ESP-NETIF

[中文]

The purpose of the ESP-NETIF library is twofold:

- It provides an abstraction layer for the application on top of the TCP/IP stack. This allows applications to choose between IP stacks in the future.
- The APIs it provides are thread-safe, even if the underlying TCP/IP stack APIs are not.

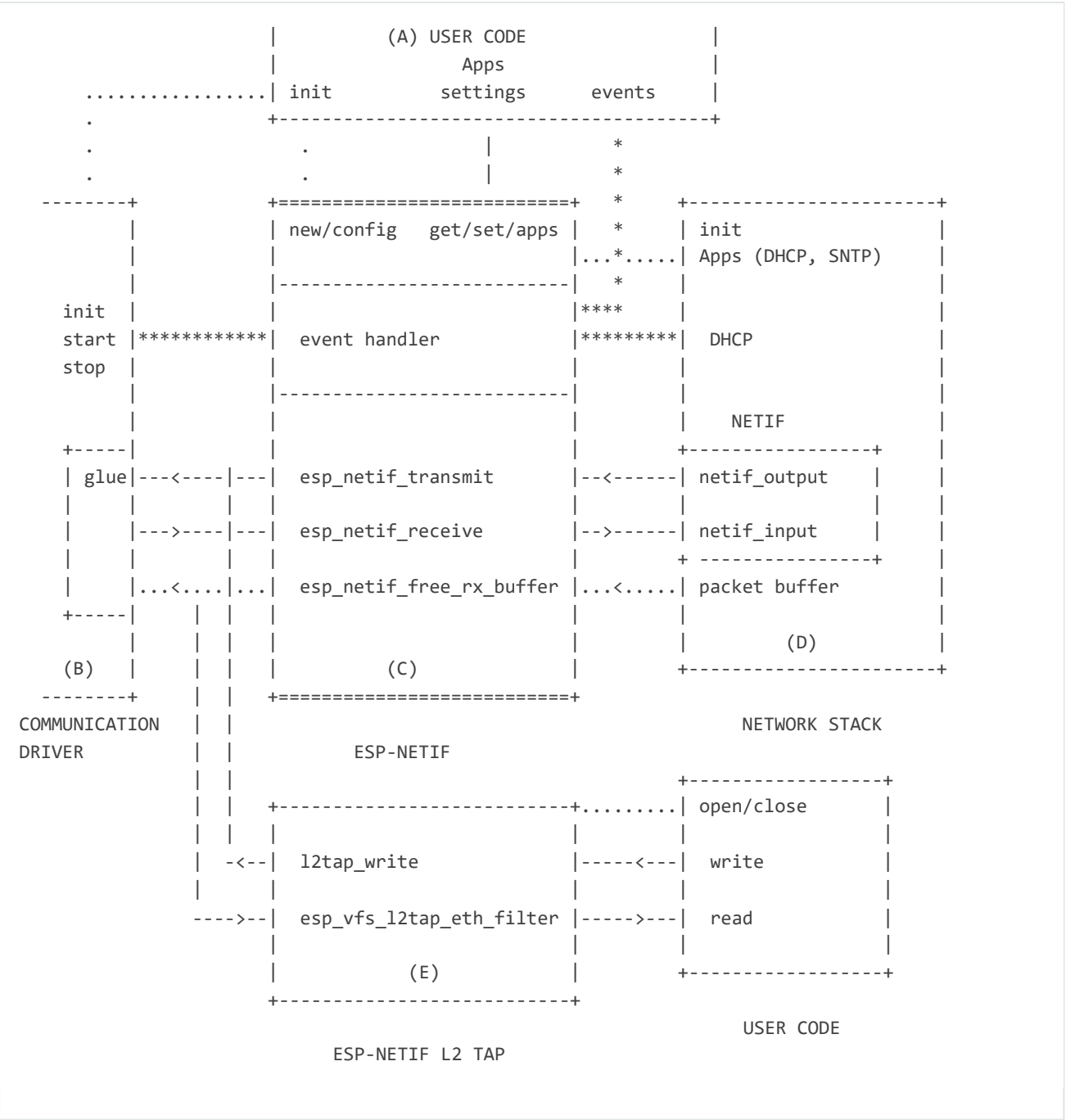
ESP-IDF currently implements ESP-NETIF for the lwIP TCP/IP stack only. However, the adapter itself is TCP/IP implementation-agnostic and allows different implementations.

It is also possible to use a custom TCP/IP stack with ESP-IDF, provided it implements BSD API. For more information on building ESP-IDF without lwIP, please refer to components/esp_netif_stack/README.md.

Some ESP-NETIF API functions are intended to be called by application code, for example, to get or set interface IP addresses, and configure DHCP. Other functions are intended for internal ESP-IDF use by the network driver layer.

In many cases, applications do not need to call ESP-NETIF APIs directly as they are called by the default network event handlers.

ESP-NETIF Architecture



Data and Event Flow in the Diagram

- `.....` Initialization line from user code to ESP-NETIF and communication driver
- `--<--->--` Data packets going from communication media to TCP/IP stack and back
- `********` Events aggregated in ESP-NETIF propagate to the driver, user code, and network stack
- `|` User settings and runtime configuration

ESP-NETIF Interaction

A) User Code, Boilerplate

Overall application interaction with a specific IO driver for communication media and configured TCP/IP network stack is abstracted using ESP-NETIF APIs and is outlined as below:

A. Initialization code

1. Initializes IO driver
2. Creates a new instance of ESP-NETIF and configure it with
    - ESP-NETIF specific options (flags, behavior, name)
    - Network stack options (netif init and input functions, not publicly available)
    - IO driver specific options (transmit, free rx buffer functions, IO driver handle)
3. Attaches the IO driver handle to the ESP-NETIF instance created in the above steps
4. Configures event handlers
    - Use default handlers for common interfaces defined in IO drivers; or define a specific handler for customized behavior or new interfaces
    - Register handlers for app-related events (such as IP lost or acquired)

B. Interaction with network interfaces using ESP-NETIF API

1. Gets and sets TCP/IP-related parameters (DHCP, IP, etc)
2. Receives IP events (connect or disconnect)
3. Controls application lifecycle (set interface up or down)

B) Communication Driver, IO Driver, and Media Driver

Communication driver plays these two important roles in relation to ESP-NETIF:

1. Event handlers: Defines behavior patterns of interaction with ESP-NETIF (e.g., ethernet link-up -> turn netif on)
2. Glue IO layer: Adapts the input or output functions to use ESP-NETIF transmit, receive, and free receive buffer
    - Installs driver_transmit to the appropriate ESP-NETIF object so that outgoing packets from the network stack are passed to the IO driver
    - Calls `esp_netif_receive()` to pass incoming data to the network stack

C) ESP-NETIF

ESP-NETIF serves as an intermediary between an IO driver and a network stack, connecting the packet data path between the two. It provides a set of interfaces for attaching a driver to an ESP-NETIF object at runtime and configures a network stack during compiling. Additionally, a set of APIs is provided to control the network interface lifecycle and its TCP/IP properties. As an overview, the ESP-NETIF public interface can be divided into six groups:

1. Initialization APIs (to create and configure ESP-NETIF instance)
2. Input or Output API (for passing data between IO driver and network stack)
3. Event or Action API

    - Used for network interface lifecycle management
    - ESP-NETIF provides building blocks for designing event handlers

4. Setters and Getters API for basic network interface properties
5. Network stack abstraction API: enabling user interaction with TCP/IP stack

    - Set interface up or down
    - DHCP server and client API
    - DNS API
    - SNTP API

6. Driver conversion utilities API

D) Network Stack

The network stack has no public interaction with application code with regard to public interfaces and shall be fully abstracted by ESP-NETIF API.

E) ESP-NETIF L2 TAP Interface

The ESP-NETIF L2 TAP interface is a mechanism in ESP-IDF used to access Data Link Layer (L2 per OSI/ISO) for frame reception and transmission from the user application. Its typical usage in the embedded world might be the implementation of non-IP-related protocols, e.g., PTP, Wake on LAN. Note that only Ethernet (IEEE 802.3) is currently supported.

From a user perspective, the ESP-NETIF L2 TAP interface is accessed using file descriptors of VFS, which provides file-like interfacing (using functions like `open()`, `read()`, `write()`, etc). To learn more, refer to Virtual Filesystem Component.

There is only one ESP-NETIF L2 TAP interface device (path name) available. However multiple file descriptors with different configurations can be opened at a time since the ESP-NETIF L2 TAP interface can be understood as a generic entry point to the Layer 2 infrastructure. What is important is then the specific configuration of the particular file descriptor. It can be configured to give access to a specific Network Interface identified by `if_key` (e.g., *ETH_DEF*) and to filter only specific frames based on their type (e.g., Ethernet type in the case of IEEE 802.3). Filtering only specific frames is crucial since the ESP-NETIF L2 TAP needs to exist along with the IP stack and so the IP-related traffic (IP, ARP, etc.) should not be passed directly to the user application. Even though this option is still configurable, it is not recommended in standard use cases. Filtering is also advantageous from the perspective of the user's application, as it only gets access to the frame types it is interested in, and the remaining traffic is either passed to other L2 TAP file descriptors or to the IP stack.

ESP-NETIF L2 TAP Interface Usage Manual

Initialization

To be able to use the ESP-NETIF L2 TAP interface, it needs to be enabled in Kconfig by CONFIG_ESP_NETIF_L2_TAP first and then registered by `esp_vfs_l2tap_intf_register()` prior usage of any VFS function.

`open()`

Once the ESP-NETIF L2 TAP is registered, it can be opened at path name "/dev/net/tap". The same path name can be opened multiple times up to CONFIG_ESP_NETIF_L2_TAP_MAX_FDS and multiple file descriptors with a different configuration may access the Data Link Layer frames.

The ESP-NETIF L2 TAP can be opened with the `O_NONBLOCK` file status flag to make sure the `read()` does not block. Note that the `write()` may block in the current implementation when accessing a Network interface since it is a shared resource among multiple ESP-NETIF L2 TAP file descriptors and IP stack, and there is currently no queuing mechanism deployed. The file status flag can be retrieved and modified using `fcntl()`.

