ESP-AT Lib

Tilen MAJERLE

Feb 16, 2020

CONTENTS

Welcome to the documentation for version latest-develop.

ESP-AT Lib is generic, platform independent, library for control of *ESP8266* or *ESP32* WiFi-based microcontrollers from *Espressif systems*. Its objective is to run on master system, while Espressif device runs official AT commands firmware developed and maintained by *Espressif systems*.

Download library · *Getting started* · Open Github

CHAPTER

ONE

FEATURES

- Supports latest ESP8266 and ESP32 RTOS-SDK AT commands firmware
- Platform independent and easy to port, written in C99
  - Library is developed under Win32 platform
  - Provided examples for ARM Cortex-M or Win32 platforms
- Allows different configurations to optimize user requirements
- Optimized for systems with operating systems (or RTOS)
  - Currently only OS mode is supported
  - 2 different threads to process user inputs and received data
    - Producer thread to collect user commands from application threads and to start command execution
    - Process thread to process received data from *ESP* device
- Allows sequential API for connections in client and server mode
- Includes several applications built on top of library
  - HTTP server with dynamic files (file system) support
  - MQTT client for MQTT connection
  - MQTT client Cayenne API for Cayenne MQTT server
- Embeds other AT features, such as WPS
- User friendly MIT license

CHAPTER

TWO

REQUIREMENTS

- C compiler
- *ESP8266* or *ESP32* device with running AT-Commands firmware

CHAPTER

THREE

CONTRIBUTE

Fresh contributions are always welcome. Simple instructions to proceed:

1. Fork Github repository
2. Respect C style & coding rules used by the library
3. Create a pull request to `develop` branch with new features or bug fixes

Alternatively you may:

1. Report a bug
2. Ask for a feature request

CHAPTER

FOUR

LICENSE

```
MIT License

Copyright (c) 2020 Tilen MAJERLE

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

CHAPTER

FIVE

TABLE OF CONTENTS

5.1 Getting started

5.1.1 Download library

Library is primarly hosted on Github.

- Download latest release from releases area on Github
- Clone *develop* branch for latest development

Download from releases

All releases are available on Github releases area.

Clone from Github

First-time clone

- Download and install `git` if not already
- Open console and navigate to path in the system to clone repository to. Use command `cd your_path`
- Clone repository with one of available `3` options
  - Run `git clone --recurse-submodules https://github.com/MaJerle/esp-at-lib` command to clone entire repository, including submodules
  - Run `git clone --recurse-submodules --branch develop https://github.com/MaJerle/esp-at-lib` to clone *development* branch, including submodules
  - Run `git clone --recurse-submodules --branch master https://github.com/MaJerle/esp-at-lib` to clone *latest stable* branch, including submodules
- Navigate to `examples` directory and run favourite example

Update cloned to latest version

- Open console and navigate to path in the system where your resources repository is. Use command `cd your_path`
- Run `git pull origin master --recurse-submodules` command to pull latest changes and to fetch latest changes from submodules
- Run `git submodule foreach git pull origin master` to update & merge all submodules

Note: This is preferred option to use when you want to evaluate library and run prepared examples. Repository consists of multiple submodules which can be automatically downloaded when cloning and pulling changes from root repository.

5.1.2 Add library to project

At this point it is assumed that you have successfully download library, either cloned it or from releases page.

- Copy `esp_at_lib` folder to your project
- Add `esp_at_lib/src/include` folder to *include path* of your toolchain
- Add port architecture `esp_at_lib/src/include/system/port/_arch_` folder to *include path* of your toolchain
- Add source files from `esp_at_lib/src/` folder to toolchain build
- Add source files from `esp_at_lib/src/system/` folder to toolchain build for arch port
- Copy `esp_at_lib/src/include/esp/esp_config_template.h` to project folder and rename it to `esp_config.h`
- Build the project

5.1.3 Configuration file

Library comes with template config file, which can be modified according to needs. This file shall be named `esp_config.h` and its default template looks like the one below:

Tip: Check *ESP Configuration* section for possible configuration settings

Listing 1: Config file template

```
1  /**
2   * \file            esp_config_template.h
3   * \brief           Template config file
4   */
5
6  /*
7   * Copyright (c) 2020 Tilen MAJERLE
8   *
9   * Permission is hereby granted, free of charge, to any person
10  * obtaining a copy of this software and associated documentation
11  * files (the "Software"), to deal in the Software without restriction,
```

(continues on next page)

(continued from previous page)

```
 * including without limitation the rights to use, copy, modify, merge,
 * publish, distribute, sublicense, and/or sell copies of the Software,
 * and to permit persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be
 * included in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
 * OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
 * AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
 * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
 * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
 * OTHER DEALINGS IN THE SOFTWARE.
 *
 * This file is part of ESP-AT library.
 *
 * Author:          Tilen MAJERLE <tilen@majerle.eu>
 * Version:         $_version_$
 */
#ifndef ESP_HDR_CONFIG_H
#define ESP_HDR_CONFIG_H

/* Rename this file to "esp_config.h" for your application */

/*
 * Open "include/esp/esp_config_default.h" and
 * copy & replace here settings you want to change values
 */

/* After user configuration, call default config to merge config together */
#include "esp/esp_config_default.h"

#endif /* ESP_HDR_CONFIG_H */
```

5.2 User manual

5.2.1 Overview

WiFi devices (focus on *ESP8266* and *ESP32*) from *Espressif Systems* are low-cost and very useful for embedded projects. These are classic microcontrollers without embedded flash memory. Application needs to assure external Quad-SPI flash to execute code from it directly.

Espressif offers SDK to program these microcontrollers directly and run code from there. It is called *RTOS-based SDK*, written in C language, and allows customers to program MCU starting with `main` function. These devices have some basic peripherals, such as GPIO, ADC, SPI, I2C, UART, etc. Pretty basic though.

Wifi connectivity is often part of bigger system with more powerful MCU. There is usually bigger MCU + Wifi transceiver (usually module) aside with UART/SPI communication. MCU handles application, such as display & graphics, runs operating systems, drives motor and has additional external memories.

Espressif is not only developing *RTOS SDK* firmware, it also develops *AT Slave firmware* based on *RTOS-SDK*. This is a special application, which is running on *ESP* device and allows host MCU to send *AT commands* and get response

Fig. 1: Typical application example with access to WiFi

for it. Now it is time to use *ESP-AT Lib* you are reading this manual for.

ESP-AT Lib has been developed to allow customers to:

- Develop on single (host MCU) architecture at the same time and do not care about *Espressif* arch
- Shorten time to market

Customers using *ESP-AT Lib* do not need to take care about proper command for specific task, they can call API functions, such as `esp_sta_join()` to join WiFi network instead. Library will take the necessary steps in order to send right command to device via low-level driver (usually UART) and process incoming response from device before it will notify application layer if it was successfuly or not.

Note: *ESP-AT Lib* offers efficient communication between host MCU at one side and *Espressif* wifi transceiver on another side.

To summarize:

- *ESP* device runs official *AT* firmware, provided by *Espressif systems*
- Host MCU runs custom application, together with *ESP-AT Lib* library
- Host MCU communicates with *ESP* device with UART or similar interface.

5.2.2 Architecture

Architecture of the library consists of `4` layers.

Fig. 2: ESP-AT layer architecture overview

Application layer

User layer is the highest layer of the final application. This is the part where API functions are called to execute some command.

Middleware layer

Middleware part is actively developed and shall not be modified by customer by any means. If there is a necessity to do it, often it means that developer of the application uses it wrongly. This part is platform independent and does not use any specific compiler features for proper operation.

Note: There is no compiler specific features implemented in this layer.

System & low-level layer

Application needs to fully implement this part and resolve it with care. Functions are related to actual implementation with *ESP* device and are highly architecture oriented. Some examples for *WIN32* and *ARM Cortex-M* are included with library.

Tip: Check *Porting guide* for detailed instructions and examples.

System functions

System functions are bridge between operating system running on embedded system and ESP-AT Library. Functions need to provide:

- Thread management
- Binary semaphore management
- Recursive mutex management
- Message queue management
- Current time status information

Tip: System function prototypes are available in *System functions* section.

Low-level implementation

Low-Level, or *ESP_LL*, is part, dedicated for communication between *ESP-AT* middleware and *ESP* physical device. Application needs to implement output function to send necessary *AT command* instruction aswell as implement *input module* to send received data from *ESP* device to *ESP-AT* middleware.

Application must also assure memory assignment for *Memory manager* when default allocation is used.

Tip: Low level, input module & memory function prototypes are available in *Low-Level functions*, *Input module* and *Memory manager* respectfully.

ESP physical device

5.2.3 Inter thread communication

ESP-AT Library is only available with operating system. For successful resources management, it uses 2 threads within library and allows multiple application threads to post new command to be processed.

Fig. 3: Inter-thread architecture block diagram

Producing and *Processing* threads are part of library, its implementation is in `esp_threads.c` file.

Processing thread

Processing thread is in charge of processing each and every received character from *ESP* device. It can process *URC* messages which are received from *ESP* device without any command request. Some of them are:

- *+IPD* indicating new data packet received from remote side on active connection
- *WIFI CONNECTED* indicating *ESP* has been just connected to access point
- and more others

Note: Received messages without any command (URC messages) are sent to application layer using events, where they can be processed and used in further steps

---

This thread also checks and processes specific received messages based on active command. As an example, when application tries to make a new connection to remote server, it starts command with `AT+CIPSTART` message. Thread understands that active command is to connect to remote side and will wait for potential `+LINK_CONN:<...>` message, indicating connection status. it will also wait for `OK` or `ERROR`, indicating *command finished* status before it unlocks **sync_sem** to unblock *producing thread*.

Tip: When thread tries to unlock **sync_sem**, it first checks if it has been locked by *producing thread*.

Producing thread

Producing thread waits for command messages posted from application thread. When new message has been received, it sends initial *AT message* over AT port.

- It checks if command is valid and if it has corresponding initial AT sequence, such as `AT+CIPSTART`
- It locks **sync_sem** semaphore and waits for processing thread to unlock it
  - *Processing thread* is in charge to read respone from *ESP* and react accordingly. See previous section for details.
- If application uses *blocking mode*, it unlocks command **sem** semaphore and returns response
- If application uses *non-blocking mode*, it frees memory for message and sends event with response message

Application thread

Application thread is considered any thread which calls API functions and therefore writes new messages to *producing message queue*, later processed by *producing thread*.

A new message memory is allocated in this thread and type of command is assigned to it, together with required input data for command. It also sets *blocking* or *non-blocking* mode, how command shall be executed.

When application tries to execute command in *blocking mode*, it creates new sync semaphore **sem**, locks it, writes message to *producing queue* and waits for **sem** to get unlocked. This effectively puts thread to blocked state by operating system and removes it from scheduler until semaphore is unlocked again. Semaphore **sem** gets unlocked in *producing thread* when response has been received for specific command.

Tip: **sem** semaphore is unlocked in *producing* thread after **sync_sem** is unlocked in *processing* thread

Note: Every command message uses its own **sem** semaphore to sync multiple *application* threads at the same time.

If message is to be executed in *non-blocking* mode, **sem** is not created as there is no need to block application thread. When this is the case, application thread will only write message command to *producing queue* and return status of writing to application.

5.2.4 Events and callback functions

Library uses events to notify application layer for (possible, but not limited to) unexpected events. This concept is used aswell for commands with longer executing time, such as *scanning access points* or when application starts new connection as client mode.

There are 3 types of events/callbacks available:

- *Global event* callback function, assigned when initializing library
- *Connection specific event* callback function, to process only events related to connection, such as *connection error*, *data send*, *data receive*, *connection closed*
- *API function* call based event callback function

Every callback is always called from protected area of middleware (when exclusing access is granted to single thread only), and it can be called from one of these 3 threads:

- *Producing thread*
- *Processing thread*
- *Input thread*, when `ESP_CFG_INPUT_USE_PROCESS` is enabled and `esp_input_process()` function is called

Tip: Check *Inter thread communication* for more details about *Producing* and *Processing* thread.

Global event callback

Global event callback function is assigned at library initialization. It is used by the application to receive any kind of event, except the one related to connection:

- ESP station successfully connected to access point
- ESP physical device reset has been detected
- Restore operation finished
- New station has connected to access point
- and many more..

Tip: Check *Event management* section for different kind of events

By default, global event function is single function. If the application tries to split different events with different callback functions, it is possible to do so by using `esp_evt_register()` function to register a new, custom, event function.

Tip: Implementation of *Netconn API* leverages `esp_evt_register()` to receive event when station disconnected from wifi access point. Check its source file for actual implementation.

Listing 2: Netconn API module actual implementation

```
/**
 * \file            esp_netconn.c
 * \brief           API functions for sequential calls
 */

/*
 * Copyright (c) 2020 Tilen MAJERLE
 *
 * Permission is hereby granted, free of charge, to any person
 * obtaining a copy of this software and associated documentation
 * files (the "Software"), to deal in the Software without restriction,
 * including without limitation the rights to use, copy, modify, merge,
 * publish, distribute, sublicense, and/or sell copies of the Software,
 * and to permit persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be
 * included in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
 * OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
 * AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
 * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
 * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
 * OTHER DEALINGS IN THE SOFTWARE.
 *
 * This file is part of ESP-AT library.
 *
```

(continues on next page)

(continued from previous page)

```
 * Author:          Tilen MAJERLE <tilen@majerle.eu>
 * Version:         $_version_$
 */
#include "esp/esp_netconn.h"
#include "esp/esp_private.h"
#include "esp/esp_conn.h"
#include "esp/esp_mem.h"

#if ESP_CFG_NETCONN || __DOXYGEN__

/* Check conditions */
#if ESP_CFG_NETCONN_RECEIVE_QUEUE_LEN < 2
#error "ESP_CFG_NETCONN_RECEIVE_QUEUE_LEN must be greater or equal to 2"
#endif /* ESP_CFG_NETCONN_RECEIVE_QUEUE_LEN < 2 */

#if ESP_CFG_NETCONN_ACCEPT_QUEUE_LEN < 2
#error "ESP_CFG_NETCONN_ACCEPT_QUEUE_LEN must be greater or equal to 2"
#endif /* ESP_CFG_NETCONN_ACCEPT_QUEUE_LEN < 2 */

/**
 * \brief           Sequential API structure
 */
typedef struct esp_netconn {
    struct esp_netconn* next;                   /*!< Linked list entry */

    esp_netconn_type_t type;                    /*!< Netconn type */
    esp_port_t listen_port;                     /*!< Port on which we are listening */

    size_t rcv_packets;                         /*!< Number of received packets so␣
↪far on this connection */
    esp_conn_p conn;                            /*!< Pointer to actual connection */

    esp_sys_mbox_t mbox_accept;                 /*!< List of active connections␣
↪waiting to be processed */
    esp_sys_mbox_t mbox_receive;                /*!< Message queue for receive mbox */
    size_t mbox_receive_entries;                /*!< Number of entries written to␣
↪receive mbox */

    esp_linbuff_t buff;                         /*!< Linear buffer structure */

    uint16_t conn_timeout;                      /*!< Connection timeout in units of␣
↪seconds when
                                                    netconn is in server (listen)␣
↪mode.
                                                    Connection will be automatically␣
↪closed if there is no
                                                    data exchange in time. Set to `0`␣
↪when timeout feature is disabled. */

#if ESP_CFG_NETCONN_RECEIVE_TIMEOUT || __DOXYGEN__
    uint32_t rcv_timeout;                       /*!< Receive timeout in unit of␣
↪milliseconds */
#endif
} esp_netconn_t;

static uint8_t recv_closed = 0xFF, recv_not_present = 0xFF;
static esp_netconn_t* listen_api;               /*!< Main connection in listening␣
↪mode */
```

(continues on next page)

(continued from previous page)

```c
static esp_netconn_t* netconn_list;             /*!< Linked list of netconn entries */

/**
 * \brief           Flush all mboxes and clear possible used memories
 * \param[in]       nc: Pointer to netconn to flush
 * \param[in]       protect: Set to 1 to protect against multi-thread access
 */
static void
flush_mboxes(esp_netconn_t* nc, uint8_t protect) {
    esp_pbuf_p pbuf;
    esp_netconn_t* new_nc;
    if (protect) {
        esp_core_lock();
    }
    if (esp_sys_mbox_isvalid(&nc->mbox_receive)) {
        while (esp_sys_mbox_getnow(&nc->mbox_receive, (void **)&pbuf)) {
            if (nc->mbox_receive_entries > 0) {
                --nc->mbox_receive_entries;
            }
            if (pbuf != NULL && (uint8_t *)pbuf != (uint8_t *)&recv_closed) {
                esp_pbuf_free(pbuf);            /* Free received data buffers */
            }
        }
        esp_sys_mbox_delete(&nc->mbox_receive); /* Delete message queue */
        esp_sys_mbox_invalid(&nc->mbox_receive);/* Invalid handle */
    }
    if (esp_sys_mbox_isvalid(&nc->mbox_accept)) {
        while (esp_sys_mbox_getnow(&nc->mbox_accept, (void **)&new_nc)) {
            if (new_nc != NULL
                && (uint8_t *)new_nc != (uint8_t *)&recv_closed
                && (uint8_t *)new_nc != (uint8_t *)&recv_not_present) {
                esp_netconn_close(new_nc);      /* Close netconn connection */
            }
        }
        esp_sys_mbox_delete(&nc->mbox_accept);  /* Delete message queue */
        esp_sys_mbox_invalid(&nc->mbox_accept); /* Invalid handle */
    }
    if (protect) {
        esp_core_unlock();
    }
}

/**
 * \brief           Callback function for every server connection
 * \param[in]       evt: Pointer to callback structure
 * \return          Member of \ref espr_t enumeration
 */
static espr_t
netconn_evt(esp_evt_t* evt) {
    esp_conn_p conn;
    esp_netconn_t* nc = NULL;
    uint8_t close = 0;

    conn = esp_conn_get_from_evt(evt);          /* Get connection from event */
    switch (esp_evt_get_type(evt)) {
        /*
         * A new connection has been active
```

(continues on next page)

(continued from previous page)

```
         * and should be handled by netconn API
         */
        case ESP_EVT_CONN_ACTIVE: {             /* A new connection active is active */
            if (esp_conn_is_client(conn)) {     /* Was connection started by us? */
                nc = esp_conn_get_arg(conn);    /* Argument should be already set */
                if (nc != NULL) {
                    nc->conn = conn;            /* Save actual connection */
                } else {
                    close = 1;                  /* Close this connection, invalid netconn */
                }
            } else if (esp_conn_is_server(conn) && listen_api != NULL) {    /* Is the connection server type and we have known listening API? */
                /*
                 * Create a new netconn structure
                 * and set it as connection argument.
                 */
                nc = esp_netconn_new(ESP_NETCONN_TYPE_TCP); /* Create new API */
                ESP_DEBUGW(ESP_CFG_DBG_NETCONN | ESP_DBG_TYPE_TRACE | ESP_DBG_LVL_WARNING,
                    nc == NULL, "[NETCONN] Cannot create new structure for incoming server connection!\r\n");

                if (nc != NULL) {
                    nc->conn = conn;            /* Set connection handle */
                    esp_conn_set_arg(conn, nc); /* Set argument for connection */

                    /*
                     * In case there is no listening connection,
                     * simply close the connection
                     */
                    if (!esp_sys_mbox_isvalid(&listen_api->mbox_accept)
                        || !esp_sys_mbox_putnow(&listen_api->mbox_accept, nc)) {
                        close = 1;
                    }
                } else {
                    close = 1;
                }
            } else {
                ESP_DEBUGW(ESP_CFG_DBG_NETCONN | ESP_DBG_TYPE_TRACE | ESP_DBG_LVL_WARNING, listen_api == NULL,
                    "[NETCONN] Closing connection as there is no listening API in netconn!\r\n");
                close = 1;                      /* Close the connection at this point */
            }

            /* Decide if some events want to close the connection */
            if (close) {
                if (nc != NULL) {
                    esp_conn_set_arg(conn, NULL);   /* Reset argument */
                    esp_netconn_delete(nc);     /* Free memory for API */
                }
                esp_conn_close(conn, 0);        /* Close the connection */
                close = 0;
            }
```

(continues on next page)

(continued from previous page)

```
            break;
        }

        /*
         * We have a new data received which
         * should have netconn structure as argument
         */
        case ESP_EVT_CONN_RECV: {
            esp_pbuf_p pbuf;

            nc = esp_conn_get_arg(conn);        /* Get API from connection */
            pbuf = esp_evt_conn_recv_get_buff(evt); /* Get received buff */

#if !ESP_CFG_CONN_MANUAL_TCP_RECEIVE
            esp_conn_recved(conn, pbuf);        /* Notify stack about received data */
#endif /* !ESP_CFG_CONN_MANUAL_TCP_RECEIVE */

            esp_pbuf_ref(pbuf);                 /* Increase reference counter */
            if (nc == NULL || !esp_sys_mbox_isvalid(&nc->mbox_receive)
                || !esp_sys_mbox_putnow(&nc->mbox_receive, pbuf)) {
                ESP_DEBUGF(ESP_CFG_DBG_NETCONN,
                    "[NETCONN] Ignoring more data for receive!\r\n");
                esp_pbuf_free(pbuf);            /* Free pbuf */
                return espOKIGNOREMORE;         /* Return OK to free the memory and ignore further data */
            }
            ++nc->mbox_receive_entries;         /* Increase number of packets in receive mbox */
#if ESP_CFG_CONN_MANUAL_TCP_RECEIVE
            /* Check against 1 less to still allow potential close event to be written to queue */
            if (nc->mbox_receive_entries >= (ESP_CFG_NETCONN_RECEIVE_QUEUE_LEN - 1)) {
                conn->status.f.receive_blocked = 1; /* Block reading more data */
            }
#endif /* ESP_CFG_CONN_MANUAL_TCP_RECEIVE */

            ++nc->rcv_packets;                  /* Increase number of packets received */
            ESP_DEBUGF(ESP_CFG_DBG_NETCONN | ESP_DBG_TYPE_TRACE,
                "[NETCONN] Received pbuf contains %d bytes. Handle written to receive mbox\r\n",
                (int)esp_pbuf_length(pbuf, 0));
            break;
        }

        /* Connection was just closed */
        case ESP_EVT_CONN_CLOSE: {
            nc = esp_conn_get_arg(conn);        /* Get API from connection */

            /*
             * In case we have a netconn available,
             * simply write pointer to received variable to indicate closed state
             */
            if (nc != NULL && esp_sys_mbox_isvalid(&nc->mbox_receive)) {
                if (esp_sys_mbox_putnow(&nc->mbox_receive, (void *)&recv_closed)) {
                    ++nc->mbox_receive_entries;
                }
```

(continues on next page)

(continued from previous page)

```
            }

            break;
        }
        default:
            return espERR;
    }
    return espOK;
}

/**
 * \brief           Global event callback function
 * \param[in]       evt: Callback information and data
 * \return          \ref espOK on success, member of \ref espr_t otherwise
 */
static espr_t
esp_evt(esp_evt_t* evt) {
    switch (esp_evt_get_type(evt)) {
        case ESP_EVT_WIFI_DISCONNECTED: {       /* Wifi disconnected event */
            if (listen_api != NULL) {           /* Check if listen API active */
                esp_sys_mbox_putnow(&listen_api->mbox_accept, &recv_closed);
            }
            break;
        }
        case ESP_EVT_DEVICE_PRESENT: {          /* Device present event */
            if (listen_api != NULL && !esp_device_is_present()) {   /* Check if device present */
                esp_sys_mbox_putnow(&listen_api->mbox_accept, &recv_not_present);
            }
        }
        default: break;
    }
    return espOK;
}

/**
 * \brief           Create new netconn connection
 * \param[in]       type: Netconn connection type
 * \return          New netconn connection on success, `NULL` otherwise
 */
esp_netconn_p
esp_netconn_new(esp_netconn_type_t type) {
    esp_netconn_t* a;
    static uint8_t first = 1;

    /* Register only once! */
    esp_core_lock();
    if (first) {
        first = 0;
        esp_evt_register(esp_evt);              /* Register global event function */
    }
    esp_core_unlock();
    a = esp_mem_calloc(1, sizeof(*a));          /* Allocate memory for core object */
    if (a != NULL) {
        a->type = type;                         /* Save netconn type */
        a->conn_timeout = 0;                    /* Default connection timeout */
        if (!esp_sys_mbox_create(&a->mbox_accept, ESP_CFG_NETCONN_ACCEPT_QUEUE_LEN)) {  /* Allocate memory for accepting message box */
```

(continues on next page)

(continued from previous page)

```
            ESP_DEBUGF(ESP_CFG_DBG_NETCONN | ESP_DBG_TYPE_TRACE | ESP_DBG_LVL_DANGER,
                "[NETCONN] Cannot create accept MBOX\r\n");
            goto free_ret;
        }
        if (!esp_sys_mbox_create(&a->mbox_receive, ESP_CFG_NETCONN_RECEIVE_QUEUE_
↪LEN)) {    /* Allocate memory for receiving message box */
            ESP_DEBUGF(ESP_CFG_DBG_NETCONN | ESP_DBG_TYPE_TRACE | ESP_DBG_LVL_DANGER,
                "[NETCONN] Cannot create receive MBOX\r\n");
            goto free_ret;
        }
        esp_core_lock();
        if (netconn_list == NULL) {             /* Add new netconn to the existing␣
↪list */
            netconn_list = a;
        } else {
            a->next = netconn_list;             /* Add it to beginning of the list */
            netconn_list = a;
        }
        esp_core_unlock();
    }
    return a;
free_ret:
    if (esp_sys_mbox_isvalid(&a->mbox_accept)) {
        esp_sys_mbox_delete(&a->mbox_accept);
        esp_sys_mbox_invalid(&a->mbox_accept);
    }
    if (esp_sys_mbox_isvalid(&a->mbox_receive)) {
        esp_sys_mbox_delete(&a->mbox_receive);
        esp_sys_mbox_invalid(&a->mbox_receive);
    }
    if (a != NULL) {
        esp_mem_free_s((void **)&a);
    }
    return NULL;
}

/**
 * \brief           Delete netconn connection
 * \param[in]       nc: Netconn handle
 * \return          \ref espOK on success, member of \ref espr_t enumeration otherwise
 */
espr_t
esp_netconn_delete(esp_netconn_p nc) {
    ESP_ASSERT("netconn != NULL", nc != NULL);

    esp_core_lock();
    flush_mboxes(nc, 0);                        /* Clear mboxes */

    /* Stop listening on netconn */
    if (nc == listen_api) {
        listen_api = NULL;
        esp_core_unlock();
        esp_set_server(0, nc->listen_port, 0, 0, NULL, NULL, NULL, 1);
        esp_core_lock();
    }

    /* Remove netconn from linkedlist */
```

(continues on next page)

(continued from previous page)

```
    if (nc == netconn_list) {
        netconn_list = netconn_list->next;      /* Remove first from linked list */
    } else if (netconn_list != NULL) {
        esp_netconn_p tmp, prev;
        /* Find element on the list */
        for (prev = netconn_list, tmp = netconn_list->next;
            tmp != NULL; prev = tmp, tmp = tmp->next) {
            if (nc == tmp) {
                prev->next = tmp->next;         /* Remove tmp from linked list */
                break;
            }
        }
    }
    esp_core_unlock();

    esp_mem_free_s((void **)&nc);
    return espOK;
}

/**
 * \brief           Connect to server as client
 * \param[in]       nc: Netconn handle
 * \param[in]       host: Pointer to host, such as domain name or IP address in string format
 * \param[in]       port: Target port to use
 * \return          \ref espOK if successfully connected, member of \ref espr_t otherwise
 */
espr_t
esp_netconn_connect(esp_netconn_p nc, const char* host, esp_port_t port) {
    espr_t res;

    ESP_ASSERT("nc != NULL", nc != NULL);
    ESP_ASSERT("host != NULL", host != NULL);
    ESP_ASSERT("port > 0", port > 0);

    /*
     * Start a new connection as client and:
     *
     *  - Set current netconn structure as argument
     *  - Set netconn callback function for connection management
     *  - Start connection in blocking mode
     */
    res = esp_conn_start(NULL, (esp_conn_type_t)nc->type, host, port, nc, netconn_evt, 1);
    return res;
}

/**
 * \brief           Connect to server as client, allow keep-alive option
 * \param[in]       nc: Netconn handle
 * \param[in]       host: Pointer to host, such as domain name or IP address in string format
 * \param[in]       port: Target port to use
 * \param[in]       keep_alive: Keep alive period seconds
 * \param[in]       local_ip: Local ip in connected command
 * \param[in]       local_port: Local port address
```

(continues on next page)

(continued from previous page)

```
 * \param[in]       mode: UDP mode
 * \return          \ref espOK if successfully connected, member of \ref espr_t otherwise
 */
espr_t
esp_netconn_connect_ex(esp_netconn_p nc, const char* host, esp_port_t port, uint16_t keep_alive, const char* local_ip, esp_port_t local_port, uint8_t mode) {
    esp_conn_start_t cs = {0};
    espr_t res;

    ESP_ASSERT("nc != NULL", nc != NULL);
    ESP_ASSERT("host != NULL", host != NULL);
    ESP_ASSERT("port > 0", port > 0);

    /*
     * Start a new connection as client and:
     *
     *  - Set current netconn structure as argument
     *  - Set netconn callback function for connection management
     *  - Start connection in blocking mode
     */
    cs.type = nc->type;
    cs.remote_host = host;
    cs.remote_port = port;
    cs.local_ip = local_ip;
    if (nc->type == ESP_NETCONN_TYPE_TCP || nc->type == ESP_NETCONN_TYPE_SSL) {
        cs.ext.tcp_ssl.keep_alive = keep_alive;
    } else {
        cs.ext.udp.local_port = local_port;
        cs.ext.udp.mode = mode;
    }
    res = esp_conn_startex(NULL, &cs, nc, netconn_evt, 1);
    return res;
}

/**
 * \brief           Bind a connection to specific port, can be only used for server connections
 * \param[in]       nc: Netconn handle
 * \param[in]       port: Port used to bind a connection to
 * \return          \ref espOK on success, member of \ref espr_t enumeration otherwise
 */
espr_t
esp_netconn_bind(esp_netconn_p nc, esp_port_t port) {
    espr_t res = espOK;

    ESP_ASSERT("nc != NULL", nc != NULL);

    /*
     * Protection is not needed as it is expected
     * that this function is called only from single
     * thread for single netconn connection,
     * thus it is considered reentrant
     */

    nc->listen_port = port;

```

(continues on next page)

(continued from previous page)

```
456     return res;
457 }
458 
459 /**
460  * \brief           Set timeout value in units of seconds when connection is in listening mode
461  *                  If new connection is accepted, it will be automatically closed after `seconds` elapsed
462  *                  without any data exchange.
463  * \note            Call this function before you put connection to listen mode with \ref esp_netconn_listen
464  * \param[in]       nc: Netconn handle used for listen mode
465  * \param[in]       timeout: Time in units of seconds. Set to `0` to disable timeout feature
466  * \return          \ref espOK on success, member of \ref espr_t otherwise
467  */
468 espr_t
469 esp_netconn_set_listen_conn_timeout(esp_netconn_p nc, uint16_t timeout) {
470     espr_t res = espOK;
471     ESP_ASSERT("nc != NULL", nc != NULL);
472 
473     /*
474      * Protection is not needed as it is expected
475      * that this function is called only from single
476      * thread for single netconn connection,
477      * thus it is reentrant in this case
478      */
479 
480     nc->conn_timeout = timeout;
481 
482     return res;
483 }
484 
485 /**
486  * \brief           Listen on previously binded connection
487  * \param[in]       nc: Netconn handle used to listen for new connections
488  * \return          \ref espOK on success, member of \ref espr_t enumeration otherwise
489  */
490 espr_t
491 esp_netconn_listen(esp_netconn_p nc) {
492     return esp_netconn_listen_with_max_conn(nc, ESP_CFG_MAX_CONNS);
493 }
494 
495 /**
496  * \brief           Listen on previously binded connection with max allowed connections at a time
497  * \param[in]       nc: Netconn handle used to listen for new connections
498  * \param[in]       max_connections: Maximal number of connections server can accept at a time
499  *                      This parameter may not be larger than \ref ESP_CFG_MAX_CONNS
500  * \return          \ref espOK on success, member of \ref espr_t otherwise
501  */
502 espr_t
503 esp_netconn_listen_with_max_conn(esp_netconn_p nc, uint16_t max_connections) {
504     espr_t res;
505 
506     ESP_ASSERT("nc != NULL", nc != NULL);
```

(continues on next page)

(continued from previous page)

```
507     ESP_ASSERT("nc->type must be TCP", nc->type == ESP_NETCONN_TYPE_TCP);
508 
509     /* Enable server on port and set default netconn callback */
510     if ((res = esp_set_server(1, nc->listen_port,
511         ESP_U16(ESP_MIN(max_connections, ESP_CFG_MAX_CONNS)),
512         nc->conn_timeout, netconn_evt, NULL, NULL, 1)) == espOK) {
513         esp_core_lock();
514         listen_api = nc;                        /* Set current main API in listening
    ↪state */
515         esp_core_unlock();
516     }
517     return res;
518 }
519 
520 /**
521  * \brief           Accept a new connection
522  * \param[in]       nc: Netconn handle used as base connection to accept new clients
523  * \param[out]      client: Pointer to netconn handle to save new connection to
524  * \return          \ref espOK on success, member of \ref espr_t enumeration otherwise
525  */
526 espr_t
527 esp_netconn_accept(esp_netconn_p nc, esp_netconn_p* client) {
528     esp_netconn_t* tmp;
529     uint32_t time;
530 
531     ESP_ASSERT("nc != NULL", nc != NULL);
532     ESP_ASSERT("client != NULL", client != NULL);
533     ESP_ASSERT("nc->type must be TCP", nc->type == ESP_NETCONN_TYPE_TCP);
534     ESP_ASSERT("nc == listen_api", nc == listen_api);
535 
536     *client = NULL;
537     time = esp_sys_mbox_get(&nc->mbox_accept, (void **)&tmp, 0);
538     if (time == ESP_SYS_TIMEOUT) {
539         return espTIMEOUT;
540     }
541     if ((uint8_t *)tmp == (uint8_t *)&recv_closed) {
542         esp_core_lock();
543         listen_api = NULL;                      /* Disable listening at this point */
544         esp_core_unlock();
545         return espERRWIFINOTCONNECTED;          /* Wifi disconnected */
546     } else if ((uint8_t *)tmp == (uint8_t *)&recv_not_present) {
547         esp_core_lock();
548         listen_api = NULL;                      /* Disable listening at this point */
549         esp_core_unlock();
550         return espERRNODEVICE;                  /* Device not present */
551     }
552     *client = tmp;                              /* Set new pointer */
553     return espOK;                               /* We have a new connection */
554 }
555 
556 /**
557  * \brief           Write data to connection output buffers
558  * \note            This function may only be used on TCP or SSL connections
559  * \param[in]       nc: Netconn handle used to write data to
560  * \param[in]       data: Pointer to data to write
561  * \param[in]       btw: Number of bytes to write
562  * \return          \ref espOK on success, member of \ref espr_t enumeration otherwise
```

(continues on next page)

(continued from previous page)

```
 */
espr_t
esp_netconn_write(esp_netconn_p nc, const void* data, size_t btw) {
    size_t len, sent;
    const uint8_t* d = data;
    espr_t res;

    ESP_ASSERT("nc != NULL", nc != NULL);
    ESP_ASSERT("nc->type must be TCP or SSL", nc->type == ESP_NETCONN_TYPE_TCP || nc->type == ESP_NETCONN_TYPE_SSL);
    ESP_ASSERT("nc->conn must be active", esp_conn_is_active(nc->conn));

    /*
     * Several steps are done in write process
     *
     * 1. Check if buffer is set and check if there is something to write to it.
     *    1. In case buffer will be full after copy, send it and free memory.
     * 2. Check how many bytes we can write directly without needed to copy
     * 3. Try to allocate a new buffer and copy remaining input data to it
     * 4. In case buffer allocation fails, send data directly (may affect on speed and effectivenes)
     */

    /* Step 1 */
    if (nc->buff.buff != NULL) {                /* Is there a write buffer ready to accept more data? */
        len = ESP_MIN(nc->buff.len - nc->buff.ptr, btw);    /* Get number of bytes we can write to buffer */
        if (len > 0) {
            ESP_MEMCPY(&nc->buff.buff[nc->buff.ptr], data, len);/* Copy memory to temporary write buffer */
            d += len;
            nc->buff.ptr += len;
            btw -= len;
        }

        /* Step 1.1 */
        if (nc->buff.ptr == nc->buff.len) {
            res = esp_conn_send(nc->conn, nc->buff.buff, nc->buff.len, &sent, 1);

            esp_mem_free_s((void **)&nc->buff.buff);
            if (res != espOK) {
                return res;
            }
        } else {
            return espOK;                       /* Buffer is not yet full yet */
        }
    }

    /* Step 2 */
    if (btw >= ESP_CFG_CONN_MAX_DATA_LEN) {
        size_t rem;
        rem = btw % ESP_CFG_CONN_MAX_DATA_LEN;  /* Get remaining bytes for max data length */
        res = esp_conn_send(nc->conn, d, btw - rem, &sent, 1);  /* Write data directly */
        if (res != espOK) {
```

(continues on next page)

(continued from previous page)

