# ESP-IDF Mesh Stack Practical Guide

ESP-IDF V3.1

# ESP-IDF Mesh Stack Practical Guide

The purpose of this guide is to provide a step-by-step tutorial for developing a mesh application using the mesh stack of ESP32-based boards, and it represents a supplement to the Mesh API Reference and the Mesh API Guidelines that can be found in the ESP-IDF Programming Guide.

## License



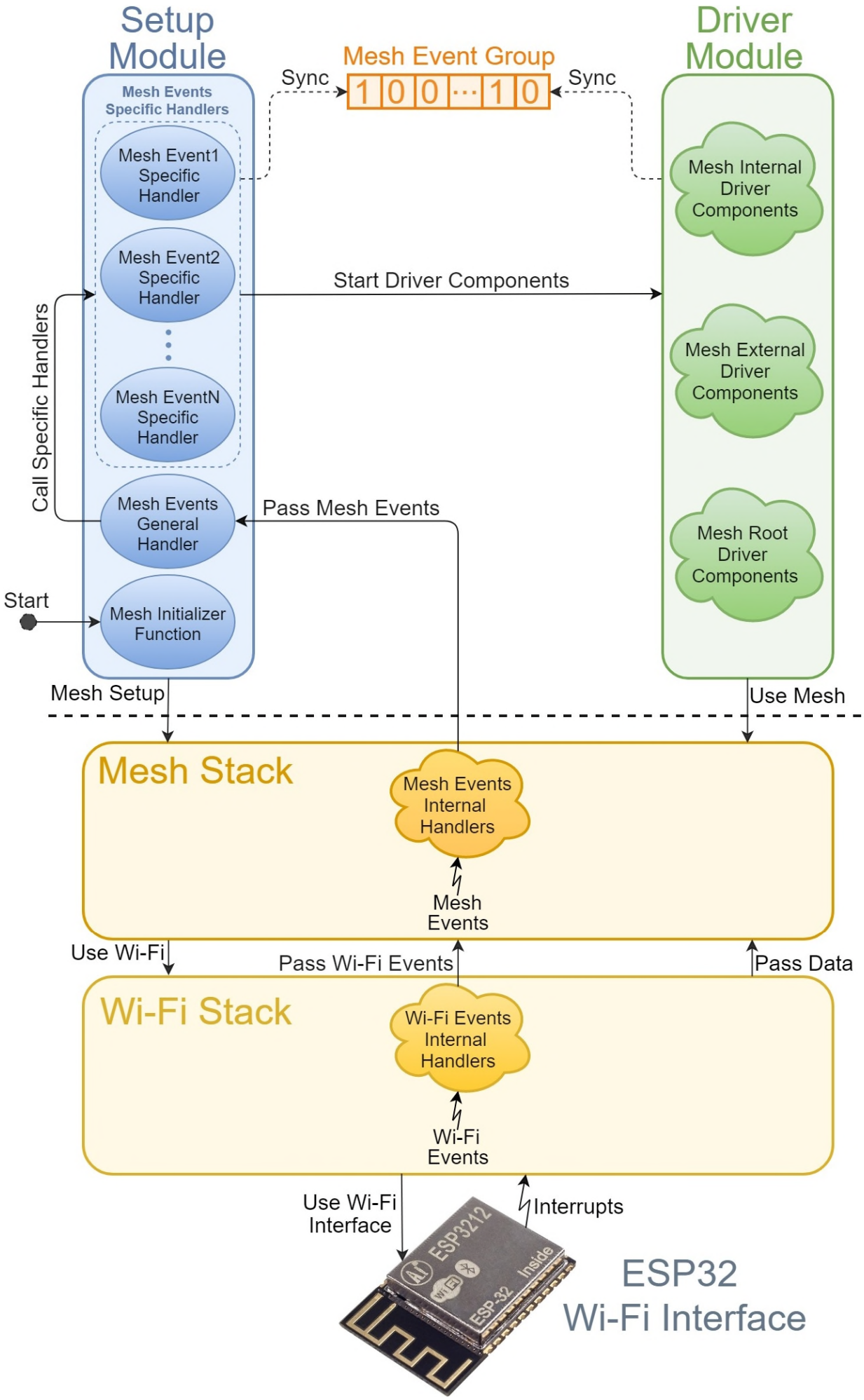
## Requirements

To fully benefit from the contents of this guide, prior knowledge of the following topics is recommended:

- ESP-IDF Wi-Fi Basics (API reference and guidelines)
- ESP-IDF projects authoring (guide)
- ESP32 application startup flow (guide)
- ESP32 logging mechanism (guide)
- FreeRTOS tasks basics (guide)
- FreeRTOS event groups (guide)

# Structure of a Mesh Application

An application which uses the mesh stack of ESP32 boards can generally be divided into the following components:



The collection of functions and data structures provided by the ESP-IDF to offer mesh networking functionalities represents the **Mesh Stack**, which is implemented on top and uses the functionalities offered by the **Wi-Fi Stack**, which directly manages the device's Wi-Fi interface.

From here at the application level a mesh application can be divided into the following two modules:

## Setup Module

The **Setup Module**, whose purpose is to carry out the setup process of mesh networking on the device, consists of the following components:

- A **Mesh Initializer Function**, which represents the entry point of a mesh application and whose tasks are to initialize and configure the Wi-Fi and Mesh stacks and to enable mesh networking on the device.

- The **Mesh Events General Handler**, which is a function called asynchronously by the mesh stack after its internal handlers each time a mesh event occurs and whose task is to pass each event with its provided additional information to its relative specific handler.

- A set of **Mesh Events Specific Handlers**, which are functions whose tasks are to perform the application-level handling of mesh events raised by the mesh stack and to start driver components when the appropriate requirements for their execution are met.

## Driver Module

The **Driver Module** represents the part of the application that utilizes the features offered by the mesh stack to implement a certain service, and while its actual logic and structure depend on its purpose, its components can be generally divided into the following categories according to the mesh networking features they use, and so the requirements that must be met before their execution can begin:

- The **Internal Driver Components** are components that require the device to be connected to a mesh network but don't require to communicate with the external Distribution System (DS), and so they can be started as soon as the device connects to a parent in the mesh network.

- The **External Driver Components** are components that require the device to communicate with the external DS, and so they can be started as soon as the device is informed of the accessibility of the external DS.

- The **Root Driver Components** represent components that are to be executed exclusively on the root node of the mesh network, and they can be started as soon as the root node is able to communicate with the external router at the IP level.

The inter-task synchronization between the two modules is performed by using an **Event Group**, which represents an abstract type offered by the FreeRTOS kernel consisting of an array of bits which can be used as semaphores via the relative API.

# Code Premises

## Required Headers

The minimal subset of libraries required to develop a mesh application, with their paths relative to the $ESP-IDF environment variable, is as follows:

```
/*-- C standard libraries --*/
#include <string.h>                         //C standard string library

/*-- Environment-specific libraries --*/
#include "esp_system.h"                     //ESP32 base system library
#include "esp_log.h"                        //ESP32 logging library
#include "nvs_flash.h"                      //ESP32 flash memory library
#include "esp_wifi.h"                       //ESP32 main Wi-Fi library
#include "esp_event_loop.h"                 //ESP32 Wi-Fi events library
#include "esp_mesh.h"                       //ESP32 main Mesh library
#include "freertos/FreeRTOS.h"              //FreeRTOS base library
#include "freertos/task.h"                  //FreeRTOS tasks library
#include "freertos/event_groups.h"          //FreeRTOS event groups library
#include "lwip/sockets.h"                   //LwIP base library
```

## Type Definitions

The following additional custom data types are used in the context of this guide for developing a mesh application, whose members and uses are described thoroughly in the following sections:

```
typedef enum                                //Mesh Organization Mode
 {
  SELF_ORGANIZED,                           //Self-Organized Networking
  FIXED_ROOT,                               //Fixed-Root Networking
  MANUAL_NETWORKING                         //Manual Networking
 } mesh_org_t;

typedef struct                              //Describes the status of a
 {                                            node in the mesh network
  uint8_t mid[6];                           //Mesh Network Identifier (MID)
  uint8_t channel;                          //Mesh Wi-Fi Channel
  mesh_org_t org;                           //Mesh Organization Mode
  uint8_t root_addr[6];                     //Root node's SoftAP MAC address
  uint8_t parent_addr[6];                   //Node's parent SoftAP MAC address
  mesh_type_t my_type;                      //Node's type
  int8_t my_layer;                          //Node's layer in the mesh network
 } mesh_status_t;
```

## Global Variables

The following global variables are used in this guide for developing a mesh application:

```
EventGroupHandle_t mesh_event_group;        //Mesh Event Group Handler

/*-- Mesh Setup Flags --*/
#define MESH_ON       BIT0      //Whether mesh networking is enabled on the node or not
#define MESH_PARENT   BIT1      //Whether the node is connected to a parent or not
#define MESH_CHILD    BIT2      //Whether the node has children connected or not
#define MESH_TODS     BIT3      //Whether the external DS is reachable or not
#define MESH_VOTE     BIT4      //Whether a new root election is currently in progress
                                  in the mesh network or not (this flag is used active
                                  low: 0 = election in progress, 1 = no election)
```

```
/*-- Driver Components Executions Flags --*/  //Indicate whether each driver component
                                                 is running or not (where here a single
                                                 component for each of the three driver
                                                 categories is used)
#define MESH_DRIVER_INTERNAL  BIT5    //Whether the internal drivers are running or not
#define MESH_DRIVER_EXTERNAL  BIT6    //Whether the external drivers are running or not
#define MESH_DRIVER_ROOT      BIT7    //Whether the root drivers are running or not

mesh_status_t mesh_state;     //Describes the node's current status in the mesh network
```

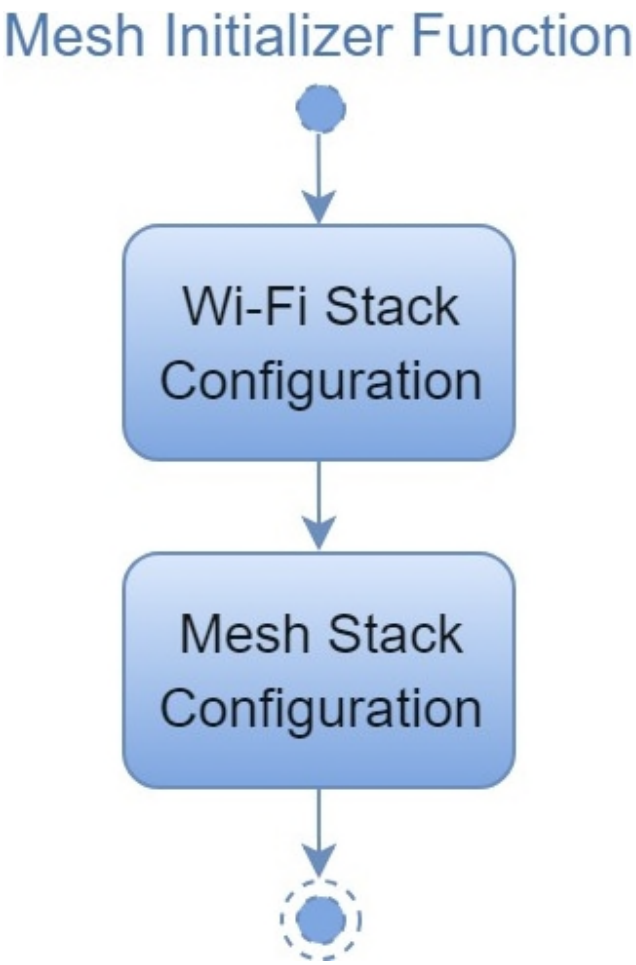
Also note that within this guide the parameters used in the code examples are shown in capital letters and represent predefined constants (e.g. `MESH_ROUTER_SSID`, `MESH_ROOT_IPSTATIC`, etc.), whose values in an actual project can be set for example by providing an appropriate *Kconfig.projbuild* configuration file and using the *menuconfig* utility.

# Setup Module

The setup process of a mesh application is divided into a first phase relative to the inizializations and configurations required to enable mesh networking on a device, which are carried out by the **Mesh Initializer Function**, and a second event-driven phase represented by the application-level handling of the mesh events raised by the mesh stack, which are carried out by the **Mesh Events General Handler** and the set of **Mesh Events Specific Handlers**.

# Mesh Initializer Function

The Mesh Initializer Function represents the entry point of a mesh application, and its purposes are to initialize and configure the Wi-Fi and Mesh Stacks on the device and then enable mesh networking,




## Wi-Fi Stack Configuration

The tasks that must be performed to configure the device's Wi-Fi stack for the purposes of mesh networking are similiar to its standard configuration process, with the following differences due to the fact that its management is not carried out directly at the application level but is devolved almost entirely to the mesh stack.

- It's not necessary to provide a Wi-Fi event group to synchronize tasks on the state of the Wi-Fi Stack.
- The address of the function that the Wi-Fi stack will call for the handling of Wi-Fi events must be set to NULL to later devolve such handling to the mesh stack.
- It's not necessary to directly configure or enable the Wi-Fi interface modes (Station or SoftAP).
- Since mesh networking operates at the data-link level, it's not necessary to set IP configurations for the device Station and SoftAP interfaces, and furthermore to ensure that they don't obtain or offer dynamic IP configurations, the station DHCP client and the SoftAP DHCP server must be preemptively disabled.
  Note that the only exception to this rule is given by the node that will become the root of the mesh network, whose station interface requires an IP configuration to allow it to connect to the external router (we'll see later).

From here the steps required to configure the Wi-Fi stack for the purposes of mesh networking are:

## 1) Initialize the system flash storage

By default the Wi-Fi stack stores its configuration in the system's flash memory, which consequently needs to be initialized beforehand by using the following function:

```
//File nvs.flash.h

esp_err_t nvs_flash_init(void)
```

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_NVS_NO_FREE_PAGES | The flash storage was initialized, but contains no empty pages |
| ESP_ERR_NOT_FOUND | The "nvs" partition was not found in the partition table |
| ESP_FAIL | Unknown error in the storage driver |

The ESP_ERR_NVS_NO_FREE_PAGES represents a recoverable error, whose recovery can be attempted by completely erasing the flash memory through the following function and then trying to initialize it again:

```
//File nvs.flash.h

esp_err_t nvs_flash_erase(void)
```

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_NOT_FOUND | The "nvs" partition was not found in the partition table |
| ESP_FAIL | Unknown error in the storage driver |

No errors in this function can be recovered, so the initialization procedure of the system flash storage appears as below:

Example

```
esp_err_t mesh_init()
 {
  /*-- Wi-Fi Stack Configuration --*/
  esp_err_t ret = nvs_flash_init();         //Flash storage initialization
  if(ret == ESP_ERR_NVS_NO_FREE_PAGES)      //Error recovery attempt
   {
    ESP_ERROR_CHECK(nvs_flash_erase());
    ret = nvs_flash_init();
   }
  ESP_ERROR_CHECK(ret);  //At this point if errors persist, it is not
  …                        possible to proceed with the program's execution
 }
```

## 2) Initialize the TCP/LwIP Stack

Next we need to initialize the data structures relative to the TCP/LwIP stack and create the core LwIP task, which can be obtained by calling the following function:

```
//File tcpip_adapter.h (automatically included by the previous headers)

void tcpip_adapter_init(void)
```

Note that this step is always required even if the TCP/LwIP stack in a mesh network is used only on the root node on its station interface to connect with the external router.

Example

```
esp_err_t mesh_init()
 {
  …
  tcpip_adapter_init();   //Initialize the TCP/LwIP stack
  …
 }
```

## 3) Devolve the Handling of Wi-Fi Events

To devolve the handling of Wi-Fi events, which will be later performed by the mesh stack, the following function must be called with two NULL arguments:

```
//File esp_event_loop.h

esp_err_t esp_event_loop_init(system_event_cb_t cb, void* ctx)
```

| Parameters | |
|---|---|
| cb | The memory address of the Wi-Fi Events General Handler function (in this case NULL) |
| ctx | Reserved for the user (in this case NULL) |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_FAIL | Unknown error in the Wi-Fi stack |

Example

```
esp_err_t mesh_init()
 {
  …
  ESP_ERROR_CHECK(esp_event_loop_init(NULL,NULL));  //Devolve the handling
                                                      of Wi-Fi events to
  …                                                   the Mesh Stack
 }
```

## 4) Initialize the Wi-Fi Stack

Next we need to initialize the Wi-Fi stack, which is obtained by calling the following function:

```
//File esp_wifi.h

typedef struct              //Wi-Fi stack Initialization Parameters
 {
  system_event_handler_t  event_handler;       //Wi-Fi event handler
  wifi_osi_funcs_t*       osi_funcs;           //Wi-Fi OS functions
  wpa_crypto_funcs_t      wpa_crypto_funcs;    //Wi-Fi station crypto functions
  int                     static_rx_buf_num;   //Wi-Fi static RX buffer number
  int                     dynamic_rx_buf_num;  //Wi-Fi dynamic RX buffer number
```

```
    int                     tx_buf_type;          //Wi-Fi TX buffer type
    int                     static_tx_buf_num;    //Wi-Fi static TX buffer number
    int                     dynamic_tx_buf_num;   //Wi-Fi dynamic TX buffer number
    int                     csi_enable;           //Wi-Fi CSI enable flag
    int                     ampdu_rx_enable;      //Wi-Fi AMPDU RX enable flag
    int                     ampdu_tx_enable;      //Wi-Fi AMPDU TX enable flag
    int                     nvs_enable;           //Wi-Fi NVS flash enable flag
    int                     nano_enable;          //printf/scan family enable flag
    int                     tx_ba_win;            //Wi-Fi Block Ack TX window size
    int                     rx_ba_win;            //Wi-Fi Block Ack RX window size
    int                     wifi_task_core_id;    //Wi-Fi Task Core ID
    int                     magic;                //Wi-Fi init magic number
} wifi_init_config_t;
```

```
esp_err_t esp_wifi_init(const wifi_init_config_t* config)
```

| Parameters | |
|---|---|
| config | The address of the struct holding the inizialization parameters of the Wi-Fi Stack |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_NO_MEM | Out of memory |
| ESP_FAIL | Unknown error in the Wi-Fi stack |

For the purposes of mesh networking we can initialize the Wi-Fi Stack with its default parameters by using the following macro:

```
//File esp_wifi.h

#define WIFI_INIT_CONFIG_DEFAULT()                                  \
{                                                                   \
 .event_handler = &esp_event_send,                                  \
 .osi_funcs = &g_wifi_osi_funcs,                                    \
 .wpa_crypto_funcs = g_wifi_default_wpa_crypto_funcs,               \
 .static_rx_buf_num = CONFIG_ESP32_WIFI_STATIC_RX_BUFFER_NUM,       \
 .dynamic_rx_buf_num = CONFIG_ESP32_WIFI_DYNAMIC_RX_BUFFER_NUM,     \
 .tx_buf_type = CONFIG_ESP32_WIFI_TX_BUFFER_TYPE,                   \
 .static_tx_buf_num = WIFI_STATIC_TX_BUFFER_NUM,                    \
 .dynamic_tx_buf_num = WIFI_DYNAMIC_TX_BUFFER_NUM,                  \
 .csi_enable = WIFI_CSI_ENABLED,                                    \
 .ampdu_rx_enable = WIFI_AMPDU_RX_ENABLED,                          \
 .ampdu_tx_enable = WIFI_AMPDU_TX_ENABLED,                          \
 .nvs_enable = WIFI_NVS_ENABLED,                                    \
 .nano_enable = WIFI_NANO_FORMAT_ENABLED,                           \
 .tx_ba_win = WIFI_DEFAULT_TX_BA_WIN,                               \
 .rx_ba_win = WIFI_DEFAULT_RX_BA_WIN,                               \
 .wifi_task_core_id = WIFI_TASK_CORE_ID,                            \
 .magic = WIFI_INIT_CONFIG_MAGIC                                    \
};
```

Example

```
esp_err_t mesh_init()
 {
  …
   wifi_init_config_t initcfg = WIFI_INIT_CONFIG_DEFAULT();
   ESP_ERROR_CHECK(esp_wifi_init(&initcfg));    //Initialize the Wi-Fi Stack
  …
 }
```

## 5) Disable Station DHCP Client and SoftAP DHCP Server

At this point we must disable the Station interface DHCP client and SoftAP interface DHCP server, which is obtained by calling the following functions:

```
//File tcpip_adapter.h

typedef enum                    //TCP/IP interface enumerates
 {
  TCPIP_ADAPTER_IF_STA = 0,   //Station interface
  TCPIP_ADAPTER_IF_AP,        //SoftAP interface
  TCPIP_ADAPTER_IF_ETH,       //Ethernet interface
  TCPIP_ADAPTER_IF_MAX,
 } tcpip_adapter_if_t;
```

```
esp_err_t tcpip_adapter_dhcpc_stop(tcpip_adapter_if_t tcpip_if)
```

| Parameters | |
|---|---|
| tcpip_if | The interface where to stop the DHCP client<br>❑ TCPIP_ADAPTER_IF_STA → Station interface |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_TCPIP_ADAPTER_IF_NOT_READY | The Wi-Fi stack is not initialized (call esp_wifi_init()first) |
| ESP_ERR_TCPIP_ADAPTER_DHCP_ALREADY_STOPED | The DHCP client on the interface was already disabled |
| ESP_ERR_TCPIP_ADAPTER_INVALID_PARAMS | Invalid argument(s) |
| ESP_FAIL | Unknown error in the Wi-Fi stack |

```
esp_err_t tcpip_adapter_dhcps_stop(tcpip_adapter_if_t tcpip_if)
```

| Parameters | |
|---|---|
| tcpip_if | The interface where to stop the DHCP server<br>❑ TCPIP_ADAPTER_IF_AP → SoftAP interface |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_TCPIP_ADAPTER_IF_NOT_READY | The Wi-Fi stack is not initialized (call esp_wifi_init()first) |
| ESP_ERR_TCPIP_ADAPTER_DHCP_ALREADY_STOPED | The DHCP server on the interface was already disabled |
| ESP_ERR_TCPIP_ADAPTER_INVALID_PARAMS | Invalid argument(s) |
| ESP_FAIL | Unknown error in the Wi-Fi stack |

Example

```
esp_err_t mesh_init()
 {
  …
  ESP_ERROR_CHECK(tcpip_adapter_dhcpc_stop(TCPIP_ADAPTER_IF_STA));
  ESP_ERROR_CHECK(tcpip_adapter_dhcps_stop(TCPIP_ADAPTER_IF_AP));
  …
 }
```

## 6) Enable the Wi-Fi Interface

The last step of the Wi-Fi configuration is to enable the Wi-Fi interface itself, which is obtained by calling the following function:

```
//File esp_wifi.h

esp_err_t esp_wifi_start(void)
```

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_WIFI_NOT_INIT | The Wi-Fi stack is not initialized (call esp_wifi_init()first) |
| ESP_ERR_WIFI_CONN | Wi-Fi internal error, station or SoftAP control block wrong |
| ESP_ERR_NO_MEM | Out of memory |
| ESP_ERR_INVALID_ARG | Invalid argument(s) |
| ESP_FAIL | Unknown error in the Wi-Fi stack |

Example

```
esp_err_t mesh_init()
 {
  …
  ESP_ERROR_CHECK(esp_wifi_start());    //Enable the Wi-Fi interface
  …
 }
```

## Mesh Stack Configuration

Once the Wi-Fi interface on the device has been enabled the Mesh Stack configuration can commence, which is performed through the following steps:

### 1) Initialize the Mesh Stack

The mesh stack on a device can be initialized by calling the following function:

```
//File esp_mesh.h

esp_err_t esp_mesh_init(void)
```

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_FAIL | Unknown error in the mesh stack |

Note that once the mesh stack has been initialized on a device we should refrain from directly modifying settings related to the Wi-Fi stack, since such changes may interfere with mesh networking functionalities.