On success, `open()` returns the new file descriptor (a nonnegative integer). On error, -1 is returned, and `errno` is set to indicate the error.

`ioctl()`

The newly opened ESP-NETIF L2 TAP file descriptor needs to be configured prior to its usage since it is not bounded to any specific Network Interface and no frame type filter is configured. The following configuration options are available to do so:

- `L2TAP_S_INTF_DEVICE` - bounds the file descriptor to a specific Network Interface that is identified by its `if_key`. ESP-NETIF Network Interface `if_key` is passed to `ioctl()` as the third parameter. Note that default Network Interfaces `if_key`'s used in ESP-IDF can be found in esp_netif/include/esp_netif_defaults.h.
- `L2TAP_S_DEVICE_DRV_HNDL` - is another way to bound the file descriptor to a specific Network Interface. In this case, the Network interface is identified directly by IO Driver handle (e.g., `esp_eth_handle_t` in case of Ethernet). The IO Driver handle is passed to `ioctl()` as the third parameter.
- `L2TAP_S_RCV_FILTER` - sets the filter to frames with the type to be passed to the file descriptor. In the case of Ethernet frames, the frames are to be filtered based on the Length and Ethernet type field. In case the filter value is set less than or equal to 0x05DC, the Ethernet type field is considered to represent IEEE802.3 Length Field, and all frames with values in interval <0, 0x05DC> at that field are passed to the file descriptor. The IEEE802.2 logical link control (LLC) resolution is then expected to be performed by the user's application. In case the filter value is set greater than 0x05DC, the Ethernet type field is considered to represent protocol identification and only frames that are equal to the set value are to be passed to the file descriptor.

All above-set configuration options have a getter counterpart option to read the current settings.

Warning

The file descriptor needs to be firstly bounded to a specific Network Interface by `L2TAP_S_INTF_DEVICE` or `L2TAP_S_DEVICE_DRV_HNDL` to make `L2TAP_S_RCV_FILTER` option available.

Note

VLAN-tagged frames are currently not recognized. If the user needs to process VLAN-tagged frames, they need a set filter to be equal to the VLAN tag (i.e., 0x8100 or 0x88A8) and process the VLAN-tagged frames in the user application.

Note

`L2TAP_S_DEVICE_DRV_HNDL` is particularly useful when the user's application does not require the usage of an IP stack and so ESP-NETIF is not required to be initialized too. As a result, Network Interface cannot be identified by its `if_key` and hence it needs to be identified directly by its IO Driver handle.

On success, `ioctl()` returns 0. On error, -1 is returned, and `errno` is set to indicate the error.
* EBADF - not a valid file descriptor.

* EACCES - options change is denied in this state (e.g., file descriptor has not been bounded to Network interface yet).
* EINVAL - invalid configuration argument. Ethernet type filter is already used by other file descriptors on that same Network interface.
* ENODEV - no such Network Interface which is tried to be assigned to the file descriptor exists.
* ENOSYS - unsupported operation, passed configuration option does not exist.

`fcntl()`

`fcntl()` is used to manipulate with properties of opened ESP-NETIF L2 TAP file descriptor.

The following commands manipulate the status flags associated with the file descriptor:

- `F_GETFD` - the function returns the file descriptor flags, and the third argument is ignored.
- `F_SETFD` - sets the file descriptor flags to the value specified by the third argument. Zero is returned.

On success, `ioctl()` returns 0. On error, -1 is returned, and `errno` is set to indicate the error.
* EBADF - not a valid file descriptor.
* ENOSYS - unsupported command.

`read()`

Opened and configured ESP-NETIF L2 TAP file descriptor can be accessed by `read()` to get inbound frames. The read operation can be either blocking or non-blocking based on the actual state of the `O_NONBLOCK` file status flag. When the file status flag is set to blocking, the read operation waits until a frame is received and the context is switched to other tasks. When the file status flag is set to non-blocking, the read operation returns immediately. In such case, either a frame is returned if it was already queued or the function indicates the queue is empty. The number of queued frames associated with one file descriptor is limited by CONFIG_ESP_NETIF_L2_TAP_RX_QUEUE_SIZE Kconfig option. Once the number of queued frames reached a configured threshold, the newly arrived frames are dropped until the queue has enough room to accept incoming traffic (Tail Drop queue management).

On success, `read()` returns the number of bytes read. Zero is returned when the size of the destination buffer is 0. On error, -1 is returned, and `errno` is set to indicate the error.
* EBADF - not a valid file descriptor.
* EAGAIN - the file descriptor has been marked non-blocking (`O_NONBLOCK`), and the read would block.

`write()`

A raw Data Link Layer frame can be sent to Network Interface via opened and configured ESP-NETIF L2 TAP file descriptor. The user's application is responsible to construct the whole frame except for fields which are added automatically by the physical interface device. The following fields need to be constructed by the user's application in case of an Ethernet link: source/destination MAC addresses, Ethernet type, actual protocol header, and user data. The length of these fields is as follows:

Destination MAC	Source MAC	Type/Length	Payload (protocol header/data)
6 B	6 B	2 B	0-1486 B

In other words, there is no additional frame processing performed by the ESP-NETIF L2 TAP interface. It only checks the Ethernet type of the frame is the same as the filter configured in the file descriptor. If the Ethernet type is different, an error is returned and the frame is not sent. Note that the `write()` may block in the current implementation when accessing a Network interface since it is a shared resource among multiple ESP-NETIF L2 TAP file descriptors and IP stack, and there is currently no queuing mechanism deployed.

On success, `write()` returns the number of bytes written. Zero is returned when the size of the input buffer is 0. On error, -1 is returned, and `errno` is set to indicate the error.
* EBADF - not a valid file descriptor.
* EBADMSG - The Ethernet type of the frame is different from the file descriptor configured filter.
* EIO - Network interface not available or busy.

`close()`

Opened ESP-NETIF L2 TAP file descriptor can be closed by the `close()` to free its allocated resources. The ESP-NETIF L2 TAP implementation of `close()` may block. On the other hand, it is thread-safe and can be called from a different task than the file descriptor is actually used. If such a situation occurs and one task is blocked in the I/O operation and another task tries to close the file descriptor, the first task is unblocked. The first's task read operation then ends with an error.

On success, `close()` returns zero. On error, -1 is returned, and `errno` is set to indicate the error.
* EBADF - not a valid file descriptor.

`select()`

Select is used in a standard way, just CONFIG_VFS_SUPPORT_SELECT needs to be enabled to make the `select()` function available.

SNTP API

You can find a brief introduction to SNTP in general, its initialization code, and basic modes in Section SNTP Time Synchronization in System Time.

This section provides more details about specific use cases of the SNTP service, with statically configured servers, or use the DHCP-provided servers, or both. The workflow is usually very simple:

1. Initialize and configure the service using `esp_netif_sntp_init()`. This operations can only be called once (unless the SNTP service has been destroyed by `esp_netif_sntp_deinit()`)
2. Start the service via `esp_netif_sntp_start()`. This step is not needed if we auto-started the service in the previous step (default). It is useful to start the service explicitly after connecting if we want to use the DHCP-obtained NTP servers. Please note, this option needs to be enabled before connecting, but the SNTP service should be started after.
3. Wait for the system time to synchronize using `esp_netif_sntp_sync_wait()` (only if needed).
4. Stop and destroy the service using `esp_netif_sntp_deinit()`.

Basic Mode with Statically Defined Server(s)

Initialize the module with the default configuration after connecting to the network. Note that it is possible to provide multiple NTP servers in the configuration struct:

```
esp_sntp_config_t config = ESP_NETIF_SNTP_DEFAULT_CONFIG_MULTIPLE(2,
                           ESP_SNTP_SERVER_LIST("time.windows.com", "pool.ntp.org" ) );
esp_netif_sntp_init(&config);
```

Note

If we want to configure multiple SNTP servers, we have to update the lwIP configuration CONFIG_LWIP_SNTP_MAX_SERVERS.

Use DHCP-Obtained SNTP Server(s)

First of all, we have to enable the lwIP configuration option CONFIG_LWIP_DHCP_GET_NTP_SRV. Then we have to initialize the SNTP module with the DHCP option and without the NTP server:

```
esp_sntp_config_t config = ESP_NETIF_SNTP_DEFAULT_CONFIG_MULTIPLE(0, {} );
config.start = false;                       // start the SNTP service explicitly
config.server_from_dhcp = true;             // accept the NTP offer from the DHCP server
esp_netif_sntp_init(&config);
```

Then, once we are connected, we could start the service using:

```
esp_netif_sntp_start();
```

> **Note**
>
> It is also possible to start the service during initialization (default `config.start=true`). This would likely to cause the initial SNTP request to fail (since we are not connected yet) and lead to some back-off time for subsequent requests.

Use Both Static and Dynamic Servers

Very similar to the scenario above (DHCP provided SNTP server), but in this configuration, we need to make sure that the static server configuration is refreshed when obtaining NTP servers by DHCP. The underlying lwIP code cleans up the rest of the list of NTP servers when the DHCP-provided information gets accepted. Thus the ESP-NETIF SNTP module saves the statically configured server(s) and reconfigures them after obtaining a DHCP lease.

The typical configuration now looks as per below, providing the specific `IP_EVENT` to update the config and index of the first server to reconfigure (for example setting `config.index_of_first_server=1` would keep the DHCP provided server at index 0, and the statically configured server at index 1).

```
esp_sntp_config_t config = ESP_NETIF_SNTP_DEFAULT_CONFIG("pool.ntp.org");
config.start = false;                       // start the SNTP service explicitly (after
connecting)
config.server_from_dhcp = true;             // accept the NTP offers from DHCP server
config.renew_servers_after_new_IP = true;   // let esp-netif update the configured SNTP
server(s) after receiving the DHCP lease
config.index_of_first_server = 1;           // updates from server num 1, leaving server 0
(from DHCP) intact
config.ip_event_to_renew = IP_EVENT_STA_GOT_IP;  // IP event on which we refresh the
configuration
```

Then we start the service normally with `esp_netif_sntp_start()`.

