ESP-IDF Wi-Fi Stack Practical Guide

ESP-IDF V3.1

ESP-IDF Wi-Fi Stack Practical Guide

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

License

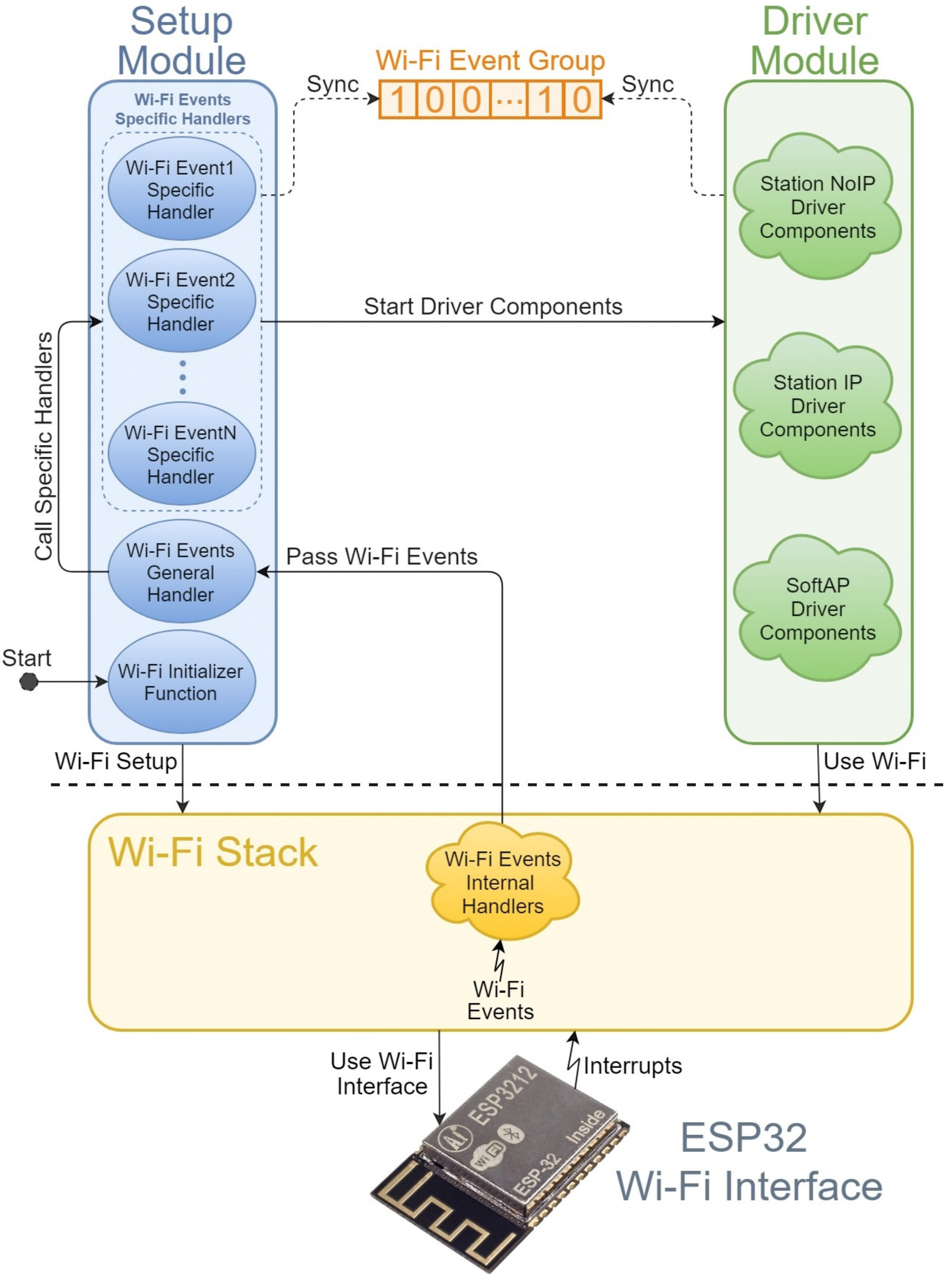
Requirements

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

- Network stack principles.
- Wi-Fi basics (concepts of SSID, BSSID, Wi-Fi channels, 802.11x protocols, authentication modes, etc.)
- Difference between the Station and the SoftAP modes (or sub-interfaces) of a Wi-Fi interface.
- 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 Wi-Fi Application

An application which uses the Wi-Fi networking functionalities offered by ESP32 boards can generally be divided into the following components:


Setup Module
Wi-Fi Events Specific Handlers
Wi-Fi Event1 Specific Handler
Wi-Fi Event2 Specific Handler
Wi-Fi EventN Specific Handler
Wi-Fi Events General Handler
Wi-Fi Initializer Function
Start
Call Specific Handlers
Sync
Wi-Fi Event Group
1 0 0 ··· 1 0
Sync
Driver Module
Station NoIP Driver Components
Start Driver Components
Station IP Driver Components
SoftAP Driver Components
Pass Wi-Fi Events
Wi-Fi Setup
Use Wi-Fi
Wi-Fi Stack
Wi-Fi Events Internal Handlers
Wi-Fi Events
Use Wi-Fi Interface
Interrupts
ESP32 Wi-Fi Interface

The collection of functions and data structures provided by the ESP-IDF to offer Wi-Fi networking functionalities represents the **Wi-Fi Stack**, which is implemented on top and directly manages the Wi-Fi interface, while at the application level a Wi-Fi application can be divided into the following two modules:

Setup Module

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

- A **Wi-Fi Initializer Function**, which represents the entry point of a Wi-Fi application and whose tasks are to initialize and configure the Wi-Fi stack and enable the device's Wi-Fi interface.

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

- A set of **Wi-Fi Events Specific Handlers**, which are functions whose tasks are to perform the application-level handling of Wi-Fi events raised by the Wi-Fi 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 Wi-Fi 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 Wi-Fi networking features they use, and so the requirements that must be met before their execution can begin:

- The **Station NoIP Driver Components** are components that require the device's station interface to be connected to an access point (AP).

- The **Station IP Driver Components** are components that require the device's station interface to be connected to an access point (AP) and to have an IP configuration set.

- The **SoftAP Driver Components** are components that require the device's SoftAP interface to be enabled, and may rely on it having clients connected.

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 Wi-Fi 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 "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
```

Wi-Fi Event Group

The Wi-Fi event group used for inter-task synchronization purposes between the Setup and the Driver module and its flags used in the context of this guide are defined as follows:

```
EventGroupHandle_t wifi_event_group;      //Wi-Fi Event Group Handler

/*-- Station interface Flags --*/
#define STA_ON      BIT0   //Whether the Station interface is enabled or not
#define STA_CONN    BIT1   //Whether the Station interface is connected to an AP or not
#define STA_GOTIP   BIT2   //Whether the Station interface has an IP configuration set
or not

/*-- SoftAP interface Flags --*/
#define SOFTAP_ON   BIT3   //Whether the SoftAP interface is enabled or not
#define SOFTAP_CLI  BIT4   //Whether the SoftAP interface has clients connected or not

/*-- 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 WIFI_DRIVER_STATION_NOIP BIT5  //Whether the Station NoIP drivers are running
#define WIFI_DRIVER_STATION_IP   BIT6  //Whether the Station IP drivers are running
#define WIFI_DRIVER_SOFTAP       BIT7  //Whether the SoftAP drivers are running
```

Also note that within this guide the parameters used in the code examples are shown in capital letters and represent predefined constants (e.g. `WIFI_STATION_SSID_PASSWORD`, `WIFI_SOFTAP_AUTHMODE`, 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 Wi-Fi application is divided into a first phase relative to the inizializations and configurations required to enable Wi-Fi networking on a device, which are carried out by the **Wi-Fi Initializer Function**, and a second event-driven phase represented by the application-level handling of the Wi-Fi events raised by the Wi-Fi stack, which are carried out by the **Wi-Fi Events General Handler** and the set of **Wi-Fi Events Specific Handlers**.

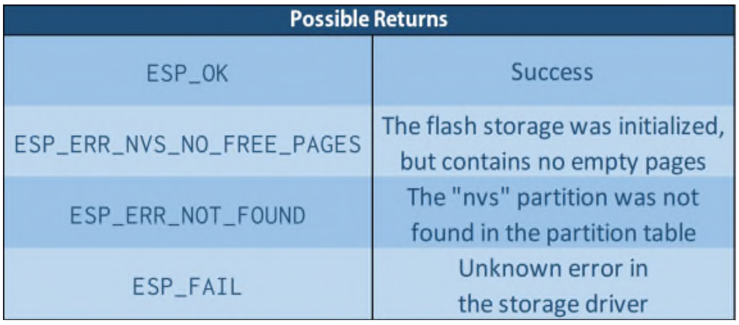
Wi-Fi Initializer Function

The Wi-Fi Initializer Function represents the entry point of a Wi-Fi application, and its purposes are to initialize and configure the Wi-Fi stack and enable the Wi-Fi interface, which is performed through the following tasks:

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:

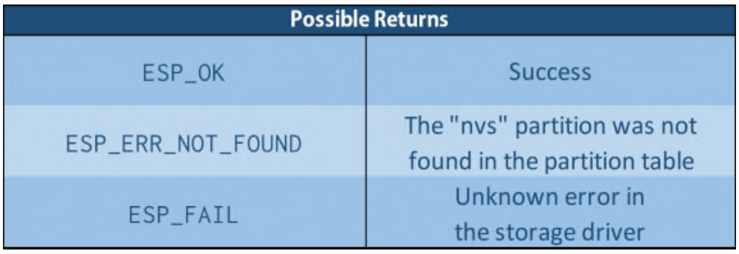
```
//File nvs.flash.h

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 follows:

Example

```
esp_err_t wifi_init()
 {
  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)
```

Example

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

3) Create the Wi-Fi Event Group

Next we need to create the Wi-Fi 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 state of the Wi-Fi interface during the application's execution the following flags are used:

Station interface Flags

A flag representing whether the Station interface is enabled or not	(STA_ON)
A flag representing whether the Station interface is connected to an AP or not	(STA_CONN)
A flag representing whether the Station interface has an IP configuration set or not	(STA_GOTIP)

SoftAP interface Flags

A flag representing whether the SoftAP interface is enabled or not	(SOFTAP_ON)
A flag representing whether the SoftAP interface has clients connected or not	(SOFTAP_CLI)

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 Station NoIP driver component is currently being executed or not (WIFI_DRIVER_STATION_NOIP)
- A flag representing whether the Station IP driver component is currently being executed or not (WIFI_DRIVER_STATION_IP)
- A flag representing whether the SoftAP driver component is currently being executed or not (WIFI_DRIVER_SOFTAP)

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

Example

```
esp_err_t wifi_init()
 {
  …
  wifi_event_group = xEventGroupCreate();     //Create the Wi-Fi Event Group
  …
 }
```

4) Register the Wi-Fi Events General Handler

Next we need to register in the Wi-Fi stack the function that must be called each time a Wi-Fi event occurs to perform the application-level handling of such event, i.e. the Wi-Fi Events General Handler function, and this is obtained by calling the following function:

```
//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
ctx	Reserved for the user (typically NULL)

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

Example

```
esp_err_t wifi_events_handler(void* ctx, system_event_t* event)
 {
  /* We'll see later */
 }

…

esp_err_t wifi_init()
 {
  …
  //Register the Wi-Fi Events General Handler
  ESP_ERROR_CHECK(esp_event_loop_init(wifi_events_handler,NULL));
  …
 }
```

5) 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

Generally the Wi-Fi Stack can be initialized to its default parameters, which is obtained by using the following macro:

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

6) Interface Modes Selection

Next we must select the modes (or sub-interfaces) in which the Wi-Fi interface will be enabled later, which depend on the services and consequently the networking features required by the driver components, selection that can be performed by calling the following function:

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

typedef enum            //Interface Mode enumerates
 {
  WIFI_MODE_NULL = 0,   //NULL mode (do not use)
  WIFI_MODE_STA,        //Station mode
  WIFI_MODE_AP,         //SoftAP mode
  WIFI_MODE_APSTA,      //Station + SoftAP mode
  WIFI_MODE_MAX,
 } wifi_mode_t;

//File esp_wifi.h

          esp_err_t esp_wifi_set_mode(wifi_mode_t mode)
```