Example

```
esp_err_t mesh_init()
 {
  …
  /*-- Mesh Stack Configuration --*/
  ESP_ERROR_CHECK(esp_mesh_init());       //Initialize the mesh stack
  …
 }
```

### 2) Create the Mesh Event Group

Next we need to create the mesh event group, which can be allocated dynamically on the heap by calling the following function:

```
//File event_groups.h

typedef void* EventGroupHandle_t

EventGroupHandle_t xEventGroupCreate(void)
```

Where the return of the function represents the allocated event group's handler, which must be assigned to the `mesh_event_group` global variable that was previously defined.
As for its bits, to describe the node's status in the mesh network we'll use the following flags:

- A flag representing whether mesh networking is enabled on the node or not (MESH_ON)
- A flag representing whether the node is connected to a parent or not (MESH_PARENT)
- A flag representing whether the node has children connected or not (MESH_CHILD)
- A flag representing whether the external DS is accessible or not (MESH_TODS)
- A flag representing whether a new root election is currently in progress in the mesh network or not (that we'll use as an active low flag, i.e. 0 = election in progress, 1 = no election) (MESH_VOTE)

In addition to these flags, to allow the setup module to synchronize with the current state of the driver module an additional flag should be provided for each of its components, representing whether the component is currently being executed or not (this is to avoid creating undesidered duplicate tasks of the same component, we'll see in more detail later), and assuming a single component for each of the three driver categories previously discussed the following three additional flags are required:

- A flag representing whether the internal driver component is currently being executed or not (MESH_DRIVER_INTERNAL)
- A flag representing whether the external driver component is currently being executed or not (MESH_DRIVER_EXTERNAL)
- A flag representing whether the root driver component is currently being executed or not (MESH_DRIVER_ROOT)

From here we can associate a bit to each required flag as shown earlier in the code premises.

Example

```
esp_err_t mesh_init()
 {
  …
  mesh_event_group = xEventGroupCreate();          //Create the Mesh Event Group
  xEventGroupSetBits(mesh_event_group,MESH_VOTE); //Set the MESH_VOTE flag in
  …                                                  the mesh event group
 }                                                   (for it's active low)
```

## 3) Set the Base Mesh Configuration

At this point we must set the device's base mesh configuration, which should be shared by all the devices that we want to partecipate in the same mesh network, a base configuration that is described by the following data structure:

```
//File esp_mesh.h

typedef struct                  //Mesh Base Configuration
 {
  mesh_addr_t mesh_id;          //Mesh Network Identifier (MID)
  uint8_t channel;              //Mesh Wi-Fi Channel (1-14)
  mesh_event_cb_t event_cb;     //Mesh Events General Handler's address
  mesh_ap_cfg_t mesh_ap;        //Mesh SoftAP Settings
  mesh_router_t router;         //Mesh Router Settings
  const mesh_crypto_funcs_t* crypto_funcs;  //Mesh packets cryptographic algorithm
 } mesh_cfg_t;
```

Where:

- The `mesh_id` member represents the Mesh Network Identifier (MID) of the mesh network relative to the node, which represents the mesh network that the device will either join or create as a root note depending on its further configuration and other factors such as if other nodes broadcasting beacon frames with the same MID are found once mesh networking is enabled on the node.
  An MID has the format of a MAC address (6 bytes), and is described by the following union that as we'll see is used for a variety of purposes in the mesh networking API:

```
typedef struct                      //Destination Address (IPv4:PORT) of a mesh
 {                                    packet destined outside the mesh network
  ip4_addr_t ip4;                   //Destination IP
  uint16_t port;                    //Destination port
 } __attribute__((packed)) mip_t;   //("__attribute__((packed))" is for the
                                      byte alignment of the in-memory
                                      representation of the struct)
```

```
typedef union                   //Used to store an address in mesh networking
 {
  uint8_t addr[6];             //Depending on the context may represent a Mesh
                                 Network Identifier (MID), a node's MAC address
                                 or a Mesh Group Identifier (GID)
  mip_t mip;                   //Destination address of a mesh packet destined for
 } mesh_addr_t;                    the external DS
```

- The `channel` member represents the Wi-Fi channel (1-14) to be used for mesh networking.
  Note that due to the devices' limitation that the channel of their Station and SoftAP interfaces must coincide, and since the root node will need to connect to the external router on its AP channel, the Mesh Wi-Fi channel must coincide with the router's AP channel, which must then be known at this point in the configuration (note that it could also be retrieved by performing a Wi-Fi scan beforehand, a possibility not discussed here).

- The `event_cb` member represents the address of the Mesh Events General Handler function, which so is registered in the mesh stack by setting the node's base mesh configuration.

- The `mesh_ap` member defines the device's base mesh SoftAP settings, and consists the following struct:

```
//File esp_mesh.h

typedef struct                  //Base Mesh SoftAP Settings
 {
  uint8_t password[64];        //Required from nodes to connect to a parent node
  uint8_t max_connection;      //Maximum number of connected children for each
 } mesh_ap_cfg_t;                  node (must be ≥1, default and max = 10)
```

  Beside these settings note that the SSID a node uses for its SoftAP interface its derived from its base MAC address using a proprietary (or in any case undisclosed) algorithm, SSID that is also hidden from the node's beacon frames but instead is carried in the Vendor IE field, which will be referred to as Mesh IE from now on and, and as we'll discuss in more detail later, is typically encrypted in a node's Wi-Fi beacon frames.
  As a consequence of this, to attempt to connect to a parent node in the mesh network a node is always required to perform a Wi-Fi scan beforehand to retrieve its candidate parent SSID from its Mesh IE, which is decrypted and used by the mesh stack in a way that is transparent to the application (this will be discussed in greater detail later in the Manual Networking organization mode).

  Also note that in the current ESP-IDF version if the Mesh SoftAP password differs between nodes, after a certain number of failed attempts to connect to a parent, if possible nodes will attempt to connect to the external router as the root node of a new mesh network, even if such mesh network and the existing one share the same MID.
  Finally note that there exist other Mesh SoftAP settings that don't belong to a node's base mesh configuration (See "Additional Mesh SoftAP Settings" later).

- The `router` member defines the device's mesh router settings, which will be used by the root node to connect with the external router's AP, and consists in the following struct:

```
//File esp_mesh.h

typedef struct                  //Mesh Router Settings
 {
  uint8_t ssid[32];            //Router's AP SSID
  uint8_t ssid_len;            //Router's AP SSID length
  uint8_t password[64];        //Router's AP password
  uint8_t bssid[6];            //Router's AP BSSID (required only if the root node
 } mesh_router_t;                  must connect to a router with a specific BSSID)
```

- The `crypto_funcs` member defines the cryptographic algorithm to be applied to the Mesh IE in the Wi-Fi beacon frames, where currently this parameter can be set to NULL to use no encryption or to the address of the defined "g_wifi_default_mesh_crypto_funcs" constant to use the AES encryption (default).

From here, once the `mesh_cfg_t` struct has been initialized with the desidered values, the device's base mesh configuration can be set by calling the following function:

```
//File esp_mesh.h
```

```
esp_err_t esp_mesh_set_config(const mesh_cfg_t* base_config)
```

| Parameters | |
|---|---|
| base_config | The address of the struct holding the node's base mesh configuration to apply |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_MESH_ARGUMENT | Invalid argument(s) |
| ESP_ERR_NOT_ALLOWED | Operation not allowed |
| ESP_FAIL | Unknown error in the mesh stack |

Example

```
void mesh_events_handler(mesh_event_t event)
 {
  /* We'll see later */
 }
…
esp_err_t mesh_init()
 {
  …
  /*-- Mesh Base Configuration --*/
  mesh_cfg_t mesh_config = {0};                    //Holds the base mesh
                                                     configuration to apply
  //Mesh Network Identifier (MID)
  memcpy(&mesh_config.mesh_id.addr,MESH_NETWORK_ID,6);
  memcpy(&mesh_state.mid,MESH_NETWORK_ID,6);      //Also update the MID in
                                                     the mesh_state struct
  //Mesh Wi-Fi Channel
  mesh_config.channel = MESH_WIFI_CHANNEL;
  mesh_state.channel = MESH_WIFI_CHANNEL;         //Also update the mesh channel
                                                     in the mesh_state struct
  //Mesh Events General Handler address
  mesh_config.event_cb = &mesh_events_handler;

  //Base Mesh SoftAP settings
  memcpy((uint8_t*)&mesh_config.mesh_ap.password,              //SoftAP Password
                    ,strlen(MESH_SOFTAP_PASSWORD));
  mesh_config.mesh_ap.max_connection = MESH_SOFTAP_MAXCONNECTIONS; //SoftAP Max
                                                                     Connection
  //Mesh Router Settings
  memcpy((uint8_t*)&mesh_config.router.ssid,                //Router SSID
                    MESH_ROUTER_SSID,strlen(MESH_ROUTER_SSID));
  mesh_config.router.ssid_len = strlen(MESH_ROUTER_SSID);  //Router SSID length
  memcpy((uint8_t*)&mesh_config.router.password,            //Router Password
                    MESH_ROUTER_PASSWORD,strlen(MESH_ROUTER_PASSWORD));
  if(MESH_ROUTER_SPECIFIC_BSSID)                 //If the Root node must connect to
   memcpy((uint8_t*)&mesh_config.router.bssid, a router with a specific BSSID
                    MESH_ROUTER_BSSID,strlen(MESH_ROUTER_BSSID));//Router BSSID
```

```
    //Mesh IE Crypto Function
    if(MESH_IE_CRYPTO)       //If the Mesh IE should be encrypted
     mesh_config.crypto_funcs = &g_wifi_default_mesh_crypto_funcs; //AES encrypt.
    else
     mesh_config.crypto_funcs = 0;                                 //No encrypt.

    ESP_ERROR_CHECK(esp_mesh_set_config(&mesh_config));   //Apply the Base
     …                                                      Mesh Configuration
   }
```

Once the base mesh configuration on a device has been set it is possible to configure a set of additional settings relating to mesh networking, some that if defined should be shared among all nodes partecipating in the same mesh network and others that can differ from one node to another in the same network, settings that if not explicitely configured will be set to their default values once mesh networking is enabled on a node.

## 4) Mesh Shared Settings (*optional*)

The following mesh settings, if defined, should be shared among all nodes participating in the same mesh network, and can be divided into the following categories:

- **Mesh Organization Mode**
- **Mesh Topology Settings**
- **Mesh Self-Organized Root Settings**
- **Other Root Settings**
- **Mesh Additional SoftAP Settings**

### Mesh Organization Mode

In general an ESP32 mesh network can be organized in three different ways:

#### 1- Self-Organized Network (default)

In a self-organized mesh network once mesh networking is enabled on a node, it will start broadcasting in the Mesh IE of its beacon frame its MID and scan for the beacon frames of other nodes and the router's AP, where:

- If one or more nodes are found that have already joined a mesh network and match all of the following conditions:
    - They advertise in their Mesh IE the same MID set on the node
    - They are not on the maximum layer of their mesh network
    - Their mesh network capacity has not been reached
    - They are able to accept further children nodes

  the node will select and attempt to connect as a child to its preferred parent, which is represented by the node whose RSSI is above a predefined threshold, is located on the shallowest layer of the mesh network and has the fewest children connected.

- If one or more nodes are found that have not yet joined a mesh network and are advertising in their Mesh IE the same MID set on the node (i.e. nodes where mesh networking has just been enabled) and the router's AP is also found, a Root Node Election will take place, which is a process carried out in a predefined number of rounds (default = 10) where in the first one each node broadcasts its own MAC address and RSSI with the router, and in the following ones the MAC address and RSSI of the node with the highest advertised RSSI, thus "voting" for such node to elected as root.

From here, at the end of the election rounds, each participant will determine its own voting percentage (number of nodes voting for it to be elected / total number of nodes in the election), and if such percentage is above a predefined threshold (default = 0.9) the node will proclaim itself root and attempt to connect to the external router to form a new mesh network.
It must be noted that due to the variations of the nodes' RSSI with the router during the election process it is possible that at its end no node reaches the voting percentage threshold required to proclaim itself root, thus resulting in a failed election which immediately triggers a new one until a root node is found for the network (also note that to limit the chances of this happening it's possible to set a different voting percentage threshold for a node, we'll see later).

- If no node matching the above conditions is found, the node will keep searching for a number of scans equal to the number of rounds in a root election (default = 10) plus a fixed number of extra scans (default = 3).
  Thereupon, if no suitable node is found but the router's AP is, the node will attempt to connect to it as the root node to form a new mesh network, while if the router's is not found, the node will not be able to either join an existing or create a new mesh network, and it will continue to perform periodic scans indefinitely until a device meeting the required conditions is found.

In a self-organized mesh network, once connected to a parent, non-root nodes will keep periodically scanning in the background for a "better" preferred parent – i.e. a node matching the conditions described previously but located on a shallower level of the mesh network than their current parent – and if such node is found the nodes will disconnect from their current parent and attempt to connect to it as their child.

A self-organized mesh network also presents the following *self-healing* features:

- Should the connection between a node and its parent become unstable or fail, the node will disconnect from it and automatically search for another preferred parent to connect to.

- Should the nodes in the second layer of the mesh network (or in the first layers in case of multiple simultaneous failures) detect a root node failure, which occurs if the nodes don't receive its beacon frame in a predefined amount of time, the nodes will disconnect from it and start a new root election process as described above.

A self-organized mesh network also presents a *root conflicts resolution* algorithm, by virtue of which should the network present more than one root node at any given moment, after an internal communication between them the one with the highest RSSI with the router will ask the other root nodes to yield, causing them to disconnect from the router and search for a preferred parent to connect to (see also the MESH_EVENT_ROOT_ASKED_YIELD event later)

Since a mesh network is self-organized by default, no action is required to set this mesh organization mode on a node, even if, should a different organization mode be used, self-organized networking can be reenabled at any time by clearing its fixed-root setting via the `esp_mesh_fix_root()` function and by reenabling its self-organized networking features by calling the `esp_mesh_set_self_organized()` function, both of which will be covered later.

## 2- Fixed-Root Network

A Fixed-Root mesh network represents a variant of a self-organized network, where a node's behaviour depends on whether, by applying an user's custom criteria, the node recognizes itself as being the fixed root of the network or not.
From here, once mesh networking has been enabled on a node:

- If by applying the user's custom criteria the node recognizes itself to be the fixed-root of the mesh network, it must manually attempt to connect with the external router as the root node to form a new mesh network.
  Note that if the external router is not in range or its AP settings don't match the ones set in the fixed root's router configuration, the node will remain stuck in a loop repeatedly attempting and failing to connect with it.

- If by applying the user's custom criteria the node recognizes itself not to be the fixed-root of the network, it will scan and attempt to connect to a *preferred parent* as with self-organized networking, where note that in any case the node won't attempt to connect with the router or start a new root election, which are disabled in this mesh organization mode, and so if the node fails to find a preferred parent it will continue to search for it indefinetely.

As with self-organized networking once connected to a parent non-root (i.e. non-fixed-root) nodes will keep periodically scanning in the background for a "better" preferred parent, disconnecting from their current one and attempting to connect to it if found.

Again, as a *self-healing* feature, should the connection between a non-fixed-root node and its parent become unstable or fail, the node will disconnect from it and automatically search for another preferred parent to connect to, even if in this case, unlike what happens in self-organized network, should the nodes in the first layers of the network detect a root node failure they won't start a new root election or attempt in any case to connect with the router, thus keeping on scanning indefinetely for a fixed-root node to connect to.

Fixed-Root networks also present the same *root conflicts resolution* algorithm described for self-organized networks, where in this case, upon receiving the request to yield from the fixed root with the highest RSSI with the router, all the other fixed roots must fall back to operate as non-fixed-root nodes, thus disconnecting from the router and searching for a preferred parent to connect to.

To set up a fixed-root mesh network, the following fixed-root setting must be enabled on all of its nodes:

```
//File esp_mesh.h
```

```
esp_err_t esp_mesh_fix_root(bool enable)
```

| Parameters | |
|---|---|
| enable | Whether the root node in the network is fixed or not |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_FAIL | Unknown error in the mesh stack |

Also note that when a node connects to a parent node, should their fixed root settings differ, the child node will automatically change it to match the one of its parent (see also the MESH_EVENT_ROOT_FIXED event later).

### 3- Manual Networking

In a manual mesh network nodes apply none of the implicit actions or behaviours described in the previous mesh organization modes, and so the setup and management of the mesh network is left entirely to the programmer.
From here, once mesh networking has been enabled on a node, a preliminary manual Wi-Fi scan must be performed to search for the mesh nodes and/or the external router to attempt to connect to according to a user's custom criteria, where the status in the mesh network of the nodes that were found, and so their capability to accept child connections, can be retrieved from their Mesh IE (as we'll see in more detail later).
Note that performing a preliminary manual Wi-Fi scan in Manual Networking is also mandatory to allow a connection to a parent node to be performed, since as discussed before the SSID of nodes in a mesh network, which is derived from their base MAC address using a proprietary (or in any case undisclosed) algorithm, is hidden from their Wi-Fi beacon frames but is put (typically encrypted) in their Mesh IE, SSID that as we'll see is decrypted by the Mesh Stack in a manner transparent to the application.

From here, once a node successfully connects to a parent, it will remain its child indefinitely until the connection between the two becomes unstable or fails, and while the mesh stack will still attempt to reconnect the node to its parent, in this case if desidered it's also possible to perform another Wi-Fi scan to search for another parent to fall back to.

Also note that a manual mesh network presents no automatic procedures such as the root node's self-healing or root node conflicts resolution, and indeed in this organization mode the root node is treated as the fixed-root of a fixed-root network.

To set up a manual mesh network, other than enabling the fixed-root setting as previously seen with Fixed-Root networking, the self-organized networking features of each of its nodes must be disabled, which is obtained by calling the following function:

```
//File esp_mesh.h

esp_err_t esp_mesh_set_self_organized(bool enable,
                                      bool select_parent)
```

| Parameters | |
|---|---|
| enable | Whether the node self-organized networking features should be enabled or not (false to enable manual networking) |
| select_parent | If enabling the node's self-organized networking features, let the device choose its preferred parent or keep it connected to the current one |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_FAIL | Unknown error in the mesh stack |

Going back to the code premises of our mesh application, the "mesh_org_t" enumerated type represents the possible organization modes of a mesh network, while in the "mesh_status_t" struct the "org" member represents the current organization mode used by a node in the network, which as said before should be shared among all other nodes partecipating in the same mesh.

Example

```
esp_err_t mesh_init()
 {
  …
  /*-- Mesh Organization Mode --*/
  switch(MESH_ORGANIZATION_MODE)
   {
    /* No action is required to set the mesh
       organization mode to self-organized   */
    case SELF_ORGANIZED:
     mesh_state.org = SELF_ORGANIZED;

    /* Setting the mesh organization mode to Fixed-Root
       requires the fixed root setting on a node to be enabled */
    case FIXED_ROOT:
     ESP_ERROR_CHECK(esp_mesh_fix_root(true));
     mesh_state.org = FIXED_ROOT;
     break;

    /* Setting the mesh organization mode to Manual Networking
       requires the fixed root setting to be enabled and the
       node's self-organized networking features to be disabled */
    case MANUAL_NETWORKING:
     ESP_ERROR_CHECK(esp_mesh_fix_root(true));
     ESP_ERROR_CHECK(esp_mesh_set_self_organized(false,false));
     mesh_state.org = MANUAL_NETWORKING;
     break;
   }
  …
 }
```

## Mesh Topology Settings

- **Set the Mesh Network Maximum Layer**

  The maximum layer of a mesh network represents the layer where joining nodes are configured as leaf nodes, i.e. nodes that are not allowed to have children nodes, and it can be set via the following function:

```
//File esp_mesh.h

esp_err_t esp_mesh_set_max_layer(int max_layer)
```

| Parameters | |
|---|---|
| max_layer | The maximum layer of the mesh network (default = 15, max = 25) |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_MESH_ARGUMENT | Invalid argument(s) |
| ESP_ERR_NOT_ALLOWED | Operation not allowed |
| ESP_FAIL | Unknown error in the mesh stack |

Example

```
esp_err_t mesh_init()
 {
  …
  ESP_ERROR_CHECK(esp_mesh_set_max_layer(MESH_MAX_LAYER));  //Set the mesh
  …                                                           maximum layer
 }
```

- **Set the Mesh Network Maximum Capacity**

  In general the maximum capacity of a mesh network, i.e. the maximum number of nodes that can be part of the same network, is constrained by its maximum layer "L" and the maximum number of children allowed for each node "C", and is equal to:

$$\text{max nodes} = \sum_{i=1}^{L} C^{(i-1)}$$

  with a default theoretical maximum capacity of 111'111'111'111'111 nodes (L = 15, C = 10). Theoretical scenarios aside, it is also possible to set an upper limit for the capacity of a mesh network, and this is obtained by calling the following function:

```
//File esp_mesh.h

esp_err_t esp_mesh_set_capacity_num(int max_nodes)
```

| Parameters | |
|---|---|
| max_nodes | The maximum node capacity of the mesh network (default = 300) |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_MESH_ARGUMENT | Invalid argument(s) |
| ESP_ERR_NOT_ALLOWED | Operation not allowed |
| ESP_FAIL | Unknown error in the mesh stack |

Example

```
esp_err_t mesh_init()
 {
  …                          //Set the mesh network maximum capacity
  ESP_ERROR_CHECK(esp_mesh_set_capacity_num(MESH_MAX_CAPACITY));
  …
 }
```

## Mesh Self-Organized Root Settings

The following settings are relevant to a self-organized mesh network only, even if they can be configured regardless of the intended mesh organization mode since in general it may vary during the application's execution.

- **Set the Root Election Voting Percentage Threshold**

  The voting percentage threshold for a node to proclaim itself as root once the rounds of a root election are over can be changed via the following function:

  ```
  //File esp_mesh.h

  esp_err_t esp_mesh_set_vote_percentage(float threshold)
  ```

| Parameters | |
|---|---|
| threshold | The voting percentage threshold for a node to proclaim itself as root at the end of a root election (default = 0.9) |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_FAIL | Unknown error in the mesh stack |

  Note that lowering the voting percentage threshold from its default value proves useful in dynamic environments where the nodes' RSSI with the router vary very quickly, which may lead to none of them reaching the default voting percentage threshold thus resulting in a failed root election.
  Note however that setting this parameter to a low value may cause multiple nodes to proclaim themselves as root in the network, conflicts that as discussed before can be resolved by the self-organized root conflicts resolution algorithm.

  Example

  ```
  esp_err_t mesh_init()
   {
    …
    /* Possibly check the validity of the custom root
       election threshold, i.e. threshold∈(0.0,1.0]  */
    //Set the root election election threshold
    ESP_ERROR_CHECK(esp_mesh_set_vote_percentage(ROOT_ELECTION_THRESHOLD));
    …
   }
  ```

- **Set the minimum number of Rounds in a Root Election**

  The minimum number of rounds in a root node election and consequently its duration (and also the minimum number of scans in a self-organized network before newly activated nodes attempt to connect to the router as root if no other node is found) can be changed via the following function:

```
//File esp_mesh_internal.h (automatically included by the previous headers)

typedef struct      //Self-Organized implicit actions' number of attempts
 {
  int scan;         //Minimum root election rounds (default=10)  <----
  int vote;         //Maximum self-healing root election rounds (default=15)
  int fail;         //Maximum parent reconnection tries before
                      switching to another parent (default=120)
  int monitor_ie;   //Maximum number of beacon frames with an updated Mesh IE
                      received from the node's parent before the node must
                      update its own Mesh (default=10)
 } mesh_attempts_t;
```

```
esp_err_t esp_mesh_set_attempts(mesh_attempts_t* attempts)
```

| Parameters | |
|---|---|
| attempts | Address of the struct holding the self-organized implicit actions number of attempts |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_FAIL | Unknown error in the mesh stack |

The other members of the `mesh_attempts_t` struct define the number of attempts of other implicit actions in a self-organized network as described above, and they must be set to valid values (possibly their defaults) before calling the `esp_mesh_set_attempts()` function.

In any case the minimum number of rounds in a root election should be tailored to the expected number of nodes and possibly the physical topology of the mesh network, where the larger the network the more rounds are advised to increase the possibility for the optimal root node to be elected while reducing the chances of root node conflicts to arise.