ESP-NETIF Programmer's Manual

Please refer to the following example to understand the initialization process of the default interface:

- Wi-Fi Station: wifi/getting_started/station/main/station_example_main.c

- Ethernet: ethernet/basic/main/ethernet_example_main.c
- L2 TAP: protocols/l2tap/main/l2tap_main.c

- Wi-Fi Access Point: wifi/getting_started/softAP/main/softap_example_main.c

For more specific cases, please consult this guide: ESP-NETIF Custom I/O Driver.

Wi-Fi Default Initialization

The initialization code as well as registering event handlers for default interfaces, such as softAP and station, are provided in separate APIs to facilitate simple startup code for most applications:

- `esp_netif_create_default_wifi_sta()`
- `esp_netif_create_default_wifi_ap()`

Please note that these functions return the `esp_netif` handle, i.e., a pointer to a network interface object allocated and configured with default settings, as a consequence, which means that:

- The created object has to be destroyed if a network de-initialization is provided by an application using `esp_netif_destroy_default_wifi()`.
- These *default* interfaces must not be created multiple times unless the created handle is deleted using `esp_netif_destroy()`.
- When using Wi-Fi in `AP+STA` mode, both these interfaces have to be created.

API Reference

Header File

- components/esp_netif/include/esp_netif.h
- This header file can be included with:

```
#include "esp_netif.h"
```

- This header file is a part of the API provided by the `esp_netif` component. To declare that your component depends on `esp_netif`, add the following to your CMakeLists.txt:

```
REQUIRES esp_netif
```

or

```
PRIV_REQUIRES esp_netif
```

Functions

esp_err_t **esp_netif_init**(void)

Initialize the underlying TCP/IP stack.

Note

This function should be called exactly once from application code, when the application starts up.

Returns:

- ESP_OK on success
- ESP_FAIL if initializing failed

esp_err_t **esp_netif_deinit**(void)

Deinitialize the esp-netif component (and the underlying TCP/IP stack)

```
Note: Deinitialization is not supported yet
```

Returns:

- ESP_ERR_INVALID_STATE if esp_netif not initialized
- ESP_ERR_NOT_SUPPORTED otherwise

esp_netif_t *__esp_netif_new__(*const* esp_netif_config_t *esp_netif_config)

Creates an instance of new esp-netif object based on provided config.

Parameters: **esp_netif_config** -- **[in]** pointer esp-netif configuration

Returns:

- pointer to esp-netif object on success
- NULL otherwise

void **esp_netif_destroy**(esp_netif_t *esp_netif)

Destroys the esp_netif object.

Parameters: **esp_netif** -- **[in]** pointer to the object to be deleted

---

esp_err_t **esp_netif_set_driver_config**(esp_netif_t *esp_netif, *const* esp_netif_driver_ifconfig_t *driver_config)

Configures driver related options of esp_netif object.

Parameters:
- **esp_netif** -- **[inout]** pointer to the object to be configured
- **driver_config** -- **[in]** pointer esp-netif io driver related configuration

Returns:
- ESP_OK on success
- ESP_ERR_ESP_NETIF_INVALID_PARAMS if invalid parameters provided

---

esp_err_t **esp_netif_attach**(esp_netif_t *esp_netif, esp_netif_iodriver_handle driver_handle)

Attaches esp_netif instance to the io driver handle.

Calling this function enables connecting specific esp_netif object with already initialized io driver to update esp_netif object with driver specific configuration (i.e. calls post_attach callback, which typically sets io driver callbacks to esp_netif instance and starts the driver)

Parameters:
- **esp_netif** -- **[inout]** pointer to esp_netif object to be attached
- **driver_handle** -- **[in]** pointer to the driver handle

Returns:
- ESP_OK on success
- ESP_ERR_ESP_NETIF_DRIVER_ATTACH_FAILED if driver's pot_attach callback failed

---

esp_err_t **esp_netif_receive**(esp_netif_t *esp_netif, void *buffer, size_t len, void *eb)

Passes the raw packets from communication media to the appropriate TCP/IP stack.

This function is called from the configured (peripheral) driver layer. The data are then forwarded as frames to the TCP/IP stack.

Parameters:
- **esp_netif** -- **[in]** Handle to esp-netif instance
- **buffer** -- **[in]** Received data
- **len** -- **[in]** Length of the data frame
- **eb** -- **[in]** Pointer to internal buffer (used in Wi-Fi driver)

Returns:
- ESP_OK

---

void **esp_netif_action_start**(void *esp_netif, esp_event_base_t base, int32_t event_id, void *data)

Default building block for network interface action upon IO driver start event Creates network interface, if AUTOUP enabled turns the interface on, if DHCPS enabled starts dhcp server.

Note

This API can be directly used as event handler

Parameters:
- **esp_netif** -- **[in]** Handle to esp-netif instance
- **base** -- The base type of the event
- **event_id** -- The specific ID of the event
- **data** -- Optional data associated with the event

---

void **esp_netif_action_stop**(void *esp_netif, esp_event_base_t base, int32_t event_id, void *data)

Default building block for network interface action upon IO driver stop event.

Note

This API can be directly used as event handler

Parameters:
- **esp_netif** -- **[in]** Handle to esp-netif instance
- **base** -- The base type of the event
- **event_id** -- The specific ID of the event
- **data** -- Optional data associated with the event

---

void **esp_netif_action_connected**(void *esp_netif, esp_event_base_t base, int32_t event_id, void *data)

Default building block for network interface action upon IO driver connected event.

Note

This API can be directly used as event handler

Parameters:
- **esp_netif** -- **[in]** Handle to esp-netif instance
- **base** -- The base type of the event
- **event_id** -- The specific ID of the event
- **data** -- Optional data associated with the event

---

void **esp_netif_action_disconnected**(void *esp_netif, esp_event_base_t base, int32_t event_id, void *data)

Default building block for network interface action upon IO driver disconnected event.

Note

This API can be directly used as event handler

Parameters:
- **esp_netif** -- **[in]** Handle to esp-netif instance
- **base** -- The base type of the event
- **event_id** -- The specific ID of the event
- **data** -- Optional data associated with the event

---

void **esp_netif_action_got_ip**(void *esp_netif, esp_event_base_t base, int32_t event_id, void *data)

Default building block for network interface action upon network got IP event.

Note

This API can be directly used as event handler

Parameters:
- **esp_netif** -- **[in]** Handle to esp-netif instance
- **base** -- The base type of the event
- **event_id** -- The specific ID of the event
- **data** -- Optional data associated with the event

---

void **esp_netif_action_join_ip6_multicast_group**(void *esp_netif, esp_event_base_t base, int32_t event_id, void *data)

Default building block for network interface action upon IPv6 multicast group join.

Note

This API can be directly used as event handler

Parameters:
- **esp_netif** -- **[in]** Handle to esp-netif instance
- **base** -- The base type of the event
- **event_id** -- The specific ID of the event
- **data** -- Optional data associated with the event

---

void **esp_netif_action_leave_ip6_multicast_group**(void *esp_netif, esp_event_base_t base, int32_t event_id, void *data)

Default building block for network interface action upon IPv6 multicast group leave.

Note

This API can be directly used as event handler

Parameters:
- **esp_netif** -- **[in]** Handle to esp-netif instance
- **base** -- The base type of the event
- **event_id** -- The specific ID of the event
- **data** -- Optional data associated with the event

---

void **esp_netif_action_add_ip6_address**(void *esp_netif, esp_event_base_t base, int32_t event_id, void *data)

Default building block for network interface action upon IPv6 address added by the underlying stack.

Note

This API can be directly used as event handler

Parameters:
- **esp_netif** -- **[in]** Handle to esp-netif instance
- **base** -- The base type of the event
- **event_id** -- The specific ID of the event
- **data** -- Optional data associated with the event

---

void **esp_netif_action_remove_ip6_address**(void *esp_netif, esp_event_base_t base, int32_t event_id, void *data)

Default building block for network interface action upon IPv6 address removed by the underlying stack.

Note

This API can be directly used as event handler

Parameters:
- **esp_netif** -- **[in]** Handle to esp-netif instance
- **base** -- The base type of the event
- **event_id** -- The specific ID of the event
- **data** -- Optional data associated with the event

---

esp_err_t **esp_netif_set_default_netif**(esp_netif_t *esp_netif)

Manual configuration of the default netif.

This API overrides the automatic configuration of the default interface based on the route_prio If the selected netif is set default using this API, no other interface could be set-default disregarding its route_prio number (unless the selected netif gets destroyed)

Parameters: **esp_netif** -- **[in]** Handle to esp-netif instance

Returns: ESP_OK on success

esp_netif_t *esp_netif_get_default_netif(void)

Getter function of the default netif.

This API returns the selected default netif.

Returns: Handle to esp-netif instance of the default netif.

---

esp_err_t esp_netif_join_ip6_multicast_group(esp_netif_t *esp_netif, *const* esp_ip6_addr_t *addr)

Cause the TCP/IP stack to join a IPv6 multicast group.

Parameters:
- **esp_netif** -- **[in]** Handle to esp-netif instance
- **addr** -- **[in]** The multicast group to join

Returns:
- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
- ESP_ERR_ESP_NETIF_MLD6_FAILED
- ESP_ERR_NO_MEM

---

esp_err_t esp_netif_leave_ip6_multicast_group(esp_netif_t *esp_netif, *const* esp_ip6_addr_t *addr)

Cause the TCP/IP stack to leave a IPv6 multicast group.

Parameters:
- **esp_netif** -- **[in]** Handle to esp-netif instance
- **addr** -- **[in]** The multicast group to leave

Returns:
- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
- ESP_ERR_ESP_NETIF_MLD6_FAILED
- ESP_ERR_NO_MEM

---

esp_err_t esp_netif_set_mac(esp_netif_t *esp_netif, uint8_t mac[])

Set the mac address for the interface instance.