Parameters	
mode	The mode(s) in which the Wi-Fi interface will be enabled later

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

Example

```
esp_err_t wifi_init()
 {
  …
  if(WIFI_STATION_ENABLE&&WIFI_SOFTAP_ENABLE)          //If both interface
   ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_APSTA));  modes are selected
  else
   if(WIFI_STATION_ENABLE)   //If only the station mode is selected
    ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA));
   else
    if(WIFI_SOFTAP_ENABLE    //If only the SoftAP mode is selected
     ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_AP));
    else                     //If no mode is selected for the Wi-Fi interface
     abort();                  the program's execution cannot continue
  …
 }
```

7) Interface Modes Base Configurations

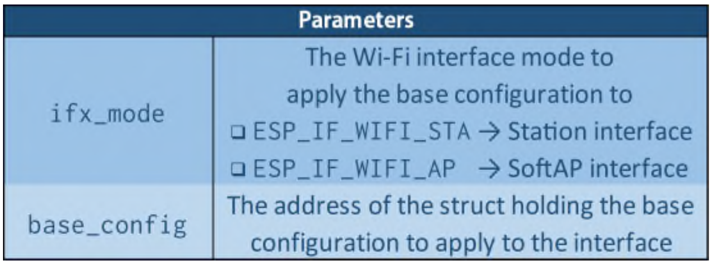
Next we must set the base configuration of the Wi-Fi interface modes that were selected for use previously, which are described by the following data structures:

```
//File esp_wifi_types.h

/*-- Station interface Base Configuration --*/

typedef struct  //A set of information on the target AP the Station interface must
 {                attempt to connect to once the Wi-Fi interface has been enabled
  uint8_t ssid[32];         //SSID of the target AP
  uint8_t password[64];     //Password of the target AP
  bool bssid_set;           //If set the Station interface must attempt to connect
                              only to the AP with the following specific BSSID
  uint8_t bssid[6];         //Specific BSSID of the target AP
  uint16_t listen_interval; //The number of DTIM periods the Station interface
                              remains in the sleep state before checking whether
                              it has frames pending to be received from its AP
  …                           (effective only if the maximum power saving mode
 } wifi_sta_config_t;         is set for the station interface, we'll see later)

/*-- SoftAP interface Base Configuration --*/

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;

typedef struct                //SoftAP interface settings
 {
  uint8_t ssid[32];           //SoftAP SSID
  uint8_t ssid_len;           //SoftAP SSID Length
  uint8_t ssid_hidden;        //Whether the SoftAP SSID should be hidden from its
                                Wi-Fi beacon frames (default = 0, SSID visible)
  wifi_auth_mode_t authmode;  //The authentication protocol used by the
                                SoftAP interface to associate clients
  uint8_t password[64];       //The password required from clients
                                to connect to the SoftAP interface
  uint8_t channel;            //The Wi-Fi channel used by the SoftAP interface
  uint8_t max_connection;     //The maximum number of clients allowed to be
                                connected simultaneously to the SoftAP interface
                                (default and maximum = 4)
  uint16_t beacon_interval;   //The sending interval in millisecond of the SoftAP
                                Wi-Fi beacon frames (default = 100ms, max = 6000ms)
 } wifi_ap_config_t;

//Wi-Fi interface mode base configuration union

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

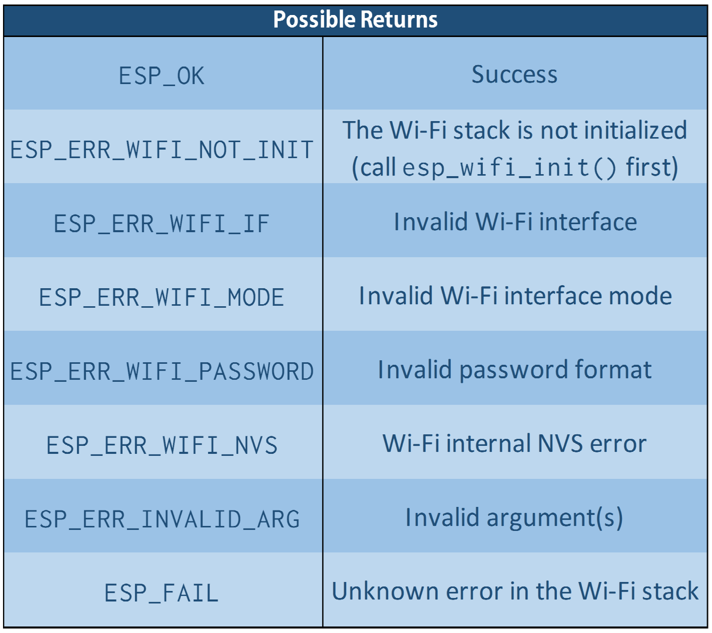
Once the base configuration(s) of the interface mode(s) have been set, they can be applied by using the following function:

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

typedef enum                  //Networking interface enumerates
 {
  ESP_IF_WIFI_STA = 0,        //Wi-Fi Station interface
  ESP_IF_WIFI_AP,             //Wi-Fi SoftAP interface
  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_config(wifi_interface_t ifx_mode,
                                    wifi_config_t* base_config)
```



Parameters	
ifx_mode	The Wi-Fi interface mode to apply the base configuration to ❑ ESP_IF_WIFI_STA → Station interface ❑ ESP_IF_WIFI_AP → SoftAP interface
base_config	The address of the struct holding the base configuration to apply to 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	Invalid Wi-Fi interface
ESP_ERR_WIFI_MODE	Invalid Wi-Fi interface mode
ESP_ERR_WIFI_PASSWORD	Invalid password format
ESP_ERR_WIFI_NVS	Wi-Fi internal NVS error
ESP_ERR_INVALID_ARG	Invalid argument(s)
ESP_FAIL	Unknown error in the Wi-Fi stack

Settings in an interface mode's base configuration that are left uninitialized will be set to their default values with the function call, and note that being the ESP32 Wi-Fi interface limited to a single active Wi-Fi channel, if both interface modes are enabled their Wi-Fi channel will coincide, where the Wi-Fi stack will always switch the SoftAP Wi-Fi channel onto the one currently used by the station interface.

Also note that the Station interface base configuration can also be set after the Wi-Fi interface has been enabled by performing a **Wi-Fi scan** of the available APs (we'll see later).

Example

```
esp_err_t wifi_init()
 {
  …
  //Station interface Base Configuration
  if(WIFI_STATION_ENABLE)
   {
    wifi_config_t station_config = {0};     //Stores the Station interface
                                              base configuration to apply
    //Set the Target AP's SSID
    strcpy((char*)station_config.sta.ssid,WIFI_STATION_AP_SSID);

    //Set the Target AP's Password
    strcpy((char*)station_config.sta.password,WIFI_STATION_AP_PASSWORD);

    //Whether to connect to a target AP with a specific BSSID
    if(WIFI_STATION_USE_SPECIFIC_BSSID)
     {
      station_config.sta.bssid_set = true; //Set the specific BSSID
      memcpy(station_config.sta.bssid,WIFI_STATION_SPECIFIC_BSSID,6);
     }

    //Apply the Station interface Base Configuration
    ESP_ERROR_CHECK(esp_wifi_set_config(ESP_IF_WIFI_STA,&station_config));
   }

  //SoftAP interface Base Configuration
  if(WIFI_SOFTAP_ENABLE)
   {
    wifi_config_t softap_config = {0};      //Stores the SoftAP interface
                                              base configuration to apply
    //Set the SoftAP SSID
    strcpy((char*)softap_config.ap.ssid,WIFI_SOFTAP_SSID);

    //Set the SoftAP SSID Length
    softap_config.ap.ssid_len = strlen(WIFI_SOFTAP_SSID);

    //Whether to hide the SoftAP SSID from its Wi-Fi beacon frames
    if(WIFI_SOFTAP_SSID_HIDDEN)
     softap_config.ap.ssid_hidden = 1;

    //Set the SoftAP Authmode
    softap_config.ap.authmode = WIFI_SOFTAP_AUTHMODE;

    //Set the SoftAP Password
    strcpy((char*)softap_config.ap.password,WIFI_SOFTAP_PASSWORD);

    //Set the Wi-Fi channel to be used by the SoftAP interface
    softap_config.ap.channel = WIFI_SOFTAP_CHANNEL;

    //Set the SoftAP Maximum Connections
    softap_config.ap.max_connection = WIFI_SOFTAP_MAXCONNECTIONS;

    //Set the SoftAP Wi-Fi Beacon Sending Interval
    softap_config.ap.beacon_interval = WIFI_SOFTAP_BEACON_INTERVAL;

    //Apply the SoftAP interface Base Configuration
    ESP_ERROR_CHECK(esp_wifi_set_config(ESP_IF_WIFI_AP,&softap_config));
   }
  …
 }
```

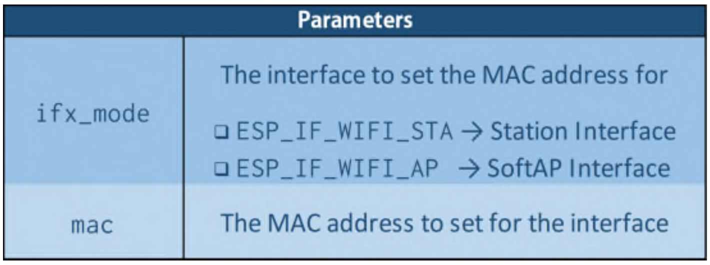
8) Other Interface Modes Settings (*optional*)

Once their base configurations have been set it is possible to configure additional settings related to the Wi-Fi interface modes, such as:

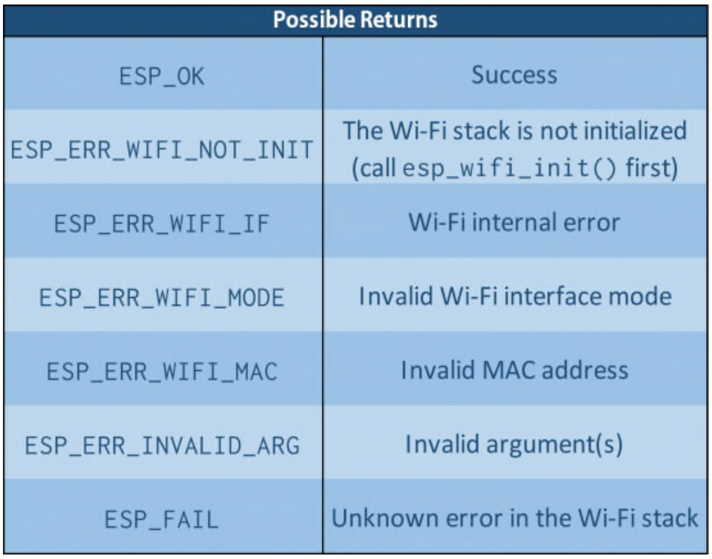
- **Set a custom MAC address for an interface**

By default the Station interface uses as its MAC address the device Base MAC address burnt into the NIC's ROM, while the SoftAP interface uses the same address incremented by one in the least significant byte, although, if desired, it's possible to use custom MAC addresses for both interfaces by using the following function:

```
//File esp_wifi.h

     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 ❑ ESP_IF_WIFI_STA → Station Interface ❑ 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).

Example

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

  //If a custom MAC address is used for the Station interface
  if(WIFI_STATION_USE_CUSTOM_MAC)
   {
    memcpy(mac,WIFI_STATION_CUSTOM_MAC,6);
    ESP_ERROR_CHECK(esp_wifi_set_mac(ESP_IF_WIFI_STA,mac));
   }

  //If a custom MAC address is used for the SoftAP interface
  if(WIFI_SOFTAP_USE_CUSTOM_MAC)
   {
    memcpy(mac,WIFI_SOFTAP_CUSTOM_MAC,6);
    ESP_ERROR_CHECK(esp_wifi_set_mac(ESP_IF_WIFI_AP,mac));
   }
  …
 }
```

- **Set the Station interface Power Saving Mode**

  To limit its power consumption the following power saving modes can be enabled for the station interface:

  - The minimum power saving mode, where the interface awakens every DTIM period to verify whether it has frames pending to be received by checking its associated AP's TIM map (default).
  - The maximum power saving mode, where the interface awakens every *listen_interval* DTIM periods to verify whether it has frames pending to be received by checking its associated AP's TIM map, where the *listen_interval* parameter was previously set in the station base configuration.