```
            return res;
        }
        d += sent;                                  /* Advance in data pointer */
        btw -= sent;                                /* Decrease remaining data to send */
    }

    if (btw == 0) {                                 /* Sent everything? */
        return espOK;
    }

    /* Step 3 */
    if (nc->buff.buff == NULL) {                    /* Check if we should allocate a new buffer */
        nc->buff.buff = esp_mem_malloc(sizeof(*nc->buff.buff) * ESP_CFG_CONN_MAX_DATA_LEN);
        nc->buff.len = ESP_CFG_CONN_MAX_DATA_LEN;   /* Save buffer length */
        nc->buff.ptr = 0;                           /* Save buffer pointer */
    }

    /* Step 4 */
    if (nc->buff.buff != NULL) {                    /* Memory available? */
        ESP_MEMCPY(&nc->buff.buff[nc->buff.ptr], d, btw);   /* Copy data to buffer */
        nc->buff.ptr += btw;
    } else {                                        /* Still no memory available? */
        return esp_conn_send(nc->conn, data, btw, NULL, 1); /* Simply send directly blocking */
    }
    return espOK;
}

/**
 * \brief           Flush buffered data on netconn \e TCP/SSL connection
 * \note            This function may only be used on \e TCP/SSL connection
 * \param[in]       nc: Netconn handle to flush data
 * \return          \ref espOK on success, member of \ref espr_t enumeration otherwise
 */
espr_t
esp_netconn_flush(esp_netconn_p nc) {
    ESP_ASSERT("nc != NULL", nc != NULL);
    ESP_ASSERT("nc->type must be TCP or SSL", nc->type == ESP_NETCONN_TYPE_TCP || nc->type == ESP_NETCONN_TYPE_SSL);
    ESP_ASSERT("nc->conn must be active", esp_conn_is_active(nc->conn));

    /*
     * In case we have data in write buffer,
     * flush them out to network
     */
    if (nc->buff.buff != NULL) {                    /* Check remaining data */
        if (nc->buff.ptr > 0) {                     /* Do we have data in current buffer? */
            esp_conn_send(nc->conn, nc->buff.buff, nc->buff.ptr, NULL, 1);  /* Send data */
        }
        esp_mem_free_s((void **)&nc->buff.buff);
    }
    return espOK;
}
```

(continues on next page)

(continued from previous page)

```

/**
 * \brief           Send data on \e UDP connection to default IP and port
 * \param[in]       nc: Netconn handle used to send
 * \param[in]       data: Pointer to data to write
 * \param[in]       btw: Number of bytes to write
 * \return          \ref espOK on success, member of \ref espr_t enumeration otherwise
 */
espr_t
esp_netconn_send(esp_netconn_p nc, const void* data, size_t btw) {
    ESP_ASSERT("nc != NULL", nc != NULL);
    ESP_ASSERT("nc->type must be UDP", nc->type == ESP_NETCONN_TYPE_UDP);
    ESP_ASSERT("nc->conn must be active", esp_conn_is_active(nc->conn));

    return esp_conn_send(nc->conn, data, btw, NULL, 1);
}

/**
 * \brief           Send data on \e UDP connection to specific IP and port
 * \note            Use this function in case of UDP type netconn
 * \param[in]       nc: Netconn handle used to send
 * \param[in]       ip: Pointer to IP address
 * \param[in]       port: Port number used to send data
 * \param[in]       data: Pointer to data to write
 * \param[in]       btw: Number of bytes to write
 * \return          \ref espOK on success, member of \ref espr_t enumeration otherwise
 */
espr_t
esp_netconn_sendto(esp_netconn_p nc, const esp_ip_t* ip, esp_port_t port, const void*␣
↪data, size_t btw) {
    ESP_ASSERT("nc != NULL", nc != NULL);
    ESP_ASSERT("nc->type must be UDP", nc->type == ESP_NETCONN_TYPE_UDP);
    ESP_ASSERT("nc->conn must be active", esp_conn_is_active(nc->conn));

    return esp_conn_sendto(nc->conn, ip, port, data, btw, NULL, 1);
}

/**
 * \brief           Receive data from connection
 * \param[in]       nc: Netconn handle used to receive from
 * \param[in]       pbuf: Pointer to pointer to save new receive buffer to.
 *                     When function returns, user must check for valid pbuf value␣
↪`pbuf != NULL`
 * \return          \ref espOK when new data ready
 * \return          \ref espCLOSED when connection closed by remote side
 * \return          \ref espTIMEOUT when receive timeout occurs
 * \return          Any other member of \ref espr_t otherwise
 */
espr_t
esp_netconn_receive(esp_netconn_p nc, esp_pbuf_p* pbuf) {
    ESP_ASSERT("nc != NULL", nc != NULL);
    ESP_ASSERT("pbuf != NULL", pbuf != NULL);

    *pbuf = NULL;
#if ESP_CFG_NETCONN_RECEIVE_TIMEOUT
    /*
     * Wait for new received data for up to specific timeout
```

(continues on next page)

(continued from previous page)

```
     * or throw error for timeout notification
     */
    if (nc->rcv_timeout == ESP_NETCONN_RECEIVE_NO_WAIT) {
        if (!esp_sys_mbox_getnow(&nc->mbox_receive, (void **)pbuf)) {
            return espTIMEOUT;
        }
    } else if (esp_sys_mbox_get(&nc->mbox_receive, (void **)pbuf, nc->rcv_timeout) ==
↪ESP_SYS_TIMEOUT) {
        return espTIMEOUT;
    }
#else /* ESP_CFG_NETCONN_RECEIVE_TIMEOUT */
    /* Forever wait for new receive packet */
    esp_sys_mbox_get(&nc->mbox_receive, (void **)pbuf, 0);
#endif /* !ESP_CFG_NETCONN_RECEIVE_TIMEOUT */

    esp_core_lock();
    if (nc->mbox_receive_entries > 0) {
        --nc->mbox_receive_entries;
    }
    esp_core_unlock();

    /* Check if connection closed */
    if ((uint8_t *)(*pbuf) == (uint8_t *)&recv_closed) {
        *pbuf = NULL;                           /* Reset pbuf */
        return espCLOSED;
    }
#if ESP_CFG_CONN_MANUAL_TCP_RECEIVE
    else {
        esp_core_lock();
        nc->conn->status.f.receive_blocked = 0; /* Resume reading more data */
        esp_conn_recved(nc->conn, *pbuf);       /* Notify stack about received data */
        esp_core_unlock();
    }
#endif /* ESP_CFG_CONN_MANUAL_TCP_RECEIVE */
    return espOK;                               /* We have data available */
}

/**
 * \brief           Close a netconn connection
 * \param[in]       nc: Netconn handle to close
 * \return          \ref espOK on success, member of \ref espr_t enumeration otherwise
 */
espr_t
esp_netconn_close(esp_netconn_p nc) {
    esp_conn_p conn;

    ESP_ASSERT("nc != NULL", nc != NULL);
    ESP_ASSERT("nc->conn != NULL", nc->conn != NULL);
    ESP_ASSERT("nc->conn must be active", esp_conn_is_active(nc->conn));

    esp_netconn_flush(nc);                      /* Flush data and ignore result */
    conn = nc->conn;
    nc->conn = NULL;

    esp_conn_set_arg(conn, NULL);               /* Reset argument */
    esp_conn_close(conn, 1);                    /* Close the connection */
    flush_mboxes(nc, 1);                        /* Flush message queues */
```

(continues on next page)

(continued from previous page)

```
    return espOK;
}

/**
 * \brief           Get connection number used for netconn
 * \param[in]       nc: Netconn handle
 * \return          `-1` on failure, connection number between `0` and \ref ESP_CFG_MAX_CONNS otherwise
 */
int8_t
esp_netconn_get_connnum(esp_netconn_p nc) {
    if (nc != NULL && nc->conn != NULL) {
        return esp_conn_getnum(nc->conn);
    }
    return -1;
}

#if ESP_CFG_NETCONN_RECEIVE_TIMEOUT || __DOXYGEN__

/**
 * \brief           Set timeout value for receiving data.
 *
 * When enabled, \ref esp_netconn_receive will only block for up to
 * \e timeout value and will return if no new data within this time
 *
 * \param[in]       nc: Netconn handle
 * \param[in]       timeout: Timeout in units of milliseconds.
 *                      Set to `0` to disable timeout feature
 *                      Set to `> 0` to set maximum milliseconds to wait before timeout
 *                      Set to \ref ESP_NETCONN_RECEIVE_NO_WAIT to enable non-blocking receive
 */
void
esp_netconn_set_receive_timeout(esp_netconn_p nc, uint32_t timeout) {
    nc->rcv_timeout = timeout;
}

/**
 * \brief           Get netconn receive timeout value
 * \param[in]       nc: Netconn handle
 * \return          Timeout in units of milliseconds.
 *                  If value is `0`, timeout is disabled (wait forever)
 */
uint32_t
esp_netconn_get_receive_timeout(esp_netconn_p nc) {
    return nc->rcv_timeout;
}

#endif /* ESP_CFG_NETCONN_RECEIVE_TIMEOUT || __DOXYGEN__ */

/**
 * \brief           Get netconn connection handle
 * \param[in]       nc: Netconn handle
 * \return          ESP connection handle
 */
esp_conn_p
```

(continues on next page)

(continued from previous page)

```
esp_netconn_get_conn(esp_netconn_p nc) {
    return nc->conn;
}

#endif /* ESP_CFG_NETCONN || __DOXYGEN__ */
```

Connection specific event

This events are subset of global event callback. They work exactly the same way as global, but only receive events related to connections.

Tip: Connection related events start with `ESP_EVT_CONN_*`, such as `ESP_EVT_CONN_RECV`. Check *Event management* for list of all connection events.

Connection events callback function is set for `2` cases:

- Each client (when application starts connection) sets event callback function when trying to connect with *`esp_conn_start()`* function
- Application sets global event callback function when enabling server mode with *`esp_set_server()`* function

Listing 3: An example of client with its dedicated event callback function

```
#include "client.h"
#include "esp/esp.h"

/* Host parameter */
#define CONN_HOST           "example.com"
#define CONN_PORT           80

static espr_t   conn_callback_func(esp_evt_t* evt);

/**
 * \brief           Request data for connection
 */
static const
uint8_t req_data[] = ""
"GET / HTTP/1.1\r\n"
"Host: " CONN_HOST "\r\n"
"Connection: close\r\n"
"\r\n";

/**
 * \brief           Start a new connection(s) as client
 */
void
client_connect(void) {
    espr_t res;

    /* Start a new connection as client in non-blocking mode */
    if ((res = esp_conn_start(NULL, ESP_CONN_TYPE_TCP, "example.com", 80, NULL, conn_
↪callback_func, 0)) == espOK) {
        printf("Connection to " CONN_HOST " started...\r\n");
```

(continues on next page)

(continued from previous page)

```
    } else {
        printf("Cannot start connection to " CONN_HOST "!\r\n");
    }

    /* Start 2 more */
    esp_conn_start(NULL, ESP_CONN_TYPE_TCP, CONN_HOST, CONN_PORT, NULL, conn_callback_
↪func, 0);

    /*
     * An example of connection which should fail in connecting.
     * When this is the case, \ref ESP_EVT_CONN_ERROR event should be triggered
     * in callback function processing
     */
    esp_conn_start(NULL, ESP_CONN_TYPE_TCP, CONN_HOST, 10, NULL, conn_callback_func,␣
↪0);
}

/**
 * \brief           Event callback function for connection-only
 * \param[in]       evt: Event information with data
 * \return          \ref espOK on success, member of \ref espr_t otherwise
 */
static espr_t
conn_callback_func(esp_evt_t* evt) {
    esp_conn_p conn;
    espr_t res;
    uint8_t conn_num;

    conn = esp_conn_get_from_evt(evt);          /* Get connection handle from event */
    if (conn == NULL) {
        return espERR;
    }
    conn_num = esp_conn_getnum(conn);           /* Get connection number for␣
↪identification */
    switch (esp_evt_get_type(evt)) {
        case ESP_EVT_CONN_ACTIVE: {             /* Connection just active */
            printf("Connection %d active!\r\n", (int)conn_num);
            res = esp_conn_send(conn, req_data, sizeof(req_data) - 1, NULL, 0); /*␣
↪Start sending data in non-blocking mode */
            if (res == espOK) {
                printf("Sending request data to server...\r\n");
            } else {
                printf("Cannot send request data to server. Closing connection␣
↪manually...\r\n");
                esp_conn_close(conn, 0);        /* Close the connection */
            }
            break;
        }
        case ESP_EVT_CONN_CLOSE: {              /* Connection closed */
            if (esp_evt_conn_close_is_forced(evt)) {
                printf("Connection %d closed by client!\r\n", (int)conn_num);
            } else {
                printf("Connection %d closed by remote side!\r\n", (int)conn_num);
            }
            break;
        }
        case ESP_EVT_CONN_SEND: {               /* Data send event */
```

(continues on next page)

(continued from previous page)

```
            espr_t res = esp_evt_conn_send_get_result(evt);
            if (res == espOK) {
                printf("Data sent successfully on connection %d...waiting to receive data from remote side...\r\n", (int)conn_num);
            } else {
                printf("Error while sending data on connection %d!\r\n", (int)conn_num);
            }
            break;
        }
        case ESP_EVT_CONN_RECV: {               /* Data received from remote side */
            esp_pbuf_p pbuf = esp_evt_conn_recv_get_buff(evt);
            esp_conn_recved(conn, pbuf);        /* Notify stack about received pbuf */
            printf("Received %d bytes on connection %d..\r\n", (int)esp_pbuf_length(pbuf, 1), (int)conn_num);
            break;
        }
        case ESP_EVT_CONN_ERROR: {              /* Error connecting to server */
            const char* host = esp_evt_conn_error_get_host(evt);
            esp_port_t port = esp_evt_conn_error_get_port(evt);
            printf("Error connecting to %s:%d\r\n", host, (int)port);
            break;
        }
        default: break;
    }
    return espOK;
}
```

API call event

API function call event function is special type of event and is linked to command execution. It is especially useful when dealing with non-blocking commands to understand when specific command execution finished and when next operation could start.

Every API function, which directly operates with AT command on physical device layer, has optional `2` parameters for API call event:

- Callback function, called when command finished
- Custom user parameter for callback function

Below is an example code for DNS resolver. It uses custom API callback function with custom argument, used to distinguis domain name (when multiple domains are to be resolved).

Listing 4: Simple example for API call event, using DNS module

```
#include "dns.h"
#include "esp/esp.h"

/* Host to resolve */
#define DNS_HOST1           "example.com"
#define DNS_HOST2           "example.net"

/**
 * \brief           Variable to hold result of DNS resolver
 */
```

(continues on next page)

(continued from previous page)

```c
static esp_ip_t ip;

/**
 * \brief           Event callback function for API call,
 *                  called when API command finished with execution
 */
static void
dns_resolve_evt(espr_t res, void* arg) {
    /* Check result of command */
    if (res == espOK) {
        /* DNS resolver has IP address */
        printf("DNS record for %s (from API callback): %d.%d.%d.%d\r\n",
            (const char *)arg, (int)ip.ip[0], (int)ip.ip[1], (int)ip.ip[2], (int)ip.ip[3]);
    }
}

/**
 * \brief           Start DNS resolver
 */
void
dns_start(void) {
    /* Use DNS protocol to get IP address of domain name */

    /* Get IP with non-blocking mode */
    if (esp_dns_gethostbyname(DNS_HOST2, &ip, dns_resolve_evt, DNS_HOST2, 0) == espOK) {
        printf("Request for DNS record for " DNS_HOST2 " has started\r\n");
    } else {
        printf("Could not start command for DNS\r\n");
    }

    /* Get IP with blocking mode */
    if (esp_dns_gethostbyname(DNS_HOST1, &ip, dns_resolve_evt, DNS_HOST1, 1) == espOK) {
        printf("DNS record for " DNS_HOST1 " (from lin code): %d.%d.%d.%d\r\n",
            (int)ip.ip[0], (int)ip.ip[1], (int)ip.ip[2], (int)ip.ip[3]);
    } else {
        printf("Could not retrieve IP address for " DNS_HOST1 "\r\n");
    }
}
```

5.2.5 Blocking or non-blocking API calls

API functions often allow application to set `blocking` parameter indicating if function shall be blocking or non-blocking.

Blocking mode

When the function is called in blocking mode `blocking = 1`, application thread gets suspended until response from *ESP* device is received. If there is a queue of multiple commands, thread may wait a while before receiving data.

When API function returns, application has valid response data and can react immediately.

- Linear programming model may be used
- Application may use multiple threads for real-time execution to prevent system stalling when running function call

Warning: Due to internal architecture, it is not allowed to call API functions in *blocking mode* from events or callbacks. Any attempt to do so will result in function returning error.

Example code:

Listing 5: Blocking command example

```
char hostname[20];

/* Somewhere in thread function */

/* Get device hostname in blocking mode */
/* Function returns actual result */
if (esp_hostname_get(hostname, sizeof(hostname), NULL, NULL, 1 /* 1 means blocking␣
↪call */) == espOK) {
    /* At this point we have valid result and parameters from API function */
    printf("ESP hostname is %s\r\n", hostname);
} else {
    printf("Error reading ESP hostname..\r\n");
}
```

Non-blocking mode

If the API function is called in non-blocking mode, function will return immediately with status indicating if command request has been successfully sent to internal command queue. Response has to be processed in event callback function.

Warning: Due to internal architecture, it is only allowed to call API functions in *non-blocking mode* from events or callbacks. Any attempt to do so will result in function returning error.

Example code:

Listing 6: Non-blocking command example

```
char hostname[20];

/* Hostname event function, called when esp_hostname_get() function finishes */
void
hostname_fn(espr_t res, void* arg) {
    /* Check actual result from device */
    if (res == espOK) {
```

(continues on next page)

(continued from previous page)

```
        printf("ESP hostname is %s\r\n", hostname);
    } else {
        printf("Error reading ESP hostname...\r\n");
    }
}

/* Somewhere in thread and/or other ESP event function */

/* Get device hostname in non-blocking mode */
/* Function now returns if command has been sent to internal message queue */
if (esp_hostname_get(hostname, sizeof(hostname), hostname_fn, NULL, 0 /* 0 means non-
↪blocking call */) == espOK) {
    /* At this point application knows that command has been sent to queue */
    /* But it does not have yet valid data in "hostname" variable */
    printf("ESP hostname get command sent to queue.\r\n");
} else {
    /* Error writing message to queue */
    printf("Cannot send hostname get command to queue.\r\n");
}
```

Warning: When using non-blocking API calls, do not use local variables as parameter. This may introduce *undefined behavior* and *memory corruption* if application function returns before command is executed.

Example of a bad code:

Listing 7: Example of bad usage of non-blocking command

```
char hostname[20];

/* Hostname event function, called when esp_hostname_get() function finishes */
void
hostname_fn(espr_t res, void* arg) {
    /* Check actual result from device */
    if (res == espOK) {
        printf("ESP hostname is %s\r\n", hostname);
    } else {
        printf("Error reading ESP hostname...\r\n");
    }
}

/* Check hostname */
void
check_hostname(void) {
    char hostname[20];

    /* Somewhere in thread and/or other ESP event function */

    /* Get device hostname in non-blocking mode */
    /* Function now returns if command has been sent to internal message queue */
    /* Function will use local "hostname" variable and will write to undefined memory␣
↪*/
    if (esp_hostname_get(hostname, sizeof(hostname), hostname_fn, NULL, 0 /* 0 means␣
↪non-blocking call */) == espOK) {
        /* At this point application knows that command has been sent to queue */
```

(continues on next page)

(continued from previous page)

```
        /* But it does not have yet valid data in "hostname" variable */
        printf("ESP hostname get command sent to queue.\r\n");
    } else {
        /* Error writing message to queue */
        printf("Cannot send hostname get command to queue.\r\n");
    }
}
```

5.2.6 Porting guide

High level of *ESP-AT* library is platform independent, written in ANSI C99, however there is an important part where middleware needs to communicate with target *ESP* device and it must work under different optional operating systems selected by final customer.

Porting consists of:

- Implementation of *low-level* part, for actual communication between host device and *ESP* device
- Implementation of system functions, link between target operating system and middleware functions
- Assignment of memory for allocation manager

Implement low-level driver

To successfully implement all parts of *low-level* driver, application must take care of:

- Implementing `esp_ll_init()` and `esp_ll_deinit()` callback functions
- Implement and assign *send data* and optional *hardware reset* functions callbacks
- Assign memory for allocation manager when using default allocator
- Process received data from *ESP* device and send them to input module for further processing

---

Tip: Port examples are available for STM32 and WIN32 architectures. Both actual working and up-to-date implementations are available within the library.

Note: Check *Input module* for more information about direct & indirect input processing.

Implement system functions

System functions are bridge between operating system calls and *ESP* middleware. *ESP* library relies on stable operating system features and its implementation and does not require any special features which do not normally come with operating systems.

Operating system must support:

- Thread management functions
- Mutex management functions
- Binary semaphores only, no need for counting semaphores
- Message queue management functions

Warning: If any of the features are not available within targeted operating system, customer needs to resolve it with care. As an example, message queue is not available in WIN32 OS API therefore custom message queue has been implemented using binary semaphores

Application needs to implement all system call functions, starting with `esp_sys_`. It must also prepare header file for standard types in order to support OS types within *ESP* middleware.

An example code is provided latter section of this page for WIN32 and STM32.

Steps to follow

- Copy `esp_at_lib/src/system/esp_sys_template.c` to the same folder and rename it to application port, eg. `esp_sys_win32.c`
- Open newly created file and implement all system functions
- Copy folder `esp_at_lib/src/include/system/port/template/*` to the same folder and rename *folder name* to application port, eg. `cmsis_os`
- Open `esp_sys_port.h` file from newly created folder and implement all *typedefs* and *macros* for specific target
- Add source file to compiler sources and add path to header file to include paths in compiler options

Note: Check *System functions* for function prototypes.

Example: Low-level driver for WIN32

Example code for low-level porting on *WIN32* platform. It uses native *Windows* features to open *COM* port and read/write from/to it.

Notes:

- It uses separate thread for received data processing. It uses *esp_input_process()* or *esp_input()* functions, based on application configuration of *ESP_CFG_INPUT_USE_PROCESS* parameter.
  - When *ESP_CFG_INPUT_USE_PROCESS* is disabled, dedicated receive buffer is created by *ESP-AT* library and *esp_input()* function just writes data to it and does not process received characters immediately. This is handled by *Processing* thread at later stage instead.
  - When *ESP_CFG_INPUT_USE_PROCESS* is enabled, *esp_input_process()* is used, which directly processes input data and sends potential callback/event functions to application layer.
- Memory manager has been assigned to `1` region of `ESP_MEM_SIZE` size
- It sets *send* and *reset* callback functions for *ESP-AT* library

Listing 8: Actual implementation of low-level driver for WIN32

```
1  /**
2   * \file            esp_ll_win32.c
3   * \brief           Low-level communication with ESP device for WIN32
4   */
5
6  /*
```

(continues on next page)

(continued from previous page)

```
 * Copyright (c) 2020 Tilen MAJERLE
 *
 * Permission is hereby granted, free of charge, to any person
 * obtaining a copy of this software and associated documentation
 * files (the "Software"), to deal in the Software without restriction,
 * including without limitation the rights to use, copy, modify, merge,
 * publish, distribute, sublicense, and/or sell copies of the Software,
 * and to permit persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be
 * included in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
 * OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
 * AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
 * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
 * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
 * OTHER DEALINGS IN THE SOFTWARE.
 *
 * This file is part of ESP-AT library.
 *
 * Author:          Tilen MAJERLE <tilen@majerle.eu>
 * Version:         $_version_$
 */
#include "system/esp_ll.h"
#include "esp/esp.h"
#include "esp/esp_mem.h"
#include "esp/esp_input.h"

#if !__DOXYGEN__

volatile uint8_t esp_ll_win32_driver_ignore_data;
static uint8_t initialized = 0;
static HANDLE thread_handle;
static volatile HANDLE com_port;                /*!< COM port handle */
static uint8_t data_buffer[0x1000];             /*!< Received data array */

static void uart_thread(void* param);

/**
 * \brief           Send data to ESP device, function called from ESP stack when we have data to send
 */
static size_t
send_data(const void* data, size_t len) {
    DWORD written;
    if (com_port != NULL) {
#if !ESP_CFG_AT_ECHO
        const uint8_t* d = data;
        HANDLE hConsole;

        hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
        SetConsoleTextAttribute(hConsole, FOREGROUND_RED);
        for (DWORD i = 0; i < len; ++i) {
```

(continues on next page)

(continued from previous page)

```
            printf("%c", d[i]);
        }
        SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN |␣
↪FOREGROUND_BLUE);
#endif /* !ESP_CFG_AT_ECHO */

        WriteFile(com_port, data, len, &written, NULL);
        FlushFileBuffers(com_port);
        return written;
    }
    return 0;
}

/**
 * \brief           Configure UART (USB to UART)
 */
static void
configure_uart(uint32_t baudrate) {
    DCB dcb = { 0 };
    dcb.DCBlength = sizeof(dcb);

    /*
     * On first call,
     * create virtual file on selected COM port and open it
     * as generic read and write
     */
    if (!initialized) {
        com_port = CreateFile(L"\\\\.\\COM4",
            GENERIC_READ | GENERIC_WRITE,
            0,
            0,
            OPEN_EXISTING,
            0,
            NULL
        );
    }

    /* Configure COM port parameters */
    if (GetCommState(com_port, &dcb)) {
        COMMTIMEOUTS timeouts;

        dcb.BaudRate = baudrate;
        dcb.ByteSize = 8;
        dcb.Parity = NOPARITY;
        dcb.StopBits = ONESTOPBIT;

        if (!SetCommState(com_port, &dcb)) {
            printf("Cannot set COM PORT info\r\n");
        }
        if (GetCommTimeouts(com_port, &timeouts)) {
            /* Set timeout to return immediately from ReadFile function */
            timeouts.ReadIntervalTimeout = MAXDWORD;
            timeouts.ReadTotalTimeoutConstant = 0;
            timeouts.ReadTotalTimeoutMultiplier = 0;
            if (!SetCommTimeouts(com_port, &timeouts)) {
                printf("Cannot set COM PORT timeouts\r\n");
            }
```

(continues on next page)

(continued from previous page)

```
            GetCommTimeouts(com_port, &timeouts);
        } else {
            printf("Cannot get COM PORT timeouts\r\n");
        }
    } else {
        printf("Cannot get COM PORT info\r\n");
    }

    /* On first function call, create a thread to read data from COM port */
    if (!initialized) {
        esp_sys_thread_create(&thread_handle, "esp_ll_thread", uart_thread, NULL, 0, 0);
    }
}

/**
 * \brief           UART thread
 */
static void
uart_thread(void* param) {
    DWORD bytes_read;
    esp_sys_sem_t sem;
    FILE* file = NULL;

    esp_sys_sem_create(&sem, 0);                /* Create semaphore for delay functions */

    while (com_port == NULL) {
        esp_sys_sem_wait(&sem, 1);              /* Add some delay with yield */
    }

    fopen_s(&file, "log_file.txt", "w+");       /* Open debug file in write mode */
    while (1) {
        /*
         * Try to read data from COM port
         * and send it to upper layer for processing
         */
        do {
            ReadFile(com_port, data_buffer, sizeof(data_buffer), &bytes_read, NULL);
            if (bytes_read > 0) {
                HANDLE hConsole;
                hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
                SetConsoleTextAttribute(hConsole, FOREGROUND_GREEN);
                for (DWORD i = 0; i < bytes_read; ++i) {
                    printf("%c", data_buffer[i]);
                }
                SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE);

                if (esp_ll_win32_driver_ignore_data) {
                    printf("IGNORING..\r\n");
                    continue;
                }

                /* Send received data to input processing module */
#if ESP_CFG_INPUT_USE_PROCESS
                esp_input_process(data_buffer, (size_t)bytes_read);
```

(continues on next page)

(continued from previous page)

```
#else /* ESP_CFG_INPUT_USE_PROCESS */
                esp_input(data_buffer, (size_t)bytes_read);
#endif /* !ESP_CFG_INPUT_USE_PROCESS */

                /* Write received data to output debug file */
                if (file != NULL) {
                    fwrite(data_buffer, 1, bytes_read, file);
                    fflush(file);
                }
            }
        } while (bytes_read == (DWORD)sizeof(data_buffer));

        /* Implement delay to allow other tasks processing */
        esp_sys_sem_wait(&sem, 1);
    }
}

/**
 * \brief           Reset device GPIO management
 */
static uint8_t
reset_device(uint8_t state) {
    return 0;                   /* Hardware reset was not successful */
}

/**
 * \brief           Callback function called from initialization process
 */
espr_t
esp_ll_init(esp_ll_t* ll) {
#if !ESP_CFG_MEM_CUSTOM
    /* Step 1: Configure memory for dynamic allocations */
    static uint8_t memory[0x10000];             /* Create memory for dynamic␣
↪allocations with specific size */

    /*
     * Create memory region(s) of memory.
     * If device has internal/external memory available,
     * multiple memories may be used
     */
    esp_mem_region_t mem_regions[] = {
        { memory, sizeof(memory) }
    };
    if (!initialized) {
        esp_mem_assignmemory(mem_regions, ESP_ARRAYSIZE(mem_regions));  /* Assign␣
↪memory for allocations to ESP library */
    }
#endif /* !ESP_CFG_MEM_CUSTOM */

    /* Step 2: Set AT port send function to use when we have data to transmit */
    if (!initialized) {
        ll->send_fn = send_data;                /* Set callback function to send data␣
↪*/
        ll->reset_fn = reset_device;
    }

    /* Step 3: Configure AT port to be able to send/receive data to/from ESP device */
```

(continues on next page)

(continued from previous page)

```
    configure_uart(ll->uart.baudrate);          /* Initialize UART for communication */
    initialized = 1;
    return espOK;
}

/**
 * \brief           Callback function to de-init low-level communication part
 */
espr_t
esp_ll_deinit(esp_ll_t* ll) {
    if (thread_handle != NULL) {
        esp_sys_thread_terminate(&thread_handle);
        thread_handle = NULL;
    }
    initialized = 0;                            /* Clear initialized flag */
    return espOK;
}

#endif /* !__DOXYGEN__ */
```

Example: Low-level driver for STM32

Example code for low-level porting on *STM32* platform. It uses *CMSIS-OS* based application layer functions for implementing threads & other OS dependent features.

Notes:

- It uses separate thread for received data processing. It uses `esp_input_process()` function to directly process received data without using intermediate receive buffer
- Memory manager has been assigned to `1` region of `ESP_MEM_SIZE` size
- It sets *send* and *reset* callback functions for *ESP-AT* library

Listing 9: Actual implementation of low-level driver for STM32

```
/**
 * \file            esp_ll_stm32.c
 * \brief           Generic STM32 driver, included in various STM32 driver variants
 */

/*
 * Copyright (c) 2020 Tilen MAJERLE
 *
 * Permission is hereby granted, free of charge, to any person
 * obtaining a copy of this software and associated documentation
 * files (the "Software"), to deal in the Software without restriction,
 * including without limitation the rights to use, copy, modify, merge,
 * publish, distribute, sublicense, and/or sell copies of the Software,
 * and to permit persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be
 * included in all copies or substantial portions of the Software.
 *
```

(continues on next page)

(continued from previous page)

```
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
 * OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
 * AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
 * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
 * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
 * OTHER DEALINGS IN THE SOFTWARE.
 *
 * This file is part of ESP-AT library.
 *
 * Author:          Tilen MAJERLE <tilen@majerle.eu>
 * Version:         $_version_$
 */

/*
 * How it works
 *
 * On first call to \ref esp_ll_init, new thread is created and processed in usart_ll_thread function.
 * USART is configured in RX DMA mode and any incoming bytes are processed inside thread function.
 * DMA and USART implement interrupt handlers to notify main thread about new data ready to send to upper layer.
 *
 * More about UART + RX DMA: https://github.com/MaJerle/stm32-usart-dma-rx-tx
 *
 * \ref ESP_CFG_INPUT_USE_PROCESS must be enabled in `esp_config.h` to use this driver.
 */
#include "esp/esp.h"
#include "esp/esp_mem.h"
#include "esp/esp_input.h"
#include "system/esp_ll.h"

#if !__DOXYGEN__

#if !ESP_CFG_INPUT_USE_PROCESS
#error "ESP_CFG_INPUT_USE_PROCESS must be enabled in `esp_config.h` to use this driver."
#endif /* ESP_CFG_INPUT_USE_PROCESS */

#if !defined(ESP_USART_DMA_RX_BUFF_SIZE)
#define ESP_USART_DMA_RX_BUFF_SIZE      0x1000
#endif /* !defined(ESP_USART_DMA_RX_BUFF_SIZE) */

#if !defined(ESP_MEM_SIZE)
#define ESP_MEM_SIZE                    0x1000
#endif /* !defined(ESP_MEM_SIZE) */

#if !defined(ESP_USART_RDR_NAME)
#define ESP_USART_RDR_NAME              RDR
#endif /* !defined(ESP_USART_RDR_NAME) */

/* USART memory */
static uint8_t      usart_mem[ESP_USART_DMA_RX_BUFF_SIZE];
static uint8_t      is_running, initialized;
```

(continues on next page)

(continued from previous page)

```
static size_t       old_pos;

/* USART thread */
static void usart_ll_thread(void* arg);
static osThreadId_t usart_ll_thread_id;

/* Message queue */
static osMessageQueueId_t usart_ll_mbox_id;

/**
 * \brief           USART data processing
 */
static void
usart_ll_thread(void* arg) {
    size_t pos;

    ESP_UNUSED(arg);

    while (1) {
        void* d;
        /* Wait for the event message from DMA or USART */
        osMessageQueueGet(usart_ll_mbox_id, &d, NULL, osWaitForever);

        /* Read data */
#if defined(ESP_USART_DMA_RX_STREAM)
        pos = sizeof(usart_mem) - LL_DMA_GetDataLength(ESP_USART_DMA, ESP_USART_DMA_
↪RX_STREAM);
#else
        pos = sizeof(usart_mem) - LL_DMA_GetDataLength(ESP_USART_DMA, ESP_USART_DMA_
↪RX_CH);
#endif /* defined(ESP_USART_DMA_RX_STREAM) */
        if (pos != old_pos && is_running) {
            if (pos > old_pos) {
                esp_input_process(&usart_mem[old_pos], pos - old_pos);
            } else {
                esp_input_process(&usart_mem[old_pos], sizeof(usart_mem) - old_pos);
                if (pos > 0) {
                    esp_input_process(&usart_mem[0], pos);
                }
            }
            old_pos = pos;
            if (old_pos == sizeof(usart_mem)) {
                old_pos = 0;
            }
        }
    }
}

/**
 * \brief           Configure UART using DMA for receive in double buffer mode and␣
↪IDLE line detection
 */
static void
configure_uart(uint32_t baudrate) {
    static LL_USART_InitTypeDef usart_init;
    static LL_DMA_InitTypeDef dma_init;
    LL_GPIO_InitTypeDef gpio_init;
```

(continues on next page)

(continued from previous page)

```
126
127     if (!initialized) {
128         /* Enable peripheral clocks */
129         ESP_USART_CLK;
130         ESP_USART_DMA_CLK;
131         ESP_USART_TX_PORT_CLK;
132         ESP_USART_RX_PORT_CLK;
133
134 #if defined(ESP_RESET_PIN)
135         ESP_RESET_PORT_CLK;
136 #endif /* defined(ESP_RESET_PIN) */
137
138 #if defined(ESP_GPIO0_PIN)
139         ESP_GPIO0_PORT_CLK;
140 #endif /* defined(ESP_GPIO0_PIN) */
141
142 #if defined(ESP_GPIO2_PIN)
143         ESP_GPIO2_PORT_CLK;
144 #endif /* defined(ESP_GPIO2_PIN) */
145
146 #if defined(ESP_CH_PD_PIN)
147         ESP_CH_PD_PORT_CLK;
148 #endif /* defined(ESP_CH_PD_PIN) */
149
150         /* Global pin configuration */
151         LL_GPIO_StructInit(&gpio_init);
152         gpio_init.OutputType = LL_GPIO_OUTPUT_PUSHPULL;
153         gpio_init.Pull = LL_GPIO_PULL_UP;
154         gpio_init.Speed = LL_GPIO_SPEED_FREQ_VERY_HIGH;
155         gpio_init.Mode = LL_GPIO_MODE_OUTPUT;
156
157 #if defined(ESP_RESET_PIN)
158         /* Configure RESET pin */
159         gpio_init.Pin = ESP_RESET_PIN;
160         LL_GPIO_Init(ESP_RESET_PORT, &gpio_init);
161 #endif /* defined(ESP_RESET_PIN) */
162
163 #if defined(ESP_GPIO0_PIN)
164         /* Configure GPIO0 pin */
165         gpio_init.Pin = ESP_GPIO0_PIN;
166         LL_GPIO_Init(ESP_GPIO0_PORT, &gpio_init);
167         LL_GPIO_SetOutputPin(ESP_GPIO0_PORT, ESP_GPIO0_PIN);
168 #endif /* defined(ESP_GPIO0_PIN) */
169
170 #if defined(ESP_GPIO2_PIN)
171         /* Configure GPIO2 pin */
172         gpio_init.Pin = ESP_GPIO2_PIN;
173         LL_GPIO_Init(ESP_GPIO2_PORT, &gpio_init);
174         LL_GPIO_SetOutputPin(ESP_GPIO2_PORT, ESP_GPIO2_PIN);
175 #endif /* defined(ESP_GPIO2_PIN) */
176
177 #if defined(ESP_CH_PD_PIN)
178         /* Configure CH_PD pin */
179         gpio_init.Pin = ESP_CH_PD_PIN;
180         LL_GPIO_Init(ESP_CH_PD_PORT, &gpio_init);
181         LL_GPIO_SetOutputPin(ESP_CH_PD_PORT, ESP_CH_PD_PIN);
182 #endif /* defined(ESP_CH_PD_PIN) */
```

(continues on next page)

(continued from previous page)