Example

```
esp_err_t mesh_init()
 {
  …
  mesh_attempts_t self_attempts;

  self_attempts.scan = ROOT_ELECTION_MIN_ROUNDS;  //Set the minimum rounds
  self_attempts.vote = 15;                          in a root election
  self_attempts.fail = 120;                 //In this case the other members
  self_attempts.monitor_ie = 10;              of the mesh_attempts_t struct
                                              are set to their default values
  ESP_ERROR_CHECK(esp_mesh_set_attempts(&self_attempts)); //Apply the set
  …                                                         number of
 }                                                          attempts
```

- **Set the Root Node Healing Delay**

  The root node healing delay, i.e. the time from the moment the nodes on the second layer of the mesh network (or in general in the first layers in case of multiple simultaneous failures) receive the last beacon frame from the root node and the moment they disconnect from it and start a new root election can be set via the following function:

```
//File esp_mesh.h

esp_err_t esp_mesh_set_root_healing_delay(int delay_ms)
```

| Parameters | |
|---|---|
| delay_ms | The root node healing delay to set (default = 6000ms) |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_FAIL | Unknown error in the mesh stack |

It should be noted that while lowering the root healing delay allows for quicker self-healing of the mesh network and the restoring of connectivity with the external DS, setting it to too low a value may cause unnecessary root elections and so delays if due to its load or other internal causes the beacon frame sent by the root note is delayed to the point of triggering the root node self-healing process in the rest of the network.

Example

```
esp_err_t mesh_init()
 {
  …                                                    //Set the root node healing delay
  ESP_ERROR_CHECK(esp_mesh_set_root_healing_delay(MESH_ROOT_HEALING_DELAY));
  …
 }
```

## Other Root Node Settings

- **Set the size of the root node's receiving queue for packets destined for the external DS**

  The size of the root node's receiving queue for packets destined for the external DS (ot TODS queue) can be set via the following function:

```
//File esp_mesh.h

esp_err_t esp_mesh_set_xon_qsize(int qsize)
```

| Parameters | |
|---|---|
| qsize | The size of the root node's receiving queue for packets destined to the external DS (default = 32, min = 16) |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_FAIL | Unknown error in the mesh stack |

In general the size of the root node's TODS queue should be tailored to the expected network throughput destined for the external DS, where an increased queue size corresponds to a lower probability of packet loss at the root node at the cost of greater memory occupation.

Example

```
esp_err_t mesh_init()
 {
  …                          //Set the root node's TODS receiving queue size
  ESP_ERROR_CHECK(esp_mesh_set_xon_qsize(MESH_ROOT_TODS_QUEUE_SIZE));
  …
 }
```

## Mesh Additional SoftAP Settings

The following settings complement the Mesh SoftAP settings previously discussed in a node's base mesh configuration.

- ### Set the Mesh SoftAP Authmode

  The authentication protocol used by nodes to communicate over the mesh network can be set via the following function:

```
//File esp_wifi_types.h (automatically included by the previous headers)

typedef enum                    //SoftAP authmode enumerates
 {
  WIFI_AUTH_OPEN = 0,           //Open (no authentication)
  WIFI_AUTH_WEP,                //WEP  (buggy, avoid)
  WIFI_AUTH_WPA_PSK,            //WPA_PSK
  WIFI_AUTH_WPA2_PSK,           //WPA2_PSK
  WIFI_AUTH_WPA_WPA2_PSK,       //WPA_WPA2_PSK
  WIFI_AUTH_WPA2_ENTERPRISE,    //WPA2_ENTERPRISE
  WIFI_AUTH_MAX
 } wifi_auth_mode_t;

//File esp_mesh.h
```

```
esp_err_t esp_mesh_set_ap_authmode(wifi_authmode_t authmode)
```

| Parameters | |
|---|---|
| authmode | The authentication protocol used to transmit data over the mesh network (default = WPA2/PSK) |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_MESH_ARGUMENT | Invalid argument(s) |
| ESP_ERR_NOT_ALLOWED | Operation not allowed |
| ESP_FAIL | Unknown error in the mesh stack |

Also note that the data received via mesh networking is processed along with the chosen authentication protocol entirely by the mesh stack, while the nodes' SoftAP interfaces are left with no authentication or password set, and so result *open* (but with their SSID hidden) at the Wi-Fi interface level.

Example

```
esp_err_t mesh_init()
 {
  …                                          //Set Mesh SoftAP Authmode
  ESP_ERROR_CHECK(esp_mesh_set_ap_authmode(MESH_SOFTAP_AUTHMODE));
  …
 }
```

- **Set the Mesh Children Disassociaton Delay**

  The mesh children disassociation delay is the time from the moment the last data is received from a node's child to the moment the node marks it as inactive (i.e. failed), disassociating it.
  The mesh children disassociation delay can be set by calling the following function:

```
//File esp_mesh.h

esp_err_t esp_mesh_set_ap_assoc_expire(int delay_s)
```

| Parameters | |
|---|---|
| delay_s | The children disassociation delay to set (min and default = 10s) |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_FAIL | Unknown error in the mesh stack |

Note that it's not necessary for a child node to explicitly send a mesh packet to its parent for its disassociation timer to reset, since implicit mesh "state" packets are periodically sent from child to parent by the mesh internal management.

Example

```
esp_err_t mesh_init()
 {
  …                                  //Set the mesh children disassociation delay
  ESP_ERROR_CHECK(esp_mesh_set_ap_assoc_expire(CHILD_DISASSOCIATION_DELAY));
  …
 }
```

## 5) Mesh Node-specific Settings (*optional*)

Unlike the mesh shared settings, the following settings if defined may vary from one node to another in the mesh network.

- **Add the node to one or more Mesh Groups**

  A mesh group represents a subset of a mesh network comprised of zero or more nodes used for multicasting purposes to allow the sending of mesh packets to all nodes belonging to a certain group.
  A mesh group is identified by its Group Identifier (GID), which as with the Mesh Network Identifier (MID) has the format of a MAC address (6 bytes) and is described by the `mesh_addr_t` union previously detailed.
  From here a node can be added to one or more mesh groups by calling the following function:

```
//File esp_mesh.h

esp_err_t esp_mesh_set_group_id(const mesh_addr_t* gids,int num)
```

| Parameters | |
|---|---|
| gids | The address of the array holding the GIDs of the mesh groups to add the node to |
| num | The size of such array, i.e. the number of mesh groups to add the node to |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_MESH_ARGUMENT | Invalid argument(s) |
| ESP_FAIL | Unknown error in the mesh stack |

Example

```
esp_err_t mesh_init()
 {
  …
  mesh_addr_t group_ids[NODE_GROUPS_NUMBER];
  /* Set the GIDs of the groups we want to add the node to */
  //Add the node to the aforementioned mesh groups
  ESP_ERROR_CHECK(esp_mesh_set_group_id(&group_ids,NODE_GROUPS_NUMBER));
  …
 }
```

- **Set custom MAC Addresses for the node's Wi-Fi Interfaces (Station and/or SoftAP)**
  As an additional configuration it is also possible to set custom MAC addresses for the node's Station and/or SoftAP interfaces, which may prove useful in the design and configuration of a manual mesh network, and is obtained by calling the following function:

```
//File esp_interface.h (automatically included by the previous headers)

typedef enum                  //Networking interface enumerates
 {
  ESP_IF_WIFI_STA = 0,        //Wi-Fi station interface (or mode)
  ESP_IF_WIFI_AP,             //Wi-Fi softAP interface (or mode)
  ESP_IF_ETH,                 //Ethernet interface
  ESP_IF_MAX
 } esp_interface_t;

//File esp_wifi.h

typedef esp_interface_t wifi_interface_t;
```

```
esp_err_t esp_wifi_set_mac(wifi_interface_t ifx_mode,
                           const uint8_t* mac[])
```

| Parameters | |
|---|---|
| ifx_mode | The interface to set the MAC address for<br>❑ ESP_IF_WIFI_STA → Station Interface<br>❑ ESP_IF_WIFI_AP → SoftAP Interface |
| mac | The MAC address to set for the interface |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_WIFI_NOT_INIT | The Wi-Fi stack is not initialized (call esp_wifi_init() first) |
| ESP_ERR_WIFI_IF | Wi-Fi internal error |
| ESP_ERR_WIFI_MODE | Invalid Wi-Fi interface mode |
| ESP_ERR_WIFI_MAC | Invalid MAC address |
| ESP_ERR_INVALID_ARG | Invalid argument(s) |
| ESP_FAIL | Unknown error in the Wi-Fi stack |

Note that the MAC addresses of a node's Station and SoftAP interfaces cannot coincide, and trying to do so will cause the function to return the ESP_ERR_WIFI_MAC error, and as an additional constraint the bit 0 of the most significant byte of the MAC address cannot be set (for example xA:xx:xx:xx:xx:xx is a valid address, while x5:xx:xx:xx:xx:xx is not).

Also note that in mesh network the MAC address of a node's SoftAP interface (corresponding to its BSSID) is used exclusively by nodes to connect to it as children, while the MAC address of a node's Station interface is also propagated upwards in the routing tables of all the node's ancestors, thus representing the label by which a node is addressable in the mesh network.

Example

```
esp_err_t mesh_init()
 {
  …
  uint8_t mac[6];        //Used to set a custom MAC address for an interface

  if(MESH_STATION_USE_CUSTOM_MAC)         //If a custom MAC address for
   {                                        the station interface is used
    memcpy(mac,MESH_STATION_CUSTOM_MAC,6);
    ESP_ERROR_CHECK(esp_wifi_set_mac(ESP_IF_WIFI_STA,mac));
   }
  if(MESH_SOFTAP_USE_CUSTOM_MAC)          //If a custom MAC address for
   {                                        the softAP interface is used
    memcpy(mac,MESH_SOFTAP_CUSTOM_MAC,6);
    ESP_ERROR_CHECK(esp_wifi_set_mac(ESP_IF_WIFI_AP,mac));
   }
  …
 }
```

## 6) Enable Mesh Networking

Once all the desidered settings have been configured, mesh networking can be enabled on the device, which is obtained by calling the following function:

```
//File esp_mesh.h

esp_err_t esp_mesh_start(void)
```

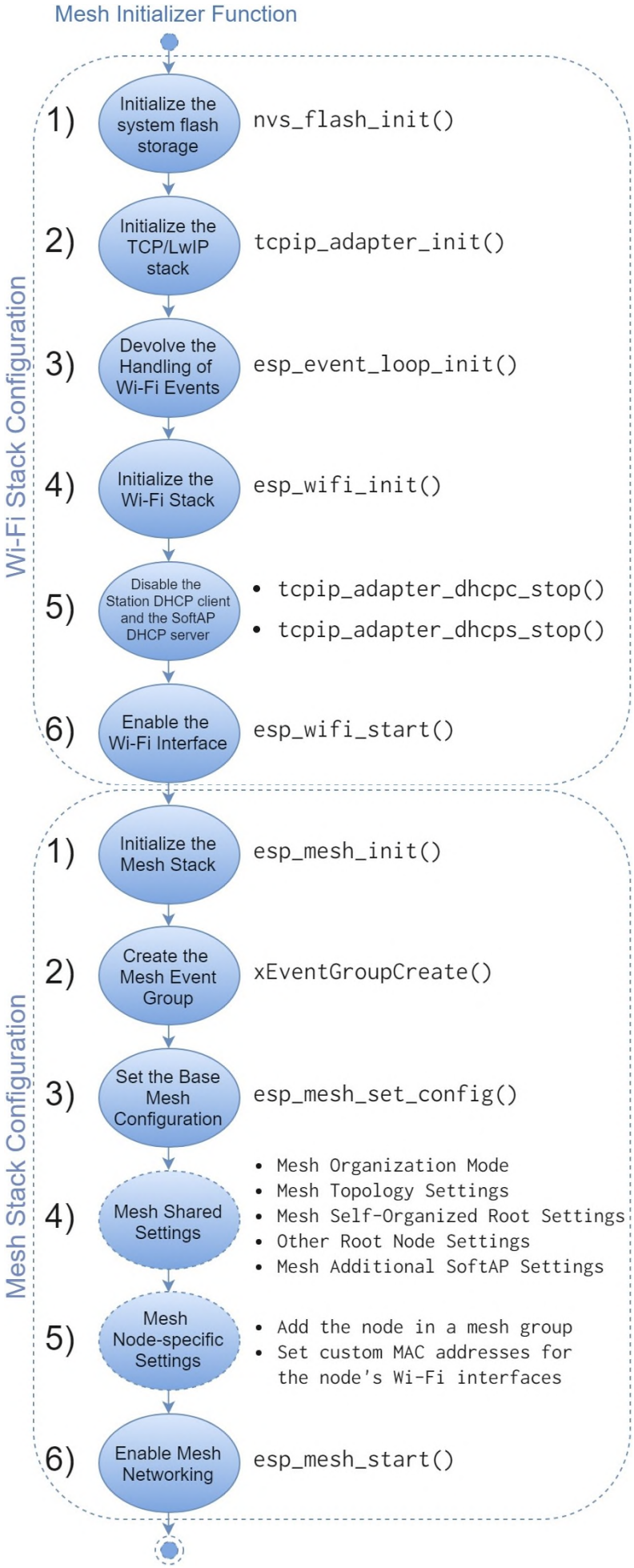
| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_MESH_NOT_INIT | The mesh stack is not initialized (call esp_mesh_init() first) |
| ESP_ERR_MESH_NOT_CONFIG | The base mesh configuration is not set (call esp_mesh_set_config() first) |
| ESP_ERR_MESH_NO_MEMORY | Out of memory |
| ESP_FAIL | Unknown error in the mesh stack |

Example

```
esp_err_t mesh_init()
 {
  …
  ESP_ERROR_CHECK(esp_mesh_start());    //Enable Mesh Networking
  return ESP_OK;                        //End of the Mesh Initializer Function
 }
```

Once the esp_mesh_start() function returns successfully mesh networking will be enabled on the device, which will cause the MESH_EVENT_STARTED event to raise in the mesh stack, thereby causing the mesh setup process to pass into its event-driven phase.

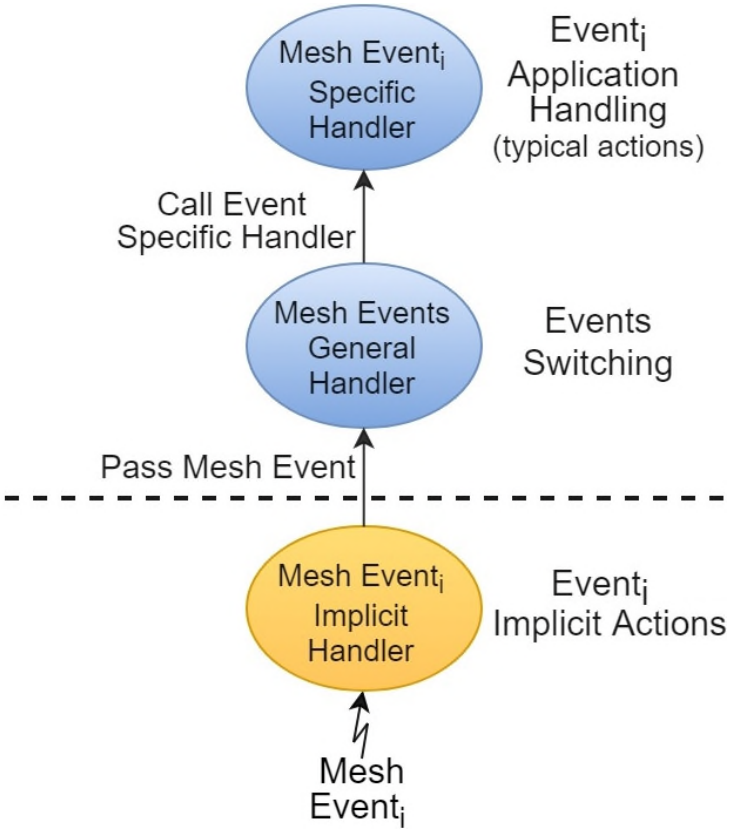
Summarizing, the tasks that must be performed by the Mesh Initializer Function are:




Also note that the majority of the mesh settings we have discussed so far can also be modified after mesh networking has been enabled on a device, which may cause it to disconnect or change its status in its current mesh network depending on the changes applied.

## Event-driven Setup Phase

Once mesh networking has been enabled on a node, its setup process enters its event-driven phase, which consists in the application-level handling of the **Mesh Events** raised by the Mesh Stack.
A Mesh Event can be represented as a set of conditions on the state of the mesh stack on a node which, other than evolving internally through the application's execution, is affected by the Wi-Fi Events and the data received from the Wi-Fi stack, and once all the conditions representing an event have been met such event will be raised by the mesh stack.
Once raised, a mesh event is first handled internally in the mesh stack by its **Specific Implicit Handler**, which performs a set of **Implicit Actions** that depend on other conditions relative to the mesh stack's state, in particular on the Mesh Organization Mode set on the node.
Once its internal handling is complete the event with its ID and a set of additional information are passed by the mesh stack at the application level to the **Mesh Events General Handler** function previously registered by setting the node's base mesh configuration, which in turn will call, passing the additional information provided, the **Specific Handler** relative to such event, where the actual application-level handling of the event is performed, whose **Typical Actions** depend again on the mesh status of the node (once more in particular on its Mesh Organization Mode), the additional information provided by the mesh stack, and possibly the intended logic of the setup process.




The mesh events that can be raised by the mesh stack can also be organized into the following categories:

- The **Mesh Main Setup Events**, which are relative to the main setup process of a node, which ends when it connects to a parent in the mesh network.
- The **Mesh Network Update Events**, which are raised once a node's main setup process is finished upon receiving specific updates from the mesh network.
- The **Mesh Root-specific Events**, which may rise on the current, former or candidate root nodes in the mesh network.

The full list of mesh events along with their description, the implicit actions performed by their implicit handlers, the additional information passed at the application-level by the mesh stack, and the typical actions that should be performed by their specific handlers are summarized in the the following tables:

# Mesh Main Setup Events

| Event ID | Description | Raised when | Implicit Actions (Event Implicit Handler) | | | Additional Information passed by the Mesh Stack | Typical Actions (Event Specific Handler) | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | Self-Organized Networking | Fixed-Root Networking | Manual Networking | | Self-Organized Networking | Fixed-Root Networking | Manual Networking |
| **MESH_EVENT_STARTED** | Mesh networking has been enabled on the device | Mesh networking is enabled by calling the `esp_mesh_start()` function | Start sending Wi-Fi beacon frames and begin scanning for a *preferred parent*, a node broadcasting the same MID that has not yet joined a mesh network, and the external router | Start sending Wi-Fi beacon frames and begin scanning for a preferred parent to connect to | Start sending Wi-Fi beacon frames | **None** | Reset the node's mesh status | Reset the node's mesh status, and if the node recognizes itself as the fixed root, manually attempt to directly connect to the external router | Reset the node's mesh status and start a Wi-Fi scan to search for a parent to connect to |
| **MESH_EVENT_STOPPED** | Mesh networking has been disabled on the device | • Mesh networking is disabled by calling the `esp_mesh_stop()` function<br>• A fatal error has occured in the Wi-Fi stack | Reset the mesh stack's status on the device | | | **None** | Reset the node's mesh status and, if the disabling was unintentional, possibly attempt to re-enable mesh networking by calling the `esp_mesh_start()` function | | |
| **MESH_EVENT_SCAN_DONE** (Manual Networking only) | The mesh stack has completed a Wi-Fi scan requested by the application | The mesh stack completes a Wi-Fi scan that was requested by calling the `esp_wifi_scan_start()` function | | | Store in dynamic memory the records and the vendor IEs of the APs that were found in the scan | • The number of APs that were found in the scan | | | By applying a user's custom criteria determine a viable parent to attempt to connect to, or perform another Wi-Fi scan after a defined interval if no suitable parent for the node was found |
| **MESH_EVENT_PARENT_CONNECTED** | The node has connected to a parent in the mesh network | The node successfully connects to a parent in the mesh network, which may either be another node or the external router (as root) | If the node is not root, retrieve a set of information from its parent (see later) and schedule periodical scans searching for a "better" preferred parent for the node | | If the node is not root, retrieve a set of information from its parent (see later) | • A set of information on the parent's AP<br>• The layer of the node in the mesh network | • Update the node's parent address, mesh layer and thus its type.<br>• If the node is root and it uses a dynamic IP configuration on its station interface, start its station DHCP client, otherwise apply a custom static IP configuration.<br>• Start the node's internal driver components. | | |
| **MESH_EVENT_PARENT_DISCONNECTED** | The node has disconnected or has failed to connect or to reconnect to its parent in the mesh network | The node disconnects from its parent for any reason or it fails to connect or automatically reconnect to it | Perform a fixed number of attempts to reconnect the node to its previous parent before searching for another preferred parent to connect to | Perform a fixed number of attempts to reconnect the node to its previous parent before searching for another preferred parent to connect to (non-fixed-root nodes) or attempt indefinetely to reconnect the node to its previous parent (fixed-root node) | Attempt indefinetely to reconnect the node to its previous parent | • A set of information on the disconnected parent's AP | If the node disconnected from its parent:<br>• If it's root, inform the rest of the network that the external DS is no longer accessible and reset its station IP configuration.<br>• Reset the node's mesh status.<br>For Manual Networking only, possibly perform a Wi-Fi scan to search for another parent to fall back to | | |
| **MESH_EVENT_NO_PARENT_FOUND** | The node has failed to find a viable parent to attempt to connect to | No viable parent was found for the node in a fixed number of scans | Periodically scan for a preferred parent, a node broadcasting the same MID that has not yet joined a mesh network, or the router to connect to | Periodically scan for a preferred parent to connect to (non-fixed-root only) | | • The total number of scans performed searching for a viable parent for the node | Possibly perform advanced error-recovery routines such as carrying out an all-channel Wi-Fi scan searching for a parent on a different Mesh Wi-Fi channel and/or changing the node's mesh configuration | | |