  The power saving mode to use on the Station interface can be selected via the following function:

```
//File esp_wifi_types.h

typedef enum            //Station interface power saving mode enumerates
 {
  WIFI_PS_NONE,         //No power saving
  WIFI_PS_MIN_MODEM,    //Minimum power saving mode (default)
  WIFI_PS_MAX_MODEM,    //Maximum power saving mode
 } wifi_ps_type_t;

//File esp_wifi.h
```

```
esp_err_t esp_wifi_set_ps(wifi_ps_type_t ps_mode)
```

Parameters	
ps_mode	The power saving mode to use for the Station interface

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

Example

```
esp_err_t wifi_init()
 {
  …
  if(WIFI_STATION_ENABLE)     //Set the Station interface power saving mode
   ESP_ERROR_CHECK(esp_wifi_set_ps(WIFI_STATION_POWERSAVING_MODE));
  …
 }
```

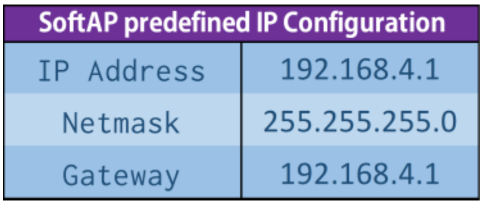
Please refer to the Wi-Fi API Reference for the full list of settings related to the Wi-Fi interface modes, where note that non-configured settings will be set to their default values once the Wi-Fi interface is enabled.

9) Interface Modes IP Settings (*optional*)

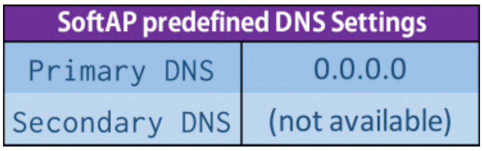
The default IP configurations used by the Station and the SoftAP interfaces consist of the following:

- The Station interface is configured to retrieve a dynamic IP configuration via its DHCP client as soon as it connects to an AP.

- The SoftAP interface is configured with the following predefined IP configuration:



SoftAP predefined IP Configuration	
IP Address	192.168.4.1
Netmask	255.255.255.0
Gateway	192.168.4.1



SoftAP predefined DNS Settings	
Primary DNS	0.0.0.0
Secondary DNS	(not available)

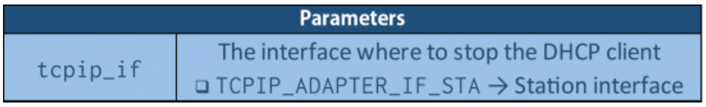
Which represents a Class C IP address with no forwarding or name-resolution capabilities (also note that as of the current ESP-IDF version the SoftAP interface doesn't support a secondary DNS server set). Also note that, once the SoftAP interface is enabled, a DHCP server is started on it to offer its clients dynamic IP configurations, which are obtained from a pool derived from the interface's own IP configuration.

From here, should their default IP configurations be unsuitable for the purposes of the application, it's possible to set custom static IP configurations for the interfaces by performing the following steps:

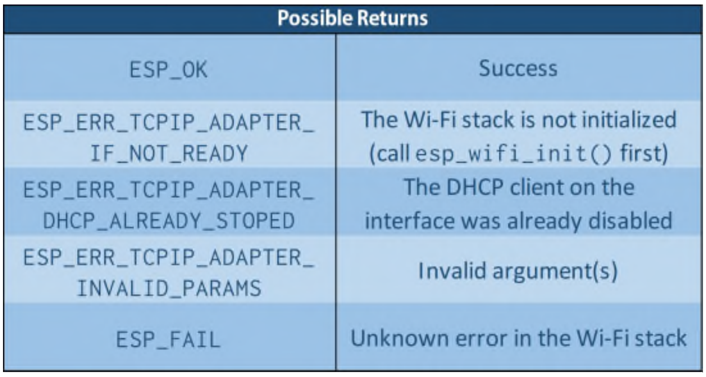
Station interface custom Static IP Configuration

To set a static IP configuration for the Station interface, its DHCP client must be preliminarly disabled, which is obtained by calling the following function:

```
//File tcpip_adapter.h

typedef enum                    //Network interfaces 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)
```



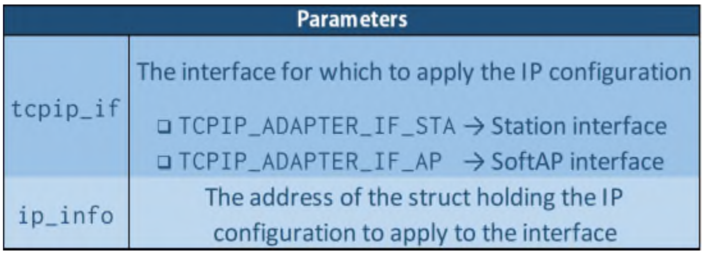
Parameters	
tcpip_if	The interface where to stop the DHCP client ❑ TCPIP_ADAPTER_IF_STA → Station interface



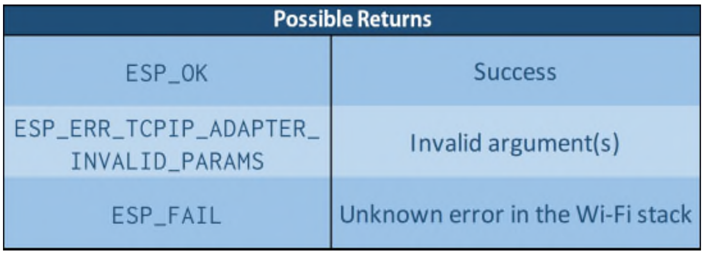
Possible Returns	
ESP_OK	Success
ESP_ERR_TCPIP_ADAPTER_IF_NOT_READY	The Wi-Fi stack is not initialized (call esp_wifi_init() first)
ESP_ERR_TCPIP_ADAPTER_DHCP_ALREADY_STOPED	The DHCP client on the interface was already disabled
ESP_ERR_TCPIP_ADAPTER_INVALID_PARAMS	Invalid argument(s)
ESP_FAIL	Unknown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

Once the Station DHCP client has been disabled a static IP configuration can be set for the interface by using the following function:

```
//File tcpip_adapter.h

typedef struct                 //Interface IP configuration
 {
  ip4_addr_t ip;               //Interface IP address
  ip4_addr_t netmask;          //Interface Netmask
  ip4_addr_t gw;               //Interface default Gateway
 } tcp_ip_adapter_ip_info_t;

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



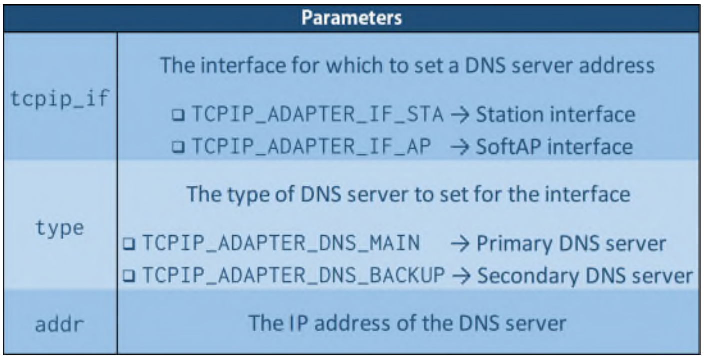
Parameters	
tcpip_if	The interface for which to apply the IP configuration ❑ TCPIP_ADAPTER_IF_STA → Station interface ❑ TCPIP_ADAPTER_IF_AP → SoftAP interface
ip_info	The address of the struct holding the IP configuration to apply to the interface



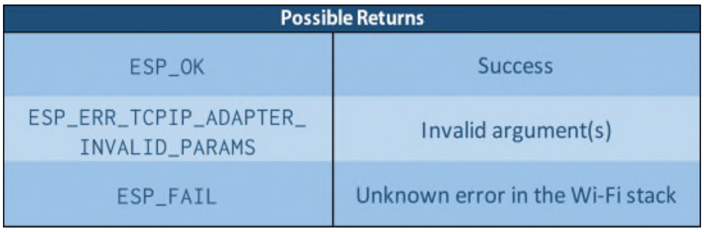
Possible Returns	
ESP_OK	Success
ESP_ERR_TCPIP_ADAPTER_INVALID_PARAMS	Invalid argument(s)
ESP_FAIL	Unknown error in the Wi-Fi stack

If desidered it's also possible to set the addresses of the Station interface primary and secondary DNS servers, which is obtained by using the following function:

```
//File tcpip_adapter.h

typedef enum                      //DNS Servers types enumerates
 {
  TCPIP_ADAPTER_DNS_MAIN = 0,     //Primary (or main) DNS server
  TCPIP_ADAPTER_DNS_BACKUP,       //Secondary (or backup) DNS server
  TCPIP_ADAPTER_DNS_FALLBACK,     //Fallback DNS server (Station interface only)
  TCPIP_ADAPTER_DNS_MAX,
 } tcpip_adapter_dns_type_t;

typedef struct                    //DNS server information
 {
  ip_addr_t ip;                   //DNS server IP address
 } tcp_ip_adapter_dns_info_t;

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 ❑ TCPIP_ADAPTER_IF_STA → Station interface ❑ TCPIP_ADAPTER_IF_AP → SoftAP interface
type	The type of DNS server to set for the interface ❑ TCPIP_ADAPTER_DNS_MAIN → Primary DNS server ❑ 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

SoftAP interface custom Static IP Configuration

Before setting a (custom) static IP configuration for the SoftAP interface, its DHCP server must be temporarily disabled, which is obtained by calling the following function:

```
//File tcpip_adapter.h

  esp_err_t tcpip_adapter_dhcps_stop(tcpip_adapter_if_t tcpip_if)
```

Parameters	
tcpip_if	The interface where to stop the DHCP server ❑ 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

Once its DHCP server has been disabled a custom static IP configuration and possibly the address of its primary DNS server can be set for the SoftAP interface by using the `tcpip_adapter_set_ip_info()` and the `tcpip_adapter_set_dns_info()` functions described earlier, after which its possible to re-enable its DHCP server (whose pool of dynamic IP configurations will be derived from the new interface IP configuration) by using the following function:

```
//File tcpip_adapter.h

  esp_err_t tcpip_adapter_dhcps_start(tcpip_adapter_if_t tcpip_if)
```

Parameters	
tcpip_if	The interface where to start the DHCP server ❑ 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_STARTED	The DHCP server on the interface was already enabled
ESP_ERR_TCPIP_ADAPTER_INVALID_PARAMS	Invalid argument(s)
ESP_FAIL	Unknown error in the Wi-Fi stack

