transmitted, etc.)
  - ➢ The sequence number

- ✓ Information used locally
  - ➢ Pointer to the buffer allocated for the transmission
  - ➢ Pointer to the A-MPDU descriptor
  - ➢ Pointer to the TX Confirmation descriptor

The TX Descriptors are used by the LMAC SW only, and no part of it is handled by the MAC HW. It can therefore be placed in a cacheable memory area in order to achieve better performance.

In case A-MSDU transmission feature is compiled, the physical address and length fields are replaced by 2 arrays of physical addresses and lengths, allowing to attach several MSDUs to a single MPDU.

#### 2.4.3.1.2 TX Descriptor Lists

As mentioned above, the TX Descriptors are used during all the steps of the transmission. Between each steps of the transmission, the descriptors are pushed into a specific list. These lists are instantiated per TX queue, i.e. the 4 Access Categories plus the broadcast/multicast queue when applicable.

The figure below shows the different lists used for each step of the transmission:

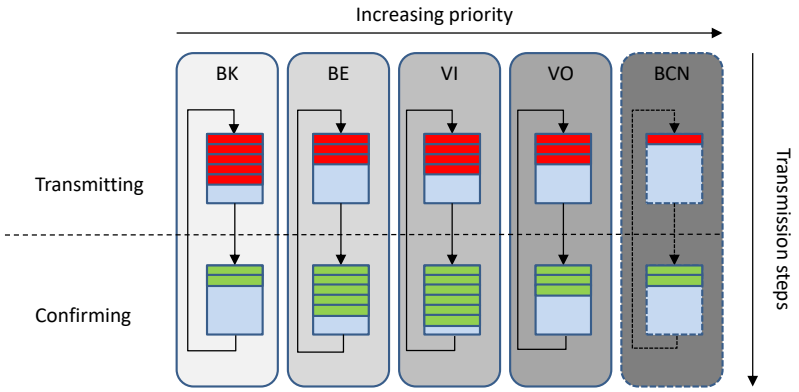



**Figure 21: Transmission lists**

#### 2.4.3.1.3 TX Buffer Lists

For each Access Category a chained list of buffers is defined. This list contains the TX buffers that have been programmed for download, and for which a specific action is required once downloaded.

#### 2.4.3.1.4 A-MPDU Descriptor

The A-MPDU Descriptor is the data structure describing an A-MPDU to be transmitted. It is used at the different stages of an A-MPDU transmission, and contains the following information:

- ✓ The HW descriptors used specifically for the A-MPDU transmission
  - ➢ A-MPDU Transmit Header Descriptor
  - ➢ BAR frame descriptors
  - ➢ Policy table
- ✓ The pointer to the RX Descriptor of the BlockAck frame
- ✓ The amount of payload data currently downloaded for this A-MPDU
- ✓ The index of the station to which the A-MPDU is transmitted
- ✓ The priority
- ✓ The status flags indicating if the A-MPDU is fully built, if it is waiting for the BlockAck reception, if the BlockAck frame has been received or not, etc.

#### 2.4.3.1.5 TX Confirmation Descriptor

The TX Confirmation Descriptor is the data structure used for uploading the transmission status to the TX buffer located in Upper MAC memory. It contains the following information:

- ✓ The platform DMA descriptor that will be used to upload the status
- ✓ The status field (defined as a 32-bit word), that is uploaded to the Status field of the TX Control Header in the upper MAC TX buffer

In order to optimize the management of the TX Confirmation descriptor pool, each of these descriptors is associated on a 1:1 basis with a TX Descriptor. The link between the two descriptors is done at initialization time.

#### 2.4.3.1.6 TX Buffer

The TX Buffer is the data structure that contains all the MPDU data, as well as all the HW descriptors that will be used for the transmission of the MPDU. This buffer is allocated from a per-AC memory pool once there is some space available inside it, filled by programming a DMA transfer from Upper MAC memory buffer, and freed as soon as the transmission has been performed by the HW.

The format of the TX buffer is as shown in the figure below:

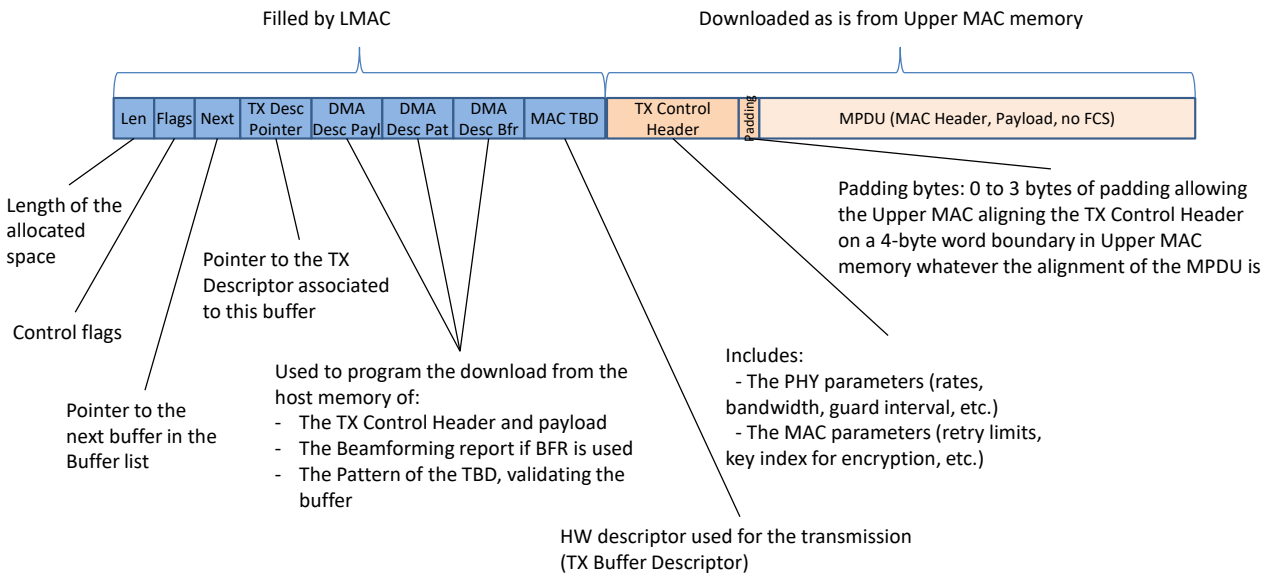



**Figure 22: TX buffer format**

#### 2.4.3.2 Transmission Main Steps

The main transmission steps are described in the following sections.

##### 2.4.3.2.1 Transmit IPC Event

When the Host has to request a transmission, it first fills a TX Descriptor that is then passed to the LMAC (or the UMAC in case of FullMAC). The way the descriptor is passed from Host to FW is outside the scope of this specification (as it depends on the communication interface used between Host and FW, i.e. PCIE, USB, SDIO, etc.). An example of such mechanism is described in [5]. The IPC embedded layer, which is responsible for the communication between the Host and the MAC FW, will call the LMAC or UMAC TX frame push function with a pointer to the descriptor as parameter.

This handling is performed in a background kernel event called the TX IPC Event.

###### 2.4.3.2.1.1 TX Descriptor Initial Handling

Once the LMAC receives a TX Descriptor, it first calls the A-MPDU formatting algorithm, pushes the descriptor in the *TX Transmitting* list of the corresponding access category, and then checks if some free space is currently available in the TX Buffer pool of this access category. If yes, then it allocates the TX buffers that will be used for the pushed packets and program the payload transfer from Upper MAC memory to the TX Buffer using the platform SW DMA.

The figure below shows the different steps performed when a TX descriptor is pushed:

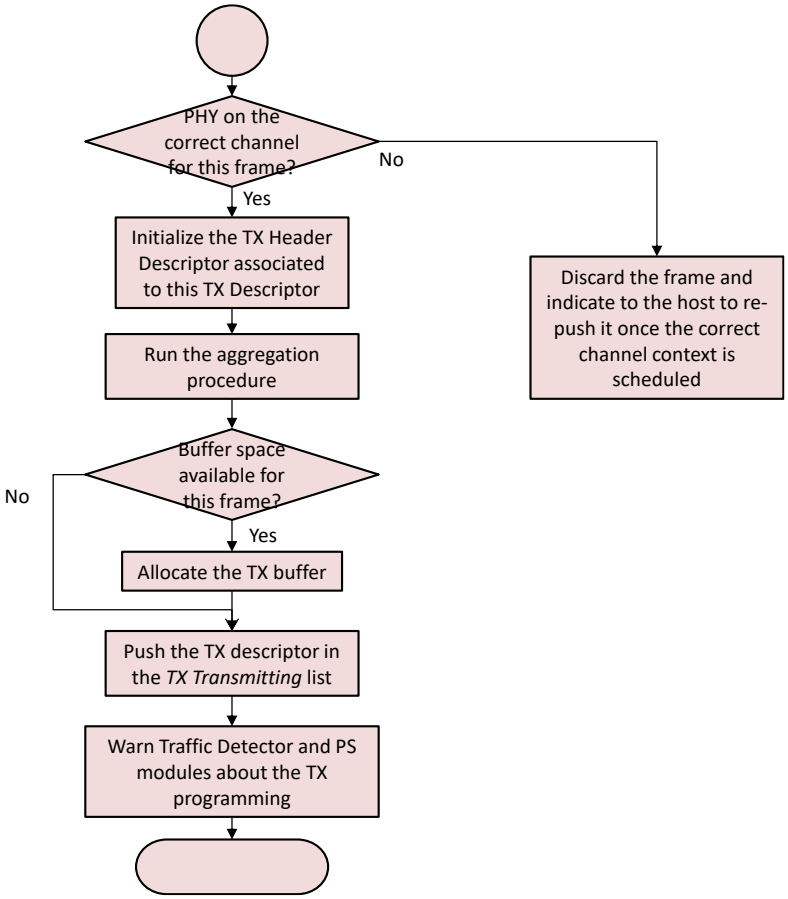



**Figure 23: TX Descriptor pushed in LMAC**

#### 2.4.3.2.2 Transmit Payload Handler

The TX Payload Handler is called in the context of the platform DMA interrupt that indicates that the payload downloads that were programmed are completed. This handler goes through the *TX Buffer* list and performs the following actions for each descriptor:

- ✓ Update the MAC Header when required (i.e. increment the sequence number for management and non-QoS data packets)
- ✓ Update the MAC HW descriptors associated to this transmission (i.e. update the start and end pointers, set the pointer to the policy table, set the InterruptEn bit, etc.)
- ✓ Chain the MAC HW descriptors to the MAC HW to start the transmission
- ✓ Update the policy table for the beamforming if applicable

This TX Payload Handler, i.e. the DMA payload interrupt, is not triggered for all the MPDUs to be transmitted. Indeed it is triggered only when the previous operations need to be performed, i.e. for singleton MPDUs, first MPDU of A-MPDUs, and for the MPDU indicating that enough data of an A-MPDU have been downloaded to allow the chaining to the HW (this last one might be merged with the first MPDU of the A-MPDU if it is long enough). The other MPDUs of an A-MPDU are downloaded silently.

The InterruptEn bit allows getting a MAC HW interrupt when the corresponding Transmit Header Descriptor has been handled by the HW.

It is set on each packet programmed for transmission. This mechanism allows filling again the TX buffer pool as long as transmissions have been performed, so that the transmission flow is not broken as long as there are packets to transmit.

#### 2.4.3.2.3 Transmit Trigger Handler

The TX Trigger Handler is the handler of the MAC HW TX trigger interrupt. The MAC HW triggers this interrupt when it finishes handling a Transmit Header Descriptor that has the InterruptEn bit set. This handler behaves differently depending on a state variable.

This state variable can take two values:

- ✓ TX Descriptor Check
- ✓ A-MPDU Descriptor Check

The default state is “TX Descriptor Check”. In this state, the TX trigger handler goes through the *TX Transmitting* list and performs the following actions on each descriptor:

- ✓ Check if the HW has handled the packet by reading its status
- ✓ Free the TX Buffer if handled by the MAC HW
- ✓ Check if the TX descriptor is the last one of an A-MPDU. In such case the state variable is put to the value “A-MPDU Check”
- ✓ If the TX descriptor is the last one of an A-MPDU or a singleton MPDU, and no other frames are programmed after it, then the A-MPDU in formation, if any, is finished and programmed to the HW.
- ✓ Push the descriptor in the *Tx Confirming* list

While in “A-MPDU Check” state, the TX trigger handler is not checking anymore the HW descriptors attached to the TX descriptors, but it is checking the BAR frame descriptor attached to the current A-MPDU transmission.

The status of this descriptor indicates if the BlockAck frame has been correctly received or if the BAR frame was retransmitted until it reached the retry limit without receiving the BlockAck.

If the BlockAck has been received, the TX Trigger Handler calls the RX Handler to be able to attach the BlockAck frame descriptor to the A-MPDU descriptor. Otherwise a NULL pointer is set in the A-MPDU descriptor. The state variable is then put back to “TX Descriptor Check”.

Before returning the TX trigger handler calls the TX Prepare function that will allocate new TX Buffers and program the associated payload download.

In case at least one TX Descriptor is ready to be confirmed to the Upper MAC, the Transmission Confirmation kernel event is also triggered.

#### 2.4.3.2.4 Transmit Confirmation Event

The TX Confirmation event is responsible for going through the *TX Confirming* list to upload the TX confirmations of the acknowledged packets. The operations performed by this event are described below:

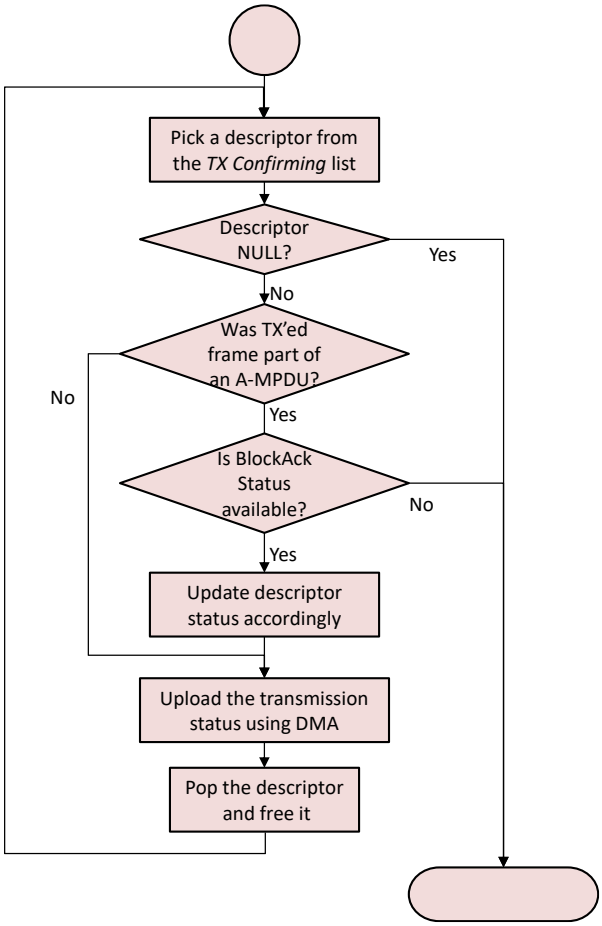



**Figure 24: Transmission Confirmation Event handling**

The freeing of the descriptor is performed by the IPC embedded layer as the descriptor pool is managed by this module.

#### 2.4.3.2.5 Transmit Confirmation DMA Interrupt

Once the upload of the programmed confirmation(s) is complete, the FW gets a DMA interrupt in which it asserts as interrupt to the Host CPU via the IPC mechanism. The host can then free the buffers in host memory and confirm the transmission to its upper layers.

### 2.4.3.3 Transmission Buffer Management

The transmission buffer management is not based on a pool of n buffers, but on a per-AC TX buffer area that can be divided into an undefined number of TX buffers. This implementation adapts to any type of traffic (big or small packets, and mixed) by allowing the buffering of the maximum number of packets.

A TX buffer area is defined as a 32-bit word array. A data structure is associated to each TX buffer area, that contains the parameters describing the current occupancy of the array:

- ✓ The index to the free space of the TX buffer area
- ✓ The size of the free space of the TX buffer area (in number of 32-bit words)
- ✓ The index to the lastly allocated TX buffer

When no buffer is allocated for a specific AC, the TX buffer area of this AC looks like this:

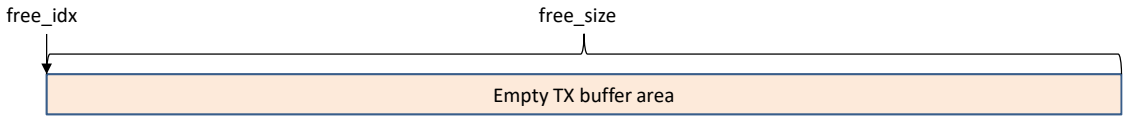



**Figure 25: Empty TX buffer area**

After allocating a TX buffer, the TX buffer area becomes:

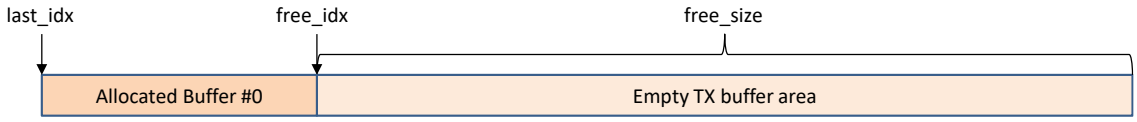



**Figure 26: Tx buffer area with 1 allocated buffer**

After a few allocations, the remaining free space may become almost full, disallowing the allocation of the next buffer. Two cases have to be distinguished:

1. The free space is smaller than a complete TX buffer header (i.e. the descriptors located in the buffer before the payload of the frame)

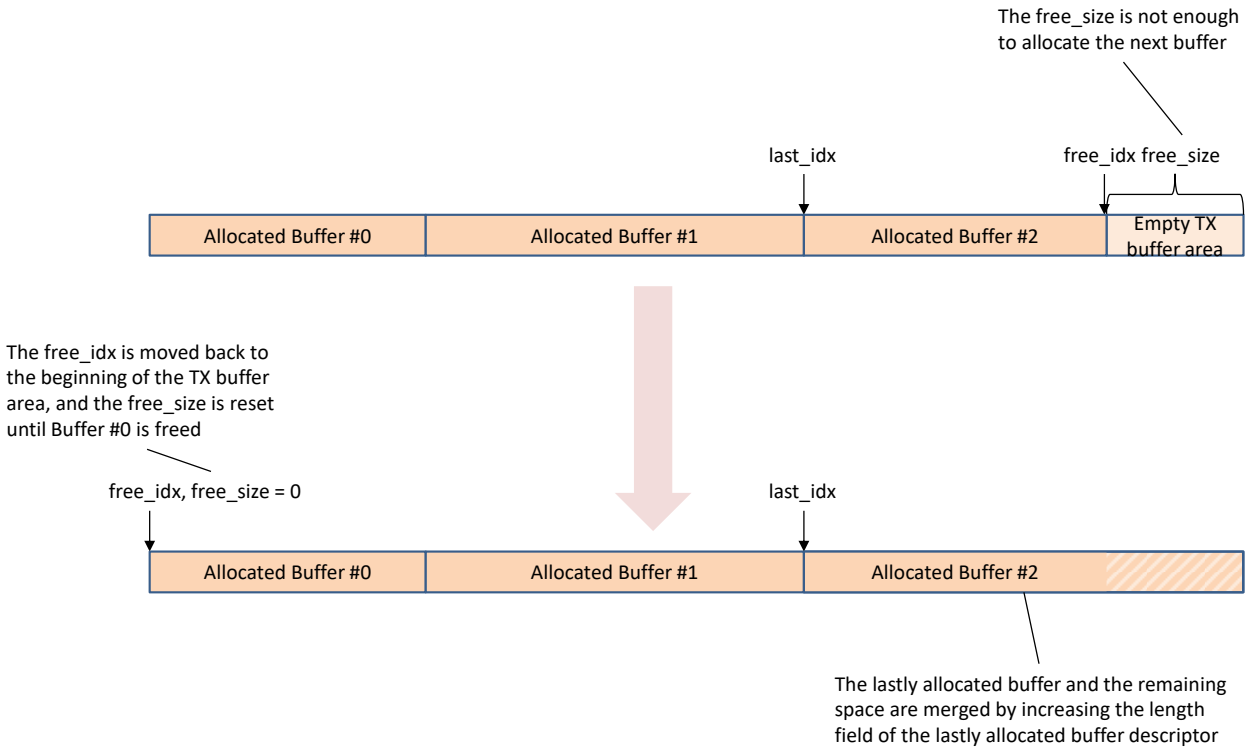



**Figure 27: Failing allocation and buffer wrap**

The next buffer will then be allocated once Buffer #0 is freed, if the space available is sufficient:

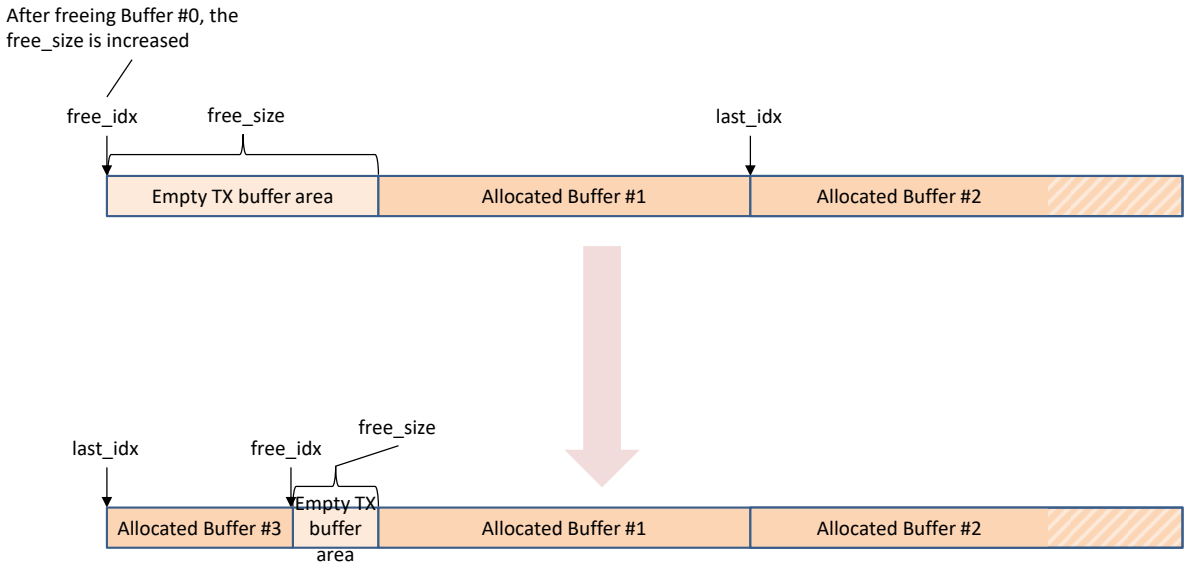



**Figure 28: Freeing and allocation of a new buffer**

2. The free space is enough to receive a complete TX buffer header

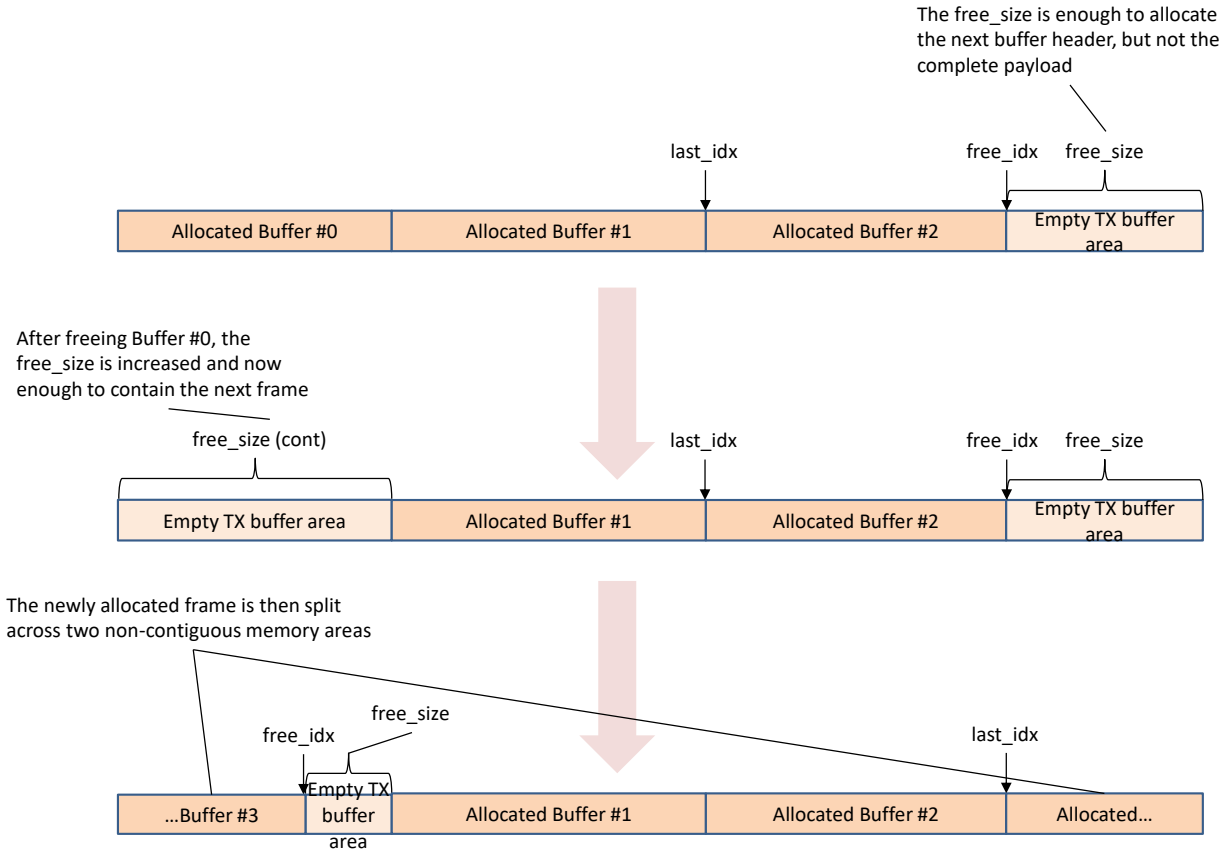



**Figure 29: TX buffer split**

The download of the payload to two non-contiguous areas will then be handled by the use of an additional PLF DMA descriptor that will be chained to the one stored inside the buffer. In the same way the chaining of the payload to the MAC HW will make use of an additional TBD.

### 2.4.3.4 Transmit Path Timing Diagrams – Singleton MPDUs

The figures below show the operation of the FW during two transmission cases of singleton MPDUs, i.e. a single MPDU or a stream of MPDUs. The different signals displayed are coming from the SW profiling feature described in 2.4.9.1.

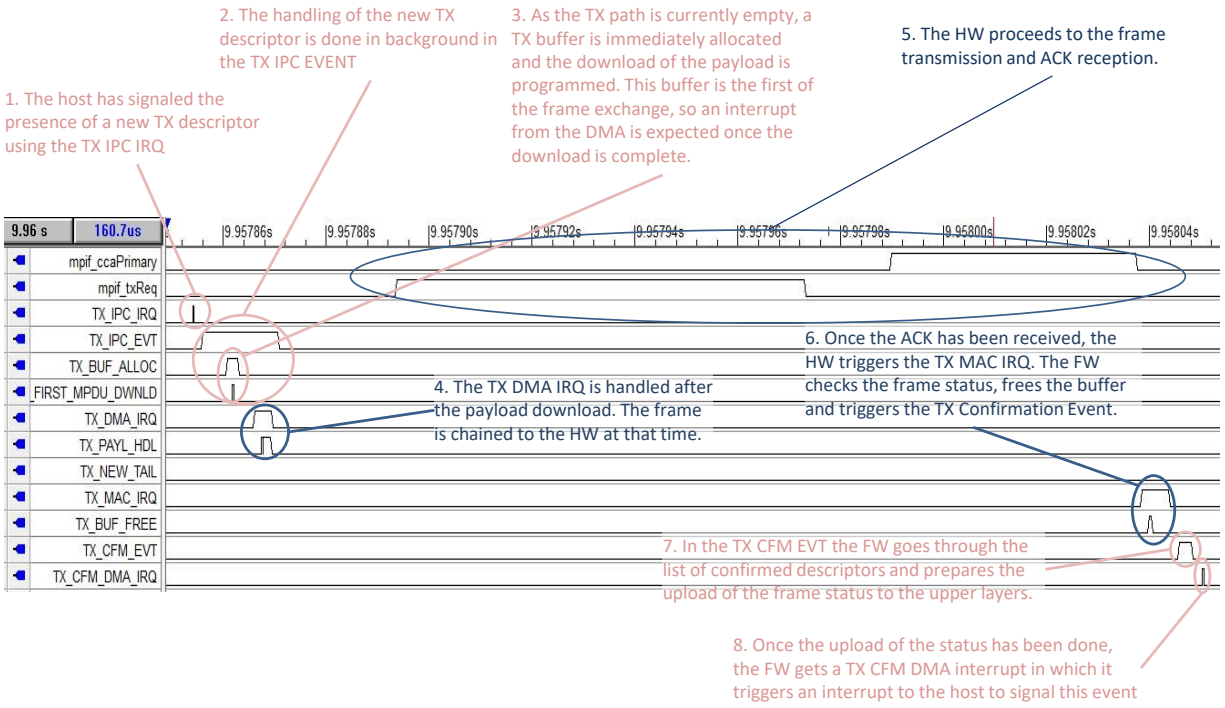



**Figure 30: Transmission of one singleton MPDU (no BlockAck agreement established)**

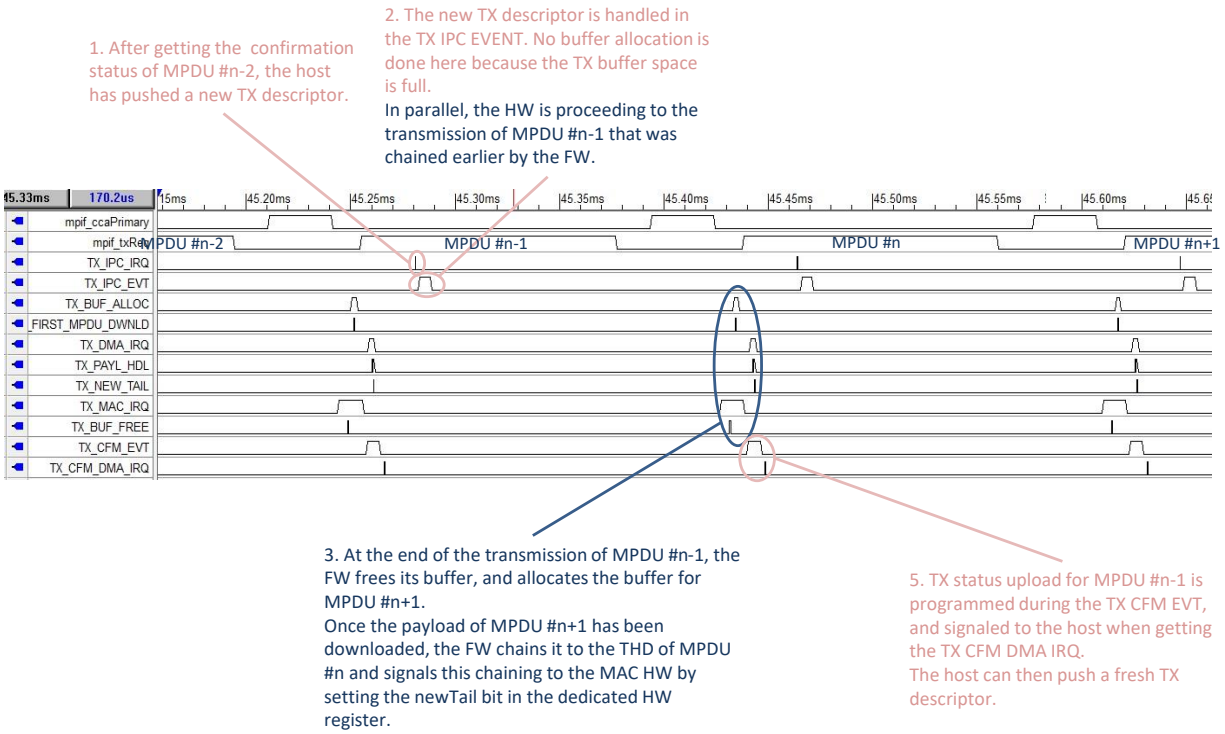



**Figure 31: Transmission of a stream of singleton MPDUs (no BlockAck agreement established)**

#### 2.4.3.5 A-MPDU Formatting

The formatting of an A-MPDU is split in three phases:

- Starting a new A-MPDU
- Adding MPDUs to the A-MPDU under construction
- Finishing the A-MPDU under construction

An A-MPDU is never programmed to the HW for transmission while it is under construction. It has to be finished first. Only a single A-MPDU per TX queue can be under construction at any time.

The basic principle of the A-MPDU construction algorithm is to wait as much as possible before closing the A-MPDU under construction, so that new frames can be added to this A-MPDU during the transmission of previous A-MPDUs or singleton MPDUs.

These A-MPDU formatting procedures are executed when a new TX descriptor is pushed by the UMAC to the LMAC (see 2.4.3.2.1), according to the algorithm below:

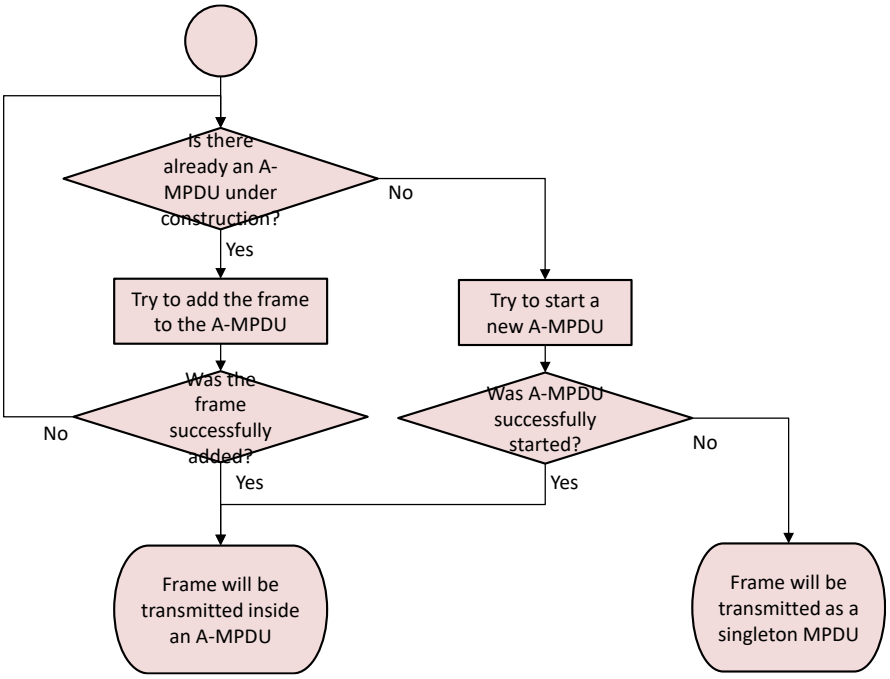



**Figure 32: A-MPDU formatting - Global view**

##### 2.4.3.5.1 A-MPDU Starting procedure

The start procedure mentioned in the algorithm described in Figure 32 behaves as stated in the figure below.

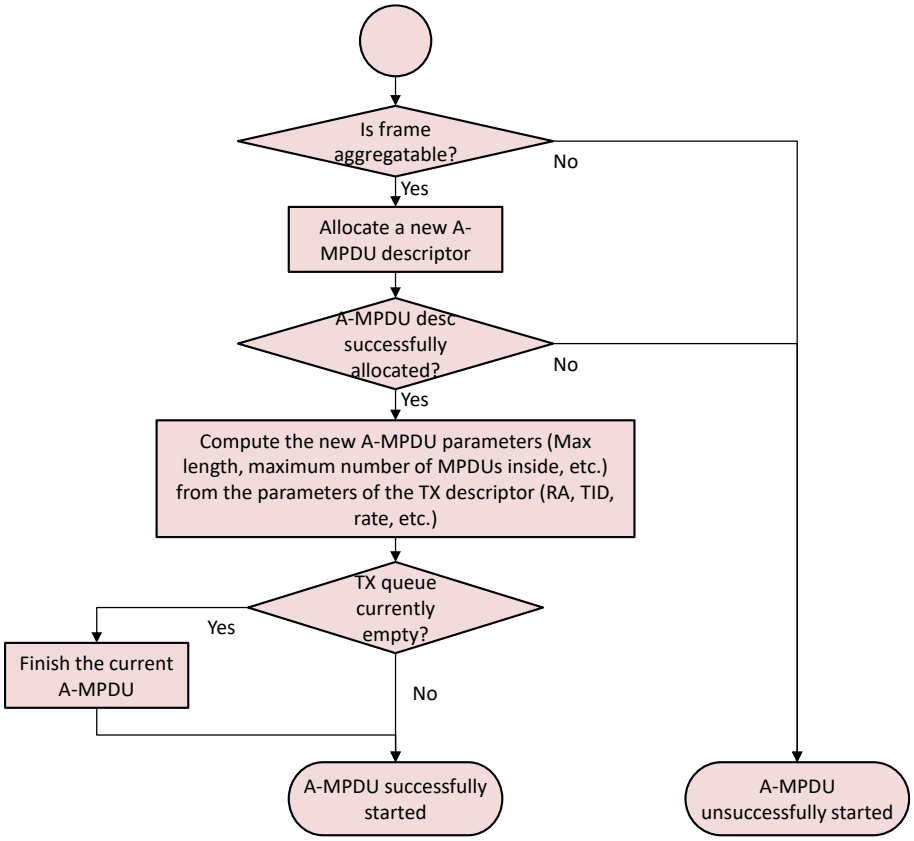



**Figure 33: Starting an A-MPDU**

The check against the TX queue status (TX queue currently empty?) in the figure above consists in verifying if there are already singleton MPDUs or A-MPDUs pending for transmission in the TX queue. If yes, then the current A-MPDU formatting is left open, because in such case it will be closed later, either because new frames added to the A-MPDU under construction allowed filling it at the maximum, or when the last singleton MPDU or A-MPDU pending for transmission is about to be transmitted.

If TX queue is empty, then the new A-MPDU is finished immediately (containing a single MPDU), in order to minimize the latency of the transmission.

#### 2.4.3.5.2 MPDU Adding procedure

This procedure is executed when a TX descriptor is pushed to the LMAC while an A-MPDU is under construction. It will try to add the new MPDU pushed to the A-MPDU under construction. The figure below shows the different checks that are performed during this procedure:

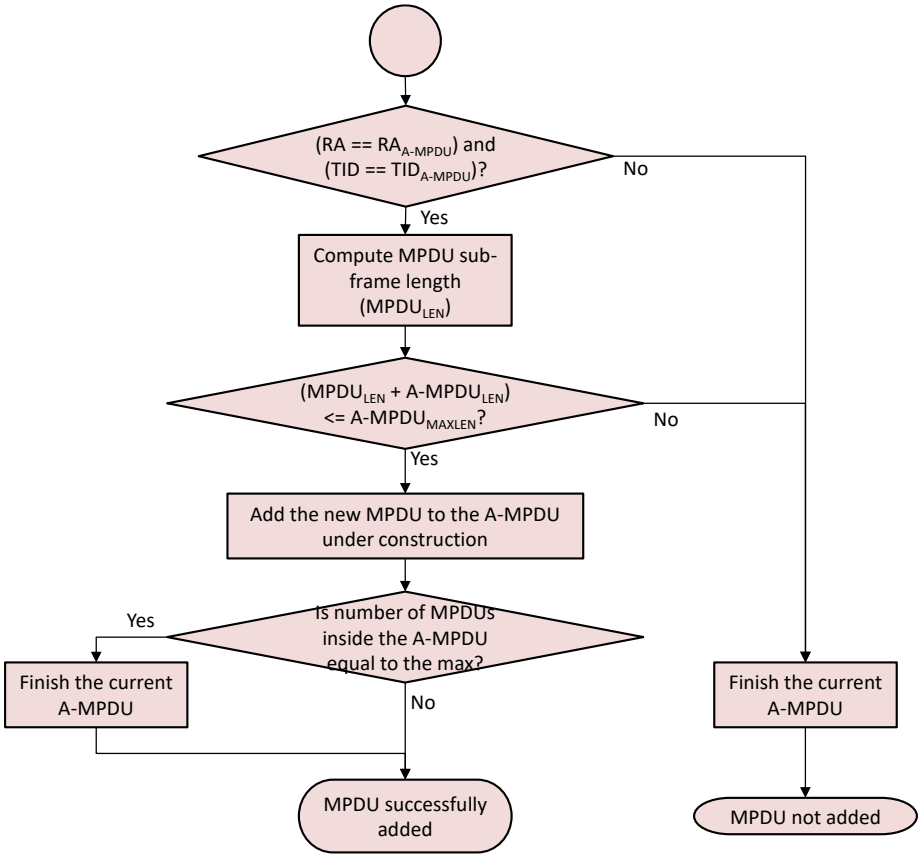



**Figure 34: Adding an MPDU to an A-MPDU under construction**

#### 2.4.3.5.3 A-MPDU Finishing procedure

An A-MPDU under construction is closed by invoking the A-MPDU finishing procedure. This procedure is triggered in the cases listed below:

- The TX queue is empty at the time the A-MPDU is started
- The maximum number of MPDUs or the maximum length of A-MPDU has been reached
- The new frame being pushed cannot be aggregated with the previous one (e.g. different RA/TID)
- The last singleton MPDU or A-MPDU pending for transmission is about to be transmitted

The finishing procedure updates the last descriptor of the built A-MPDU to configure it as a last MPDU of an A-MPDU. If the payload of the first MPDU of the A-MPDU has already been downloaded, the A-MPDU is immediately chained to the MAC HW. Otherwise, it will be chained once the first MPDU has been downloaded.