# Mesh Network Update Events

| Event ID | Description | Raised when | Implicit Actions (Event Implicit Handler) | | | Additional Information passed by the Mesh Stack | Typical Actions (Event Specific Handler) | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | Self-Organized Networking | Fixed-Root Networking | Manual Networking | | Self-Organized Networking | Fixed-Root Networking | Manual Networking |
| **MESH_EVENT_CHILD_CONNECTED** | A child has connected to the node | A child node successfully connects to the node's SoftAP interface | **None** | | | • The station MAC address of the connected child<br>• The *aid* given by the SoftAP interface to the connected child | **None** | | |
| **MESH_EVENT_CHILD_DISCONNECTED** | A child has disconnected from the node | A child node disconnects from the node's SoftAP interface, intentionally, due to a network error, or because its disassociation delay has triggered | **None** | | | • Station MAC address of the disconnected child<br>• The *aid* that was given by the SoftAP interface to the disconnected child | **None** | | |
| **MESH_EVENT_ROUTING_TABLE_ADD** | One or more entries are added to the node's routing table | A node's descendant (with its possible further descendants) joins the mesh network | Add the descendant and its further descendants' station MAC addresses to the node's routing table | | | • The number of entries that were added to the node's routing table<br>• The current number of entries in the node's routing table | **None** | | |
| **MESH_EVENT_ROUTING_TABLE_REMOVE** | One or more entries are removed from the node's routing table | A node's descendant (with its possible further descendants) disconnects from the mesh network | Remove the descendant and its further descendants' station MAC addresses from the node's routing table | | | • The number of entries that were removed from the node's routing table<br>• The current number of entries in the node's routing table | **None** | | |
| **MESH_EVENT_ROOT_ADDRESS** | The node has retrieved the MAC address of the root node's SoftAP interface | Automatically when a node connects to a parent (root node included) and when a root switch occurs, where the new root address is propagated from parent to children through the entire network | **None** | | | • The SoftAP MAC address of the root node | Update the root address in the node's mesh status | | |
| **MESH_EVENT_ROOT_FIXED** | The fixed root setting on the node differs from the one of its parent | Automatically when a node connects to a parent should their fixed root settings differ | Apply the parent's fixed root setting | | | • The new value of the fixed root setting | Change the node's mesh organization mode accordingly (note that if fixed-root networking is set in this way, the node can only be a non-fixed-root node) | | |
| **MESH_EVENT_TODS_STATE** | The node is informed of a change in the accessibility of the external DS | When a node connects to its parent node, if this was previously informed that the external DS is accessible, or if the root node posts an update on the accessibility of the external DS, information that is propagated from parent to children through the entire network | **None** | | | • Whether the external DS is accessible or not | If the external DS is accessible, start the node's external driver components | | |
| **MESH_EVENT_VOTE_STARTED** (Self-Organized Networking only) | A new root election is started in the mesh network | If the root node self-healing process triggers or if the root node waives its root status | Carry out the root election process with the rest of the mesh network | | | • The reason for the new root election<br>• The maximum number of rounds in the new root election | **None** | | |
| **MESH_EVENT_VOTE_STOPPED** (Self-Organized Networking only) | A new root election in the mesh network has ended | Once the new root election rounds are over | If the node recognizes itself as the new root, if the election was caused by the root node self-healing process, disconnect from the current parent and attempt to connect to the router as root, otherwise if the election was triggered by the current root node waiving its root status, send it a root switch request | | | **None** | **None** | | |
| **MESH_EVENT_LAYER_CHANGE** | The node's layer in the mesh network has changed | A topology change affects the node's ancestors | If the layer change caused the node's type to switch from intermediate parent to leaf or vice versa, modify its capability to accept children nodes accordingly (where note that in any case its current children won't be disassociated if the node's type changes to leaf node) | | | • The node's new layer in the mesh network | Update the node's layer and possibly its type in its mesh status | | |
| **MESH_EVENT_CHANNEL_SWITCH** | The Mesh Wi-Fi Channel of the network has changed | The external router switches its AP onto a different Wi-Fi channel, which causes all nodes in the mesh network to switch in cascade onto such channel, which so represents the new Mesh Wi-Fi Channel | Update the Mesh Wi-Fi Channel | | | • The new Mesh Wi-Fi Channel | Update the node's Mesh Wi-Fi Channel in its mesh status | | |

## Mesh Root-specific Events

| Event ID | Description | Raised when | Implicit Actions (Event Implicit Handler) | | | Additional Information passed by the Mesh Stack | Typical Actions (Event Specific Handler) | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | Self-Organized Networking | Fixed-Root Networking | Manual Networking | | Self-Organized Networking | Fixed-Root Networking | Manual Networking |
| MESH_EVENT_ROOT_GOT_IP | The root node has obtained an IP configuration for its Station interface | The station DHCP client retrieves a dynamic IP configuration or a static IP configuration is applied | None | | | • The Station's IP configuration<br>• Whether this IP configuration overrode a previous one | Start the node's root driver components and inform the rest of the mesh network that the external DS is now accessible | | |
| MESH_EVENT_ROOT_LOST_IP | The lease time of the node's station dynamic IP configuration has expired | The current root node station DHCP client fails to renew or otherwise retrieve a new IP configuration, or automatically after the lease time on a former root node has expired | None | | | None | If the node is the current root, inform the rest of the mesh network that the external DS is no longer accessible | | |
| MESH_EVENT_ROOT_SWITCH_REQ | The root node has received a root switch request from a candidate root | The root node receives a root switch request from a candidate root node following the election triggered by the root node waiving its root status | Send the root switch acknowledgment to the candidate root, disconnect from the router and search for a preferred parent to attempt connect to | | | • The reason for the root switch request (currently inconsequential)<br>• The station MAC address of the candidate root | Reset the (now former) root station IP configuration and its mesh status | | |
| MESH_EVENT_ROOT_SWITCH_ACK | The candidate root node has received the root switch acknowledgment from the former root node | The former root node acknowledges the root switch requested by the candidate root following the election triggered by the former root node waiving its root status | Disconnect from the current parent and attempt to connect to the router as the root node | | | None | None | | |
| MESH_EVENT_ROOT_ASKED_YIELD | Another root node with a higher RSSI with the router has asked this root node to yield | Should a root conflict occur in the mesh network, where after an internal communication between all root nodes the one with the highest RSSI with the router will ask the others to yield | Disconnect from the router and search for a preferred parent to connect to | | | • The requesting root's RSSI with the router<br>• The number of nodes in the requesting root's network<br>• The requesting root's station MAC address | Reset the (now former) root station IP configuration and its mesh status | | |

## Mesh Events General Handler

The ID and possible additional information on the mesh events that occur are passed by the mesh stack to the Mesh Events General Handler using the following data structures:

### Mesh Events ID Definitions

```
//File esp_mesh.h

typedef enum                                //Mesh Events IDs
 {
  /*-- Mesh Main Setup Events --*/
  MESH_EVENT_STARTED,
  MESH_EVENT_STOPPED,
  MESH_EVENT_SCAN_DONE,
  MESH_EVENT_PARENT_CONNECTED,
  MESH_EVENT_PARENT_DISCONNECTED,
  MESH_EVENT_NO_PARENT_FOUND,

  /*-- Mesh Network Update Events --*/
  MESH_EVENT_CHILD_CONNECTED,
  MESH_EVENT_CHILD_DISCONNECTED,
  MESH_EVENT_ROUTING_TABLE_ADD,
  MESH_EVENT_ROUTING_TABLE_REMOVE,
  MESH_EVENT_ROOT_ADDRESS,
  MESH_EVENT_ROOT_FIXED,
  MESH_EVENT_TODS_STATE,
  MESH_EVENT_VOTE_STARTED,
  MESH_EVENT_VOTE_STOPPED,
  MESH_EVENT_LAYER_CHANGE,
  MESH_EVENT_CHANNEL_SWITCH,

  /*-- Mesh Root-specific Events --*/
  MESH_EVENT_ROOT_GOT_IP,
  MESH_EVENT_ROOT_LOST_IP,
  MESH_EVENT_ROOT_SWITCH_REQ,
  MESH_EVENT_ROOT_SWITCH_ACK,
  MESH_EVENT_ROOT_ASKED_YIELD,
 } mesh_event_id_t;
```

## Mesh Events Additional Information Types

### Mesh Main Setup Events

```
/*============================= MESH_EVENT_SCAN_DONE ============================*/

//File esp_mesh.h

typedef struct
 {
  uint8_t number;                 //The number of APs that were found in the Wi-Fi scan
 } mesh_event_scan_done_t;

/*========================== MESH_EVENT_PARENT_CONNECTED ==========================*/

//File esp_event.h (automatically included by the previous headers)

typedef struct                     //SYSTEM_EVENT_STA_CONNECTED event-specific info
 {
  uint8_t ssid[32];                //SSID of the AP the station connected to
  uint8_t ssid_len;                //SSID length of the AP the station connected to
  uint8_t bssid[6];                //BSSID of the AP the station connected to
  uint8_t channel;                 //Wi-Fi channel of the AP the station connected to
  wifi_auth_mode_t authmode;       //The authmode used by AP the station connected to
 } system_event_sta_connected_t;

//File esp_mesh.h

typedef struct
 {
  system_event_sta_connected_t connected; //Information on the parent's AP (as with the
                                            the SYSTEM_EVENT_STA_CONNECTED Wi-Fi event)
  uint8_t self_layer;                     //The node's layer in the mesh network
 } mesh_event_connected_t;

/*========================= MESH_EVENT_PARENT_DISCONNECTED =========================*/

//File esp_event.h

typedef struct                     //SYSTEM_EVENT_STA_DISCONNECTED event-specific info
 {
  uint8_t ssid[32];                //SSID of the AP the station disconnected from
  uint8_t ssid_len;                //SSID length of the AP the station disconnected from
  uint8_t bssid[6];                //BSSID of the AP the station disconnected from
  uint8_t reason;                  //The reason for the disconnection
 } system_event_sta_disconnected_t;

//File esp_mesh.h

typedef system_event_sta_disconnected_t mesh_event_disconnected_t;
//Information on the disconnected parent's AP
  (same as with the SYSTEM_EVENT_STA_DISCONNECTED Wi-Fi event)

/*============================ MESH_EVENT_NO_PARENT_FOUND ============================*/

//File esp_mesh.h

typedef struct
 {
  int scan_times;                   //The total number of scans performed searching
 } mesh_event_no_parent_found_t;     for a viable parent for the node
```

## Mesh Network Update Events

```
/*========================== MESH_EVENT_CHILD_CONNECTED ==========================*/

//File esp_event.h

typedef struct         //SYSTEM_EVENT_AP_STACONNECTED event-specific info
 {
  uint8_t mac[6];      //MAC address of the client that has connected to the SoftAP
  uint_t aid;          //The aid given by the SoftAP to the connected client
 } system_event_ap_staconnected_t;

//File esp_mesh.h

typedef system_event_ap_staconnected_t mesh_event_child_connected_t;
//Connected station interface information
  (same as with the SYSTEM_EVENT_AP_STACONNECTED Wi-Fi event)

/*========================== MESH_EVENT_CHILD_DISCONNECTED ==========================*/

//File esp_event.h

typedef struct      //SYSTEM_EVENT_AP_STADISCONNECTED event-specific info
 {
  uint8_t mac[6];   //MAC address of the client that has disconnected from the SoftAP
  uint_t aid;       //The aid given by the SoftAP to the disconnected client
 } system_event_ap_stadisconnected_t;

//File esp_mesh.h

typedef system_event_ap_stadisconnected_t mesh_event_child_disconnected_t;
//Disconnected station interface information
  (same as with the SYSTEM_EVENT_AP_STADISCONNECTED Wi-Fi event)

/*========================== MESH_EVENT_ROUTING_TABLE_ADD ==========================*/
/*========================== MESH_EVENT_ROUTING_TABLE_REMOVE ==========================*/

//File esp_mesh.h

typedef struct
 {
  uint16_t rt_size_change;          //The number of entries that were added or removed
                                      from the node's routing table
  uint16_t rt_size_new;             //The current number of entries in the node's
                                      routing table
 } mesh_event_routing_table_change_t;

/*============================== MESH_EVENT_ROOT_ADDRESS ==============================*/

//File esp_mesh.h

typedef mesh_addr_t mesh_event_root_address_t;   //The root node's SoftAP MAC address

/*============================== MESH_EVENT_ROOT_FIXED ==============================*/

//File esp_mesh.h

typedef struct
 {
  bool is_fixed;                   //The parent's and thus the node's
 } mesh_event_root_fixed_t;          new value of the fixed root setting
```

```
/*============================= MESH_EVENT_TODS_STATE ============================*/

//File esp_mesh.h

typedef enum                  //Whether the external DS is currently accessible or not
 {
  MESH_TODS_REACHABLE,        //External DS accessible
  MESH_TODS_UNREACHABLE,      //External DS not accessible
 } mesh_event_toDS_state_t;

/*============================= MESH_EVENT_VOTE_STARTED ============================*/

//File esp_mesh.h

typedef struct
 {
  int reason;           //The reason for the new root election (currently only due to
                          root node self-healing or the root waiving its root status)
  int attempts;         //The maximum number of rounds in the new root election
  mesh_addr_t rc_addr;  //The SoftAP MAC address of the node that was designated by the
                          current root to become the new root (currently unimplemented)
 } mesh_event_vote_started_t;

/*============================= MESH_EVENT_LAYER_CHANGE ============================*/

//File esp_mesh.h

typedef struct
 {
  uint8_t new_layer;            //The new node's layer in the mesh network
 } mesh_event_layer_change_t;

/*============================= MESH_EVENT_CHANNEL_SWITCH ============================*/

//File esp_mesh.h

typedef struct
 {
  uint8_t channel;              //The new Mesh Wi-Fi Channel
 } mesh_event_channel_switch_t;
```

**Mesh Root-specific Events**

```
/*============================= MESH_EVENT_ROOT_GOT_IP ============================*/

//File esp_event.h

typedef struct                    //SYSTEM_EVENT_STA_GOT_IP event-specific info
 {
  tcpip_adapter_ip_info_t ip_info; //IP configuration obtained by the station interface
  bool ip_changed;                //Whether this IP configuration
 } system_event_sta_got_ip_t;       overrode a previous one

//File esp_mesh.h

typedef system_event_sta_got_ip_t mesh_event_root_got_ip_t;
//The IP configuration obtained by the root node
  (same as with the SYSTEM_EVENT_STA_GOT_IP Wi-Fi event)
```

```
/*========================== MESH_EVENT_ROOT_SWITCH_REQ ==========================*/

//File esp_mesh.h

typedef struct
 {
  int reason;           //Reason for the switch request
                          (currently only because the root node waived its root status)
  mesh_addr_t rc_addr;  //The MAC address of the candidate root's Station interface
 } mesh_event_root_switch_req_t;

/*========================== MESH_EVENT_ROOT_ASKED_YIELD ==========================*/

//File esp_mesh.h

typedef struct
 {
  int8_t rssi;          //The requesting root's RSSI with the router
  uint16_t capacity;    //The number of nodes in the requesting root's network
  uint8_t addr[6];      //The MAC address of the requesting root's station interface
 } mesh_event_root_conflict_t;
```

## Mesh Events Additional Information union

```
typedef union           //Mesh Events additional information union
 {
  /*-- Mesh Main Setup Events additional information --*/
  mesh_event_connected_t connected;                   //MESH_EVENT_PARENT_CONNECTED
  mesh_event_disconnected_t disconnected;             //MESH_EVENT_PARENT_DISCONNECTED
  mesh_event_scan_done_t scan_done;                   //MESH_EVENT_SCAN_DONE
  mesh_event_no_parent_found_t no_parent;             //MESH_EVENT_NO_PARENT_FOUND

  /*-- Mesh Network Update Events additional information --*/
  mesh_event_child_connected_t child_connected;       //MESH_EVENT_CHILD_CONNECTED
  mesh_event_child_disconnected_t child_disconnected; //MESH_EVENT_CHILD_DISCONNECTED
  mesh_event_routing_table_change_t routing_table;    //MESH_EVENT_ROUTING_TABLE_ADD +
                                                        MESH_EVENT_ROUTING_TABLE_REMOVE
  mesh_event_root_address_t root_addr;                //MESH_EVENT_ROOT_ADDRESS
  mesh_event_root_fixed_t root_fixed;                 //MESH_EVENT_ROOT_FIXED
  mesh_event_toDS_state_t toDS_state;                 //MESH_EVENT_TODS_STATE
  mesh_event_vote_started_t vote_started;             //MESH_EVENT_VOTE_STARTED
  mesh_event_layer_change_t layer_change;             //MESH_EVENT_LAYER_CHANGE
  mesh_event_channel_switch_t channel_switch;         //MESH_EVENT_CHANNEL_SWITCH

  /*-- Mesh Root-specific Events additional information--*/
  mesh_event_root_got_ip_t got_ip;                    //MESH_EVENT_ROOT_GOT_IP
  mesh_event_root_switch_req_t switch_req;            //MESH_EVENT_ROOT_SWITCH_REQ
  mesh_event_root_conflict_t root_conflict;           //MESH_EVENT_ROOT_ASKED_YIELD
 } mesh_event_info_t;
```

## Mesh Events Summary Struct

This represents the summary struct that is passed by the mesh stack to the Mesh Events General Handler for the application-level handling of mesh events:

```
typedef struct              //Summary struct passed by the mesh stack
 {                            to the Mesh Events General Handler
  mesh_event_id_t id;       //Event ID
  mesh_event_info_t info;   //Event additional information (if applicable)
 } mesh_event_t;
```

From here at the application level the Mesh Events General Handler function should be declared as follows:

```
void mesh_events_handler(mesh_event_t event)
```

where the `event` parameter represents the summary struct passed by the mesh stack containing the information on the mesh event that has occured.

Regarding its definition, as discussed before the task of the Mesh Events General Handler consists in calling the specific handler relative to each event that occurs, passing it the additional information provided by the mesh stack where applicable, and therefore its general structure appears as follows:

```
void mesh_events_handler(mesh_event_t event)
 {
  switch(event.id)
   {
    case EVENT1:
     mesh_EVENT1_handler(&event.info.EVENT1_t); //call EVENT1 specific handler
     break;
    case EVENT2:
     mesh_EVENT2_handler(&event.info.EVENT2_t); //call EVENT2 specific handler
     break;
     …
    case EVENTN:
     mesh_EVENTN_handler(&event.info.EVENTN_t); //call EVENTN specific handler
     break;

    default:
     ESP_LOGE(TAG,"Unknown Mesh Event with ID: %u",event.id);
     break;
   }
  return;
 }
```

So, considering the mesh events that can currently be raised by the mesh stack, the actual definition of the Mesh Events General Handler appears as follows:

```
void mesh_events_handler(mesh_event_t event)
 {
  switch(event.id)
   {
    /*-- Mesh Main Setup Events --*/
    case MESH_EVENT_STARTED:
     mesh_STARTED_handler();
     break;
    case MESH_EVENT_STOPPED:
     mesh_STOPPED_handler();
     break;
    case MESH_EVENT_SCAN_DONE:
     mesh_SCAN_DONE_handler(&event.info.scan_done);
     break;
    case MESH_EVENT_PARENT_CONNECTED:
     mesh_PARENT_CONNECTED_handler(&event.info.connected);
     break;
    case MESH_EVENT_PARENT_DISCONNECTED:
     mesh_PARENT_DISCONNECTED_handler(&event.info.disconnected);
     break;
    case MESH_EVENT_NO_PARENT_FOUND:
     mesh_NO_PARENT_FOUND_handler(&event.info.no_parent);
     break;
```

```
        /*-- Mesh Network Update Events --*/
        case MESH_EVENT_CHILD_CONNECTED:
         mesh_CHILD_CONNECTED_handler(&event.info.child_connected);
         break;
        case MESH_EVENT_CHILD_DISCONNECTED:
         mesh_CHILD_DISCONNECTED_handler(&event.info.child_disconnected);
         break;
        case MESH_EVENT_ROUTING_TABLE_ADD:
         mesh_ROUTING_TABLE_ADD_handler(&event.info.routing_table);
         break;
        case MESH_EVENT_ROUTING_TABLE_REMOVE:
         mesh_ROUTING_TABLE_REMOVE_handler(&event.info.routing_table);
         break;
        case MESH_EVENT_ROOT_ADDRESS:
         mesh_ROOT_ADDRESS_handler(&event.info.root_addr);
         break;
        case MESH_EVENT_ROOT_FIXED:
         mesh_ROOT_FIXED_handler(&event.info.root_fixed);
         break;
        case MESH_EVENT_TODS_STATE:
         mesh_TODS_STATE_handler(&event.info.toDS_state);
         break;
        case MESH_EVENT_VOTE_STARTED:
         mesh_VOTE_STARTED_handler(&event.info.vote_started);
         break;
        case MESH_EVENT_VOTE_STOPPED:
         mesh_VOTE_STOPPED_handler();
         break;
        case MESH_EVENT_LAYER_CHANGE:
         mesh_LAYER_CHANGE_handler(&event.info.layer_change);
         break;
        case MESH_EVENT_CHANNEL_SWITCH:
         mesh_CHANNEL_SWITCH_handler(&event.info.channel_switch);
         break;

        /*-- Mesh Root-specific Events --*/
        case MESH_EVENT_ROOT_GOT_IP:
         mesh_ROOT_GOT_IP_handler(&event.info.got_ip);
         break;
        case MESH_EVENT_ROOT_LOST_IP:
         mesh_ROOT_LOST_IP_handler();
         break;
        case MESH_EVENT_ROOT_SWITCH_REQ:
         mesh_ROOT_SWITCH_REQ_handler(&event.info.switch_req);
         break;
        case MESH_EVENT_ROOT_SWITCH_ACK:
         mesh_ROOT_SWITCH_ACK_handler();
         break;
        case MESH_EVENT_ROOT_ASKED_YIELD:
         mesh_ROOT_ASKED_YIELD_handler(&event.info.root_conflict);
         break;

        default:
         ESP_LOGE(TAG,"Unknown Mesh Event with ID: %u",event.id);
         break;
      }
     return;
    }
```

It should also be noted that depending on the mesh stack's configuration previously set via the mesh initializer function and the logic of the driver module, some mesh events may never be raised by the mesh stack, thus making their application-level handling unnecessary in specific contexts.

## Mesh Events Specific Handlers

Following the previous definition of the Mesh Events General Handler, the mesh events specific handlers should be defined according to the following general structure:

```
void mesh_EVENTi_handler(EVENTi_t* info)
{
 /* Specific handler logic */
 return;
}
```

where the `info` argument must be present only if additional information is provided by the mesh stack for the event, which as discussed before is passed by the Mesh Events General Handler to the specific handler in question.

### Utility Functions

The following utility functions will be used in the follow-up analysis of the mesh events specific handlers:

```
/* Resets the node's mesh status and the flags in the mesh event group to a
   set of default values representing a node that is not connected to a parent */

void mesh_reset_status()
{
 EventBits_t eventbits;      //Used to check the flags in the mesh event group

 memset(mesh_state->root_addr,0,6);        //Reset the root address
 memset(mesh_state->parent_addr,0,6);      //Reset the node's parent address
 mesh_state->my_layer = -1;                //Reset the node's mesh layer
 if(((eventbits = xEventGroupGetBits(mesh_event_group))>>2)&1) //Reset the node's
  mesh_state->my_type = MESH_NODE;                              type depending on
 else                                                           it having children
  mesh_state->my_type = MESH_IDLE;                              connected or not
 xEventGroupClearBits(mesh_event_group,MESH_PARENT|MESH_TODS); //Clear the MESH_PARENT
                                                                and MESH_TODS flags
 xEventGroupSetBits(mesh_event_group,MESH_VOTE);              //Set the MESH_VOTE flag
 return;                                                       (no root election)
}


/* Resets the root node's station IP configuration (and implicitly its DNS Servers) */

void mesh_root_reset_stationIP()
{
 tcpip_adapter_ip_info_t ipinfo;         //Used to set an interface's IP configuration

 if(MESH_ROOT_IPDYNAMIC)     //Prevents an error from happening later should the node
  ESP_ERROR_CHECK(tcpip_adapter_dhcpc_stop(TCPIP_ADAPTER_IF_STA));  become root again

 //Reset the root node's station IP configuration
 inet_pton(AF_INET,"0.0.0.0",&ipinfo.ip);
 inet_pton(AF_INET,"0.0.0.0",&ipinfo.netmask);
 inet_pton(AF_INET,"0.0.0.0",&ipinfo.gw);
 ESP_ERROR_CHECK(tcpip_adapter_set_ip_info(TCPIP_ADAPTER_IF_STA,&ipinfo));
 return;
}
```

Where the `tcpip_adapter_set_ip_info()` function allows to set an interface's IP configuration, and is defined as follows:

```
//File tcpip_adapter.h

esp_err_t tcpip_adapter_set_ip_info(tcpip_adapter_if_t tcpip_if,
                                    tcpip_adapter_ip_info_t* ip_info)
```

| Parameters | |
|---|---|
| tcpip_if | The interface for which to apply the IP configuration<br>❑ TCPIP_ADAPTER_IF_STA → Station interface<br>❑ TCPIP_ADAPTER_IF_AP → SoftAP interface |
| ip_info | The address of the struct holding the IP configuration to apply to the interface |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_TCPIP_ADAPTER_INVALID_PARAMS | Invalid argument(s) |
| ESP_FAIL | Unknown error in the Wi-Fi stack |

## Mesh Events Specific Handlers Typical Actions

Described below are the typical actions that should be performed by each mesh specific handler:

### Mesh Main Setup Events

#### MESH_EVENT_STARTED

This event is raised when mesh networking has been enabled on a device by calling the `esp_mesh_start()` function and, in addition to setting the MESH_ON flag in the mesh event group and resetting the node's status via the `mesh_reset_status()` utility function previously described, further action depends on the node's mesh organization mode as follows:

- **Self-Organized Networking**
  If self-organized networking is used on the node, the implicit handler of this event would have already started the automatic procedure to join the node in an existing mesh network or create a new one as previously described, and so no further action is required in the application-level handling of this event.