Example

```
//File lwip/sockets.h

int inet_pton(int af, const char* src, void* dst); //Converts an IP address
                                                     from its presentational
                                                     to its network form, i.e.
                                                     from string to uint32_t
```

```
esp_err_t wifi_init()
 {
  …
  tcpip_adapter_ip_info_t ipinfo;     //Used to set a custom static IP
                                        configuration for an interface
  tcpip_adapter_dns_type_t dnsaddr;   //Used to set an interface's
                                        DNS servers addresses

  //If a static IP configuration is used for the Station interface
  if(WIFI_STATION_IPSTATIC)
   {
    //Stop the Station DHCP Client
    ESP_ERROR_CHECK(tcpip_adapter_dhcpc_stop(TCPIP_ADAPTER_IF_STA));

    //Set the Station static IP configuration
    inet_pton(AF_INET,WIFI_STATION_STATIC_IP,&ipinfo.ip);
    inet_pton(AF_INET,WIFI_STATION_STATIC_NETMASK,&ipinfo.netmask);
    inet_pton(AF_INET,WIFI_STATION_STATIC_GATEWAY,&ipinfo.gw);
    ESP_ERROR_CHECK(tcpip_adapter_set_ip_info(TCPIP_ADAPTER_IF_STA,&ipinfo));

    //Set the Station DNS servers
    inet_pton(AF_INET,WIFI_STATION_STATIC_DNS_PRIMARY,&dnsaddr);
    ESP_ERROR_CHECK(tcpip_adapter_set_dns_info(TCPIP_ADAPTER_IF_STA,
                                               TCPIP_ADAPTER_DNS_MAIN,
                                               &dnsaddr));
    inet_pton(AF_INET,WIFI_STATION_STATIC_DNS_SECONDARY,&dnsaddr);
    ESP_ERROR_CHECK(tcpip_adapter_set_dns_info(TCPIP_ADAPTER_IF_STA,
                                               TCPIP_ADAPTER_DNS_BACKUP,
                                               &dnsaddr));
   }

  //If a (custom) static IP configuration is used for the SoftAP interface
  if(WIFI_SOFTAP_IPSTATIC)
   {
    //Temporarily disable the SoftAP DHCP server
    ESP_ERROR_CHECK(tcpip_adapter_dhcps_stop(TCPIP_ADAPTER_IF_AP));

    //Set the SoftAP static IP configuration
    inet_pton(AF_INET,WIFI_SOFTAP_STATIC_IP,&ipinfo.ip);
    inet_pton(AF_INET,WIFI_SOFTAP_STATIC_NETMASK,&ipinfo.netmask);
    inet_pton(AF_INET,WIFI_SOFTAP_STATIC_GATEWAY,&ipinfo.gw);
    ESP_ERROR_CHECK(tcpip_adapter_set_ip_info(TCPIP_ADAPTER_IF_AP,&ipinfo));

    //Set the SoftAP primary DNS server
    inet_pton(AF_INET,WIFI_SOFTAP_STATIC_DNS_PRIMARY,&dnsaddr);
    ESP_ERROR_CHECK(tcpip_adapter_set_dns_info(TCPIP_ADAPTER_IF_AP,
                                               TCPIP_ADAPTER_DNS_MAIN,
                                               &dnsaddr));
    //Re-enable the SoftAP DHCP server
    ESP_ERROR_CHECK(tcpip_adapter_dhcps_start(TCPIP_ADAPTER_IF_AP));
   }
  …
 }
```

Aside from their IP configurations it should also be pointed out that currently the ESP-IDF Wi-Fi stack doesn't offer any built-in functionality to tunnel IP packets between interface modes, thus limiting the possibility of using a board as a standalone access point.

10) Enable the Wi-Fi Interface

Once the desidered interface modes configurations have been set, the Wi-Fi interface can be enabled in the modes previously selected via the `esp_wifi_set_mode()` function by calling the following function:

```
//File esp_wifi.h

                    esp_err_t esp_wifi_start(void)
```

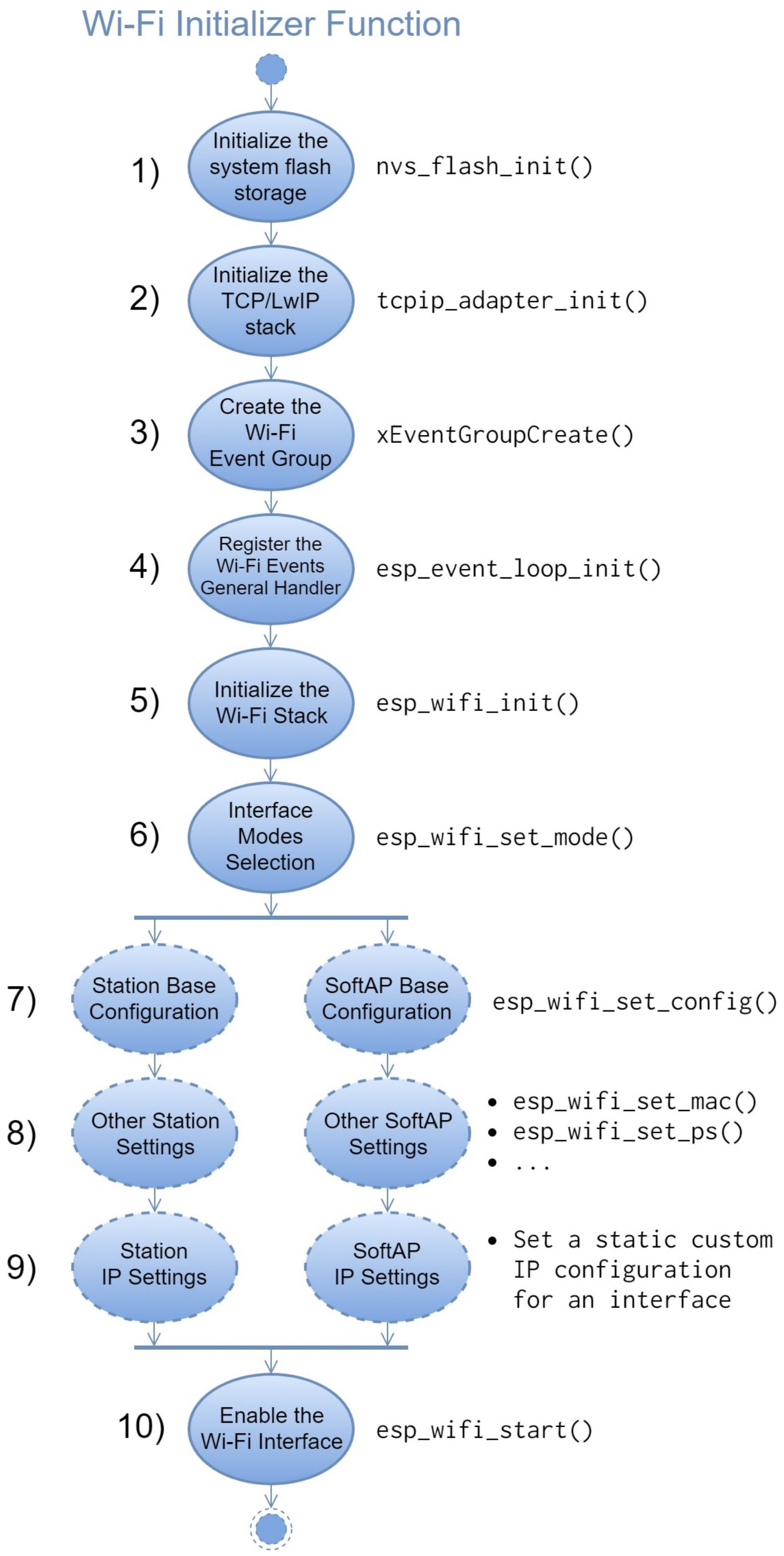
Possible Returns	
ESP_OK	Success
ESP_ERR_WIFI_NOT_INIT	The Wi-Fi stack is not initialized (call `esp_wifi_init()`first)
ESP_ERR_WIFI_CONN	Wi-Fi internal error, station or SoftAP control block wrong
ESP_ERR_NO_MEM	Out of memory
ESP_ERR_INVALID_ARG	Invalid argument(s)
ESP_FAIL	Unknown error in the Wi-Fi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

Example

```
esp_err_t wifi_init()
 {
  …
  ESP_ERROR_CHECK(esp_wifi_start());    //Enable the Wi-Fi interface
  return ESP_OK;                        //End of the Wi-Fi Initializer Function
 }
```

Once the `esp_wifi_start()` function returns successfully the Wi-Fi interface will be enabled in the mode(s) previously defined by the `esp_wifi_set_mode()` function call, which will cause the SYSTEM_EVENT_STA_START and/or the SYSTEM_EVENT_AP_START events to trigger in the Wi-Fi stack, thereby causing the Wi-Fi setup process to pass into its event-driven phase.

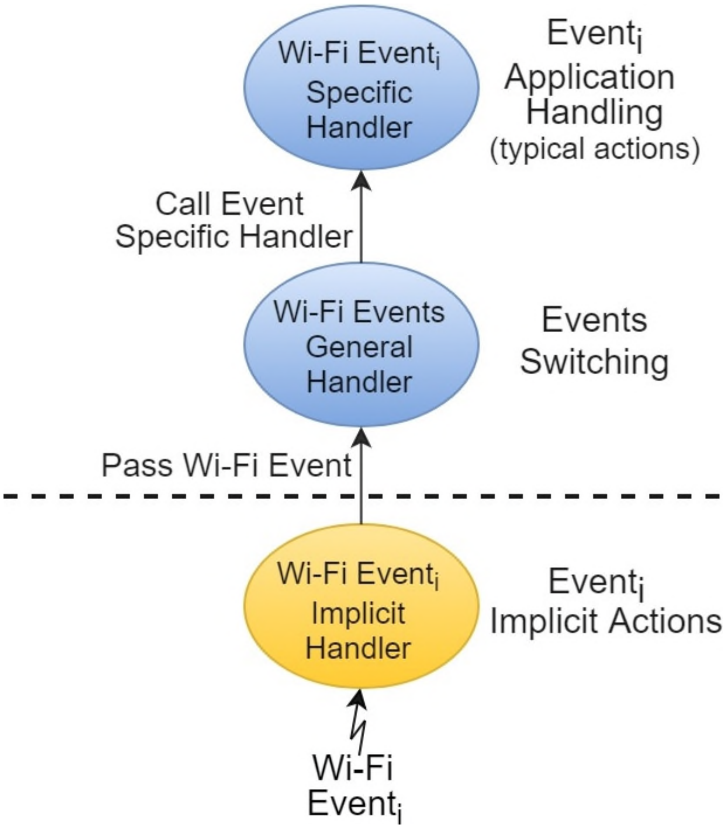
Summarizing, the tasks that must be performed by the Wi-Fi Initializer Function are:


Wi-Fi Initializer Function
1) Initialize the system flash storage
nvs_flash_init()
2) Initialize the TCP/LwIP stack
tcpip_adapter_init()
3) Create the Wi-Fi Event Group
xEventGroupCreate()
4) Register the Wi-Fi Events General Handler
esp_event_loop_init()
5) Initialize the Wi-Fi Stack
esp_wifi_init()
6) Interface Modes Selection
esp_wifi_set_mode()
7) Station Base Configuration
SoftAP Base Configuration
esp_wifi_set_config()
8) Other Station Settings
Other SoftAP Settings
• esp_wifi_set_mac()
• esp_wifi_set_ps()
• ...
9) Station IP Settings
SoftAP IP Settings
• Set a static custom IP configuration for an interface
10) Enable the Wi-Fi Interface
esp_wifi_start()