### 2.4.3.6 Transmit Path Timing Diagrams – A-MPDUs

The figures below show the operation of the FW during the transmission of a stream of A-MPDUs. The different signals displayed are coming from the SW profiling feature described in 2.4.9.1.

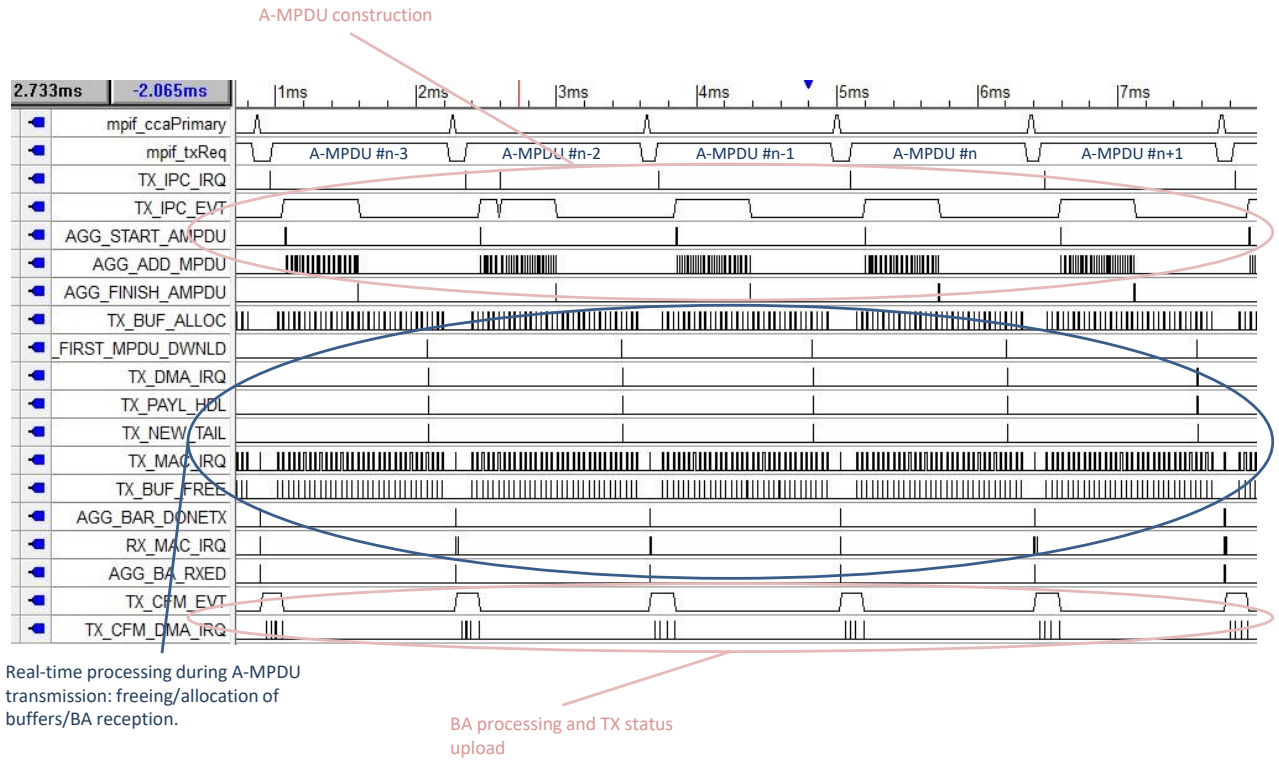



**Figure 35: Transmission of a stream of A-MPDUs (global view)**

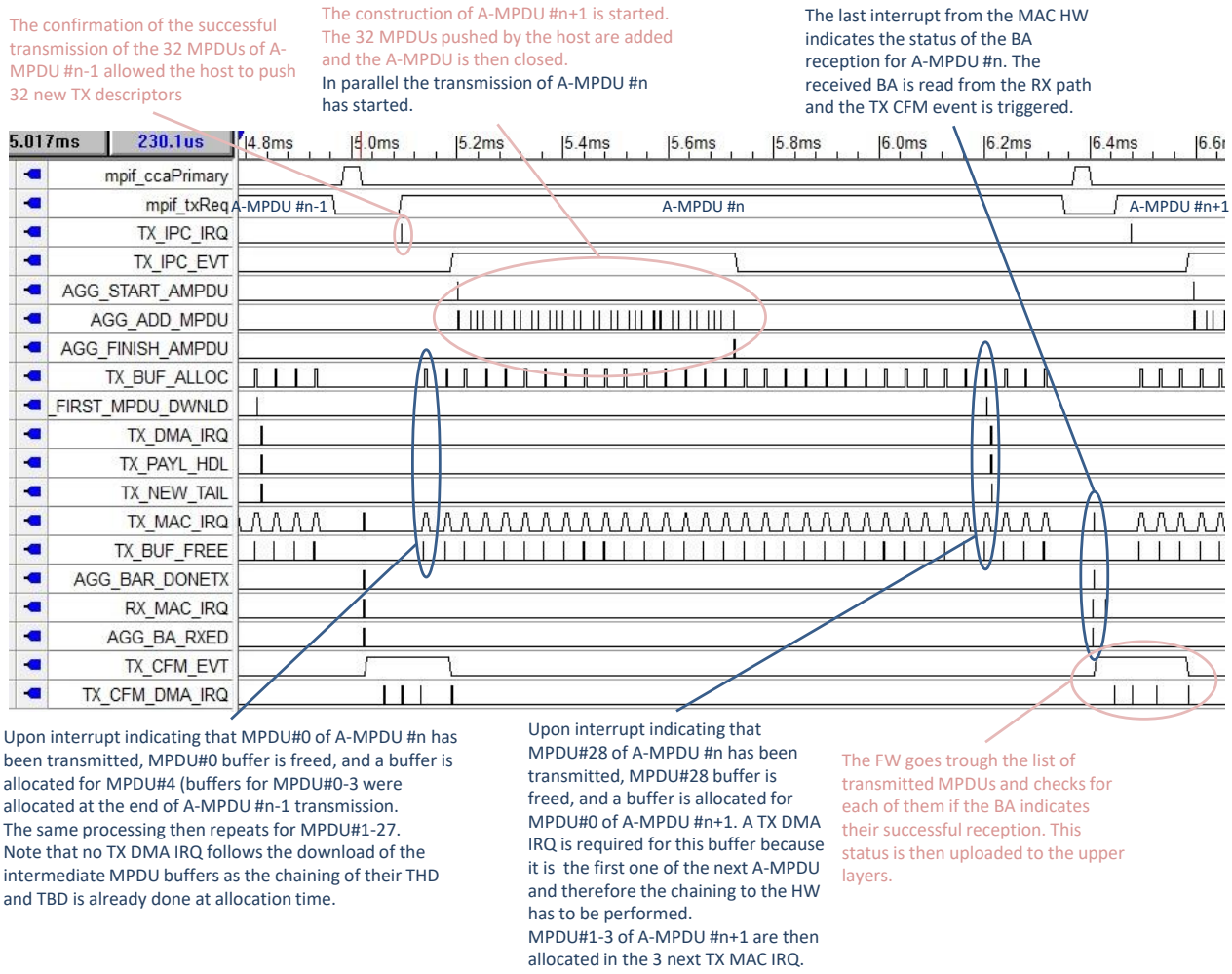



**Figure 36: Transmission of a stream of A-MPDUs (detail view)**

### 2.4.3.7 A-MSDU Transmission

The LMAC FW is able to transmit MPDUs composed of several MSDUs when the appropriate feature is activated at compile time. This feature modifies the API of the FW by adding several host address/length pairs to the TX descriptor instead of a single one.

The FW is not responsible for the formatting of the A-MSDU. The driver running on the host CPU takes the decision to assemble several MSDUs together and attach them to a single TX descriptor.

To build an A-MSDU each MSDU attached to the descriptor has to be formatted by the host CPU as an A-MSDU subframe (as defined in [1], chapter 8.3.2.2). The length passed in the descriptor is the complete length of the A-MSDU subframe including the padding length. In SoftMAC partitioning, the first MSDU buffer attached to the TX descriptor by the host CPU has to include the MAC Header additionally to the A-MSDU subframe as well as the IV/EIV if security is used.

Note that as the FW is simply concatenating the buffers pointed by the TX descriptor without any consideration of the actual buffer format (A-MSDU subframe or not), it is also possible to use this feature to transmit MSDUs the payload of which is split across several host buffers.

When getting a TX descriptor containing several buffer addresses, the FW computes the total length of the resulting A-MPDU by cumulating the different lengths and then proceeds to the A-MPDU construction algorithm. The A-MPDU construction is still done per MPDU as well as the BlockAck check phase.

The TX buffer management (allocation/freeing) is done per sub-frame inside the A-MSDU and a MAC HW TX interrupt is generated for each transmitted sub-frame, allowing the freeing of the current sub-frame and allocation of the next one.

The figure below shows a stream A-MSDUs over A-MPDUs. The A-MSDUs are composed of 2 MSDUs.

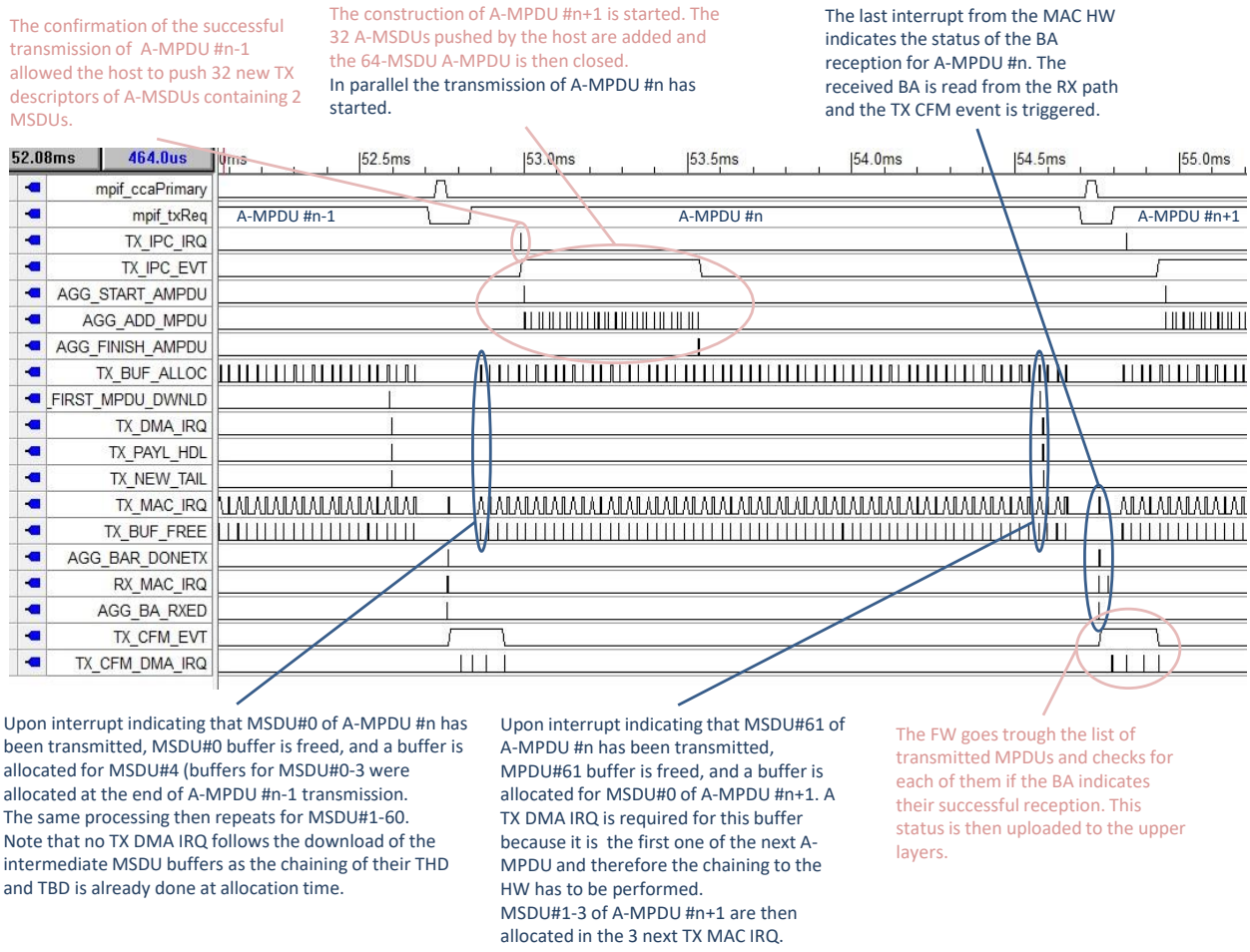



**Figure 37: A-MSDU over A-MPDU transmision**

### 2.4.4 Receive Path

These modules process the frames received from MAC HW. The receive process is mainly composed of the following actions:

- ✓ Invoke the DMA driver to move the payload to the application system if applicable.
- ✓ Provide an indication to the upper MAC SW

Additionally to this, the RX path is also responsible to handle some control frames (e.g. BlockAck frames that are injected to the TX confirmation module for completing the A-MPDU transmissions).

The main steps of the receive flow are described in the figure below:

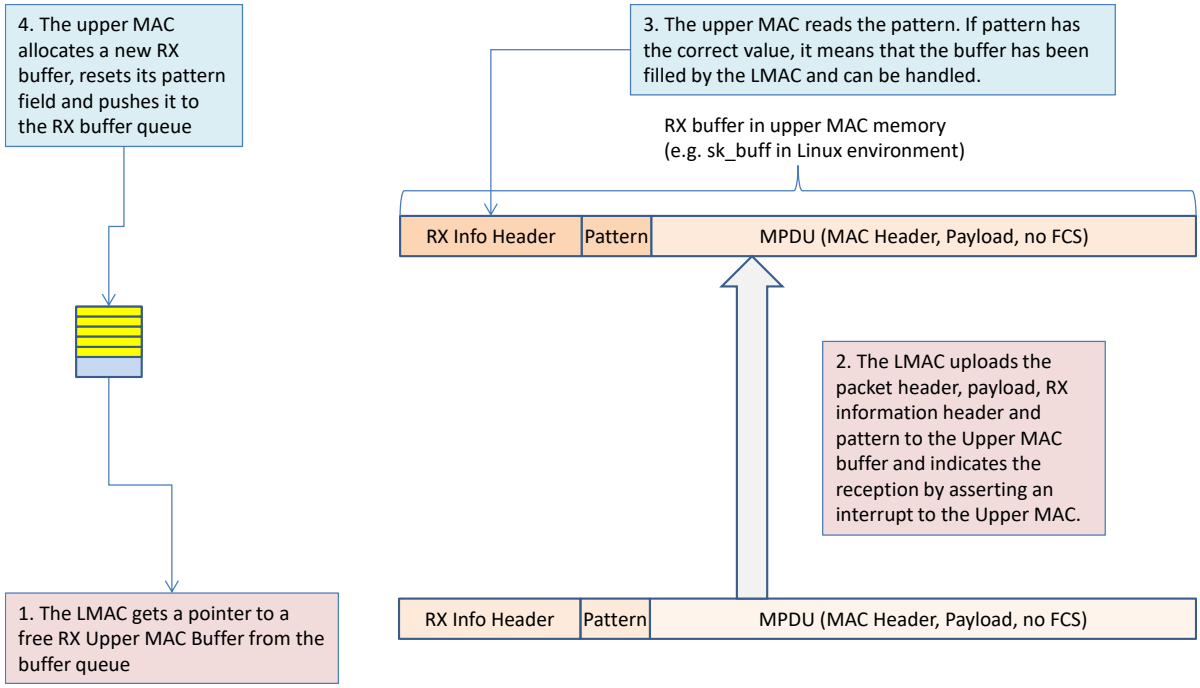



**Figure 38: RX flow**

### 2.4.4.1 RX Data Structures

#### 2.4.4.1.1 RX DMA Header descriptor

The RX DMA Header Descriptors contain all the information about the received packet. Only one of these descriptors is used per-packet. They are composed as follows:

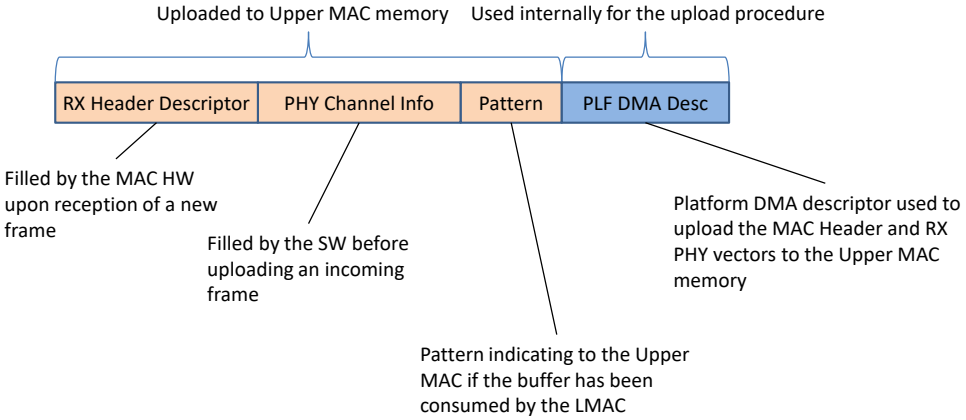



**Figure 39: RX DMA Header descriptor**

A part of the RX DMA descriptor (the RHD) is chained to the list of RHD that is passed to the HW. When a frame is received, the RHD is filled by the HW with the information about the packet (Length, Received rate, RSSI, etc.). See [4] for more information on the MAC HW/LMAC SW interface.

Another part of the RX DMA descriptor (the PHY channel information) is filled by the SW with the information on the channel on which the packet was received.

The Receive Header Descriptor and PHY channel information structures are both uploaded to the Upper MAC buffer allocated for the reception to form the RX Information Header (see Figure 38).

The Pattern is a 32-bit word used to synchronize the Upper MAC and the LMAC. When the Upper MAC allocates a new buffer and makes it available to the LMAC, it resets this field. Once the LMAC receives a frame, it will upload a specific value to this field, in order to indicate to the Upper MAC that the buffer has been handled by the LMAC and is now available for the Upper MAC.

The MAC Header Buffer is filled by the MAC HW with the MAC Header it receives. This field is uploaded to the Upper MAC buffer.

The platform DMA descriptor is used to upload the RHD, PHY Channel Info and Pattern. The pattern is the last data uploaded to ensure that once it is written to the Upper MAC Buffer, the complete frame is available.

#### 2.4.4.1.2 RX DMA Payload descriptor

The RX DMA Payload descriptors contain the payload data of the received frames, as well as descriptors allowing uploading this data to the Upper MAC memory. By default the size of the payload data that can be stored in one RX DMA Payload descriptor is 512 bytes (this value can be changed at LMAC compilation time), meaning that more than one descriptor can be required to store a complete packet.