- **Fixed-Root Networking**
  If fixed-root networking is used, a node must determine according to a user's custom criteria if it represents the fixed root of the network or not, and in case it does it should promptly attempt to manually connect to the external router using the following function:

```
//File esp_wifi_types.h (automatically included by the previous headers)

typedef struct                  //Station Mode Configuration
 {
  uint8_t ssid[32];             //SSID of the AP to connect to
  uint8_t password[64];         //Password of the AP to connect to
  bool bssid_set;               //If set, connect only to the AP with
                                  the following specific BSSID
  uint8_t bssid[6];             //Specific BSSID of the AP to connect to
  uint8_t channel;              //AP's channel
  …                             /* Other station mode-related members */
 } wifi_sta_config_t;

typedef union                  //Holds a Wi-Fi interface mode configuration
 {                               (Station OR SoftAP)
  wifi_sta_config_t sta;       //Station Mode Configuration <---
  wifi_ap_config_t ap;         //SoftAP Mode Configuration
 } wifi_config_t;
```

```
//File esp_mesh.h

typedef enum    //Mesh node types
 {
  MESH_IDLE,    //Idle Node
  MESH_ROOT,    //Root Node
  MESH_NODE,    //Intermediate Parent Node
  MESH_LEAF,    //Leaf Node
 } mesh_type_t;
```

```
esp_err_t esp_mesh_set_parent(const wifi_config_t* parent,
                              const mesh_addr_t* my_mid,
                              mesh_type_t my_type,
                              int my_layer)
```

| Parameters | |
|---|---|
| parent | The AP parameters of the parent to connect to (which correspond to the node's base station configuration) |
| my_mid | The node's Mesh Network Identifier (MID) *(optional)* |
| my_type | The node's expected type should it successfully connect to such parent |
| my_layer | The node's expected mesh layer should it successfully connect to such parent |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_MESH_ARGUMENT | Invalid argument(s) |
| ESP_ERR_MESH_NOT_CONFIG | The base mesh configuration is not set (call esp_mesh_set_config() first) |
| ESP_FAIL | Unknown error in the mesh stack |

It should be noted, as already pointed out, that if the external router is not in range or its AP settings don't match the ones set in the fixed root's router configuration, the node will remain stuck in a loop attempting and failing to connect with it.

Otherwise, if the node doesn't recognize itself to be the fixed-root of the mesh network, no further action is required in the application-handling of this event, since its implicit handler would have already started to search for the node's *preferred parent* to connect to (where note that if none is found the node will keep searching indefinetely for a viable parent node to connect to).

- **Manual Networking**

If manual networking is used, a manual Wi-Fi scan must be performed to search for the mesh nodes and/or the external router, which can be carried out by calling the following function:

```
//File esp_wifi_types.h

typedef enum                    //Wi-Fi Scan types
 {
  WIFI_SCAN_TYPE_ACTIVE = 0,   //Active Wi-Fi scan (scan by sending a probe request)
  WIFI_SCAN_TYPE_PASSIVE,      //Passive Wi-Fi scan (scan by waiting for a beacon
 } wifi_scan_type_t;             frame without explicitly sending a probe request)

typedef struct                  //Active scan time per Wi-Fi channel
 {
  uint32_t min;                //The minimum active scan time per Wi-Fi channel
                                 (default = 120ms)
  uint32_t max;                //The maximum active scan time per Wi-Fi channel
 } wifi_active_scan_time_t;      (default = 120ms, must be <=1500ms)

typedef struct                      //Scan time per Wi-Fi channel
 {
  wifi_active_scan_time_t active;  //Active scan time per Wi-Fi channel
  uint32_t passive;                //Passive scan time per Wi-Fi channel
 } wifi_scan_time_t;                 (must be <=1500ms)
```

```
typedef struct                  //Wi-Fi Scan configuration
 {
  uint8_t* ssid;                //Whether to scan for an AP with a specific SSID only
  uint8_t* bssid;               //Whether to scan for an AP with a specific BSSID only
  uint8_t channel;              //Whether to scan on a specific Wi-Fi channel only
                                  (1-13) or perform an all-channel scan (0, default)
  bool show_hidden;             //Whether to include the APs with a hidden SSID
                                  in their Wi-Fi beacon frames in the scan results
  wifi_scan_type_t scan_type; //The type of the Wi-Fi scan to perform
                                  (active or passive)
  wifi_scan_time_t scan_time; //The scan time for each Wi-Fi channel
 } wifi_scan_config_t;
```

```
esp_err_t esp_wifi_scan_start(const wifi_scan_config_t* scan_conf,
                              bool block)
```

| Parameters | |
|---|---|
| scan_conf | The address of the struct holding the configuration of the Wi-Fi scan to perform |
| block | Whether the function should block until the Wi-Fi scan is completed |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_WIFI_NOT_INIT | The Wi-Fi stack is not initialized (call esp_wifi_init() first) |
| ESP_ERR_WIFI_TIMEOUT | The blocking scan timeout has ended (unimplemented) |
| ESP_ERR_WIFI_STATE | Wi-Fi internal state error |
| ESP_FAIL | Unknown error in the Wi-Fi stack |

Where note that the Wi-Fi scan should be limited to the Mesh Wi-Fi channel, the APs with hidden SSID must be included in its results, and performing a passive scan is preferred.

```
void mesh_STARTED_handler()
{
 wifi_config_t parent_ap = {0}; //Used to connect the fixed root to the external router
 wifi_scan_config_t scan_config = {0}; //Used to store the configuration of the Wi-Fi
                                         scan to perform in Manual Networking mode
 xEventGroupSetBits(mesh_event_group,MESH_ON);  //Set the MESH_ON flag
 mesh_reset_status();                           //Reset the node's status
 switch(mesh_state.org)                         //Further actions depend on the Mesh
  {                                               Organization Mode set on the node
   case SELF_ORGANIZED:                         //For Self-Organized networking,
    break;                                        no further action is required

   case FIXED_ROOT:
    /* Determine according to a user's custom criteria if the
       node represents the fixed-root of the mesh network or not */
    if(IS_NODE_FIXED_ROOT)            //If the node is the fixed-root, it must attempt
     {                                  to directly connect to the external router
      strcpy((char*)parent_ap.sta.ssid,MESH_ROUTER_SSID);
      strcpy((char*)parent_ap.sta.password,MESH_ROUTER_PASSWORD);
      if(MESH_ROUTER_SPECIFIC_BSSID)
       {
        parent_ap.sta.bssid_set = true;
        strToMAC(MESH_ROUTER_BSSID,parent_ap.sta.bssid);
       }
      parent_ap.sta.channel = MESH_WIFI_CHANNEL;
      ESP_ERROR_CHECK(esp_mesh_set_parent(&parent_ap,MESH_NETWORK_ID,MESH_ROOT,1));
     }
    break;    //If the node is NOT the fixed root, no further action is required

   case MANUAL_NETWORKING:
    //Set the configuration of the Wi-Fi scan to perform
    scan_config.show_hidden = 1;  //Include APs with hidden SSID in the scan
                                    results (to allow mesh nodes to be found)
    scan_config.channel = MESH_WIFI_CHANNEL;  //Scan on the Mesh Wi-Fi Channel only
    scan_config.scan_type = WIFI_SCAN_TYPE_PASSIVE; //Passive Wi-Fi scans should be
                                                      used in manual mesh networking
    ESP_ERROR_CHECK(esp_wifi_scan_start(&scan_config,false)); //Start the Wi-Fi scan
    break;
  }
 return;
}
```

## MESH_EVENT_STOPPED

This event may be raised both because mesh networking was disabled intentionally by calling the `esp_mesh_stop()` function (we'll see later), or due to a fatal error in the mesh stack, and other than resetting the node's mesh status and clearing the MESH_ON flag in the mesh event group, should the disabling have occured unintentionally, as an error recovery attempt it is possible to try to re-enable mesh networking by calling the `esp_mesh_start()` function.
Also note that the handling of errors that might occur in the application's logic due to the disabling of mesh networking is left entirely to the Driver Module.

```
void mesh_STOPPED_handler()
{
 mesh_reset_status();                           //Reset the node's status
 xEventGroupClearBits(mesh_event_group,MESH_ON);  //Clear the MESH_ON flag
 if(/* unintentional */)                        //If the disabling was unintentional,
  ESP_ERROR_CHECK(esp_mesh_start());              try to reenable mesh networking
 return;
}
```

## MESH_EVENT_SCAN_DONE (Manual Networking only)

This event is raised when the mesh stack completes a manual Wi-Fi scan requested by calling the `esp_wifi_scan_start()` function, and in its application-level handler to search for a suitable parent to attempt to connect the node to, for each of the APs that were found in the scan the length of its vendor IE field must be retrieved, which is obtained by calling the following function:

```
//File esp_mesh.h

esp_err_t esp_mesh_scan_get_ap_ie_len(int* ie_len)
```

| Parameters | |
|---|---|
| ie_len | The address where to copy the length of the AP's vendor IE |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_WIFI_NOT_INIT | The Wi-Fi stack is not initialized (call esp_wifi_init() first) |
| ESP_ERR_WIFI_ARG | Invalid argument(s) |
| ESP_ERR_WIFI_FAIL | Wi-Fi internal state error |
| ESP_FAIL | Unknown error in the mesh stack |

From here by checking the length of the AP's vendor IE against the length of a mesh IE, which is defined as follows:

```
typedef struct        //Node's Mesh IE (extract)
 {
  uint8_t mesh_id[6]; //The node's MID
  uint8_t mesh_type;  //The node's type
  uint8_t layer_cap;  //The mesh maximum layer of the node minus its current layer
  uint8_t layer;      //The current layer of the node in the mesh network
  uint8_t assoc_cap;  //The node's maximum children connections
  uint8_t assoc;      //The number of children currently connected to the node
  …
 } __attribute__((packed)) mesh_assoc_t;
```

- If the AP's vendor IE is the same length as a mesh IE, that AP is relative to a mesh node, and its information and Mesh IE can be retrieved by calling the following function:

```
//File esp_wifi.h

typedef struct   //Information (record) on an AP that was found in the Wi-Fi scan
 {
  uint8_t ssid[33];            //AP SSID
  uint8_t bssid[6];            //AP BSSID
  wifi_auth_mode_t authmode;   //AP Authmode
  uint8_t primary;             //AP (primary) Wi-Fi channel
  int8_t rssi;                 //RSSI with the AP
  …
 } wifi_ap_record_t;
```

```
//File esp_mesh.h

esp_err_t esp_mesh_scan_get_ap_record(wifi_ap_record_t* ap_rec,
                                      void* ap_ie)
```

| Parameters | |
|---|---|
| ap_rec | The address where to copy the AP's record |
| ap_ie | The address where to copy the AP's vendor IE (Mesh IE) |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_WIFI_NOT_INIT | The Wi-Fi stack is not initialized (call esp_wifi_init() first) |
| ESP_ERR_WIFI_ARG | Invalid argument(s) |
| ESP_ERR_WIFI_FAIL | Wi-Fi internal state error |
| ESP_FAIL | Unknown error in the mesh stack |

From here, by checking its AP record and Mesh IE, it can be determined whether a connection attempt to such node can be performed according to the following conditions:

- The node belongs to the same mesh network (`ap_ie.mesh_id == MESH_NETWORK_ID`)
- The node is not an idle node (`ap_ie.mesh_type != MESH_IDLE`) (where note that in this instance in self-organized networking the initial root election is triggered between the idle nodes)
- The node is not a leaf node (`ap_ie.layer_cap > 0`)
- The node's maximum children connections hasn't been reached (`ap_ie.assoc_cap > ap_ie.assoc`)
- Preferably, the RSSI with the node is above a defined minimum threshold (`ap_rec.rssi > MESH_PARENT_MIN_RSSI`)

- Otherwise, if its vendor IE is not the same length as a Mesh IE, the AP is relative to an external network, possibly the external router, which can be determined by retrieving the AP record using the same `esp_mesh_scan_get_ap_record()` function described before, this time with a second NULL argument since the vendor IE is of no interest, and then checking whether the AP's SSID and possibly its BSSID correspond to the SSID and the possible specific BSSID that were set in the router settings in the node's base mesh configuration.
  From here, if an external AP has been determined as being relative to the external router, no particular conditions must be met if attempting to connect to it as the root node is desidered, apart from possibly having an RSSI above a defined minimum threshold (`ap_rec.rssi > MESH_PARENT_MIN_RSSI`).

From here, once the records and the Mesh IEs of all APs that were found in the scan have been parsed, by applying a user's custom criteria it must be determined whether and which one of the APs the node should attempt to connect to as a child, an attempt that can be performed by using the `esp_mesh_set_parent()` function described previously with the following considerations:

**Connecting to a Node**:

- In the "`parent`" argument representing the node's station base configuration the "ssid" member must be set to the SSID that was passed by the mesh stack in the parent's AP record (which was previously transparently retrieved and decrypted from its Mesh IE), while the "password" member must be set to an empty string since, as discussed before, mesh nodes use no authentication protocol on their SoftAP interfaces.

- The "`my_layer`" argument must be set to the mesh layer of the parent node + 1 (`ap_ie.layer + 1`).

- The "`my_type`" argument must be set to either MESH_NODE or MESH_LEAF depending on the node's expected mesh layer ("`my_layer`") and the maximum layer allowed in the mesh network (MESH_MAX_LAYER)

**Connecting to the External Router**:

- The "parent" argument representing the node's station base configuration must be initialized with the information found in the router's AP record.
- The "my_layer" argument must be set to 1 (for the node will represent the root of the mesh network should it successfully connect to the external router)
- The "my_type" argument must be set to MESH_ROOT for the same reason as above.

Finally, if no suitable parent to attempt to connect the node to was found, a new Wi-Fi scan should be performed after a defined time interval, repeating this process until a viable parent for the node is found.

```
void mesh_SCAN_DONE_handler(mesh_event_scan_done_t* info)
{
 wifi_scan_config_t scan_config = {0};   //Configuration of the new Wi-Fi scan to
                                           perform if no parent is found for the node
 //Parent connection attempt variables
 bool parent_found = false;             //Whether a viable parent for the node was found
 wifi_config_t parent = {0};            //Information on the parent's AP to connect
                                          to (this node's station base configuration)
 int my_layer = -1;                     //The supposed mesh layer of the node should
                                          it successfully connect to its parent
 mesh_type_t my_type = MESH_IDLE;       //The supposed type of the node should it
                                          successfully connect to its parent
 //Information on the APs that were found in the scan
 wifi_ap_record_t ap_rec;     //Used to store the records of the APs that were found
 mesh_assoc_t ap_ie;          //Used to store the Mesh IEs of the nodes that were found
 int ap_ie_len = 0;           //Length of the vendor IEs of the APs that were found

 for(int i=0;i<info.number;i++)  //For each of the APs that were found in the scan
  {
   esp_mesh_scan_get_ap_ie_len(&ap_ie_len); //Retrieve the length of the AP vendor IE

   //If the AP is relative to a mesh node
   if(ap_ie_len == sizeof(mesh_assoc_t))
    {
     esp_mesh_scan_get_ap_record(&ap_rec,&ap_ie); //Retrieve the AP record and Mesh IE
     if((checkArrayEqual(ap_ie.mesh_id,MESH_NETWORK_ID,6) &&
         (ap_ie.mesh_type != MESH_IDLE) &&
         (ap_ie.layer_cap) &&
         (ap_ie.assoc >= ap_ie.assoc_cap) &&
         (ap_rec.rssi >= MESH_PARENT_MIN_RSSI))
      /* A connection attempt to this node is possible */
      …
    }

   //Otherwise if it's relative to an external network
   else
    {
     esp_mesh_scan_get_ap_record(&ap_rec,NULL);  //Retrieve the AP record only
     if((!strcmp((char*)ap_rec.ssid,MESH_ROUTER_SSID)) &&
         ((!MESH_ROUTER_SPECIFIC_BSSID)  ||
          (checkArrayEqual(ap_rec.bssid,MESH_ROUTER_BSSID,6))) &&
         (ap_rec.rssi >= MESH_PARENT_MIN_RSSI))
      /* This external network corresponds to the external
         router and a connection attempt is possible       */
      …
    }
  }  //end for
```

```
    /* Determine the parent to attempt to connect the node to
       (if any) and initialize the parent connection attempt variables */

    if(parent_found)  //If a suitable parent was found for the node, attempt a connection
     ESP_ERROR_CHECK(esp_mesh_set_parent(&parent,MESH_NETWORK_ID,my_type,my_layer));
    else              //Otherwise perform a new Wi-Fi scan after a defined time interval
     {
      scan_config.show_hidden = 1;    //Include APs with hidden SSID in the scan
                                        results (to allow mesh nodes to be found)
      scan_config.channel = MESH_WIFI_CHANNEL;     //Scan on the Mesh Wi-Fi Channel only
      scan_config.scan_type = WIFI_SCAN_TYPE_PASSIVE;   //Passive Wi-Fi scans should be
                                                          used in manual mesh networking
      vTaskDelay(WIFI_SCAN_RETRY_INTERVAL/portTICK_PERIOD_MS); //Wait for a time interval
      ESP_ERROR_CHECK(esp_wifi_scan_start(&scan_config,false)); //Start the Wi-Fi scan
     }
}
```

## MESH_EVENT_PARENT_CONNECTED

Once the node successfully connects to a parent in the mesh network, in addition to setting the MESH_PARENT flag in the mesh event group and updating its parent address and mesh layer, the latter also defines the node's type according to the following conditions:

- `If(my_layer == 1)`, the node represents the root node of the mesh network (MESH_ROOT)
- `If(1 < my_layer < MESH_MAX_LAYER)`, the node represents an intermediate parent node (MESH_NODE)
- `If(my_layer == MESH_MAX_LAYER)`, the node represents a leaf node (MESH_LEAF)

From here, in addition to updating the node's type accordingly, if the node is the root an IP configuration must be applied to its station interface to allow it to communicate with the external router, which can be retrieved dynamically by re-enabling its station DHCP client or by applying a predefined static IP configuration, the latter of which can be set by using the `tcpip_adapter_set_ip_info()` function described previously and, if setting the DNS server addresses for the station interface is also desidered, this can be obtained via the following function:

```
//File tcpip_adapter.h

esp_err_t tcpip_adapter_set_dns_info(tcpip_adapter_if_t tcpip_if,
                                     tcpip_adapter_dns_type_t type,
                                     tcpip_adapter_dns_info_t* addr)
```

| Parameters | |
|---|---|
| tcpip_if | The interface for which to set a DNS server address<br>❑ TCPIP_ADAPTER_IF_STA → Station interface<br>❑ TCPIP_ADAPTER_IF_AP → SoftAP interface |
| type | The type of DNS server to set for the interface<br>❑ TCPIP_ADAPTER_DNS_MAIN → Primary DNS server<br>❑ TCPIP_ADAPTER_DNS_BACKUP → Secondary DNS server |
| addr | The IP address of the DNS server |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_TCPIP_ADAPTER_INVALID_PARAMS | Invalid argument(s) |
| ESP_FAIL | Unknown error in the Wi-Fi stack |

Lastly, since upon receving this event the node is effectively connected to a mesh network and so its main setup phase is over, its internal driver components, i.e. the driver components that don't the require the node to access the external DS, can be started.

```
void mesh_PARENT_CONNECTED_handler(mesh_event_connected_t* info)
{
 tcpip_adapter_ip_info_t ipinfo; //Possibly used to set a custom static IP
                                 configuration for the root node's station interface
 tcpip_adapter_dns_info_t dnsaddr; //Possibly used to set the DNS servers addresses
                                   of the root node's station interface

 xEventGroupSetBits(mesh_event_group,MESH_PARENT);       //Set the MESH_PARENT flag
 memcpy(mesh_state.parent_addr,info->connected.bssid,6); //Update the node's parent MAC
 mesh_state.my_layer = info->self_layer;                 //Update the node's mesh layer
 if(mesh_state.my_layer == 1)        //If the node is the root, an IP configuration
  {                                    must be applied to its station interface to
   mesh_state.my_type = MESH_ROOT;     allow it to communicate with the external router
   if(MESH_ROOT_IPDYNAMIC) //If a dynamic IP configuration is used, enable the station
    ESP_ERROR_CHECK(tcpip_adapter_dhcpc_start(TCPIP_ADAPTER_IF_STA)); DHCP client
   else                    //Otherwise if a static IP configuration is used, apply it
    {
     //Set the Root station static IP configuration
     inet_pton(AF_INET,MESH_ROOT_STATIC_IP,&ipinfo.ip);
     inet_pton(AF_INET,MESH_ROOT_STATIC_NETMASK,&ipinfo.netmask);
     inet_pton(AF_INET,MESH_ROOT_STATIC_GATEWAY,&ipinfo.gw);
     ESP_ERROR_CHECK(tcpip_adapter_set_ip_info(TCPIP_ADAPTER_IF_STA,&ipinfo)); //Apply

     //Set the Root station DNS servers (if desidered)
     inet_pton(AF_INET,MESH_ROOT_STATIC_DNS_PRIMARY,&dnsaddr.ip);
     ESP_ERROR_CHECK(tcpip_adapter_set_dns_info(TCPIP_ADAPTER_IF_STA,  //Primary DNS
                                                TCPIP_ADAPTER_DNS_MAIN,&dnsaddr));
     inet_pton(AF_INET,MESH_ROOT_STATIC_DNS_SECONDARY,&dnsaddr.ip);
     ESP_ERROR_CHECK(tcpip_adapter_set_dns_info(TCPIP_ADAPTER_IF_STA,  //Secondary DNS
                                                TCPIP_ADAPTER_DNS_BACKUP,&dnsaddr));
    }
  }
 else //If the node is not root, just update its type
  if(mesh_state.my_layer < MESH_MAX_LAYER)  //INTERMEDIATE PARENT node
    mesh_state.my_type = MESH_NODE;
  else                                     //LEAF node (my_layer == MESH_MAX_LAYER)
    mesh_state.my_type = MESH_LEAF;
 startMeshInternalDrivers();  //Start the node's internal driver components
 return;
}
```

Also note that once a node connects to a parent node in the mesh network, the parent passes its child the following information in this order:

- The MAC address of the root node's SoftAP interface (triggering the MESH_EVENT_ROOT_ADDRESS event)
- Its fixed-root setting (which overrides the one set on the child should they differ, triggering the MESH_EVENT_ROOT_FIXED event)
- If the parent had previously been informed of it, that the external DS is accessible (triggering the MESH_EVENT_TODS_STATE event)

## MESH_EVENT_PARENT_DISCONNECTED

This event is raised when a node disconnects from its parent or when it fails to connect or automatically reconnect to it (we'll see in more detail later when this happens), where the two circumstances can be discriminated by checking and appropriately managing the MESH_PARENT flag in the mesh event group.
From here, while this event's implicit handler will attempt to reconnect the node to its parent, at the application-level, should the node have disconnected from it:

- If the node is root it must reset its station IP configuration by calling `mesh_root_reset_stationIP()` utility function described previously, and furthermore it must inform the mesh network that the external DS is no longer accessible, which can be obtained by calling the following function:

```
//File esp_mesh.h

esp_err_t esp_mesh_post_toDS_state(bool reachable)
```

| Parameters | |
|---|---|
| reachable | Whether the external DS is currently accessible or not |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_FAIL | Unknown error in the mesh stack |

- The node's mesh status must be reset, which can be obtained by calling the `mesh_status_reset()` utility function described earlier, which will also cause its MESH_PARENT flag to be cleared in order to discriminate the semantics of further instances of this event as discussed above.

If, on the other hand, the node failed to connect or automatically reconnect to its parent no actions are required, and note that in Manual Networking mode, while this event's implicit handler will still attempt to reconnect the node to its parent, if desidered it's possible to start a Wi-Fi scan to search for another parent to fall back to.