```
        /* Configure USART pins */
        gpio_init.Mode = LL_GPIO_MODE_ALTERNATE;

        /* TX PIN */
        gpio_init.Alternate = ESP_USART_TX_PIN_AF;
        gpio_init.Pin = ESP_USART_TX_PIN;
        LL_GPIO_Init(ESP_USART_TX_PORT, &gpio_init);

        /* RX PIN */
        gpio_init.Alternate = ESP_USART_RX_PIN_AF;
        gpio_init.Pin = ESP_USART_RX_PIN;
        LL_GPIO_Init(ESP_USART_RX_PORT, &gpio_init);

        /* Configure UART */
        LL_USART_DeInit(ESP_USART);
        LL_USART_StructInit(&usart_init);
        usart_init.BaudRate = baudrate;
        usart_init.DataWidth = LL_USART_DATAWIDTH_8B;
        usart_init.HardwareFlowControl = LL_USART_HWCONTROL_NONE;
        usart_init.OverSampling = LL_USART_OVERSAMPLING_16;
        usart_init.Parity = LL_USART_PARITY_NONE;
        usart_init.StopBits = LL_USART_STOPBITS_1;
        usart_init.TransferDirection = LL_USART_DIRECTION_TX_RX;
        LL_USART_Init(ESP_USART, &usart_init);

        /* Enable USART interrupts and DMA request */
        LL_USART_EnableIT_IDLE(ESP_USART);
        LL_USART_EnableIT_PE(ESP_USART);
        LL_USART_EnableIT_ERROR(ESP_USART);
        LL_USART_EnableDMAReq_RX(ESP_USART);

        /* Enable USART interrupts */
        NVIC_SetPriority(ESP_USART_IRQ, NVIC_EncodePriority(NVIC_
↪GetPriorityGrouping(), 0x07, 0x00));
        NVIC_EnableIRQ(ESP_USART_IRQ);

        /* Configure DMA */
        is_running = 0;
#if defined(ESP_USART_DMA_RX_STREAM)
        LL_DMA_DeInit(ESP_USART_DMA, ESP_USART_DMA_RX_STREAM);
        dma_init.Channel = ESP_USART_DMA_RX_CH;
#else
        LL_DMA_DeInit(ESP_USART_DMA, ESP_USART_DMA_RX_CH);
        dma_init.PeriphRequest = ESP_USART_DMA_RX_REQ_NUM;
#endif /* defined(ESP_USART_DMA_RX_STREAM) */
        dma_init.PeriphOrM2MSrcAddress = (uint32_t)&ESP_USART->ESP_USART_RDR_NAME;
        dma_init.MemoryOrM2MDstAddress = (uint32_t)usart_mem;
        dma_init.Direction = LL_DMA_DIRECTION_PERIPH_TO_MEMORY;
        dma_init.Mode = LL_DMA_MODE_CIRCULAR;
        dma_init.PeriphOrM2MSrcIncMode = LL_DMA_PERIPH_NOINCREMENT;
        dma_init.MemoryOrM2MDstIncMode = LL_DMA_MEMORY_INCREMENT;
        dma_init.PeriphOrM2MSrcDataSize = LL_DMA_PDATAALIGN_BYTE;
        dma_init.MemoryOrM2MDstDataSize = LL_DMA_MDATAALIGN_BYTE;
        dma_init.NbData = sizeof(usart_mem);
        dma_init.Priority = LL_DMA_PRIORITY_MEDIUM;
#if defined(ESP_USART_DMA_RX_STREAM)
```

(continues on next page)

(continued from previous page)

```
        LL_DMA_Init(ESP_USART_DMA, ESP_USART_DMA_RX_STREAM, &dma_init);
#else
        LL_DMA_Init(ESP_USART_DMA, ESP_USART_DMA_RX_CH, &dma_init);
#endif /* defined(ESP_USART_DMA_RX_STREAM) */

        /* Enable DMA interrupts */
#if defined(ESP_USART_DMA_RX_STREAM)
        LL_DMA_EnableIT_HT(ESP_USART_DMA, ESP_USART_DMA_RX_STREAM);
        LL_DMA_EnableIT_TC(ESP_USART_DMA, ESP_USART_DMA_RX_STREAM);
        LL_DMA_EnableIT_TE(ESP_USART_DMA, ESP_USART_DMA_RX_STREAM);
        LL_DMA_EnableIT_FE(ESP_USART_DMA, ESP_USART_DMA_RX_STREAM);
        LL_DMA_EnableIT_DME(ESP_USART_DMA, ESP_USART_DMA_RX_STREAM);
#else
        LL_DMA_EnableIT_HT(ESP_USART_DMA, ESP_USART_DMA_RX_CH);
        LL_DMA_EnableIT_TC(ESP_USART_DMA, ESP_USART_DMA_RX_CH);
        LL_DMA_EnableIT_TE(ESP_USART_DMA, ESP_USART_DMA_RX_CH);
#endif /* defined(ESP_USART_DMA_RX_STREAM) */

        /* Enable DMA interrupts */
        NVIC_SetPriority(ESP_USART_DMA_RX_IRQ, NVIC_EncodePriority(NVIC_
↪GetPriorityGrouping(), 0x07, 0x00));
        NVIC_EnableIRQ(ESP_USART_DMA_RX_IRQ);

        old_pos = 0;
        is_running = 1;

        /* Start DMA and USART */
#if defined(ESP_USART_DMA_RX_STREAM)
        LL_DMA_EnableStream(ESP_USART_DMA, ESP_USART_DMA_RX_STREAM);
#else
        LL_DMA_EnableChannel(ESP_USART_DMA, ESP_USART_DMA_RX_CH);
#endif /* defined(ESP_USART_DMA_RX_STREAM) */
        LL_USART_Enable(ESP_USART);
    } else {
        osDelay(10);
        LL_USART_Disable(ESP_USART);
        usart_init.BaudRate = baudrate;
        LL_USART_Init(ESP_USART, &usart_init);
        LL_USART_Enable(ESP_USART);
    }

    /* Create mbox and start thread */
    if (usart_ll_mbox_id == NULL) {
        usart_ll_mbox_id = osMessageQueueNew(10, sizeof(void *), NULL);
    }
    if (usart_ll_thread_id == NULL) {
        const osThreadAttr_t attr = {
            .stack_size = 1024
        };
        usart_ll_thread_id = osThreadNew(usart_ll_thread, usart_ll_mbox_id, &attr);
    }
}

#if defined(ESP_RESET_PIN)
/**
 * \brief           Hardware reset callback
 */
```

(continues on next page)

(continued from previous page)

```
295 static uint8_t
296 reset_device(uint8_t state) {
297     if (state) {                                    /* Activate reset line */
298         LL_GPIO_ResetOutputPin(ESP_RESET_PORT, ESP_RESET_PIN);
299     } else {
300         LL_GPIO_SetOutputPin(ESP_RESET_PORT, ESP_RESET_PIN);
301     }
302     return 1;
303 }
304 #endif /* defined(ESP_RESET_PIN) */
305
306 /**
307  * \brief           Send data to ESP device
308  * \param[in]       data: Pointer to data to send
309  * \param[in]       len: Number of bytes to send
310  * \return          Number of bytes sent
311  */
312 static size_t
313 send_data(const void* data, size_t len) {
314     const uint8_t* d = data;
315
316     for (size_t i = 0; i < len; ++i, ++d) {
317         LL_USART_TransmitData8(ESP_USART, *d);
318         while (!LL_USART_IsActiveFlag_TXE(ESP_USART)) {}
319     }
320     return len;
321 }
322
323 /**
324  * \brief           Callback function called from initialization process
325  */
326 espr_t
327 esp_ll_init(esp_ll_t* ll) {
328 #if !ESP_CFG_MEM_CUSTOM
329     static uint8_t memory[ESP_MEM_SIZE];
330     esp_mem_region_t mem_regions[] = {
331         { memory, sizeof(memory) }
332     };
333
334     if (!initialized) {
335         esp_mem_assignmemory(mem_regions, ESP_ARRAYSIZE(mem_regions));  /* Assign␣
    ↪memory for allocations */
336     }
337 #endif /* !ESP_CFG_MEM_CUSTOM */
338
339     if (!initialized) {
340         ll->send_fn = send_data;                    /* Set callback function to send data␣
    ↪*/
341 #if defined(ESP_RESET_PIN)
342         ll->reset_fn = reset_device;                /* Set callback for hardware reset */
343 #endif /* defined(ESP_RESET_PIN) */
344     }
345
346     configure_uart(ll->uart.baudrate);              /* Initialize UART for communication␣
    ↪*/
347     initialized = 1;
348     return espOK;
```

(continues on next page)

(continued from previous page)

```
}

/**
 * \brief           Callback function to de-init low-level communication part
 */
espr_t
esp_ll_deinit(esp_ll_t* ll) {
    if (usart_ll_mbox_id != NULL) {
        osMessageQueueId_t tmp = usart_ll_mbox_id;
        usart_ll_mbox_id = NULL;
        osMessageQueueDelete(tmp);
    }
    if (usart_ll_thread_id != NULL) {
        osThreadId_t tmp = usart_ll_thread_id;
        usart_ll_thread_id = NULL;
        osThreadTerminate(tmp);
    }
    initialized = 0;
    ESP_UNUSED(ll);
    return espOK;
}

/**
 * \brief           UART global interrupt handler
 */
void
ESP_USART_IRQHANDLER(void) {
    LL_USART_ClearFlag_IDLE(ESP_USART);
    LL_USART_ClearFlag_PE(ESP_USART);
    LL_USART_ClearFlag_FE(ESP_USART);
    LL_USART_ClearFlag_ORE(ESP_USART);
    LL_USART_ClearFlag_NE(ESP_USART);

    if (usart_ll_mbox_id != NULL) {
        void* d = (void *)1;
        osMessageQueuePut(usart_ll_mbox_id, &d, 0, 0);
    }
}

/**
 * \brief           UART DMA stream/channel handler
 */
void
ESP_USART_DMA_RX_IRQHANDLER(void) {
    ESP_USART_DMA_RX_CLEAR_TC;
    ESP_USART_DMA_RX_CLEAR_HT;

    if (usart_ll_mbox_id != NULL) {
        void* d = (void *)1;
        osMessageQueuePut(usart_ll_mbox_id, &d, 0, 0);
    }
}

#endif /* !__DOXYGEN__ */
```

Example: System functions for WIN32

Listing 10: Actual header implementation of system functions for WIN32

```
/**
 * \file            esp_sys_port.h
 * \brief           WIN32 based system file implementation
 */

/*
 * Copyright (c) 2020 Tilen MAJERLE
 *
 * Permission is hereby granted, free of charge, to any person
 * obtaining a copy of this software and associated documentation
 * files (the "Software"), to deal in the Software without restriction,
 * including without limitation the rights to use, copy, modify, merge,
 * publish, distribute, sublicense, and/or sell copies of the Software,
 * and to permit persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be
 * included in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
 * OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
 * AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
 * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
 * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
 * OTHER DEALINGS IN THE SOFTWARE.
 *
 * This file is part of ESP-AT library.
 *
 * Author:          Tilen MAJERLE <tilen@majerle.eu>
 * Version:         $_version_$
 */
#ifndef ESP_HDR_SYSTEM_PORT_H
#define ESP_HDR_SYSTEM_PORT_H

#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

#include <stdint.h>
#include <stdlib.h>

#include "esp_config.h"
#include "windows.h"

#if ESP_CFG_OS && !__DOXYGEN__

typedef HANDLE                      esp_sys_mutex_t;
typedef HANDLE                      esp_sys_sem_t;
typedef HANDLE                      esp_sys_mbox_t;
typedef HANDLE                      esp_sys_thread_t;
```

(continues on next page)

(continued from previous page)

```
typedef int                             esp_sys_thread_prio_t;

#define ESP_SYS_MBOX_NULL               ((HANDLE)0)
#define ESP_SYS_SEM_NULL                ((HANDLE)0)
#define ESP_SYS_MUTEX_NULL              ((HANDLE)0)
#define ESP_SYS_TIMEOUT                 (INFINITE)
#define ESP_SYS_THREAD_PRIO             (0)
#define ESP_SYS_THREAD_SS               (1024)

#endif /* ESP_CFG_OS && !__DOXYGEN__ */

#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif /* ESP_HDR_SYSTEM_PORT_H */
```

Listing 11: Actual implementation of system functions for WIN32

```
/**
 * \file            esp_sys_win32.c
 * \brief           System dependant functions for WIN32
 */

/*
 * Copyright (c) 2020 Tilen MAJERLE
 *
 * Permission is hereby granted, free of charge, to any person
 * obtaining a copy of this software and associated documentation
 * files (the "Software"), to deal in the Software without restriction,
 * including without limitation the rights to use, copy, modify, merge,
 * publish, distribute, sublicense, and/or sell copies of the Software,
 * and to permit persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be
 * included in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
 * OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
 * AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
 * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
 * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
 * OTHER DEALINGS IN THE SOFTWARE.
 *
 * This file is part of ESP-AT library.
 *
 * Author:          Tilen MAJERLE <tilen@majerle.eu>
 * Version:         $_version_$
 */
#include "system/esp_sys.h"
#include <string.h>
#include <stdlib.h>
#include "windows.h"
```

(continues on next page)

(continued from previous page)

```

#if !__DOXYGEN__

/**
 * \brief           Custom message queue implementation for WIN32
 */
typedef struct {
    esp_sys_sem_t sem_not_empty;                /*!< Semaphore indicates not empty */
    esp_sys_sem_t sem_not_full;                 /*!< Semaphore indicates not full */
    esp_sys_sem_t sem;                          /*!< Semaphore to lock access */
    size_t in, out, size;
    void* entries[1];
} win32_mbox_t;

static LARGE_INTEGER freq, sys_start_time;
static esp_sys_mutex_t sys_mutex;               /* Mutex ID for main protection */

/**
 * \brief           Check if message box is full
 * \param[in]       m: Message box handle
 * \return          1 if full, 0 otherwise
 */
static uint8_t
mbox_is_full(win32_mbox_t* m) {
    size_t size = 0;
    if (m->in > m->out) {
        size = (m->in - m->out);
    } else if (m->out > m->in) {
        size = m->size - m->out + m->in;
    }
    return size == m->size - 1;
}

/**
 * \brief           Check if message box is empty
 * \param[in]       m: Message box handle
 * \return          1 if empty, 0 otherwise
 */
static uint8_t
mbox_is_empty(win32_mbox_t* m) {
    return m->in == m->out;
}

/**
 * \brief           Get current kernel time in units of milliseconds
 */
static uint32_t
osKernelSysTick(void) {
    LONGLONG ret;
    LARGE_INTEGER now;

    QueryPerformanceFrequency(&freq);           /* Get frequency */
    QueryPerformanceCounter(&now);              /* Get current time */
    ret = now.QuadPart - sys_start_time.QuadPart;
    return (uint32_t)(((ret) * 1000) / freq.QuadPart);
}

```

(continues on next page)

(continued from previous page)

```
uint8_t
esp_sys_init(void) {
    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&sys_start_time);

    esp_sys_mutex_create(&sys_mutex);
    return 1;
}

uint32_t
esp_sys_now(void) {
    return osKernelSysTick();
}

#if ESP_CFG_OS
uint8_t
esp_sys_protect(void) {
    esp_sys_mutex_lock(&sys_mutex);
    return 1;
}

uint8_t
esp_sys_unprotect(void) {
    esp_sys_mutex_unlock(&sys_mutex);
    return 1;
}

uint8_t
esp_sys_mutex_create(esp_sys_mutex_t* p) {
    *p = CreateMutex(NULL, FALSE, NULL);
    return *p != NULL;
}

uint8_t
esp_sys_mutex_delete(esp_sys_mutex_t* p) {
    return CloseHandle(*p);
}

uint8_t
esp_sys_mutex_lock(esp_sys_mutex_t* p) {
    DWORD ret;
    ret = WaitForSingleObject(*p, INFINITE);
    if (ret != WAIT_OBJECT_0) {
        return 0;
    }
    return 1;
}

uint8_t
esp_sys_mutex_unlock(esp_sys_mutex_t* p) {
    return ReleaseMutex(*p);
}

uint8_t
esp_sys_mutex_isvalid(esp_sys_mutex_t* p) {
    return p != NULL && *p != NULL;
}
```

(continues on next page)

(continued from previous page)

```
152
153 uint8_t
154 esp_sys_mutex_invalid(esp_sys_mutex_t* p) {
155     *p = ESP_SYS_MUTEX_NULL;
156     return 1;
157 }
158
159 uint8_t
160 esp_sys_sem_create(esp_sys_sem_t* p, uint8_t cnt) {
161     HANDLE h;
162     h = CreateSemaphore(NULL, !!cnt, 1, NULL);
163     *p = h;
164     return *p != NULL;
165 }
166
167 uint8_t
168 esp_sys_sem_delete(esp_sys_sem_t* p) {
169     return CloseHandle(*p);
170 }
171
172 uint32_t
173 esp_sys_sem_wait(esp_sys_sem_t* p, uint32_t timeout) {
174     DWORD ret;
175     uint32_t tick = osKernelSysTick();
176
177     if (timeout == 0) {
178         ret = WaitForSingleObject(*p, INFINITE);
179         return 1;
180     } else {
181         ret = WaitForSingleObject(*p, timeout);
182         if (ret == WAIT_OBJECT_0) {
183             return 1;
184         } else {
185             return ESP_SYS_TIMEOUT;
186         }
187     }
188 }
189
190 uint8_t
191 esp_sys_sem_release(esp_sys_sem_t* p) {
192     return ReleaseSemaphore(*p, 1, NULL);
193 }
194
195 uint8_t
196 esp_sys_sem_isvalid(esp_sys_sem_t* p) {
197     return p != NULL && *p != NULL;
198 }
199
200 uint8_t
201 esp_sys_sem_invalid(esp_sys_sem_t* p) {
202     *p = ESP_SYS_SEM_NULL;
203     return 1;
204 }
205
206 uint8_t
207 esp_sys_mbox_create(esp_sys_mbox_t* b, size_t size) {
208     win32_mbox_t* mbox;
```

(continues on next page)

(continued from previous page)

```
    *b = 0;

    mbox = malloc(sizeof(*mbox) + size * sizeof(void *));
    if (mbox != NULL) {
        memset(mbox, 0x00, sizeof(*mbox));
        mbox->size = size + 1;                  /* Set it to 1 more as cyclic buffer has only one less than size */
        esp_sys_sem_create(&mbox->sem, 1);
        esp_sys_sem_create(&mbox->sem_not_empty, 0);
        esp_sys_sem_create(&mbox->sem_not_full, 0);
        *b = mbox;
    }
    return *b != NULL;
}

uint8_t
esp_sys_mbox_delete(esp_sys_mbox_t* b) {
    win32_mbox_t* mbox = *b;
    esp_sys_sem_delete(&mbox->sem);
    esp_sys_sem_delete(&mbox->sem_not_full);
    esp_sys_sem_delete(&mbox->sem_not_empty);
    free(mbox);
    return 1;
}

uint32_t
esp_sys_mbox_put(esp_sys_mbox_t* b, void* m) {
    win32_mbox_t* mbox = *b;
    uint32_t time = osKernelSysTick();          /* Get start time */

    esp_sys_sem_wait(&mbox->sem, 0);            /* Wait for access */

    /*
     * Since function is blocking until ready to write something to queue,
     * wait and release the semaphores to allow other threads
     * to process the queue before we can write new value.
     */
    while (mbox_is_full(mbox)) {
        esp_sys_sem_release(&mbox->sem);        /* Release semaphore */
        esp_sys_sem_wait(&mbox->sem_not_full, 0);   /* Wait for semaphore indicating not full */
        esp_sys_sem_wait(&mbox->sem, 0);        /* Wait availability again */
    }
    mbox->entries[mbox->in] = m;
    if (++mbox->in >= mbox->size) {
        mbox->in = 0;
    }
    esp_sys_sem_release(&mbox->sem_not_empty);  /* Signal non-empty state */
    esp_sys_sem_release(&mbox->sem);            /* Release access for other threads */
    return osKernelSysTick() - time;
}

uint32_t
esp_sys_mbox_get(esp_sys_mbox_t* b, void** m, uint32_t timeout) {
    win32_mbox_t* mbox = *b;
    uint32_t time;
```

(continues on next page)

(continued from previous page)

```

    time = osKernelSysTick();

    /* Get exclusive access to message queue */
    if (esp_sys_sem_wait(&mbox->sem, timeout) == ESP_SYS_TIMEOUT) {
        return ESP_SYS_TIMEOUT;
    }
    while (mbox_is_empty(mbox)) {
        esp_sys_sem_release(&mbox->sem);
        if (esp_sys_sem_wait(&mbox->sem_not_empty, timeout) == ESP_SYS_TIMEOUT) {
            return ESP_SYS_TIMEOUT;
        }
        esp_sys_sem_wait(&mbox->sem, timeout);
    }
    *m = mbox->entries[mbox->out];
    if (++mbox->out >= mbox->size) {
        mbox->out = 0;
    }
    esp_sys_sem_release(&mbox->sem_not_full);
    esp_sys_sem_release(&mbox->sem);

    return osKernelSysTick() - time;
}

uint8_t
esp_sys_mbox_putnow(esp_sys_mbox_t* b, void* m) {
    win32_mbox_t* mbox = *b;

    esp_sys_sem_wait(&mbox->sem, 0);
    if (mbox_is_full(mbox)) {
        esp_sys_sem_release(&mbox->sem);
        return 0;
    }
    mbox->entries[mbox->in] = m;
    if (mbox->in == mbox->out) {
        esp_sys_sem_release(&mbox->sem_not_empty);
    }
    if (++mbox->in >= mbox->size) {
        mbox->in = 0;
    }
    esp_sys_sem_release(&mbox->sem);
    return 1;
}

uint8_t
esp_sys_mbox_getnow(esp_sys_mbox_t* b, void** m) {
    win32_mbox_t* mbox = *b;

    esp_sys_sem_wait(&mbox->sem, 0);            /* Wait exclusive access */
    if (mbox->in == mbox->out) {
        esp_sys_sem_release(&mbox->sem);        /* Release access */
        return 0;
    }

    *m = mbox->entries[mbox->out];
    if (++mbox->out >= mbox->size) {
        mbox->out = 0;
```

(continues on next page)

(continued from previous page)

```
    }
    esp_sys_sem_release(&mbox->sem_not_full);    /* Queue not full anymore */
    esp_sys_sem_release(&mbox->sem);             /* Release semaphore */
    return 1;
}

uint8_t
esp_sys_mbox_isvalid(esp_sys_mbox_t* b) {
    return b != NULL && *b != NULL;
}

uint8_t
esp_sys_mbox_invalid(esp_sys_mbox_t* b) {
    *b = ESP_SYS_MBOX_NULL;
    return 1;
}

uint8_t
esp_sys_thread_create(esp_sys_thread_t* t, const char* name, esp_sys_thread_fn thread_
↪func, void* const arg, size_t stack_size, esp_sys_thread_prio_t prio) {
    HANDLE h;
    DWORD id;
    h = CreateThread(0, 0, (LPTHREAD_START_ROUTINE)thread_func, arg, 0, &id);
    if (t != NULL) {
        *t = h;
    }
    return h != NULL;
}

uint8_t
esp_sys_thread_terminate(esp_sys_thread_t* t) {
    HANDLE h = NULL;

    if (t == NULL) {                             /* Shall we terminate ourself? */
        h = GetCurrentThread();                  /* Get current thread handle */
    } else {                                     /* We have known thread, find handle␣
↪by looking at ID */
        h = *t;
    }
    TerminateThread(h, 0);
    return 1;
}

uint8_t
esp_sys_thread_yield(void) {
    /* Not implemented */
    return 1;
}

#endif /* ESP_CFG_OS */
#endif /* !__DOXYGEN__ */
```

Example: System functions for CMSIS-OS

Listing 12: Actual header implementation of system functions for CMSIS-OS based operating systems

```
/**
 * \file            esp_sys_port.h
 * \brief           CMSIS-OS based system file
 */

/*
 * Copyright (c) 2020 Tilen MAJERLE
 *
 * Permission is hereby granted, free of charge, to any person
 * obtaining a copy of this software and associated documentation
 * files (the "Software"), to deal in the Software without restriction,
 * including without limitation the rights to use, copy, modify, merge,
 * publish, distribute, sublicense, and/or sell copies of the Software,
 * and to permit persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be
 * included in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
 * OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
 * AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
 * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
 * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
 * OTHER DEALINGS IN THE SOFTWARE.
 *
 * This file is part of ESP-AT library.
 *
 * Author:          Tilen MAJERLE <tilen@majerle.eu>
 * Version:         $_version_$
 */
#ifndef ESP_HDR_SYSTEM_PORT_H
#define ESP_HDR_SYSTEM_PORT_H

#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

#include <stdint.h>
#include <stdlib.h>

#include "esp_config.h"
#include "cmsis_os.h"

#if ESP_CFG_OS && !__DOXYGEN__

typedef osMutexId_t                 esp_sys_mutex_t;
typedef osSemaphoreId_t             esp_sys_sem_t;
typedef osMessageQueueId_t          esp_sys_mbox_t;
typedef osThreadId_t                esp_sys_thread_t;
```

(continues on next page)

(continued from previous page)

```
typedef osPriority_t                esp_sys_thread_prio_t;

#define ESP_SYS_MUTEX_NULL          ((esp_sys_mutex_t)0)
#define ESP_SYS_SEM_NULL            ((esp_sys_sem_t)0)
#define ESP_SYS_MBOX_NULL           ((esp_sys_mbox_t)0)
#define ESP_SYS_TIMEOUT             ((uint32_t)osWaitForever)
#define ESP_SYS_THREAD_PRIO         (osPriorityNormal)
#define ESP_SYS_THREAD_SS           (512)

#endif /* ESP_CFG_OS && !__DOXYGEN__ */

#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif /* ESP_HDR_SYSTEM_PORT_H */
```

Listing 13: Actual implementation of system functions for CMSIS-OS based operating systems

```
/**
 * \file            esp_sys_cmsis_os.c
 * \brief           System dependent functions for CMSIS based operating system
 */

/*
 * Copyright (c) 2020 Tilen MAJERLE
 *
 * Permission is hereby granted, free of charge, to any person
 * obtaining a copy of this software and associated documentation
 * files (the "Software"), to deal in the Software without restriction,
 * including without limitation the rights to use, copy, modify, merge,
 * publish, distribute, sublicense, and/or sell copies of the Software,
 * and to permit persons to whom the Software is furnished to do so,
 * subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be
 * included in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
 * OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
 * AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
 * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
 * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
 * OTHER DEALINGS IN THE SOFTWARE.
 *
 * This file is part of ESP-AT library.
 *
 * Author:          Tilen MAJERLE <tilen@majerle.eu>
 * Version:         $_version_$
 */
#include "system/esp_sys.h"
#include "cmsis_os.h"

```

(continues on next page)

(continued from previous page)

```
#if !__DOXYGEN__

static osMutexId_t sys_mutex;

uint8_t
esp_sys_init(void) {
    esp_sys_mutex_create(&sys_mutex);
    return 1;
}

uint32_t
esp_sys_now(void) {
    return osKernelSysTick();
}

uint8_t
esp_sys_protect(void) {
    esp_sys_mutex_lock(&sys_mutex);
    return 1;
}

uint8_t
esp_sys_unprotect(void) {
    esp_sys_mutex_unlock(&sys_mutex);
    return 1;
}

uint8_t
esp_sys_mutex_create(esp_sys_mutex_t* p) {
    const osMutexAttr_t attr = {
            .attr_bits = osMutexRecursive
    };
    *p = osMutexNew(&attr);
    return *p != NULL;
}

uint8_t
esp_sys_mutex_delete(esp_sys_mutex_t* p) {
    return osMutexDelete(*p) == osOK;
}

uint8_t
esp_sys_mutex_lock(esp_sys_mutex_t* p) {
    return osMutexAcquire(*p, osWaitForever) == osOK;
}

uint8_t
esp_sys_mutex_unlock(esp_sys_mutex_t* p) {
    return osMutexRelease(*p) == osOK;
}

uint8_t
esp_sys_mutex_isvalid(esp_sys_mutex_t* p) {
    return p != NULL && *p != NULL;
}

uint8_t
```

(continues on next page)

(continued from previous page)

```
esp_sys_mutex_invalid(esp_sys_mutex_t* p) {
    *p = ESP_SYS_MUTEX_NULL;
    return 1;
}

uint8_t
esp_sys_sem_create(esp_sys_sem_t* p, uint8_t cnt) {
    return (*p = osSemaphoreNew(1, cnt > 0 ? 1 : 0, NULL)) != NULL;
}

uint8_t
esp_sys_sem_delete(esp_sys_sem_t* p) {
    return osSemaphoreDelete(*p) == osOK;
}

uint32_t
esp_sys_sem_wait(esp_sys_sem_t* p, uint32_t timeout) {
    uint32_t tick = osKernelSysTick();
    return (osSemaphoreAcquire(*p, timeout == 0 ? osWaitForever : timeout) == osOK) ? (osKernelSysTick() - tick) : ESP_SYS_TIMEOUT;
}

uint8_t
esp_sys_sem_release(esp_sys_sem_t* p) {
    return osSemaphoreRelease(*p) == osOK;
}

uint8_t
esp_sys_sem_isvalid(esp_sys_sem_t* p) {
    return p != NULL && *p != NULL;
}

uint8_t
esp_sys_sem_invalid(esp_sys_sem_t* p) {
    *p = ESP_SYS_SEM_NULL;
    return 1;
}

uint8_t
esp_sys_mbox_create(esp_sys_mbox_t* b, size_t size) {
    return (*b = osMessageQueueNew(size, sizeof(void *), NULL)) != NULL;
}

uint8_t
esp_sys_mbox_delete(esp_sys_mbox_t* b) {
    if (osMessageQueueGetCount(*b) > 0) {
        return 0;
    }
    return osMessageQueueDelete(*b) == osOK;
}

uint32_t
esp_sys_mbox_put(esp_sys_mbox_t* b, void* m) {
    uint32_t tick = osKernelSysTick();
    return osMessageQueuePut(*b, &m, 0, osWaitForever) == osOK ? (osKernelSysTick() - tick) : ESP_SYS_TIMEOUT;
}
```

(continues on next page)

(continued from previous page)

```
149
150 uint32_t
151 esp_sys_mbox_get(esp_sys_mbox_t* b, void** m, uint32_t timeout) {
152     uint32_t tick = osKernelSysTick();
153     return (osMessageQueueGet(*b, m, NULL, timeout == 0 ? osWaitForever : timeout) ==␣
    ↪osOK) ? (osKernelSysTick() - tick) : ESP_SYS_TIMEOUT;
154 }
155
156 uint8_t
157 esp_sys_mbox_putnow(esp_sys_mbox_t* b, void* m) {
158     return osMessageQueuePut(*b, &m, 0, 0) == osOK;
159 }
160
161 uint8_t
162 esp_sys_mbox_getnow(esp_sys_mbox_t* b, void** m) {
163     return osMessageQueueGet(*b, m, NULL, 0) == osOK;
164 }
165
166 uint8_t
167 esp_sys_mbox_isvalid(esp_sys_mbox_t* b) {
168     return b != NULL && *b != NULL;
169 }
170
171 uint8_t
172 esp_sys_mbox_invalid(esp_sys_mbox_t* b) {
173     *b = ESP_SYS_MBOX_NULL;
174     return 1;
175 }
176
177 uint8_t
178 esp_sys_thread_create(esp_sys_thread_t* t, const char* name, esp_sys_thread_fn thread_
    ↪func, void* const arg, size_t stack_size, esp_sys_thread_prio_t prio) {
179     esp_sys_thread_t id;
180     const osThreadAttr_t thread_attr = {
181             .name = (char *)name,
182             .priority = (osPriority)prio,
183             .stack_size = stack_size > 0 ? stack_size : ESP_SYS_THREAD_SS
184     };
185
186     id = osThreadNew(thread_func, arg, &thread_attr);
187     if (t != NULL) {
188         *t = id;
189     }
190     return id != NULL;
191 }
192
193 uint8_t
194 esp_sys_thread_terminate(esp_sys_thread_t* t) {
195     if (t != NULL) {
196         osThreadTerminate(*t);
197     } else {
198         osThreadExit();
199     }
200     return 1;
201 }
202
203 uint8_t
```

(continues on next page)

(continued from previous page)