This descriptor is composed of the following fields:

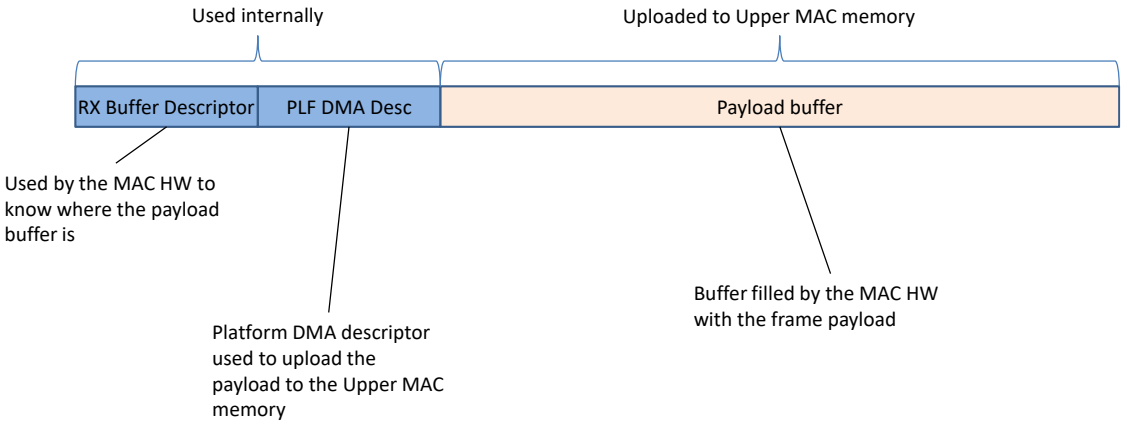



**Figure 40: RX DMA Payload Descriptor**

The RX Buffer Descriptor is used to indicate to the MAC HW where the payload buffer is located. This descriptor is chained to a list that is passed to the MAC HW (see [4] for more information on the MAC HW/LMAC SW interface).

The Platform DMA Descriptor is used to upload the payload buffer to the Upper MAC RX buffer.

The Payload Buffer is filled by the MAC HW. Once the frame is indicated to the LMAC SW, the data contained in this buffer is uploaded to the Upper MAC RX buffer.

#### 2.4.4.1.3 RX Descriptor

The RX Descriptor is a small element that is used only by the LMAC SW (i.e. not shared with the HW). It is linked on a 1:1 basis with a RX DMA Header descriptor and it is used as a packet descriptor passed from the MAC HW RX interrupt to the Platform RX DMA interrupt via the *RX Pending* list.

It is composed of the following fields:

- ✓ The pointer to the Receive Header Descriptor attached to the frame
- ✓ The pointer to the last Receive Buffer Descriptor carrying the payload of the frame

These fields are needed after packet upload, in order to free the different HW descriptors that were involved in this frame reception.

#### 2.4.4.1.4 RX HW Descriptor lists

The LMAC SW provides to the MAC HW the pointers to two descriptor lists:

- ✓ A list of Receive Header Descriptors
- ✓ A list of Receive Buffer Descriptors

The handling of these lists by the HW is described in [4]. From the LMAC SW point of view these two lists are composed of RX DMA Header Descriptors and RX DMA Payload Descriptors.

#### 2.4.4.1.5 RX Ready list

The RX Ready list is used to exchange the SW descriptors attached to the received frames between the MAC HW RX interrupt context and the RX kernel event responsible for the frame upload..

The SW descriptors are pushed to the RX Pending list when the associated frame upload has been programmed to the platform DMA, and popped from the list once the platform DMA transfer is complete.

#### 2.4.4.1.6 RX Pending list

The RX Pending list is used to exchange the SW descriptors attached to the received frames between the RX kernel event context and the platform DMA interrupt context.

The SW descriptors are pushed to the RX Pending list when the associated frame upload has been programmed to the platform DMA, and popped from the list once the platform DMA transfer is complete.

### 2.4.4.2 Reception Main Steps

The main reception steps are executed in four main contexts:

- ✓ The MAC HW RX interrupt context
- ✓ The RX kernel event
- ✓ The Platform DMA event context
- ✓ The RX LMAC to Host interrupt mitigation timer interrupt context

The sections below describe the operations performed in these two contexts.

#### 2.4.4.2.1 MAC HW RX interrupt

The MAC HW Rx interrupt is asserted by the HW when one or more frames have been received (depending on the RX IRQ mitigation mechanism, see [4] for more details about this mechanism).

The operations performed in this context are described below:

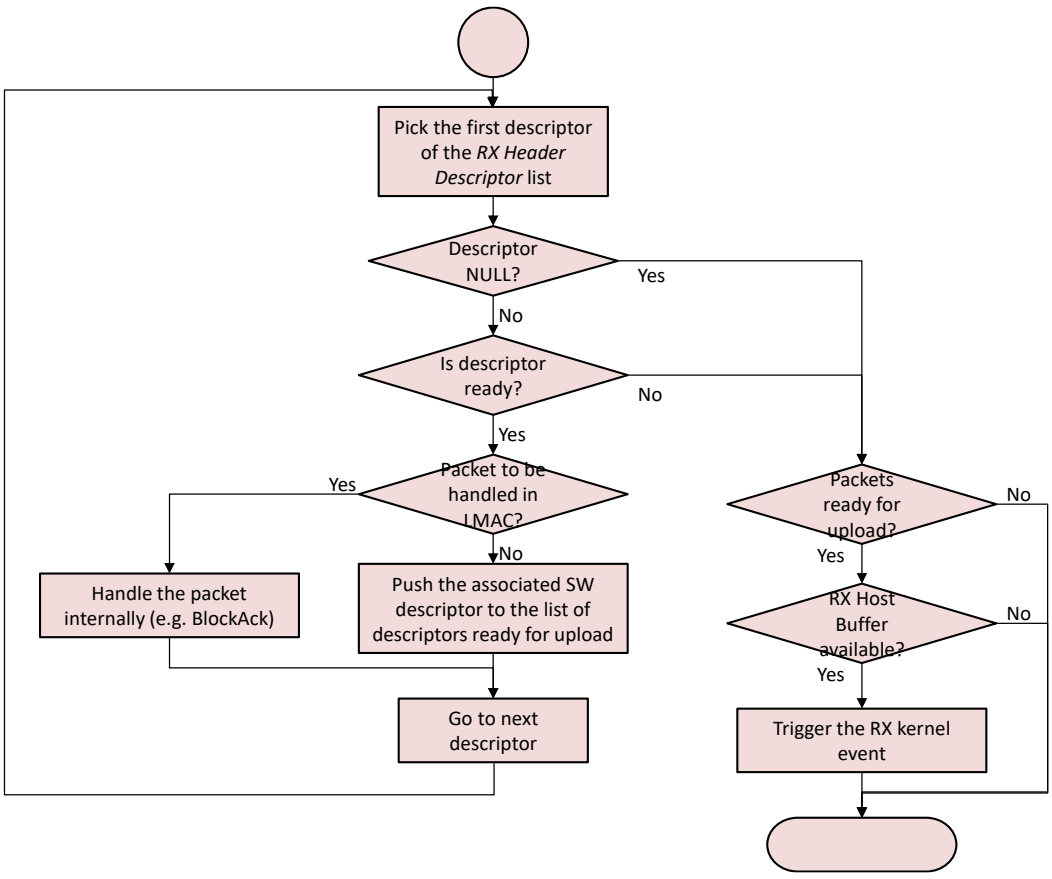



**Figure 41: MAC HW RX interrupt handling**

The descriptor is considered ready when its status is indicated as “Done” by the MAC HW, indicating that the frame is available.

The way the RX Host Buffer pointer is retrieved is outside the scope of this document as it depends on the inter-processor communication mechanism that is available on the platform. An example of such implementation is described in [5].

In case no RX Host Buffer is available for the frame upload, the RX interrupt handler returns immediately and it is the responsibility of the inter-processor communication mechanism to call again the RX interrupt handler once a RX buffer is made available by the Upper MAC.

The SW descriptor attached to each received frame is pushed into the *RX Ready* list to be later handled in the RX kernel event.

The SW descriptor attached to each frame uploaded to the Upper MAC memory is pushed into the *RX Pending* list while the DMA transfer to Upper MAC memory is ongoing.

#### 2.4.4.2.2 RX kernel event

The Rx kernel event is triggered from the MAC HW RX interrupt. This event is responsible for preparing the frame upload to the upper MAC memory.

The operations performed in this context are described below:

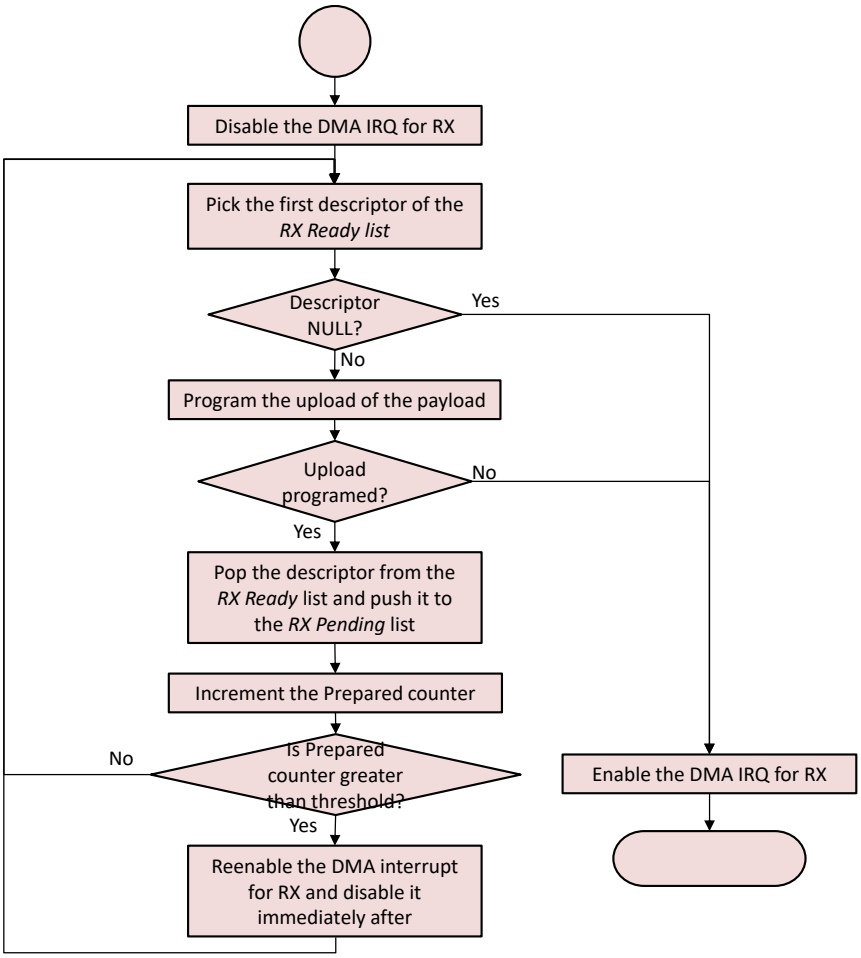



**Figure 42: RX kernel event processing**

The way the RX Host Buffer pointers are retrieved is outside the scope of this document as it depends on the inter-processor communication mechanism that is available on the platform. An example of such implementation is described in [5].

In case no RX Host Buffer is available for the frame upload, the RX interrupt handler returns immediately and it is the responsibility of the inter-processor communication mechanism to trigger again again the RX kernel event once a RX buffer is made available by the Upper MAC.

The PreparationCounter is used to periodically allow the RX DMA interrupt to be executed (thus freeing MAC HW RX descriptors).

The SW descriptor attached to each frame uploaded to the Upper MAC memory is pushed into the *RX Pending* list while the DMA transfer to Upper MAC memory is ongoing.

#### 2.4.4.2.3 Platform DMA event

This event is the deferred action triggered from the platform DMA interrupt handler. This interrupt is asserted once one or more frames have been uploaded to the upper MAC memory.

The operations performed in this context are described below:

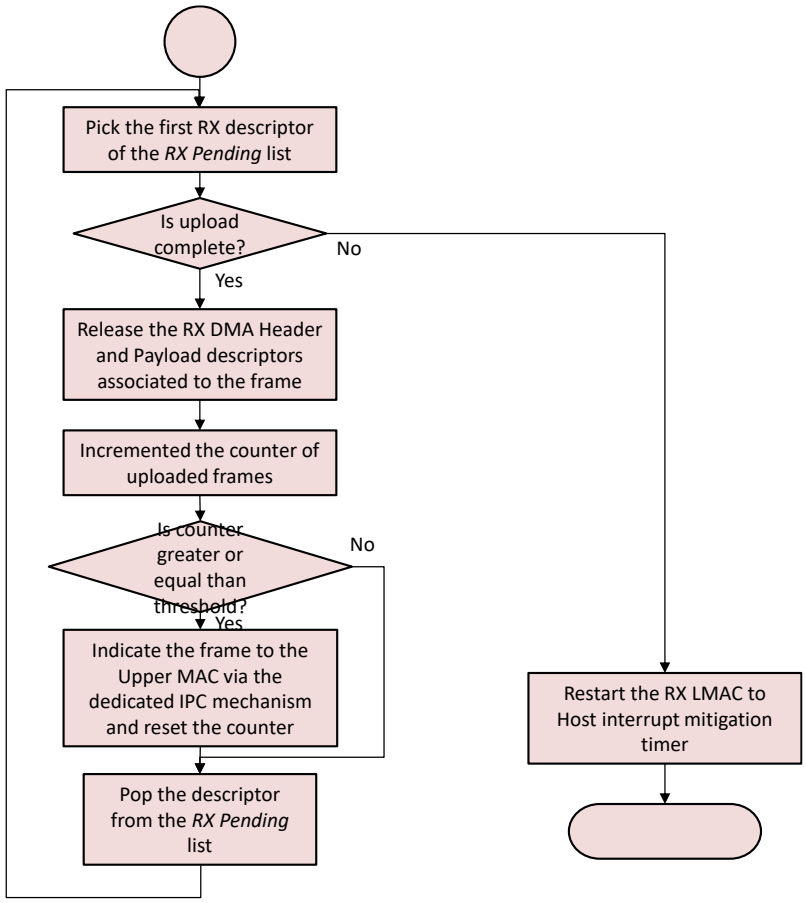



**Figure 43: Platform DMA RX interrupt handling**

The releasing of the RX DMA Header and Payload descriptors is done by appending these descriptors to the corresponding list of HW descriptors (see 2.4.4.1.4).

A LMAC to Host interrupt mitigation mechanism is implemented to limit the number of interrupts asserted to the Host CPU. This mechanism is based on an uploaded frame counter and a HW timer. The frames are indicated to Host

CPU only when the frame counter reaches a specific threshold, or when the timer expires. In order not to add latency, the first frame of a burst of frames is indicated immediately to the Host, while following frames will be indicated depending on the threshold or the timer expiry.

#### 2.4.4.2.4 RX LMAC to Host IRQ mitigation timer interrupt context

This interrupt is asserted by the MAC HW when the interrupt mitigation timer expires.

The operations performed in this context are described below:

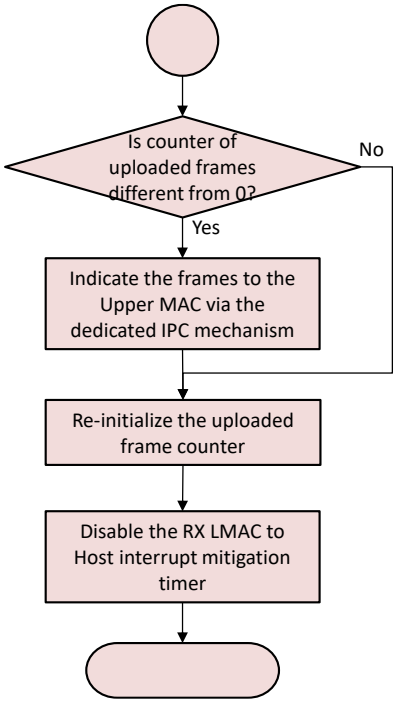



**Figure 44: LMAC to Host IRQ mitigation timer interrupt handling**

### 2.4.5 Traffic Detection Block

The Traffic Detection (TD) block purpose is to maintain statistics on received and transmitted packets. These information can then be used by other modules (PS, CHAN and P2P modules) in order to adapt their behavior accordingly with the requirements of the established links.

This block is automatically added if DPSM (see 2.4.6.3) or Channel Context (see 2.4.7.5) feature or P2P PS as GO (see 2.4.7) is supported.

Implemented version of the Traffic Detection Block is very basic and only counts number of packets received and transmitted **on each VIF** over a period of time called Traffic Detection Interval. If this number is higher than a defined threshold, it will be considered that traffic is generated on the RX and/or the TX path (see Figure 45).

The Traffic Detection Interval can be updated by modifying the TD_DEFAULT_INTV_US value in td.h file.

The Traffic Detection Threshold can be updated by modifying the TD_DEFAULT_PCK_NB_THRESS value in td.h file.

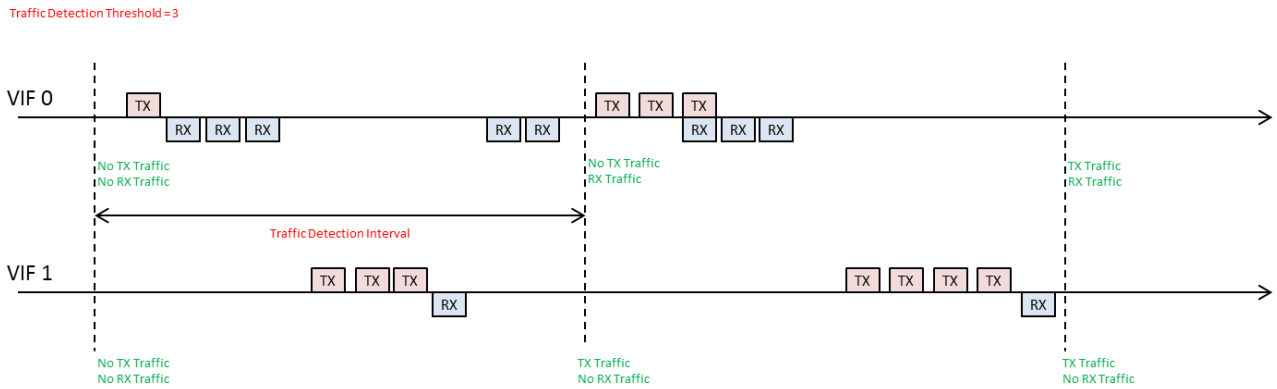



**Figure 45: Traffic Detection**

### 2.4.6 Power Saving Block

The Power Saving (PS) block is responsible for the management of the sleep state of the device.

This block can be added by setting the ***PS*** compilation flag to on.

#### 2.4.6.1 Legacy Power Save

Legacy PS mode allows a STA device to enter in DOZE state between beacon receptions in order to decrease its power consumption. A STA indicates that it is using PS mode by setting the PM bit to 1 in a transmitted NULL frame. The AP will then bufferise all data frames dedicated to this STA.

Using the TIM information present in an AP's beacon, a STA knows that the AP has buffered frame for it. By sending PS Poll packets, the station can download all these bufferised packets. Once all packets have been received (indicated by a More Data bit (MD) set to 0), the station can go back to the DOZE state until next TBTT.

The figure below exposes a data transfer between an AP and a STA under PS mode:

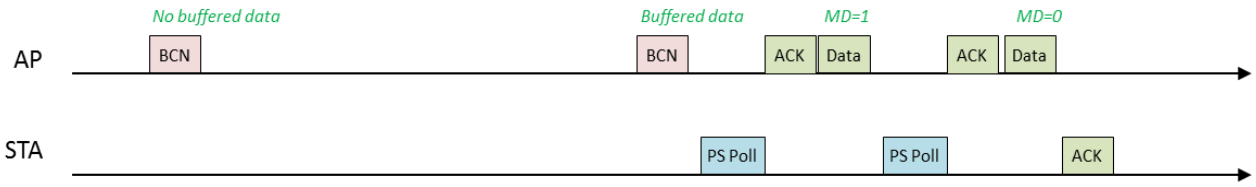



**Figure 46: Legacy PS Data Transfer**

Use of Legacy PS mode can be enabled or disabled by using the MM_SET_PS_MODE_REQ message. More details on this message can be found in [3].

#### 2.4.6.2 WMM-PS/UAPSD

Unscheduled Asynchronous Power Save Delivery (UAPSD) improves efficiency of the Legacy Power Save by decreasing the number of frames a client will have to send in order to download a same number of frame buffered on AP side.