Also note that, again, the handling of errors that might occur in the application's logic due to the node having disconnected from its parent (and thus from the mesh network) is left entirely to the Driver Module.

```
void mesh_PARENT_DISCONNECTED_handler(mesh_event_disconnected_t* info)
{
 EventBits_t eventbits;    //Used to check the flags in the mesh event group
 wifi_scan_config_t scan_config = {0};  //Configuration of the Wi-Fi scan to perform
                                          be performed to fall back on another parent
                                          (Manual Networking only)
 //If the MESH_PARENT flag is set, the node disconnected from its parent
 if(((eventbits = xEventGroupGetBits(mesh_event_group))>>1)&1)
  {
   if(mesh_state.my_type == MESH_ROOT)                    //If the node is root, inform
    {                                                       the mesh network that the
     ESP_ERROR_CHECK(esp_mesh_post_toDS_state(false));      external DS is no longer
     mesh_root_reset_station_IP();                          accessible and reset its
    }                                                       station IP configuration
   mesh_reset_status();                                   //Reset the node's status
  }
 //Otherwise if the node failed to connect or
   reconnect to its parent, no action is required
```

```
if((mesh_state.org == MANUAL_NETWORKING)&&(/* a fallback is desidered */))
  {
   ESP_ERROR_CHECK(esp_wifi_scan_stop());  //Stop a Wi-Fi scan already in progress
   //Set the configuration of the Wi-Fi scan to perform to search for a fallback parent
   scan_config.show_hidden = 1;  //Include APs with hidden SSID in the scan
                                 results (to allow mesh nodes to be found)
   scan_config.channel = MESH_WIFI_CHANNEL;  //Scan on the Mesh Wi-Fi Channel only
   scan_config.scan_type = WIFI_SCAN_TYPE_PASSIVE; //Passive Wi-Fi scans should be
                                                   used in manual mesh networking
   ESP_ERROR_CHECK(esp_wifi_scan_start(&scan_config,false)); //Start the Wi-Fi scan
  }
 return;
}
```

Note that in general, should the disconnection from a node's parent have occured unintentionally (i.e. not due to automatic routines such as the finding of a better preferred parent for the node or a node being elected root that disconnects from its parent to connect with the router), the mesh stack will perform a fixed number of attempts to reconnect the node to its previous parent before searching for another preferred parent to connect to (self-organized networking and for the non-fixed-root nodes of a fixed-root network) or will attempt indefinetely to reconnect the node to its previous parent (manual networking and fixed-root nodes in a fixed-root network).

## MESH_EVENT_NO_PARENT_FOUND

This event is raised in a self-organized network and for the non-fixed-root nodes in a fixed-root network should a node fail to find a viable parent to attempt to connect to in a fixed number or scans.
From here, while the mesh stack will keep performing periodic scans to search for a viable parent for the node, in the application-level handling of this event it's possible to implement advanced error-recovery procedures such as performing an all-channel Wi-Fi scan searching for an available parent on a different Mesh Wi-Fi Channel and/or changing the node's mesh configuration.
Note that in any case the node's main setup phase and consequently the program's execution cannot proceed until a viable parent is found.

```
void mesh_NO_PARENT_FOUND_handler(mesh_event_no_parent_found_t* info)
 {
  /* Possibly perform error recovery procedures to find
     a mesh network or an external router to connect to */
  return;
 }
```

## Mesh Network Update Events

### MESH_EVENT_CHILD_CONNECTED

This event is raised when a child successfully connects to a node, and apart from setting the MESH_CHILD flag in the mesh event group if it was not previously set (i.e. if it's the first child to connect to the node), no other action is required in the application-level handler of this event.

```
void mesh_CHILD_CONNECTED_handler(mesh_event_child_connected_t* info)
{
 EventBits_t eventbits;       //Used to check the flags in the mesh event group

  //If the MESH_CHILD flag is not set in the mesh event group
    (i.e. this is the first child node to connect), set it
  if(!(((eventbits = xEventGroupGetBits(wifi_event_group))>>2)&1))
   xEventGroupSetBits(mesh_event_group,MESH_CHILD);
  return;
 }
```

Also note that after a child connects to a parent, its own and its descendants' station interfaces MAC addresses will be added to the parent's routing table, triggering the MESH_EVENT_ROUTING_TABLE_ADD event.

### MESH_EVENT_CHILD_DISCONNECTED

This event is raised when a child disconnects from a node, and other than clearing the MESH_CHILD flag in the mesh event group if it was the last child to disconnect from the node, no other action is required in the application-level handler of this event.

```
void mesh_CHILD_DISCONNECTED_handler(mesh_event_child_disconnected_t* info)
{
 wifi_sta_list_t children;                //Used to store information on the clients
                                            connected to the node's SoftAP interface
 ESP_ERROR_CHECK(esp_wifi_ap_get_sta_list(&children));  //(see below)
 if(children.num == 0)                                  //If it was the last child
  xEventGroupClearBits(mesh_event_group,MESH_CHILD);      to disconnect, clear the
 return;                                                  MESH_CHILD flag
}
```

Where the `esp_wifi_ap_get_sta_list()` function, which allows to retrieve information on the clients (children nodes) connected to the node's SoftAP interface, is defined as follows:

```
//File esp_wifi_types.h

typedef struct     //Information on a specific client (child)
 {                    connected to the node's SoftAP interface
  uint8_t mac[6];  //Client's (child's) MAC address
  …
 } wifi_sta_info_t;

typedef struct     //Information on the clients (children)
 {                   connected to the node's SoftAP interface
  wifi_sta_info_t sta[ESP_WIFI_MAX_CONN_NUM];//Information on each client (child)
  int num;                                    //Number of clients (children) connected
  …                                             to the node's SoftAP interface
 } wifi_sta_list_t;
```

```
//File esp_wifi.h
```

```
esp_err_t esp_wifi_ap_get_sta_list(wifi_sta_list_t* stalist)
```

| Parameters | |
|---|---|
| stalist | The address where to copy the information related to the clients connected to the SoftAP interface |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_WIFI_NOT_INIT | The Wi-Fi stack is not initialized (call esp_wifi_init() first) |
| ESP_ERR_WIFI_MODE | The SoftAP interface is not enabled |
| ESP_ERR_WIFI_CONN | Wi-Fi internal error or SoftAP control block wrong |
| ESP_ERR_INVALID_ARG | Invalid argument |
| ESP_FAIL | Unknown error in the Wi-Fi stack |

Also note that after a child disconnects from its parent node, its own and its descendants' station interfaces MAC addresses will be removed from the parent's routing table, triggering the MESH_EVENT_ROUTING_TABLE_REMOVE event.

## MESH_EVENT_ROUTING_TABLE_ADD

This event is raised after a node's descendant (with its possible further descendants) joins the mesh network, whose station MAC addresses are added to the node's routing table by this event's implicit handler, while at the application-level handling, apart from some possible logging, no other action is required.

```
void mesh_ROUTING_TABLE_ADD_handler(mesh_event_routing_table_change_t* info)
 {
  /* Possible logging */
  return;
 }
```

## MESH_EVENT_ROUTING_TABLE_REMOVE

This event is raised after a node's descendant (with its possible further descendants) disconnects from the mesh network, whose station MAC addresses are removed from the node's routing table by this event's implicit handler, while at the application-level handling, apart from some possible logging, no other action is required.

```
void mesh_ROUTING_TABLE_REMOVE_handler(mesh_event_routing_table_change_t* info)
 {
  /* Possible logging */
  return;
 }
```

## MESH_EVENT_ROOT_ADDRESS

This event is raised after a node connects to its parent, which passes it the root's SoftAP MAC address, and whenever a root switch occurs in the mesh network, where the new root's SoftAP MAC address is propagated from parent to children through the entire network.
From here, apart from updating the root node's MAC address in the node's mesh status, no further actions are required in the application-level handling of this event.

```
void mesh_ROOT_ADDRESS_handler(mesh_event_root_address_t* info)
 {
  memcpy(mesh_state.root_addr,info->addr,6);      //Update the root node's MAC address
  return;                                           in the node's mesh status
 }
```

## MESH_EVENT_ROOT_FIXED

This event is raised after a node connects to its parent if their fixed-root settings differ, where this event's implicit handler will change its value to match the one of its parent.
From here, at the application-level handling, the mesh organization mode on the node should be updated to reflect the new value of its fixed root setting, where note that if fixed-root networking is set in this way the node can only be a non-fixed-root node.

```
void mesh_ROOT_FIXED_handler(mesh_event_root_fixed_t* info)
{
 if(info->is_fixed)             //Change the node's mesh organization mode accordingly
  mesh_state.org = FIXED_ROOT;
 else
  mesh_state.org = SELF_ORGANIZED;
 return;
}
```

## MESH_EVENT_TODS_STATE

This event is raised after a node connects to its parent if the latter was previously informed that the external DS is reachable, or whenever the root node posts an update on its accessibility via the `esp_mesh_post_toDS_state()` function, information that is propagated from parent to children through the entire network.
From here, in the application-level handler, if the node is informed that the external DS is reachable, in addition to setting the MESH_TODS flag in the mesh event group its external driver components can be started, otherwise, if the external DS is no longer reachable, the MESH_TODS flag must be cleared, again leaving the handling of the possible errors that might occur in the application's logic due to the external DS being not longer accessible entirely to the Driver Module.

```
void mesh_TODS_STATE_handler(mesh_event_toDS_state_t* info)
{
 if(*info == MESH_TODS_REACHABLE)                    //If the external DS is reachable
  {
   xEventGroupSetBits(mesh_event_group,MESH_TODS);  //Set the MESH_TODS flag
   startMeshExternalDrivers();                      //Start external driver components
  }
 else                                              //If the external DS is NOT reachable
  xEventGroupClearBits(mesh_event_group,MESH_TODS); //Clear the MESH_TODS flag
 return;
}
```

## MESH_EVENT_VOTE_STARTED (Self-Organized Network Only)

This event is raised when a new root election is started in a self-organized network, which may be caused by the root node self-healing process or by the root node waiving its root status (we'll see later).
From here, the implicit handler of this event will carry out the election process along with the other nodes in the network, while at the application level, other than clearing the MESH_VOTE flag (since it's active-low) no further action is required.

```
void mesh_VOTE_STARTED_handler(mesh_event_vote_started_t* info)
{
 xEventGroupClearBits(mesh_event_group,MESH_VOTE);     //Clear the MESH_VOTE flag
 return;                                                 (for it's active low)
}
```

## MESH_EVENT_VOTE_STOPPED (Self-Organized Network Only)

This event is raised once a new root election in a self-organized network ends, where the event's implicit handler will check whether the voting percentage threshold on the node has been reached, and if this is the case:

- If the root election was triggered due to the root node self-healing process, the node will disconnect from its current parent and attempt to connect to the router as the root node.
- If the root election was triggered as a result of the current root node waiving its root status, the node will send it a root switch request (triggering the MESH_EVENT_ROOT_SWITCH_REQ event on the current root).

From here at the application level, apart from setting the MESH_VOTE flag (since it's active-low) no further action is required.

```
void mesh_VOTE_STOPPED_handler()
{
 xEventGroupSetBits(mesh_event_group,MESH_VOTE);    //Set the MESH_VOTE flag
 return;                                              (for it's active low)
}
```

## MESH_EVENT_LAYER_CHANGE

This event is raised whenever a network topology change affects the ancestors of a node, which other than changing its mesh layer may also cause its type to change from intermediate parent to leaf or vice versa, where if this is the case the event's implicit handler will modify the node's possibility to accept children nodes appropriately. From here in the application-level handler, apart from updating the node's mesh layer and possibly its type in its mesh status, no further action is required.

```
void mesh_LAYER_CHANGE_handler(mesh_event_layer_change_t* info)
 {
  mesh_state.my_layer = info->new_layer;  //Update the node's mesh layer
  //Check whether the layer change causes the node's type
    to change from INTERMEDIATE PARENT to LEAF or vice versa
  if((mesh_state.my_layer < MESH_MAX_LAYER)&&(mesh_state.my_type != MESH_NODE))
   mesh_state.my_type = MESH_NODE;
  else
   if(mesh_state.my_layer == MESH_MAX_LAYER)
    mesh_state.my_type = MESH_LEAF;
  return;
 }
```

Also note that the node's current children won't be disconnected if its type changes to leaf node, and that changes to the root's type are not taken into account since when a node becomes root it always switches parents (to the router), and so its type will be updated in the MESH_EVENT_PARENT_CONNECTED event specific handler.

## MESH_EVENT_CHANNEL_SWITCH

This event is raised should the external router switch its AP onto a different Wi-Fi channel, which will cause the root node's station and thus its SoftAP interfaces to switch onto such channel, which in turn will cause all the nodes in the mesh network to switch their Wi-Fi interfaces in cascade onto the new channel, which so will represent the new Mesh Wi-Fi Channel.
The actual Mesh Wi-Fi Channel switch is performed by the implicit handler of this event, while at the application level, apart from updating such information in the node's mesh status, no further action is required.

```
void mesh_CHANNEL_SWITCH_handler(mesh_event_channel_switch_t* info)
{
 mesh_state.channel = info->channel;        //Update the node's Mesh Wi-Fi Channel
 return;
}
```

## Mesh Root-specific Events

### MESH_EVENT_ROOT_GOT_IP

This event is raised on the root node once it obtains an IP configuration for its station interface, which as discussed previously in the MESH_EVENT_PARENT_CONNECTED handler can be retrieved both dynamically from its station DHCP client or by applying a predefined static IP configuration.
From here, in the application-level handler of this event, in addition to starting the root driver components the mesh network must be informed that the router and consequently the external DS is now accessible.

```
void mesh_ROOT_GOT_IP_handler(mesh_event_root_got_ip_t* info)
 {
  startMeshRootDrivers();                              //Start root driver components
  ESP_ERROR_CHECK(esp_mesh_post_toDS_state(true));  //Inform the mesh network that the
  return;                                                 external DS is accessible
 }
```

### MESH_EVENT_ROOT_LOST_IP

This event is raised on the current or a former root node once the lease time of its station dynamic IP configuration expires (which for the current root node also implies that its DHCP client failed to renew or otherwise retrieve a new IP configuration for the interface).
From here if the node is still the root it won't be able to communicate with the external router until its station DHCP client retrieves a new IP configuration for the interface, and thus the mesh network must be informed that the external DS is currently no longer accessible.

```
void mesh_ROOT_LOST_IP_handler()
 {
  if(mesh_state.my_type == MESH_ROOT)                 //If the node is the current root
   ESP_ERROR_CHECK(esp_mesh_post_toDS_state(false)); //Inform the mesh network that
  return;                                                 the external DS is no longer
 }                                                        accessible
```

### MESH_EVENT_ROOT_SWITCH_REQ (Self-Organized Network Only)

This event is raised on the root node when it receives a root switch request from a candidate root node following the election caused by the root waiving its root status, and while this event's implicit handler will send back a root switch acknowledgment to the candidate root, disconnect the node from the router and start searching for a preferred parent to connect to, in the application-level handler the root station IP configuration must be reset and the node's type must be changed to intermediate parent (a temporary change since shortly afterwards the node will disconnect from its parent).

```
void mesh_ROOT_SWITCH_REQ_handler(mesh_event_root_switch_req_t* info)
 {
  mesh_root_reset_stationIP();        //Reset the root node's station IP configuration
  mesh_state.my_type = MESH_NODE;     //Change the node's type from
  return;                                root to intermediate parent
 }
```

## MESH_EVENT_ROOT_SWITCH_ACK (Self-Organized Network Only)

This event is raised on the candidate root node when it receives the root switch acknowledgment from the former root following the election caused by it waiving its root status, and while this event's implicit handler will disconnect the node from its current parent and attempt to connect it to the router as the new root, in the application-level handler apart from possible logging no action is required.

```
void mesh_ROOT_SWITCH_ACK_handler()
 {
  /* Possible logging */
  return;
 }
```

## MESH_EVENT_ROOT_ASKED_YIELD

Whenever a root conflict is detected in a mesh network, after an internal communication between all the root nodes, the one with the highest RSSI with the router will ask the others to yield, causing this event to be raised. From here while the event's implicit handler will disconnect the node from the router and search for a preferred parent to connect to, in the application-level handler as with the MESH_EVENT_ROOT_SWITCH_REQ specific handler the root station IP configuration must be reset and the node's type must be changed to intermediate parent (a temporary change since shortly afterwards the node will disconnect from its parent).
Also note that if this event raises on a fixed-root of a fixed-root network, the node will permanently change to a non-fixed-root node.

```
void mesh_ROOT_ASKED_YIELD_handler(mesh_event_root_conflict_t* info)
 {
  mesh_root_reset_stationIP();       //Reset the root node's station IP configuration
  mesh_state.my_type = MESH_NODE;    //Change the node's type from
  return;                              root to intermediate parent
 }
```

# Driver Module

Following our previous analysis of the Mesh Events Specific Handlers, depending on their category the components constituting the Driver Module should be started:

- The **Internal Driver Components**, i.e. the driver components that don't require a node to communicate with the external DS, as soon as the main setup phase on a node is finished, that is as soon as the node connects to a parent in the mesh network (MESH_EVENT_PARENT_CONNECTED).

- The **External Driver Components**, i.e. the driver components that require a node to communicate with the the external DS, as soon as the node is informed that it is reachable (MESH_EVENT_TODS_REACHABLE with *info == MESH_TODS_REACHABLE).

- The **Root Driver Components**, i.e. the driver components that are to be executed exclusively on the root node of the network, as soon as the root obtains an IP configuration for its station interface (MESH_EVENT_ROOT_GOT_IP).

Note that, since the events that start the driver components can generally be raised multiple times during the application's execution, it's necessary for the setup module to keep track of which driver components are currently being executed to avoid creating undesidered duplicate tasks of the same components, and this can be obtained by providing in the mesh event group a flag for each driver component representing whether it is currently running or not, as was shown previously in the code premises section.

From here the actual semantics of the synchronization of the execution of the driver module components is left to the programmer, where a simple yet blunt solution is given, in the specific handlers of events that start driver components, by previously checking and killing any existing tasks relative to the components they would start before actually creating them again, which can be obtained as follows:

Example

```
void InternalDriverComponent_A()
{
 /* internal driver component A logic */
 …
 //Clear this driver component's execution flag in the mesh event group
 xEventGroupClearBits(mesh_event_group,MESH_DRIVER_INTERNAL_A);
 vTaskDelete(NULL);              //Delete this driver component's task
}

void InternalDriverComponent_B()
{
 /* internal driver component B logic */
 …
 //Clear this driver component's execution flag in the mesh event group
 xEventGroupClearBits(mesh_event_group,MESH_DRIVER_INTERNAL_B);
 vTaskDelete(NULL);              //Delete this driver component's task
}

…
```

```
/* Called at the end of the MESH_EVENT_PARENT_CONNECTED event specific handler */
void startMeshInternalDrivers()
{
 EventBits_t eventbits;           //Used to check the flags in the mesh event group
 static TaskHandle_t componentA_handler;    //ComponentA driver task handler
 static TaskHandle_t componentB_handler;    //ComponentB driver task handler

 /* For each driver component, if its relative task is running, kill it
    and set its flag in the mesh event group before starting its task   */

 if(((eventbits = xEventGroupGetBits(mesh_event_group))>>MESH_DRIVER_INTERNAL_A)&1)
  vTaskDelete(componentA_handler);
 xEventGroupSetBits(mesh_event_group,MESH_DRIVER_INTERNAL_A);
 xTaskCreate(internalDriver_A,"internaldriverA",4096,NULL,10,&componentA_handler);

 if(((eventbits = xEventGroupGetBits(mesh_event_group))>>MESH_DRIVER_INTERNAL_B)&1)
  vTaskDelete(componentB_handler);
 xEventGroupSetBits(mesh_event_group,MESH_DRIVER_INTERNAL_B);
 xTaskCreate(internalDriver_B,"internaldriverB",4096,NULL,10,&componentB_handler);
}
```

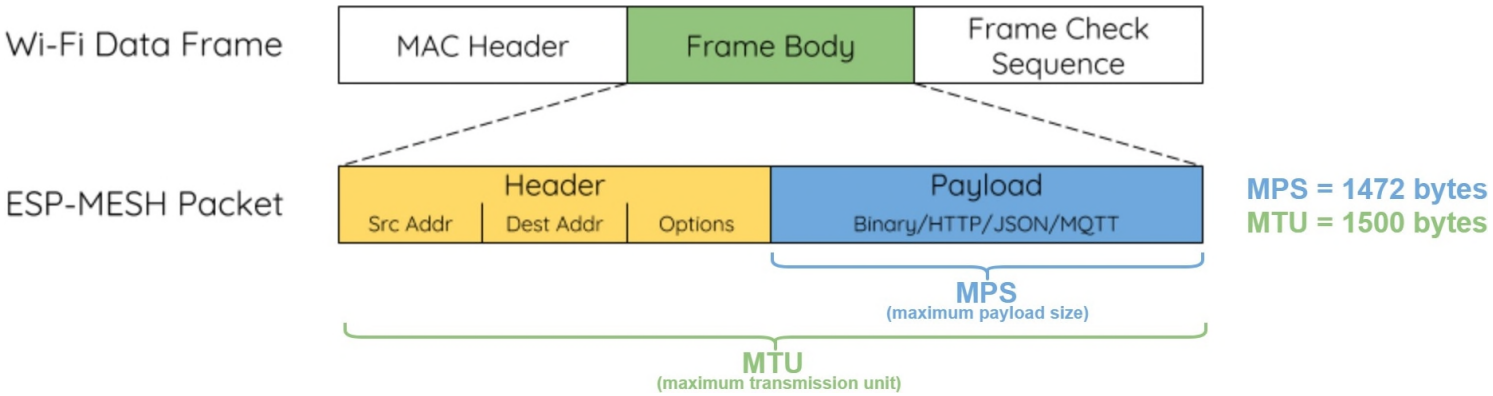
Also note that, as with all networking applications, the driver components should include routines for handling errors that may occur during their execution, which can be triggered by checking the return values of the Mesh API functions used and/or the appropriate flags in the mesh event group, whose values as we have seen are asynchronously updated by the appropriate event specific handlers during the application's execution.

# Mesh API for the Driver Module

Listed below are the functions offered by the Mesh API organized into categories that can be used for developing the driver module of a mesh application:

## Mesh Network Communication

As described in the Mesh API Guide the communication between nodes in a mesh network is implemented through the exchange of **Mesh Packets**, which represent data structures that can be embedded entirely into the body of a Wi-Fi frame as follows:




In addition to its payload, which represents the relevant information exchanged between nodes, a mesh packet is comprised of the following information:

- The **Source Address**, representing the station MAC address of the node inside the mesh network the packet originated from, and is implemented in the code via a `mesh_addr_t` union (or more precisely its "addr" member).
  Note that packets destined for a node whose data originated from a host outside the mesh network carry the source address of the root node, while the actual address of the external host is carried within an additional option in the packet (we'll see later).

- The **Destination Address**, representing the address of the host the packet is destined for, which for nodes inside the mesh network consists in their Station MAC address, while for external hosts in their IPv4:PORT combination, destination address that again is implemented in the code via a `mesh_addr_t` union.
  Note that packets destined for an external host are delivered by the mesh network to the root node, where the data will be extracted from the packet and sent to the external router by using IP sockets.