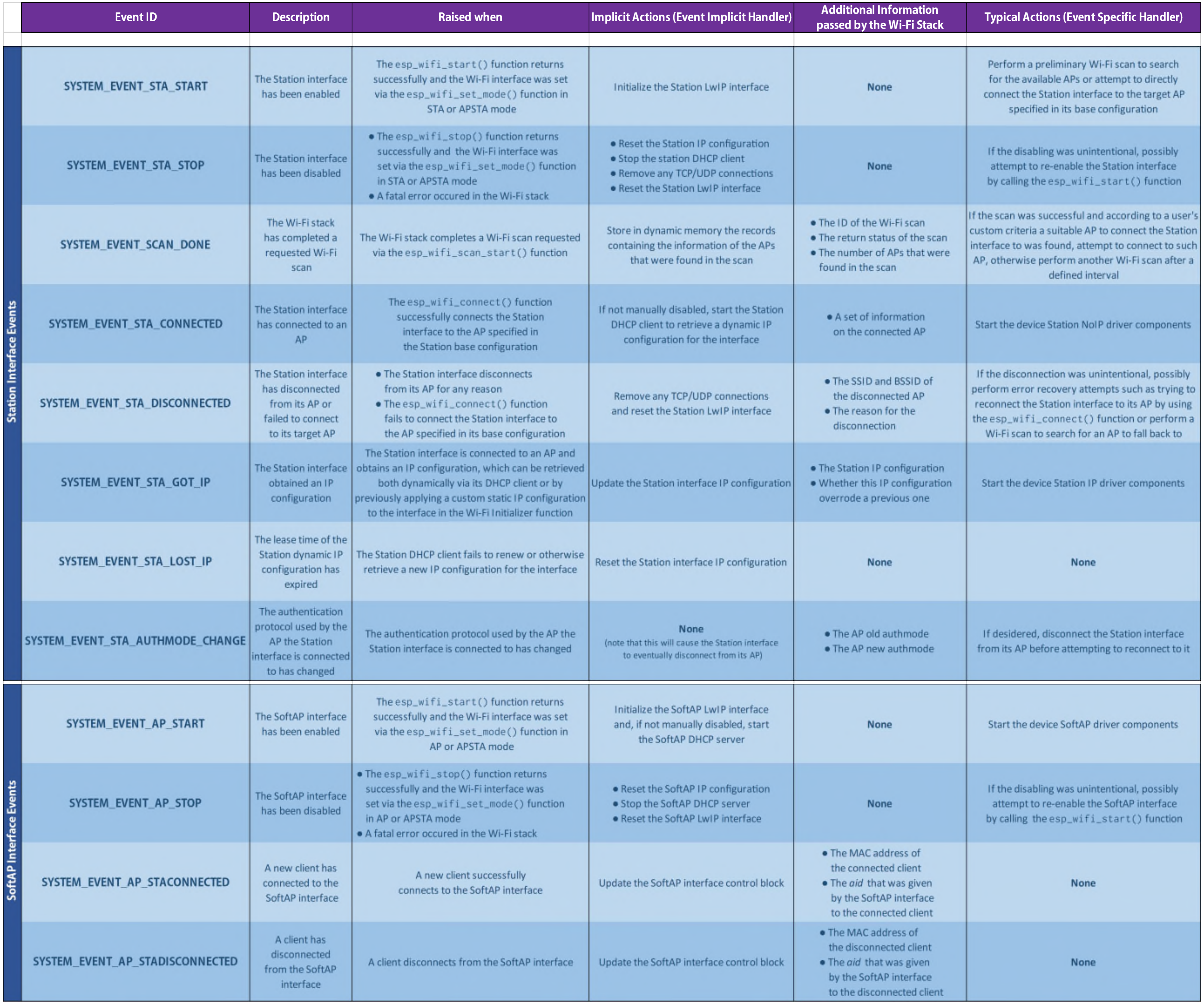
Also note that the majority of the Wi-Fi settings we have discussed so far can also be modified after the Wi-Fi interface has been enabled on a device, which may cause its Station and/or SoftAP interfaces to be restarted depending on the changes applied.

Event-driven Setup Phase

Once the Wi-Fi interface has been enabled on a device, the Wi-Fi setup process enters its event-driven phase, which consists in the application-level handling of the **Wi-Fi Events** raised by the Wi-Fi Stack.
A Wi-Fi Event can be represented as a set of conditions on the state of the Wi-Fi stack on the device which, other than evolving internally through the application's execution, is affected by the data received from the Wi-Fi interface, and once all the conditions representing an event have been met, such event will be raised by the Wi-Fi stack.
Once raised, a Wi-Fi event is first handled internally in the Wi-Fi stack by its **Specific Implicit Handler**, which performs a set of **Implicit Actions** that depend on other conditions relative to the state of the Wi-Fi stack.
Once its internal handling is complete, the event with its ID and a set of additional information are passed by the Wi-Fi stack at the application level to the **Wi-Fi Events General Handler** function previously registered via the `esp_event_loop_init()` function, 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 state of the Wi-Fi stack, the additional information provided and possibly the intended logic of the setup process.


Wi-Fi Event$_i$ Specific Handler
Event$_i$ Application Handling (typical actions)
Call Event Specific Handler
Wi-Fi Events General Handler
Events Switching
Pass Wi-Fi Event
Wi-Fi Event$_i$ Implicit Handler
Event$_i$ Implicit Actions
Wi-Fi Event$_i$

The Wi-Fi events that can be raised by the Wi-Fi stack can be divided into **Station Interface Events**, which are relative to the Station interface, and **SoftAP Interface Events**, which are relative to the SoftAP interface.
The full list of Wi-Fi events with their description, the implicit actions performed by their implicit handlers, the additional information passed at the application-level by the Wi-Fi stack, and the typical actions that should be performed by their specific handlers are summarized in the following table:



	Event ID	Description	Raised when	Implicit Actions (Event Implicit Handler)	Additional Information passed by the Wi-Fi Stack	Typical Actions (Event Specific Handler)
Station Interface Events	SYSTEM_EVENT_STA_START	The Station interface has been enabled	The `esp_wifi_start()` function returns successfully and the Wi-Fi interface was set via the `esp_wifi_set_mode()` function in STA or APSTA mode	Initialize the Station LwIP interface	**None**	Perform a preliminary Wi-Fi scan to search for the available APs or attempt to directly connect the Station interface to the target AP specified in its base configuration
	SYSTEM_EVENT_STA_STOP	The Station interface has been disabled	● The `esp_wifi_stop()` function returns successfully and the Wi-Fi interface was set via the `esp_wifi_set_mode()` function in STA or APSTA mode ● A fatal error occured in the Wi-Fi stack	● Reset the Station IP configuration ● Stop the station DHCP client ● Remove any TCP/UDP connections ● Reset the Station LwIP interface	**None**	If the disabling was unintentional, possibly attempt to re-enable the Station interface by calling the `esp_wifi_start()` function
	SYSTEM_EVENT_SCAN_DONE	The Wi-Fi stack has completed a requested Wi-Fi scan	The Wi-Fi stack completes a Wi-Fi scan requested via the `esp_wifi_scan_start()` function	Store in dynamic memory the records containing the information of the APs that were found in the scan	● The ID of the Wi-Fi scan ● The return status of the scan ● The number of APs that were found in the scan	If the scan was successful and according to a user's custom criteria a suitable AP to connect the Station interface to was found, attempt to connect to such AP, otherwise perform another Wi-Fi scan after a defined interval
	SYSTEM_EVENT_STA_CONNECTED	The Station interface has connected to an AP	The `esp_wifi_connect()` function successfully connects the Station interface to the AP specified in the Station base configuration	If not manually disabled, start the Station DHCP client to retrieve a dynamic IP configuration for the interface	● A set of information on the connected AP	Start the device Station NoIP driver components
	SYSTEM_EVENT_STA_DISCONNECTED	The Station interface has disconnected from its AP or failed to connect to its target AP	● The Station interface disconnects from its AP for any reason ● The `esp_wifi_connect()` function fails to connect the Station interface to the AP specified in its base configuration	Remove any TCP/UDP connections and reset the Station LwIP interface	● The SSID and BSSID of the disconnected AP ● The reason for the disconnection	If the disconnection was unintentional, possibly perform error recovery attempts such as trying to reconnect the Station interface to its AP by using the `esp_wifi_connect()` function or perform a Wi-Fi scan to search for an AP to fall back to
	SYSTEM_EVENT_STA_GOT_IP	The Station interface obtained an IP configuration	The Station interface is connected to an AP and obtains an IP configuration, which can be retrieved both dynamically via its DHCP client or by previously applying a custom static IP configuration to the interface in the Wi-Fi Initializer function	Update the Station interface IP configuration	● The Station IP configuration ● Whether this IP configuration overrode a previous one	Start the device Station IP driver components
	SYSTEM_EVENT_STA_LOST_IP	The lease time of the Station dynamic IP configuration has expired	The Station DHCP client fails to renew or otherwise retrieve a new IP configuration for the interface	Reset the Station interface IP configuration	**None**	**None**
	SYSTEM_EVENT_STA_AUTHMODE_CHANGE	The authentication protocol used by the AP the Station interface is connected to has changed	The authentication protocol used by the AP the Station interface is connected to has changed	**None** (note that this will cause the Station interface to eventually disconnect from its AP)	● The AP old authmode ● The AP new authmode	If desidered, disconnect the Station interface from its AP before attempting to reconnect to it
SoftAP Interface Events	SYSTEM_EVENT_AP_START	The SoftAP interface has been enabled	The `esp_wifi_start()` function returns successfully and the Wi-Fi interface was set via the `esp_wifi_set_mode()` function in AP or APSTA mode	Initialize the SoftAP LwIP interface and, if not manually disabled, start the SoftAP DHCP server	**None**	Start the device SoftAP driver components
	SYSTEM_EVENT_AP_STOP	The SoftAP interface has been disabled	● The `esp_wifi_stop()` function returns successfully and the Wi-Fi interface was set via the `esp_wifi_set_mode()` function in AP or APSTA mode ● A fatal error occured in the Wi-Fi stack	● Reset the SoftAP IP configuration ● Stop the SoftAP DHCP server ● Reset the SoftAP LwIP interface	**None**	If the disabling was unintentional, possibly attempt to re-enable the SoftAP interface by calling the `esp_wifi_start()` function
	SYSTEM_EVENT_AP_STACONNECTED	A new client has connected to the SoftAP interface	A new client successfully connects to the SoftAP interface	Update the SoftAP interface control block	● The MAC address of the connected client ● The *aid* that was given by the SoftAP interface to the connected client	**None**
	SYSTEM_EVENT_AP_STADISCONNECTED	A client has disconnected from the SoftAP interface	A client disconnects from the SoftAP interface	Update the SoftAP interface control block	● The MAC address of the disconnected client ● The *aid* that was given by the SoftAP interface to the disconnected client	**None**

Wi-Fi Events General Handler

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

Wi-Fi Events ID Definitions

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

typedef enum                               //Wi-Fi Events IDs
 {
  /*-- Station Interface Events --*/
  SYSTEM_EVENT_STA_START,
  SYSTEM_EVENT_STA_STOP,
  SYSTEM_EVENT_SCAN_DONE,
  SYSTEM_EVENT_STA_CONNECTED,
  SYSTEM_EVENT_STA_DISCONNECTED,
  SYSTEM_EVENT_STA_GOT_IP,
  SYSTEM_EVENT_STA_LOST_IP,
  SYSTEM_EVENT_STA_AUTHMODE_CHANGE,

  /*-- SoftAP Interface Events --*/
  SYSTEM_EVENT_AP_START,
  SYSTEM_EVENT_AP_STOP,
  SYSTEM_EVENT_AP_STACONNECTED,
  SYSTEM_EVENT_AP_STADISCONNECTED,
  …
 } system_event_id_t;
```

Wi-Fi Events Additional Information Types

Station Interface Events

```
/*============================= SYSTEM_EVENT_SCAN_DONE =============================*/

//File esp_event_legacy.h

typedef struct
 {
  uint8_t  scan_id;            //The ID of the Wi-Fi scan
  uint32_t status;             //The return status of the Wi-Fi scan
  uint8_t  number;             //The number of APs that were found in the scan
 } system_event_sta_scan_done_t;

/*=========================== SYSTEM_EVENT_STA_CONNECTED ===========================*/

//File esp_event_legacy.h

typedef struct
 {
  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 used by the AP the Station connected to
  wifi_auth_mode_t authmode;   //The authmode used by the AP the Station connected to
 } system_event_sta_connected_t;

/*========================== SYSTEM_EVENT_STA_DISCONNECTED ==========================*/

//File esp_event_legacy.h

typedef struct
 {
  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 (wifi_err_reason_t)
 } system_event_sta_disconnected_t;

/*============================= SYSTEM_EVENT_STA_GOT_IP =============================*/

//File esp_event_legacy.h

typedef struct
 {
  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

/*============================= SYSTEM_EVENT_STA_GOT_IP =============================*/

//File esp_event_legacy.h

typedef struct
 {
  wifi_authmode_t old_mode;        //The AP old authmode
  wifi_authmode_t old_mode;        //The AP new authmode
 } system_event_sta_authmode_change_t;
```

SoftAP Interface Events

```
/*========================== SYSTEM_EVENT_AP_STACONNECTED ==========================*/

//File esp_event_legacy.h

typedef struct
 {
  uint8_t mac[6];       //The MAC address of the connected client
  uint_t aid;           //The aid that was given by the SoftAP to the connected client
 } system_event_ap_staconnected_t;

/*========================= SYSTEM_EVENT_AP_STADISCONNECTED =========================*/

//File esp_event_legacy.h

typedef struct
 {
  uint8_t mac[6];    //The MAC address of the disconnected client
  uint_t aid;        //The aid that was given by the SoftAP to the disconnected client
 } system_event_ap_stadisconnected_t;
```

Wi-Fi Events Additional Information union