Parameters:
- **esp_netif** -- **[in]** Handle to esp-netif instance
- **mac** -- **[in]** Desired mac address for the related network interface

Returns:
- ESP_OK - success
- ESP_ERR_ESP_NETIF_IF_NOT_READY - interface status error
- ESP_ERR_NOT_SUPPORTED - mac not supported on this interface

---

esp_err_t esp_netif_get_mac(esp_netif_t *esp_netif, uint8_t mac[])

Get the mac address for the interface instance.

Parameters:
- **esp_netif** -- **[in]** Handle to esp-netif instance
- **mac** -- **[out]** Resultant mac address for the related network interface

Returns:
- ESP_OK - success
- ESP_ERR_ESP_NETIF_IF_NOT_READY - interface status error
- ESP_ERR_NOT_SUPPORTED - mac not supported on this interface

---

esp_err_t **esp_netif_set_hostname**(esp_netif_t *esp_netif, *const* char *hostname)

Set the hostname of an interface.

The configured hostname overrides the default configuration value CONFIG_LWIP_LOCAL_HOSTNAME. Please note that when the hostname is altered after interface started/connected the changes would only be reflected once the interface restarts/reconnects

Parameters:
- **esp_netif** -- **[in]** Handle to esp-netif instance
- **hostname** -- **[in]** New hostname for the interface. Maximum length 32 bytes.

Returns:
- ESP_OK - success
- ESP_ERR_ESP_NETIF_IF_NOT_READY - interface status error
- ESP_ERR_ESP_NETIF_INVALID_PARAMS - parameter error

---

esp_err_t **esp_netif_get_hostname**(esp_netif_t *esp_netif, *const* char **hostname)

Get interface hostname.

Parameters:
- **esp_netif** -- **[in]** Handle to esp-netif instance
- **hostname** -- **[out]** Returns a pointer to the hostname. May be NULL if no hostname is set. If set non-NULL, pointer remains valid (and string may change if the hostname changes).

Returns:
- ESP_OK - success
- ESP_ERR_ESP_NETIF_IF_NOT_READY - interface status error
- ESP_ERR_ESP_NETIF_INVALID_PARAMS - parameter error

---

bool **esp_netif_is_netif_up**(esp_netif_t *esp_netif)

Test if supplied interface is up or down.

Parameters: **esp_netif** -- **[in]** Handle to esp-netif instance

Returns:
- true - Interface is up
- false - Interface is down

esp_err_t **esp_netif_get_ip_info**(esp_netif_t *esp_netif, esp_netif_ip_info_t *ip_info)

Get interface's IP address information.

If the interface is up, IP information is read directly from the TCP/IP stack. If the interface is down, IP information is read from a copy kept in the ESP-NETIF instance

Parameters:
- **esp_netif** -- **[in]** Handle to esp-netif instance
- **ip_info** -- **[out]** If successful, IP information will be returned in this argument.

Returns:
- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS

esp_err_t **esp_netif_get_old_ip_info**(esp_netif_t *esp_netif, esp_netif_ip_info_t *ip_info)

Get interface's old IP information.

Returns an "old" IP address previously stored for the interface when the valid IP changed.

If the IP lost timer has expired (meaning the interface was down for longer than the configured interval) then the old IP information will be zero.

Parameters:
- **esp_netif** -- **[in]** Handle to esp-netif instance
- **ip_info** -- **[out]** If successful, IP information will be returned in this argument.

Returns:
- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS

esp_err_t **esp_netif_set_ip_info**(esp_netif_t *esp_netif, *const* esp_netif_ip_info_t *ip_info)

Set interface's IP address information.

This function is mainly used to set a static IP on an interface.

If the interface is up, the new IP information is set directly in the TCP/IP stack.

The copy of IP information kept in the ESP-NETIF instance is also updated (this copy is returned if the IP is queried while the interface is still down.)

Note

DHCP client/server must be stopped (if enabled for this interface) before setting new IP information.

Note

Calling this interface for may generate a SYSTEM_EVENT_STA_GOT_IP or SYSTEM_EVENT_ETH_GOT_IP event.

Parameters:
- **esp_netif** -- **[in]** Handle to esp-netif instance
- **ip_info** -- **[in]** IP information to set on the specified interface

Returns:
- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
- ESP_ERR_ESP_NETIF_DHCP_NOT_STOPPED If DHCP server or client is still running

---

esp_err_t **esp_netif_set_old_ip_info**(esp_netif_t *esp_netif, *const* esp_netif_ip_info_t *ip_info)

Set interface old IP information.

This function is called from the DHCP client (if enabled), before a new IP is set. It is also called from the default handlers for the SYSTEM_EVENT_STA_CONNECTED and SYSTEM_EVENT_ETH_CONNECTED events.

Calling this function stores the previously configured IP, which can be used to determine if the IP changes in the future.

If the interface is disconnected or down for too long, the "IP lost timer" will expire (after the configured interval) and set the old IP information to zero.

Parameters:
- **esp_netif** -- **[in]** Handle to esp-netif instance
- **ip_info** -- **[in]** Store the old IP information for the specified interface

Returns:
- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS

---

int **esp_netif_get_netif_impl_index**(esp_netif_t *esp_netif)

Get net interface index from network stack implementation.

Note

This index could be used in `setsockopt()` to bind socket with multicast interface

Parameters: **esp_netif** -- **[in]** Handle to esp-netif instance

Returns: implementation specific index of interface represented with supplied esp_netif

---

esp_err_t **esp_netif_get_netif_impl_name**(esp_netif_t *esp_netif, char *name)

Get net interface name from network stack implementation.

Note

This name could be used in `setsockopt()` to bind socket with appropriate interface

Parameters:
- **esp_netif** -- **[in]** Handle to esp-netif instance
- **name** -- **[out]** Interface name as specified in underlying TCP/IP stack. Note that the actual name will be copied to the specified buffer, which must be allocated to hold maximum interface name size (6 characters for lwIP)

Returns:
- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS

---

esp_err_t **esp_netif_napt_enable**(esp_netif_t *esp_netif)

Enable NAPT on an interface.

Note

Enable operation can be performed only on one interface at a time. NAPT cannot be enabled on multiple interfaces according to this implementation.

Parameters: **esp_netif** -- **[in]** Handle to esp-netif instance

Returns:
- ESP_OK
- ESP_FAIL
- ESP_ERR_NOT_SUPPORTED

---

esp_err_t **esp_netif_napt_disable**(esp_netif_t *esp_netif)

Disable NAPT on an interface.

Parameters: **esp_netif** -- **[in]** Handle to esp-netif instance

Returns:
- ESP_OK
- ESP_FAIL
- ESP_ERR_NOT_SUPPORTED

---

esp_err_t **esp_netif_dhcps_option**(esp_netif_t *esp_netif, esp_netif_dhcp_option_mode_t opt_op, esp_netif_dhcp_option_id_t opt_id, void *opt_val, uint32_t opt_len)

Set or Get DHCP server option.

Parameters:
- **esp_netif** -- **[in]** Handle to esp-netif instance

- **opt_op** -- **[in]** ESP_NETIF_OP_SET to set an option, ESP_NETIF_OP_GET to get an option.
- **opt_id** -- **[in]** Option index to get or set, must be one of the supported enum values.
- **opt_val** -- **[inout]** Pointer to the option parameter.
- **opt_len** -- **[in]** Length of the option parameter.

Returns:

- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
- ESP_ERR_ESP_NETIF_DHCP_ALREADY_STOPPED
- ESP_ERR_ESP_NETIF_DHCP_ALREADY_STARTED

---

esp_err_t **esp_netif_dhcpc_option**(esp_netif_t *esp_netif, esp_netif_dhcp_option_mode_t opt_op, esp_netif_dhcp_option_id_t opt_id, void *opt_val, uint32_t opt_len)

Set or Get DHCP client option.

Parameters:

- **esp_netif** -- **[in]** Handle to esp-netif instance
- **opt_op** -- **[in]** ESP_NETIF_OP_SET to set an option, ESP_NETIF_OP_GET to get an option.
- **opt_id** -- **[in]** Option index to get or set, must be one of the supported enum values.
- **opt_val** -- **[inout]** Pointer to the option parameter.
- **opt_len** -- **[in]** Length of the option parameter.

Returns:

- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
- ESP_ERR_ESP_NETIF_DHCP_ALREADY_STOPPED
- ESP_ERR_ESP_NETIF_DHCP_ALREADY_STARTED

---

esp_err_t **esp_netif_dhcpc_start**(esp_netif_t *esp_netif)

Start DHCP client (only if enabled in interface object)

Note

The default event handlers for the SYSTEM_EVENT_STA_CONNECTED and SYSTEM_EVENT_ETH_CONNECTED events call this function.

Parameters: **esp_netif** -- **[in]** Handle to esp-netif instance

Returns:

- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
- ESP_ERR_ESP_NETIF_DHCP_ALREADY_STARTED
- ESP_ERR_ESP_NETIF_DHCPC_START_FAILED

esp_err_t **esp_netif_dhcpc_stop**(esp_netif_t *esp_netif)

Stop DHCP client (only if enabled in interface object)

Note

Calling action_netif_stop() will also stop the DHCP Client if it is running.

Parameters: **esp_netif** -- **[in]** Handle to esp-netif instance

Returns:

- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
- ESP_ERR_ESP_NETIF_DHCP_ALREADY_STOPPED
- ESP_ERR_ESP_NETIF_IF_NOT_READY

esp_err_t **esp_netif_dhcpc_get_status**(esp_netif_t *esp_netif, esp_netif_dhcp_status_t *status)

Get DHCP client status.

Parameters:

- **esp_netif** -- **[in]** Handle to esp-netif instance
- **status** -- **[out]** If successful, the status of DHCP client will be returned in this argument.

Returns:

- ESP_OK

esp_err_t **esp_netif_dhcps_get_status**(esp_netif_t *esp_netif, esp_netif_dhcp_status_t *status)

Get DHCP Server status.