Four Access Categories (AC) are defined in EDCA (AC VO, AC VI, AC BE, and AC BK) corresponding to voice, video, best effort and background. Each AC of a station can be configured separately to be delivery/trigger-enabled or not, either at association time or through the usage of specific frames.

This is a per-queue and per-VIF mechanism, data transfers occurring on a non-UAPSD enabled queue will use the mechanism exposed in previous section.

By sending a trigger frame (basically a QoS Null or a QoS Data frame), a station can regularly ask for the AP to transfer all the packets that had been bufferised for the queue that are UAPSD-enabled.

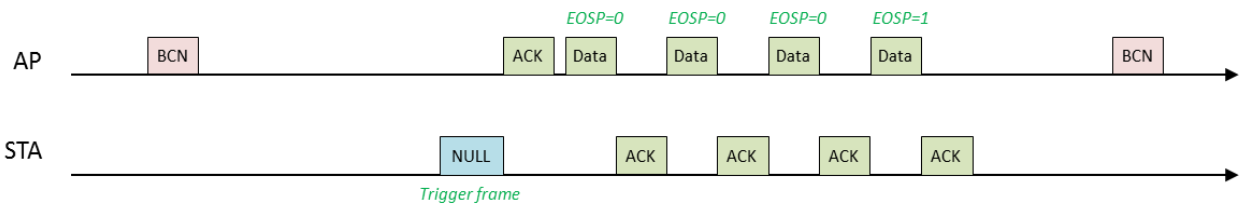



**Figure 47: WMM-PS/UAPSD Data Transfer**

Support of UAPSD as STA is defined at compilation time by setting the ***UAPSD*** to on.

Time between two consequent trigger frames sent on a VIF can be set in the MM_START_REQ message (uapsd_timeout parameter). UAPSD status of a queue for a given VIF can be updated by using the MM_SET_EDCA_REQ message. More details on these two messages can be found in [3].

#### 2.4.6.3 Dynamic Power Save Mode

Throughput achieved during a data transfer between an AP and a STA using Legacy PS is limited due to the necessity for the STA to send PS-Poll packets between each data packets.

Aim of the Dynamic Power Save Mode (DPSM) is to use information collected by the Traffic Detection Block (see 2.4.5) in order to pause the PS mode upon detection of a data transfer. This will lead to an higher throughput during the whole data exchange. Once this exchange is finished, PS mode is automatically restarted in order to reduce the power consumption.

This feature can be added to the code by setting the ***DPSM*** compilation flag to on.

#### 2.4.6.4 Sleep State

Once all kernel activities have been completed, the stack checks the status maintained by the PS module is order to know if it is allowed to enter in doze mode.

Two levels of status are used:

- A global status preventing from going to sleep during global operations (see Table 2)
- A status for each VIF preventing from entering in sleep mode during beacon reception and following data exchanges (see Table 3)

| Bit Number | Flag Name | Description |
|---|---|---|
| **0** | PS_TX_CFM_UPLOADING | TX module is uploading TX Confirmation for the host |
| **1** | PS_SCAN_ONGOING | Scan procedure is ongoing |
| **2** | PS_IDLE_REQ_PENDING | SW has asked for the HW to enter in IDLE and is waiting for the IDLE interruption. |
| **3** | PS_PSM_PAUSED | PS is paused due to traffic detection (DPSM). |

**Table 2: PS Environment Prevent Sleep Bits**

| Bit Number | Flag Name | Description |
|---|---|---|
| **0** | PS_VIF_WAITING_BCN | STA is waiting for AP Beacon reception |
| **1** | PS_VIF_WAITING_BCMC | STA is waiting for end of broadcast/multicast packets reception |
| **2** | PS_VIF_WAITING_UC | STA is waiting for end of unicast packets reception |
| **3** | PS_VIF_WAITING_EOSP | STA is waiting for the end of the service period |
| **4** | PS_VIF_ASSOCIATING | Association with AP is in progress |
| **5** | PS_VIF_P2P_GO_PRESENT | P2P GO device is present |

**Table 3: VIF Environment Prevent Sleep Bits**

Figure 48 exposes the different steps of the decision tree allowing to determine if device can enter in sleep mode or not.

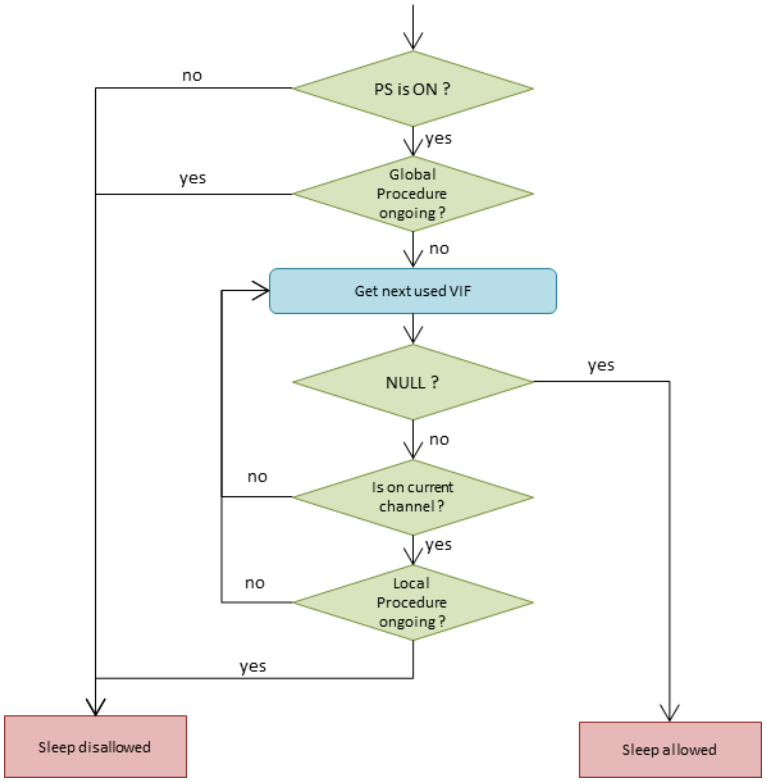



**Figure 48: Sleep Check**

### 2.4.7 P2P Block

The P2P block is responsible for managing the P2P Power Management features described in [7].

A classic AP device is considered to be always awoken and is always able to receive data packet coming from a STA device. WiFi P2P technical specification introduces two new Power Management mechanisms allowing a P2P Group Owner (P2P GO) to be absent for defined periods. These mechanisms are called Notice of Absence (NoA) and Opportunistic Power Save (OppPS).

By default, only the P2P Client (P2P CLI) part of the protocol is supported when the ***P2P*** flag is set to a non-zero value. Support of the P2P GO is part of the stack only if ***P2P_GO*** is defined. Section 2.4.7.4 introduces use of the ***P2P_GO_PS*** compilation flag.

#### 2.4.7.1 P2P VIF Creation

A VIF dedicated for a P2P use has to be created using the same API as the one used for a classic VIF and defined in [3]. The p2p boolean value has to be set to true in that case.

The number of P2P VIFs that can be added is defined at compilation time (P2P value in scutils.py file).

When a P2P VIF is used, the LMAC will take care of:

- Not sending packets using a 11b rate
- Managing P2P GO absences as P2P CLI
- As P2P GO, computing NOA parameters to be used in a topology using Concurrent Modes.

#### 2.4.7.2 P2P PS Procedures

This section briefly describes the PS procedures introduced by the WiFi P2P protocol.

##### 2.4.7.2.1 Notice of Absence Procedure

The NoA procedure allows a P2P GO device to announce a schedule of absence periods so that CLI devices are aware that no data exchanges can happen during these periods.

It can be used for two main purposes:

- Reducing the GO power consumption
- Allowing the GO to be connected with another device on another channel. In that case, absence periods will match with period during which GO device in on another channel

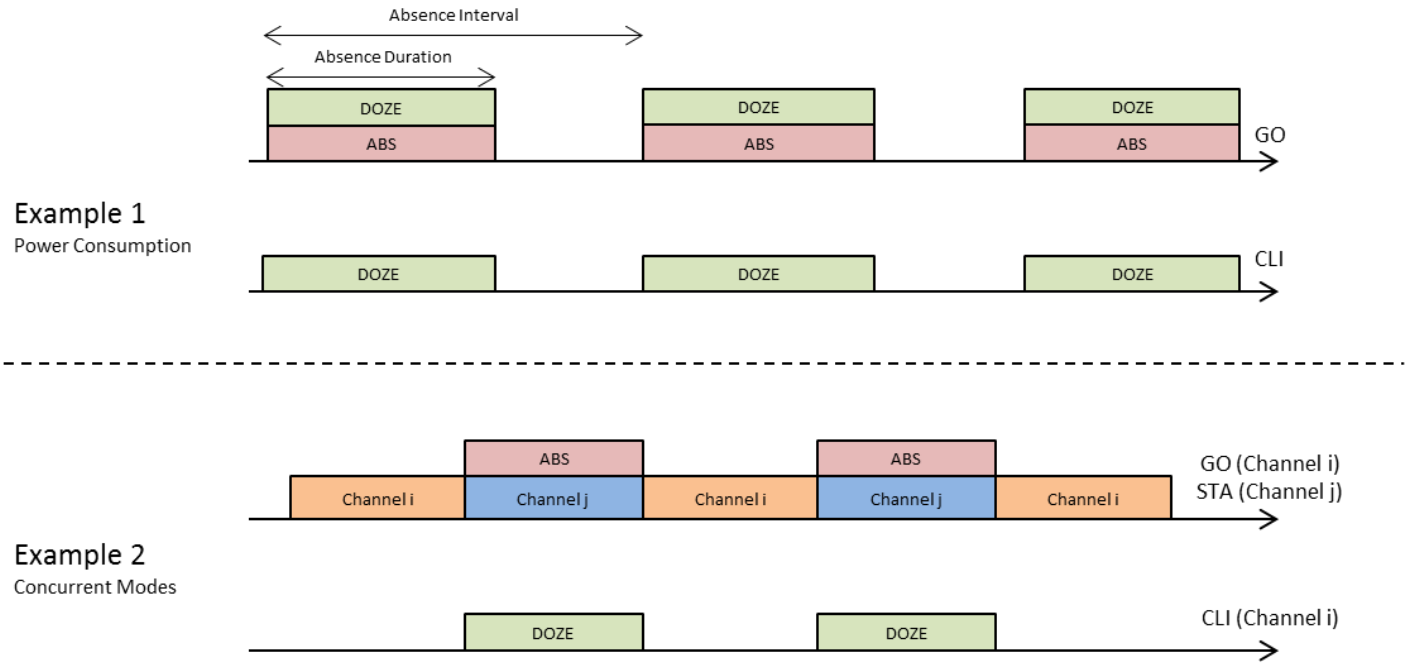



**Figure 49: Notice of Absence**

#### 2.4.7.2.2 Opportunistic Power Save Procedure

Opportunistic Power Save is a power management scheme that allows a P2P GO to gain additional power savings on an opportunistic basis. If this mode is supported, the GO device will monitor the PS state of each connected CLI device; if at any time outside a CTWindow starting from its TBTT, all its peer devices are entered in doze mode, the GO device is authorized to enter in doze mode until its next TBTT.

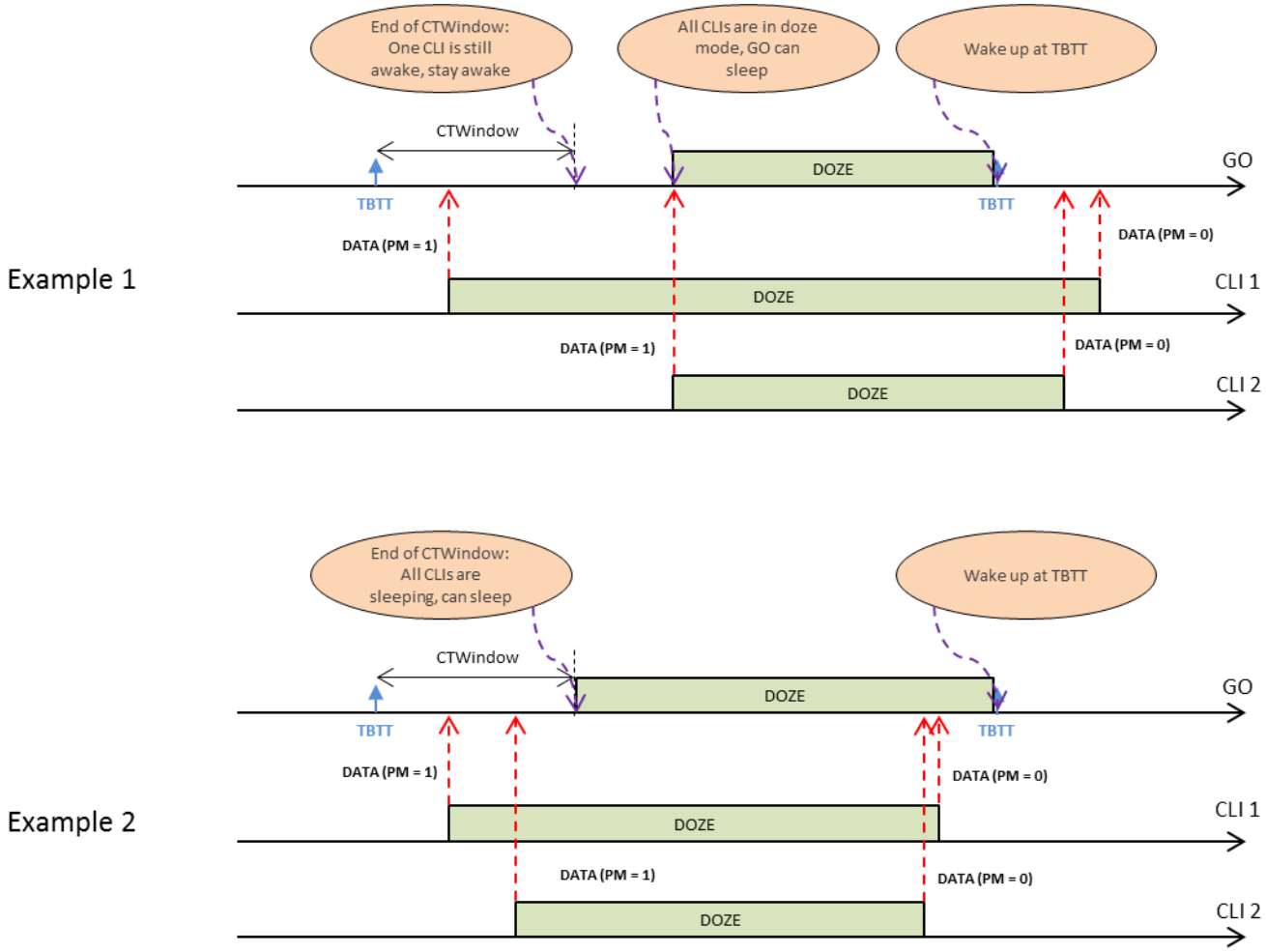



**Figure 50 – Opportunistic Power Save**

### 2.4.7.3 P2P Client and P2P PS

When a P2P GO PS state is doze, it is required that the associated P2P CLI does not try to send data packet. Then it is mandatory for a P2P CLI to monitor the beacons received from a P2P GO in order to determine the periods during which the P2P GO device can be absent.

Content of each new received beacon (meaning received beacon CRC is different from the previous received CRC) is checked in order to detect presence or not of the NoA attribute. The following tables introduce the content of this attribute and the meaning of the different parameters handled by the RW LMAC:

| Field Name | Size (octets) | Value | Description |
|---|---|---|---|
| **Length** | 2 | n*(13) + 2 | Length of the P2P Notice of Absence attribute body in octets |
| **Index** | 1 | 0 - 255 | Identifies an instance of Notice of Absence timing. This value has to be modified each time one of the values present in the NoA attribute is updated. |
| **CTWindow and OppPS Parameters** | 1 | - | Parameters indicating P2P Group Owner's availability window and opportunistic power save capability (see Table 5) |

| | | | |
|---|---|---|---|
| **Notice of Absence Descriptor(s)** | n*13 | - | Zero or more Notice of Absence Descriptors each defining a Notice of Absence timing schedule (see Table 6) |

**Table 4: Notice of Absence attribute format**

| Bit | Subfield | Notes |
|---|---|---|
| 7 | **OppPS** | Set to 1 to indicate that the P2P GO is using opportunistic power save.<br>The CTWindow field shall be non-zero when the OppPS bit is set to 1. |
| 0-6 | **CTWindow** | Client Traffic Window. A period of time in TU after a TBTT during which the P2P GO is present. |

**Table 5: CTWindow and OppPS Parameters field format**

| Field Name | Size (octets) | Value | Description |
|---|---|---|---|
| **Count/Type** | 1 | 1 - 255 | Indicates the number of absence intervals. 255 shall mean a continuous schedule. |
| **Duration** | 4 | - | Indicates the maximum duration in units of microseconds that the P2P GO can remain absent the start of a Notice of Absence interval |
| **Interval** | 4 | - | Indicates the length of the Notice of Absence interval in units of microseconds |
| **Start Time** | 4 | | The start time for the scheduled expressed in terms of the lower 4 bytes of the TSF timer |

**Table 6: Notice of Absence Descriptor format**

Note that several NoA descriptors can be part of the NoA attribute (n value in Table 4). In current implementation, the number of NoA that can be handled in parallel by the RW LMAC is 2.

#### 2.4.7.4 P2P GO Presence State Update

As NOA, OPPPS and TBTT related operations are independent from each other, precedence rules have been defined in order to decide if a P2P GO device is active or not. The following rules are sorted from highest to lowest priority:

1. *Absence due to a non-periodic Notice of Absence (only one absence period)*
2. *Presence from TBTT until the end of Beacon frame transmission*
3. *Presence during the CTWindow*
4. *Absence for a periodic Notice of Absence*

Figure 51 describes the state machine used to decide if P2P GO is present or not based on the status of each above situation.

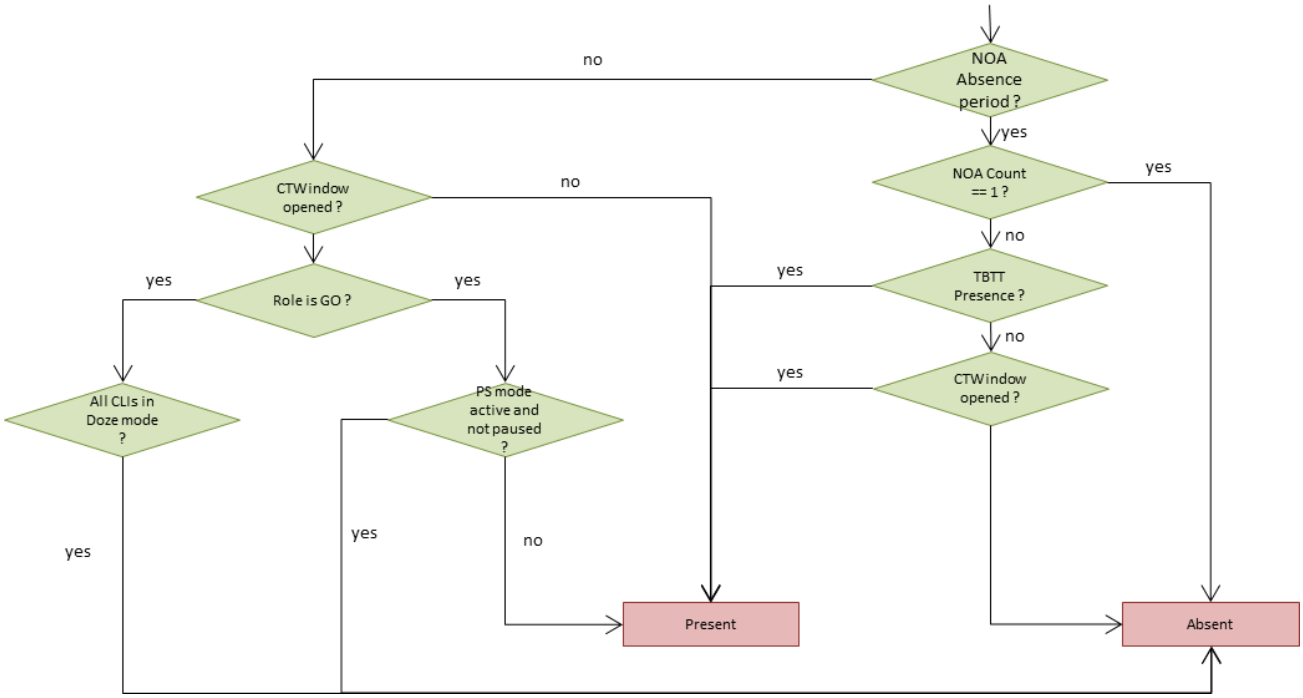



**Figure 51: P2P GO Presence State Update**

### 2.4.7.5 P2P GO PS Feature

By default use of NOA is only triggered if a STA VIF is scheduled in parallel with a GO VIF. ***P2P_GO_PS*** compilation flag allows to always start NOA when a GO VIF is activated.