```
typedef union        //Wi-Fi Events additional information union
 {
  /*-- Station Interface Events additional information --*/
  system_event_sta_scan_done_t scan_done;           //SYSTEM_EVENT_SCAN_DONE
  system_event_sta_connected_t connected;           //SYSTEM_EVENT_STA_CONNECTED
  system_event_sta_disconnected_t disconnected;     //SYSTEM_EVENT_STA_DISCONNECTED
  system_event_sta_got_ip_t got_ip;                 //SYSTEM_EVENT_STA_GOT_IP
  system_event_sta_authmode_change_t auth_change;   //SYSTEM_EVENT_STA_AUTHMODE_CHANGE

  /*-- SoftAP Interface Events additional information --*/
  system_event_ap_staconnected_t sta_connected;       //SYSTEM_EVENT_AP_STACONNECTED
  system_event_ap_stadisconnected_t sta_disconnected; //SYSTEM_EVENT_AP_STADISCONNECTED
 } system_event_info_t;
```

Wi-Fi Events Summary Struct

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

```
typedef struct                     //Summary struct passed by the Wi-Fi stack
 {                                   to the Wi-Fi Events General Handler
  system_event_id_t event_id;      //Event ID
  system_event_info_t event_info;  //Event-specific information (if applicable)
 } system_event_t;
```

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

```
esp_err_t wifi_events_handler(void* ctx, system_event_t* event)
```

Where:

- The ctx parameter is reserved for the user (it should typically be ignored within the function).
- The event parameter represents the address of the summary struct passed by the Wi-Fi stack containing the information on the Wi-Fi event that has occured.

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

```
esp_err_t wifi_events_handler(void* ctx, system_event_t* event)
 {
  switch(event->event_id)
   {
    case EVENT1:
     wifi_EVENT1_handler(&event->event_info.EVENT1_t); //call EVENT1 specific handler
     break;
    case EVENT2:
     wifi_EVENT2_handler(&event->event_info.EVENT2_t); //call EVENT2 specific handler
     break;

    …

    case EVENTN:
     wifi_EVENTN_handler(&event->event_info.EVENTN_t); //call EVENTN specific handler
     break;

    default:
     ESP_LOGE(TAG,"Unknown Wi-Fi Event with ID: %u",event->event_id);
     break;
   }
  return ESP_OK;
 }
```

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

```
esp_err_t wifi_events_handler(void* ctx, system_event_t* event)
 {
  switch(event->event_id)
   {
    /*-- Station Interface Events --*/
    case SYSTEM_EVENT_STA_START:
     wifi_STA_START_handler();
     break;
    case SYSTEM_EVENT_STA_STOP:
     wifi_STA_STOP_handler();
     break;
    case SYSTEM_EVENT_SCAN_DONE:
     wifi_SCAN_DONE_handler(&event->event_info.scan_done);
     break;
    case SYSTEM_EVENT_STA_CONNECTED:
     wifi_STA_CONNECTED_handler(&event->event_info.connected);
     break;
    case SYSTEM_EVENT_STA_DISCONNECTED:
     wifi_STA_DISCONNECTED_handler(&event->event_info.disconnected);
     break;
    case SYSTEM_EVENT_STA_GOT_IP:
     wifi_STA_GOT_IP_handler(&event->event_info.got_ip);
     break;
    case SYSTEM_EVENT_STA_LOST_IP:
     wifi_STA_LOST_IP_handler();
     break;
    case SYSTEM_EVENT_STA_AUTHMODE_CHANGE:
     wifi_STA_AUTHMODE_CHANGE_handler(&event->event_info.auth_change);
     break;

    /*-- SoftAP Interface Events --*/
    case SYSTEM_EVENT_AP_START:
     wifi_AP_START_handler();
     break;
    case SYSTEM_EVENT_AP_STACONNECTED:
     wifi_AP_STACONNECTED_handler(&event->event_info.sta_connected);
     break;
    case SYSTEM_EVENT_AP_STADISCONNECTED:
     wifi_AP_STADISCONNECTED_handler(&event->event_info.sta_disconnected);
     break;

    default:
     ESP_LOGE(TAG,"Unknown Wi-Fi Event with ID: %u",event->event_id);
     break;
   }
  return ESP_OK;
 }
```

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

Wi-Fi Events Specific Handlers

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

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

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

Described below are the typical actions that should be performed by each Wi-Fi specific handler:

Station Interface Events

SYSTEM_EVENT_STA_START

This event is raised after the Station interface has been enabled on the device, which is obtained by calling the `esp_wifi_start()` function after having set the Wi-Fi interface in STA or APSTA mode via the `esp_wifi_set_mode()` function.

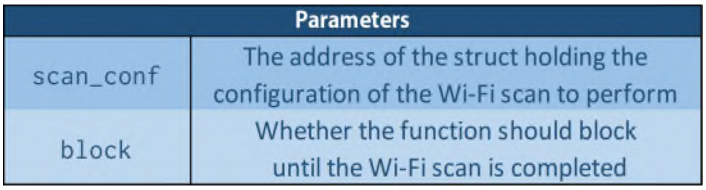
From here, while this event's implicit handler will have initialized the Station network interface, at the application-level, in addition to setting the STA_ON flag in the Wi-Fi event group, it's either possible to perform a preliminary Wi-Fi scan to search for the available APs or attempt to directly connect the Station interface to the target AP specified in its base configuration.

- If a preliminary Wi-Fi scan is desidered, this can be started by calling the following function:

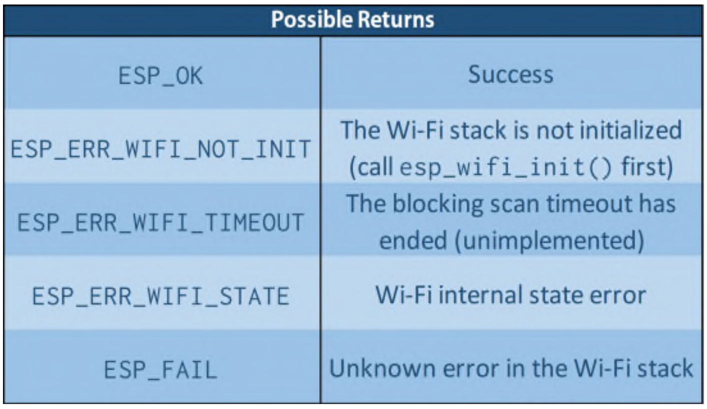
```
//File esp_wifi_types.h

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



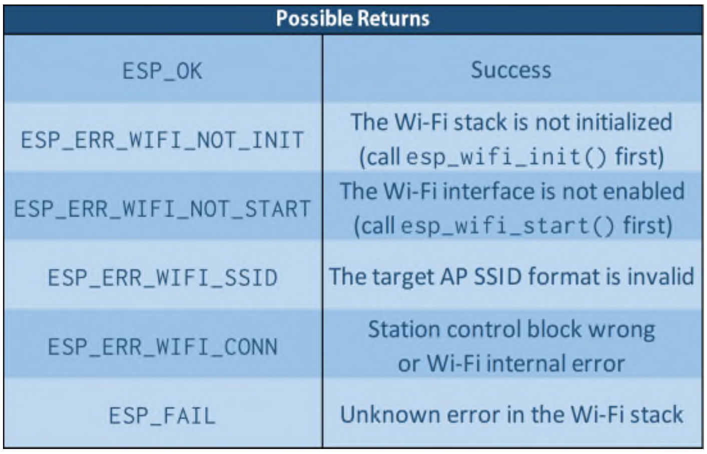
Possible Returns	
ESP_OK	Success
ESP_ERR_WIFI_NOT_INIT	The Wi-Fi stack is not initialized (call esp_wifi_init() first)
ESP_ERR_WIFI_TIMEOUT	The blocking scan timeout has ended (unimplemented)
ESP_ERR_WIFI_STATE	Wi-Fi internal state error
ESP_FAIL	Unknown error in the Wi-Fi stack

From here, all that is needed to perform an active all-channel Wi-Fi scan with the default scan times, is to initialize the entire `wifi_scan_config_t` struct to "0", and the function itself should be used in its non-blocking version (`block = false`), since once the Wi-Fi scan is complete the SYSTEM_EVENT_SCAN_DONE event will be raised by the Wi-Fi stack, from whose specific handler it is possible to check the scan results.

- If instead the Station interface should attempt to directly connect to the target AP specified in its base configuration, this can be done by calling the following function:

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

```
esp_err_t esp_wifi_connect(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_NOT_START	The Wi-Fi interface is not enabled (call esp_wifi_start() first)
ESP_ERR_WIFI_SSID	The target AP SSID format is invalid
ESP_ERR_WIFI_CONN	Station control block wrong or Wi-Fi internal error
ESP_FAIL	Unknown error in the Wi-Fi stack

This function implicitly performs an active all-channel Wi-Fi scan for the available APs without triggering the SYSTEM_EVENT_SCAN_DONE event where, if at the end of the scan the target AP specified in the Station base configuration has been found, a connection attempt will be performed (where the SYSTEM_EVENT_STA_CONNECTED event will be raised in case of success), otherwise if APs with hidden SSIDs have been found, the device will attempt to directly connect to each of them, and lastly if the target AP was not found a SYSTEM_EVENT_STA_DISCONNECTED event will be raised by the Wi-Fi stack.

```
void wifi_STA_START_handler()
 {
  wifi_scan_config_t scan_config = {0};       //Preliminary Wi-Fi Scan configuration

  xEventGroupSetBits(wifi_event_group,STA_ON);//Set the STA_ON flag
  if(WIFI_STATION_PRELIMINARY_SCAN)           //If performing a preliminary Wi-Fi scan
   ESP_ERROR_CHECK(esp_wifi_scan_start(&scan_config,false));
  else                                        //Otherwise if attempting to
   ESP_ERROR_CHECK(esp_wifi_connect());         directly connect the Station
  return;                                       interface to its target AP
 }
```

SYSTEM_EVENT_STA_STOP

This event may be raised both because the Station interface was disabled intentionally by calling the `esp_wifi_stop()` function (we'll see later), or due to a fatal error in the Wi-Fi stack, and in addition to clearing the STA_ON flag in the Wi-Fi event group, should the disabling have occured unintentionally, as an error recovery attempt it is possible to try to re-enable the Station interface by calling the `esp_wifi_start()` function.
Also note that the handling of errors that might occur in the application's logic due to the disabling of the Station interface is left entirely to the Driver Module.

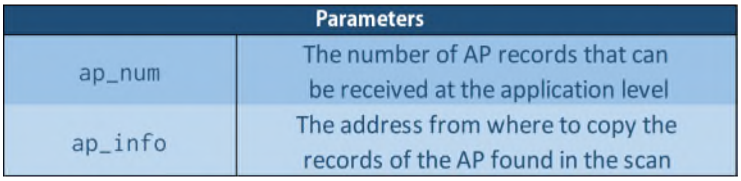
```
void wifi_STA_STOP_handler()
{
 xEventGroupClearBits(wifi_event_group,STA_ON); //Clear the STA_ON flag
 if(/* unintentional */)                        //If the disabling was unintentional,
  ESP_ERROR_CHECK(esp_wifi_start());              try to reenable the Station interface
 return;
}
```

SYSTEM_EVENT_SCAN_DONE

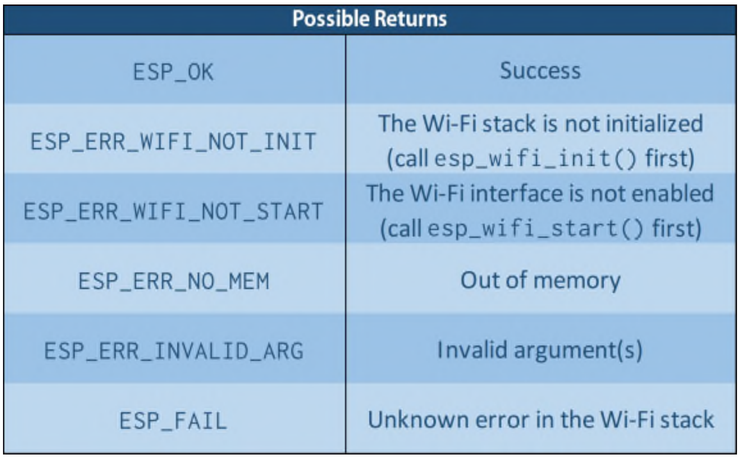
This event is raised after the Wi-Fi stack completes a Wi-Fi scan requested through the `esp_wifi_scan_start()` function, and from here if the scan was completed successfully (`info->status == ESP_OK`) and at least one AP was found (`info->number > 0`), to retrieve the information on the APs that were found in the scan, in addition to allocating the dynamic memory required to store it, the following function must be called:

```
//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;
```

```
esp_err_t esp_wifi_scan_get_ap_records(uint16_t* ap_num,
                                       wifi_ap_record_t* ap_info)
```



Parameters	
ap_num	The number of AP records that can be received at the application level
ap_info	The address from where to copy the records of the AP found in the scan



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_NOT_START	The Wi-Fi interface is not enabled (call esp_wifi_start() first)
ESP_ERR_NO_MEM	Out of memory
ESP_ERR_INVALID_ARG	Invalid argument(s)
ESP_FAIL	Unknown error in the Wi-Fi stack

From here, by parsing the records of the APs that were found in the scan and by applying a user's custom criteria, if a suitable AP to connect to is found, if it's not the target AP specified in the Station base configuration, such configuration should be updated with the AP's information, and then a connection attempt should be performed by using the `esp_wifi_connect()` function described earlier, while in all other cases, since no suitable AP to connect was found, a new Wi-Fi scan should be scheduled after a defined time interval to search again for an AP to connect to.