```
204 esp_sys_thread_yield(void) {
205     osThreadYield();
206     return 1;
207 }
208
209 #endif /* !__DOXYGEN__ */
```

5.3 API reference

List of all the modules:

5.3.1 ESP AT Lib

Access point

group **ESP_AP**

Access point.

Functions to manage access point (AP) on ESP device.

In order to be able to use AP feature, *ESP_CFG_MODE_ACCESS_POINT* must be enabled.

Functions

espr_t **esp_ap_getip**(*esp_ip_t* *ip, *esp_ip_t* *gw, *esp_ip_t* *nm, **const** *esp_api_cmd_evt_fn* *evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)

Get IP of access point.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[out] ip`: Pointer to variable to write IP address
- `[out] gw`: Pointer to variable to write gateway address
- `[out] nm`: Pointer to variable to write netmask address
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

espr_t **esp_ap_setip**(**const** *esp_ip_t* *ip, **const** *esp_ip_t* *gw, **const** *esp_ip_t* *nm, **const** *esp_api_cmd_evt_fn* *evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)

Set IP of access point.

Configuration changes will be saved in the NVS area of ESP device.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] ip`: Pointer to IP address

- [in] gw: Pointer to gateway address. Set to NULL to use default gateway
- [in] nm: Pointer to netmask address. Set to NULL to use default netmask
- [in] evt_fn: Callback function called when command has finished. Set to NULL when not used
- [in] evt_arg: Custom argument for event callback function
- [in] blocking: Status whether command should be blocking or not

espr_t **esp_ap_getmac**(*esp_mac_t* *mac, **const** *esp_api_cmd_evt_fn* *evt_fn*, void ***const*** *evt_arg*, **const** uint32_t *blocking*)

Get MAC of access point.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- [out] mac: Pointer to output variable to save MAC address
- [in] evt_fn: Callback function called when command has finished. Set to NULL when not used
- [in] evt_arg: Custom argument for event callback function
- [in] blocking: Status whether command should be blocking or not

espr_t **esp_ap_setmac**(**const** *esp_mac_t* *mac, **const** *esp_api_cmd_evt_fn* *evt_fn*, void ***const*** *evt_arg*, **const** uint32_t *blocking*)

Set MAC of access point.

Configuration changes will be saved in the NVS area of ESP device.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- [in] mac: Pointer to variable with MAC address. Memory of at least 6 bytes is required
- [in] evt_fn: Callback function called when command has finished. Set to NULL when not used
- [in] evt_arg: Custom argument for event callback function
- [in] blocking: Status whether command should be blocking or not

espr_t **esp_ap_get_config**(*esp_ap_conf_t* *ap_conf, **const** *esp_api_cmd_evt_fn* *evt_fn*, void ***const*** *evt_arg*, **const** uint32_t *blocking*)

Get configuration of Soft Access Point.

Note Before you can get configuration access point, ESP device must be in AP mode. Check *esp_set_wifi_mode* for more information

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- [out] ap_conf: soft access point configuration
- [in] evt_fn: Callback function called when command has finished. Set to NULL when not used
- [in] evt_arg: Custom argument for event callback function

- `[in] blocking`: Status whether command should be blocking or not

espr_t **esp_ap_configure**(**const** char *_ssid_, **const** char *_pwd_, uint8_t _ch_, esp_ecn_t _ecn_, uint8_t _max_sta_, uint8_t _hid_, **const** _esp_api_cmd_evt_fn_ _evt_fn_, void ***const** _evt_arg_, **const** uint32_t _blocking_)

Configure access point.

Configuration changes will be saved in the NVS area of ESP device.

Note Before you can configure access point, ESP device must be in AP mode. Check _esp_set_wifi_mode_ for more information

Return _espOK_ on success, member of _espr_t_ enumeration otherwise

Parameters

- `[in] ssid`: SSID name of access point
- `[in] pwd`: Password for network. Either set it to `NULL` or less than `64` characters
- `[in] ch`: Wifi RF channel
- `[in] ecn`: Encryption type. Valid options are `OPEN`, `WPA_PSK`, `WPA2_PSK` and `WPA_WPA2_PSK`
- `[in] max_sta`: Maximal number of stations access point can accept. Valid between 1 and 10 stations
- `[in] hid`: Set to `1` to hide access point from public access
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

espr_t **esp_ap_list_sta**(_esp_sta_t_ *_sta_, size_t _stal_, size_t *_staf_, **const** _esp_api_cmd_evt_fn_ _evt_fn_, void ***const** _evt_arg_, **const** uint32_t _blocking_)

List stations connected to access point.

Return _espOK_ on success, member of _espr_t_ enumeration otherwise

Parameters

- `[in] sta`: Pointer to array of _esp_sta_t_ structure to fill with stations
- `[in] stal`: Number of array entries of sta parameter
- `[out] staf`: Number of stations connected to access point
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

espr_t **esp_ap_disconn_sta**(**const** _esp_mac_t_ *_mac_, **const** _esp_api_cmd_evt_fn_ _evt_fn_, void ***const** _evt_arg_, **const** uint32_t _blocking_)

Disconnects connected station from SoftAP access point.

Return _espOK_ on success, member of _espr_t_ enumeration otherwise

Parameters

- `[in] mac`: Device MAC address to disconnect. Application may use *esp_ap_list_sta* to obtain list of connected stations to SoftAP.
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

struct esp_ap_t

#include <esp_typedefs.h> Access point data structure.

Public Members

esp_ecn_t **ecn**
Encryption mode

char **ssid**[**ESP_CFG_MAX_SSID_LENGTH**]
Access point name

int16_t **rssi**
Received signal strength indicator

esp_mac_t **mac**
MAC physical address

uint8_t **ch**
WiFi channel used on access point

uint8_t **bgn**
Information about 802.11[blgln] support

struct esp_sta_info_ap_t

#include <esp_typedefs.h> Access point information on which station is connected to.

Public Members

char **ssid**[**ESP_CFG_MAX_SSID_LENGTH**]
Access point name

int16_t **rssi**
RSSI

esp_mac_t **mac**
MAC address

uint8_t **ch**
Channel information

struct esp_ap_conf_t

#include <esp_typedefs.h> Soft access point data structure.

Public Members

char **ssid**[**ESP_CFG_MAX_SSID_LENGTH**]
Access point name

char **pwd**[**ESP_CFG_MAX_PWD_LENGTH**]
Access point password/passphrase

uint8_t **ch**
WiFi channel used on access point

esp_ecn_t **ecn**
Encryption mode

uint8_t **max_cons**
Maximum number of stations allowed connected to this AP

uint8_t **hidden**
broadcast the SSID, 0 No, 1 Yes

Ring buffer

group **ESP_BUFF**
Generic ring buffer.

Defines

BUF_PREF(x)
Buffer function/typedef prefix string.

It is used to change function names in zero time to easily re-use same library between applications. Use `#define BUF_PREF(x) my_prefix_ ## x` to change all function names to (for example) `my_prefix_buff_init`

Note Modification of this macro must be done in header and source file aswell

Functions

uint8_t **esp_buff_init**(*esp_buff_t* *buff, size_t *size*)
Initialize buffer.

Return `1` on success, `0` otherwise

Parameters

- `[in] buff`: Pointer to buffer structure
- `[in] size`: Size of buffer in units of bytes

void **esp_buff_free**(*esp_buff_t* *buff)
Free dynamic allocation if used on memory.

Parameters

- `[in] buff`: Pointer to buffer structure

void **esp_buff_reset**(*esp_buff_t *buff*)
Resets buffer to default values. Buffer size is not modified.

Parameters

- `[in] buff`: Buffer handle

size_t **esp_buff_write**(*esp_buff_t *buff*, **const** void *data, size_t *btw*)
Write data to buffer Copies data from `data` array to buffer and marks buffer as full for maximum `count` number of bytes.

Return Number of bytes written to buffer. When returned value is less than `btw`, there was no enough memory available to copy full data array

Parameters

- `[in] buff`: Buffer handle
- `[in] data`: Pointer to data to write into buffer
- `[in] btw`: Number of bytes to write

size_t **esp_buff_read**(*esp_buff_t *buff*, void *data, size_t *btr*)
Read data from buffer Copies data from buffer to `data` array and marks buffer as free for maximum `btr` number of bytes.

Return Number of bytes read and copied to data array

Parameters

- `[in] buff`: Buffer handle
- `[out] data`: Pointer to output memory to copy buffer data to
- `[in] btr`: Number of bytes to read

size_t **esp_buff_peek**(*esp_buff_t *buff*, size_t *skip_count*, void *data, size_t *btp*)
Read from buffer without changing read pointer (peek only)

Return Number of bytes peeked and written to output array

Parameters

- `[in] buff`: Buffer handle
- `[in] skip_count`: Number of bytes to skip before reading data
- `[out] data`: Pointer to output memory to copy buffer data to
- `[in] btp`: Number of bytes to peek

size_t **esp_buff_get_free**(*esp_buff_t *buff*)
Get number of bytes in buffer available to write.

Return Number of free bytes in memory

Parameters

- `[in] buff`: Buffer handle

size_t **esp_buff_get_full**(*esp_buff_t *buff*)
Get number of bytes in buffer available to read.

Return Number of bytes ready to be read

Parameters

- `[in] buff`: Buffer handle

void ***esp_buff_get_linear_block_read_address***(*esp_buff_t *buff*)
Get linear address for buffer for fast read.

Return Linear buffer start address

Parameters

- `[in] buff`: Buffer handle

size_t ***esp_buff_get_linear_block_read_length***(*esp_buff_t *buff*)
Get length of linear block address before it overflows for read operation.

Return Linear buffer size in units of bytes for read operation

Parameters

- `[in] buff`: Buffer handle

size_t ***esp_buff_skip***(*esp_buff_t *buff*, size_t *len*)
Skip (ignore; advance read pointer) buffer data Marks data as read in the buffer and increases free memory for up to `len` bytes.

Note Useful at the end of streaming transfer such as DMA

Return Number of bytes skipped

Parameters

- `[in] buff`: Buffer handle
- `[in] len`: Number of bytes to skip and mark as read

void ***esp_buff_get_linear_block_write_address***(*esp_buff_t *buff*)
Get linear address for buffer for fast read.

Return Linear buffer start address

Parameters

- `[in] buff`: Buffer handle

size_t ***esp_buff_get_linear_block_write_length***(*esp_buff_t *buff*)
Get length of linear block address before it overflows for write operation.

Return Linear buffer size in units of bytes for write operation

Parameters

- `[in] buff`: Buffer handle

size_t ***esp_buff_advance***(*esp_buff_t *buff*, size_t *len*)
Advance write pointer in the buffer. Similar to skip function but modifies write pointer instead of read.

Note Useful when hardware is writing to buffer and application needs to increase number of bytes written to buffer by hardware

Return Number of bytes advanced for write operation

Parameters

- `[in] buff`: Buffer handle
- `[in] len`: Number of bytes to advance

struct esp_buff_t

#include <esp_typedefs.h> Buffer structure.

Public Members

uint8_t ***buff**

Pointer to buffer data. Buffer is considered initialized when `buff != NULL`

size_t **size**

Size of buffer data. Size of actual buffer is `1` byte less than this value

size_t **r**

Next read pointer. Buffer is considered empty when `r == w` and full when `w == r - 1`

size_t **w**

Next write pointer. Buffer is considered empty when `r == w` and full when `w == r - 1`

Connections

Connections are essential feature of WiFi device and middleware. It is developed with strong focus on its performance and since it may interact with huge amount of data, it tries to use zero-copy (when available) feature, to decrease processing time.

ESP AT Firmware by default supports up to `5` connections being active at the same time and supports:

- Up to `5` TCP connections active at the same time
- Up to `5` UDP connections active at the same time
- Up to `1` SSL connection active at a time

Note: Client or server connections are available. Same API function call are used to send/receive data or close connection.

Architecture of the connection API is using callback event functions. This allows maximal optimization in terms of responsiveness on different kind of events.

Example below shows *bare minimum* implementation to:

- Start a new connection to remote host
- Send *HTTP GET* request to remote host
- Process received data in event and print number of received bytes

Listing 14: Client connection minimum example

```
1 #include "client.h"
2 #include "esp/esp.h"
3
```

(continues on next page)

(continued from previous page)

```
/* Host parameter */
#define CONN_HOST           "example.com"
#define CONN_PORT           80

static espr_t   conn_callback_func(esp_evt_t* evt);

/**
 * \brief           Request data for connection
 */
static const
uint8_t req_data[] = ""
"GET / HTTP/1.1\r\n"
"Host: " CONN_HOST "\r\n"
"Connection: close\r\n"
"\r\n";

/**
 * \brief           Start a new connection(s) as client
 */
void
client_connect(void) {
    espr_t res;

    /* Start a new connection as client in non-blocking mode */
    if ((res = esp_conn_start(NULL, ESP_CONN_TYPE_TCP, "example.com", 80, NULL, conn_
↪callback_func, 0)) == espOK) {
        printf("Connection to " CONN_HOST " started...\r\n");
    } else {
        printf("Cannot start connection to " CONN_HOST "!\r\n");
    }

    /* Start 2 more */
    esp_conn_start(NULL, ESP_CONN_TYPE_TCP, CONN_HOST, CONN_PORT, NULL, conn_callback_
↪func, 0);

    /*
     * An example of connection which should fail in connecting.
     * When this is the case, \ref ESP_EVT_CONN_ERROR event should be triggered
     * in callback function processing
     */
    esp_conn_start(NULL, ESP_CONN_TYPE_TCP, CONN_HOST, 10, NULL, conn_callback_func,␣
↪0);
}

/**
 * \brief           Event callback function for connection-only
 * \param[in]       evt: Event information with data
 * \return          \ref espOK on success, member of \ref espr_t otherwise
 */
static espr_t
conn_callback_func(esp_evt_t* evt) {
    esp_conn_p conn;
    espr_t res;
    uint8_t conn_num;

    conn = esp_conn_get_from_evt(evt);          /* Get connection handle from event */
    if (conn == NULL) {
```

(continues on next page)

(continued from previous page)

```
        return espERR;
    }
    conn_num = esp_conn_getnum(conn);           /* Get connection number for identification */
    switch (esp_evt_get_type(evt)) {
        case ESP_EVT_CONN_ACTIVE: {             /* Connection just active */
            printf("Connection %d active!\r\n", (int)conn_num);
            res = esp_conn_send(conn, req_data, sizeof(req_data) - 1, NULL, 0); /* Start sending data in non-blocking mode */
            if (res == espOK) {
                printf("Sending request data to server...\r\n");
            } else {
                printf("Cannot send request data to server. Closing connection manually...\r\n");
                esp_conn_close(conn, 0);        /* Close the connection */
            }
            break;
        }
        case ESP_EVT_CONN_CLOSE: {              /* Connection closed */
            if (esp_evt_conn_close_is_forced(evt)) {
                printf("Connection %d closed by client!\r\n", (int)conn_num);
            } else {
                printf("Connection %d closed by remote side!\r\n", (int)conn_num);
            }
            break;
        }
        case ESP_EVT_CONN_SEND: {               /* Data send event */
            espr_t res = esp_evt_conn_send_get_result(evt);
            if (res == espOK) {
                printf("Data sent successfully on connection %d...waiting to receive data from remote side...\r\n", (int)conn_num);
            } else {
                printf("Error while sending data on connection %d!\r\n", (int)conn_num);
            }
            break;
        }
        case ESP_EVT_CONN_RECV: {               /* Data received from remote side */
            esp_pbuf_p pbuf = esp_evt_conn_recv_get_buff(evt);
            esp_conn_recved(conn, pbuf);        /* Notify stack about received pbuf */
            printf("Received %d bytes on connection %d..\r\n", (int)esp_pbuf_length(pbuf, 1), (int)conn_num);
            break;
        }
        case ESP_EVT_CONN_ERROR: {              /* Error connecting to server */
            const char* host = esp_evt_conn_error_get_host(evt);
            esp_port_t port = esp_evt_conn_error_get_port(evt);
            printf("Error connecting to %s:%d\r\n", host, (int)port);
            break;
        }
        default: break;
    }
    return espOK;
}
```

Sending data

Receiving data flow is always the same. Whenever new data packet arrives, corresponding event is called to notify application layer. When it comes to sending data, application may decide between 2 options (*this is valid only for non-UDP connections):

- Write data to temporary transmit buffer
- Execute *send command* for every API function call

Temporary transmit buffer

By calling *esp_conn_write()* on active connection, temporary buffer is allocated and input data are copied to it. There is always up to 1 internal buffer active. When it is full (or if input data length is longer than maximal size), data are immediately send out and are not written to buffer.

ESP AT Firmware allows (current revision) to transmit up to 2048 bytes at a time with single command. When trying to send more than this, application would need to issue multiple *send commands* on *AT commands level*.

Write option is used mostly when application needs to write many different small chunks of data. Temporary buffer hence prevents many *send command* instructions as it is faster to send single command with big buffer, than many of them with smaller chunks of bytes.

Listing 15: Write data to connection output buffer

```
size_t rem_len;
esp_conn_p conn;
espr_t res;

/* ... other tasks to make sure connection is established */

/* We are connected to server at this point! */
/*
 * Call write function to write data to memory
 * and do not send immediately unless buffer is full after this write
 *
 * rem_len will give us response how much bytes
 * is available in memory after write
 */
res = esp_conn_write(conn, "My string", 9, 0, &rem_len);
if (rem_len == 0) {
    printf("No more memory available for next write!\r\n");
}
res = esp_conn_write(conn, "example.com", 11, 0, &rem_len);

/*
 * Data will stay in buffer until buffer is full,
 * except if user wants to force send,
 * call write function with flush mode enabled
 *
 * It will send out together 20 bytes
 */
esp_conn_write(conn, NULL, 0, 1, NULL);
```

Transmit packet manually

In some cases it is not possible to use temporary buffers, mostly because of memory constraints. Application can directly start *send data* instructions on *AT* level by using `esp_conn_send()` or `esp_conn_sendto()` functions.

group **ESP_CONN**

Connection API functions.

Typedefs

typedef struct esp_conn ***esp_conn_p**

Pointer to *esp_conn_t* structure.

Enums

enum esp_conn_type_t

List of possible connection types.

Values:

ESP_CONN_TYPE_TCP

Connection type is TCP

ESP_CONN_TYPE_UDP

Connection type is UDP

ESP_CONN_TYPE_SSL

Connection type is SSL

Functions

espr_t **esp_conn_start**(*esp_conn_p* **conn*, esp_conn_type_t *type*, **const** char ***const** *remote_host*, *esp_port_t* *remote_port*, void ***const** *arg*, *esp_evt_fn* *conn_evt_fn*, **const** uint32_t *blocking*)

Start a new connection of specific type.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[out] conn`: Pointer to connection handle to set new connection reference in case of successfully connected
- `[in] type`: Connection type. This parameter can be a value of *esp_conn_type_t* enumeration
- `[in] remote_host`: Connection host. In case of IP, write it as string, ex. "192.168.1.1"
- `[in] remote_port`: Connection port
- `[in] arg`: Pointer to user argument passed to connection if successfully connected
- `[in] conn_evt_fn`: Callback function for this connection
- `[in] blocking`: Status whether command should be blocking or not

espr_t **esp_conn_startex**(*esp_conn_p* **conn*, *esp_conn_start_t* **start_struct*, void ***const** *arg*, *esp_evt_fn* *conn_evt_fn*, **const** uint32_t *blocking*)

Start a new connection of specific type in extended mode.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[out] conn`: Pointer to connection handle to set new connection reference in case of successfully connected
- `[in] start_struct`: Connection information are handled by one giant structure
- `[in] arg`: Pointer to user argument passed to connection if successfully connected
- `[in] conn_evt_fn`: Callback function for this connection
- `[in] blocking`: Status whether command should be blocking or not

espr_t **esp_conn_close**(*esp_conn_p conn*, **const** uint32_t *blocking*)
Close specific or all connections.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] conn`: Connection handle to close. Set to NULL if you want to close all connections.
- `[in] blocking`: Status whether command should be blocking or not

espr_t **esp_conn_send**(*esp_conn_p conn*, **const** void **data*, size_t *btw*, size_t ***const** *bw*, **const** uint32_t *blocking*)
Send data on already active connection either as client or server.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] conn`: Connection handle to send data
- `[in] data`: Data to send
- `[in] btw`: Number of bytes to send
- `[out] bw`: Pointer to output variable to save number of sent data when successfully sent. Parameter value might not be accurate if you combine *esp_conn_write* and *esp_conn_send* functions
- `[in] blocking`: Status whether command should be blocking or not

espr_t **esp_conn_sendto**(*esp_conn_p conn*, **const** *esp_ip_t* ***const** *ip*, *esp_port_t port*, **const** void **data*, size_t *btw*, size_t **bw*, **const** uint32_t *blocking*)
Send data on active connection of type UDP to specific remote IP and port.

Note In case IP and port values are not set, it will behave as normal send function (suitable for TCP too)

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] conn`: Connection handle to send data
- `[in] ip`: Remote IP address for UDP connection
- `[in] port`: Remote port connection
- `[in] data`: Pointer to data to send
- `[in] btw`: Number of bytes to send
- `[out] bw`: Pointer to output variable to save number of sent data when successfully sent

- `[in] blocking`: Status whether command should be blocking or not

espr_t **esp_conn_set_arg**(*esp_conn_p conn*, void ***const** *arg*)
Set argument variable for connection.

Return *espOK* on success, member of *espr_t* enumeration otherwise

See *esp_conn_get_arg*

Parameters

- `[in] conn`: Connection handle to set argument
- `[in] arg`: Pointer to argument

void ***esp_conn_get_arg**(*esp_conn_p conn*)
Get user defined connection argument.

Return User argument

See *esp_conn_set_arg*

Parameters

- `[in] conn`: Connection handle to get argument

uint8_t **esp_conn_is_client**(*esp_conn_p conn*)
Check if connection type is client.

Return `1` on success, `0` otherwise

Parameters

- `[in] conn`: Pointer to connection to check for status

uint8_t **esp_conn_is_server**(*esp_conn_p conn*)
Check if connection type is server.

Return `1` on success, `0` otherwise

Parameters

- `[in] conn`: Pointer to connection to check for status

uint8_t **esp_conn_is_active**(*esp_conn_p conn*)
Check if connection is active.

Return `1` on success, `0` otherwise

Parameters

- `[in] conn`: Pointer to connection to check for status

uint8_t **esp_conn_is_closed**(*esp_conn_p conn*)
Check if connection is closed.

Return `1` on success, `0` otherwise

Parameters

- `[in] conn`: Pointer to connection to check for status

int8_t **esp_conn_getnum**(*esp_conn_p conn*)
Get the number from connection.

Return Connection number in case of success or -1 on failure

Parameters

- `[in] conn`: Connection pointer

espr_t **esp_conn_set_ssl_buffersize**(size_t *size*, **const** uint32_t *blocking*)
Set internal buffer size for SSL connection on ESP device.

Note Use this function before you start first SSL connection

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] size`: Size of buffer in units of bytes. Valid range is between 2048 and 4096 bytes
- `[in] blocking`: Status whether command should be blocking or not

espr_t **esp_get_conns_status**(**const** uint32_t *blocking*)
Gets connections status.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] blocking`: Status whether command should be blocking or not

esp_conn_p **esp_conn_get_from_evt**(*esp_evt_t* *evt)
Get connection from connection based event.

Return Connection pointer on success, `NULL` otherwise

Parameters

- `[in] evt`: Event which happened for connection

espr_t **esp_conn_write**(*esp_conn_p conn*, **const** void *data, size_t *btw*, uint8_t *flush*, size_t *****const** *mem_available*)
Write data to connection buffer and if it is full, send it non-blocking way.

Note This function may only be called from core (connection callbacks)

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] conn`: Connection to write
- `[in] data`: Data to copy to write buffer
- `[in] btw`: Number of bytes to write
- `[in] flush`: Flush flag. Set to `1` if you want to send data immediately after copying
- `[out] mem_available`: Available memory size available in current write buffer. When the buffer length is reached, current one is sent and a new one is automatically created. If function returns *espOK* and `*mem_available = 0`, there was a problem allocating a new buffer for next operation

espr_t **esp_conn_recved**(*esp_conn_p conn*, *esp_pbuf_p pbuf*)
Notify connection about received data which means connection is ready to accept more data.

Once data reception is confirmed, stack will try to send more data to user.

Note Since this feature is not supported yet by AT commands, function is only prototype and should be used in connection callback when data are received

Note Function is not thread safe and may only be called from connection event function

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] conn`: Connection handle
- `[in] pbuf`: Packet buffer received on connection

size_t **esp_conn_get_total_recved_count**(*esp_conn_p conn*)
Get total number of bytes ever received on connection and sent to user.

Return Total number of received bytes on connection

Parameters

- `[in] conn`: Connection handle

uint8_t **esp_conn_get_remote_ip**(*esp_conn_p conn*, *esp_ip_t *ip*)
Get connection remote IP address.

Return `1` on success, `0` otherwise

Parameters

- `[in] conn`: Connection handle
- `[out] ip`: Pointer to IP output handle

esp_port_t **esp_conn_get_remote_port**(*esp_conn_p conn*)
Get connection remote port number.

Return Port number on success, `0` otherwise

Parameters

- `[in] conn`: Connection handle

esp_port_t **esp_conn_get_local_port**(*esp_conn_p conn*)
Get connection local port number.

Return Port number on success, `0` otherwise

Parameters

- `[in] conn`: Connection handle

espr_t **esp_conn_ssl_configure**(uint8_t *link_id*, uint8_t *auth_mode*, uint8_t *pki_number*, uint8_t *ca_number*, **const** *esp_api_cmd_evt_fn evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)
Configure SSL parameters.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] link_id`: ID of the connection (0~max), for multiple connections, if the value is max, it means all connections. By default, max is *ESP_CFG_MAX_CONNS*.
- `[in] auth_mode`: Authentication mode `0`: no authorization `1`: load cert and private key for server authorization `2`: load CA for client authorize server cert and private key `3`: both authorization
- `[in] pki_number`: The index of cert and private key, if only one cert and private key, the value should be `0`.
- `[in] ca_number`: The index of CA, if only one CA, the value should be 0.
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

struct esp_conn_start_t

#include <esp_typedefs.h> Connection start structure, used to start the connection in extended mode.

Public Members

esp_conn_type_t **type**
Connection type

const char ***remote_host**
Host name or IP address in string format

esp_port_t **remote_port**
Remote server port

const char ***local_ip**
Local IP. Optional parameter, set to NULL if not used (most cases)

uint16_t **keep_alive**
Keep alive parameter for TCP/SSL connection in units of seconds. Value can be between `0 - 7200` where `0` means no keep alive

struct *esp_conn_start_t*::**[anonymous]**::**[anonymous]** **tcp_ssl**
TCP/SSL specific features

esp_port_t **local_port**
Custom local port for UDP

uint8_t **mode**
UDP mode. Set to `0` by default. Check ESP AT commands instruction set for more info when needed

struct *esp_conn_start_t*::**[anonymous]**::**[anonymous]** **udp**
UPD specific features

union *esp_conn_start_t*::**[anonymous]** **ext**
Extended support union

Debug support

Middleware has extended debugging capabilities. These consist of different debugging levels and types of debug messages, allowing to track and catch different types of warnings, severe problems or simply output messages program flow messages (trace messages).

Module is highly configurable using library configuration methods. Application must enable some options to decide what type of messages and for which modules it would like to output messages.

With default configuration, `printf` is used as output function. This behavior can be changed with `ESP_CF_DBG_OUT` configuration.

For successful debugging, application must:

- Enable global debugging by setting *ESP_CFG_DBG* to *ESP_DBG_ON*
- Configure which types of messages to output
- Configure debugging level, from all messages to severe only
- Enable specific modules to debug, by setting its configuration value to *ESP_DBG_ON*

Tip: Check *ESP Configuration* for all modules with debug implementation.

An example code with config and latter usage:

Listing 16: Debug configuration setup

```
/* Modifications of esp_config.h file for configuration */

/* Enable global debug */
#define ESP_CFG_DBG                 ESP_DBG_ON

/*
 * Enable debug types.
 * Application may use bitwise OR | to use multiple types:
 *    ESP_DBG_TYPE_TRACE | ESP_DBG_TYPE_STATE
 */
#define ESP_CFG_DBG_TYPES_ON        ESP_DBG_TYPE_TRACE

/* Enable debug on custom module */
#define MY_DBG_MODULE               ESP_DBG_ON
```

Listing 17: Debug usage within middleware

```
#include "esp/esp_debug.h"