The parameters of the created NOA are as followed:

- *Count = 255 (Continuous NOA)*
- *Absence Interval = Beacon Interval*
- *Absence Duration = 80 % of the Absence Interval*
- *Start Time is placed so that end of Absence Period occurs 5ms before TBTT.*
- *Opportunistic PS Supported*
- *CTWindow = 10ms*

The NOA is stopped if Traffic Detection module detects an high traffic during the presence period. It is automatically restarted once no more traffic is seen.

### 2.4.8 Channel Context Block

Purpose of the Channel Context block is to schedule time spent on a specific channel in a configuration where several connections on different channels are established.

This block is optional and is compiled only if ***CHNL_CTXT*** compilation flag is set to on.

#### 2.4.8.1 Channel Context Life Cycle

The API to be used by a driver in order to:

- Add a Channel Context
- Link a Channel Context with a VIF
- Schedule a Channel Context
- Unlink a Channel Context and a VIF
- Delete a Channel Context

is defined in [3].

One given VIF can't be linked with more than one channel context, one channel context can be linked with several VIFs.

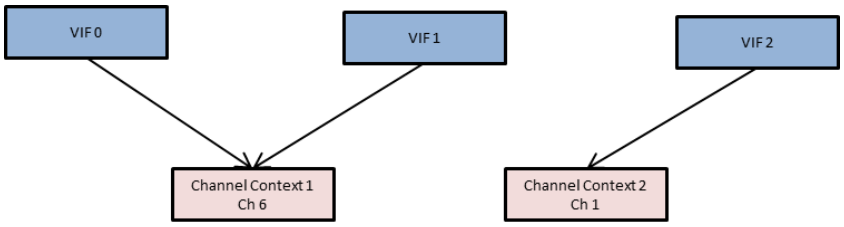



**Figure 52: Example of VIF/Channel Context links**

Note also that it is required that no VIF/Channel Context link exists when trying to delete a channel context.

When the SW takes the decision to switch from a channel context to another, it has to request the HW to enter in the IDLE state in order to properly terminate the current RX/TX operation. This step can take several milliseconds depending on the used rate and the packet size (see figure Figure 53).

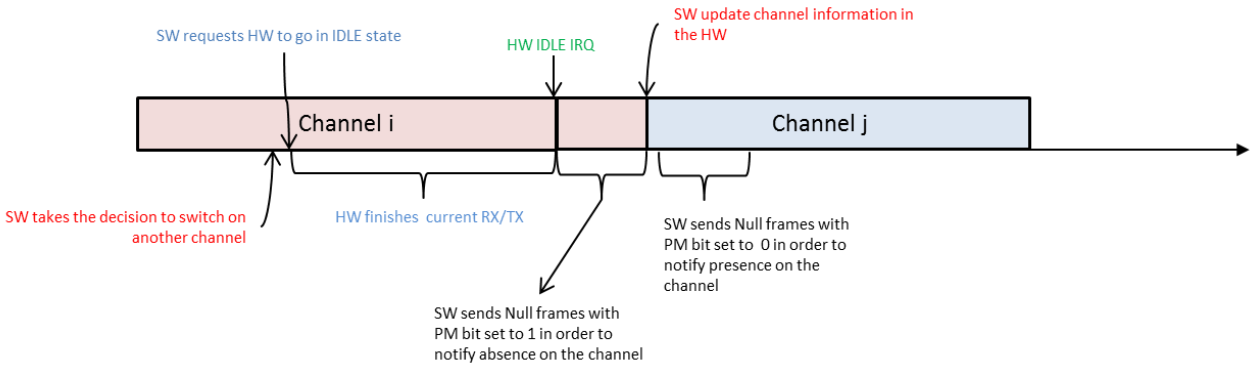



**Figure 53: Channel Switch Steps**

Once the HW_IDLE_IRQ is raised, the SW sends one Null Frame per VIF with the PM bit set to 1 in order to announce that he is leaving the channel. To do so, the HW state is briefly set to ACTIVE until all TX confirmation have been received. Then, the PHY is programmed in order to use a new channel. Final step is sending of Null frame with PM bit set to 0 in order to announce presence on the channel.

### 2.4.8.2 Channel context block overview

It is mandatory for a STA device to be present on a channel at least for reception of DTIM beacons that can be followed by reception of broadcast/multicast data packets. An AP has to be present on its traffic channel every beacon interval in order to transmit its beacon. In the rest of this chapter those periods are referred as TBTT window, whatever the interface type is.

When several channel contexts are active, it is then necessary to schedule contexts switch is order to respect the TBTT window of each interface.

Note: Currently a maximum of two active contexts is supported (excluding contexts used for connection-less operations described in [2.4.8.7]).

The scheduling between contexts is done when a TBTT window ends, following these steps:

1. Add next TBTT window for this interface in the list.
2. Get the next TBTT window from the list.
3. Schedule context switches, knowing that when the next TBTT windows starts the current active context must be the context of the TBTT.

Before describing in more details those steps, here is a basic example of to illustrate.

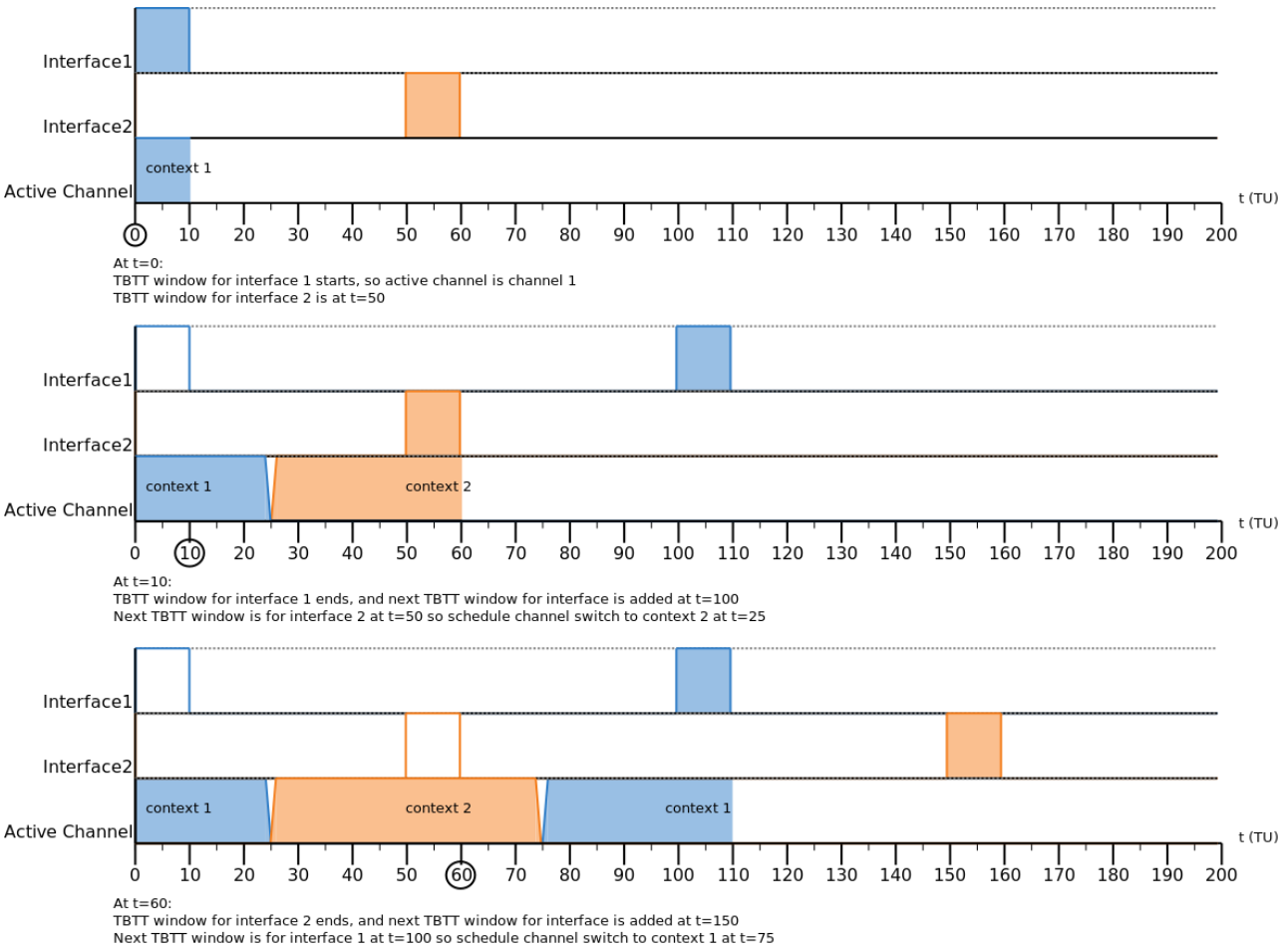



**Figure 54: Channel context scheduling example with two interfaces: interface1 using context1 and interface2 using context2**

#### 2.4.8.3 TBTT window list

The Channel Context block keeps an ordered list of TBTT windows for all active interfaces. When a TBTT window ends then the next TBTT window for this interface is added in the list using those simple rules:

1. The date on the next TBTT window is provided by the MAC Management block.
2. If this TBTT window overlaps with another TBTT window already on the list and using a different channel context, it is “skipped”. It is recursively re-added with a date incremented by one beacon period until there is no more overlap. The overlap check also take into account the channel switch duration.

When a TBTT window is “skipped”, it means that the Channel Context block will not schedule a channel switch to respect it.

Those rules gives more importance to TBTT window with higher beacon interval (as there are fewer of them) and fairly distribute “skipped” windows among interface with similar beacon interval.

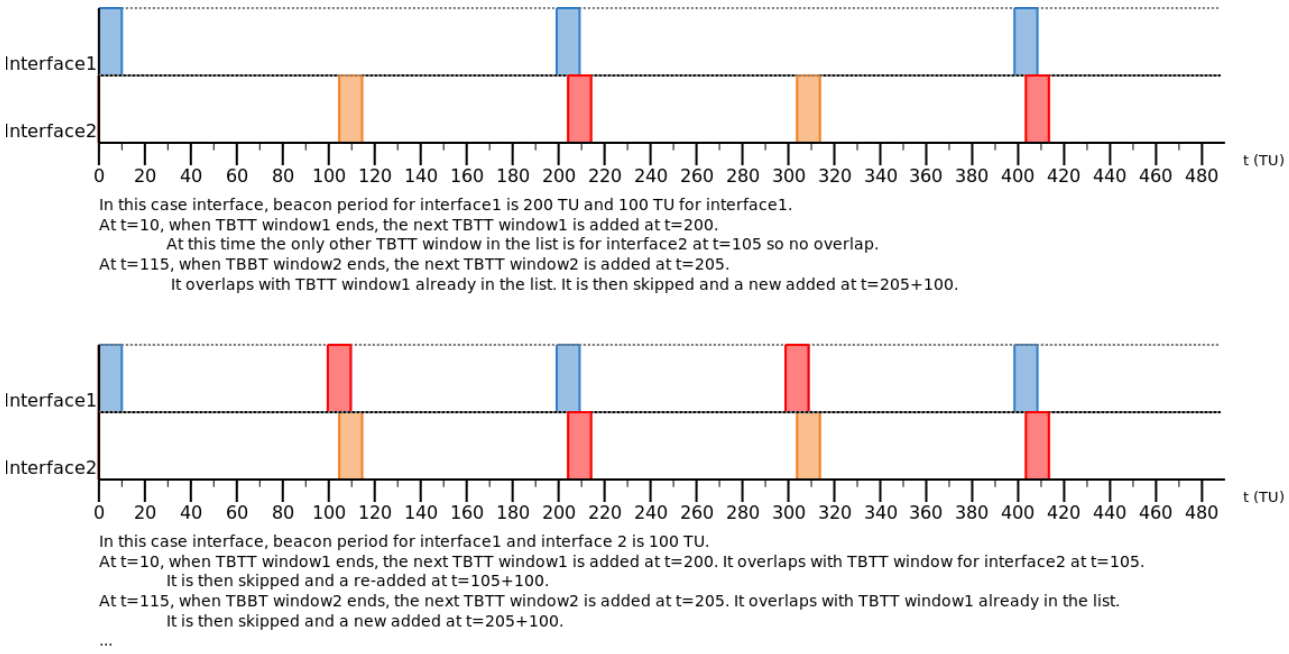



**Figure 55: Example of cases when TBTT windows are skipped**

When a TBTT window of a P2P GO interface is skipped, a new non-periodic NOA is created to inform clients that the GO will be absent. Indeed a GO interface must be present for TBTT even if this happens during an absence period of a period NOA.

#### 2.4.8.4 Channel context switch scheduling

Channel switch scheduling is done when a TBTT window ends until the start of the next TBTT window. The former fixes the source context and the latter fixes the destination context.

The scheduling is different whether a P2P GO interface is active or not. We’ll first describe the rules used when only STA interfaces are active and then describe the rules used when a P2P GO interface is active.

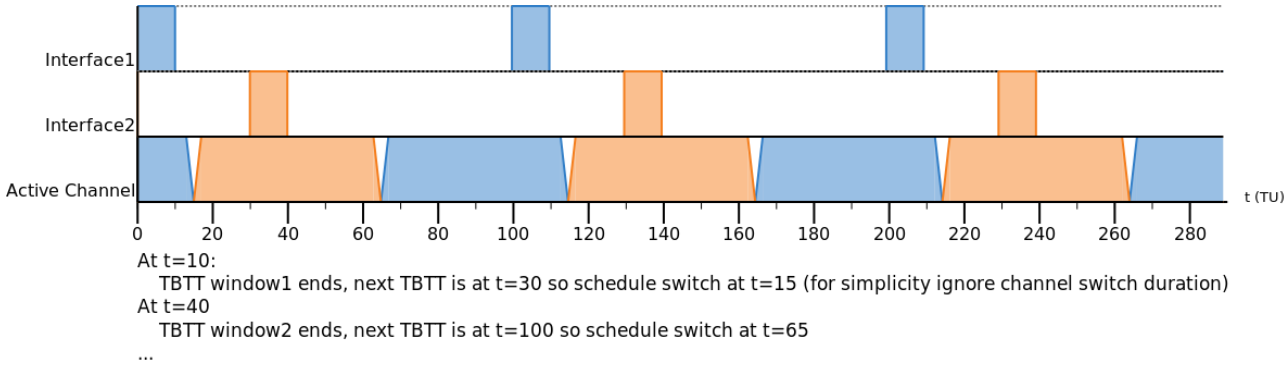
More complete channel scheduling are described in Other channel scheduling example.

##### 2.4.8.4.1 Scheduling without P2P GO interface

When only STA interfaces are active the scheduling is quite simple:

1. **Source and destination context are differen**t:
   In this case a single context switch to the destination context is scheduled. The date of the switch is currently selected so that both context have the same medium time. The distribution may easily be adapted to allow more medium time to interface with more traffic.

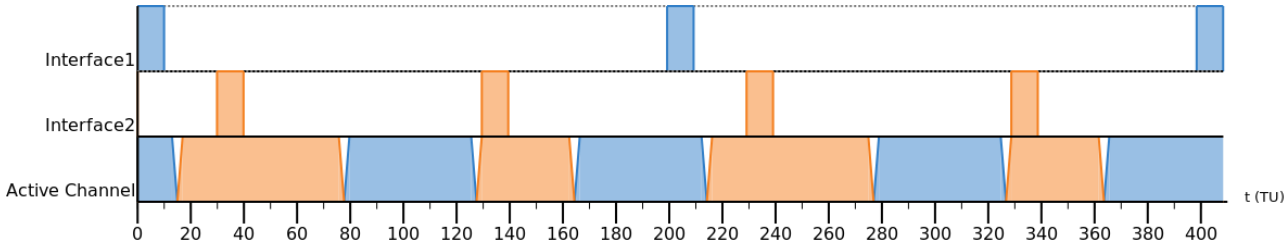
Note that tests done when adding TBTT window in the list ensure that there is always enough time to switch to the other context before start of next TBTT (otherwise that next TBTT would have been skipped).




At t=10:
TBTT window1 ends, next TBTT is at t=30 so schedule switch at t=15 (for simplicity ignore channel switch duration)
At t=40
TBTT window2 ends, next TBTT is at t=100 so schedule switch at t=65
...

**Figure 56: Example of channel scheduling with switch at each TBTT**

2. **Source and destination context are the same and next TBTT is far enough:**
In this case two channel switches are scheduled before next TBTT. One to switch to the other context and one to switch back to the destination context. Again the switch date are selected to fairly split medium time between both contexts.




At t=10:
TBTT window1 ends, next TBTT is at t=30 so schedule switch at t=15
At t=40:
TBTT window2 ends, next TBTT is at t=130 on the same context.
Since there is enough time, so schedule two context switch to spend some time on the other context
Schedule is done so that the same amount of time is spend of both channels from the start of the current window until the start of the next one.
At t=140:
TBTT window2 ends, next TBTT is at t=200 on the other context. Schedule a single switch at t=165
...

**Figure 57: Example of channel scheduling with two consecutive TBTT windows on the same channel with enough time in between**

3. **Source and destination context are the same and next TBTT is near:**
In this case no channel switch is scheduled. “next TBTT is near” meaning that there isn’t enough time to schedule two context switches like previously.
Here the amount of time spent on both channels is not equal because, currently, the time spent between two consecutive TBTT windows on the same channel without switch is not taken into account.

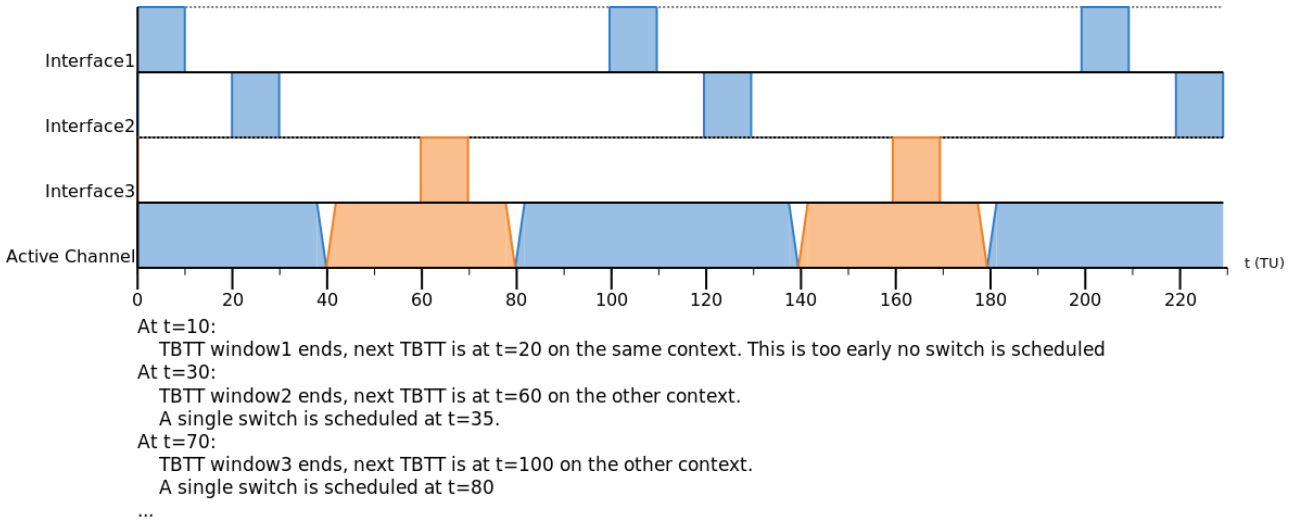



**Figure 58: Example of channel scheduling with two consecutive TBTT windows on the same channel with not enough time in between**

#### 2.4.8.4.2 Scheduling with P2P GO interface

When a P2P GO interface is active the channel context block must also take into account the NOA of the interface when channel switch are scheduled. See [Notice of Absence Procedure and [Support for P2P GO interface paragraphs for more details on how NOA are started on the P2P GO interface, but the important points are:

1. TBTT windows for interface not on the same channel as the P2P GO interface always happen during an absence period of the P2P GO interface. (actually NOA are started to respect this rule)
2. The medium time repartition between contexts is dictated by the NOA: the channel active is the channel of the P2P GO channel during presence periods and the other context during absence periods.
3. There is one exception to the previous rule: The active channel must be the P2P GO channel even during absence period during a TBTT window for an interface on the P2P GO channel (It may be the TBTT window of the P2P GO interface or possibly another interface on the same channel).