- A **Flag**, consisting of an integer used as a bitmap to associate a set of properties to a packet as follows:

```
//File esp_mesh.h

/*-- Mesh Flag Properties --*/
#define MESH_DATA_P2P       (0x02)  //Denotes that the packet is destined for
                                      a non-root node inside the mesh network
#define MESH_DATA_FROMDS    (0x04)  //Denotes that the packet contains data
                                      which originated from an external host
#define MESH_DATA_TODS      (0x08)  //Denotes that the packet is
                                      destined for an external host
#define MESH_DATA_NONBLOCK  (0x10)  //Denotes that the non-blocking variant of the
                                      esp_mesh_send() or esp_mesh_recv() function
                                      must be used to send/receive the packet
#define MESH_DATA_DROP      (0x20)  //Denotes that the packet's data is not of
                                      critical importance and thus can be
                                      discarded in case of congestion on
                                      a newly elected root node
#define MESH_DATA_GROUP     (0x40)  //Denotes that the packet is destined for all
                                      nodes belonging to a certain mesh group
                                      (multicast)
```

- A **Packet Data Descriptor**, holding a set of information related to the packet's data and represented by the following data structure:

```
//File esp_mesh.h

typedef enum           //Identifies the protocol the payload
 {                       of the mesh packet belongs to
  MESH_PROTO_BIN,     //Binary Data (default)
  MESH_PROTO_HTTP,    //HTTP protocol
  MESH_PROTO_JSON,    //JSON protocol
  MESH_PROTO_MQTT,    //MQTT protocol
 } mesh_proto_t;

typedef enum           //Defines the type of service that must
 {                       or was used to deliver the mesh packet
  MESH_TOS_P2P,       //Reliable point-to-point transmission (default)
  MESH_TOS_E2E,       //Reliable end-to-end transmission (currently unimplemented)
  MESH_TOS_DEF,       //Unreliable transmission
 } mesh_tos_t;

typedef struct         //Mesh Packet Data Descriptor
 {
  uint8_t* data;      //Pointer to the send/receive buffer
  uint16_t size;      //Size of the data to send/receive or that has been received
  mesh_proto_t proto; //The protocol the payload of the mesh packet belongs to
  mesh_tos_t tos;     //The type of service that must or
 } mesh_data_t;         was used to deliver the mesh packet
```

- A list of **Additional Options** associated with the packet, each is represented by the following data structure:

```
//File esp_mesh.h

/*-- Mesh Packet Additional Option Types --*/
#define MESH_OPT_SEND_GROUP   (7)     //Defines that the additional option's value
                                        represents the list of MAC addresses which
                                        the packet is destined for (this is a
                                        multicast method alternative to sending
                                        packets to a specific mesh group)
#define MESH_OPT_RECV_DS_ADDR (8)     //Defines that the additional option's
                                        value represents the IPv4:PORT address
                                        of the external host the data in
                                        the mesh packet originated from

typedef struct     //Mesh Packet Additional Option Descriptor
 {
  uint8_t type;    //The additional option's type (either of the above)
  uint8_t* val;    //The address where the additional
                     option's value is (or must be) stored
  uint16_t len;    //The length in bytes of the additional option's value
 } __attribute__((packed)) mesh_opt_t;  //__attribute__((packed)) is for byte
                                          alignment purposes of the in-memory
                                          representation of the struct
```

Also note that in the current ESP-IDF version only up to 1 additional option is supported per mesh packet.

## Mesh Packets Routing Algorithm

The routing algorithm used by nodes to forward mesh packets they receive on their Station or SoftAP interfaces consists in the following:

1) If the packet's destination address matches the node's Station or SoftAP MAC addresses, the packet is copied into the related receiving queue (we'll see later) in the mesh stack.
2) If the packet's destination address is found in the node's routing table, the packet is sent to the node's child whose sub-routing table contains such MAC address (we'll see in more detail later).
3) If the packet's destination address is not found in the node's routing table, the packet is sent to the node's parent, except for the root node where it is discarded since no route was found to deliver it.

## Notes on Mesh Communication

- Since it is the station MAC address of nodes that is propagated upwards in the routing table of their ancestors, when sending to another node inside the network its station MAC address must be used as destination address (except for the root node, see later) to ensure that a route is found to deliver the packet, otherwise, as with the routing algorithm above, the packet may reach the root node where it is discarded since no route was found to deliver it (note however that a node still recognizes itself as the target of a packet destined for its SoftAP MAC address, if such packet happens to pass through the node itself).

- Since the ESP-IDF mesh stack currently offers no advertisement system on the nodes connected to a mesh network, each node is aware only of the presence of its descendants whose addresses are stored in its routing table, with the root being the only node that is aware of the presence of every node connected to the mesh network.
  Furthermore, since a non-root node has no means of acknowledging the presence or the MAC address of a node which is not its descendant, to send it a packet its MAC address must be known prior to the program's execution (moreover there is also no guarantee that the node is actually connected to the network, with the risk of wasting network resources in transmitting packets that travel up to the root node before being dropped since no route was found to deliver them).

- As described also in the Mesh API Guide, nodes in a mesh network implement an **Upstream Flow Control** mechanism, where a parent node allocates to each of its children a receiving window, which they must apply for before they are allowed to tramit data upstream to their parent.
  From here, in case of congestion in the network or if packets destined for a parent are not parsed in time by its application level, its receving window relative to one or more children may be depleted, which will prevent the nodes from transmitting data upstream in the network until the congestion is mitigated or the application level in the parent parses the received data.

- **Send a packet over the mesh network**

  A node can send a mesh packet destined for another node inside the network or to an external host by calling the following function:

```
//File esp_mesh.h

esp_err_t esp_mesh_send(const mesh_addr_t* to,
                        const mesh_data_t* data,
                        int flag,
                        const mesh_opt_t opt[],
                        int opt_count)
```

| Parameters | |
|---|---|
| to | The packet's destination address (node's MAC address or external host's IPv4:PORT) |
| data | The address of the packet's data descriptor |
| flag | The bitmap of properties to apply to the packet |
| opt | An array of additional options for the packet (if any) |
| opt_count | The number of additional options for the packet (currently only 0 or 1 is supported) |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_MESH_ARGUMENT | Invalid argument(s) |
| ESP_ERR_MESH_NOT_START | Mesh networking is not enabled (call esp_mesh_start() first) |
| ESP_ERR_MESH_DISCONNECTED | The packet's next hop has disconnected from the mesh network |
| ESP_ERR_MESH_OPT_UNKNOWN | Unknown additional packet option |
| ESP_ERR_MESH_EXCEED_MTU | The data to send exceeds the maximum payload size (= 1472 bytes) |
| ESP_ERR_MESH_NO_MEMORY | Out of memory |
| ESP_ERR_MESH_TIMEOUT | The function timeout has triggered |
| ESP_ERR_MESH_QUEUE_FULL | The node's sending queue is full |
| ESP_ERR_MESH_NO_ROUTE_FOUND | No route was found to forward the packet |
| ESP_ERR_MESH_DISCARD | The packet was discarded by the mesh stack |
| ESP_FAIL | Unknown error in the mesh stack |

Where note that if sending a packet to the root node the "to" argument must be NULL.

The usage of the esp_mesh_send() function is best described through the following example:

Example

```
void SenderDriver()
 {
  esp_err_t send_ret;              //Used to store the result of the send()
  uint8_t send_buffer[MESH_MPS];   //Send Buffer (MESH_MPS = 1472 bytes)
  mesh_addr_t dest;                //Packet's destination address
  mesh_data_t data_des;            //Packet's data descriptor
  int send_flag = 0;               //Packet's flag
  mesh_opt_t send_opt;             //Possible additional send option
  uint8_t opt_val[50];             //Possible additional option's value

  //1) Copy the data to send (the payload) into the sending buffer
  memcpy(send_buffer,PACKET_CONTENTS,sizeof(PACKET_CONTENTS));

  //2) Packet Data Description Initialization
  data_des.data = send_buffer;                  //Address of the sending buffer
  data_des.size = sizeof(PACKET_CONTENTS);      //Size of the packet's payload
  data_des.proto = PACKET_DATA_PROTOCOL;        //Payload's protocol
  data_des.tos = PACKET_TYPE_OF_SERVICE;        //Packet's type of service
```

```
    //3) Send Flag Initialization
    if(SEND_NONBLOCK)                //If we want to use the non-blocking
     send_flag += MESH_DATA_NONBLOCK;  version of the esp_mesh_send() function
    if(SEND_NONCRITICAL)             //If the packet to send is not of critical
     send_flag += MESH_DATA_DROP;       importance and so can be dropped in case
                                        of congestion on a newly elected root node

    //4) Destination-specific initializations and send() calls

    /*================== Send to a specific (non-root) node ==================*/
    send_flag += MESH_DATA_P2P;          //Defines that the packet is destined
                                           for a non-root node inside the network
    memcpy(dest.addr,DEST_NODE_MAC,6);   //Set the destination address to the
                                           node's station MAC Address
    send_ret = esp_mesh_send(&dest,&data_des,send_flag,NULL,0); //Send the packet

    /*========================= Send to the Root Node =========================*/
    send_ret = esp_mesh_send(NULL,&data_des,send_flag,NULL,0); //Send the packet
    (Note that the "dest" argument must be NULL when sending to the root node)

    /*========================= Send to a Mesh Group ==========================*/
    send_flag += MESH_DATA_P2P;          //Defines that the packet is destined
                                           for a non-root node inside the network
    send_flag += MESH_DATA_GROUP;        //Define that the packet is destined
                                           for a mesh group
    memcpy(dest.addr,DEST_GROUP_GID,6);  //Set the destination address to the
                                           destination mesh group's ID
    send_ret = esp_mesh_send(&dest,&data_des,send_flag,NULL,0); //Send the packet

    /*======================== Send to an External Host =======================*/
    send_flag += MESH_DATA_TODS;         //Define that the packet's destination
                                           is outside the mesh network
    inet_pton(AF_INET,DEST_HOST_IP,&dest.mip.ip4);  //Set the destination IP
    dest.mip.port = DEST_HOST_PORT;                 //Set the destination port
    send_ret = esp_mesh_send(&dest,&data_des,send_flag,NULL,0); //Send the packet

                         Forward data coming from an external
    /*================== host to a specific node (root only) ==================*/
    send_flag += MESH_DATA_P2P;          //Defines that the packet is destined
                                           for a non-root node inside the network
    send_flag += MESH_DATA_FROMDS;       //Define that the data in the packet
                                           originated from an external host
    memcpy(opt_val,SOURCE_IPV4_PORT,6);  //Copy the IPv4:PORT address of the
                                           external source to the option's buffer
    send_opt.type = MESH_OPT_RECV_DS_ADDR; //Defines that the additional option's
                                             value represents the IPv4:PORT
                                             address of the external host the
                                             data in the packet originated from
    send_opt.val = send_opt;             //Set the address of the option's value
    send_opt.len = 6;                    //Set the length of the option's value
    memcpy(dest.addr,DEST_NODE_MAC,6);   //Set the packet's destination address
    //Send the packet (with the additional option)
    send_ret = esp_mesh_send(&dest,&data_des,send_flag,&send_opt,1);

    //5) Check the result of the send operation
    if(send_ret != ESP_OK)
     {
      /* Possible error recovery operations */
     }
}
```

- **Receive a packet at the application-level**

  While packets destined for a node are autonomously received by its mesh stack and stored in the appropriate receiving queue, to pass such packets at the application level the following function must be called:

```
//file esp_mesh.h

esp_err_t esp_mesh_recv(mesh_addr_t* from,
                        mesh_data_t* data,
                        int timeout_ms,
                        int* flag,
                        mesh_opt_t opt[],
                        int opt_count)
```

| Parameters | |
|---|---|
| from | The address where to copy the packet's source address (node's MAC address or external host's IPv4:PORT) |
| data | The address where to copy the received packet's data descriptor |
| timeout_ms | The function timeout if there are no packets to receive (0 = non-blocking, portMAX_DELAY = blocking) |
| flag | The address where to copy the received packet's flag |
| opt | The address where to copy the received packet's additional option (if any) |
| opt_count | The number of additional options to receive (currently only 0 or 1 is supported) |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_MESH_ARGUMENT | Invalid argument(s) |
| ESP_ERR_MESH_NOT_START | Mesh networking is not enabled (call esp_mesh_start() first) |
| ESP_ERR_MESH_TIMEOUT | The function timeout has triggered |
| ESP_ERR_MESH_DISCARD | The packet was discarded by the mesh stack |
| ESP_FAIL | Unknown error in the mesh stack |

Where the "size" member of the mesh_data_t struct pointed by the data argument must be initialized beforehand to the maximum data length in bytes that can be received at the application level, i.e. the size of the receiving buffer, while once the function has been called such attribute will hold the actual length of the received data.

Note that packets whose data originated from an external host can be distinguished from packets whose data originated from inside the mesh network by checking the MESH_DATA_FROMDS bit in the received packet's flag and by the presence of the additional option of type "MESH_OPT_RECV_DS_ADDR", which as discussed before is used to attach the source's IPv4:PORT to the packet.

The following example describes the usage of the esp_mesh_recv() function:

Example

```
void ReceiverDriver()
 {
  esp_err_t recv_ret;            //Used to store the result of the recv()
  uint8_t recv_buffer[MESH_MTU]; //Receive Buffer (MESH_MTU = 1500 bytes)
  mesh_addr_t src = {0};         //Received packet's source address
  mesh_data_t data_des = {0};    //Received packet's data descriptor
  int recv_flag = 0;             //Received packet's flag
  mesh_opt_t recv_opt;           //Received packet's additional option
  uint8_t opt_val[50];           //Received packet's additional option's value

  //1) Set the address of the receive buffer
  data_des.data = recv_buffer;

  //2) Set the address of the buffer where to store the value of an additional
       option in the received packet (if any)
  recv_opt.val = opt_val;
```

```
        //Receive Loop
        while(1)
         {
          //3) Reset the "size" attribute of the packet's data descriptor to the data
               length in bytes that can be received at the application level, i.e.
               the size of the receive buffer
          data_des.size = MESH_MTU;

          //4) Reset the additional option received in the previous packet, if any
          recv_opt.type = 0;

          //5) Receive the packet
          recv_ret = esp_mesh_recv(&src,&data_des,RECV_TIMEOUT,
                                   &recv_flag,&recv_opt,1)

          //6) Check the result of the receive operation
          if(recv_ret != ESP_OK)
           {
            /* Possible error recovery operations */
           }
          else
           {
            //7) If there were no errors in the receive, parse the received data
            if(((recv_flag>>3)&1)||(recv_opt.type == MESH_OPT_RECV_DS_ADDR))
             {
              /* The data in the packet originated from an external host */
             }
            else
             {
              /* The data in the packet originated from
                 another node inside the mesh network */
             }
            …
           }
         }
       }
```

- **Retrieve the number of packets pending to be sent in the mesh stack's sending queues**

  The packets pending to be sent by a node are organized into multiple queues in the mesh stack, where the number of packets pending in each queue can be retrieved by calling the following function:

```
//file esp_mesh.h

typedef struct          //Number of packets pending to be sent in
 {                        each of the mesh stack's sending queues
  int to_parent;        //Parent sending queue
  int to_parent_p2p;    //Parent (P2P) sending queue
  int to_child;         //Child sending queue
  int to_child_p2p;     //Child (P2P) sending queue
  int mgmt;             //Management sending queue
  int broadcast;        //Broadcast and multicast sending queue
 } mesh_tx_pending_t;
```

```
esp_err_t esp_mesh_get_tx_pending(mesh_tx_pending_t* pending)
```

| Parameters | |
|---|---|
| pending | The addess of the struct where to copy the number of packets pending to be sent by the mesh stack |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_FAIL | Unknown error in the mesh stack |

Example

```
void appDriver()
{
  …
  mesh_tx_pending_t tx_pending;    //Stores the number of packets pending to be
                                     sent in each of the mesh stack's queues
  ESP_ERROR_CHECK(esp_mesh_get_tx_pending(&tx_pending));
  …
}
```

- **Retrieve the number of packets pending to be received at the application level**

The packets pending to be received at the application level are organized in the mesh stack into a queue relative to the packets destined for the node and into a special TODS queue exclusive to the root node relative to the packets that are to be forwarded to the external DS.

```
//file esp_mesh.h

typedef struct    //Number of packets pending to be received at the application
 {                  level in each of the mesh stack's receiving queues
  int toSelf;     //Receiving queue of packets destined for the node
  int toDS;       //Receiving queue of packets destined for the external DS
                    (root node only)
 } mesh_rx_pending_t;
```

```
esp_err_t esp_mesh_get_rx_pending(mesh_rx_pending_t* pending)
```

| Parameters | |
|---|---|
| pending | The addess of the struct where to copy the number of packets pending to be received at the application level |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_FAIL | Unknown error in the mesh stack |

Example

```
void appDriver()
{
  …
  mesh_rx_pending_t rx_pending;   //Stores the number of packets pending to
                                   be received at the application level
  ESP_ERROR_CHECK(esp_mesh_get_rx_pending(&rx_pending));
  …
}
```

## Mesh Root-specific API

The following API can be called only on the root node of a mesh network, and so should be used in root driver components only.

- ### Receive a packet destined for the external DS at the application level

  The packets destined for a remote host can be received at the application level on the root node by calling the following function, which is for the most part analogous to the `esp_mesh_recv()` function:

```
//file esp_mesh.h

      esp_err_t esp_mesh_recv_toDS(mesh_addr_t* from,
                                   mesh_addr_t* to,
                                   mesh_data_t* data,
                                   int timeout_ms,
                                   int* flag,
                                   mesh_opt_t opt[],
                                   int opt_count)
```

| Parameters | |
|---|---|
| from | The address where to copy the packet's source address (a node's MAC address) |
| to | The data contained in the packet's destination address (an external host's IPv4:PORT) |
| data | The address where to copy the received packet's data descriptor |
| timeout_ms | The function timeout if there are no packets to receive (0 = non-blocking, portMAX_DELAY = blocking) |
| flag | The address where to copy the received packet's flag |
| opt | The address where to copy the received packet's additional option (if any) |
| opt_count | The number of additional options to receive (currently only 0 or 1 is supported) |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_MESH_ARGUMENT | Invalid argument(s) |
| ESP_ERR_MESH_NOT_START | Mesh networking is not enabled (call `esp_mesh_start()` first) |
| ESP_ERR_MESH_TIMEOUT | The function timeout has triggered |
| ESP_ERR_MESH_DISCARD | The packet was discarded by the mesh stack |
| ESP_FAIL | Unknown error in the mesh stack |

Where note that in the received packet's flag the MESH_DATA_TODS bit will always be set and that currently no packet additional option is involved or of use in sending a packet towards an external host.

Example

```
void ReceiverDSDriver()
 {
  esp_err_t recv_ret;             //Used to store the result of the recv()
  uint8_t recv_buffer[MESH_MTU]; //Receive Buffer (MESH_MTU = 1500 bytes)
  mesh_addr_t src = {0};          //Received packet's source address(node's MAC)
  mesh_addr_t dest;               //Packet's destination address (IPv4:PORT)
  mesh_data_t data_des = {0};     //Received packet's data descriptor
  int recv_flag = 0;              //Received packet's flag
  mesh_opt_t recv_opt;            //Received packet's additional option
  uint8_t opt_val[50];            //Received packet's additional option's value

  //1) Set the address of the receive buffer
  data_des.data = recv_buffer;

  //2) Set the address of the buffer where to store the value of an additional
       option in the received packet (if any)
  recv_opt.val = opt_val;
```

```
    //Receive Loop
    while(1)
     {
      //3) Reset the "size" attribute to the size of the receive buffer
      data_des.size = MESH_MTU;

      //4) Reset the additional option received in the previous packet
      recv_opt.type = 0;

      //5) Receive the packet whose data is destined for an external host
      recv_ret = esp_mesh_recv_toDS(&src,&dest,&data_des,RECV_TIMEOUT,
                                    &recv_flag,&recv_opt,1)

      //6) Check the result of the receive operation
      if(recv_ret != ESP_OK)
       {
        /* Possible error recovery operations */
       }
      else
       {
        //7) If there were no errors in the receive, parse the received
             data and forward it to the external router via IP sockets
        …
       }
     }
   }
```

Once the data destined for a remote host has been received on the root node at the application level, it can be forwarded to the external router by using IP sockets, where a complete mechanism allowing a full bidirectional communication between the nodes in the mesh network and the external IP network, with the possibility of maintaining active sockets on the root node on behalf of the other nodes, requires additional communication protocols inside the network and possibly a **Port Address Translation (PAT)** mechanism on the root, topics that are beyond the purposes of this guide.

- **Trigger a New Root Election (Self-Organized Network Only)**

  In a self-organized network once a node has been elected root, apart from the root self-healing and root conflicts resolution procedures the node will remain the root indefinetely, even if other nodes with a higher RSSI with the router connect to the network or if its own RSSI with the router degrades.
  From here a good practice, should the root node's RSSI with the router fall below an acceptable threshold, is for it to trigger a new root node election, which is obtained by calling the following function:

```
//file esp_mesh.h

typedef union              //Root switch request type-specific configuration
 {
  int attempts;            //Maximum rounds in the new root election (default = 15)
  mesh_addr_t rc_addr;     //The address of a node appointed to be the next
 } mesh_rc_config_t;         root by the current root node (UNIMPLEMENTED)

typedef struct             //Root switch request configuration
 {
  bool is_rc_specified;    //Root switch request type where:
                             - false: A new root election is triggered (default)
                             - true:  The new root node is appointed directly by
                                      the current root (UNIMPLEMENTED)
  mesh_rc_config_t config; //Root switch request type-specific configuration
  float percentage;        //The election's voting percentage threshold for a
                             node to proclaim itself root (default = 0.9)
 } mesh_vote_t;
```

```
esp_err_t esp_mesh_waive_root(const mesh_vote_t* vote, int reason)
```

| Parameters | |
|---|---|
| vote | The address of the struct holding the root switch request configuration to be used |
| reason | The reason for the root switch request |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_MESH_QUEUE_FULL | The device's sending queue is full |
| ESP_ERR_MESH_DISCARD | The packet was discarded by the mesh stack |
| ESP_FAIL | Unknown error in the mesh stack |

Where the "reason" argument currently allows as value only the "MESH_VOTE_REASON_INITIATED" constant.

Once the function returns successfully a new root election will start in the mesh network, at the end of which if a new candidate root is found such node will send the current root a root switch request (triggering the MESH_EVENT_ROOT_SWITCH_REQ event), which in turn will disconnect from the router, search for a preferred parent to connect to, and send back to the new root candidate a root switch acknowledgment (triggering the MESH_EVENT_ROOT_SWITCH_ACK event), which in turn will disconnect from its current parent and attempt to connect to the router as the new root.
Note that if no candidate root is found at the end of the election or if it coincides with the current root node, no root switch will occur.

Example

```
void rootWaiveDriver()
 {
  EventBits_t eventbits;      //Used to check the flags of the mesh event group
  mesh_vote_t vote_settings;  //Used to store the new root election settings
  wifi_ap_record_t routerAP;  //Used to store information on the router's AP
                                the root's station interface is connected to

  //Initialize the new root election settings
  vote_settings.is_rc_specified = false;  //Define the root switch request type
                                            (currently only false is supported,
                                             i.e. trigger a new root election)
  vote_settings.config.attempts = ROOT_ELECTION_MAX_ROUNDS; //Set the election
                                                              maximum rounds
  vote_settings.percentage = ROOT_ELECTION_THRESHOLD;    //Set the root voting
                                                           percentage threshold
  //New root election trigger cycle
  while(mesh_state.my_type == MESH_ROOT)  //While the node is the root
   {
    ESP_ERROR_CHECK(esp_wifi_sta_get_ap_info(&routerAP));//Retrieve information
                                                          on the router's AP
    eventbits = xEventGroupGetBits(mesh_event_group);  //Retrieve the value of
                                                        the mesh event group
    if((routerAP.rssi <= ROOT_MIN_RSSI)&&   //If the RSSI with the router falls
      ((eventbits>>2)&1)&&((eventbits>>4)&1)) below its minimum threshold, the
                                              node has children connected
                                              (MESH_CHILD) and a new root
                                              election is not already in
                                              progress (MESH_VOTE)
     ESP_ERROR_CHECK(esp_mesh_waive_root(&vote_settings, //Start a new election
                                         MESH_VOTE_REASON_INITIATED);
    vTaskDelay(WAIVE_CHECK_INTERVAL/portTICK_PERIOD_MS); //Await a predefined
   }                                                      interval before
  ...                                                     checking again
 }
```

## Mesh Network Status API

- **Retrieve the number of nodes in the mesh network**

```
//file esp_mesh.h

int esp_mesh_get_total_node_num(void)
```

| Possible Returns | |
|---|---|
| *nodes_num* | The current number of nodes in the mesh network |

Example

```
void appDriver()
 {
  …
  int tot_nodes = esp_mesh_get_total_node_num();
  …
 }
```

## Routing Table API

A node's routing table is comprised of its station MAC address and the station MAC addresses of all its descendants in the mesh network, where as previously discussed in the mesh packets routing algorithm a node also keeps track of the children through which each of its descendants is reachable.