Parameters:

- **esp_netif** -- **[in]** Handle to esp-netif instance
- **status** -- **[out]** If successful, the status of the DHCP server will be returned in this argument.

Returns:

- ESP_OK

esp_err_t **esp_netif_dhcps_start**(esp_netif_t *esp_netif)

Start DHCP server (only if enabled in interface object)

Parameters: **esp_netif** -- **[in]** Handle to esp-netif instance

Returns:

- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
- ESP_ERR_ESP_NETIF_DHCP_ALREADY_STARTED

esp_err_t **esp_netif_dhcps_stop**(esp_netif_t *esp_netif)

Stop DHCP server (only if enabled in interface object)

Parameters: **esp_netif** -- **[in]** Handle to esp-netif instance

Returns:

- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
- ESP_ERR_ESP_NETIF_DHCP_ALREADY_STOPPED
- ESP_ERR_ESP_NETIF_IF_NOT_READY

esp_err_t **esp_netif_dhcps_get_clients_by_mac**(esp_netif_t *esp_netif, int num, esp_netif_pair_mac_ip_t *mac_ip_pair)

Populate IP addresses of clients connected to DHCP server listed by their MAC addresses.

Parameters:

- **esp_netif** -- **[in]** Handle to esp-netif instance
- **num** -- **[in]** Number of clients with specified MAC addresses in the array of pairs
- **mac_ip_pair** -- **[inout]** Array of pairs of MAC and IP addresses (MAC are inputs, IP outputs)

Returns:

- ESP_OK on success
- ESP_ERR_ESP_NETIF_INVALID_PARAMS on invalid params
- ESP_ERR_NOT_SUPPORTED if DHCP server not enabled

esp_err_t **esp_netif_set_dns_info**(esp_netif_t *esp_netif, esp_netif_dns_type_t type, esp_netif_dns_info_t *dns)

Set DNS Server information.

This function behaves differently if DHCP server or client is enabled

If DHCP client is enabled, main and backup DNS servers will be updated automatically from the DHCP lease if the relevant DHCP options are set. Fallback DNS Server is never updated from the DHCP lease and is designed to be set via this API. If DHCP client is disabled, all DNS server types can be set via this API only.

Note that LWIP stores DNS server information globally, not per interface, so the first parameter is unused in the default LWIP configuration. If CONFIG_ESP_NETIF_SET_DNS_PER_DEFAULT_NETIF=1 this API sets internal DNS server information per netif. It's also possible to set the global DNS server info by supplying esp_netif=NULL

If DHCP server is enabled, the Main DNS Server setting is used by the DHCP server to provide a DNS Server option to DHCP clients (Wi-Fi stations).

- The default Main DNS server is typically the IP of the DHCP server itself.

- This function can override it by setting server type ESP_NETIF_DNS_MAIN.
- Other DNS Server types are not supported for the DHCP server.
- To propagate the DNS info to client, please stop the DHCP server before using this API.

Parameters:
- **esp_netif** -- **[in]** Handle to esp-netif instance
- **type** -- **[in]** Type of DNS Server to set: ESP_NETIF_DNS_MAIN, ESP_NETIF_DNS_BACKUP, ESP_NETIF_DNS_FALLBACK
- **dns** -- **[in]** DNS Server address to set

Returns:
- ESP_OK on success
- ESP_ERR_ESP_NETIF_INVALID_PARAMS invalid params

---

esp_err_t **esp_netif_get_dns_info**(esp_netif_t *esp_netif, esp_netif_dns_type_t type, esp_netif_dns_info_t *dns)

Get DNS Server information.

Return the currently configured DNS Server address for the specified interface and Server type.

This may be result of a previous call to esp_netif_set_dns_info(). If the interface's DHCP client is enabled, the Main or Backup DNS Server may be set by the current DHCP lease.

Note that LWIP stores DNS server information globally, not per interface, so the first parameter is unused in the default LWIP configuration. If CONFIG_ESP_NETIF_SET_DNS_PER_DEFAULT_NETIF=1 this API returns internally saved DNS server information per netif. It's also possible to ask for the global DNS server info by supplying esp_netif=NULL

Parameters:
- **esp_netif** -- **[in]** Handle to esp-netif instance
- **type** -- **[in]** Type of DNS Server to get: ESP_NETIF_DNS_MAIN, ESP_NETIF_DNS_BACKUP, ESP_NETIF_DNS_FALLBACK
- **dns** -- **[out]** DNS Server result is written here on success

Returns:
- ESP_OK on success
- ESP_ERR_ESP_NETIF_INVALID_PARAMS invalid params

---

esp_err_t **esp_netif_create_ip6_linklocal**(esp_netif_t *esp_netif)

Create interface link-local IPv6 address.

Cause the TCP/IP stack to create a link-local IPv6 address for the specified interface.

This function also registers a callback for the specified interface, so that if the link-local address becomes verified as the preferred address then a SYSTEM_EVENT_GOT_IP6 event will be sent.

Parameters: **esp_netif** -- **[in]** Handle to esp-netif instance

Returns:

- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS

---

esp_err_t **esp_netif_get_ip6_linklocal**(esp_netif_t *esp_netif, esp_ip6_addr_t *if_ip6)

Get interface link-local IPv6 address.

If the specified interface is up and a preferred link-local IPv6 address has been created for the interface, return a copy of it.

Parameters:

- **esp_netif** -- **[in]** Handle to esp-netif instance
- **if_ip6** -- **[out]** IPv6 information will be returned in this argument if successful.

Returns:

- ESP_OK
- ESP_FAIL If interface is down, does not have a link-local IPv6 address, or the link-local IPv6 address is not a preferred address.

---

esp_err_t **esp_netif_get_ip6_global**(esp_netif_t *esp_netif, esp_ip6_addr_t *if_ip6)

Get interface global IPv6 address.

If the specified interface is up and a preferred global IPv6 address has been created for the interface, return a copy of it.

Parameters:

- **esp_netif** -- **[in]** Handle to esp-netif instance
- **if_ip6** -- **[out]** IPv6 information will be returned in this argument if successful.

Returns:

- ESP_OK
- ESP_FAIL If interface is down, does not have a global IPv6 address, or the global IPv6 address is not a preferred address.

---

int **esp_netif_get_all_ip6**(esp_netif_t *esp_netif, esp_ip6_addr_t if_ip6[])

Get all IPv6 addresses of the specified interface.

Parameters:

- **esp_netif** -- **[in]** Handle to esp-netif instance
- **if_ip6** -- **[out]** Array of IPv6 addresses will be copied to the argument

Returns: number of returned IPv6 addresses

---

int **esp_netif_get_all_preferred_ip6**(esp_netif_t *esp_netif, esp_ip6_addr_t if_ip6[])

Get all preferred IPv6 addresses of the specified interface.

Parameters:
- **esp_netif** -- **[in]** Handle to esp-netif instance
- **if_ip6** -- **[out]** Array of IPv6 addresses will be copied to the argument

Returns: number of returned IPv6 addresses

---

esp_err_t **esp_netif_add_ip6_address**(esp_netif_t *esp_netif, *const* esp_ip6_addr_t addr, bool preferred)

Cause the TCP/IP stack to add an IPv6 address to the interface.

Parameters:
- **esp_netif** -- **[in]** Handle to esp-netif instance
- **addr** -- **[in]** The address to be added
- **preferred** -- **[in]** The preferred status of the address

Returns:
- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
- ESP_ERR_ESP_NETIF_IP6_ADDR_FAILED
- ESP_ERR_NO_MEM

---

esp_err_t **esp_netif_remove_ip6_address**(esp_netif_t *esp_netif, *const* esp_ip6_addr_t *addr)

Cause the TCP/IP stack to remove an IPv6 address from the interface.

Parameters:
- **esp_netif** -- **[in]** Handle to esp-netif instance
- **addr** -- **[in]** The address to be removed

Returns:
- ESP_OK
- ESP_ERR_ESP_NETIF_INVALID_PARAMS
- ESP_ERR_ESP_NETIF_IP6_ADDR_FAILED
- ESP_ERR_NO_MEM

---

void **esp_netif_set_ip4_addr**(esp_ip4_addr_t *addr, uint8_t a, uint8_t b, uint8_t c, uint8_t d)

Sets IPv4 address to the specified octets.

Parameters:
- **addr** -- **[out]** IP address to be set
- **a** -- the first octet (127 for IP 127.0.0.1)
- **b** --
- **c** --
- **d** --

---

char ***esp_ip4addr_ntoa**(*const* esp_ip4_addr_t *addr, char *buf, int buflen)

Converts numeric IP address into decimal dotted ASCII representation.

Parameters:
- **addr** -- ip address in network order to convert

- **buf** -- target buffer where the string is stored
- **buflen** -- length of buf

Returns: either pointer to buf which now holds the ASCII representation of addr or NULL if buf was too small

---

uint32_t **esp_ip4addr_aton**(*const* char *addr)

Ascii internet address interpretation routine The value returned is in network order.

Parameters: **addr** -- IP address in ascii representation (e.g. "127.0.0.1")

Returns: ip address in network order

---

esp_err_t **esp_netif_str_to_ip4**(*const* char *src, esp_ip4_addr_t *dst)

Converts Ascii internet IPv4 address into esp_ip4_addr_t.

Parameters:
- **src** -- **[in]** IPv4 address in ascii representation (e.g. "127.0.0.1")
- **dst** -- **[out]** Address of the target esp_ip4_addr_t structure to receive converted address

Returns:
- ESP_OK on success
- ESP_FAIL if conversion failed
- ESP_ERR_INVALID_ARG if invalid parameter is passed into

---

esp_err_t **esp_netif_str_to_ip6**(*const* char *src, esp_ip6_addr_t *dst)

Converts Ascii internet IPv6 address into esp_ip4_addr_t Zeros in the IP address can be stripped or completely ommited: "2001:db8:85a3:0:0:0:2:1" or "2001:db8::2:1")

Parameters:
- **src** -- **[in]** IPv6 address in ascii representation (e.g. ""2001:0db8:85a3:0000:0000:0000:0002:0001")
- **dst** -- **[out]** Address of the target esp_ip6_addr_t structure to receive converted address

Returns:
- ESP_OK on success
- ESP_FAIL if conversion failed
- ESP_ERR_INVALID_ARG if invalid parameter is passed into

---

esp_netif_iodriver_handle **esp_netif_get_io_driver**(esp_netif_t *esp_netif)

Gets media driver handle for this esp-netif instance.