With this new constraints the scheduling now looks like:

1. **Source and destination context are differen**t:
   Like in 58, one channel switch will be scheduled to switch to the destination context. The difference is that the switch date will be determined by the NOA.

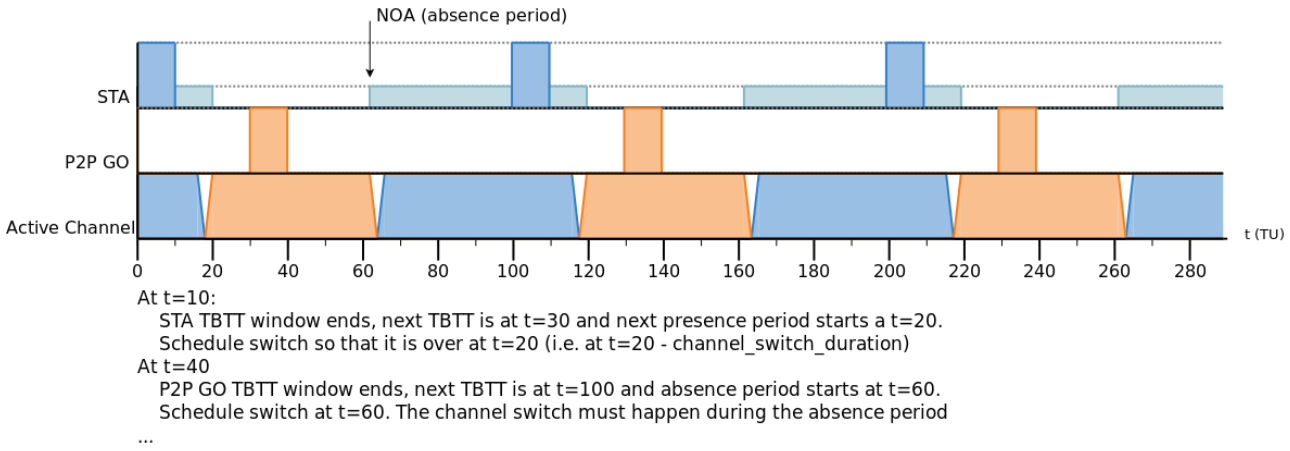



**Figure 59: Example of channel scheduling with a P2P GO interface**

2. **Source and destination context are the same:**
   Here contrary to the 59 case, the switch to and return from the other context is scheduled only if it is necessary to respect NOA.

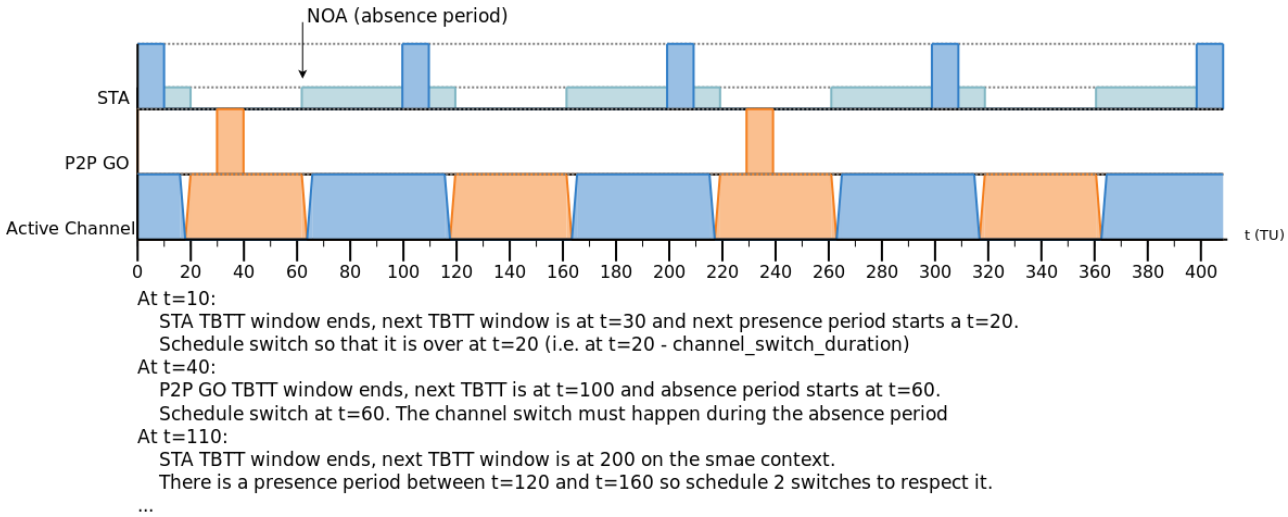



**Figure 60: Example of channel scheduling with a P2P GO interface and two consecutive TBTT window on the same context**

There are some corner cases not detailed here where extra switches may be scheduled to avoid spending too much time on the P2P GO channel during absence period. For example when P2P GO TBTT window happens during an absence period.

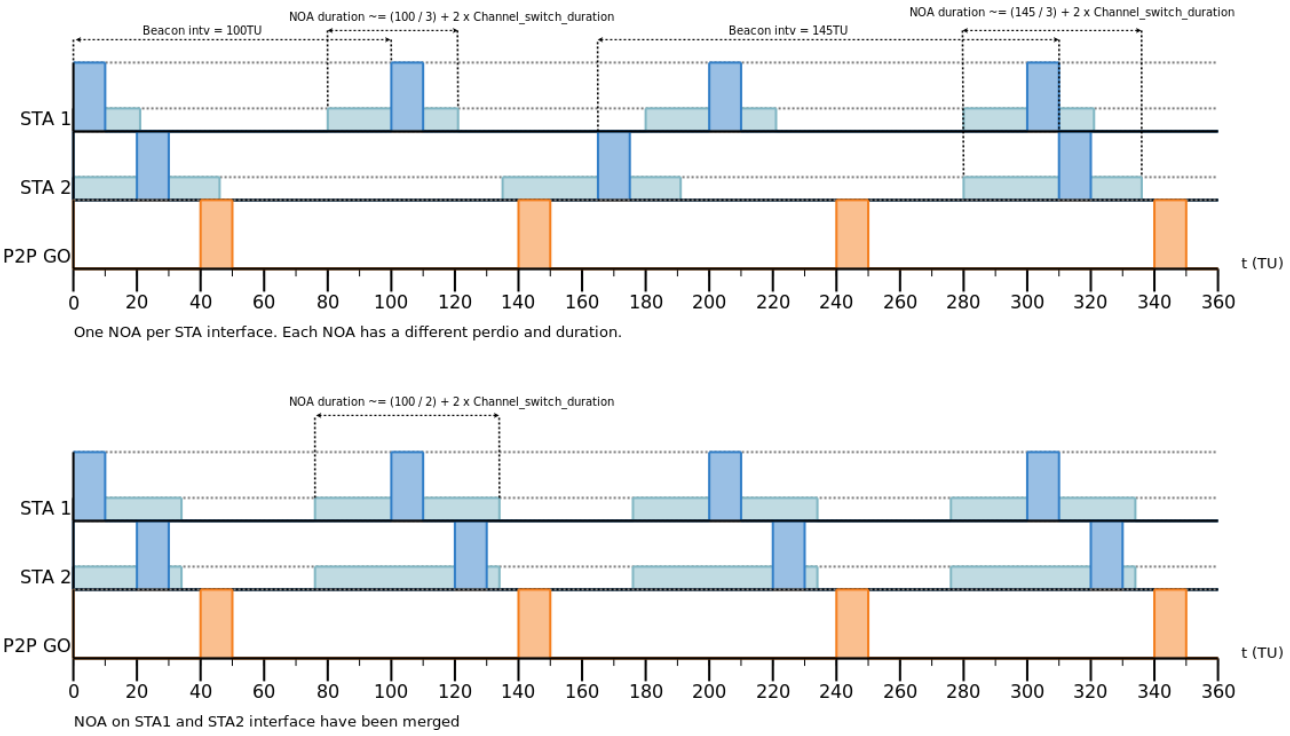
#### 2.4.8.5 Support for P2P GO interface

As P2P GO are the only AP interfaces supported by the Channel Context block they require specific code. Indeed normal AP interfaces cannot be used with multiple active context as they require to be always present on their channel.

Only one P2P GO interface is supported when multiple channel context are active.

When a P2P GO interface is activated, the first thing to do is to start periodic Notice of Absence (NOA) to inform its clients that it won't always be present of the channel. Those NOA will allow switching to the other context to respect TBTT windows and for traffic, and are created as such:

1. One NOA is created per interface on the other context with a period set to the beacon interval on this interface
2. The duration of the NOA is selected so that its absence period represent about 1/(1+nb_NOA) of the time. Also since P2P GO interface must be present during its presence period, the channel switch duration are included in the absence period.
3. When several interface share the same beacon interval and their TBTT windows are close enough, their NOA are merged.




When starting NOA whose period is a multiple of the beacon interval of the P2P GO interface, it is a good idea to ensure that P2P GO TBTT window won't happen during an absence period. To do so it is sometimes needed to move the AP TBTT event in the MAC Hardware.

It is also necessary to check during runtime that NOAs keep aligned with the TBTT window they protect. Indeed the NOA start/stop are handled with an internal timer whereas the date of the TBTT window of STA interface is controlled by the timer of the remote AP. Both timers are not in-sync and they will drift. If this drift grows to the point of moving TBTT window out of the NOA then the first rule mentioned in 2.4.8.4.2 is no longer true leading to incorrect channel switches. To prevent this drift is check at each TBTT window and when it reach a threshold local TSF counter is adjusted to re-align. If the drift reaches a high limit threshold NOAs are re-started.

#### 2.4.8.6 Support for P2P-CLI interface

Currently P2P CLI interface are handled like normal STA interface.

This may lead to non-optimal channel scheduling if the P2P GO of the link started NOAs.

#### 2.4.8.7 Connection less Operations

Two kind of connection less operations exist and are supported by the LMAC: Scan and Remain on Channel (RoC) operations.

During a complete scan procedure, several jumps on the different channels to be scanned are programmed in a row. Scan module is built so that once scan duration for one channel is over, a new scan request is sent for the next channel. In order to avoid spending too much time on non-traffic channels without interruption (and then missing several beacon reception/transmission), some delay has to be inserted between the different non-connected operations (see Figure 61).

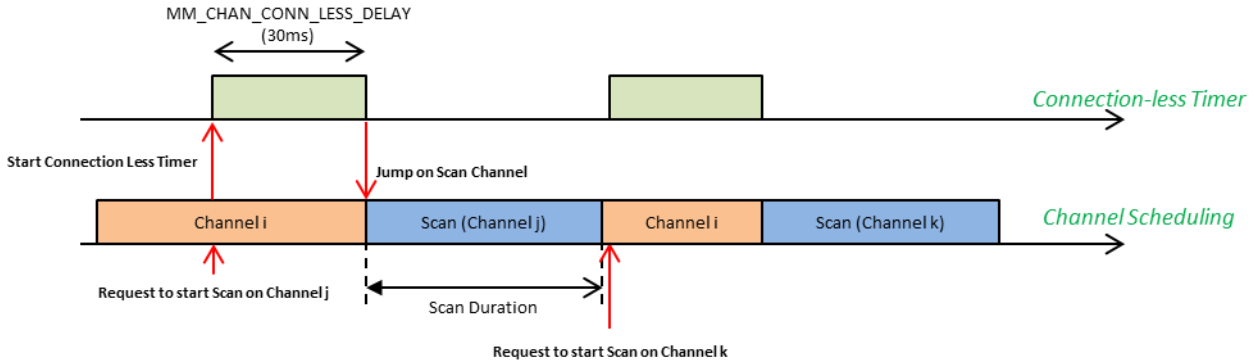



**Figure 61: Illustration of Connection-less timer**

The channel context block will delete all previously scheduled channel switches and skip all TBTT windows that would happen when a connection-less context is active.

### 2.4.8.8 Beacon Detection

A mechanism based on the Remain on Channel (RoC) procedure is implemented in order to spend more time on a given channel in following situations:

- When a new STA VIF is set as active, the time at which it will have to listen for first reception of a Beacon is not known. Hence, it might be tricky to catch the beacon sent by the AP if the bandwidth has to be shared with others VIFs operating on different channels.

  For that purpose, the RoC procedure is used upon VIF activation (if number of active VIF becomes higher than 2) in order to be sure to stay at least one full beacon interval on the channel and thus increase the probability to catch the needed beacon.

  See Figure 62.

- When several beacons have been missed in a row, the RoC procedure is launched in order to detect if beacon transmission time has shifted outside the supposed TBTT presence window.
  Basically the RoC procedure is started one beacon loss before transmission of a keep-alive NULL frame to the AP.

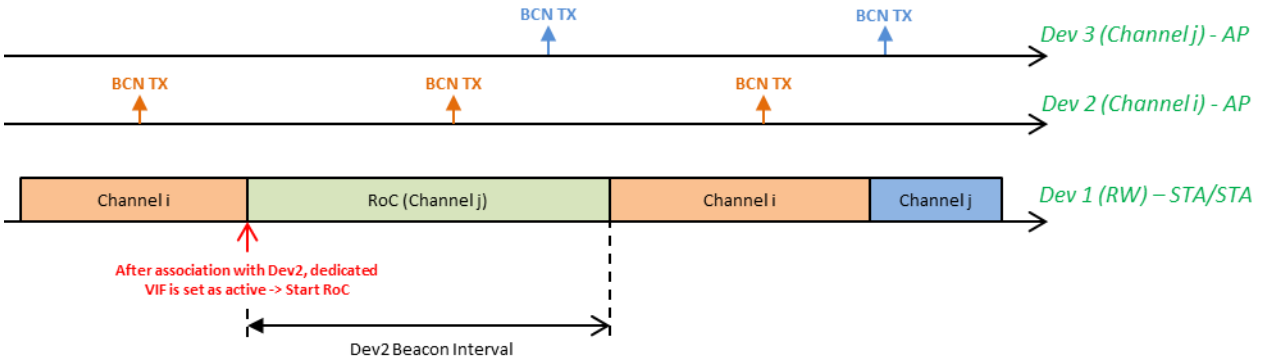



**Figure 62: Beacon Detection upon VIF activation**

### 2.4.8.9 Other channel scheduling example

Here is the channel scheduling of the example presented in 2.4.8.5:

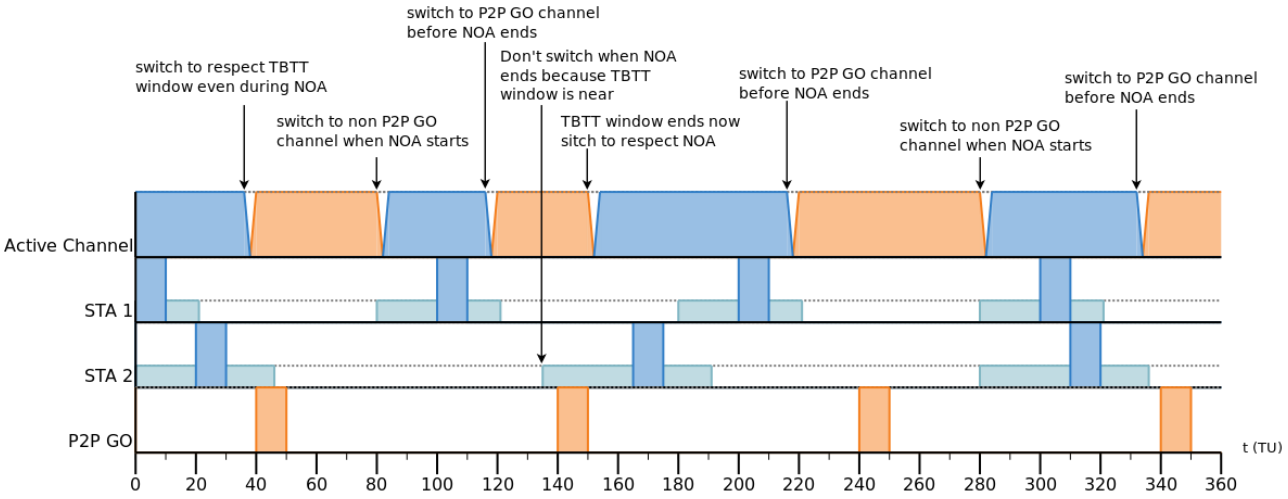



**Figure 63: Channel scheduling example with P2P GO TBTT window during NOA**

And here is another example with two different beacon interval: 100TU for STA1 and 90TU for STA2.

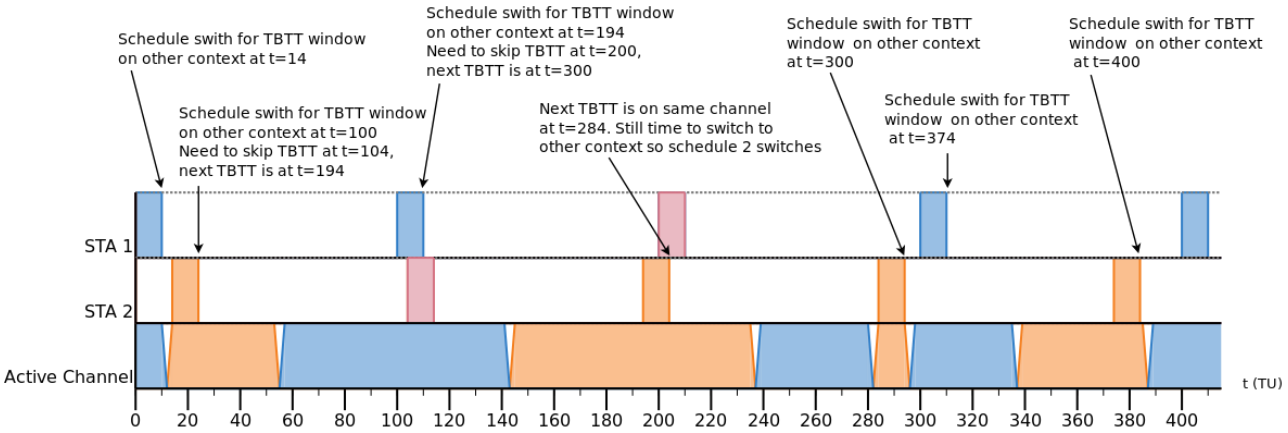



**Figure 64: Another channel scheduling example with different beacon period**

### 2.4.9 Debug and Error Recovery Mechanisms

#### 2.4.9.1 Software Profiling

The LMAC FW (as well as the FMAC FW if the LMAC is associated with the RW UMAC) implements a profiling functionality that allows capturing the real-time operations (RX steps, TX steps, channel context scheduling, etc.) in a Logic Analyzer (either external or internal). To do that the FW makes use of the swProfilingReg of the MAC HW (see [4]. The content of this register can be output as part of the diagnostic port functionality of the MAC HW.

The signals that can be output are defined in file dbg_profiling.h, and the selection of the signals that are enabled (up to 32) is done at compile time. The available signals are described below:

MAC Management procedures:

MM_HW_IDLE: This signal indicates that the FW has requested the MAC HW to move to IDLE state and that the request is pending. This signal moves down when the MAC HW triggers the IDLE interrupt. The move to IDLE is requested for some procedures, such as a RF channel switch, or various MAC HW configurations.

MM_SET_CHANNEL: A RF channel switch is ongoing.

BCN_PRIM_TBTT_IRQ: The primary TBTT interrupt from the MAC HW is currently handled.

BCN_SEC_TBTT_IRQ: The secondary TBTT interrupt from the MAC HW is currently handled.

AP_TBTT: The AP TBTT kernel event is ongoing. It is triggered during the handling of the BCN_PRIM_TBTT_IRQ or BCN_SEC_TBTT_IRQ interrupts. During this event the beacon is programmed for transmission.

STA_TBTT: The STA TBTT kernel event is ongoing. It is triggered by the TBTT timer programmed by the FW to wake up prior to the beacon reception. After the handling of this event the FW prevents from going to DOZE state until the beacon has been received.