```
void wifi_SCAN_DONE_handler(system_event_sta_scan_done_t* info)
{
 wifi_scan_config_t scan_config = {0}; //Configuration of the Wi-Fi scan to perform
                                         should no suitable AP to connect to be found
 wifi_ap_record_t* ap_list = 0;    //Used to dynamically store the found APs' records
 uint16_t ap_num = info->number;   //Number of APs that were found in the Wi-Fi scan

 if((!info->status)&&ap_num)       //If the Wi-Fi scan completed successfully
  {                                  and at least one AP was found
   //Allocate the dynamic memory required to store the AP records
   ap_list = (wifi_ap_record_t*)malloc(ap_num*sizeof(wifi_ap_record_t));
   //Retrieve the AP records from the Wi-Fi stack
   ESP_ERROR_CHECK(wifi_scan_get_ap_records(&ap_num,ap_list));

   /* Parse the AP records for a suitable AP to connect the Station interface to */

   if(/* a suitable AP to connect the Station interface to was found */)
    {
     if(/* the suitable AP differs from the Station target AP */)
      /* Update the Station base configuration with the information of the AP */
     ESP_ERROR_CHECK(esp_wifi_connect());
    }
   else  //If no suitable AP to connect to was found, perform
    {      another Wi-Fi scan after a defined interval
     vTaskDelay(WIFI_STATION_SCAN_RETRY_INTERVAL/portTICK_PERIOD_MS);
     ESP_ERROR_CHECK(esp_wifi_scan_start(&scan_config,false));
    }
   free(ap_list); //Release the dynamic memory used to store the APs records
  }
 else    //If the Wi-Fi scan returned an error or no APs were found,
  {        perform another Wi-Fi scan after a defined interval
   vTaskDelay(WIFI_STATION_SCAN_RETRY_INTERVAL/portTICK_PERIOD_MS);
   ESP_ERROR_CHECK(esp_wifi_scan_start(&scan_config,false));
  }
 return;
}
```

SYSTEM_EVENT_STA_CONNECTED

This event is raised once the Station interface successfully connects to the AP specified in its base configuration via the `esp_wifi_connect()` function, and while this event's implicit handler, if not previously manually disabled, will have started the Station DHCP client to request a dynamic IP configuration for the interface, in the application-level handler, in addition to setting the STA_CONN flag in the Wi-Fi event group, the device's Station NoIP driver components, i.e. the driver components that require the device to be connected to an AP, can be started.

```
void wifi_STA_CONNECTED_handler(system_event_sta_connected_t* info)
 {
  xEventGroupSetBits(wifi_event_group,STA_CONN); //Set the STA_CONN flag
  startStationNoIPDrivers();                     //Start Station NoIP driver components
  return;
 }
```

SYSTEM_EVENT_STA_DISCONNECTED

This event is raised if the Station interface disconnects from its AP for any reason or if the `esp_wifi_connect()` function fails to connect the Station interface to the AP specified in its base configuration, where the two circumstances can be discriminated by checking and appropriately managing the STA_CONN flag in the Wi-Fi event group.

From here, in addition to clearing the STA_CONN flag in the Wi-Fi event group if the Station interface disconnected from its AP, at the application-level, also based on the *reason* (`info->reason`) for the disconnection, it's possible to perform error recovery attempts consisting in either trying to re-connect the Station interface to the AP by calling the `esp_wifi_connect()` function or performing a Wi-Fi scan to search for a backup AP to fall back to, and note that again the handling of errors that might occur in the application's logic due to the Station interface becoming disconnected is left entirely to the Driver Module.

```
void wifi_STA_DISCONNECTED_handler(system_event_sta_disconnected_t* info)
 {
  EventBits_t eventbits;     //Used to check the flags in the Wi-Fi event group

  //If the STA_CONN flag is set, the Station interface disconnected from its AP
  if((eventbits = xEventGroupGetBits(wifi_event_group))>>1)&1)
   xEventGroupClearBits(wifi_event_group,STA_CONN);
  //Otherwise, the esp_wifi_connect() function failed to connect the
    Station interface to the AP specified in its base configuration
  if(/* unintentional */)
   {
    /* Error recovery attempts (try to re-connect the Station
       interface by calling the esp_wifi_connect() function or
       perform a Wi-Fi scan to search for an AP to fall back to */
   }
  return;
 }
```

SYSTEM_EVENT_STA_GOT_IP

This event is raised if a Station interface which is connected to an AP obtains an IP configuration, which as discussed before can be retrieved both dynamically from its DHCP client or by previously applying a custom static IP configuration to the interface in the Wi-Fi Initializer Function.
From here, while this event's implicit handler will have updated the Station IP configuration, in the application-level handler, in addition to setting the STA_GOTIP flag in the Wi-Fi event group, the device Station IP driver components, i.e. the driver components that require a device's Station interface to be connected to an AP and have an IP configuration set, can be started.

```
void wifi_STA_GOT_IP_handler(system_event_sta_got_ip_t* info)
 {
  xEventGroupSetBits(wifi_event_group,STA_GOTIP); //Set the STA_GOTIP flag
  startStationIPDrivers();                        //Start Station IP driver components
  return;
 }
```

SYSTEM_EVENT_STA_LOST_IP

This event is raised should the lease time of the Station interface dynamic IP configuration expire, which also implies that its DHCP client failed to renew or otherwise retrieve a new IP configuration for the interface.
From here, while this event's implicit handler will have reset the Station interface IP configuration, in the application-level handler, other than clearing the STA_GOTIP flag in the Wi-Fi event group, no further action is required, where again the handling of errors that might occur in the application's logic due to the Station interface having no longer an IP configuration set is left entirely to the Driver Module.

```
void wifi_STA_LOST_IP_handler()
 {
  xEventGroupClearBits(wifi_event_group,STA_GOTIP);  //Clear the STA_GOTIP flag
  return;
 }
```

SYSTEM_EVENT_STA_AUTHMODE_CHANGE

This event is raised should the AP the Station interface is connected to change its authentication protocol, where note that since this event's implicit handler performs no action this will cause the Station interface to eventually disconnect from its AP.
From here in the application-level handler, supposing that the new authentication protocol is supported and the same password is used, it is possible to attempt to reconnect the Station interface to its AP after disconnecting from it, which can be obtained by calling the following function:

```
//File esp_wifi.h

                    esp_err_t esp_wifi_disconnect(void)
```

Possible Returns	
ESP_OK	Success
ESP_ERR_WIFI_NOT_INIT	The Wi-Fi interface is not initialized (call esp_wifi_init() first)
ESP_ERR_WIFI_NOT_START	The Wi-Fi interface is not enabled (call esp_wifi_start() first)
ESP_FAIL	Unknown error in the Wi-Fi stack

Once the Station interface has disconnected from its AP, supposing that no reconnection attempts or a Wi-Fi scan fallback is performed in the SYSTEM_EVENT_STA_DISCONNECTED event specific handler, it is possible to attempt to reconnect the interface to its AP by using the `esp_wifi_connect()` function discussed previously.

It should also be noted that, in the specific case the AP switched to no authentication (open), if a reconnection attempt is desired the password in the Station base configuration must be cleared beforehand.

```
void wifi_STA_AUTHMODE_CHANGE_handler(system_event_sta_authmode_change_t* info)
 {
  wifi_config_t sta_config;    //Possibly used to reset the password in the Station
                                 base configuration to allow it to reconnect to its
                                 AP should it have switched to an open authentication
  //If the AP the Station interface is connected to switched to
    an open authentication and a reconnection attempt is desired
  if((info->new_mode == WIFI_AUTH_OPEN)&&(/* is desired */))
   {
    //Retrieve the Station base configuration (we'll see in more detail later)
    ESP_ERROR_CHECK(esp_wifi_get_config(ESP_IF_WIFI_STA,&sta_config));

    //Reset the password in the Station base configuration
    sta_config.sta.password[0] = '\0';

    //Update the password in the Station base configuration
    ESP_ERROR_CHECK(esp_wifi_set_config(ESP_IF_WIFI_STA,&sta_config));
   }
  //Disconnect the Station interface from its AP
  ESP_ERROR_CHECK(esp_wifi_disconnect());
  //Attempt to reconnect the Station interface to its AP (supposing such attempt is
    not already performed in the SYSTEM_EVENT_STA_DISCONNECTED event specific handler)
  ESP_ERROR_CHECK(esp_wifi_connect());
  return;
 }
```

SoftAP Interface Events

SYSTEM_EVENT_AP_START

This event is raised after the SoftAP interface has been enabled on the device, which is obtained by calling the `esp_wifi_start()` function after having set the Wi-Fi interface in AP or APSTA mode via the `esp_wifi_set_mode()` function.

From here, while this event's implicit handler will have initialized the SoftAP network interface and, if not previously manually disabled, started its DHCP server, at the application-level, in addition to setting the SOFTAP_ON flag in the Wi-Fi event group, the device's Station SoftAP driver components, i.e. the driver components that require the SoftAP interface to be enabled, can be started.

```
void wifi_AP_START_handler()
 {
  xEventGroupSetBits(wifi_event_group,SOFTAP_ON);  //Set the SOFTAP_ON flag
  startSoftAPDrivers();                            //Start SoftAP driver components
  return;
 }
```

SYSTEM_EVENT_AP_STOP

This event may be raised both because the SoftAP interface was disabled intentionally by calling the `esp_wifi_stop()` function (we'll see later), or due to a fatal error in the Wi-Fi stack, and in addition to clearing the SOFTAP_ON flag in the Wi-Fi event group, should the disabling have occured unintentionally, as an error recovery attempt it is possible to try to re-enable the SoftAP interface by calling the `esp_wifi_start()` function.
Also note that the handling of errors that might occur in the application's logic due to the disabling of the SoftAP interface is left entirely to the Driver Module.

```
void wifi_AP_STOP_handler()
{
 xEventGroupClearBits(wifi_event_group,SOFTAP_ON);  //Clear the SOFTAP_ON flag
 if(/* unintentional */)                 //If the disabling was unintentional,
  ESP_ERROR_CHECK(esp_wifi_start());       try to reenable the SoftAP interface
 return;
}
```

SYSTEM_EVENT_AP_STACONNECTED

This event is raised when a new client successfully connects to the SoftAP interface, and apart from setting the SOFTAP_CLI flag in the Wi-Fi event group if it was not previously set (i.e. if it's the first client to connect to the SoftAP interface), no other action in required in the application-level handler of this event.