Parameters: **esp_netif** -- **[in]** Handle to esp-netif instance

Returns: opaque pointer of related IO driver

esp_netif_t *esp_netif_get_handle_from_ifkey(const char *if_key)

Searches over a list of created objects to find an instance with supplied if key.

Parameters: **if_key** -- Textual description of network interface

Returns: Handle to esp-netif instance

esp_netif_flags_t esp_netif_get_flags(esp_netif_t *esp_netif)

Returns configured flags for this interface.

Parameters: **esp_netif** -- **[in]** Handle to esp-netif instance

Returns: Configuration flags

const char *esp_netif_get_ifkey(esp_netif_t *esp_netif)

Returns configured interface key for this esp-netif instance.

Parameters: **esp_netif** -- **[in]** Handle to esp-netif instance

Returns: Textual description of related interface

const char *esp_netif_get_desc(esp_netif_t *esp_netif)

Returns configured interface type for this esp-netif instance.

Parameters: **esp_netif** -- **[in]** Handle to esp-netif instance

Returns: Enumerated type of this interface, such as station, AP, ethernet

int esp_netif_get_route_prio(esp_netif_t *esp_netif)

Returns configured routing priority number.

Parameters: **esp_netif** -- **[in]** Handle to esp-netif instance

Returns: Integer representing the instance's route-prio, or -1 if invalid paramters

int32_t esp_netif_get_event_id(esp_netif_t *esp_netif, esp_netif_ip_event_type_t event_type)

Returns configured event for this esp-netif instance and supplied event type.

Parameters:

- **esp_netif** -- **[in]** Handle to esp-netif instance
- **event_type** -- (either get or lost IP)

Returns: specific event id which is configured to be raised if the interface lost or acquired IP address -1 if supplied event_type is not known

esp_netif_t *esp_netif_next(esp_netif_t *esp_netif)

Iterates over list of interfaces. Returns first netif if NULL given as parameter.

Note

This API doesn't lock the list, nor the TCPIP context, as this it's usually required to get atomic access between iteration steps rather that within a single iteration. Therefore it is recommended to iterate over the interfaces inside esp_netif_tcpip_exec()

Note

This API is deprecated. Please use esp_netif_next_unsafe() directly if all the system interfaces are under your control and you can safely iterate over them. Otherwise, iterate over interfaces using esp_netif_tcpip_exec(), or use esp_netif_find_if() to search in the list of netifs with defined predicate.

Parameters: **esp_netif** -- **[in]** Handle to esp-netif instance

Returns: First netif from the list if supplied parameter is NULL, next one otherwise

esp_netif_t *esp_netif_next_unsafe(esp_netif_t *esp_netif)

Iterates over list of interfaces without list locking. Returns first netif if NULL given as parameter.

Used for bulk search loops within TCPIP context, e.g. using esp_netif_tcpip_exec(), or if we're sure that the iteration is safe from our application perspective (e.g. no interface is removed between iterations)

Parameters: **esp_netif** -- **[in]** Handle to esp-netif instance

Returns: First netif from the list if supplied parameter is NULL, next one otherwise

esp_netif_t *esp_netif_find_if(esp_netif_find_predicate_t fn, void *ctx)

Return a netif pointer for the first interface that meets criteria defined by the callback.

Parameters:
- **fn** -- Predicate function returning true for the desired interface
- **ctx** -- Context pointer passed to the predicate, typically a descriptor to compare with

Returns: valid netif pointer if found, NULL if not

size_t esp_netif_get_nr_of_ifs(void)

Returns number of registered esp_netif objects.

Returns: Number of esp_netifs

---

void **esp_netif_netstack_buf_ref**(void *netstack_buf)

increase the reference counter of net stack buffer

Parameters: **netstack_buf** -- **[in]** the net stack buffer

---

void **esp_netif_netstack_buf_free**(void *netstack_buf)

free the netstack buffer

Parameters: **netstack_buf** -- **[in]** the net stack buffer

---

esp_err_t **esp_netif_tcpip_exec**(esp_netif_callback_fn fn, void *ctx)

Utility to execute the supplied callback in TCP/IP context.

Parameters:
- **fn** -- Pointer to the callback
- **ctx** -- Parameter to the callback

Returns: The error code (esp_err_t) returned by the callback

Type Definitions

---

typedef bool (***esp_netif_find_predicate_t**)(esp_netif_t *netif, void *ctx)

Predicate callback for esp_netif_find_if() used to find interface which meets defined criteria.

---

typedef esp_err_t (***esp_netif_callback_fn**)(void *ctx)

TCPIP thread safe callback used with esp_netif_tcpip_exec()

Header File

- components/esp_netif/include/esp_netif_sntp.h
- This header file can be included with:

```
#include "esp_netif_sntp.h"
```

- This header file is a part of the API provided by the `esp_netif` component. To declare that your component depends on `esp_netif`, add the following to your CMakeLists.txt:

```
REQUIRES esp_netif
```

or

```
PRIV_REQUIRES esp_netif
```

Functions

esp_err_t **esp_netif_sntp_init**(*const* esp_sntp_config_t *config)

Initialize SNTP with supplied config struct.

Parameters: **config** -- Config struct

Returns: ESP_OK on success

esp_err_t **esp_netif_sntp_start**(void)

Start SNTP service if it wasn't started during init (config.start = false) or restart it if already started.

Returns: ESP_OK on success

void **esp_netif_sntp_deinit**(void)

Deinitialize esp_netif SNTP module.

esp_err_t **esp_netif_sntp_sync_wait**(TickType_t tout)

Wait for time sync event.

Parameters: **tout** -- Specified timeout in RTOS ticks

Returns: ESP_TIMEOUT if sync event didn't came withing the timeout ESP_ERR_NOT_FINISHED if the sync event came, but we're in smooth update mode and still in progress (SNTP_SYNC_STATUS_IN_PROGRESS) ESP_OK if time sync'ed

esp_err_t **esp_netif_sntp_reachability**(unsigned int index, unsigned int *reachability)

Returns SNTP server's reachability shift register as described in RFC 5905.

Parameters:

- **index** -- Index of the SERVER
- **reachability** -- reachability shift register

Returns: ESP_OK on success, ESP_ERR_INVALID_STATE if SNTP not initialized ESP_ERR_INVALID_ARG if invalid arguments

Structures

struct **esp_sntp_config**

SNTP configuration struct.

Public Members

bool **smooth_sync**

set to true if smooth sync required

bool **server_from_dhcp**

set to true to request NTP server config from DHCP

bool **wait_for_sync**

if true, we create a semaphore to signal time sync event

bool **start**

set to true to automatically start the SNTP service

esp_sntp_time_cb_t **sync_cb**

optionally sets callback function on time sync event

bool **renew_servers_after_new_IP**

this is used to refresh server list if NTP provided by DHCP (which cleans other pre-configured servers)

ip_event_t **ip_event_to_renew**

set the IP event id on which we refresh server list (if renew_servers_after_new_IP=true)

size_t **index_of_first_server**

refresh server list after this server (if renew_servers_after_new_IP=true)

size_t **num_of_servers**

number of preconfigured NTP servers

const char ***servers**[1]

list of servers

Macros

ESP_SNTP_SERVER_LIST(...)

Utility macro for providing multiple servers in parentheses.

ESP_NETIF_SNTP_DEFAULT_CONFIG_MULTIPLE(servers_in_list, list_of_servers)

Default configuration to init SNTP with multiple servers.

Parameters:
- **servers_in_list** -- Number of servers in the list
- **list_of_servers** -- List of servers (use ESP_SNTP_SERVER_LIST(...))

ESP_NETIF_SNTP_DEFAULT_CONFIG(server)

Default configuration with a single server.

Type Definitions

typedef void (*esp_sntp_time_cb_t)(*struct* timeval *tv)

Time sync notification function.

typedef struct esp_sntp_config esp_sntp_config_t

SNTP configuration struct.

Header File

- components/esp_netif/include/esp_netif_types.h
- This header file can be included with:

```
#include "esp_netif_types.h"
```

- This header file is a part of the API provided by the `esp_netif` component. To declare that your component depends on `esp_netif`, add the following to your CMakeLists.txt:

```
REQUIRES esp_netif
```

or

```
PRIV_REQUIRES esp_netif
```

Structures

struct `esp_netif_dns_info_t`

DNS server info.

Public Members

esp_ip_addr_t **ip**

IPV4 address of DNS server

struct `esp_netif_ip_info_t`

Event structure for IP_EVENT_STA_GOT_IP, IP_EVENT_ETH_GOT_IP events

Public Members

esp_ip4_addr_t **ip**

Interface IPV4 address

esp_ip4_addr_t **netmask**

Interface IPV4 netmask

esp_ip4_addr_t **gw**

Interface IPV4 gateway address

struct `esp_netif_ip6_info_t`

IPV6 IP address information.

Public Members

esp_ip6_addr_t **ip**

Interface IPV6 address

struct `ip_event_got_ip_t`

Event structure for IP_EVENT_GOT_IP event.