/*
 * Print debug message to the screen
 * Trace message will be printed as it is enabled in types
 * while state message will not be printed.
 */
ESP_DEBUGF(MY_DBG_MODULE | ESP_DBG_TYPE_TRACE, "This is trace message on my program\r\
↪n");
ESP_DEBUGF(MY_DBG_MODULE | ESP_DBG_TYPE_STATE, "This is state message on my program\r\
↪n");
```

group **ESP_DEBUG**

Debug support module to track library flow.

Debug levels

List of debug levels

ESP_DBG_LVL_ALL
Print all messages of all types

ESP_DBG_LVL_WARNING
Print warning and upper messages

ESP_DBG_LVL_DANGER
Print danger errors

ESP_DBG_LVL_SEVERE
Print severe problems affecting program flow

ESP_DBG_LVL_MASK
Mask for getting debug level

Debug types

List of debug types

ESP_DBG_TYPE_TRACE
Debug trace messages for program flow

ESP_DBG_TYPE_STATE
Debug state messages (such as state machines)

ESP_DBG_TYPE_ALL
All debug types

Defines

ESP_DBG_ON
Indicates debug is enabled

ESP_DBG_OFF
Indicates debug is disabled

ESP_DEBUGF(c, fmt, ...)
Print message to the debug "window" if enabled.

Parameters

- `[in]` `c`: Condition if debug of specific type is enabled
- `[in]` `fmt`: Formatted string for debug
- `[in]` `...`: Variable parameters for formatted string

ESP_DEBUGW(c, cond, fmt, ...)
Print message to the debug "window" if enabled when specific condition is met.

Parameters

- `[in]` `c`: Condition if debug of specific type is enabled
- `[in]` `cond`: Debug only if this condition is true

- `[in] fmt`: Formatted string for debug
- `[in] ...`: Variable parameters for formatted string

Dynamic Host Configuration Protocol

group **ESP_DHCP**
DHCP config.

Functions

espr_t **esp_dhcp_configure**(uint8_t *sta*, uint8_t *ap*, uint8_t *en*, **const** *esp_api_cmd_evt_fn* *evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)
Configure DHCP settings for station or access point (or both)

Configuration changes will be saved in the NVS area of ESP device.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] sta`: Set to `1` to affect station DHCP configuration, set to `0` to keep current setup
- `[in] ap`: Set to `1` to affect access point DHCP configuration, set to `0` to keep current setup
- `[in] en`: Set to `1` to enable DHCP, or `0` to disable (static IP)
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

Domain Name System

group **ESP_DNS**
Domain name server.

Functions

espr_t **esp_dns_gethostbyname**(**const** char **host*, *esp_ip_t* ***const** *ip*, **const** *esp_api_cmd_evt_fn* *evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)
Get IP address from host name.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] host`: Pointer to host name to get IP for
- `[out] ip`: Pointer to *esp_ip_t* variable to save IP
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function

- `[in] blocking`: Status whether command should be blocking or not

espr_t **esp_dns_get_config**(*esp_ip_t* *s1*, *esp_ip_t* *s2*, **const** *esp_api_cmd_evt_fn* *evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)

Get the DNS server configuration.

Retrive configuration saved in the NVS area of ESP device.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[out] s1`: First server IP address in *esp_ip_t* format, set to 0.0.0.0 if not used
- `[out] s2`: Second server IP address in *esp_ip_t* format, set to to 0.0.0.0 if not used. Address `s1` cannot be the same as `s2`
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

espr_t **esp_dns_set_config**(uint8_t *en*, **const** char *s1*, **const** char *s2*, **const** *esp_api_cmd_evt_fn* *evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)

Enable or disable custom DNS server configuration.

Configuration changes will be saved in the NVS area of ESP device.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] en`: Set to `1` to enable, `0` to disable custom DNS configuration. When disabled, default DNS servers are used as proposed by ESP AT commands firmware
- `[in] s1`: First server IP address in string format, set to `NULL` if not used
- `[in] s2`: Second server IP address in string format, set to `NULL` if not used. Address `s1` cannot be the same as `s2`
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

Event management

group **ESP_EVT**

Event helper functions.

Reset detected

Event helper functions for ESP_EVT_RESET_DETECTED event

uint8_t **esp_evt_reset_detected_is_forced**(*esp_evt_t* *cc)
Check if detected reset was forced by user.

Return 1 if forced by user, 0 otherwise

Parameters

- [in] cc: Event handle

Reset event

Event helper functions for ESP_EVT_RESET event

espr_t **esp_evt_reset_get_result**(*esp_evt_t* *cc)
Get reset sequence operation status.

Return Member of *espr_t* enumeration

Parameters

- [in] cc: Event data

Restore event

Event helper functions for ESP_EVT_RESTORE event

espr_t **esp_evt_restore_get_result**(*esp_evt_t* *cc)
Get restore sequence operation status.

Return Member of *espr_t* enumeration

Parameters

- [in] cc: Event data

Access point or station IP or MAC

Event helper functions for ESP_EVT_AP_IP_STA event

esp_mac_t ***esp_evt_ap_ip_sta_get_mac**(*esp_evt_t* *cc)
Get MAC address from station.

Return MAC address

Parameters

- [in] cc: Event handle

esp_ip_t ***esp_evt_ap_ip_sta_get_ip**(*esp_evt_t* *cc)
Get IP address from station.

Return IP address

Parameters

- `[in] cc`: Event handle

Connected station to access point

Event helper functions for ESP_EVT_AP_CONNECTED_STA event

esp_mac_t ***esp_evt_ap_connected_sta_get_mac**(*esp_evt_t* *cc)*
Get MAC address from connected station.

Return MAC address

Parameters

- `[in] cc`: Event handle

Disconnected station from access point

Event helper functions for ESP_EVT_AP_DISCONNECTED_STA event

esp_mac_t ***esp_evt_ap_disconnected_sta_get_mac**(*esp_evt_t* *cc)*
Get MAC address from disconnected station.

Return MAC address

Parameters

- `[in] cc`: Event handle

Connection data received

Event helper functions for ESP_EVT_CONN_RECV event

esp_pbuf_p **esp_evt_conn_recv_get_buff**(*esp_evt_t* *cc)
Get buffer from received data.

Return Buffer handle

Parameters

- `[in] cc`: Event handle

esp_conn_p **esp_evt_conn_recv_get_conn**(*esp_evt_t* *cc)
Get connection handle for receive.

Return Connection handle

Parameters

- `[in] cc`: Event handle

Connection data send

Event helper functions for ESP_EVT_CONN_SEND event

esp_conn_p **esp_evt_conn_send_get_conn**(*esp_evt_t* *cc)
Get connection handle for data sent event.

Return Connection handle

Parameters

- `[in] cc`: Event handle

size_t **esp_evt_conn_send_get_length**(*esp_evt_t* *cc)
Get number of bytes sent on connection.

Return Number of bytes sent

Parameters

- `[in] cc`: Event handle

espr_t **esp_evt_conn_send_get_result**(*esp_evt_t* *cc)
Check if connection send was successful.

Return Member of *espr_t* enumeration

Parameters

- `[in] cc`: Event handle

Connection active

Event helper functions for ESP_EVT_CONN_ACTIVE event

esp_conn_p **esp_evt_conn_active_get_conn**(*esp_evt_t* *cc)
Get connection handle.

Return Connection handle

Parameters

- `[in] cc`: Event handle

uint8_t **esp_evt_conn_active_is_client**(*esp_evt_t* *cc)
Check if new connection is client.

Return `1` if client, `0` otherwise

Parameters

- `[in] cc`: Event handle

Connection close event

Event helper functions for ESP_EVT_CONN_CLOSE event

esp_conn_p **esp_evt_conn_close_get_conn**(*esp_evt_t* *cc)
Get connection handle.

Return Connection handle

Parameters

- `[in]` `cc`: Event handle

uint8_t **esp_evt_conn_close_is_client**(*esp_evt_t* *cc)
Check if just closed connection was client.

Return `1` if client, `0` otherwise

Parameters

- `[in]` `cc`: Event handle

uint8_t **esp_evt_conn_close_is_forced**(*esp_evt_t* *cc)
Check if connection close even was forced by user.

Return `1` if forced, `0` otherwise

Parameters

- `[in]` `cc`: Event handle

espr_t **esp_evt_conn_close_get_result**(*esp_evt_t* *cc)
Get connection close event result.

Return Member of *espr_t* enumeration

Parameters

- `[in]` `cc`: Event handle

Connection poll

Event helper functions for ESP_EVT_CONN_POLL event

esp_conn_p **esp_evt_conn_poll_get_conn**(*esp_evt_t* *cc)
Get connection handle.

Return Connection handle

Parameters

- `[in]` `cc`: Event handle

Connection error

Event helper functions for ESP_EVT_CONN_ERROR event

espr_t **esp_evt_conn_error_get_error**(*esp_evt_t* *cc)
Get connection error type.

Return Member of *espr_t* enumeration

Parameters

- `[in] cc`: Event handle

esp_conn_type_t **esp_evt_conn_error_get_type**(*esp_evt_t* *cc)
Get connection type.

Return Member of *espr_t* enumeration

Parameters

- `[in] cc`: Event handle

const char ***esp_evt_conn_error_get_host**(*esp_evt_t* *cc)
Get connection host.

Return Host name for connection

Parameters

- `[in] cc`: Event handle

esp_port_t **esp_evt_conn_error_get_port**(*esp_evt_t* *cc)
Get connection port.

Return Host port number

Parameters

- `[in] cc`: Event handle

void ***esp_evt_conn_error_get_arg**(*esp_evt_t* *cc)
Get user argument.

Return User argument

Parameters

- `[in] cc`: Event handle

List access points

Event helper functions for ESP_EVT_STA_LIST_AP event

espr_t **esp_evt_sta_list_ap_get_result**(*esp_evt_t* *cc)
Get command success result.

Return Member of *espr_t* enumeration

Parameters

- `[in] cc`: Event handle

esp_ap_t ***esp_evt_sta_list_ap_get_aps**(*esp_evt_t* *cc)
Get access points.

Return Pointer to *esp_ap_t* with first access point description

Parameters

- `[in] cc`: Event handle

size_t **esp_evt_sta_list_ap_get_length**(*esp_evt_t* *cc)
Get number of access points found.

Return Number of access points found

Parameters

- `[in] cc`: Event handle

Join access point

Event helper functions for ESP_EVT_STA_JOIN_AP event

espr_t **esp_evt_sta_join_ap_get_result**(*esp_evt_t* *cc)
Get command success result.

Return Member of *espr_t* enumeration

Parameters

- `[in] cc`: Event handle

Get access point info

Event helper functions for ESP_EVT_STA_INFO_AP event

espr_t **esp_evt_sta_info_ap_get_result**(*esp_evt_t* *cc)
Get command result.

Return Member of *espr_t* enumeration

Parameters

- `[in] cc`: Event handle

const char ***esp_evt_sta_info_ap_get_ssid**(*esp_evt_t* *cc)
Get current AP name.

Return AP name

Parameters

- `[in] cc`: Event handle

esp_mac_t **esp_evt_sta_info_ap_get_mac**(*esp_evt_t* *cc)
Get current AP MAC address.

Return AP MAC address

Parameters

- `[in] cc`: Event handle

uint8_t **esp_evt_sta_info_ap_get_channel**(*esp_evt_t* *cc)
Get current AP channel.

Return AP channel

Parameters

- `[in] cc`: Event handle

int16_t **esp_evt_sta_info_ap_get_rssi**(*esp_evt_t* *cc)
Get current AP rssi.

Return AP rssi

Parameters

- `[in] cc`: Event handle

Get host address by name

Event helper functions for ESP_EVT_DNS_HOSTBYNAME event

espr_t **esp_evt_dns_hostbyname_get_result**(*esp_evt_t* *cc)
Get resolve result.

Return Member of *espr_t* enumeration

Parameters

- `[in] cc`: Event handle

const char ***esp_evt_dns_hostbyname_get_host**(*esp_evt_t* *cc)
Get hostname used to resolve IP address.

Return Hostname

Parameters

- `[in] cc`: Event handle

esp_ip_t ***esp_evt_dns_hostbyname_get_ip**(*esp_evt_t* *cc)
Get IP address from DNS function.

Return IP address

Parameters

- `[in] cc`: Event handle

Ping

Event helper functions for ESP_EVT_PING event

espr_t **esp_evt_ping_get_result**(*esp_evt_t* *cc)
Get ping status.

Return Member of *espr_t* enumeration

Parameters

- `[in] cc`: Event handle

const char ***esp_evt_ping_get_host**(*esp_evt_t* *cc)
Get hostname used to ping.

Return Hostname

Parameters

- `[in] cc`: Event handle

uint32_t **esp_evt_ping_get_time**(*esp_evt_t* *cc)
Get time required for ping.

Return Ping time

Parameters

- `[in] cc`: Event handle

Server

Event helper functions for ESP_EVT_SERVER event

espr_t **esp_evt_server_get_result**(*esp_evt_t* *cc)
Get server command result.

Return Member of *espr_t* enumeration

Parameters

- `[in] cc`: Event handle

esp_port_t **esp_evt_server_get_port**(*esp_evt_t* *cc)
Get port for server operation.

Return Server port

Parameters

- `[in] cc`: Event handle

uint8_t **esp_evt_server_is_enable**(*esp_evt_t* *cc)
Check if operation was to enable or disable server.

Return `1` if enable, `0` otherwise

Parameters

- `[in] cc`: Event handle

Typedefs

typedef espr_t (*__esp_evt_fn__)(**struct** esp_evt *evt)
Event function prototype.

Return *espOK* on success, member of *espr_t* otherwise

Parameters

- `[in] evt`: Callback event data

Enums

enum esp_evt_type_t
List of possible callback types received to user.

Values:

ESP_EVT_INIT_FINISH
Initialization has been finished at this point

ESP_EVT_RESET_DETECTED
Device reset detected

ESP_EVT_RESET
Device reset operation finished

ESP_EVT_RESTORE
Device restore operation finished

ESP_EVT_CMD_TIMEOUT
Timeout on command. When application receives this event, it may reset system as there was (maybe) a problem in device

ESP_EVT_DEVICE_PRESENT
Notification when device present status changes

ESP_EVT_AT_VERSION_NOT_SUPPORTED
Library does not support firmware version on ESP device.

ESP_EVT_CONN_RECV
Connection data received

ESP_EVT_CONN_SEND
Connection data send

ESP_EVT_CONN_ACTIVE
Connection just became active

ESP_EVT_CONN_ERROR
Client connection start was not successful

ESP_EVT_CONN_CLOSE
Connection close event. Check status if successful

ESP_EVT_CONN_POLL
Poll for connection if there are any changes

ESP_EVT_SERVER
Server status changed

ESP_EVT_WIFI_CONNECTED
Station just connected to AP

ESP_EVT_WIFI_GOT_IP
Station has valid IP. When this event is received to application, no IP has been read from device. Stack will proceed with IP read from device and will later send *ESP_EVT_WIFI_IP_ACQUIRED* event

ESP_EVT_WIFI_DISCONNECTED
Station just disconnected from AP

ESP_EVT_WIFI_IP_ACQUIRED
Station IP address acquired. At this point, valid IP address has been received from device. Application may use *esp_sta_copy_ip* function to read it

ESP_EVT_STA_LIST_AP
Station listed APs event

ESP_EVT_STA_JOIN_AP
Join to access point

ESP_EVT_STA_INFO_AP
Station AP info (name, mac, channel, rssi)

ESP_EVT_AP_CONNECTED_STA
New station just connected to ESP's access point

ESP_EVT_AP_DISCONNECTED_STA
New station just disconnected from ESP's access point

ESP_EVT_AP_IP_STA
New station just received IP from ESP's access point

ESP_EVT_DNS_HOSTBYNAME
DNS domain service finished

ESP_EVT_PING
PING service finished

Functions

espr_t **esp_evt_register**(*esp_evt_fn fn*)
Register event function for global (non-connection based) events.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] fn`: Callback function to call on specific event

espr_t **esp_evt_unregister**(*esp_evt_fn fn*)
Unregister callback function for global (non-connection based) events.

Note Function must be first registered using *esp_evt_register*

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] fn`: Callback function to remove from event list

esp_evt_type_t **esp_evt_get_type**(*esp_evt_t* *cc)
Get event type.

Return Event type. Member of *esp_evt_type_t* enumeration

Parameters

- `[in] cc`: Event handle

struct esp_evt_t
#include <*esp_typedefs.h*> Global callback structure to pass as parameter to callback function.

Public Members

esp_evt_type_t **type**
Callback type

uint8_t **forced**
Set to `1` if reset forced by user

Set to 1 if connection action was forced (when active: 1 = CLIENT, 0 = SERVER: when closed, 1 = CMD, 0 = REMOTE)

struct *esp_evt_t*::**[anonymous]**::**[anonymous] reset_detected**
Reset occurred. Use with ESP_EVT_RESET_DETECTED event

espr_t **res**
Reset operation result

Restore operation result

Send data result

Result of close event. Set to *espOK* on success

Status of command

Result of command

struct *esp_evt_t*::**[anonymous]**::**[anonymous] reset**
Reset sequence finish. Use with ESP_EVT_RESET event

struct *esp_evt_t*::**[anonymous]**::**[anonymous] restore**
Restore sequence finish. Use with ESP_EVT_RESTORE event

esp_conn_p **conn**
Connection where data were received

Connection where data were sent

Pointer to connection

Set connection pointer

esp_pbuf_p **buff**
Pointer to received data

struct *esp_evt_t*::**[anonymous]**::**[anonymous] conn_data_recv**
Network data received. Use with ESP_EVT_CONN_RECV event

size_t **sent**
Number of bytes sent on connection

struct *esp_evt_t*::**[anonymous]**::**[anonymous] conn_data_send**
Data send. Use with ESP_EVT_CONN_SEND event

const char ***host**
Host to use for connection

Host name for DNS lookup

Host name for ping

esp_port_t **port**
Remote port used for connection

Server port number

esp_conn_type_t **type**
Connection type

void ***arg**
Connection user argument

espr_t **err**
Error value

struct *esp_evt_t*::**[anonymous]**::**[anonymous] conn_error**
Client connection start error. Use with ESP_EVT_CONN_ERROR event

uint8_t **client**
Set to 1 if connection is/was client mode

struct *esp_evt_t*::**[anonymous]**::**[anonymous] conn_active_close**
Process active and closed statuses at the same time. Use with ESP_EVT_CONN_ACTIVE or ESP_EVT_CONN_CLOSE events

struct *esp_evt_t*::**[anonymous]**::**[anonymous] conn_poll**
Polling active connection to check for timeouts. Use with ESP_EVT_CONN_POLL event

uint8_t **en**
Status to enable/disable server

struct *esp_evt_t*::**[anonymous]**::**[anonymous] server**
Server change event. Use with ESP_EVT_SERVER event

esp_ap_t ***aps**
Pointer to access points

size_t **len**
Number of access points found

struct *esp_evt_t*::**[anonymous]**::**[anonymous] sta_list_ap**
Station list access points. Use with ESP_EVT_STA_LIST_AP event

struct *esp_evt_t*::**[anonymous]**::**[anonymous] sta_join_ap**
Join to access point. Use with ESP_EVT_STA_JOIN_AP event

esp_sta_info_ap_t ***info**
AP info of current station

struct *esp_evt_t*::**[anonymous]**::**[anonymous] sta_info_ap**
Current AP informations. Use with ESP_EVT_STA_INFO_AP event

esp_mac_t ***mac**
Station MAC address

struct *esp_evt_t*::**[anonymous]**::**[anonymous] ap_conn_disconn_sta**
A new station connected or disconnected to ESP's access point. Use with ESP_EVT_AP_CONNECTED_STA or ESP_EVT_AP_DISCONNECTED_STA events

esp_ip_t ***ip**
Station IP address

Pointer to IP result

struct *esp_evt_t*::**[anonymous]**::**[anonymous] ap_ip_sta**
Station got IP address from ESP's access point. Use with ESP_EVT_AP_IP_STA event

struct *esp_evt_t*::**[anonymous]**::**[anonymous] dns_hostbyname**
DNS domain service finished. Use with ESP_EVT_DNS_HOSTBYNAME event

uint32_t **time**
Time required for ping. Valid only if operation succedded

struct *esp_evt_t*::**[anonymous]**::**[anonymous] ping**
Ping finished. Use with ESP_EVT_PING event

union *esp_evt_t*::**[anonymous] evt**
Callback event union

Hostname

group **ESP_HOSTNAME**
Hostname API.

Functions

espr_t **esp_hostname_set**(**const** char **hostname*, **const** *esp_api_cmd_evt_fn* *evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)
Set hostname of WiFi station.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] hostname`: Name of ESP host
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

espr_t **esp_hostname_get**(char **hostname*, size_t *size*, **const** *esp_api_cmd_evt_fn* *evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)
Get hostname of WiFi station.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] hostname`: Pointer to output variable holding memory to save hostname

- [in] size: Size of buffer for hostname. Size includes memory for NULL termination
- [in] evt_fn: Callback function called when command has finished. Set to NULL when not used
- [in] evt_arg: Custom argument for event callback function
- [in] blocking: Status whether command should be blocking or not

Input module

Input module is used to input received data from *ESP* device to *ESP-AT-Lib* middleware part. 2 processing options are possible:

- Indirect processing with *esp_input()* (default mode)
- Direct processing with *esp_input_process()*

Tip: Direct or indirect processing mode is select by setting *ESP_CFG_INPUT_USE_PROCESS* configuration value.

Indirect processing

With indirect processing mode, every received character from *ESP* physical device is written to intermediate buffer between low-level driver and *processing* thread.

Function *esp_input()* is used to write data to buffer, which is later processed by *processing* thread.

Indirect processing mode allows embedded systems to write received data to buffer from interrupt context (outside threads). As a drawback, its performance is decreased as it involves copying every receive character to intermediate buffer, and may also introduce RAM memory footprint increase.

Direct processing

Direct processing is targeting more advanced host controllers, like STM32 or WIN32 implementation use. It is developed with DMA support in mind, allowing low-level drivers to skip intermediate data buffer and process input bytes directly.

Note: When using this mode, function *esp_input_process()* must be used and it may only be called from thread context. Processing of input bytes is done in low-level input thread, started by application.

Tip: Check *Porting guide* for implementation examples.

group **ESP_INPUT**

Input function for received data.

Functions

espr_t **esp_input**(**const** void *_data_, size_t _len_)
Write data to input buffer.

Note _ESP_CFG_INPUT_USE_PROCESS_ must be disabled to use this function

Return _espOK_ on success, member of _espr_t_ enumeration otherwise

Parameters

- `[in]` `data`: Pointer to data to write
- `[in]` `len`: Number of data elements in units of bytes

espr_t **esp_input_process**(**const** void *_data_, size_t _len_)
Process input data directly without writing it to input buffer.

Note This function may only be used when in OS mode, where single thread is dedicated for input read of AT receive

Note _ESP_CFG_INPUT_USE_PROCESS_ must be enabled to use this function

Return _espOK_ on success, member of _espr_t_ enumeration otherwise

Parameters

- `[in]` `data`: Pointer to received data to be processed
- `[in]` `len`: Length of data to process in units of bytes

Multicast DNS

group **ESP_MDNS**
mDNS function

Functions

espr_t **esp_mdns_configure**(uint8_t _en_, **const** char *_host_, **const** char *_server_, _esp_port_t_ _port_, **const** _esp_api_cmd_evt_fn_ _evt_fn_, void ***const** _evt_arg_, **const** uint32_t _blocking_)
Configure mDNS parameters with hostname and server.

Return _espOK_ on success, member of _espr_t_ enumeration otherwise

Parameters

- `[in]` `en`: Status to enable `1` or disable `0` mDNS function
- `[in]` `host`: mDNS host name
- `[in]` `server`: mDNS server name
- `[in]` `port`: mDNS server port number
- `[in]` `evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in]` `evt_arg`: Custom argument for event callback function
- `[in]` `blocking`: Status whether command should be blocking or not

Memory manager

group **ESP_MEM**
Dynamic memory manager.

Functions

uint8_t **esp_mem_assignmemory**(**const** *esp_mem_region_t* *regions, size_t *size*)
Assign memory region(s) for allocation functions.

Note You can allocate multiple regions by assigning start address and region size in units of bytes

Return `1` on success, `0` otherwise

Note Function is not available when *ESP_CFG_MEM_CUSTOM* is `1`

Parameters

- `[in] regions`: Pointer to list of regions to use for allocations
- `[in] len`: Number of regions to use

void ***esp_mem_malloc**(size_t *size*)
Allocate memory of specific size.

Return Memory address on success, `NULL` otherwise

Note Function is not available when *ESP_CFG_MEM_CUSTOM* is `1` and must be implemented by user

Parameters

- `[in] size`: Number of bytes to allocate

void ***esp_mem_realloc**(void *ptr, size_t *size*)
Reallocate memory to specific size.

Note After new memory is allocated, content of old one is copied to new memory

Return Memory address on success, `NULL` otherwise

Note Function is not available when *ESP_CFG_MEM_CUSTOM* is `1` and must be implemented by user

Parameters

- `[in] ptr`: Pointer to current allocated memory to resize, returned using *esp_mem_malloc*, *esp_mem_calloc* or *esp_mem_realloc* functions
- `[in] size`: Number of bytes to allocate on new memory

void ***esp_mem_calloc**(size_t *num*, size_t *size*)
Allocate memory of specific size and set memory to zero.

Return Memory address on success, `NULL` otherwise

Note Function is not available when *ESP_CFG_MEM_CUSTOM* is `1` and must be implemented by user

Parameters

- `[in] num`: Number of elements to allocate
- `[in] size`: Size of each element

void **esp_mem_free**(void *_ptr_)
Free memory.

Note Function is not available when *ESP_CFG_MEM_CUSTOM* is `1` and must be implemented by user

Parameters

- `[in] ptr`: Pointer to memory previously returned using *esp_mem_malloc*, *esp_mem_calloc* or *esp_mem_realloc* functions

uint8_t **esp_mem_free_s**(void **_ptr_)
Free memory in safe way by invalidating pointer after freeing.

Return `1` on success, `0` otherwise

Parameters

- `[in] ptr`: Pointer to pointer to allocated memory to free

struct esp_mem_region_t
#include <*esp_mem.h*> Single memory region descriptor.

Public Members

void ***start_addr**
Start address of region

size_t **size**
Size in units of bytes of region

Packet buffer

Packet buffer (or *pbuf*) is buffer manager to handle received data from any connection. It is optimized to construct big buffer of smaller chunks of fragmented data as received bytes are not always coming as single packet.

Pbuf block diagram

Fig. 4: Block diagram of pbuf chain

Image above shows structure of *pbuf* chain. Each *pbuf* consists of:

- Pointer to next *pbuf*, or `NULL` when it is last in chain
- Length of current packet length
- Length of current packet and all next in chain
  - If *pbuf* is last in chain, total length is the same as current packet length
- Reference counter, indicating how many pointers point to current *pbuf*
- Actual buffer data

Top image shows `3` pbufs connected to single chain. There are `2` custom pointer variables to point at different *pbuf* structures. Second *pbuf* has reference counter set to `2`, as `2` variables point to it:

- *next* of *pbuf 1* is the first one

- *User variable 2* is the second one

Table 1: Block structure

Block number	Next pbuf	Block size	Total size in chain	Reference counter
Block 1	*Block 2*	150	550	1
Block 2	*Block 3*	130	400	2
Block 3	NULL	270	270	1

Reference counter

Reference counter holds number of references (or variables) pointing to this block. It is used to properly handle memory free operation, especially when *pbuf* is used by lib core and application layer.

Note: If there would be no reference counter information and application would free memory while another part of library still uses its reference, application would invoke *undefined behavior* and system could crash instantly.

When application tries to free pbuf chain as on first image, it would normally call `esp_pbuf_free()` function. That would:

- Decrease reference counter by 1
- If reference counter == 0, it removes it from chain list and frees packet buffer memory
- If reference counter != 0 after decrease, it stops free procedure
- Go to next pbuf in chain and repeat steps

As per first example, result of freeing from *user variable 1* would look similar to image and table below. First block (blue) had reference counter set to 1 prior freeing operation. It was successfully removed as *user variable 1* was the only one pointing to it, while second (green) block had reference counter set to 2, preventing free operation.

Fig. 5: Block diagram of pbuf chain after free from *user variable 1*

Table 2: Block diagram of pbuf chain after free from *user variable 1*

Block number	Next pbuf	Block size	Total size in chain	Reference counter
Block 2	*Block 3*	130	400	1
Block 3	NULL	270	270	1

Note: *Block 1* has been successfully freed, but since *block 2* had reference counter set to 2 before, it was only decreased by 1 to a new value 1 and free operation stopped instead. *User variable 2* is still using *pbuf* starting at *block 2* and must manually call `esp_pbuf_free()` to free it.

Concatenating vs chaining

This section will explain difference between *concat* and *chain* operations. Both operations link 2 pbufs together in a chain of pbufs, difference is that *chain* operation increases *reference counter* to linked pbuf, while *concat* keeps *reference counter* at its current status.

Fig. 6: Different pbufs, each pointed to by its own variable

Concat operation

Concat operation shall be used when 2 pbufs are linked together and reference to *second* is no longer used.

Fig. 7: Structure after pbuf concat

After concating *2 pbufs* together, reference counter of second is still set to 1, however we can see that 2 pointers point to *second pbuf*.

Note: After application calls *esp_pbuf_cat()*, it must not use pointer which points to *second pbuf*. This would invoke *undefined behavior* if one pointer tries to free memory while second still points to it.

An example code showing proper usage of concat operation:

Listing 18: Packet buffer concat example

```
esp_pbuf_p a, b;

/* Create 2 pbufs of different sizes */
a = esp_pbuf_new(10);
b = esp_pbuf_new(20);

/* Link them together with concat operation */
/* Reference on b will stay as is, won't be increased */
esp_pbuf_cat(a, b);

/*
 * Operating with b variable has from now on undefined behavior,
 * application shall stop using variable b to access pbuf.
 *
 * The best way would be to set b reference to NULL
 */
b = NULL;

/*
 * When application doesn't need pbufs anymore,
 * free a and it will also free b
 */
esp_pbuf_free(a);
```

Chain operation

Chain operation shall be used when 2 pbufs are linked together and reference to *second* is still required.

Fig. 8: Structure after pbuf chain

After chainin *2 pbufs* together, reference counter of second is increased by 1, which allows application to reference second *pbuf* separatelly.

Note: After application calls *esp_pbuf_chain()*, it also has to manually free its reference using *esp_pbuf_free()* function. Forgetting to free pbuf invokes memory leak

An example code showing proper usage of chain operation:

Listing 19: Packet buffer chain example

```
esp_pbuf_p a, b;

/* Create 2 pbufs of different sizes */
a = esp_pbuf_new(10);
b = esp_pbuf_new(20);

/* Chain both pbufs together */
/* This will increase reference on b as 2 variables now point to it */
esp_pbuf_chain(a, b);

/*
 * When application does not need a anymore, it may free it

 * This will free only pbuf a, as pbuf b has now 2 references:
 *  - one from pbuf a
 *  - one from variable b
 */

/* If application calls this, it will free only first pbuf */
/* As there is link to b pbuf somewhere */
esp_pbuf_free(a);

/* Reset a variable, not used anymore */
a = NULL;

/*
 * At this point, b is still valid memory block,
 * but when application doesn't need it anymore,
 * it should free it, otherwise memory leak appears
 */
esp_pbuf_free(b);

/* Reset b variable */
b = NULL;
```

Extract pbuf data

Each *pbuf* holds some amount of data bytes. When multiple *pbufs* are linked together (either chained or concated), blocks of raw data are not linked to contiguous memory block. It is necessary to process block by block manually.

An example code showing proper reading of any *pbuf*:

Listing 20: Packet buffer data extraction

```
const void* data;
size_t pos, len;
esp_pbuf_p a, b, c;

const char str_a[] = "This is one long";
const char str_a[] = "string. We want to save";
const char str_a[] = "chain of pbufs to file";

/* Create pbufs to hold these strings */
a = esp_pbuf_new(strlen(str_a));
b = esp_pbuf_new(strlen(str_b));
c = esp_pbuf_new(strlen(str_c));

/* Write data to pbufs */
esp_pbuf_take(a, str_a, strlen(str_a), 0);
esp_pbuf_take(b, str_b, strlen(str_b), 0);
esp_pbuf_take(c, str_c, strlen(str_c), 0);

/* Connect pbufs together */
esp_pbuf_chain(a, b);
esp_pbuf_chain(a, c);

/*
 * pbuf a now contains chain of b and c together
 * and at this point application wants to print (or save) data from chained pbuf
 *
 * Process pbuf by pbuf with code below
 */

/*
 * Get linear address of current pbuf at specific offset
 * Function will return pointer to memory address at specific position
 * and `len` will hold length of data block
 */
pos = 0;
while ((data = esp_pbuf_get_linear_addr(a, pos, &len)) != NULL) {
    /* Custom process function... */
    /* Process data with data pointer and block length */
    process_data(data, len);
    printf("Str: %.*s", len, data);

    /* Increase offset position for next block */
    pos += len;
}

/* Call free only on a pbuf. Since it is chained, b and c will be freed too */
esp_pbuf_free(a);
```

group **ESP_PBUF**

Packet buffer manager.

Typedefs

typedef struct esp_pbuf ***esp_pbuf_p**
Pointer to *esp_pbuf_t* structure.

Functions

esp_pbuf_p **esp_pbuf_new**(size_t *len*)
Allocate packet buffer for network data of specific size.

Return Pointer to allocated memory, `NULL` otherwise

Parameters

- `[in] len`: Length of payload memory to allocate

size_t **esp_pbuf_free**(*esp_pbuf_p pbuf*)
Free previously allocated packet buffer.

Return Number of freed pbufs from head

Parameters

- `[in] pbuf`: Packet buffer to free

void ***esp_pbuf_data**(**const** *esp_pbuf_p pbuf*)
Get data pointer from packet buffer.

Return Pointer to data buffer on success, `NULL` otherwise

Parameters

- `[in] pbuf`: Packet buffer

size_t **esp_pbuf_length**(**const** *esp_pbuf_p pbuf*, uint8_t *tot*)
Get length of packet buffer.

Return Length of data in units of bytes

Parameters

- `[in] pbuf`: Packet buffer to get length for
- `[in] tot`: Set to `1` to return total packet chain length or `0` to get only first packet length

uint8_t **esp_pbuf_set_length**(*esp_pbuf_p pbuf*, size_t *new_len*)
Set new length of pbuf.

Note New length can only be smaller than existing one. It has no effect when greater than existing one

Note This function can be used on single-chain pbufs only, without `next` pbuf in chain

Return `1` on success, `0` otherwise

Parameters

- `[in] pbuf`: Pbuf to make it smaller

- `[in] new_len`: New length in units of bytes

espr_t **esp_pbuf_take**(*esp_pbuf_p pbuf*, **const** void **data*, size_t *len*, size_t *offset*)
Copy user data to chain of pbufs.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] pbuf`: First pbuf in chain to start copying to
- `[in] data`: Input data to copy to pbuf memory
- `[in] len`: Length of input data to copy
- `[in] offset`: Start offset in pbuf where to start copying

size_t **esp_pbuf_copy**(*esp_pbuf_p pbuf*, void **data*, size_t *len*, size_t *offset*)
Copy memory from pbuf to user linear memory.

Return Number of bytes copied

Parameters

- `[in] pbuf`: Pbuf to copy from
- `[out] data`: User linear memory to copy to
- `[in] len`: Length of data in units of bytes
- `[in] offset`: Possible start offset in pbuf

espr_t **esp_pbuf_cat**(*esp_pbuf_p head*, **const** *esp_pbuf_p tail*)
Concatenate `2` packet buffers together to one big packet.

Note After `tail` pbuf has been added to `head` pbuf chain, it must not be referenced by user anymore as it is now completelly controlled by `head` pbuf. In simple words, when user calls this function, it should not call *esp_pbuf_free* function anymore, as it might make memory undefined for `head` pbuf.

Return *espOK* on success, member of *espr_t* enumeration otherwise

See *esp_pbuf_chain*

Parameters

- `[in] head`: Head packet buffer to append new pbuf to
- `[in] tail`: Tail packet buffer to append to head pbuf

espr_t **esp_pbuf_chain**(*esp_pbuf_p head*, *esp_pbuf_p tail*)
Chain 2 pbufs together. Similar to *esp_pbuf_cat* but now new reference is done from head pbuf to tail pbuf.

Note After this function call, user must call *esp_pbuf_free* to remove its reference to tail pbuf and allow control to head pbuf: esp_pbuf_free(tail)

Return *espOK* on success, member of *espr_t* enumeration otherwise

See *esp_pbuf_cat*

Parameters

- `[in] head`: Head packet buffer to append new pbuf to
- `[in] tail`: Tail packet buffer to append to head pbuf

esp_pbuf_p **esp_pbuf_unchain**(*esp_pbuf_p head*)
Unchain first pbuf from list and return second one.

`tot_len` and `len` fields are adjusted to reflect new values and reference counter is `as is`

Note After unchain, user must take care of both pbufs (`head` and `new returned one`)

Return Next pbuf after `head`

Parameters

- `[in] head`: First pbuf in chain to remove from chain

espr_t **esp_pbuf_ref**(*esp_pbuf_p pbuf*)
Increment reference count on pbuf.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] pbuf`: pbuf to increase reference

uint8_t **esp_pbuf_get_at**(**const** *esp_pbuf_p pbuf*, size_t *pos*, uint8_t **el*)
Get value from pbuf at specific position.

Return `1` on success, `0` otherwise

Parameters

- `[in] pbuf`: Pbuf used to get data from
- `[in] pos`: Position at which to get element
- `[out] el`: Output variable to save element value at desired position

size_t **esp_pbuf_memcmp**(**const** *esp_pbuf_p pbuf*, **const** void **data*, size_t *len*, size_t *offset*)
Compare pbuf memory with memory from data.

Note Compare is done on entire pbuf chain

Return `0` if equal, `ESP_SIZET_MAX` if memory/offset too big or anything between if not equal

See *esp_pbuf_strcmp*

Parameters

- `[in] pbuf`: Pbuf used to compare with data memory
- `[in] data`: Actual data to compare with
- `[in] len`: Length of input data in units of bytes
- `[in] offset`: Start offset to use when comparing data

size_t **esp_pbuf_strcmp**(**const** *esp_pbuf_p pbuf*, **const** char **str*, size_t *offset*)
Compare pbuf memory with input string.

Note Compare is done on entire pbuf chain

Return `0` if equal, `ESP_SIZET_MAX` if memory/offset too big or anything between if not equal

See *esp_pbuf_memcmp*

Parameters

- `[in] pbuf`: Pbuf used to compare with data memory
- `[in] str`: String to be compared with pbuf
- `[in] offset`: Start memory offset in pbuf

size_t **esp_pbuf_memfind**(**const** *esp_pbuf_p pbuf*, **const** void **data*, size_t *len*, size_t *off*)
Find desired needle in a haystack.

Return `ESP_SIZET_MAX` if no match or position where in pbuf we have a match

See *esp_pbuf_strfind*

Parameters

- `[in] pbuf`: Pbuf used as haystack
- `[in] needle`: Data memory used as needle
- `[in] len`: Length of needle memory
- `[in] off`: Starting offset in pbuf memory

size_t **esp_pbuf_strfind**(**const** *esp_pbuf_p pbuf*, **const** char **str*, size_t *off*)
Find desired needle (str) in a haystack (pbuf)

Return `ESP_SIZET_MAX` if no match or position where in pbuf we have a match

See *esp_pbuf_memfind*

Parameters

- `[in] pbuf`: Pbuf used as haystack
- `[in] str`: String to search for in pbuf
- `[in] off`: Starting offset in pbuf memory

uint8_t **esp_pbuf_advance**(*esp_pbuf_p pbuf*, int *len*)
Advance pbuf payload pointer by number of len bytes. It can only advance single pbuf in a chain.

Note When other pbufs are referencing current one, they are not adjusted in length and total length

Return `1` on success, `0` otherwise

Parameters

- `[in] pbuf`: Pbuf to advance
- `[in] len`: Number of bytes to advance. when negative is used, buffer size is increased only if it was decreased before

esp_pbuf_p **esp_pbuf_skip**(*esp_pbuf_p pbuf*, size_t *offset*, size_t **new_offset*)
Skip a list of pbufs for desired offset.

Note Reference is not changed after return and user must not free the memory of new pbuf directly

Return New pbuf on success, `NULL` otherwise

Parameters

- `[in] pbuf`: Start of pbuf chain
- `[in] offset`: Offset in units of bytes to skip

- `[out] new_offset`: Pointer to output variable to save new offset in returned pbuf

void ***esp_pbuf_get_linear_addr**(**const** *esp_pbuf_p pbuf*, size_t *offset*, size_t **new_len*)
Get linear offset address for pbuf from specific offset.

Note Since pbuf memory can be fragmented in chain, you may need to call function multiple times to get memory for entire pbuf chain

Return Pointer to memory on success, `NULL` otherwise

Parameters

- `[in] pbuf`: Pbuf to get linear address
- `[in] offset`: Start offset from where to start
- `[out] new_len`: Length of memory returned by function

void **esp_pbuf_set_ip**(*esp_pbuf_p pbuf*, **const** *esp_ip_t *ip*, *esp_port_t port*)
Set IP address and port number for received data.

Parameters

- `[in] pbuf`: Packet buffer
- `[in] ip`: IP to assing to packet buffer
- `[in] port`: Port number to assign to packet buffer

void **esp_pbuf_dump**(*esp_pbuf_p p*, uint8_t *seq*)
Dump and debug pbuf chain.

Parameters

- `[in] p`: Head pbuf to dump
- `[in] seq`: Set to `1` to dump all `pbufs` in linked list or `0` to dump first one only

struct esp_pbuf_t
#include <esp_private.h> Packet buffer structure.

Public Members

struct esp_pbuf ***next**
Next pbuf in chain list

size_t **tot_len**
Total length of pbuf chain

size_t **len**
Length of payload

size_t **ref**
Number of references to this structure

uint8_t ***payload**
Pointer to payload memory

esp_ip_t **ip**
Remote address for received IPD data

esp_port_t **port**
Remote port for received IPD data

Ping support

group **ESP_PING**
Ping server and get response time.

Functions

espr_t **esp_ping**(**const** char *host, uint32_t *time, **const** *esp_api_cmd_evt_fn* *evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)
Ping server and get response time from it.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] host`: Host name to ping
- `[out] time`: Pointer to output variable to save ping time in units of milliseconds
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

Smart config

group **ESP_SMART**
SMART function on ESP device.

Functions

espr_t **esp_smart_configure**(uint8_t *en*, **const** *esp_api_cmd_evt_fn* *evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)
Configure SMART function on ESP device.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] en`: Set to `1` to start SMART or `0` to stop SMART
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

Simple Network Time Protocol

ESP has built-in support for *Simple Network Time Protocol (SNTP)*. It is support through middleware API calls for configuring servers and reading actual date and time.

Listing 21: Minimum SNTP example

```
#include "sntp.h"
#include "esp/esp.h"

/**
 * \brief           Run SNTP
 */
void
sntp_gettime(void) {
    esp_datetime_t dt;

    /* Enable SNTP with default configuration for NTP servers */
    if (esp_sntp_configure(1, 1, NULL, NULL, NULL, NULL, NULL, 1) == espOK) {
        esp_delay(5000);

        /* Get actual time and print it */
        if (esp_sntp_gettime(&dt, NULL, NULL, 1) == espOK) {
            printf("Date & time: %d.%d.%d, %d:%d:%d\r\n",
                (int)dt.date, (int)dt.month, (int)dt.year,
                (int)dt.hours, (int)dt.minutes, (int)dt.seconds);
        }
    }
}
```

group **ESP_SNTP**

Simple network time protocol supported by AT commands.

Functions

espr_t **esp_sntp_configure**(uint8_t *en*, int8_t *tz*, **const** char **h1*, **const** char **h2*, **const** char **h3*, **const** *esp_api_cmd_evt_fn* *evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)

Configure SNTP mode parameters.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in]` `en`: Status whether SNTP mode is enabled or disabled on ESP device
- `[in]` `tz`: Timezone to use when SNTP acquires time, between `-11` and `13`
- `[in]` `h1`: Optional first SNTP server for time. Set to `NULL` if not used
- `[in]` `h2`: Optional second SNTP server for time. Set to `NULL` if not used
- `[in]` `h3`: Optional third SNTP server for time. Set to `NULL` if not used
- `[in]` `evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in]` `evt_arg`: Custom argument for event callback function
- `[in]` `blocking`: Status whether command should be blocking or not

espr_t **esp_sntp_gettime**(*esp_datetime_t* *dt, **const** *esp_api_cmd_evt_fn* *evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)

Get time from SNTP servers.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[out] dt`: Pointer to *esp_datetime_t* structure to fill with date and time values
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

Station API

Station API is used to work with *ESP* acting in station mode. It allows to join other access point, scan for available access points or simply disconnect from it.

An example below is showing how all examples (coming with this library) scan for access point and then try to connect to AP from list of preferred one.

Listing 22: Station manager used with all examples

```
#include "station_manager.h"
#include "esp/esp.h"

/*
 * List of preferred access points for ESP device
 * SSID and password
 *
 * ESP will try to scan for access points
 * and then compare them with the one on the list below
 */
ap_entry_t
ap_list[] = {
    //{ "SSID name", "SSID password" },
    { "TilenM_ST", "its private" },
    { "Majerle WIFI", "majerle_internet_private" },
    { "Majerle AMIS", "majerle_internet_private" },
};

/**
 * \brief           List of access points found by ESP device
 */
static
esp_ap_t aps[100];

/**
 * \brief           Number of valid access points in \ref aps array
 */
static
size_t apf;

/**
 * \brief           Connect to preferred access point
```

(continues on next page)

(continued from previous page)

```
 *
 * \note            List of access points should be set by user in \ref ap_list structure
 * \param[in]       unlimited: When set to 1, function will block until SSID is found and connected
 * \return          \ref espOK on success, member of \ref espr_t enumeration otherwise
 */
espr_t
connect_to_preferred_access_point(uint8_t unlimited) {
    espr_t eres;
    uint8_t tried;

    /*
     * Scan for network access points
     * In case we have access point,
     * try to connect to known AP
     */
    do {
        if (esp_sta_has_ip()) {
            return espOK;
        }

        /* Scan for access points visible to ESP device */
        printf("Scanning access points...\r\n");
        if ((eres = esp_sta_list_ap(NULL, aps, ESP_ARRAYSIZE(aps), &apf, NULL, NULL, 1)) == espOK) {
            tried = 0;
            /* Print all access points found by ESP */
            for (size_t i = 0; i < apf; i++) {
                printf("AP found: %s, CH: %d, RSSI: %d\r\n", aps[i].ssid, aps[i].ch, aps[i].rssi);
            }

            /* Process array of preferred access points with array of found points */
            for (size_t j = 0; j < ESP_ARRAYSIZE(ap_list); j++) {
                for (size_t i = 0; i < apf; i++) {
                    if (!strcmp(aps[i].ssid, ap_list[j].ssid)) {
                        tried = 1;
                        printf("Connecting to \"%s\" network...\r\n", ap_list[j].ssid);
                        /* Try to join to access point */
                        if ((eres = esp_sta_join(ap_list[j].ssid, ap_list[j].pass, NULL, NULL, NULL, 1)) == espOK) {
                            esp_ip_t ip;
                            uint8_t is_dhcp;
                            esp_sta_copy_ip(&ip, NULL, NULL, &is_dhcp);

                            printf("Connected to %s network!\r\n", ap_list[j].ssid);
                            printf("Station IP address: %d.%d.%d.%d; Is DHCP: %d\r\n",
                                (int)ip.ip[0], (int)ip.ip[1], (int)ip.ip[2], (int)ip.ip[3], (int)is_dhcp);
                            return espOK;
                        } else {
                            printf("Connection error: %d\r\n", (int)eres);
                        }
                    }
                }
```

(continues on next page)

(continued from previous page)