```
void wifi_AP_STACONNECTED_handler(system_event_ap_staconnected_t* info)
 {
  EventBits_t eventbits;  //Used to check the flags in the Wi-Fi event group

  //If the SOFTAP_CLI flag is not set in the Wi-Fi event group (i.e.
    this is the first client to connect to the SoftAP interface), set it
  if(!(((eventbits = xEventGroupGetBits(wifi_event_group))>>4)&1))
   xEventGroupSetBits(wifi_event_group,SOFTAP_CLI);
  return;
 }
```

SYSTEM_EVENT_AP_STADISCONNECTED

This event is raised when a client disconnects from the SoftAP interface, and other than clearing the SOFTAP_CLI flag in the Wi-Fi event group if it was the last child to disconnect from the node (which can be determined by using the `esp_wifi_ap_get_sta_list()` function that will be discussed in detail later), no other action is required in the application-level handler of this event.

```
void wifi_AP_STADISCONNECTED_handler(system_event_ap_stadisconnected_t* info)
 {
  wifi_sta_list_t clients;     //Used to store information on the clients
                                 connected to the SoftAP interface
  //Retrieve information on the clients connected to the SoftAP interface
  ESP_ERROR_CHECK(esp_wifi_ap_get_sta_list(&clients));
  //If no client is connected to the SoftAP interface, clear the SOFTAP_CLI flag
  if(clients.num == 0)
   xEventGroupClearBits(wifi_event_group,SOFTAP_CLI);
  return;
 }
```

Driver Module

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

- The **Station NoIP Driver Components**, i.e. the driver components that require the device's Station interface to be connected to an access point, as soon as such connection is established (SYSTEM_EVENT_STA_CONNECTED).

- The **Station IP Driver Components**, i.e. the driver components that require the device's Station interface to be connected to an access point and to have an IP configuration set, as soon as the Station interface acquires an IP configuration (SYSTEM_EVENT_STA_GOT_IP).

- The **SoftAP Driver Components**, i.e. the driver components that require the device's SoftAP interface to be enabled, as soon as it is (SYSTEM_EVENT_AP_START), components that generally may also require the interface to have clients connected, which can be checked by polling the SOFTAP_CLI flag in the Wi-Fi event group.

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 Wi-Fi 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 stationNoIPDriverComponent_A()
{
 /* Station NOIP driver component A logic */
 …
 //Clear this driver component's execution flag in the Wi-Fi event group
 xEventGroupClearBits(wifi_event_group,WIFI_DRIVER_STATION_NOIP_A);
 vTaskDelete(NULL);              //Delete this driver component's task
}

void stationNoIPDriverComponent_B()
{
 /* Station NOIP driver component B logic */
 …
 //Clear this driver component's execution flag in the Wi-Fi event group
 xEventGroupClearBits(wifi_event_group,WIFI_DRIVER_STATION_NOIP_B);
 vTaskDelete(NULL);              //Delete this driver component's task
}

…
```

```
/* Called at the end of the SYSTEM_EVENT_STA_CONNECTED event specific handler */
void startStationNOIPDrivers()
{
 EventBits_t eventbits;          //Used to check the flags in the Wi-Fi 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 Wi-Fi event group before starting its task  */

 if(((eventbits = xEventGroupGetBits(wifi_event_group))>>
    WIFI_DRIVER_STATION_NOIP_A)&1)
  vTaskDelete(componentA_handler);
 xEventGroupSetBits(wifi_event_group,WIFI_DRIVER_STATION_NOIP_A);
 xTaskCreate(stationNoIPDriverComponent_A,"StationNoIPdriverA",
             4096,NULL,10,&componentA_handler);

 if(((eventbits = xEventGroupGetBits(wifi_event_group))>>
      WIFI_DRIVER_STATION_NOIP_B)&1)
  vTaskDelete(componentB_handler);
 xEventGroupSetBits(wifi_event_group,WIFI_DRIVER_STATION_NOIP_B);
 xTaskCreate(stationNoIPDriverComponent_A,"StationNoIPdriverA",
             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 Wi-Fi API functions used and/or the appropriate flags in the Wi-Fi event group, whose values as we have seen are asynchronously updated by the appropriate event specific handlers during the application's execution.

Wi-Fi API for the Driver Module

Listed below are the functions offered by the ESP-IDF Wi-FI API organized into categories that can be used for developing the driver module of a Wi-Fi application:

Station-specific API

- ### Retrieve information on the AP the Station interface is connected to

```
//File esp_wifi.h

   esp_err_t esp_wifi_sta_get_ap_info(wifi_ap_record_t* apinfo)
```

Parameters	
apinfo	The address where to copy the information relative to the AP the Station interface is connected to

Possible Returns	
ESP_OK	Success
ESP_ERR_WIFI_CONN	The Station interface is not enabled (call esp_wifi_init() first)
ESP_ERR_WIFI_NOT_CONNECT	The Station interface is not connected to an AP
ESP_FAIL	Unknown error in the Wi-Fi stack

Example

```
void staAppDriver()
 {
  …
  wifi_ap_record_t apinfo;  //Used to store information on the AP
                              the Station interface is connected to
  ESP_ERROR_CHECK(esp_wifi_sta_get_ap_info(&apinfo));
  …
 }
```

SoftAP-specific API

- ### Retrieve information on the clients connected to the SoftAP interface

```
//File esp_wifi_types.h

#define ESP_WIFI_MAX_CONN_NUM (10) //Maximum number of clients that can be
                                     simultaneously connected to the SoftAP
                                     interface

typedef struct      //Information on a client connected to the SoftAP interface
 {
  uint8_t mac[6];   //MAC address of the client
  int8_t rssi;      //SoftAP's RSSI with the client
  …
 } wifi_sta_info_t;

typedef struct      //Information on the clients connected to the SoftAP interface
 {
  wifi_sta_info_t sta[ESP_WIFI_MAX_CONN_NUM];  //Client-specific information
  int num;          //Number of clients connected to the 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 on 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	SoftAP control block wrong or Wi-Fi internal error
ESP_ERR_INVALID_ARG	Invalid argument
ESP_FAIL	Unknown error in the Wi-Fi stack

Example

```
void softAPAppDriver()
 {
  …
  wifi_sta_info_t clients; //Used to store information on the clients
                                   connected to the SoftAP interface
  ESP_ERROR_CHECK(esp_wifi_ap_get_sta_list(&clients));
  …
 }
```

- **Deauthenticate one or more clients from the SoftAP interface**

```
//File esp_wifi.h

            esp_err_t esp_wifi_deauth_sta(uint16_t aid)
```

Parameters	
aid	The AID of the client to deauthenticate, or deauthenticate all clients if set to "0"

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_INVALID_ARG	Invalid argument
ESP_FAIL	Unknown error in the Wi-Fi stack

Example

```
void softAPAppDriver()
 {
  …
  ESP_ERROR_CHECK(esp_wifi_deauth_sta(CLIENT_AID));   //Deauthenticate a client
  ESP_ERROR_CHECK(esp_wifi_deauth_sta(0));          //Deauthenticate all clients
  …
 }
```

Wi-Fi Configuration Retrieval API

The following API allows to retrieve the configuration of the Wi-Fi stack on a device:

- **Retrieve the Wi-Fi interface modes (or sub-interfaces) selected for use**

```
//File esp_wifi.h

            esp_err_t esp_wifi_get_mode(wifi_mode_t* wifi_mode)
```

Parameters	
wifi_mode	The address where to copy the Wi-Fi interface modes selected for use

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

Example

```
void appDriver()
 {
  …
  wifi_mode_t wifi_mode;  //Used to store the Wi-Fi interface modes
                            selected for use (STA, AP or APSTA)
  ESP_ERROR_CHECK(esp_wifi_get_mode(&wifi_mode));
  …
 }
```

- **Retrieve an interface mode's base configuration**

```
//File esp_wifi.h

      esp_err_t esp_wifi_get_config(wifi_interface_t ifx_mode,
                                    wifi_config_t* base_conf)
```

Parameters	
ifx_mode	The interface mode for which to retrieve the base configuration ❑ ESP_IF_WIFI_STA → Station Interface ❑ ESP_IF_WIFI_AP → SoftAP Interface
base_conf	The address where to copy the interface mode's base configuration

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()
 {
  …
  wifi_config_t sta_config;      //Used to store an interface
  wifi_config_t softap_config;     mode's base configuration

  ESP_ERROR_CHECK(esp_wifi_get_config(ESP_IF_WIFI_STA,&sta_config);
  ESP_ERROR_CHECK(esp_wifi_get_config(ESP_IF_WIFI_AP,&softap_conf);
  …
 }
```

- **Retrieve an interface mode's MAC address**

```
//File esp_wifi.h

        esp_err_t esp_wifi_get_mac(wifi_interface_t ifx_mode,
                                   uint8_t[] mac)
```

Parameters	
ifx_mode	The interface mode for which to retrieve the MAC address ❑ ESP_IF_WIFI_STA → Station Interface ❑ ESP_IF_WIFI_AP → SoftAP Interface
mac	The address where to copy the interface mode'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 mac[6];  //Used to store an interface mode's MAC address

  ESP_ERROR_CHECK(esp_wifi_get_mac(ESP_IF_WIFI_STA,&mac));//Station MAC address
  …
 }
```

- **Retrieve the Power Saving Mode used on the Station interface**

```
//File esp_wifi.h

          esp_err_t esp_wifi_get_ps(wifi_ps_type_t* ps_mode)
```

Parameters	
ps_mode	The address where to copy the power saving mode used on the Station interface

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

Example

```
void appDriver()
 {
  …
  wifi_ps_type_t sta_psmode;  //Used to store the power saving mode
                                used on the Station interface
  ESP_ERROR_CHECK(esp_wifi_get_ps(&sta_psmode);
  …
 }
```

- **Retrieve an interface mode's 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 mode for which to retrieve the IP configuration ❑ TCPIP_ADAPTER_IF_STA → Station interface ❑ TCPIP_ADAPTER_IF_AP → SoftAP interface
ip_info	The address where to copy the interface mode'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 softap_ipinfo;   //Used to store an interface
                                             mode's IP configuration
  ESP_ERROR_CHECK(tcpip_adapter_get_ip_info(TCPIP_ADAPTER_IF_AP,&ipinfo));
  …
 }
```

- **Retrieve the address of an interface mode's DNS server**

```
//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 mode for which to retrieve a DNS server's address ❑ TCPIP_ADAPTER_IF_STA → Station interface ❑ TCPIP_ADAPTER_IF_AP → SoftAP interface
type	The type of DNS server for the interface mode for which to retrieve the address ❑ TCPIP_ADAPTER_DNS_MAIN → Primary DNS server ❑ TCPIP_ADAPTER_DNS_BACKUP → Secondary DNS server
addr	Where to copy the interface mode'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_addr1;   //Used to store the address of
  tcpip_adapter_dns_type_t dns_addr2;     an interface mode's DNS server

  ESP_ERROR_CHECK(tcpip_adapter_get_dns_info(TCPIP_ADAPTER_IF_AP,  //SoftAP
                                             TCPIP_ADAPTER_DNS_MAIN, Primary
                                             &dns_addr1));           DNS Server
  ESP_ERROR_CHECK(tcpip_adapter_get_dns_info(TCPIP_ADAPTER_IF_STA,  //Station
                                             TCPIP_ADAPTER_DNS_BACKUP,Second.
                                             &dns_addr2));           DNS Server
  …
 }
```

Wi-Fi Stop API

- ## Disable the Wi-Fi Interface

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

```
esp_err_t esp_wifi_stop(void)
```

Possible Returns	
ESP_OK	Success
ESP_ERR_WIFI_NOT_INIT	The Wi-Fi stack is not initialized (call esp_wifi_init() first)
ESP_FAIL	Unknown error in the Wi-Fi stack

Calling this function causes the disabling of the Wi-Fi interface modes that were selected for use via the esp_wifi_set_mode() function, causing the the SYSTEM_EVENT_STA_STOP and/or the SYSTEM_EVENT_AP_STOP events to be raised accordingly.

Example

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

- ## Deinitialize the Wi-Fi Stack

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

```
esp_err_t esp_wifi_deinit(void)
```

Possible Returns	
ESP_OK	Success
ESP_ERR_WIFI_NOT_STOPPED	The Wi-Fi interface is not disabled (call esp_wifi_stop() first)
ESP_FAIL	Unknown error in the Wi-Fi stack

This function, which can be called only if the Wi-Fi interface is disabled, causes the full release of the resources allocated to the device's Wi-Fi stack, and thus its complete deinitialization.

Example

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