Public Members

esp_netif_t ***esp_netif**

Pointer to corresponding esp-netif object

esp_netif_ip_info_t **ip_info**

IP address, netmask, gatway IP address

bool **ip_changed**

Whether the assigned IP has changed or not

---

struct **ip_event_got_ip6_t**

Event structure for IP_EVENT_GOT_IP6 event

Public Members

esp_netif_t ***esp_netif**

Pointer to corresponding esp-netif object

esp_netif_ip6_info_t **ip6_info**

IPv6 address of the interface

int **ip_index**

IPv6 address index

---

struct **ip_event_add_ip6_t**

Event structure for ADD_IP6 event

Public Members

esp_ip6_addr_t **addr**

The address to be added to the interface

bool **preferred**

The default preference of the address

---

struct **ip_event_ap_staipassigned_t**

Event structure for IP_EVENT_AP_STAIPASSIGNED event

Public Members

esp_netif_t ***esp_netif**

Pointer to the associated netif handle

esp_ip4_addr_t **ip**

IP address which was assigned to the station

uint8_t **mac**[6]

MAC address of the connected client

struct `bridgeif_config`

LwIP bridge configuration

Public Members

uint16_t **`max_fdb_dyn_entries`**

maximum number of entries in dynamic forwarding database

uint16_t **`max_fdb_sta_entries`**

maximum number of entries in static forwarding database

uint8_t **`max_ports`**

maximum number of ports the bridge can consist of

struct `esp_netif_inherent_config`

ESP-netif inherent config parameters.

Public Members

esp_netif_flags_t **`flags`**

flags that define esp-netif behavior

uint8_t **`mac`**[6]

initial mac address for this interface

const esp_netif_ip_info_t ***`ip_info`**

initial ip address for this interface

uint32_t **`get_ip_event`**

event id to be raised when interface gets an IP

uint32_t **`lost_ip_event`**

event id to be raised when interface losts its IP

const char ***`if_key`**

string identifier of the interface

const char ***`if_desc`**

textual description of the interface

int **route_prio**

numeric priority of this interface to become a default routing if (if other netifs are up). A higher value of route_prio indicates a higher priority

bridgeif_config_t ***bridge_info**

LwIP bridge configuration

---

struct **esp_netif_driver_base_s**

ESP-netif driver base handle.

Public Members

esp_err_t (***post_attach**)(esp_netif_t *netif, esp_netif_iodriver_handle h)

post attach function pointer

esp_netif_t ***netif**

netif handle

---

struct **esp_netif_driver_ifconfig**

Specific IO driver configuration.

Public Members

esp_netif_iodriver_handle **handle**

io-driver handle

esp_err_t (***transmit**)(void *h, void *buffer, size_t len)

transmit function pointer

esp_err_t (***transmit_wrap**)(void *h, void *buffer, size_t len, void *netstack_buffer)

transmit wrap function pointer

void (***driver_free_rx_buffer**)(void *h, void *buffer)

free rx buffer function pointer

---

struct **esp_netif_config**

Generic esp_netif configuration.

Public Members

const esp_netif_inherent_config_t ***base**

base config

const esp_netif_driver_ifconfig_t *driver

driver config

const esp_netif_netstack_config_t *stack

stack config

struct esp_netif_pair_mac_ip_t

DHCP client's addr info (pair of MAC and IP address)

Public Members

uint8_t mac[6]

Clients MAC address

esp_ip4_addr_t ip

Clients IP address

Macros

ESP_ERR_ESP_NETIF_BASE

Definition of ESP-NETIF based errors.

ESP_ERR_ESP_NETIF_INVALID_PARAMS

ESP_ERR_ESP_NETIF_IF_NOT_READY

ESP_ERR_ESP_NETIF_DHCPC_START_FAILED

ESP_ERR_ESP_NETIF_DHCP_ALREADY_STARTED

ESP_ERR_ESP_NETIF_DHCP_ALREADY_STOPPED

ESP_ERR_ESP_NETIF_NO_MEM

ESP_ERR_ESP_NETIF_DHCP_NOT_STOPPED

ESP_ERR_ESP_NETIF_DRIVER_ATTACH_FAILED

ESP_ERR_ESP_NETIF_INIT_FAILED

ESP_ERR_ESP_NETIF_DNS_NOT_CONFIGURED

ESP_ERR_ESP_NETIF_MLD6_FAILED

ESP_ERR_ESP_NETIF_IP6_ADDR_FAILED

ESP_ERR_ESP_NETIF_DHCPS_START_FAILED

ESP_NETIF_BR_FLOOD

Definition of ESP-NETIF bridge controll.

ESP_NETIF_BR_DROP

ESP_NETIF_BR_FDW_CPU

Type Definitions

typedef struct esp_netif_obj **esp_netif_t**

typedef enum esp_netif_flags **esp_netif_flags_t**

typedef enum esp_netif_ip_event_type **esp_netif_ip_event_type_t**

typedef struct bridgeif_config **bridgeif_config_t**

LwIP bridge configuration

typedef struct esp_netif_inherent_config **esp_netif_inherent_config_t**

ESP-netif inherent config parameters.

typedef struct esp_netif_config **esp_netif_config_t**

typedef void ***esp_netif_iodriver_handle**

IO driver handle type.

typedef struct esp_netif_driver_base_s **esp_netif_driver_base_t**

ESP-netif driver base handle.

typedef struct esp_netif_driver_ifconfig **esp_netif_driver_ifconfig_t**

typedef struct esp_netif_netstack_config **esp_netif_netstack_config_t**

Specific L3 network stack configuration.

typedef esp_err_t (***esp_netif_receive_t**)(esp_netif_t *esp_netif, void *buffer, size_t len, void *eb)

ESP-NETIF Receive function type.

Enumerations

enum **esp_netif_dns_type_t**

Type of DNS server.

Values:

enumerator **ESP_NETIF_DNS_MAIN**

DNS main server address

enumerator **ESP_NETIF_DNS_BACKUP**

DNS backup server address (Wi-Fi STA and Ethernet only)

enumerator **ESP_NETIF_DNS_FALLBACK**

DNS fallback server address (Wi-Fi STA and Ethernet only)

enumerator **ESP_NETIF_DNS_MAX**

enum **esp_netif_dhcp_status_t**

Status of DHCP client or DHCP server.

Values:

enumerator **ESP_NETIF_DHCP_INIT**

DHCP client/server is in initial state (not yet started)

enumerator **ESP_NETIF_DHCP_STARTED**

DHCP client/server has been started

enumerator **ESP_NETIF_DHCP_STOPPED**

DHCP client/server has been stopped

enumerator **ESP_NETIF_DHCP_STATUS_MAX**

enum **esp_netif_dhcp_option_mode_t**

Mode for DHCP client or DHCP server option functions.

Values:

enumerator **ESP_NETIF_OP_START**

enumerator **ESP_NETIF_OP_SET**

Set option

enumerator **ESP_NETIF_OP_GET**

Get option

enumerator **ESP_NETIF_OP_MAX**

enum **esp_netif_dhcp_option_id_t**

Supported options for DHCP client or DHCP server.

Values:

enumerator **ESP_NETIF_SUBNET_MASK**

Network mask

enumerator **ESP_NETIF_DOMAIN_NAME_SERVER**

Domain name server

enumerator **ESP_NETIF_ROUTER_SOLICITATION_ADDRESS**

Solicitation router address

enumerator **ESP_NETIF_REQUESTED_IP_ADDRESS**

Request specific IP address

enumerator **ESP_NETIF_IP_ADDRESS_LEASE_TIME**

Request IP address lease time

enumerator **ESP_NETIF_IP_REQUEST_RETRY_TIME**

Request IP address retry counter

enumerator **ESP_NETIF_VENDOR_CLASS_IDENTIFIER**

Vendor Class Identifier of a DHCP client

enumerator **ESP_NETIF_VENDOR_SPECIFIC_INFO**

Vendor Specific Information of a DHCP server

enum **ip_event_t**

IP event declarations

Values:

enumerator **IP_EVENT_STA_GOT_IP**

station got IP from connected AP

enumerator **IP_EVENT_STA_LOST_IP**

station lost IP and the IP is reset to 0

enumerator **IP_EVENT_AP_STAIPASSIGNED**

soft-AP assign an IP to a connected station

enumerator **IP_EVENT_GOT_IP6**

station or ap or ethernet interface v6IP addr is preferred

enumerator **IP_EVENT_ETH_GOT_IP**

ethernet got IP from connected AP

enumerator **IP_EVENT_ETH_LOST_IP**

ethernet lost IP and the IP is reset to 0

enumerator **IP_EVENT_PPP_GOT_IP**

PPP interface got IP

enumerator **IP_EVENT_PPP_LOST_IP**

PPP interface lost IP

enum **esp_netif_flags**

Values:

enumerator **ESP_NETIF_DHCP_CLIENT**

enumerator **ESP_NETIF_DHCP_SERVER**

enumerator **ESP_NETIF_FLAG_AUTOUP**

enumerator **ESP_NETIF_FLAG_GARP**

enumerator **ESP_NETIF_FLAG_EVENT_IP_MODIFIED**

enumerator **ESP_NETIF_FLAG_IS_PPP**

enumerator **ESP_NETIF_FLAG_IS_BRIDGE**

enumerator **ESP_NETIF_FLAG_MLDV6_REPORT**

enum **esp_netif_ip_event_type**

Values:

enumerator **ESP_NETIF_IP_EVENT_GOT_IP**

enumerator **ESP_NETIF_IP_EVENT_LOST_IP**

Header File

- components/esp_netif/include/esp_netif_ip_addr.h
- This header file can be included with:

```
#include "esp_netif_ip_addr.h"
```

- This header file is a part of the API provided by the `esp_netif` component. To declare that your component depends on `esp_netif`, add the following to your CMakeLists.txt:

```
REQUIRES esp_netif
```

or

```
PRIV_REQUIRES esp_netif
```

Functions

esp_ip6_addr_type_t **esp_netif_ip6_get_addr_type**(esp_ip6_addr_t *ip6_addr)

Get the IPv6 address type.

Parameters: **ip6_addr** -- **[in]** IPv6 type

Returns: IPv6 type in form of enum esp_ip6_addr_type_t

---

static inline void **esp_netif_ip_addr_copy**(esp_ip_addr_t *dest, *const* esp_ip_addr_t *src)

Copy IP addresses.

Parameters:
- **dest** -- **[out]** destination IP
- **src** -- **[in]** source IP

Structures

---

struct **esp_ip6_addr**

IPv6 address.