```
            }
            if (!tried) {
                printf("No access points available with preferred SSID!\r\nPlease check station_manager.c file and edit preferred SSID access points!\r\n");
            }
        } else if (eres == espERRNODEVICE) {
            printf("Device is not present!\r\n");
            break;
        } else {
            printf("Error on WIFI scan procedure!\r\n");
        }
        if (!unlimited) {
            break;
        }
    } while (1);
    return espERR;
}
```

group **ESP_STA**

Station API.

Functions

espr_t **esp_sta_join**(**const** char **name*, **const** char **pass*, **const** *esp_mac_t* **mac*, **const** *esp_api_cmd_evt_fn* *evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)

Join as station to access point.

Configuration changes will be saved in the NVS area of ESP device.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in]` `name`: SSID of access point to connect to
- `[in]` `pass`: Password of access point. Use `NULL` if AP does not have password
- `[in]` `mac`: Pointer to MAC address of AP. If multiple APs with same name exist, MAC may help to select proper one. Set to `NULL` if not needed
- `[in]` `evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in]` `evt_arg`: Custom argument for event callback function
- `[in]` `blocking`: Status whether command should be blocking or not

espr_t **esp_sta_quit**(**const** *esp_api_cmd_evt_fn* *evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)

Quit (disconnect) from access point.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in]` `evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in]` `evt_arg`: Custom argument for event callback function

- `[in] blocking`: Status whether command should be blocking or not

espr_t **esp_sta_autojoin**(uint8_t *en*, **const** *esp_api_cmd_evt_fn evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)

Configure auto join to access point on startup.

Note For auto join feature, you need to do a join to access point with default mode. Check *esp_sta_join* for more information

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] en`: Set to `1` to enable or `0` to disable
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

espr_t **esp_sta_getip**(*esp_ip_t* **ip*, *esp_ip_t* **gw*, *esp_ip_t* **nm*, **const** *esp_api_cmd_evt_fn evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)

Get station IP address.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[out] ip`: Pointer to variable to save IP address
- `[out] gw`: Pointer to output variable to save gateway address
- `[out] nm`: Pointer to output variable to save netmask address
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

espr_t **esp_sta_setip**(**const** *esp_ip_t* **ip*, **const** *esp_ip_t* **gw*, **const** *esp_ip_t* **nm*, **const** *esp_api_cmd_evt_fn evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)

Set station IP address.

Application may manually set IP address. When this happens, stack will check for DHCP settings and will read actual IP address from device. Once procedure is finished, *ESP_EVT_WIFI_IP_ACQUIRED* event will be sent to application where user may read the actual new IP and DHCP settings.

Configuration changes will be saved in the NVS area of ESP device.

Note DHCP is automatically disabled when using static IP address

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] ip`: Pointer to IP address
- `[in] gw`: Pointer to gateway address. Set to `NULL` to use default gateway
- `[in] nm`: Pointer to netmask address. Set to `NULL` to use default netmask

- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

espr_t **esp_sta_getmac**(*esp_mac_t* *mac, **const** *esp_api_cmd_evt_fn* *evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)

Get station MAC address.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[out] mac`: Pointer to output variable to save MAC address
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

espr_t **esp_sta_setmac**(**const** *esp_mac_t* *mac, **const** *esp_api_cmd_evt_fn* *evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)

Set station MAC address.

Configuration changes will be saved in the NVS area of ESP device.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] mac`: Pointer to variable with MAC address
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

uint8_t **esp_sta_has_ip**(void)

Check if ESP got IP from access point.

Return `1` on success, `0` otherwise

uint8_t **esp_sta_is_joined**(void)

Check if station is connected to WiFi network.

Return `1` on success, `0` otherwise

espr_t **esp_sta_copy_ip**(*esp_ip_t* *ip, *esp_ip_t* *gw, *esp_ip_t* *nm, uint8_t *is_dhcp)

Copy IP address from internal value to user variable.

Note Use *esp_sta_getip* to refresh actual IP value from device

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- [out] ip: Pointer to output IP variable. Set to NULL if not interested in IP address
- [out] gw: Pointer to output gateway variable. Set to NULL if not interested in gateway address
- [out] nm: Pointer to output netmask variable. Set to NULL if not interested in netmask address
- [out] is_dhcp: Pointer to output DHCP status variable. Set to NULL if not interested

espr_t **esp_sta_list_ap**(**const** char *ssid, *esp_ap_t* *aps, size_t apsl, size_t *apf, **const** *esp_api_cmd_evt_fn* evt_fn, void ***const** evt_arg, **const** uint32_t *blocking*)

List for available access points ESP can connect to.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- [in] ssid: Optional SSID name to search for. Set to NULL to disable filter
- [in] aps: Pointer to array of available access point parameters
- [in] apsl: Length of aps array
- [out] apf: Pointer to output variable to save number of access points found
- [in] evt_fn: Callback function called when command has finished. Set to NULL when not used
- [in] evt_arg: Custom argument for event callback function
- [in] blocking: Status whether command should be blocking or not

espr_t **esp_sta_get_ap_info**(*esp_sta_info_ap_t* *info, **const** *esp_api_cmd_evt_fn* evt_fn, void ***const** evt_arg, **const** uint32_t *blocking*)

Get current access point information (name, mac, channel, rssi)

Note Access point station is currently connected to

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- [in] info: Pointer to connected access point information
- [in] evt_fn: Callback function called when command has finished. Set to NULL when not used
- [in] evt_arg: Custom argument for event callback function
- [in] blocking: Status whether command should be blocking or not

uint8_t **esp_sta_is_ap_802_11b**(*esp_ap_t* *ap)

Check if access point is 802.11b compatible.

Return 1 on success, 0 otherwise

Parameters

- [in] ap: Access point detailes acquired by *esp_sta_list_ap*

uint8_t **esp_sta_is_ap_802_11g**(*esp_ap_t* *ap)

Check if access point is 802.11g compatible.

Return 1 on success, 0 otherwise

Parameters

- `[in] ap`: Access point detailes acquired by *esp_sta_list_ap*

uint8_t **esp_sta_is_ap_802_11n**(*esp_ap_t* *ap)
Check if access point is `802.11n` compatible.

Return `1` on success, `0` otherwise

Parameters

- `[in] ap`: Access point detailes acquired by *esp_sta_list_ap*

struct esp_sta_t
#include <esp_typedefs.h> Station data structure.

Public Members

esp_ip_t **ip**
IP address of connected station

esp_mac_t **mac**
MAC address of connected station

Timeout manager

Timeout manager allows application to call specific function at desired time. It is used in middleware (and can be used by application too) to poll active connections.

Note: Callback function is called from *processing* thread. It is not allowed to call any blocking API function from it.

When application registers timeout, it needs to set timeout, callback function and optional user argument. When timeout elapses, ESP middleware will call timeout callback.

This feature can be considered as single-shot software timer.

group **ESP_TIMEOUT**
Timeout manager.

Typedefs

typedef void (***esp_timeout_fn**)(void *arg)
Timeout callback function prototype.

Parameters

- `[in] arg`: Custom user argument

Functions

espr_t **esp_timeout_add**(uint32_t *time*, *esp_timeout_fn* *fn*, void **arg*)
Add new timeout to processing list.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] time`: Time in units of milliseconds for timeout execution
- `[in] fn`: Callback function to call when timeout expires
- `[in] arg`: Pointer to user specific argument to call when timeout callback function is executed

espr_t **esp_timeout_remove**(*esp_timeout_fn* *fn*)
Remove callback from timeout list.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] fn`: Callback function to identify timeout to remove

struct esp_timeout_t
#include <*esp_typedefs.h*> Timeout structure.

Public Members

struct esp_timeout ***next**
Pointer to next timeout entry

uint32_t **time**
Time difference from previous entry

void ***arg**
Argument to pass to callback function

esp_timeout_fn **fn**
Callback function for timeout

Structures and enumerations

group **ESP_TYPEDEFS**
List of core structures and enumerations.

Typedefs

typedef uint16_t **esp_port_t**
Port variable.

typedef void (***esp_api_cmd_evt_fn**)(espr_t res, void *arg)
Function declaration for API function command event callback function.

Parameters

- `[in] res`: Operation result, member of *espr_t* enumeration

- `[in] arg`: Custom user argument

Enums

enum esp_cmd_t
List of possible messages.

Values:

ESP_CMD_IDLE = 0
IDLE mode

ESP_CMD_RESET
Reset device

ESP_CMD_ATE0
Disable ECHO mode on AT commands

ESP_CMD_ATE1
Enable ECHO mode on AT commands

ESP_CMD_GMR
Get AT commands version

ESP_CMD_GSLP
Set ESP to sleep mode

ESP_CMD_RESTORE
Restore ESP internal settings to default values

ESP_CMD_UART

ESP_CMD_SLEEP

ESP_CMD_WAKEUPGPIO

ESP_CMD_RFPOWER

ESP_CMD_RFVDD

ESP_CMD_RFAUTOTRACE

ESP_CMD_SYSRAM

ESP_CMD_SYSADC

ESP_CMD_SYSMSG

ESP_CMD_SYSLOG

ESP_CMD_WIFI_CWMODE
Set wifi mode

ESP_CMD_WIFI_CWMODE_GET
Get wifi mode

ESP_CMD_WIFI_CWLAPOPT
Configure what is visible on CWLAP response

ESP_CMD_WIFI_CWJAP
Connect to access point

ESP_CMD_WIFI_CWJAP_GET
Info of the connected access point

ESP_CMD_WIFI_CWQAP
Disconnect from access point

ESP_CMD_WIFI_CWLAP
List available access points

ESP_CMD_WIFI_CIPSTAMAC_GET
Get MAC address of ESP station

ESP_CMD_WIFI_CIPSTAMAC_SET
Set MAC address of ESP station

ESP_CMD_WIFI_CIPSTA_GET
Get IP address of ESP station

ESP_CMD_WIFI_CIPSTA_SET
Set IP address of ESP station

ESP_CMD_WIFI_CWAUTOCONN
Configure auto connection to access point

ESP_CMD_WIFI_CWDHCP_SET
Set DHCP config

ESP_CMD_WIFI_CWDHCP_GET
Get DHCP config

ESP_CMD_WIFI_CWDHCPS_SET
Set DHCP SoftAP IP config

ESP_CMD_WIFI_CWDHCPS_GET
Get DHCP SoftAP IP config

ESP_CMD_WIFI_CWSAP_GET
Get software access point configuration

ESP_CMD_WIFI_CWSAP_SET
Set software access point configuration

ESP_CMD_WIFI_CIPAPMAC_GET
Get MAC address of ESP access point

ESP_CMD_WIFI_CIPAPMAC_SET
Set MAC address of ESP access point

ESP_CMD_WIFI_CIPAP_GET
Get IP address of ESP access point

ESP_CMD_WIFI_CIPAP_SET
Set IP address of ESP access point

ESP_CMD_WIFI_CWLIF
Get connected stations on access point

ESP_CMD_WIFI_CWQIF
Discnnect station from SoftAP

ESP_CMD_WIFI_WPS
Set WPS option

ESP_CMD_WIFI_MDNS
Configure MDNS function

ESP_CMD_WIFI_CWHOSTNAME_SET
Set device hostname

ESP_CMD_WIFI_CWHOSTNAME_GET
Get device hostname

ESP_CMD_TCPIP_CIPDOMAIN
Get IP address from domain name = DNS function

ESP_CMD_TCPIP_CIPDNS_SET
Configure user specific DNS servers

ESP_CMD_TCPIP_CIPDNS_GET
Get DNS configuration

ESP_CMD_TCPIP_CIPSTATUS
Get status of connections

ESP_CMD_TCPIP_CIPSTART
Start client connection

ESP_CMD_TCPIP_CIPSEND
Send network data

ESP_CMD_TCPIP_CIPCLOSE
Close active connection

ESP_CMD_TCPIP_CIPSSLSIZE
Set SSL buffer size for SSL connection

ESP_CMD_TCPIP_CIPSSLCCONF
Set the SSL configuration

ESP_CMD_TCPIP_CIFSR
Get local IP

ESP_CMD_TCPIP_CIPMUX
Set single or multiple connections

ESP_CMD_TCPIP_CIPSERVER
Enables/Disables server mode

ESP_CMD_TCPIP_CIPSERVERMAXCONN
Sets maximal number of connections allowed for server population

ESP_CMD_TCPIP_CIPMODE
Transmission mode, either transparent or normal one

ESP_CMD_TCPIP_CIPSTO
Sets connection timeout

ESP_CMD_TCPIP_CIPRECVMODE
Sets mode for TCP data receive (manual or automatic)

ESP_CMD_TCPIP_CIPRECVDATA
Manually reads TCP data from device

ESP_CMD_TCPIP_CIPRECVLEN
Gets number of available bytes in connection to be read

ESP_CMD_TCPIP_CIUPDATE
Perform self-update

ESP_CMD_TCPIP_CIPSNTPCFG
Configure SNTP servers

ESP_CMD_TCPIP_CIPSNTPTIME
Get current time using SNTP

ESP_CMD_TCPIP_CIPDINFO
Configure what data are received on +IPD statement

ESP_CMD_TCPIP_PING
Ping domain

ESP_CMD_WIFI_SMART_START
Start smart config

ESP_CMD_WIFI_SMART_STOP
Stop smart config

ESP_CMD_BLEINIT_GET
Get BLE status

enum espr_t
Result enumeration used across application functions.

Values:

espOK = 0
Function succeeded

espOKIGNOREMORE
Function succedded, should continue as espOK but ignore sending more data. This result is possible on connection data receive callback

espERR

espPARERR
Wrong parameters on function call

espERRMEM
Memory error occurred

espTIMEOUT
Timeout occurred on command

espCONT
There is still some command to be processed in current command

espCLOSED
Connection just closed

espINPROG
Operation is in progress

espERRNOIP
Station does not have IP address

espERRNOFREECONN
There is no free connection available to start

espERRCONNTIMEOUT
Timeout received when connection to access point

espERRPASS
Invalid password for access point

espERRNOAP
No access point found with specific SSID and MAC address

espERRCONNFAIL
Connection failed to access point

espERRWIFINOTCONNECTED
Wifi not connected to access point

espERRNODEVICE
Device is not present

espERRBLOCKING
Blocking mode command is not allowed

enum esp_device_t
List of support ESP devices by firmware.

Values:

ESP_DEVICE_ESP8266
Device is ESP8266

ESP_DEVICE_ESP32
Device is ESP32

ESP_DEVICE_UNKNOWN
Unknown device

enum esp_ecn_t
List of encryptions of access point.

Values:

ESP_ECN_OPEN = 0x00
No encryption on access point

ESP_ECN_WEP
WEP (Wired Equivalent Privacy) encryption

ESP_ECN_WPA_PSK
WPA (Wifi Protected Access) encryption

ESP_ECN_WPA2_PSK
WPA2 (Wifi Protected Access 2) encryption

ESP_ECN_WPA_WPA2_PSK
WPA/2 (Wifi Protected Access 1/2) encryption

ESP_ECN_WPA2_Enterprise
Enterprise encryption.
Note ESP is currently not able to connect to access point of this encryption type

enum esp_mode_t
List of possible WiFi modes.

Values:

ESP_MODE_STA = 1
Set WiFi mode to station only

ESP_MODE_AP = 2
Set WiFi mode to access point only

ESP_MODE_STA_AP = 3
Set WiFi mode to station and access point

enum esp_http_method_t
List of possible HTTP methods.

Values:

ESP_HTTP_METHOD_GET
HTTP method GET

ESP_HTTP_METHOD_HEAD
HTTP method HEAD

ESP_HTTP_METHOD_POST
HTTP method POST

ESP_HTTP_METHOD_PUT
HTTP method PUT

ESP_HTTP_METHOD_DELETE
HTTP method DELETE

ESP_HTTP_METHOD_CONNECT
HTTP method CONNECT

ESP_HTTP_METHOD_OPTIONS
HTTP method OPTIONS

ESP_HTTP_METHOD_TRACE
HTTP method TRACE

ESP_HTTP_METHOD_PATCH
HTTP method PATCH

struct esp_conn_t
#include <esp_private.h> Connection structure.

Public Members

esp_conn_type_t **type**
Connection type

uint8_t **num**
Connection number

esp_ip_t **remote_ip**
Remote IP address

esp_port_t **remote_port**
Remote port number

esp_port_t **local_port**
Local IP address

esp_evt_fn **evt_func**
Callback function for connection

void ***arg**
User custom argument

uint8_t **val_id**
Validation ID number. It is increased each time a new connection is established. It protects sending data to wrong connection in case we have data in send queue, and connection was closed and active again in between.

esp_linbuff_t **buff**
Linear buffer structure

size_t **total_recved**
Total number of bytes received

size_t **tcp_available_bytes**
Number of bytes in ESP ready to be read on connection. This variable always holds last known info from ESP device and is not decremented (or incremented) by application

size_t **tcp_not_ack_bytes**
Number of bytes not acknowledge by application done with processing This variable is increased everytime new packet is read to be sent to application and decreased when application acknowledges it

uint8_t **active** : 1
Status whether connection is active

uint8_t **client** : 1
Status whether connection is in client mode

uint8_t **data_received** : 1
Status whether first data were received on connection

uint8_t **in_closing** : 1
Status if connection is in closing mode. When in closing mode, ignore any possible received data from function

uint8_t **receive_blocked** : 1
Status whether we should block manual receive for some time

uint8_t **receive_is_command_queued** : 1
Status whether manual read command is in the queue already

struct *esp_conn_t*::**[anonymous]**::**[anonymous]** **f**
Connection flags

union *esp_conn_t*::**[anonymous]** **status**
Connection status union with flag bits

struct esp_pbuf_t
#include <*esp_private.h*> Packet buffer structure.

Public Members

struct esp_pbuf ***next**
Next pbuf in chain list

size_t **tot_len**
Total length of pbuf chain

size_t **len**
Length of payload

size_t **ref**
Number of references to this structure

uint8_t *payload
Pointer to payload memory

esp_ip_t ip
Remote address for received IPD data

esp_port_t port
Remote port for received IPD data

struct esp_ipd_t
#include <esp_private.h> Incoming network data read structure.

Public Members

uint8_t read
Set to 1 when we should process input data as connection data

size_t tot_len
Total length of packet

size_t rem_len
Remaining bytes to read in current +IPD statement

esp_conn_p conn
Pointer to connection for network data

esp_ip_t ip
Remote IP address on from IPD data

esp_port_t port
Remote port on IPD data

size_t buff_ptr
Buffer pointer to save data to. When set to `NULL` while `read = 1`, reading should ignore incoming data

esp_pbuf_p buff
Pointer to data buffer used for receiving data

struct esp_msg_t
#include <esp_private.h> Message queue structure to share between threads.

Public Members

esp_cmd_t cmd_def
Default message type received from queue

esp_cmd_t cmd
Since some commands can have different subcommands, sub command is used here

uint8_t i
Variable to indicate order number of subcommands

esp_sys_sem_t sem
Semaphore for the message

uint8_t is_blocking
Status if command is blocking

uint32_t block_time
Maximal blocking time in units of milliseconds. Use 0 to for non-blocking call

espr_t **res**
Result of message operation

espr_t (***fn**) (**struct** esp_msg *)
Processing callback function to process packet

uint32_t **delay**
Delay in units of milliseconds before executing first RESET command

struct *esp_msg_t*::**[anonymous]**::**[anonymous]** **reset**
Reset device

uint32_t **baudrate**
Baudrate for AT port

struct *esp_msg_t*::**[anonymous]**::**[anonymous]** **uart**
UART configuration

esp_mode_t **mode**
Mode of operation

esp_mode_t ***mode_get**
Get mode

struct *esp_msg_t*::**[anonymous]**::**[anonymous]** **wifi_mode**
When message type *ESP_CMD_WIFI_CWMODE* is used

const char ***name**
AP name

const char ***pass**
AP password

const *esp_mac_t* ***mac**
Specific MAC address to use when connecting to AP

uint8_t **error_num**
Error number on connecting

struct *esp_msg_t*::**[anonymous]**::**[anonymous]** **sta_join**
Message for joining to access point

uint8_t **en**
Status to enable/disable auto join feature

Enable/disable DHCP settings

Enable/Disable server status

Status if SNTP is enabled or not

Status if WPS is enabled or not

Set to 1 to enable or 0 to disable

struct *esp_msg_t*::**[anonymous]**::**[anonymous]** **sta_autojoin**
Message for auto join procedure

esp_sta_info_ap_t ***info**
Information structure

struct *esp_msg_t*::**[anonymous]**::**[anonymous]** **sta_info_ap**
Message for reading the AP information

const char ***ssid**
Pointer to optional filter SSID name to search

Name of access point

esp_ap_t ***aps**
Pointer to array to save access points

size_t **apsl**
Length of input array of access points

size_t **apsi**
Current access point array

size_t ***apf**
Pointer to output variable holding number of access points found

struct *esp_msg_t*::**[anonymous]**::**[anonymous]** **ap_list**
List for available access points to connect to

const char ***pwd**
Password of access point

esp_ecn_t **ecn**
Ecryption used

uint8_t **ch**
RF Channel used

uint8_t **max_sta**
Max allowed connected stations

uint8_t **hid**
Configuration if network is hidden or visible

struct *esp_msg_t*::**[anonymous]**::**[anonymous]** **ap_conf**
Parameters to configure access point

esp_ap_conf_t ***ap_conf**
AP configuration

struct *esp_msg_t*::**[anonymous]**::**[anonymous]** **ap_conf_get**
Get the soft AP configuration

esp_sta_t ***stas**
Pointer to array to save access points

size_t **stal**
Length of input array of access points

size_t **stai**
Current access point array

size_t ***staf**
Pointer to output variable holding number of access points found

struct *esp_msg_t*::**[anonymous]**::**[anonymous]** **sta_list**
List for stations connected to SoftAP

esp_mac_t **mac**
MAC address to disconnect from access point

Pointer to MAC variable

struct *esp_msg_t*::**[anonymous]**::**[anonymous]** **ap_disconn_sta**
Disconnect station from access point

esp_ip_t ***ip**
Pointer to IP variable

esp_ip_t ***gw**
Pointer to gateway variable

esp_ip_t ***nm**
Pointer to netmask variable

struct *esp_msg_t*::**[anonymous]**::**[anonymous]** **sta_ap_getip**
Message for reading station or access point IP

esp_mac_t ***mac**
Pointer to MAC variable

struct *esp_msg_t*::**[anonymous]**::**[anonymous]** **sta_ap_getmac**
Message for reading station or access point MAC address

esp_ip_t **ip**
Pointer to IP variable

esp_ip_t **gw**
Pointer to gateway variable

esp_ip_t **nm**
Pointer to netmask variable

struct *esp_msg_t*::**[anonymous]**::**[anonymous]** **sta_ap_setip**
Message for setting station or access point IP

struct *esp_msg_t*::**[anonymous]**::**[anonymous]** **sta_ap_setmac**
Message for setting station or access point MAC address

uint8_t **sta**
Set station DHCP settings

uint8_t **ap**
Set access point DHCP settings

struct *esp_msg_t*::**[anonymous]**::**[anonymous]** **wifi_cwdhcp**
Set DHCP settings

const char ***hostname_set**
Hostname set value

char ***hostname_get**
Hostname get value

size_t **length**
Length of buffer when reading hostname

struct *esp_msg_t*::**[anonymous]**::**[anonymous]** **wifi_hostname**
Set or get hostname structure

esp_conn_t ****conn**
Pointer to pointer to save connection used

const char ***remote_host**
Host to use for connection

esp_port_t **remote_port**
Remote port used for connection

Remote port address for UDP connection

esp_conn_type_t **type**
Connection type

const char ***local_ip**
Local IP address. Normally set to NULL

uint16_t **tcp_ssl_keep_alive**
Keep alive parameter for TCP

uint8_t **udp_mode**
UDP mode

esp_port_t **udp_local_port**
UDP local port

void ***arg**
Connection custom argument

esp_evt_fn **evt_func**
Callback function to use on connection

uint8_t **num**
Connection number used for start

uint8_t **success**
Status if connection AT+CIPSTART succedded

struct *esp_msg_t*::**[anonymous]**::**[anonymous]** **conn_start**
Structure for starting new connection

esp_conn_t ***conn**
Pointer to connection to close

Pointer to connection to send data

uint8_t **val_id**
Connection current validation ID when command was sent to queue

struct *esp_msg_t*::**[anonymous]**::**[anonymous]** **conn_close**
Close connection

size_t **btw**
Number of remaining bytes to write

size_t **ptr**
Current write pointer for data

const uint8_t ***data**
Data to send

size_t **sent**
Number of bytes sent in last packet

size_t **sent_all**
Number of bytes sent all together

uint8_t **tries**
Number of tries used for last packet

uint8_t **wait_send_ok_err**
Set to 1 when we wait for SEND OK or SEND ERROR

const *esp_ip_t* ***remote_ip**
Remote IP address for UDP connection

uint8_t **fau**
Free after use flag to free memory after data are sent (or not)

size_t ***bw**
Number of bytes written so far

struct *esp_msg_t*::**[anonymous]**::**[anonymous]** **conn_send**
Structure to send data on connection

esp_port_t **port**
Server port number

mDNS server port

uint16_t **max_conn**
Maximal number of connections available for server

uint16_t **timeout**
Connection timeout

esp_evt_fn **cb**
Server default callback function

struct *esp_msg_t*::**[anonymous]**::**[anonymous]** **tcpip_server**
Server configuration

size_t **size**
Size for SSL in uints of bytes

struct *esp_msg_t*::**[anonymous]**::**[anonymous]** **tcpip_sslsize**
TCP SSL size for SSL connections

const char ***host**
Hostname to ping

mDNS host name

uint32_t **time**
Time used for ping

uint32_t ***time_out**
Pointer to time output variable

struct *esp_msg_t*::**[anonymous]**::**[anonymous]** **tcpip_ping**
Pinging structure

int8_t **tz**
Timezone setup

const char ***h1**
Optional server 1

const char ***h2**
Optional server 2

const char ***h3**
Optional server 3

struct *esp_msg_t*::**[anonymous]**::**[anonymous]** **tcpip_sntp_cfg**
SNTP configuration

esp_datetime_t ***dt**
Pointer to datetime structure

struct *esp_msg_t*::**[anonymous]**::**[anonymous]** **tcpip_sntp_time**
SNTP get time

struct *esp_msg_t*::**[anonymous]**::**[anonymous]** **wps_cfg**
WPS configuration

const char ***server**
mDNS server

struct *esp_msg_t*::**[anonymous]**::**[anonymous]** **mdns**
mDNS configuration

uint8_t **link_id**
Link ID of connection to set SSL configuration for

uint8_t **auth_mode**
Timezone setup

uint8_t **pki_number**
The index of cert and private key, if only one cert and private key, the value should be 0.

uint8_t **ca_number**
The index of CA, if only one CA, the value should be 0.

struct *esp_msg_t*::**[anonymous]**::**[anonymous]** **tcpip_ssl_cfg**
SSl configuration for connection

union *esp_msg_t*::**[anonymous]** **msg**
Group of different message contents

struct esp_ip_mac_t
#include <esp_private.h> IP and MAC structure with netmask and gateway addresses.

Public Members

esp_ip_t **ip**
IP address

esp_ip_t **gw**
Gateway address

esp_ip_t **nm**
Netmask address

esp_mac_t **mac**
MAC address

uint8_t **dhcp**
Flag indicating DHCP is enabled

uint8_t **has_ip**
Flag indicating ESP has IP

uint8_t **is_connected**
Flag indicating ESP is connected to wifi

struct esp_link_conn_t
#include <esp_private.h> Link connection active info.

Public Members

uint8_t **failed**
Status if connection successful

uint8_t **num**
Connection number

uint8_t **is_server**
Status if connection is client or server

esp_conn_type_t **type**
Connection type

esp_ip_t **remote_ip**
Remote IP address

esp_port_t **remote_port**
Remote port

esp_port_t **local_port**
Local port number

struct esp_evt_func_t
#include <esp_private.h> Callback function linked list prototype.

Public Members

struct esp_evt_func ***next**
Next function in the list

esp_evt_fn **fn**
Function pointer itself

struct esp_modules_t
#include <esp_private.h> ESP modules structure.

Public Members

esp_device_t **device**
ESP device type

esp_sw_version_t **version_at**
Version of AT command software on ESP device

esp_sw_version_t **version_sdk**
Version of SDK used to build AT software

uint32_t **active_conns**
Bit field of currently active connections,

uint32_t **active_conns_last**
The same as previous but status before last check

esp_link_conn_t **link_conn**
Link connection handle

esp_ipd_t **ipd**
Connection incoming data structure

esp_conn_t **conns**[**ESP_CFG_MAX_CONNS**]
Array of all connection structures

esp_ip_mac_t **sta**
Station IP and MAC addressed

esp_ip_mac_t **ap**
Access point IP and MAC addressed

struct esp_t
#include <*esp_private.h*> ESP global structure.

Public Members

size_t **locked_cnt**
Counter how many times (recursive) stack is currently locked

esp_sys_sem_t **sem_sync**
Synchronization semaphore between threads

esp_sys_mbox_t **mbox_producer**
Producer message queue handle

esp_sys_mbox_t **mbox_process**
Consumer message queue handle

esp_sys_thread_t **thread_produce**
Producer thread handle

esp_sys_thread_t **thread_process**
Processing thread handle

esp_buff_t **buff**
Input processing buffer

esp_ll_t **ll**
Low level functions

esp_msg_t ***msg**
Pointer to current user message being executed

esp_evt_t **evt**
Callback processing structure

esp_evt_func_t ***evt_func**
Callback function linked list

esp_evt_fn **evt_server**
Default callback function for server connections

esp_modules_t **m**
All modules. When resetting, reset structure

uint8_t **initialized** : 1
Flag indicating ESP library is initialized

uint8_t **dev_present** : 1
Flag indicating if physical device is connected to host device

struct *esp_t*::**[anonymous]**::**[anonymous] f**
Flags structure

union *esp_t*::**[anonymous] status**
Status structure

uint8_t **conn_val_id**
Validation ID increased each time device connects to wifi network or on reset. It is used for connections

struct esp_unicode_t
#include <*esp_private.h*> Unicode support structure.

Public Members

uint8_t **ch**[4]
UTF-8 max characters

uint8_t **t**
Total expected length in UTF-8 sequence

uint8_t **r**
Remaining bytes in UTF-8 sequence

espr_t **res**
Current result of processing

struct esp_ip_t
#include <*esp_typedefs.h*> IP structure.

Public Members

uint8_t **ip**[4]
IPv4 address

struct esp_mac_t
#include <*esp_typedefs.h*> MAC address.

Public Members

uint8_t **mac**[6]
MAC address

struct esp_sw_version_t
#include <*esp_typedefs.h*> SW version in semantic versioning format.

Public Members

uint8_t **major**
Major version

uint8_t **minor**
Minor version

uint8_t **patch**
Patch version

struct esp_datetime_t
#include <*esp_typedefs.h*> Date and time structure.

Public Members

uint8_t **date**
Day in a month, from 1 to up to 31

uint8_t **month**
Month in a year, from 1 to 12

uint16_t **year**
Year

uint8_t **day**
Day in a week, from 1 to 7

uint8_t **hours**
Hours in a day, from 0 to 23

uint8_t **minutes**
Minutes in a hour, from 0 to 59

uint8_t **seconds**
Seconds in a minute, from 0 to 59

struct esp_linbuff_t
#include <*esp_typedefs.h*> Linear buffer structure.

Public Members

uint8_t ***buff**
Pointer to buffer data array

size_t **len**
Length of buffer array

size_t **ptr**
Current buffer pointer

Unicode

Unicode decoder block. It can decode sequence of *UTF-8* characters, between `1` and `4` bytes long.

Note: This is simple implementation and does not support string encoding.

group **ESP_UNICODE**
Unicode support manager.

Functions

espr_t **espi_unicode_decode**(*esp_unicode_t* **uni*, uint8_t *ch*)
Decode single character for unicode (UTF-8 only) format.

Parameters

- `[inout] s`: Pointer to unicode decode control structure
- `[in] c`: UTF-8 character sequence to test for device

Return Value

- `espOK`: Function succedded, there is a valid UTF-8 sequence
- `espINPROG`: Function continues well but expects some more data to finish sequence
- `espERR`: Error in UTF-8 sequence

struct esp_unicode_t
#include <esp_private.h> Unicode support structure.

Public Members

uint8_t **ch**[4]
UTF-8 max characters

uint8_t **t**
Total expected length in UTF-8 sequence

uint8_t **r**
Remaining bytes in UTF-8 sequence

espr_t **res**
Current result of processing

Utilities

Utility functions for various cases. These function are used across entire middleware and can also be used by application.

group **ESP_UTILS**
Utilities.

Defines

ESP_ASSERT(msg, c)
Assert an input parameter if in valid range.

Note Since this is a macro, it may only be used on a functions where return status is of type *espr_t* enumeration

Parameters

- [in] msg: message to print to debug if test fails
- [in] c: Condition to test

ESP_MEM_ALIGN(x)
Align x value to specific number of bytes, provided by *ESP_CFG_MEM_ALIGNMENT* configuration.

Return Input value aligned to specific number of bytes

Parameters

- [in] x: Input value to align

ESP_MIN(x, y)
Get minimal value between x and y inputs.

Return Minimal value between x and y parameters

Parameters

- [in] x: First input to test
- [in] y: Second input to test

ESP_MAX(x, y)
Get maximal value between x and y inputs.

Return Maximal value between x and y parameters

Parameters

- [in] x: First input to test
- [in] y: Second input to test

ESP_ARRAYSIZE(x)
Get size of statically declared array.

Return Number of array elements

Parameters

- [in] x: Input array

ESP_UNUSED(x)
Unused argument in a function call.

Note Use this on all parameters in a function which are not used to prevent compiler warnings complaining about "unused variables"

Parameters

- `[in] x`: Variable which is not used

ESP_U32(x)
Get input value casted to `unsigned 32-bit` value.

Parameters

- `[in] x`: Input value

ESP_U16(x)
Get input value casted to `unsigned 16-bit` value.

Parameters

- `[in] x`: Input value

ESP_U8(x)
Get input value casted to `unsigned 8-bit` value.

Parameters

- `[in] x`: Input value

ESP_I32(x)
Get input value casted to `signed 32-bit` value.

Parameters

- `[in] x`: Input value

ESP_I16(x)
Get input value casted to `signed 16-bit` value.

Parameters

- `[in] x`: Input value

ESP_I8(x)
Get input value casted to `signed 8-bit` value.

Parameters

- `[in] x`: Input value

ESP_SZ(x)
Get input value casted to `size_t` value.

Parameters

- `[in] x`: Input value

esp_u32_to_str(num, out)
Convert `unsigned 32-bit` number to string.

Return Pointer to output variable

Parameters

- `[in] num`: Number to convert
- `[out] out`: Output variable to save string

esp_u32_to_hex_str(num, out, w)
Convert `unsigned 32-bit` number to HEX string.

Return Pointer to output variable

Parameters

- `[in] num`: Number to convert
- `[out] out`: Output variable to save string
- `[in] w`: Width of output string. When number is shorter than width, leading `0` characters will apply

esp_i32_to_str(num, out)
Convert `signed 32-bit` number to string.

Return Pointer to output variable

Parameters

- `[in] num`: Number to convert
- `[out] out`: Output variable to save string

esp_u16_to_str(num, out)
Convert `unsigned 16-bit` number to string.

Return Pointer to output variable

Parameters

- `[in] num`: Number to convert
- `[out] out`: Output variable to save string

esp_u16_to_hex_str(num, out, w)
Convert `unsigned 16-bit` number to HEX string.

Return Pointer to output variable

Parameters

- `[in] num`: Number to convert
- `[out] out`: Output variable to save string
- `[in] w`: Width of output string. When number is shorter than width, leading `0` characters will apply.

esp_i16_to_str(num, out)
Convert `signed 16-bit` number to string.

Return Pointer to output variable

Parameters

- `[in] num`: Number to convert

- `[out] out`: Output variable to save string

esp_u8_to_str(num, out)
Convert `unsigned 8-bit` number to string.

Return Pointer to output variable

Parameters

- `[in] num`: Number to convert
- `[out] out`: Output variable to save string

esp_u8_to_hex_str(num, out, w)
Convert `unsigned 16-bit` number to HEX string.

Return Pointer to output variable

Parameters

- `[in] num`: Number to convert
- `[out] out`: Output variable to save string
- `[in] w`: Width of output string. When number is shorter than width, leading `0` characters will apply.

esp_i8_to_str(num, out)
Convert `signed 8-bit` number to string.

Return Pointer to output variable

Parameters

- `[in] num`: Number to convert
- `[out] out`: Output variable to save string

Functions

char ***esp_u32_to_gen_str**(uint32_t *num*, char **out*, uint8_t *is_hex*, uint8_t *padding*)
Convert `unsigned 32-bit` number to string.

Return Pointer to output variable

Parameters

- `[in] num`: Number to convert
- `[out] out`: Output variable to save string
- `[in] is_hex`: Set to `1` to output hex, 0 otherwise
- `[in] width`: Width of output string. When number is shorter than width, leading `0` characters will apply. This parameter is valid only when formatting hex numbers

char ***esp_i32_to_gen_str**(int32_t *num*, char **out*)
Convert `signed 32-bit` number to string.

Return Pointer to output variable

Parameters

- `[in] num`: Number to convert
- `[out] out`: Output variable to save string

Wi-Fi Protected Setup

group **ESP_WPS**
WPS function on ESP device.

Functions

espr_t **esp_wps_configure**(uint8_t *en*, **const** *esp_api_cmd_evt_fn evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)
Configure WPS function on ESP device.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] en`: Set to `1` to enable WPS or `0` to disable WPS
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

group **ESP**
ESP stack.

Defines

esp_set_fw_version(v, major_, minor_, patch_)
Set and format major, minor and patch values to firmware version.

Parameters

- `[in] v`: Version output, pointer to *esp_sw_version_t* structure
- `[in] major_`: Major version
- `[in] minor_`: Minor version
- `[in] patch_`: Patch version

esp_get_min_at_fw_version(v)
Get minimal AT version supported by library.

Parameters

- `[out] v`: Version output, pointer to *esp_sw_version_t* structure

Functions

espr_t **esp_init**(*esp_evt_fn cb_func*, **const** uint32_t *blocking*)
Init and prepare ESP stack for device operation.

Note Function must be called from operating system thread context. It creates necessary threads and waits them to start, thus running operating system is important.

- When *ESP_CFG_RESET_ON_INIT* is enabled, reset sequence will be sent to device otherwise manual call to *esp_reset* is required to setup device
- When *ESP_CFG_RESTORE_ON_INIT* is enabled, restore sequence will be sent to device.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] evt_func`: Global event callback function for all major events
- `[in] blocking`: Status whether command should be blocking or not. Used when *ESP_CFG_RESET_ON_INIT* or *ESP_CFG_RESTORE_ON_INIT* are enabled.

espr_t **esp_reset**(**const** *esp_api_cmd_evt_fn evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)
Execute reset and send default commands.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

espr_t **esp_reset_with_delay**(uint32_t *delay*, **const** *esp_api_cmd_evt_fn evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)
Execute reset and send default commands with delay before first command.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] delay`: Number of milliseconds to wait before initiating first command to device
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

espr_t **esp_restore**(**const** *esp_api_cmd_evt_fn evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)
Execute restore command and set module to default values.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

espr_t **esp_set_at_baudrate**(uint32_t *baud*, **const** *esp_api_cmd_evt_fn* *evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)

Sets baudrate of AT port (usually UART)

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] baud`: Baudrate in units of bits per second
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

espr_t **esp_set_wifi_mode**(esp_mode_t *mode*, **const** *esp_api_cmd_evt_fn* *evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)

Sets WiFi mode to either station only, access point only or both.

Configuration changes will be saved in the NVS area of ESP device.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] mode`: Mode of operation. This parameter can be a value of *esp_mode_t* enumeration
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

espr_t **esp_get_wifi_mode**(esp_mode_t **mode*, **const** *esp_api_cmd_evt_fn* *evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)

Gets WiFi mode of either station only, access point only or both.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] mode`: point to space of Mode to get. This parameter can be a pointer of *esp_mode_t* enumeration
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

espr_t **esp_set_server**(uint8_t *en*, *esp_port_t port*, uint16_t *max_conn*, uint16_t *timeout*, *esp_evt_fn cb*, **const** *esp_api_cmd_evt_fn evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)

Enables or disables server mode.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] en`: Set to `1` to enable server, `0` otherwise
- `[in] port`: Port number used to listen on. Must also be used when disabling server mode
- `[in] max_conn`: Number of maximal connections populated by server
- `[in] timeout`: Time used to automatically close the connection in units of seconds. Set to `0` to disable timeout feature (not recommended)
- `[in] server_evt_fn`: Connection callback function for new connections started as server
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

espr_t **esp_update_sw**(**const** *esp_api_cmd_evt_fn evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)

Update ESP software remotely.

Note ESP must be connected to access point to use this feature

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

espr_t **esp_core_lock**(void)

Lock stack from multi-thread access, enable atomic access to core.

If lock was `0` prior funcion call, lock is enabled and increased

Note Function may be called multiple times to increase locks. Application must take care to call *esp_core_unlock* the same amount of time to make sure lock gets back to `0`

Return *espOK* on success, member of *espr_t* enumeration otherwise

espr_t **esp_core_unlock**(void)

Unlock stack for multi-thread access.

Used in conjunction with *esp_core_lock* function

If lock was non-zero before function call, lock is decreased. When `lock == 0`, protection is disabled and other threads may access to core

Return *espOK* on success, member of *espr_t* enumeration otherwise

espr_t **esp_device_set_present**(uint8_t *present*, **const** *esp_api_cmd_evt_fn* *evt_fn*, void ***const** *evt_arg*, **const** uint32_t *blocking*)

Notify stack if device is present or not.

Use this function to notify stack that device is not physically connected and not ready to communicate with host device

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] present`: Flag indicating device is present
- `[in] evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- `[in] evt_arg`: Custom argument for event callback function
- `[in] blocking`: Status whether command should be blocking or not

uint8_t **esp_device_is_present**(void)

Check if device is present.

Return `1` on success, `0` otherwise

uint8_t **esp_device_is_esp8266**(void)

Check if modem device is ESP8266.

Return `1` on success, `0` otherwise

uint8_t **esp_device_is_esp32**(void)

Check if modem device is ESP32.

Return `1` on success, `0` otherwise

uint8_t **esp_delay**(**const** uint32_t *ms*)

Delay for amount of milliseconds.

Delay is based on operating system semaphores. It locks semaphore and waits for timeout in `ms` time. Based on operating system, thread may be put to *blocked* list during delay and may improve execution speed

Return `1` on success, `0` otherwise

Parameters

- `[in] ms`: Milliseconds to delay

uint8_t **esp_get_current_at_fw_version**(*esp_sw_version_t* ***const** *version*)

Get current AT firmware version of connected device.

Return `1` on success, `0` otherwise

Parameters

- `[out] version`: Output version variable

5.3.2 ESP Configuration

This is the default configuration of the middleware. When any of the settings shall be modified, it shall be done in dedicated application config `esp_config.h` file.

Note: Check *Getting started* to create configuration file.

group **ESP_CONFIG**

Configuration parameters.

Defines

ESP_CFG_ESP8266

Enables `1` or disables `0` support for ESP8266 AT commands.

ESP_CFG_ESP32

Enables `1` or disables `0` support for ESP32 AT commands.

ESP_CFG_OS

Enables `1` or disables `0` operating system support for ESP library.

Note Value must be set to `1` in the current revision

Note Check *OS configuration* group for more configuration related to operating system

ESP_CFG_MEM_CUSTOM

Enables `1` or disables `0` custom memory management functions.

When set to `1`, *Memory manager* block must be provided manually. This includes implementation of functions *esp_mem_malloc*, *esp_mem_calloc*, *esp_mem_realloc* and *esp_mem_free*

Note Function declaration follows standard C functions `malloc, calloc, realloc, free`. Declaration is available in `esp/esp_mem.h` file. Include this file to final implementation file

Note When implementing custom memory allocation, it is necessary to take care of multiple threads accessing same resource for custom allocator

ESP_CFG_MEM_ALIGNMENT

Memory alignment for dynamic memory allocations.

Note Some CPUs can work faster if memory is aligned, usually to `4` or `8` bytes. To speed up this possibilities, you can set memory alignment and library will try to allocate memory on aligned boundaries.

Note Some CPUs such ARM Cortex-M0 dont't support unaligned memory access.

Note This value must be power of `2`

ESP_CFG_USE_API_FUNC_EVT

Enables `1` or disables `0` callback function and custom parameter for API functions.

When enabled, `2` additional parameters are available in API functions. When command is executed, callback function with its parameter could be called when not set to `NULL`.

ESP_CFG_MAX_CONNS

Maximal number of connections AT software can support on ESP device.

Note In case of official AT software, leave this on default value (`5`)

ESP_CFG_CONN_MAX_DATA_LEN
Maximal number of bytes we can send at single command to ESP.

When manual TCP read mode is enabled, this parameter defines number of bytes to be read at a time

Note Value can not exceed `2048` bytes or no data will be send at all (ESP8266 AT SW limitation)

Note This is limitation of ESP AT commands and on systems where RAM is not an issue, it should be set to maximal value (`2048`) to optimize data transfer speed performance

ESP_CFG_MAX_SEND_RETRIES
Set number of retries for send data command.

Sometimes it may happen that `AT+SEND` command fails due to different problems. Trying to send the same data multiple times can raise chances for success.

ESP_CFG_CONN_MAX_RECV_BUFF_SIZE
Maximum single buffer size for network receive data on active connection.

Note When ESP sends buffer bigger than maximal, multiple buffers are created

ESP_CFG_AT_PORT_BAUDRATE
Default baudrate used for AT port.

Note User may call API function to change to desired baudrate if necessary

ESP_CFG_MODE_STATION
Enables `1` or disables `0` ESP acting as station.

Note When device is in station mode, it can connect to other access points

ESP_CFG_MODE_ACCESS_POINT
Enables `1` or disables `0` ESP acting as access point.

Note When device is in access point mode, it can accept connections from other stations

ESP_CFG_RCV_BUFF_SIZE
Buffer size for received data waiting to be processed.

Note When server mode is active and a lot of connections are in queue this should be set high otherwise your buffer may overflow

Note Buffer size also depends on TX user driver if it uses DMA or blocking mode. In case of DMA (CPU can work other tasks), buffer may be smaller as CPU will have more time to process all the incoming bytes

Note This parameter has no meaning when *ESP_CFG_INPUT_USE_PROCESS* is enabled

ESP_CFG_RESET_ON_INIT
Enables `1` or disables `0` reset sequence after *esp_init* call.

Note When this functionality is disabled, user must manually call *esp_reset* to send reset sequence to ESP device.

ESP_CFG_RESTORE_ON_INIT
Enables `1` or disables `0` device restore after *esp_init* call.

Note When this feature is enabled, it will automatically restore and clear any settings stored as *default* in ESP device

ESP_CFG_RESET_ON_DEVICE_PRESENT
Enables `1` or disables `0` reset sequence after *esp_device_set_present* call.

Note When this functionality is disabled, user must manually call *esp_reset* to send reset sequence to ESP device.

ESP_CFG_RESET_DELAY_DEFAULT
Default delay (milliseconds unit) before sending first AT command on reset sequence.

ESP_CFG_MAX_SSID_LENGTH
Maximum length of SSID for access point scan.

Note This parameter must include trailling zero

ESP_CFG_MAX_PWD_LENGTH
Maximum length of PWD for access point.

Note This parameter must include trailling zero

ESP_CFG_CONN_POLL_INTERVAL
Poll interval for connections in units of milliseconds.

Value indicates interval time to call poll event on active connections.

Note Single poll interval applies for all connections

ESP_CFG_CONN_MANUAL_TCP_RECEIVE
Enables `1` or disables `0` manual `TCP` data receive from ESP device.

Normally ESP automatically sends received TCP data to host device in async mode. When host device is slow or if there is memory constrain, it may happen that processing cannot handle all received data.

When feature is enabled, ESP will notify host device about new data available for read and then user may start read process

Note This feature is only available for `TCP` connections.

ESP_MIN_AT_VERSION_MAJOR_ESP8266
Minimal major version for ESP8266

ESP_MIN_AT_VERSION_MINOR_ESP8266
Minimal minor version for ESP8266

ESP_MIN_AT_VERSION_PATCH_ESP8266
Minimal patch version for ESP8266

ESP_MIN_AT_VERSION_MAJOR_ESP32
Minimal major version for ESP32

ESP_MIN_AT_VERSION_MINOR_ESP32
Minimal minor version for ESP32

ESP_MIN_AT_VERSION_PATCH_ESP32
Minimal patch version for ESP32

group **ESP_CONFIG_DBG**
Debugging configurations.

Defines

ESP_CFG_DBG
Set global debug support.

Possible values are *ESP_DBG_ON* or *ESP_DBG_OFF*

Note Set to *ESP_DBG_OFF* to globally disable all debugs

ESP_CFG_DBG_OUT(fmt, ...)
Debugging output function.

Called with format and optional parameters for printf-like debug

ESP_CFG_DBG_LVL_MIN
Minimal debug level.

Check ESP_DBG_LVL for possible values

ESP_CFG_DBG_TYPES_ON
Enabled debug types.

When debug is globally enabled with *ESP_CFG_DBG* parameter, user must enable debug types such as TRACE or STATE messages.

Check ESP_DBG_TYPE for possible options. Separate values with `bitwise OR` operator

ESP_CFG_DBG_INIT
Set debug level for init function.

Possible values are *ESP_DBG_ON* or *ESP_DBG_OFF*

ESP_CFG_DBG_MEM
Set debug level for memory manager.

Possible values are *ESP_DBG_ON* or *ESP_DBG_OFF*

ESP_CFG_DBG_INPUT
Set debug level for input module.

Possible values are *ESP_DBG_ON* or *ESP_DBG_OFF*

ESP_CFG_DBG_THREAD
Set debug level for ESP threads.

Possible values are *ESP_DBG_ON* or *ESP_DBG_OFF*

ESP_CFG_DBG_ASSERT
Set debug level for asserting of input variables.

Possible values are *ESP_DBG_ON* or *ESP_DBG_OFF*

ESP_CFG_DBG_IPD
Set debug level for incoming data received from device.

Possible values are *ESP_DBG_ON* or *ESP_DBG_OFF*

ESP_CFG_DBG_NETCONN
Set debug level for netconn sequential API.

Possible values are *ESP_DBG_ON* or *ESP_DBG_OFF*

ESP_CFG_DBG_PBUF
Set debug level for packet buffer manager.

Possible values are *ESP_DBG_ON* or *ESP_DBG_OFF*

ESP_CFG_DBG_CONN
Set debug level for connections.

Possible values are *ESP_DBG_ON* or *ESP_DBG_OFF*

ESP_CFG_DBG_VAR
Set debug level for dynamic variable allocations.

Possible values are *ESP_DBG_ON* or *ESP_DBG_OFF*

ESP_CFG_AT_ECHO
Enables `1` or disables `0` echo mode on AT commands sent to ESP device.

Note This mode is useful when debugging ESP communication

group **ESP_CONFIG_OS**
Operating system dependant configuration.

Defines

ESP_CFG_THREAD_PRODUCER_MBOX_SIZE
Set number of message queue entries for procuder thread.

Message queue is used for storing memory address to command data

ESP_CFG_THREAD_PROCESS_MBOX_SIZE
Set number of message queue entries for processing thread.

Message queue is used to notify processing thread about new received data on AT port

ESP_CFG_INPUT_USE_PROCESS
Enables `1` or disables `0` direct support for processing input data.

When this mode is enabled, no overhead is included for copying data to receive buffer because bytes are processed directly by *esp_input_process* function

If this mode is not enabled, then user have to send every received byte via *esp_input* function to the internal buffer for future processing. This may introduce additional overhead with data copy and may decrease library performance

Note This mode can only be used when *ESP_CFG_OS* is enabled

Note When using this mode, separate thread must be dedicated only for reading data on AT port. It is usually implemented in LL driver

Note Best case for using this mode is if DMA receive is supported by host device

ESP_THREAD_PRODUCER_HOOK()
Producer thread hook, called each time thread wakes-up and does the processing.

It can be used to check if thread is alive.

ESP_THREAD_PROCESS_HOOK()
Process thread hook, called each time thread wakes-up and does the processing.

It can be used to check if thread is alive.

group **ESP_CONFIG_STD_LIB**
Standard C library configuration.

Configuration allows you to overwrite default C language function in case of better implementation with hardware (for example DMA for data copy).

Defines

ESP_MEMCPY(dst, src, len)
Memory copy function declaration.

User is able to change the memory function, in case hardware supports copy operation, it may implement its own

Function prototype must be similar to:

```
void * my_memcpy(void* dst, const void* src, size_t len);
```

Return Destination memory start address

Parameters

- `[in]` `dst`: Destination memory start address
- `[in]` `src`: Source memory start address
- `[in]` `len`: Number of bytes to copy

ESP_MEMSET(dst, b, len)
Memory set function declaration.

Function prototype must be similar to:

```
void * my_memset(void* dst, int b, size_t len);
```

Return Destination memory start address

Parameters

- `[in]` `dst`: Destination memory start address
- `[in]` `b`: Value (byte) to set in memory
- `[in]` `len`: Number of bytes to set

group **ESP_CONFIG_MODULES**
Configuration of specific modules.

Defines

ESP_CFG_DNS
Enables `1` or disables `0` support for DNS functions.

ESP_CFG_WPS
Enables `1` or disables `0` support for WPS functions.

ESP_CFG_SNTP
Enables `1` or disables `0` support for SNTP protocol with AT commands.

ESP_CFG_HOSTNAME
Enables 1 or disables 0 support for hostname with AT commands.

ESP_CFG_PING
Enables 1 or disables 0 support for ping functions.

ESP_CFG_MDNS
Enables 1 or disables 0 support for mDNS.

ESP_CFG_SMART
Enables 1 or disables 0 support for SMART config.

group **ESP_CONFIG_MODULES_NETCONN**
Configuration of netconn API module.

Defines

ESP_CFG_NETCONN
Enables 1 or disables 0 NETCONN sequential API support for OS systems.

Note To use this feature, OS support is mandatory.

See *ESP_CFG_OS*

ESP_CFG_NETCONN_RECEIVE_TIMEOUT
Enables 1 or disables 0 receive timeout feature.

When this option is enabled, user will get an option to set timeout value for receive data on netconn, before function returns timeout error.

Note Even if this option is enabled, user must still manually set timeout, by default time will be set to 0 which means no timeout.

ESP_CFG_NETCONN_ACCEPT_QUEUE_LEN
Accept queue length for new client when netconn server is used.

Defines number of maximal clients waiting in accept queue of server connection

ESP_CFG_NETCONN_RECEIVE_QUEUE_LEN
Receive queue length for pbuf entries.

Defines maximal number of pbuf data packet references for receive

group **ESP_CONFIG_MODULES_MQTT**
Configuration of MQTT and MQTT API client modules.

Defines

ESP_CFG_MQTT_MAX_REQUESTS
Maximal number of open MQTT requests at a time.

ESP_CFG_DBG_MQTT
Set debug level for MQTT client module.

Possible values are *ESP_DBG_ON* or *ESP_DBG_OFF*

ESP_CFG_DBG_MQTT_API
Set debug level for MQTT API client module.

Possible values are *ESP_DBG_ON* or *ESP_DBG_OFF*

group **ESP_CONFIG_MODULES_CAYENNE**
Configuration of Cayenne MQTT client.

Defines

ESP_CFG_DBG_CAYENNE
Set debug level for Cayenne MQTT client module.

Possible values are *ESP_DBG_ON* or *ESP_DBG_OFF*

group **ESP_CONFIG_APP_HTTP**
Configuration of HTTP server app.

Defines

ESP_CFG_DBG_SERVER
Server debug default setting.

HTTP_SSI_TAG_START
SSI tag start string

HTTP_SSI_TAG_START_LEN
SSI tag start length

HTTP_SSI_TAG_END
SSI tag end string

HTTP_SSI_TAG_END_LEN
SSI tag end length

HTTP_SSI_TAG_MAX_LEN
Maximal length of tag name excluding start and end parts of tag.

HTTP_SUPPORT_POST
Enables `1` or disables `0` support for POST request.

HTTP_MAX_URI_LEN
Maximal length of allowed uri length including parameters in format `/uri/sub/path?param=value`

HTTP_MAX_PARAMS
Maximal number of parameters in URI.

HTTP_USE_METHOD_NOTALLOWED_RESP
Enables `1` or disables `0` method not allowed response.

Response is used in case user makes HTTP request with method which is not on the list of allowed methods. See *http_req_method_t*

Note When disabled, connection will be closed without response

HTTP_USE_DEFAULT_STATIC_FILES
Enables `1` or disables `1` default static files.

To allow fast startup of server development, several static files are included by default:

- /index.html
- /index.shtml
- /js/style.css

- /js/js.js

HTTP_DYNAMIC_HEADERS
Enables `1` or disables `0` dynamic headers support.

With dynamic headers enabled, script will try to detect most common file extensions and will try to response with:

- HTTP response code as first line
- Server name as second line
- Content type as third line including end of headers (empty line)

HTTP_DYNAMIC_HEADERS_CONTENT_LEN
Enables `1` or disables `0` content length header for response.

If response has fixed length without SSI tags, dynamic headers will try to include "Content-Length" header as part of response message sent to client

Note In order to use this, *HTTP_DYNAMIC_HEADERS* must be enabled

HTTP_SERVER_NAME
Default server name for `Server: x` response dynamic header.

5.3.3 Platform specific

List of all the modules:

Low-Level functions

Low-level module consists of callback-only functions, which are called by middleware and must be implemented by final application.

Tip: Check *Porting guide* for actual implementation

group **ESP_LL**
Low-level communication functions.

Typedefs

typedef size_t (***esp_ll_send_fn**)(**const** void *data, size_t len)
Function prototype for AT output data.

Return Number of bytes sent

Parameters

- `[in] data`: Pointer to data to send. This parameter can be set to `NULL`
- `[in] len`: Number of bytes to send. This parameter can be set to `0` to indicate that internal buffer can be flushed to stream. This is implementation defined and feature might be ignored

typedef uint8_t (***esp_ll_reset_fn**)(uint8_t state)
Function prototype for hardware reset of ESP device.

Return `1` on successful action, `0` otherwise

Parameters

- `[in] state`: State indicating reset. When set to `1`, reset must be active (usually pin active low), or set to `0` when reset is cleared

Functions

espr_t **esp_ll_init** (*esp_ll_t *ll*)
Callback function called from initialization process.

Note This function may be called multiple times if AT baudrate is changed from application. It is important that every configuration except AT baudrate is configured only once!

Note This function may be called from different threads in ESP stack when using OS. When *ESP_CFG_INPUT_USE_PROCESS* is set to `1`, this function may be called from user UART thread.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[inout] ll`: Pointer to *esp_ll_t* structure to fill data for communication functions

espr_t **esp_ll_deinit** (*esp_ll_t *ll*)
Callback function to de-init low-level communication part.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[inout] ll`: Pointer to *esp_ll_t* structure to fill data for communication functions

struct esp_ll_t
#include <*esp_typedefs.h*> Low level user specific functions.

Public Members

esp_ll_send_fn **send_fn**
Callback function to transmit data

esp_ll_reset_fn **reset_fn**
Reset callback function

uint32_t **baudrate**
UART baudrate value

struct *esp_ll_t*::**[anonymous]** **uart**
UART communication parameters

System functions

System functions are bridge between operating system system calls and middleware system calls. Middleware is tightly coupled with operating system features hence it is important to include OS features directly.

It includes support for:

- Thread management, to start/stop threads
- Mutex management for recursive mutexes
- Semaphore management for binary-only semaphores
- Message queues for thread-safe data exchange between threads
- Core system protection for mutual exclusion to access shared resources

Tip: Check *Porting guide* for actual implementation guidelines.

group **ESP_SYS**
System based function for OS management, timings, etc.

Main

uint8_t **esp_sys_init**(void)
Init system dependant parameters.

After this function is called, all other system functions must be fully ready.

Return `1` on success, `0` otherwise

uint32_t **esp_sys_now**(void)
Get current time in units of milliseconds.

Return Current time in units of milliseconds

uint8_t **esp_sys_protect**(void)
Protect middleware core.

Stack protection must support recursive mode. This function may be called multiple times, even if access has been granted before.

Note Most operating systems support recursive mutexes.

Return `1` on success, `0` otherwise

uint8_t **esp_sys_unprotect**(void)
Unprotect middleware core.

This function must follow number of calls of *esp_sys_protect* and unlock access only when counter reached back zero.

Note Most operating systems support recursive mutexes.

Return `1` on success, `0` otherwise

Mutex

uint8_t **esp_sys_mutex_create**(*esp_sys_mutex_t* *p)
Create new recursive mutex.

Note Recursive mutex has to be created as it may be locked multiple times before unlocked

Return 1 on success, 0 otherwise

Parameters

- [out] p: Pointer to mutex structure to allocate

uint8_t **esp_sys_mutex_delete**(*esp_sys_mutex_t* *p)
Delete recursive mutex from system.

Return 1 on success, 0 otherwise

Parameters

- [in] p: Pointer to mutex structure

uint8_t **esp_sys_mutex_lock**(*esp_sys_mutex_t* *p)
Lock recursive mutex, wait forever to lock.

Return 1 on success, 0 otherwise

Parameters

- [in] p: Pointer to mutex structure

uint8_t **esp_sys_mutex_unlock**(*esp_sys_mutex_t* *p)
Unlock recursive mutex.

Return 1 on success, 0 otherwise

Parameters

- [in] p: Pointer to mutex structure

uint8_t **esp_sys_mutex_isvalid**(*esp_sys_mutex_t* *p)
Check if mutex structure is valid system.

Return 1 on success, 0 otherwise

Parameters

- [in] p: Pointer to mutex structure

uint8_t **esp_sys_mutex_invalid**(*esp_sys_mutex_t* *p)
Set recursive mutex structure as invalid.

Return 1 on success, 0 otherwise

Parameters

- [in] p: Pointer to mutex structure

Semaphores

uint8_t **esp_sys_sem_create**(*esp_sys_sem_t* *p, uint8_t *cnt*)
Create a new binary semaphore and set initial state.

Note Semaphore may only have 1 token available

Return 1 on success, 0 otherwise

Parameters

- [out] p: Pointer to semaphore structure to fill with result
- [in] cnt: Count indicating default semaphore state: 0: Take semaphore token immediately 1: Keep token available

uint8_t **esp_sys_sem_delete**(*esp_sys_sem_t* *p)
Delete binary semaphore.

Return 1 on success, 0 otherwise

Parameters

- [in] p: Pointer to semaphore structure

uint32_t **esp_sys_sem_wait**(*esp_sys_sem_t* *p, uint32_t *timeout*)
Wait for semaphore to be available.

Return Number of milliseconds waited for semaphore to become available or *ESP_SYS_TIMEOUT* if not available within given time

Parameters

- [in] p: Pointer to semaphore structure
- [in] timeout: Timeout to wait in milliseconds. When 0 is applied, wait forever

uint8_t **esp_sys_sem_release**(*esp_sys_sem_t* *p)
Release semaphore.

Return 1 on success, 0 otherwise

Parameters

- [in] p: Pointer to semaphore structure

uint8_t **esp_sys_sem_isvalid**(*esp_sys_sem_t* *p)
Check if semaphore is valid.

Return 1 on success, 0 otherwise

Parameters

- [in] p: Pointer to semaphore structure

uint8_t **esp_sys_sem_invalid**(*esp_sys_sem_t* *p)
Invalid semaphore.

Return 1 on success, 0 otherwise

Parameters

- `[in] p`: Pointer to semaphore structure

Message queues

uint8_t **esp_sys_mbox_create**(*esp_sys_mbox_t* *b, size_t *size*)
Create a new message queue with entry type of `void *`

Return `1` on success, `0` otherwise

Parameters

- `[out] b`: Pointer to message queue structure
- `[in] size`: Number of entries for message queue to hold

uint8_t **esp_sys_mbox_delete**(*esp_sys_mbox_t* *b)
Delete message queue.

Return `1` on success, `0` otherwise

Parameters

- `[in] b`: Pointer to message queue structure

uint32_t **esp_sys_mbox_put**(*esp_sys_mbox_t* *b, void *m)
Put a new entry to message queue and wait until memory available.

Return Time in units of milliseconds needed to put a message to queue

Parameters

- `[in] b`: Pointer to message queue structure
- `[in] m`: Pointer to entry to insert to message queue

uint32_t **esp_sys_mbox_get**(*esp_sys_mbox_t* *b, void **m, uint32_t *timeout*)
Get a new entry from message queue with timeout.

Return Time in units of milliseconds needed to put a message to queue or *ESP_SYS_TIMEOUT* if it was not successful

Parameters

- `[in] b`: Pointer to message queue structure
- `[in] m`: Pointer to pointer to result to save value from message queue to
- `[in] timeout`: Maximal timeout to wait for new message. When `0` is applied, wait for unlimited time

uint8_t **esp_sys_mbox_putnow**(*esp_sys_mbox_t* *b, void *m)
Put a new entry to message queue without timeout (now or fail)

Return `1` on success, `0` otherwise

Parameters

- `[in] b`: Pointer to message queue structure
- `[in] m`: Pointer to message to save to queue

uint8_t **esp_sys_mbox_getnow**(*esp_sys_mbox_t* *b, void **m)
Get an entry from message queue immediately.

Return 1 on success, 0 otherwise

Parameters

- [in] b: Pointer to message queue structure
- [in] m: Pointer to pointer to result to save value from message queue to

uint8_t **esp_sys_mbox_isvalid**(*esp_sys_mbox_t* *b)
Check if message queue is valid.

Return 1 on success, 0 otherwise

Parameters

- [in] b: Pointer to message queue structure

uint8_t **esp_sys_mbox_invalid**(*esp_sys_mbox_t* *b)
Invalid message queue.

Return 1 on success, 0 otherwise

Parameters

- [in] b: Pointer to message queue structure

Threads

uint8_t **esp_sys_thread_create**(*esp_sys_thread_t* *t, **const** char *name, *esp_sys_thread_fn thread_func*, void ***const** *arg*, size_t *stack_size*, *esp_sys_thread_prio_t prio*)
Create a new thread.

Return 1 on success, 0 otherwise

Parameters

- [out] t: Pointer to thread identifier if create was successful. It may be set to NULL
- [in] name: Name of a new thread
- [in] thread_func: Thread function to use as thread body
- [in] arg: Thread function argument
- [in] stack_size: Size of thread stack in uints of bytes. If set to 0, reserve default stack size
- [in] prio: Thread priority

uint8_t **esp_sys_thread_terminate**(*esp_sys_thread_t* *t)
Terminate thread (shut it down and remove)

Return 1 on success, 0 otherwise

Parameters

- [in] t: Pointer to thread handle to terminate. If set to NULL, terminate current thread (thread from where function is called)

uint8_t **esp_sys_thread_yield**(void)
Yield current thread.

Return `1` on success, `0` otherwise

Defines

ESP_SYS_MUTEX_NULL
Mutex invalid value.

Value assigned to *esp_sys_mutex_t* type when it is not valid.

ESP_SYS_SEM_NULL
Semaphore invalid value.

Value assigned to *esp_sys_sem_t* type when it is not valid.

ESP_SYS_MBOX_NULL
Message box invalid value.

Value assigned to *esp_sys_mbox_t* type when it is not valid.

ESP_SYS_TIMEOUT
OS timeout value.

Value returned by operating system functions (mutex wait, sem wait, mbox wait) when it returns timeout and does not give valid value to application

ESP_SYS_THREAD_PRIO
Default thread priority value used by middleware to start built-in threads.

Threads can well operate with normal (default) priority and do not require any special feature in terms of priority for prioer operation.

ESP_SYS_THREAD_SS
Stack size in units of bytes for system threads.

It is used as default stack size for all built-in threads.

Typedefs

typedef void (***esp_sys_thread_fn**)(void *)
Thread function prototype.

typedef osMutexId_t **esp_sys_mutex_t**
System mutex type.

It is used by middleware as base type of mutex.

typedef osSemaphoreId_t **esp_sys_sem_t**
System semaphore type.

It is used by middleware as base type of mutex.

typedef osMessageQueueId_t **esp_sys_mbox_t**
System message queue type.

It is used by middleware as base type of mutex.

typedef osThreadId_t **esp_sys_thread_t**
System thread ID type.

typedef osPriority **esp_sys_thread_prio_t**
System thread priority type.

It is used as priority type for system function, to start new threads by middleware.

5.3.4 Applications

Cayenne MQTT API

group **ESP_APP_CAYENNE_API**
MQTT client API for Cayenne.

Defines

ESP_CAYENNE_API_VERSION
Cayenne API version in string.

ESP_CAYENNE_HOST
Cayenne host server.

ESP_CAYENNE_PORT
Cayenne port number.

ESP_CAYENNE_NO_CHANNEL
No channel macro

ESP_CAYENNE_ALL_CHANNELS
All channels macro

Typedefs

typedef espr_t (*__esp_cayenne_evt_fn__) (**struct** esp_cayenne *c, *esp_cayenne_evt_t* *evt)
Cayenne event callback function.

Return *espOK* on success, member of *espr_t* otherwise

Parameters

- `[in] c`: Cayenne handle
- `[in] evt`: Event handle

Enums

enum esp_cayenne_topic_t
List of possible cayenne topics.

Values:

ESP_CAYENNE_TOPIC_DATA
Data topic

ESP_CAYENNE_TOPIC_COMMAND
Command topic

ESP_CAYENNE_TOPIC_CONFIG

ESP_CAYENNE_TOPIC_RESPONSE

ESP_CAYENNE_TOPIC_SYS_MODEL

ESP_CAYENNE_TOPIC_SYS_VERSION

ESP_CAYENNE_TOPIC_SYS_CPU_MODEL

ESP_CAYENNE_TOPIC_SYS_CPU_SPEED

ESP_CAYENNE_TOPIC_DIGITAL

ESP_CAYENNE_TOPIC_DIGITAL_COMMAND

ESP_CAYENNE_TOPIC_DIGITAL_CONFIG

ESP_CAYENNE_TOPIC_ANALOG

ESP_CAYENNE_TOPIC_ANALOG_COMMAND

ESP_CAYENNE_TOPIC_ANALOG_CONFIG

ESP_CAYENNE_TOPIC_END
Last entry

enum esp_cayenne_resp_t
Cayenne response types.

Values:

ESP_CAYENNE_RESP_OK
Response OK

ESP_CAYENNE_RESP_ERROR
Response error

enum esp_cayenne_evt_type_t
Cayenne events.

Values:

ESP_CAYENNE_EVT_CONNECT
Connect to Cayenne event

ESP_CAYENNE_EVT_DISCONNECT
Disconnect from Cayenne event

ESP_CAYENNE_EVT_DATA
Data received event

Functions

espr_t **esp_cayenne_create**(*esp_cayenne_t* *c, **const** *esp_mqtt_client_info_t* *client_info, *esp_cayenne_evt_fn* evt_fn)
Create new instance of cayenne MQTT connection.

Note Each call to this functions starts new thread for async receive processing. Function will block until thread is created and successfully started

Return *espOK* on success, member of *espr_t* otherwise

Parameters

- `[in] c`: Cayenne empty handle

- `[in] client_info`: MQTT client info with username, password and id
- `[in] evt_fn`: Event function

espr_t **esp_cayenne_subscribe**(*esp_cayenne_t* *c, esp_cayenne_topic_t *topic*, uint16_t *channel*)
Subscribe to cayenne based topics and channels.

Return *espOK* on success, member of *espr_t* otherwise

Parameters

- `[in] c`: Cayenne handle
- `[in] topic`: Cayenne topic
- `[in] channel`: Optional channel number. Use *ESP_CAYENNE_NO_CHANNEL* when channel is not needed or ESP_CAYENNE_ALL_CHANNELS to subscribe to all channels

espr_t **esp_cayenne_publish_data**(*esp_cayenne_t* *c, esp_cayenne_topic_t *topic*, uint16_t *channel*, **const** char *type, **const** char *unit, **const** char *data)

espr_t **esp_cayenne_publish_float**(*esp_cayenne_t* *c, esp_cayenne_topic_t *topic*, uint16_t *channel*, **const** char *type, **const** char *unit, float *f*)

espr_t **esp_cayenne_publish_response**(*esp_cayenne_t* *c, *esp_cayenne_msg_t* *msg, esp_cayenne_resp_t *resp*, **const** char *message)
Publish response message to command.

Return *espOK* on success, member of *espr_t* otherwise

Parameters

- `[in] c`: Cayenne handle
- `[in] msg`: Received message with command topic
- `[in] resp`: Response type, either `OK` or `ERROR`
- `[in] message`: Message text in case of error to be displayed to Cayenne dashboard

struct esp_cayenne_key_value_t
#include <*esp_cayenne.h*> Key/Value pair structure.

Public Members

const char ***key**
Key string

const char ***value**
Value string

struct esp_cayenne_msg_t
#include <*esp_cayenne.h*> Cayenne message.

Public Members

esp_cayenne_topic_t **topic**
Message topic

uint16_t **channel**
Message channel, optional, based on topic type

const char ***seq**
Sequence string on command

esp_cayenne_key_value_t **values**[2]
Key/Value pair of values

size_t **values_count**
Count of valid pairs in values member

struct esp_cayenne_evt_t
#include <*esp_cayenne.h*> Cayenne event.

Public Members

esp_cayenne_evt_type_t **type**
Event type

esp_cayenne_msg_t ***msg**
Pointer to data message

struct *esp_cayenne_evt_t*::**[anonymous]**::**[anonymous] data**
Data event, used with *ESP_CAYENNE_EVT_DATA* event

union *esp_cayenne_evt_t*::**[anonymous] evt**
Event union

struct esp_cayenne_t
#include <*esp_cayenne.h*> Cayenne handle.

Public Members

esp_mqtt_client_api_p **api_c**
MQTT API client

const *esp_mqtt_client_info_t* ***info_c**
MQTT Client info structure

esp_cayenne_msg_t **msg**
Received data message

esp_cayenne_evt_t **evt**
Event handle

esp_cayenne_evt_fn **evt_fn**
Event callback function

esp_sys_thread_t **thread**
Cayenne thread handle

esp_sys_sem_t **sem**
Sync semaphore handle

HTTP Server

group **ESP_APP_HTTP_SERVER**
HTTP server based on callback API.

Defines

HTTP_MAX_HEADERS
Maximal number of headers we can control.

esp_http_server_write_string(hs, str)
Write string to HTTP server output.

Note May only be called from SSI callback function

Return Number of bytes written to output

See *esp_http_server_write*

Parameters

- `[in] hs`: HTTP handle
- `[in] str`: String to write

Typedefs

typedef char *(***http_cgi_fn**)(*http_param_t* *params, size_t params_len)
CGI callback function.

Return Function must return a new URI which is used later as response string, such as "/index.html" or similar

Parameters

- `[in] params`: Pointer to list of parameteres and their values
- `[in] params_len`: Number of parameters

typedef espr_t (***http_post_start_fn**)(**struct** http_state *hs, **const** char *uri, uint32_t content_length)
Post request started with non-zero content length function prototype.

Return *espOK* on success, member of *espr_t* otherwise

Parameters

- `[in] hs`: HTTP state
- `[in] uri`: POST request URI
- `[in] content_length`: Total content length (Content-Length HTTP parameter) in units of bytes

typedef espr_t (***http_post_data_fn**)(**struct** http_state *hs, *esp_pbuf_p* pbuf)
Post data received on request function prototype.

Note This function may be called multiple time until content_length from *http_post_start_fn* callback is not reached

Return *espOK* on success, member of *espr_t* otherwise

Parameters

- `[in] hs`: HTTP state
- `[in] pbuf`: Packet buffer wit reciveed data

typedef espr_t (***http_post_end_fn**)(**struct** http_state *hs)
End of POST data request function prototype.

Return *espOK* on success, member of *espr_t* otherwise

Parameters

- `[in] hs`: HTTP state

typedef size_t (***http_ssi_fn**)(**struct** http_state *hs, **const** char *tag_name, size_t tag_len)
SSI (Server Side Includes) callback function prototype.

Note User can use server write functions to directly write to connection output

Parameters

- `[in] hs`: HTTP state
- `[in] tag_name`: Name of TAG to replace with user content
- `[in] tag_len`: Length of TAG

Return Value

- `1`: Everything was written on this tag
- `0`: There are still data to write to output which means callback will be called again for user to process all the data

typedef uint8_t (***http_fs_open_fn**)(**struct** http_fs_file *file, **const** char *path)
File system open file function Function is called when user file system (FAT or similar) should be invoked to open a file from specific path.

Return 1 if file is opened, 0 otherwise

Parameters

- `[in] file`: Pointer to file where user has to set length of file if opening was successful
- `[in] path`: Path of file to open

typedef uint32_t (***http_fs_read_fn**)(**struct** http_fs_file *file, void *buff, size_t btr)
File system read file function Function may be called for `2` purposes. First is to read data and second to get remaining length of file to read.

Return Number of bytes read or number of bytes available to read

Parameters

- `[in] file`: File pointer to read content
- `[in] buff`: Buffer to read data to. When parameter is set to NULL, number of remaining bytes available to read should be returned

- `[in] btr`: Number of bytes to read from file. This parameter has no meaning when buff is NULL

typedef uint8_t (***http_fs_close_fn**)(**struct** http_fs_file *file)
Close file callback function.

Return `1` on success, `0` otherwise

Parameters

- `[in] file`: File to close

Enums

enum http_req_method_t
Request method type.

Values:

HTTP_METHOD_NOTALLOWED
HTTP method is not allowed

HTTP_METHOD_GET
HTTP request method GET

HTTP_METHOD_POST
HTTP request method POST

enum http_ssi_state_t
List of SSI TAG parsing states.

Values:

HTTP_SSI_STATE_WAIT_BEGIN = 0x00
Waiting beginning of tag

HTTP_SSI_STATE_BEGIN = 0x01
Beginning detected, parsing it

HTTP_SSI_STATE_TAG = 0x02
Parsing TAG value

HTTP_SSI_STATE_END = 0x03
Parsing end of TAG

Functions

espr_t **esp_http_server_init**(**const** *http_init_t* **init*, *esp_port_t port*)
Initialize HTTP server at specific port.

Return *espOK* on success, member of *espr_t* otherwise

Parameters

- `[in] init`: Initialization structure for server
- `[in] port`: Port for HTTP server, usually 80

size_t **esp_http_server_write**(*http_state_t* **hs*, **const** void **data*, size_t *len*)
Write data directly to connection from callback.

Note This function may only be called from SSI callback function for HTTP server

Return Number of bytes written

Parameters

- `[in] hs`: HTTP state
- `[in] data`: Data to write
- `[in] len`: Length of bytes to write

struct http_param_t

#include <*esp_http_server.h*> HTTP parameters on http URI in format `?param1=value1&param2=value2&...`

Public Members

const char ***name**

Name of parameter

const char ***value**

Parameter value

struct http_cgi_t

#include <*esp_http_server.h*> CGI structure to register handlers on URI paths.

Public Members

const char ***uri**

URI path for CGI handler

http_cgi_fn **fn**

Callback function to call when we have a CGI match

struct http_init_t

#include <*esp_http_server.h*> HTTP server initialization structure.

Public Members

http_post_start_fn **post_start_fn**

Callback function for post start

http_post_data_fn **post_data_fn**

Callback functon for post data

http_post_end_fn **post_end_fn**

Callback functon for post end

const *http_cgi_t* ***cgi**

Pointer to array of CGI entries. Set to NULL if not used

size_t **cgi_count**

Length of CGI array. Set to 0 if not used

http_ssi_fn **ssi_fn**

SSI callback function

http_fs_open_fn **fs_open**
Open file function callback

http_fs_read_fn **fs_read**
Read file function callback

http_fs_close_fn **fs_close**
Close file function callback

struct http_fs_file_table_t
#include <esp_http_server.h> HTTP file system table structure of static files in device memory.

Public Members

const char ***path**
File path, ex. "/index.html"

const void ***data**
Pointer to file data

uint32_t **size**
Size of file in units of bytes

struct http_fs_file_t
#include <esp_http_server.h> HTTP response file structure.

Public Members

const uint8_t ***data**
Pointer to data array in case file is static

uint8_t **is_static**
Flag indicating file is static and no dynamic read is required

uint32_t **size**
Total length of file

uint32_t **fptr**
File pointer to indicate next read position

const uint16_t ***rem_open_files**
Pointer to number of remaining open files. User can use value on this pointer to get number of other opened files

void ***arg**
User custom argument, may be used for user specific file system object

struct http_state_t
#include <esp_http_server.h> HTTP state structure.

Public Members

esp_conn_p **conn**
Connection handle

esp_pbuf_p **p**
Header received pbuf chain

size_t **conn_mem_available**
Available memory in connection send queue

uint32_t **written_total**
Total number of bytes written into send buffer

uint32_t **sent_total**
Number of bytes we already sent

http_req_method_t **req_method**
Used request method

uint8_t **headers_received**
Did we fully received a headers?

uint8_t **process_resp**
Process with response flag

uint32_t **content_length**
Total expected content length for request (on POST) (without headers)

uint32_t **content_received**
Content length received so far (POST request, without headers)

http_fs_file_t **resp_file**
Response file structure

uint8_t **resp_file_opened**
Status if response file is opened and ready

const uint8_t ***buff**
Buffer pointer with data

uint32_t **buff_len**
Total length of buffer

uint32_t **buff_ptr**
Current buffer pointer

void ***arg**
User optional argument

const char ***dyn_hdr_strs**[4]
Pointer to constant strings for dynamic header outputs

size_t **dyn_hdr_idx**
Current header for processing on output

size_t **dyn_hdr_pos**
Current position in current index for output

char **dyn_hdr_cnt_len**[30]
Content length header response: “Content-Length: 0123456789\r\n”

uint8_t **is_ssi**
Flag if current request is SSI enabled

http_ssi_state_t **ssi_state**
Current SSI state when parsing SSI tags

char **ssi_tag_buff**[5 + 3 + 10 + 1]
Temporary buffer for SSI tag storing

size_t **ssi_tag_buff_ptr**
Current write pointer

size_t **ssi_tag_buff_written**
Number of bytes written so far to output buffer in case tag is not valid

size_t **ssi_tag_len**
Length of SSI tag

size_t **ssi_tag_process_more**
Set to `1` when we have to process tag multiple times

group **ESP_APP_HTTP_SERVER_FS_FAT**
FATFS file system implementation for dynamic files.

Functions

uint8_t **http_fs_open**(*http_fs_file_t* *file, **const** char *path)
Open a file of specific path.

Return `1` on success, `0` otherwise

Parameters

- `[in] file`: File structure to fill if file is successfully open
- `[in] path`: File path to open in format "/js/scripts.js" or "/index.html"

uint32_t **http_fs_read**(*http_fs_file_t* *file, void *buff, size_t btr)
Read a file content.

Return Number of bytes read or number of bytes available to read

Parameters

- `[in] file`: File handle to read
- `[out] buff`: Buffer to read data to. When set to NULL, function should return remaining available data to read
- `[in] btr`: Number of bytes to read. Has no meaning when buff = NULL

uint8_t **http_fs_close**(*http_fs_file_t* *file)
Close a file handle.

Return `1` on success, `0` otherwise

Parameters

- `[in] file`: File handle

MQTT Client

MQTT client v3.1.1 implementation, based on callback (non-netconn) connection API.

Listing 23: MQTT application example code