TX Path:

TX_IPC_IRQ: The IPC interrupt indicating TX descriptor availability from host is being handled by the FW. The handling of the descriptor(s) will be performed in the TX_IPC_EVT triggered in this interrupt.

TX_IPC_EVT: The TX descriptor IPC kernel event is ongoing. This event includes the background processing done on a packet prior to its transmission (initial TX Header descriptor preparation, aggregation procedure, buffer

TX_BUF_ALLOC: A TX buffer allocation and MPDU payload download programming is ongoing. This processing might then be followed by a TX_DMA_IRQ if some processing is required when the packet is downloaded.

TX_BUF_FREE: A TX buffer freeing is being performed after transmission completion. Some new TX buffers might then be allocated in the freed space.

TX_DMA_IRQ: The interrupt indicating end of DMA transfer for payload download is being executed by the FW.

TX_PAYL_HDL: This signal indicates that the processing performed on the payload after the download is ongoing: assignment of a sequence number in case of non-QoS packet, setting of some fields in the TX descriptor, computing of the TKIP MIC (in FullMAC only).

TX_NEW_TAIL: The downloaded payload is chained to HW using the newTail bit.

TX_MAC_IRQ: The TX MAC HW IRQ is being handled. This interrupt is triggered by the MAC HW after the transmission of a MPDU. It allows the FW freeing the MPDU buffer and allocating a new one (TX_BUF_FREE and TX_BUF_ALLOC are seen in that case). This interrupt is also triggered at the end of an A-MPDU transmission. In such case the AGG_BAR_DONETX signal is supposed to be observed.

TX_CFM_EVT: The TX confirmation event is ongoing. In this event the status of the transmitted packet is checked and uploaded to the upper layers. In case the packet was part of an A-MPDU, the received BlockAck frame is analyzed to know the status of the MPDU.

TX_CFM_DMA_IRQ: The upload of the TX confirmations is completed by the DMA.

TX_FRAME_PUSH: This signal indicates that a WiFi frame generated internally to the FW is programmed for transmission. The frames generated internally are for example the PS-poll frames, the UAPSD QoS-NULL trigger frames or the NULL frames indicating PS state transitions. In FullMAC the frames related to the association to an AP or the management of the BlockAck agreement are also generated internally.

TX_FRAME_CFM: This signal indicates the completion of an internally generated frame.

TX_AC_BG[0 : 1]: This 2-bit signal represents the access category currently processed in the TX path background context. It is therefore valid only when TX_IPC_EVT or TX_CFM_EVT are active.

TX_AC_IRQ[0 : 1]: This 2-bit signal represents the access category currently processed in the TX path interrupt context. It is valid only when TX_DMA_IRQ or TX_MAC_IRQ are active.

Host/Emb message interface:

MSG_IRQ: A message from host interrupt is being handled

MSG_FWD: A message received from host is being forwarded

MSG_IPC_IND: A message from the FW is being indicated to the host

RX Path:

RX_MAC_IRQ: The MAC HW RX interrupt is being executed. Most of the processing done here is for control frames requiring immediate action, such as BlockAck frames received after an A-MPDU transmission. Less critical RX handling is done in the RX_CNTRL_EVT triggered from the interrupt.

RX_CNTRL_EVT: The RX control kernel event is being executed. In this event the following actions might be performed:

- If the frame is the beacon of the AP we are connected, some computation are made on the frame (CRC checking to know if it changed from last beacon received, next TBTT computing, TIM checking for PS mode, etc.)
- Programming of the MPDU upload
- In FullMAC mode, reassembly, duplicate frame filtering, reordering, LLC translation are also done in this event

RX_MPDU_XFER: The received MPDU upload is being programmed.

RX_MPDU_FREE: The received MPDU is being freed. Might be seen for example in STA mode after beacon reception, if the beacon has not changed since last one received and can therefore be freed immediately without

RX_DMA_IRQ: The interrupt indicating completion of the MPDU upload is being handled

RX_DMA_EVT: The kernel event following the MPDU upload is being executed

RX_IPC_IND: The received MPDU is being indicated to the host using an IPC interrupt

TX Aggregation:

AGG_START_AMPDU: A new A-MPDU is started. In this phase the characteristics of the new A-MPDU (maximum length, maximum number of MPDUs, etc.) will be determined based on the PHY rate used and the destination STA.

AGG_ADD_MPDU: An MPDU is added to the A-MPDU under construction

AGG_FINISH_AMPDU: The A-MPDU under construction is closed and can be either chained in the HW immediately (in case enough payloads for this A-MPDU have been downloaded) or later in case we still need payloads to be downloaded.

AGG_FIRST_MPDU_DWNLD: This signal indicates that after this MPDU is downloaded the A-MPDU in which it is included will have enough data available to be chained. In this case the A-MPDU is chained during the TX_PAYL_HDL processing; otherwise it is chained during the AGG_FINISH_AMPDU processing.

AGG_BAR_DONETX: At the end of an A-MPDU transmission, this signal indicates that the BAR THD status has been updated by the MAC HW to indicate if the BA was correctly received or not (either as an immediate response to the A-MPDU or after sending the BAR frame).

AGG_BA_RXED: This signal indicates that a BlockAck frame has been received for the last A-MPDU transmission and can therefore be analyzed by the SW to know which MPDUs are acknowledged.

Bandwidth drop:

The signals below are valid only if the BW drop feature is compiled and if the PHY supports BW greater than 20MHz.

BW_DROP_IRQ: The HW asserted the BW drop interrupt and the FW is currently handling it. This interrupt is triggered by the HW when a >20MHz BW transmission has been programmed by the FW upon the following conditions:

- The effective BW available on the air interface is lower than the programmed one.
- The duration of the A_MPDU transmitted at the effective BW would be greater than the maximum allowed

If both the above conditions are met then the FW needs to split the programmed A-MPDU so that its length fits into the available BW.

BW_DROP_STEP: During the A-MPDU creation this signal toggles to indicate that the current MPDU added to the A-MPDU marks the beginning of a new BW step. During the process of creation of an A-MPDU (i.e between the AGG_AMPDU_START and AGG_AMPDU_FINISH events), the meaning of this signal is therefore:

1st BW_DROP_STEP event – The subsequent MPDUs added will be transmitted in the A-MDPU only if the effective TX BW is at least 40MHz

2nd BW_DROP_STEP event – The subsequent MPDUs added will be transmitted in the A-MPDU only if the effective TX BW is at least 80MHz

3rd BW_DROP_STEP event – The subsequent MPDUs added will be transmitted in the A-MDPU only if the effective TX BW is at least 160 or 80+80MHz

Power-Save mode:

PS_SLEEP: This signal is set when the PS module allows the system moving to doze state, and reset when doze is not allowed for any reason (ongoing scan, beacon not yet received after TBTT, UAPSD service period ongoing, packets ready for transmission in the TX path, etc.)

PS_PAUSE: This signal indicates when the PS mode is temporarily disabled by the Dynamic Power Save Mode module. The system is in active state as long as this signal is set.

PS_DPSM_UPDATE: This signal is active when the DPSM module is updating the PS state of the system. The PS_PAUSE signal is set/reset during this update procedure.

PS_CHECK_RX: This signal is active when the PS module is checking a received data frame in order to take decisions on the behavior to follow with regards to doze state: check on the EOSP bit in case of UAPSD, check on the More Data bit, update of the Traffic Detector, etc. Based on the flags set or reset during this process, the doze state can then be allowed or disallowed.

PS_CHECK_TX: This signal is active when the PS module is informed of a data frame transmission. During this process the Traffic Detector is updated and the PS module verifies if the transmitted frame marks the beginning of a UAPSD service period. Based on the PS flags set or reset during this process, the doze state can then be allowed or disallowed.

PS_CHECK_BCN: This signal is active when the PS module is checking the content of the TIM information element of the beacon in order to verify if some buffered data is available at the AP side. The collected information may then trigger the transmission of a PS-poll (for legacy PS) or QoS-NULL trigger frame (for UAPSD) in order to retrieve the data. The PS flags set/reset during this process can then disallow/allow the doze state.

Traffic Detector:

TD_CHECK_RX: This signal is active when the TD is informed of a new frame reception. The corresponding packet counter is updated accordingly.

TD_CHECK_TX: This signal is active when the TD is informed of a new packet transmission. The corresponding packet counter is updated accordingly.

TD_CHECK_RX_PS: This signal is active when the TD is informed of a new frame reception, while in PS mode. The corresponding packet counter is updated accordingly.

TD_CHECK_TX_PS: This signal is active when the TD is informed of a new frame transmission, while in PS mode. The corresponding packet counter is updated accordingly.

TD_TIMER_END: This signal is active when the TD timer is handled. In this event the different packet counters are checked and the traffic state of the different interface is updated accordingly.

Channel Context Manager:

CHAN_CTXT_CDE_EVT: This signal is active when the CDE timer is handled in the chan module so when the time allocated for each channel context is reset. The CDE timer is used when at least two channels context are used in parallel (RoC and Scan channels excluded). CDE timer duration is 50ms multiplied by number of used channel contexts.

CHAN_CTXT_TBTT_SWITCH: This signal is set when the TBTT switch timer is handled hence when the chan module requests to switch on a channel for TBTT reasons meaning in order to receive or transmit a beacon on a given channel.

CHAN_CTXT_IDX: This signal indicates the index of the channel context index currently in use.

CHAN_CTXT_TX_DISCARD: This signal indicates that a frame to be transmitted has been discarded because the current channel is not the one on which we have to send this frame.

CHAN_CTXT_WAIT_END: This signal is active when CDE is used (so when more than one channel context are used) and when the timer allowing monitoring the time remaining for a channel is active.

CHAN_CTXT_SWITCH: This signal is set when the system is switching from a channel to another. It includes the whole pre-switch procedure in which HW is required to enter in IDLE mode in order to finish all the pending RX/TX operations and the PHY configuration for the new channel. MM_SET_CHANNEL should be set when CHAN_CTXT_SWITCH is reset.

CHAN_CTXT_TBTT_PRES: This signal is set when the system is a present on a channel for TBTT reasons.

Peer2Peer:

P2P_NOA_0_ABS: This signal is active during an absence period announced by the P2P GO device in the first NoA indicated in its NoA Information Element. It allows verifying that the interval and duration values set in the beacon are well respected.

P2P_NOA_1_ABS: This signal is active during an absence period announced by the P2P GO in the second NoA indicated in its NoA Information Element.

P2P_CTW: The signal is active when the SW is waiting for the end of the CT Window after the TBTT has occurred. During this period, the system shall be in active mode.

P2P_WAIT_BCN: The signal is active when the P2P module is waiting for the beacon reception or end of the beacon transmission confirmation from HW.

P2P_ABSENCE: This signal is active when the P2P module considers that the GO of the P2P connection is absent. If supported, the system should be mainly in doze mode during this period.

P2P_PS_PAUSED: The signal is active when acting as P2P GO and P2P module has been informed by TD that traffic was generated on the link. In that case, NoA status is maintained but it is no more used and no more indicated in the beacon until the end of traffic is announced by TD.

Radar Detection:

RADAR_IRQ: The PHY asserted an interrupt indicating that detected radar pulse(s) are available in the FIFO.

Additionally to the SW profiling signals described above, some HW signals are also very useful to capture in order to see the state of the MAC HW and PHY in parallel to the SW processing:

MAC/PHY interface:

The MAC/PHY interface is as its name indicates the interface between the MAC HW and the PHY. It is therefore very useful to monitor when the system in reception or transmission. It can also be used to monitor the data currently transmitted/received as well as the TX/RX vectors.

mpif_rxReq: This signal, asserted by the MAC HW to the PHY, requests to the PHY to be in listen/receive state, i.e. to allow frame reception.

mpif_ccaPrimary and mpif_ccaSecondary: These signals, asserted by the PHY to the MAC HW, indicates that the AGC/CCA HW block has detected a packet on the primary and secondary channels, respectively. These signals are therefore showing when a packet reception is ongoing.

mpif_rxEnd: This signal is triggered by the PHY to indicate to the MAC that a packet reception is finishing.

mpif_txReq: This signal is triggered by the MAC to indicate to the PHY to switch to TX and proceed to a transmission. It is active during the whole duration of the transmission.

The figure below shows an example of logic analyzer capture including SW profiling, MACPHY IF and MAC HW signals recorded in parallel:

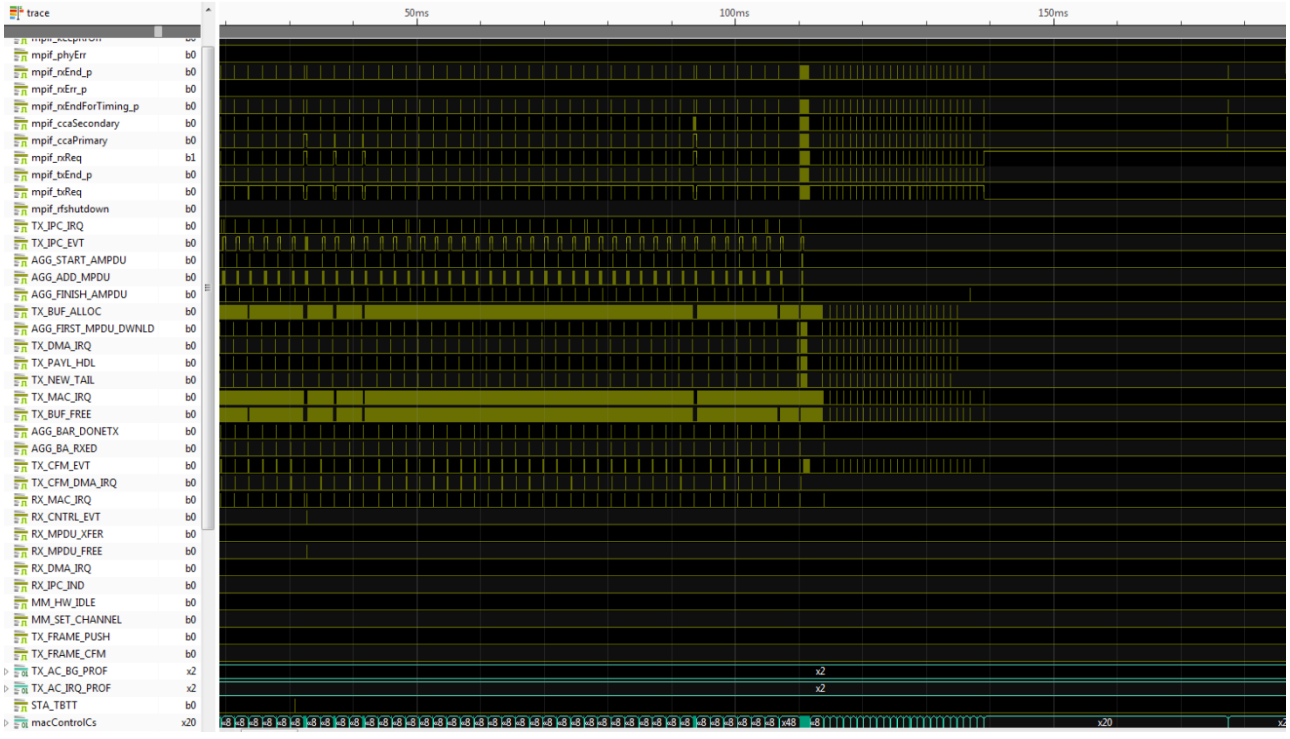



**Figure 65: Example of SW and HW profiling signals captured by a logic analyzer**

#### 2.4.9.2 Error Detection

The LMAC FW (as well as the FMAC FW if the LMAC is associated with the RW UMAC) is able to detect unexpected errors. These errors are typically coming from the HW (MAC or PHY), indicating that a TX or RX operation has failed for some reason (TX underruns, PHY errors, TX DMA dead errors, etc.). The possible errors are described in [4]. In case such an error occurs, the HW triggers an interrupt to the FW.

Other errors are detected by the FW itself, e.g. packets blocked in TX, inconsistencies in the FW states and/or internal variables, etc.

The errors checks are handled using assertion macros (ASSERT_xxx) that can be seen at many places in the code.

The detected errors are classified into two categories: Recoverable or fatal errors. An error is considered as "recoverable" when a HW reset and/or a flush of the data queues can put back the system in a safe state. An error is considered as "fatal" if it shows an inconsistency in the FW state or some illegal variable values that could be caused by a memory corruption or a similar issue.

The fatal errors are detected using the ASSERT_ERR macros, while the recoverable ones are detected using the ASSERT_REC… macros. The errors are indicated to the host CPU using the DBG_ERROR_IND message.

Note that erroneous behaviors of peer devices (e.g. malformed frame received, expected acknowledgment not received, etc.) are not considered as errors and do not trigger any recovery or system reset. This kind of errors is handled as part of the normal operation of the system.

#### 2.4.9.3 Recovery Mechanism

A recovery mechanism is implemented in the LMAC SW in order to allow the system to recover smoothly in case of error detection. This mechanism is invoked when the detected error is considered as "recoverable", as explained above.

The recoverable errors are detected in ASSERT_REC… macros. These macros typically catch the error type for later debug dump, set the reset kernel event, disable the HW interrupts and return immediately from the function where they are called.

The reset kernel event is then executed by the kernel in background. This event has the highest priority, i.e. no other kernel event will be executed before the reset event. This event will perform the following actions:

- If debug dump is enabled (see below), dump the state of the different HW signals
- Reset the HW state machines (register values are not impacted)
- If debug dump is enabled, prepare the debug buffer to be dumped to the host CPU
- Reset the RX path: This operation consists in uploading the already received frame, and re-initialize the descriptor lists chained to the HW
- Reset the TX path: This operation consists in flushing the TX path and indicates that the programmed packets were not transmitted.

#### 2.4.9.4 Debug dump

In case of error detection (that can be either fatal or recoverable), a debug dump can be forwarded by the LMAC FW to the host CPU. This debug dump is mostly composed of the following information:

- The TX descriptor lists chained to the HW at the time of the error detection (one list per HW queue). This is useful to analyze some HW errors, by getting information on programmed packets that may have triggered the error.
- The RX descriptor lists (Header and Payload descriptors) currently chained to the HW.
- The channel on which the system was operating when the error occurred

Additionally to the previous information, the dump coming from the embedded logic analyzer is also forwarded to the host CPU, in case this HW block is implemented in the system (typically in FPGA validation, but can be done also on ASIC).

# 3 Supported Topologies

The following table resumes the topologies that are supported or not by RW LMAC when one or two VIF are registered.

| Role VIF 0 | Role VIF 1 | Same Channel | Supported |
|---|---|---|---|
| **AP** | **-** | - | **YES** |
| **STA** | **-** | - | **YES** |
| **GO** | **-** | - | **YES** |
| **CLI** | **-** | - | **YES** |
| **STA** | **STA** | YES | **YES** |
| **STA** | **STA** | NO | **YES** |
| **STA** | **AP** | YES | **YES** |
| **STA** | **AP** | NO | **NO** |
| **STA** | **GO** | YES | **YES** |
| **STA** | **GO** | NO | **YES** |
| **STA** | **CLI** | YES | **YES** |
| **STA** | **CLI** | NO | **YES** |
| **AP** | **AP** | YES | **NO** |
| **AP** | **AP** | NO | **NO** |
| **AP** | **GO** | YES | **NO** |
| **AP** | **GO** | NO | **NO** |
| **AP** | **CLI** | YES | **YES** |
| **AP** | **CLI** | NO | **NO** |
| **GO** | **GO** | YES | **NO** |
| **GO** | **GO** | NO | **NO** |
| **GO** | **CLI** | YES | **YES** |
| **GO** | **CLI** | NO | **YES** |
| **CLI** | **CLI** | YES | **YES** |
| **CLI** | **CLI** | NO | **YES** |

**Table 7: Supported Topologies**

# References

[1] Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications, IEEE P802.11-2012

[2] Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: Enhancements for Very High Throughput for Operation in Bands below 6 GHz, IEEE P802.11ac™-2013

[3] RW-WLAN-nX-LMAC-SW User Manual Document (RW-WLAN-nX-LMAC-SW-UM/1.11)

[4] RW-WLAN-nX-MAC-HW User Manual Document (RW-WLAN-nX-MAC-HW- UM/3.03)

[5] RW-WLAN-nX-LMAC-SW Developer Guide Document (RW-WLAN-nX-LMAC-SW-DG/1.08)

[6] RW-KERNEL-SW Functional Specification Document (RW-KERNEL-SW-FS/1.1)

[7] Wi-Fi P2P Technical Specification v1.1