- **Retrieve the number of entries in a node's routing table**

```
//file esp_mesh.h

int esp_mesh_get_routing_table_size(void)
```

| Possible Returns | |
|---|---|
| *rtable_num* | The number of entries in the node's routing table |

Where note that the number of entries in the root node's routing table represents the total number of nodes in the mesh network ( = `esp_mesh_get_total_node_num()`), and the minimum size of a node's routing table is 1, which occurs for a node with no children connected and so with only its own station MAC address in its routing table.

Example

```
void appDriver()
 {
  …
  int rtable_num = esp_mesh_get_routing_table_size();
  …
 }
```

- **Retrieve the entries of a node's routing table**

```
//file esp_mesh.h

esp_err_t esp_mesh_get_routing_table(mesh_addr_t* rtable,
                                     int rtable_size,
                                     int* rtable_num)
```

| Parameters | |
|---|---|
| rtable | The address where to copy the entries of the node's routing table |
| rtable_size | The routing table size in bytes ( = rtable_num * 6) |
| rtable_num | The address where to copy the number of entries in the node's routing table |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_MESH_ARGUMENT | Invalid argument(s) |
| ESP_FAIL | Unknown error in the mesh stack |

Note that the memory where to store the node's routing table should be allocated dynamically, where it's size can be retrieved beforehand by using the `esp_mesh_get_routing_table_size()` function described earlier.

Example

```
void appDriver()
 {
  …
  mesh_addr_t* rtable; //Used to store dynamically the node's routing table
  int rtable_num;      //Used to store the number of entries
                         in the node's routing table
  //Retrieve the number of entries in the node's routing table
  rtable_num = esp_mesh_get_routing_table_size();

  //Dynamically allocate the memory required to store the node's routing table
  rtable = (mesh_addr_t*)malloc(rtable_entries*6);

 //Retrieve the node's routing table
  ESP_ERROR_CHECK(esp_mesh_get_routing_table(rtable,rtable_entries*6,
                                             &rtable_entries));
  …
  free(rtable); //Release the dynamic memory used
  …               to store the node's routing table
 }
```

- **Retrieve the number of entries in the sub-routing table relative to a node's child**

  The number of entries in the sub-routing table relative to a node's child, which is equal to the number of entries in the node's child routing table, can be retrieved by calling the following function:

```
//file esp_mesh.h

esp_err_t esp_mesh_get_subnet_nodes_num(const mesh_addr_t* child,
                                        int* child_rtable_num)
```

| Parameters | |
|---|---|
| child | The MAC address of the node's child for which to retrieve the number of entries in its related sub-routing table |
| child_rtable_num | The address where to copy the number of entries in the sub-routing table relative to the node's child |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_MESH_ARGUMENT | Invalid argument(s) |
| ESP_ERR_MESH_NOT_START | Mesh networking is not enabled (call esp_mesh_start() first) |
| ESP_FAIL | Unknown error in the mesh stack |

Where note that the list of children connected to a node with their MAC addresses can be retrieved by using the `esp_wifi_ap_get_sta_list()` function described earlier in the MESH_EVENT_CHILD_DISCONNECTED event specific handler.

Example

```
void appDriver()
{
  …
  mesh_addr_t child;         //Used to store the MAC address of a specific child
  int child_rtable_num;      //Used to store the number of entries in the
                               sub-routing table relative to the node's child
  wifi_sta_list_t children; //Used to store information on the clients
                               (children) connected to the node

  ESP_ERROR_CHECK(esp_wifi_ap_get_sta_list(&children));  //Retrieve information
                                                           on the connected
                                                           children nodes
  memcpy(child.addr,clients.sta[CHILD_INDEX].mac,6);     //Copy the MAC address
                                                           of a specific child
  //Retrieve the number of entries in the
    sub-routing table relative to the node's child
  ESP_ERROR_CHECK(esp_mesh_get_subnet_nodes_num(&child,&child_rtable_num));
  …
}
```

- **Retrieve the entries of the sub-routing table relative to a node's child**

  The entries of the sub-routing table relative to a node's child, which correspond to the entries in the node's child routing table, can be retrieved by calling the following function:

```
//file esp_mesh.h

esp_err_t esp_mesh_get_subnet_nodes_list(const mesh_addr_t* child,
                                         const mesh_addr_t*
                                                  child_rtable,
                                         int child_rtable_num)
```

| Parameters | |
|---|---|
| child | The MAC address of the node's child for which to retrieve the entries in its related sub-routing table |
| child_rtable | The address where to copy the entries of the sub-routing table relative to the node's child |
| child_rtable_num | The number of entries to retrieve from the sub-routing table relative to the node's child |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_MESH_ARGUMENT | Invalid argument(s) |
| ESP_ERR_MESH_NOT_START | Mesh networking is not enabled (call esp_mesh_start() first) |
| ESP_FAIL | Unknown error in the mesh stack |

Where again the list of children connected to a node with their MAC addresses can be retrieved via the `esp_wifi_ap_get_sta_list()` function, and the memory where to store the node's child sub-routing table should be allocated dynamically, where its size can be retrieved beforehand by using the `esp_mesh_get_subnet_nodes_num()` function described earlier.

Example

```
void appDriver()
 {
  …
  mesh_addr_t child;
  mesh_addr_t* child_rtable;
  int child_rtable_num;
  wifi_sta_list_t children;

  ESP_ERROR_CHECK(esp_wifi_ap_get_sta_list(&children));  //Retrieve information
                                                           on the connected
  memcpy(child.addr,clients.sta[CHILD_INDEX].mac,6);     //Copy the MAC address
                                                           of a specific child
  //Retrieve the number of entries in the
    sub-routing table relative to the node's child
  ESP_ERROR_CHECK(esp_mesh_get_subnet_nodes_num(&child,&child_rtable_num));

  //Dynamically allocate the memory required to store
    the sub-routing table relative to the node's child
  child_rtable = (mesh_addr_t*)malloc(child_rtable_entries*6);

  //Retrieve the entries of the sub-routing table relative to the node's child
  ESP_ERROR_CHECK(esp_mesh_get_subnet_nodes_list(&child,child_rtable,
                                                 child_rtable_entries));
  …
  free(child_rtable); //Release the dynamic memory used to store the
  …                     sub-routing table relative to the node's child
 }
```

## Node Mesh Status API

The following API allows to retrieve information on the mesh status of a node directly from the mesh stack, where note that by using the `mesh_status_t` struct and the mesh events specific handlers proposed in this guide their use in an application is not necessary.

- **Retrieve a node's parent BSSID**

```
//file esp_mesh.h

esp_err_t esp_mesh_get_parent_bssid(mesh_addr_t* bssid)
```

| Parameters | |
|---|---|
| bssid | The address where to store the BSSID of the node's parent |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_FAIL | Unknown error in the mesh stack |

Note that the BSSID of a node's parent changes only when the node connects to a new parent in the mesh network (MESH_EVENT_PARENT_CONNECTED t, `info.connected.bssid`).

Example

```
void appDriver()
 {
  …
  mesh_addr_t parent;            //Used to store the BSSID of the node's parent
  ESP_ERROR_CHECK(esp_mesh_get_parent_bssid(&parent));
  …
 }
```

- **Check whether a node is the root node**

```
//file esp_mesh.h

bool esp_mesh_is_root(void)
```

| Possible Returns | |
|---|---|
| *is_root* | Whether the node is the root node or not |

Note that a node's type changes to root only when it connects to a new parent (the router) and is on the first layer of the mesh network (MESH_EVENT_PARENT_CONNECTED, `info.self_layer == 1`).

Example

```
void appDriver()
 {
  …
  if(esp_mesh_is_root())
   /* The node is the root node */
  else
   /* The node is not the root node */
  …
 }
```

- **Retrieve a node's type**

```
//file esp_mesh.h
```

```
mesh_type_t esp_mesh_get_type(void)
```

| Possible Returns | |
|---|---|
| *node_type* | The current type of the node (MESH_IDLE, MESH_ROOT, MESH_NODE or MESH_LEAF) |

Note that a node's type can change only:

- When the node connects to a new parent (MESH_EVENT_PARENT_CONNECTED) depending on its layer (`info.self_layer`) and the mesh network's maximum layer (MESH_MAX_LAYER).
- When a network topology change affects the node's ancestors (MESH_EVENT_LAYER_CHANGE), depending on its new layer (`info.new_layer`) and the mesh network's maximum layer (MESH_MAX_LAYER).

Example

```
void appDriver()
{
 …
 mesh_type_t my_type = esp_mesh_get_type();
 …
}
```

- **Retrieve a node's layer in the mesh network**

```
//file esp_mesh.h
```

```
int esp_mesh_get_layer(void)
```

| Possible Returns | |
|---|---|
| *node_layer* | The current layer of the node in the mesh network |

Note that the layer of a node in the mesh network can change only:

- When the node connects to a new parent (MESH_EVENT_PARENT_CONNECTED, `info.self_layer`).
- When a network topology change affects the node's ancestors (MESH_EVENT_LAYER_CHANGE, `info.new_layer`).

Example

```
void appDriver()
{
 …
 int my_layer = esp_mesh_get_layer();
 …
}
```

## Mesh Configuration Retrieval API

The following API allow to retrieve the configuration of the mesh stack on a node:

## Base Mesh Configuration

- ### Retrieve a node's base mesh configuration

  As a reminder, a node's base mesh configuration consists of:

  - The **Mesh Network Identifier (MID)** of the mesh network the node will attempt to join or create.
  - The **Mesh Wi-Fi Channel**, representing the Wi-Fi channel used by the node to communicate over the mesh network.
  - The address of the **Mesh Events General Handler** function.
  - The **Mesh SoftAP Settings**, consisting of the password used by the node to connect to a parent and to accept children connections and the maximum number of connected children for the node.
  - The **Mesh Router Settings**, consisting of the router's AP SSID, SSID_length, password and the specific BSSID of the router to connect to.
  - The **cryptographic algorithm** used by the node to crypt the Mesh IE in its Wi-Fi beacon frames.

  From here such information can be retrieved by calling the following function:

  ```
  //file esp_mesh.h
  ```

  ```
  esp_err_t esp_mesh_get_config(mesh_cfg_t config)
  ```

| Parameters | |
|---|---|
| config | The address where to copy the node's base mesh configuration |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_MESH_ARGUMENT | Invalid argument(s) |
| ESP_FAIL | Unknown error in the mesh stack |

Example

```
void appDriver()
 {
  …
  mesh_cfg_t bmesh_conf;
  ESP_ERROR_CHECK(esp_mesh_get_config(&bmesh_conf);
  …
 }
```

## Mesh Organization Mode

- ### Check whether the self-organized networking features are enabled on a node

```
//file esp_mesh.h
```

```
bool esp_mesh_get_self_organized(void)
```

| Possible Returns | |
|---|---|
| *is_self_organized* | Whether the self-organized networking features are enabled on the node or not |

Where note that disabled self-organized networking features correspond to a manual mesh network, while enabled self-organized networking features correspond either to a self-organized or a fixed-root network depending on the node's fixed root setting.

Example

```
void appDriver()
 {
  …
  if(esp_mesh_get_self_organized())
    /* Self-Organized or Fixed-Root networking
       depending on the fixed-root setting     */
  else
    /* Manual Networking */
  …
 }
```

- ### Retrieve a node's Fixed Root setting

```
//file esp_mesh.h
```

```
bool esp_mesh_is_root_fixed(void)
```

| Possible Returns | |
|---|---|
| *is_root_fixed* | Whether the node's fixed root setting is enabled or not |

Where note that a disabled fixed root setting corresponds to a self-organized mesh network, while an enabled fixed root setting corresponds either to a fixed-root or a manual network depending on whether the self-organized networking features on the node are enabled or not.

Example

```
void appDriver()
 {
  …
  if(esp_mesh_is_root_fixed())
   /* Fixed-Root or Manual Networking depending if
      the self-networking features are enabled or not */
  else
   /* Self-Organized Networking */
  …
 }
```

From here, the mesh organization mode used on a node can be retrieved by jointly using the previous two functions as follows:

Example

```
void appDriver()
 {
  …
  if(esp_mesh_get_self_organized())
   if(esp_mesh_is_root_fixed())
    /* Fixed-Root Networking */
   else
    /* Self-Organized Networking */
  else
   /* Manual Networking */
  …
 }
```

## Mesh Topology Settings

- **Retrieve a node's mesh maximum layer setting**

```
//file esp_mesh.h

int esp_mesh_get_max_layer(void)
```

| Possible Returns | |
|---|---|
| *mesh_max_layer* | The node's mesh maximum layer setting |

Example

```
void appDriver()
 {
  …
  int mesh_max_layer = esp_mesh_get_max_layer();
  …
 }
```

- **Retrieve a node's mesh maximum capacity setting**

```
//file esp_mesh.h

int esp_mesh_get_capacity_num(void)
```

| Possible Returns | |
|---|---|
| *mesh_max_capacity* | The node's mesh maximum capacity setting |

Example

```
void appDriver()
 {
  …
  int mesh_max_capacity = esp_mesh_get_capacity_num();
  …
 }
```

## Mesh Self-Organized Root Settings

- **Retrieve a node's root election voting percentage threshold**

```
//file esp_mesh.h
```

```
float esp_mesh_get_vote_percentage(void)
```

| Possible Returns | |
|---|---|
| *voting_threshold* | The node's voting percentage threshold to elect itself as root in a root election |

Example

```
void appDriver()
 {
  …
  float voting_threshold = esp_mesh_get_vote_percentage();
  …
 }
```

- **Retrieve a node's minimum number of rounds in a root election**

```
//file esp_mesh_internal.h
```

```
esp_err_t esp_mesh_get_attempts(mesh_attempts_t* attempts)
```

| Parameters | |
|---|---|
| attempts | The address where to store the struct holding the self-organized implicit actions' number of attempts |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_MESH_ARGUMENT | Invalid argument(s) |
| ESP_FAIL | Unknown error in the mesh stack |

Example

```
void appDriver()
 {
  …
  mesh_attempts_t self_attempts;
  ESP_ERROR_CHECK(esp_mesh_get_attempts(&vote_turns);
  //From here the "scan" member of the mesh_attempts_t struct
    holds the node's minimum number of rounds in a root election
  …
 }
```

- **Retrieve a node's root self-healing delay**

```
//file esp_mesh.h
```

```
int esp_mesh_get_root_healing_delay(void)
```

| Possible Returns | |
|---|---|
| *root_healing_delay* | The node's root self-healing delay (ms) |

Example

```
void appDriver()
 {
  …
  int root_healing_delay = esp_mesh_get_root_healing_delay();
  …
 }
```

## Other Root Node Settings

- **Retrieve the size of the root node's receiving queue for packets destined for the external DS (TODS queue)**

```
//file esp_mesh.h
```

```
int esp_mesh_get_xon_qsize(void)
```

| Possible Returns | |
|---|---|
| *TODS_queue_size* | The size of the root node's receiving queue for packets destined for the external DS |

Example

```
void appDriver()
 {
  …
  int TODS_queue_size = esp_mesh_get_xon_qsize();
  …
 }
```

- **Retrieve the root node's station IP configuration**

```
//File tcpip_adapter.h
```

```
esp_err_t tcpip_adapter_get_ip_info(tcpip_adapter_if_t tcpip_if,
                                    tcpip_adapter_ip_info_t* ip_info)
```

| Parameters | |
|---|---|
| tcpip_if | The interface for which to retrieve the IP configuration<br>❑ TCPIP_ADAPTER_IF_STA → Station interface<br>❑ TCPIP_ADAPTER_IF_AP → SoftAP interface |
| ip_info | The address where to copy the interface's IP configuration |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_TCPIP_ADAPTER_INVALID_PARAMS | Invalid argument(s) |
| ESP_FAIL | Unknown error in the Wi-Fi stack |

Example

```
void appDriver()
 {
  …
  tcpip_adapter_ip_info_t ipinfo;    //Used to store an interface's
                                       IP configuration
  ESP_ERROR_CHECK(tcpip_adapter_get_ip_info(TCPIP_ADAPTER_IF_STA,&ipinfo));
  …
 }
```

- **Retrieve the address of a DNS server set on the root node's Station interface**

```
//File tcpip_adapter.h

esp_err_t tcpip_adapter_get_dns_info(tcpip_adapter_if_t tcpip_if,
                                     tcpip_adapter_dns_type_t type,
                                     tcpip_adapter_dns_info_t* addr)
```

| Parameters | |
|---|---|
| tcpip_if | The interface for which to retrieve a DNS server's address<br>❑ TCPIP_ADAPTER_IF_STA → Station interface<br>❑ TCPIP_ADAPTER_IF_AP → SoftAP interface |
| type | The interface's DNS server type for which to retrieve the address<br>❑ TCPIP_ADAPTER_DNS_MAIN → Primary DNS server<br>❑ TCPIP_ADAPTER_DNS_BACKUP → Secondary DNS server |
| addr | Where to copy the interface's DNS server address |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_TCPIP_ADAPTER_INVALID_PARAMS | Invalid argument(s) |
| ESP_FAIL | Unknown error in the Wi-Fi stack |

Example

```
void appDriver()
 {
  …
  tcpip_adapter_dns_type_t dns_primary;    //Used to store an interface's
  tcpip_adapter_dns_type_t dns_secondary;    DNS Server address
  ESP_ERROR_CHECK(tcpip_adapter_get_dns_info(TCPIP_ADAPTER_IF_STA, //Station
                                             TCPIP_ADAPTER_DNS_MAIN, Primary
                                             &dns_primary));         DNS Server
  ESP_ERROR_CHECK(tcpip_adapter_get_dns_info(TCPIP_ADAPTER_IF_STA,  //Station
                                             TCPIP_ADAPTER_DNS_BACKUP,Secondary
                                             &dns_secondary));       DNS Server

  …
 }
```

## Mesh Additional SoftAP Settings

- ### Retrieve a node's Mesh SoftAP Authmode

```
//file esp_mesh.h
```

```
wifi_auth_mode_t esp_mesh_get_ap_authmode(void)
```

| Possible Returns | |
|---|---|
| *mesh_softap_authmode* | The node's mesh SoftAP authmode |

Where remember that a node's mesh SoftAP authmode differs in general from the authentication mode used on the node's SoftAP interface (which currently is always left *open*).

Example

```
void appDriver()
 {
  …
  wifi_authmode_t mesh_softap_authmode = esp_mesh_get_ap_authmode();
  …
 }
```

- ### Retrieve a node's mesh children disassociation delay

```
//file esp_mesh.h
```

```
int esp_mesh_get_ap_assoc_expire(void)
```

| Possible Returns | |
|---|---|
| *children_disassoc_delay* | The node's mesh children disassociation delay (s) |

Example

```
void appDriver()
 {
  …
  int children_disassoc_delay = esp_mesh_get_ap_assoc_expire();
  …
 }
```

## Mesh Node-specific Settings

- **Check whether a node belongs to a specific mesh group**

```
//file esp_mesh.h

bool esp_mesh_is_my_group(const mesh_addr_t* gid)
```

| Parameters | |
|---|---|
| gid | The GID of the group to check for membership of the node |

| Possible Returns | |
|---|---|
| is_my_group | Whether the node belongs to such mesh group or not |

Example

```
void appDriver()
 {
  …
  mesh_addr_t mesh_group;

  memcpy(mesh_group.addr,GROUP_GID,6);  //Set the GID of the group to check
  if(esp_mesh_is_my_group(&group_id))     for membership of the node
   /* The node belongs to such mesh group */
  else
   /* The node doesn't belong to such mesh group */
  …
 }
```

- **Retrieve the mesh groups a node belongs to**

```
//file esp_mesh.h

esp_err_t esp_mesh_get_group_list(mesh_addr_t* gids, int num)
```

| Parameters | |
|---|---|
| gids | The address where to copy the GIDs of the mesh groups the node belongs to |
| num | The maximum number of GIDs to retrieve |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_MESH_ARGUMENT | Invalid argument(s) |
| ESP_FAIL | Unknown error in the mesh stack |

Example

```
void appDriver()
 {
  …
  mesh_addr_t node_groups[MESH_MAX_GROUPS]; //MESH_MAX_GROUPS represents
                                              the maximum number of mesh
                                              groups a node can belong to
  ESP_ERROR_CHECK(esp_mesh_get_group_list(&node_groups,MESH_MAX_GROUPS));
  …
 }
```

- **Remove a node from a set of mesh groups**

```
//file esp_mesh.h

esp_err_t esp_mesh_delete_group_id(const mesh_addr_t* gids,
                                   int num)
```

| Parameters | |
|---|---|
| gids | The address where to retrieve the GIDs of the mesh groups to remove the node from |
| num | The number of mesh groups to remove the node from |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_MESH_ARGUMENT | Invalid argument(s) |
| ESP_FAIL | Unknown error in the mesh stack |

Example

```
void appDriver()
 {
  …
  mesh_addr_t mesh_groups[NUM_GROUPS]; //NUM_GROUPS represents the number of
                                       mesh groups to remove the node from
  for(int i=0;i<NUM_GROUPS;i++)    //Set the mesh groups to remove the node from
   memcpy(mesh_groups.addr,GROUPi_GID,6);
  ESP_ERROR_CHECK(esp_mesh_delete_group_id(&mesh_groups,NUM_GROUPS));
  …
 }
```

- **Retrieve the MAC address of a node's Wi-Fi Interface mode (Station or SoftAP)**

```
//File esp_wifi.h

esp_err_t esp_wifi_get_mac(wifi_interface_t ifx, uint8_t* mac))
```

| Parameters | |
|---|---|
| ifx | The interface for which to retrieve the MAC address<br>❑ ESP_IF_WIFI_STA → Station Interface<br>❑ ESP_IF_WIFI_AP → SoftAP Interface |
| mac | The address where to copy the interface's MAC address |

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_ERR_WIFI_NOT_INIT | The Wi-Fi stack is not initialized (call esp_wifi_init() first) |
| ESP_ERR_WIFI_IF | Invalid Wi-Fi interface |
| ESP_ERR_INVALID_ARG | Invalid argument(s) |
| ESP_FAIL | Unknown error in the Wi-Fi stack |

Example

```
void appDriver()
 {
  …
  uint8_t ifx_mac[6];   //Array where to store the node's interface MAC address
  ESP_ERROR_CHECK(esp_wifi_get_mac(ESP_IF_WIFI_STA,&mac); //Station MAC address
  …
 }
```

## Mesh Stop API

- ### Disable mesh networking on a node

```
//file esp_mesh.h
```

```
esp_err_t esp_mesh_stop(void)
```

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_FAIL | Unknown error in the mesh stack |

Calling this function causes the node to:

- Deinitialize the mesh IE.
- Disconnect from its parent, if any (without raising the MESH_EVENT_PARENT_DISCONNECTED event)
- Disassociate all its children nodes, if any (without raising MESH_EVENT_CHILD_DISCONNECTED events)
- Delete all the packets sending and receiving queues.
- Stop the mesh network management service (raising the MESH_EVENT_STOPPED event)
- Unregister the Mesh Events General Handler function.
- Release the data structures used for mesh networking in the mesh stack.
- Restore the Wi-Fi SoftAP interface to its default settings.

Example

```
void appDriver()
 {
  …
  ESP_ERROR_CHECK(esp_mesh_stop());
  …
 }
```

- ### Deinitialize the mesh stack on a node

```
//file esp_mesh.h
```

```
esp_err_t esp_mesh_deinit(void)
```

| Possible Returns | |
|---|---|
| ESP_OK | Success |
| ESP_FAIL | Unknown error in the mesh stack |

Calling this function causes the disabling on mesh networking as with the `esp_mesh_stop()` function plus the release of all remaining resources in the node's mesh stack.

Example

```
void appDriver()
 {
  …
  ESP_ERROR_CHECK(esp_mesh_deinit());
  …
 }
```