Public Members

uint32_t **addr**[4]

IPv6 address

uint8_t **zone**

zone ID

---

struct **esp_ip4_addr**

IPv4 address.

Public Members

uint32_t **addr**

IPv4 address

---

struct **_ip_addr**

IP address.

Public Members

esp_ip6_addr_t **ip6**

IPv6 address type

esp_ip4_addr_t **ip4**

IPv4 address type

union _ip_addr::[anonymous] **u_addr**

IP address union

uint8_t **type**

ipaddress type

Macros

esp_netif_htonl(x)

esp_netif_ip4_makeu32(a, b, c, d)

ESP_IP6_ADDR_BLOCK1(ip6addr)

ESP_IP6_ADDR_BLOCK2(ip6addr)

ESP_IP6_ADDR_BLOCK3(ip6addr)

ESP_IP6_ADDR_BLOCK4(ip6addr)

ESP_IP6_ADDR_BLOCK5(ip6addr)

ESP_IP6_ADDR_BLOCK6(ip6addr)

ESP_IP6_ADDR_BLOCK7(ip6addr)

ESP_IP6_ADDR_BLOCK8(ip6addr)

IPSTR

esp_ip4_addr_get_byte(ipaddr, idx)

esp_ip4_addr1(ipaddr)

esp_ip4_addr2(ipaddr)

esp_ip4_addr3(ipaddr)

esp_ip4_addr4(ipaddr)

esp_ip4_addr1_16(ipaddr)

esp_ip4_addr2_16(ipaddr)

esp_ip4_addr3_16(ipaddr)

esp_ip4_addr4_16(ipaddr)

IP2STR(ipaddr)

IPV6STR

IPV62STR(ipaddr)

ESP_IPADDR_TYPE_V4

ESP_IPADDR_TYPE_V6

ESP_IPADDR_TYPE_ANY

ESP_IP4TOUINT32(a, b, c, d)

ESP_IP4TOADDR(a, b, c, d)

ESP_IP4ADDR_INIT(a, b, c, d)

ESP_IP6ADDR_INIT(a, b, c, d)

IP4ADDR_STRLEN_MAX

ESP_IP_IS_ANY(addr)

Type Definitions

typedef struct esp_ip4_addr esp_ip4_addr_t

typedef struct esp_ip6_addr esp_ip6_addr_t

typedef struct _ip_addr **esp_ip_addr_t**

IP address.

Enumerations

enum **esp_ip6_addr_type_t**

Values:

> *enumerator* **ESP_IP6_ADDR_IS_UNKNOWN**
>
> *enumerator* **ESP_IP6_ADDR_IS_GLOBAL**
>
> *enumerator* **ESP_IP6_ADDR_IS_LINK_LOCAL**
>
> *enumerator* **ESP_IP6_ADDR_IS_SITE_LOCAL**
>
> *enumerator* **ESP_IP6_ADDR_IS_UNIQUE_LOCAL**
>
> *enumerator* **ESP_IP6_ADDR_IS_IPV4_MAPPED_IPV6**

Header File

- components/esp_netif/include/esp_vfs_l2tap.h
- This header file can be included with:

```
#include "esp_vfs_l2tap.h"
```

- This header file is a part of the API provided by the `esp_netif` component. To declare that your component depends on `esp_netif`, add the following to your CMakeLists.txt:

```
REQUIRES esp_netif
```

or

```
PRIV_REQUIRES esp_netif
```

Functions

esp_err_t **esp_vfs_l2tap_intf_register**(l2tap_vfs_config_t *config)

Add L2 TAP virtual filesystem driver.

This function must be called prior usage of ESP-NETIF L2 TAP Interface

Parameters: **config** -- L2 TAP virtual filesystem driver configuration. Default base path /dev/net/tap is used when this paramenter is NULL.

Returns: esp_err_t

- ESP_OK on success

esp_err_t **esp_vfs_l2tap_intf_unregister**(*const* char *base_path)

Removes L2 TAP virtual filesystem driver.

Parameters: **base_path** -- Base path to the L2 TAP virtual filesystem driver. Default path /dev/net/tap is used when this paramenter is NULL.

Returns: esp_err_t

- ESP_OK on success

esp_err_t **esp_vfs_l2tap_eth_filter**(l2tap_iodriver_handle driver_handle, void *buff, size_t *size)

Filters received Ethernet L2 frames into L2 TAP infrastructure.

Parameters:

- **driver_handle** -- handle of driver at which the frame was received
- **buff** -- received L2 frame
- **size** -- input length of the L2 frame which is set to 0 when frame is filtered into L2 TAP

Returns: esp_err_t

- ESP_OK is always returned

Structures

struct **l2tap_vfs_config_t**

L2Tap VFS config parameters.

Public Members

const char ***base_path**

vfs base path

Macros

L2TAP_VFS_DEFAULT_PATH

L2TAP_VFS_CONFIG_DEFAULT()

Type Definitions

typedef void ***l2tap_iodriver_handle**

Enumerations

enum **l2tap_ioctl_opt_t**

Values:

enumerator **L2TAP_S_RCV_FILTER**

enumerator **L2TAP_G_RCV_FILTER**

enumerator **L2TAP_S_INTF_DEVICE**

enumerator **L2TAP_G_INTF_DEVICE**

enumerator **L2TAP_S_DEVICE_DRV_HNDL**

enumerator **L2TAP_G_DEVICE_DRV_HNDL**

Wi-Fi Default API Reference

Header File

- components/esp_wifi/include/esp_wifi_default.h
- This header file can be included with:

```
#include "esp_wifi_default.h"
```

- This header file is a part of the API provided by the `esp_wifi` component. To declare that your component depends on `esp_wifi`, add the following to your CMakeLists.txt:

```
REQUIRES esp_wifi
```

or

```
PRIV_REQUIRES esp_wifi
```

Functions

esp_err_t **esp_netif_attach_wifi_station**(esp_netif_t *esp_netif)

Attaches wifi station interface to supplied netif.

Parameters: **esp_netif** -- instance to attach the wifi station to

Returns:

- ESP_OK on success
- ESP_FAIL if attach failed

esp_err_t **esp_netif_attach_wifi_ap**(esp_netif_t *esp_netif)

Attaches wifi soft AP interface to supplied netif.

Parameters: **esp_netif** -- instance to attach the wifi AP to

Returns:

- ESP_OK on success
- ESP_FAIL if attach failed

esp_err_t **esp_wifi_set_default_wifi_sta_handlers**(void)

Sets default wifi event handlers for STA interface.

Returns:

- ESP_OK on success, error returned from esp_event_handler_register if failed

esp_err_t **esp_wifi_set_default_wifi_ap_handlers**(void)

Sets default wifi event handlers for AP interface.

Returns:

- ESP_OK on success, error returned from esp_event_handler_register if failed

esp_err_t **esp_wifi_set_default_wifi_nan_handlers**(void)

Sets default wifi event handlers for NAN interface.

Returns:

- ESP_OK on success, error returned from esp_event_handler_register if failed

---

esp_err_t **esp_wifi_clear_default_wifi_driver_and_handlers**(void *esp_netif)

Clears default wifi event handlers for supplied network interface.

Parameters: **esp_netif** -- instance of corresponding if object

Returns:

- ESP_OK on success, error returned from esp_event_handler_register if failed

---

esp_netif_t ***esp_netif_create_default_wifi_ap**(void)

Creates default WIFI AP. In case of any init error this API aborts.

Note

The API creates esp_netif object with default WiFi access point config, attaches the netif to wifi and registers wifi handlers to the default event loop. This API uses assert() to check for potential errors, so it could abort the program. (Note that the default event loop needs to be created prior to calling this API)

Returns: pointer to esp-netif instance

---

esp_netif_t ***esp_netif_create_default_wifi_sta**(void)

Creates default WIFI STA. In case of any init error this API aborts.

Note

The API creates esp_netif object with default WiFi station config, attaches the netif to wifi and registers wifi handlers to the default event loop. This API uses assert() to check for potential errors, so it could abort the program. (Note that the default event loop needs to be created prior to calling this API)

Returns: pointer to esp-netif instance

---

esp_netif_t ***esp_netif_create_default_wifi_nan**(void)

Creates default WIFI NAN. In case of any init error this API aborts.

Note

The API creates esp_netif object with default WiFi station config, attaches the netif to wifi and registers wifi handlers to the default event loop. (Note that the default event loop needs to be created prior to calling this API)

Returns: pointer to esp-netif instance

---

void **esp_netif_destroy_default_wifi**(void *esp_netif)

Destroys default WIFI netif created with esp_netif_create_default_wifi_...() API.

Note

This API unregisters wifi handlers and detaches the created object from the wifi. (this function is a no-operation if esp_netif is NULL)

Parameters: **esp_netif** -- **[in]** object to detach from WiFi and destroy

---

esp_netif_t ***esp_netif_create_wifi**(wifi_interface_t wifi_if, *const* esp_netif_inherent_config_t *esp_netif_config)

Creates esp_netif WiFi object based on the custom configuration.

Attention

This API DOES NOT register default handlers!

Parameters:
- **wifi_if** -- **[in]** type of wifi interface
- **esp_netif_config** -- **[in]** inherent esp-netif configuration pointer

Returns: pointer to esp-netif instance

---

esp_err_t **esp_netif_create_default_wifi_mesh_netifs**(esp_netif_t **p_netif_sta, esp_netif_t **p_netif_ap)

Creates default STA and AP network interfaces for esp-mesh.

Both netifs are almost identical to the default station and softAP, but with DHCP client and server disabled. Please note that the DHCP client is typically enabled only if the device is promoted to a root node.

Returns created interfaces which could be ignored setting parameters to NULL if an application code does not need to save the interface instances for further processing.

Parameters:
- **p_netif_sta** -- **[out]** pointer where the resultant STA interface is saved (if non NULL)
- **p_netif_ap** -- **[out]** pointer where the resultant AP interface is saved (if non NULL)

Returns: ESP_OK on success