```
/*
 * MQTT client example with ESP device.
 *
 * Once device is connected to network,
 * it will try to connect to mosquitto test server and start the MQTT.
 *
 * If successfully connected, it will publish data to "esp8266_mqtt_topic" topic every x seconds.
 *
 * To check if data are sent, you can use mqtt-spy PC software to inspect
 * test.mosquitto.org server and subscribe to publishing topic
 */

#include "esp/apps/esp_mqtt_client.h"
#include "esp/esp.h"
#include "esp/esp_timeout.h"
#include "mqtt_client.h"

/**
 * \brief           MQTT client structure
 */
static esp_mqtt_client_p
mqtt_client;

/**
 * \brief           Client ID is structured from ESP station MAC address
 */
static char
mqtt_client_id[13];

/**
 * \brief           Connection information for MQTT CONNECT packet
 */
static const esp_mqtt_client_info_t
mqtt_client_info = {
    .id = mqtt_client_id,                       /* The only required field for connection! */

    .keep_alive = 10,
    // .user = "test_username",
    // .pass = "test_password",
};

static void mqtt_cb(esp_mqtt_client_p client, esp_mqtt_evt_t* evt);
static void example_do_connect(esp_mqtt_client_p client);
static uint32_t retries = 0;

/**
 * \brief           Custom callback function for ESP events
 */
static espr_t
mqtt_esp_cb(esp_evt_t* evt) {
```

(continues on next page)

(continued from previous page)

```
    switch (esp_evt_get_type(evt)) {
#if ESP_CFG_MODE_STATION
        case ESP_EVT_WIFI_GOT_IP: {
            example_do_connect(mqtt_client);    /* Start connection after we have a connection to network client */
            break;
        }
#endif /* ESP_CFG_MODE_STATION */
        default: break;
    }
    return espOK;
}

/**
 * \brief           MQTT client thread
 * \param[in]       arg: User argument
 */
void
mqtt_client_thread(void const* arg) {
    esp_mac_t mac;

    esp_evt_register(mqtt_esp_cb);              /* Register new callback for general events from ESP stack */

    /* Get station MAC to format client ID */
    if (esp_sta_getmac(&mac, NULL, NULL, 1) == espOK) {
        snprintf(mqtt_client_id, sizeof(mqtt_client_id), "%02X%02X%02X%02X%02X%02X",
            (unsigned)mac.mac[0], (unsigned)mac.mac[1], (unsigned)mac.mac[2],
            (unsigned)mac.mac[3], (unsigned)mac.mac[4], (unsigned)mac.mac[5]
        );
    } else {
        strcpy(mqtt_client_id, "unknown");
    }
    printf("MQTT Client ID: %s\r\n", mqtt_client_id);

    /*
     * Create a new client with 256 bytes of RAW TX data
     * and 128 bytes of RAW incoming data
     */
    mqtt_client = esp_mqtt_client_new(256, 128);/* Create new MQTT client */
    if (esp_sta_is_joined()) {                  /* If ESP is already joined to network */
        example_do_connect(mqtt_client);        /* Start connection to MQTT server */
    }

    /* Make dummy delay of thread */
    while (1) {
        esp_delay(1000);
    }
}

/**
 * \brief           Timeout callback for MQTT events
 * \param[in]       arg: User argument
 */
void
mqtt_timeout_cb(void* arg) {
```

(continues on next page)

(continued from previous page)

```
    static uint32_t num = 10;
    esp_mqtt_client_p client = arg;
    espr_t res;

    static char tx_data[20];

    if (esp_mqtt_client_is_connected(client)) {
        sprintf(tx_data, "R: %u, N: %u", (unsigned)retries, (unsigned)num);
        if ((res = esp_mqtt_client_publish(client, "esp8266_mqtt_topic", tx_data, ESP_U16(strlen(tx_data)), ESP_MQTT_QOS_EXACTLY_ONCE, 0, (void *)num)) == espOK) {
            printf("Publishing %d...\r\n", (int)num);
            num++;
        } else {
            printf("Cannot publish...: %d\r\n", (int)res);
        }
    }
    esp_timeout_add(10000, mqtt_timeout_cb, client);
}

/**
 * \brief           MQTT event callback function
 * \param[in]       client: MQTT client where event occurred
 * \param[in]       evt: Event type and data
 */
static void
mqtt_cb(esp_mqtt_client_p client, esp_mqtt_evt_t* evt) {
    switch (esp_mqtt_client_evt_get_type(client, evt)) {
        /*
         * Connect event
         * Called if user successfully connected to MQTT server
         * or even if connection failed for some reason
         */
        case ESP_MQTT_EVT_CONNECT: {            /* MQTT connect event occurred */
            esp_mqtt_conn_status_t status = esp_mqtt_client_evt_connect_get_status(client, evt);

            if (status == ESP_MQTT_CONN_STATUS_ACCEPTED) {
                printf("MQTT accepted!\r\n");
                /*
                 * Once we are accepted by server,
                 * it is time to subscribe to different topics
                 * We will subscrive to "mqtt_esp_example_topic" topic,
                 * and will also set the same name as subscribe argument for callback later
                 */
                esp_mqtt_client_subscribe(client, "esp8266_mqtt_topic", ESP_MQTT_QOS_EXACTLY_ONCE, "esp8266_mqtt_topic");

                /* Start timeout timer after 5000ms and call mqtt_timeout_cb function */
                esp_timeout_add(5000, mqtt_timeout_cb, client);
            } else {
                printf("MQTT server connection was not successful: %d\r\n", (int)status);

                /* Try to connect all over again */
                example_do_connect(client);
```

(continues on next page)

(continued from previous page)

```
            }
            break;
        }

        /*
         * Subscribe event just happened.
         * Here it is time to check if it was successful or failed attempt
         */
        case ESP_MQTT_EVT_SUBSCRIBE: {
            const char* arg = esp_mqtt_client_evt_subscribe_get_argument(client, evt);
↪  /* Get user argument */
            espr_t res = esp_mqtt_client_evt_subscribe_get_result(client, evt); /*␣
↪Get result of subscribe event */

            if (res == espOK) {
                printf("Successfully subscribed to %s topic\r\n", arg);
                if (!strcmp(arg, "esp8266_mqtt_topic")) {   /* Check topic name we␣
↪were subscribed */
                    /* Subscribed to "esp8266_mqtt_topic" topic */

                    /*
                     * Now publish an even on example topic
                     * and set QoS to minimal value which does not guarantee message␣
↪delivery to received
                     */
                    esp_mqtt_client_publish(client, "esp8266_mqtt_topic", "test_data",
↪ 9, ESP_MQTT_QOS_AT_MOST_ONCE, 0, (void *)1);
                }
            }
            break;
        }

        /* Message published event occurred */
        case ESP_MQTT_EVT_PUBLISH: {
            uint32_t val = (uint32_t)esp_mqtt_client_evt_publish_get_argument(client,␣
↪evt); /* Get user argument, which is in fact our custom number */

            printf("Publish event, user argument on message was: %d\r\n", (int)val);
            break;
        }

        /*
         * A new message was published to us
         * and now it is time to read the data
         */
        case ESP_MQTT_EVT_PUBLISH_RECV: {
            const char* topic = esp_mqtt_client_evt_publish_recv_get_topic(client,␣
↪evt);
            size_t topic_len = esp_mqtt_client_evt_publish_recv_get_topic_len(client,␣
↪evt);
            const uint8_t* payload = esp_mqtt_client_evt_publish_recv_get_
↪payload(client, evt);
            size_t payload_len = esp_mqtt_client_evt_publish_recv_get_payload_
↪len(client, evt);

            ESP_UNUSED(payload);
            ESP_UNUSED(payload_len);
```

(continues on next page)

(continued from previous page)

```
            ESP_UNUSED(topic);
            ESP_UNUSED(topic_len);
            break;
        }

        /* Client is fully disconnected from MQTT server */
        case ESP_MQTT_EVT_DISCONNECT: {
            printf("MQTT client disconnected!\r\n");
            example_do_connect(client);         /* Connect to server all over again */
            break;
        }

        default:
            break;
    }
}

/** Make a connection to MQTT server in non-blocking mode */
static void
example_do_connect(esp_mqtt_client_p client) {
    if (client == NULL) {
        return;
    }

    /*
     * Start a simple connection to open source
     * MQTT server on mosquitto.org
     */
    retries++;
    esp_timeout_remove(mqtt_timeout_cb);
    esp_mqtt_client_connect(mqtt_client, "test.mosquitto.org", 1883, mqtt_cb, &mqtt_
↪client_info);
}
```

group **ESP_APP_MQTT_CLIENT**

MQTT client.

Typedefs

typedef struct esp_mqtt_client ***esp_mqtt_client_p**

Pointer to esp_mqtt_client_t structure.

typedef void (***esp_mqtt_evt_fn**) (*esp_mqtt_client_p* client, *esp_mqtt_evt_t* *evt)

MQTT event callback function.

Parameters

- `[in] client`: MQTT client
- `[in] evt`: MQTT event with type and related data

Enums

enum esp_mqtt_qos_t
Quality of service enumeration.

Values:

ESP_MQTT_QOS_AT_MOST_ONCE = 0x00
Delivery is not guaranteed to arrive, but can arrive `up to 1 time` = non-critical packets where losses are allowed

ESP_MQTT_QOS_AT_LEAST_ONCE = 0x01
Delivery is quaranteed `at least once`, but it may be delivered multiple times with the same content

ESP_MQTT_QOS_EXACTLY_ONCE = 0x02
Delivery is quaranteed `exactly once` = very critical packets such as billing informations or similar

enum esp_mqtt_state_t
State of MQTT client.

Values:

ESP_MQTT_CONN_DISCONNECTED = 0x00
Connection with server is not established

ESP_MQTT_CONN_CONNECTING
Client is connecting to server

ESP_MQTT_CONN_DISCONNECTING
Client connection is disconnecting from server

ESP_MQTT_CONNECTING
MQTT client is connecting... CONNECT command has been sent to server

ESP_MQTT_CONNECTED
MQTT is fully connected and ready to send data on topics

enum esp_mqtt_evt_type_t
MQTT event types.

Values:

ESP_MQTT_EVT_CONNECT
MQTT client connect event

ESP_MQTT_EVT_SUBSCRIBE
MQTT client subscribed to specific topic

ESP_MQTT_EVT_UNSUBSCRIBE
MQTT client unsubscribed from specific topic

ESP_MQTT_EVT_PUBLISH
MQTT client publish message to server event.
Note When publishing packet with quality of service *ESP_MQTT_QOS_AT_MOST_ONCE*, you may not receive event, even if packet was successfully sent, thus do not rely on this event for packet with `qos = ESP_MQTT_QOS_AT_MOST_ONCE`

ESP_MQTT_EVT_PUBLISH_RECV
MQTT client received a publish message from server

ESP_MQTT_EVT_DISCONNECT
MQTT client disconnected from MQTT server

ESP_MQTT_EVT_KEEP_ALIVE
MQTT keep-alive sent to server and reply received

enum esp_mqtt_conn_status_t
List of possible results from MQTT server when executing connect command.

Values:

ESP_MQTT_CONN_STATUS_ACCEPTED = 0x00
Connection accepted and ready to use

ESP_MQTT_CONN_STATUS_REFUSED_PROTOCOL_VERSION = 0x01
Connection Refused, unacceptable protocol version

ESP_MQTT_CONN_STATUS_REFUSED_ID = 0x02
Connection refused, identifier rejected

ESP_MQTT_CONN_STATUS_REFUSED_SERVER = 0x03
Connection refused, server unavailable

ESP_MQTT_CONN_STATUS_REFUSED_USER_PASS = 0x04
Connection refused, bad user name or password

ESP_MQTT_CONN_STATUS_REFUSED_NOT_AUTHORIZED = 0x05
Connection refused, not authorized

ESP_MQTT_CONN_STATUS_TCP_FAILED = 0x100
TCP connection to server was not successful

Functions

esp_mqtt_client_p **esp_mqtt_client_new**(size_t *tx_buff_len*, size_t *rx_buff_len*)
Allocate a new MQTT client structure.

Return Pointer to new allocated MQTT client structure or `NULL` on failure

Parameters

- `[in] tx_buff_len`: Length of raw data output buffer
- `[in] rx_buff_len`: Length of raw data input buffer

void **esp_mqtt_client_delete**(*esp_mqtt_client_p client*)
Delete MQTT client structure.

Note MQTT client must be disconnected first

Parameters

- `[in] client`: MQTT client

espr_t **esp_mqtt_client_connect**(*esp_mqtt_client_p client*, **const** char **host*, *esp_port_t port*, *esp_mqtt_evt_fn evt_fn*, **const** *esp_mqtt_client_info_t* **info*)
Connect to MQTT server.

Note After TCP connection is established, CONNECT packet is automatically sent to server

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] client`: MQTT client

- `[in] host`: Host address for server
- `[in] port`: Host port number
- `[in] evt_fn`: Callback function for all events on this MQTT client
- `[in] info`: Information structure for connection

espr_t **esp_mqtt_client_disconnect**(*esp_mqtt_client_p client*)
Disconnect from MQTT server.

Return *espOK* if request sent to queue or member of *espr_t* otherwise

Parameters

- `[in] client`: MQTT client

uint8_t **esp_mqtt_client_is_connected**(*esp_mqtt_client_p client*)
Test if client is connected to server and accepted to MQTT protocol.

Note Function will return error if TCP is connected but MQTT not accepted

Return `1` on success, `0` otherwise

Parameters

- `[in] client`: MQTT client

espr_t **esp_mqtt_client_subscribe**(*esp_mqtt_client_p client*, **const** char **topic*, esp_mqtt_qos_t *qos*, void **arg*)
Subscribe to MQTT topic.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] client`: MQTT client
- `[in] topic`: Topic name to subscribe to
- `[in] qos`: Quality of service. This parameter can be a value of *esp_mqtt_qos_t*
- `[in] arg`: User custom argument used in callback

espr_t **esp_mqtt_client_unsubscribe**(*esp_mqtt_client_p client*, **const** char **topic*, void **arg*)
Unsubscribe from MQTT topic.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] client`: MQTT client
- `[in] topic`: Topic name to unsubscribe from
- `[in] arg`: User custom argument used in callback

espr_t **esp_mqtt_client_publish**(*esp_mqtt_client_p client*, **const** char **topic*, **const** void **payload*, uint16_t *len*, esp_mqtt_qos_t *qos*, uint8_t *retain*, void **arg*)
Publish a new message on specific topic.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] client`: MQTT client
- `[in] topic`: Topic to send message to
- `[in] payload`: Message data
- `[in] payload_len`: Length of payload data
- `[in] qos`: Quality of service. This parameter can be a value of *esp_mqtt_qos_t* enumeration
- `[in] retain`: Retian parameter value
- `[in] arg`: User custom argument used in callback

void ***esp_mqtt_client_get_arg**(*esp_mqtt_client_p client*)
Get user argument on client.

Return User argument

Parameters

- `[in] client`: MQTT client handle

void **esp_mqtt_client_set_arg**(*esp_mqtt_client_p client*, void **arg*)
Set user argument on client.

Parameters

- `[in] client`: MQTT client handle
- `[in] arg`: User argument

struct esp_mqtt_client_info_t
#include <esp_mqtt_client.h> MQTT client information structure.

Public Members

const char ***id**
Client unique identifier. It is required and must be set by user

const char ***user**
Authentication username. Set to `NULL` if not required

const char ***pass**
Authentication password, set to `NULL` if not required

uint16_t **keep_alive**
Keep-alive parameter in units of seconds. When set to `0`, functionality is disabled (not recommended)

const char ***will_topic**
Will topic

const char ***will_message**
Will message

esp_mqtt_qos_t **will_qos**
Will topic quality of service

struct esp_mqtt_request_t
#include <esp_mqtt_client.h> MQTT request object.

Public Members

uint8_t **status**
Entry status flag for in use or pending bit

uint16_t **packet_id**
Packet ID generated by client on publish

void ***arg**
User defined argument

uint32_t **expected_sent_len**
Number of total bytes which must be sent on connection before we can say "packet was sent".

uint32_t **timeout_start_time**
Timeout start time in units of milliseconds

struct esp_mqtt_evt_t
#include <esp_mqtt_client.h> MQTT event structure for callback function.

Public Members

esp_mqtt_evt_type_t **type**
Event type

esp_mqtt_conn_status_t **status**
Connection status with MQTT

struct *esp_mqtt_evt_t*::**[anonymous]**::**[anonymous]** **connect**
Event for connecting to server

uint8_t **is_accepted**
Status if client was accepted to MQTT prior disconnect event

struct *esp_mqtt_evt_t*::**[anonymous]**::**[anonymous]** **disconnect**
Event for disconnecting from server

void ***arg**
User argument for callback function

espr_t **res**
Response status

struct *esp_mqtt_evt_t*::**[anonymous]**::**[anonymous]** **sub_unsub_scribed**
Event for (un)subscribe to/from topics

struct *esp_mqtt_evt_t*::**[anonymous]**::**[anonymous]** **publish**
Published event

const uint8_t ***topic**
Pointer to topic identifier

size_t **topic_len**
Length of topic

const void ***payload**
Topic payload

size_t **payload_len**
Length of topic payload

uint8_t **dup**
Duplicate flag if message was sent again

esp_mqtt_qos_t **qos**
Received packet quality of service

struct *esp_mqtt_evt_t*::**[anonymous]**::**[anonymous]** **publish_recv**
Publish received event

union *esp_mqtt_evt_t*::**[anonymous]** **evt**
Event data parameters

group **ESP_APP_MQTT_CLIENT_EVT**
Event helper functions.

Connect event

Note Use these functions on *ESP_MQTT_EVT_CONNECT* event

esp_mqtt_client_evt_connect_get_status(client, evt)
Get connection status.

Return Connection status. Member of *esp_mqtt_conn_status_t*

Parameters

- `[in] client`: MQTT client
- `[in] evt`: Event handle

Disconnect event

Note Use these functions on *ESP_MQTT_EVT_DISCONNECT* event

esp_mqtt_client_evt_disconnect_is_accepted(client, evt)
Check if MQTT client was accepted by server when disconnect event occurred.

Return `1` on success, `0` otherwise

Parameters

- `[in] client`: MQTT client
- `[in] evt`: Event handle

Subscribe/unsubscribe event

Note Use these functions on *ESP_MQTT_EVT_SUBSCRIBE* or *ESP_MQTT_EVT_UNSUBSCRIBE* events

esp_mqtt_client_evt_subscribe_get_argument(client, evt)
Get user argument used on *esp_mqtt_client_subscribe*.

Return User argument

Parameters

- `[in] client`: MQTT client

- `[in] evt`: Event handle

esp_mqtt_client_evt_subscribe_get_result (client, evt)
Get result of subscribe event.

Return *espOK* on success, member of *espr_t* otherwise

Parameters

- `[in] client`: MQTT client
- `[in] evt`: Event handle

esp_mqtt_client_evt_unsubscribe_get_argument (client, evt)
Get user argument used on *esp_mqtt_client_unsubscribe*.

Return User argument

Parameters

- `[in] client`: MQTT client
- `[in] evt`: Event handle

esp_mqtt_client_evt_unsubscribe_get_result (client, evt)
Get result of unsubscribe event.

Return *espOK* on success, member of *espr_t* otherwise

Parameters

- `[in] client`: MQTT client
- `[in] evt`: Event handle

Publish receive event

Note Use these functions on *ESP_MQTT_EVT_PUBLISH_RECV* event

esp_mqtt_client_evt_publish_recv_get_topic (client, evt)
Get topic from received publish packet.

Return Topic name

Parameters

- `[in] client`: MQTT client
- `[in] evt`: Event handle

esp_mqtt_client_evt_publish_recv_get_topic_len (client, evt)
Get topic length from received publish packet.

Return Topic length

Parameters

- `[in] client`: MQTT client
- `[in] evt`: Event handle

esp_mqtt_client_evt_publish_recv_get_payload (client, evt)
Get payload from received publish packet.

Return Packet payload

Parameters

- `[in] client`: MQTT client
- `[in] evt`: Event handle

esp_mqtt_client_evt_publish_recv_get_payload_len (client, evt)
Get payload length from received publish packet.

Return Payload length

Parameters

- `[in] client`: MQTT client
- `[in] evt`: Event handle

esp_mqtt_client_evt_publish_recv_is_duplicate (client, evt)
Check if packet is duplicated.

Return `1` if duplicated, `0` otherwise

Parameters

- `[in] client`: MQTT client
- `[in] evt`: Event handle

esp_mqtt_client_evt_publish_recv_get_qos (client, evt)
Get received quality of service.

Return Member of *esp_mqtt_qos_t* enumeration

Parameters

- `[in] client`: MQTT client
- `[in] evt`: Event handle

Publish event

Note Use these functions on *ESP_MQTT_EVT_PUBLISH* event

esp_mqtt_client_evt_publish_get_argument (client, evt)
Get user argument used on *esp_mqtt_client_publish*.

Return User argument

Parameters

- `[in] client`: MQTT client
- `[in] evt`: Event handle

esp_mqtt_client_evt_publish_get_result (client, evt)
Get result of publish event.

Return *espOK* on success, member of *espr_t* otherwise

Parameters

- `[in] client`: MQTT client
- `[in] evt`: Event handle

Defines

esp_mqtt_client_evt_get_type (client, evt)
Get MQTT event type.

Return MQTT Event type, value of *esp_mqtt_evt_type_t* enumeration

Parameters

- `[in] client`: MQTT client
- `[in] evt`: Event handle

MQTT Client API

MQTT Client API provides sequential API built on top of *MQTT Client*.

Listing 24: MQTT API application example code

```
/*
 * MQTT client API example with ESP device.
 *
 * Once device is connected to network,
 * it will try to connect to mosquitto test server and start the MQTT.
 *
 * If successfully connected, it will publish data to "esp_mqtt_topic" topic every x
↪seconds.
 *
 * To check if data are sent, you can use mqtt-spy PC software to inspect
 * test.mosquitto.org server and subscribe to publishing topic
 */

#include "esp/apps/esp_mqtt_client_api.h"
#include "mqtt_client_api.h"
#include "esp/esp_mem.h"

/**
 * \brief           Connection information for MQTT CONNECT packet
 */
static const esp_mqtt_client_info_t
mqtt_client_info = {
    .keep_alive = 10,

    /* Server login data */
    .user = "8a215f70-a644-11e8-ac49-e932ed599553",
    .pass = "26aa943f702e5e780f015cd048a91e8fb54cca28",

    /* Device identifier address */
    .id = "869f5a20-af9c-11e9-b01f-db5cf74e7fb7",
};
```

(continues on next page)

(continued from previous page)

```

/**
 * \brief           Memory for temporary topic
 */
static char
mqtt_topic_str[256];

/**
 * \brief           Generate random number and write it to string
 * \param[out]      str: Output string with new number
 */
void
generate_random(char* str) {
    static uint32_t random_beg = 0x8916;
    random_beg = random_beg * 0x00123455 + 0x85654321;
    sprintf(str, "%u", (unsigned)((random_beg >> 8) & 0xFFFF));
}

/**
 * \brief           MQTT client API thread
 */
void
mqtt_client_api_thread(void const* arg) {
    esp_mqtt_client_api_p client;
    esp_mqtt_conn_status_t conn_status;
    esp_mqtt_client_api_buf_p buf;
    espr_t res;
    char random_str[10];

    /* Create new MQTT API */
    client = esp_mqtt_client_api_new(256, 128);
    if (client == NULL) {
        goto terminate;
    }

    while (1) {
        /* Make a connection */
        printf("Joining MQTT server\r\n");

        /* Try to join */
        conn_status = esp_mqtt_client_api_connect(client, "mqtt.mydevices.com", 1883,␣
↪&mqtt_client_info);
        if (conn_status == ESP_MQTT_CONN_STATUS_ACCEPTED) {
            printf("Connected and accepted!\r\n");
            printf("Client is ready to subscribe and publish to new messages\r\n");
        } else {
            printf("Connect API response: %d\r\n", (int)conn_status);
            esp_delay(5000);
            continue;
        }

        /* Subscribe to topics */
        sprintf(mqtt_topic_str, "v1/%s/things/%s/cmd/#", mqtt_client_info.user, mqtt_
↪client_info.id);
        if (esp_mqtt_client_api_subscribe(client, mqtt_topic_str, ESP_MQTT_QOS_AT_
↪LEAST_ONCE) == espOK) {
            printf("Subscribed to topic\r\n");
```

(continues on next page)

(continued from previous page)

```
        } else {
            printf("Problem subscribing to topic!\r\n");
        }

        while (1) {
            /* Receive MQTT packet with 1000ms timeout */
            res = esp_mqtt_client_api_receive(client, &buf, 5000);
            if (res == espOK) {
                if (buf != NULL) {
                    printf("Publish received!\r\n");
                    printf("Topic: %s, payload: %s\r\n", buf->topic, buf->payload);
                    esp_mqtt_client_api_buf_free(buf);
                    buf = NULL;
                }
            } else if (res == espCLOSED) {
                printf("MQTT connection closed!\r\n");
                break;
            } else if (res == espTIMEOUT) {
                printf("Timeout on MQTT receive function. Manually publishing.\r\n");

                /* Publish data on channel 1 */
                generate_random(random_str);
                sprintf(mqtt_topic_str, "v1/%s/things/%s/data/1", mqtt_client_info.
↪user, mqtt_client_info.id);
                esp_mqtt_client_api_publish(client, mqtt_topic_str, random_str,␣
↪strlen(random_str), ESP_MQTT_QOS_AT_LEAST_ONCE, 0);
            }
        }
        //goto terminate;
    }

terminate:
    esp_mqtt_client_api_delete(client);
    printf("MQTT client thread terminate\r\n");
    esp_sys_thread_terminate(NULL);
}
```

group **ESP_APP_MQTT_CLIENT_API**

Sequential, single thread MQTT client API.

Typedefs

typedef struct esp_mqtt_client_api_buf ***esp_mqtt_client_api_buf_p**

Pointer to *esp_mqtt_client_api_buf_t* structure.

Functions

esp_mqtt_client_api_p **esp_mqtt_client_api_new**(size_t *tx_buff_len*, size_t *rx_buff_len*)
Create new MQTT client API.

Return Client handle on success, `NULL` otherwise

Parameters

- `[in] tx_buff_len`: Maximal TX buffer for maximal packet length
- `[in] rx_buff_len`: Maximal RX buffer

void **esp_mqtt_client_api_delete**(esp_mqtt_client_api_p *client*)
Delete client from memory.

Parameters

- `[in] client`: MQTT API client handle

esp_mqtt_conn_status_t **esp_mqtt_client_api_connect**(esp_mqtt_client_api_p *client*, **const** char **host*, *esp_port_t port*, **const** *esp_mqtt_client_info_t* **info*)
Connect to MQTT broker.

Return *ESP_MQTT_CONN_STATUS_ACCEPTED* on success, member of *esp_mqtt_conn_status_t* otherwise

Parameters

- `[in] client`: MQTT API client handle
- `[in] host`: TCP host
- `[in] port`: TCP port
- `[in] info`: MQTT client info

espr_t **esp_mqtt_client_api_close**(esp_mqtt_client_api_p *client*)
Close MQTT connection.

Return *espOK* on success, member of *espr_t* otherwise

Parameters

- `[in] client`: MQTT API client handle

espr_t **esp_mqtt_client_api_subscribe**(esp_mqtt_client_api_p *client*, **const** char **topic*, esp_mqtt_qos_t *qos*)
Subscribe to topic.

Return *espOK* on success, member of *espr_t* otherwise

Parameters

- `[in] client`: MQTT API client handle
- `[in] topic`: Topic to subscribe on
- `[in] qos`: Quality of service. This parameter can be a value of *esp_mqtt_qos_t*

espr_t **esp_mqtt_client_api_unsubscribe**(esp_mqtt_client_api_p *client*, **const** char **topic*)
Unsubscribe from topic.

Return *espOK* on success, member of *espr_t* otherwise

Parameters

- `[in] client`: MQTT API client handle
- `[in] topic`: Topic to unsubscribe from

espr_t **esp_mqtt_client_api_publish**(esp_mqtt_client_api_p *client*, **const** char **topic*, **const** void **data*, size_t *btw*, esp_mqtt_qos_t *qos*, uint8_t *retain*)
Publish new packet to MQTT network.

Return *espOK* on success, member of *espr_t* otherwise

Parameters

- `[in] client`: MQTT API client handle
- `[in] topic`: Topic to publish on
- `[in] data`: Data to send
- `[in] btw`: Number of bytes to send for data parameter
- `[in] qos`: Quality of service. This parameter can be a value of *esp_mqtt_qos_t*
- `[in] retain`: Set to `1` for retain flag, `0` otherwise

uint8_t **esp_mqtt_client_api_is_connected**(esp_mqtt_client_api_p *client*)
Check if client MQTT connection is active.

Return `1` on success, `0` otherwise

Parameters

- `[in] client`: MQTT API client handle

espr_t **esp_mqtt_client_api_receive**(esp_mqtt_client_api_p *client*, *esp_mqtt_client_api_buf_p* **p*, uint32_t *timeout*)
Receive next packet in specific timeout time.

Note This function can be called from separate thread than the rest of API function, which allows you to handle receive data separated with custom timeout

Return *espOK* on success, *espCLOSED* if MQTT is closed, *espTIMEOUT* on timeout

Parameters

- `[in] client`: MQTT API client handle
- `[in] p`: Pointer to output buffer
- `[in] timeout`: Maximal time to wait before function returns timeout

void **esp_mqtt_client_api_buf_free**(*esp_mqtt_client_api_buf_p p*)
Free buffer memory after usage.

Parameters

- `[in] p`: Buffer to free

struct esp_mqtt_client_api_buf_t
#include <esp_mqtt_client_api.h> MQTT API RX buffer.

Public Members

char ***topic**
Topic data

size_t **topic_len**
Topic length

uint8_t ***payload**
Payload data

size_t **payload_len**
Payload length

esp_mqtt_qos_t **qos**
Quality of service

Netconn API

Netconn API is addon on top of existing connection module and allows sending and receiving data with sequential API calls, similar to *POSIX socket* API.

It can operate in client or server mode and uses operating system features, such as message queues and semaphore to link non-blocking callback API for connections with sequential API for application thread.

Note: Connection API does not directly allow receiving data with sequential and linear code execution. All is based on connection event system. Netconn adds this functionality as it is implemented on top of regular connection API.

Warning: Netconn API are designed to be called from application threads ONLY. It is not allowed to call any of *netconn API* functions from within interrupt or callback event functions.

Netconn client

Fig. 9: Netconn API client block diagram

Above block diagram shows basic architecture of netconn client application. There is always one application thread (in green) which calls *netconn API* functions to interact with connection API in synchronous mode.

Every netconn connection uses dedicated structure to handle message queue for data received packet buffers. Each time new packet is received (red block, *data received event*), reference to it is written to message queue of netconn structure, while application thread reads new entries from the same queue to get packets.

Listing 25: Netconn client example

```c
#include "netconn_client.h"
#include "esp/esp.h"

/**
 * \brief           Host and port settings
 */
#define NETCONN_HOST        "example.com"
#define NETCONN_PORT        80

/**
 * \brief           Request header to send on successful connection
 */
static const char
request_header[] = ""
"GET / HTTP/1.1\r\n"
"Host: " NETCONN_HOST "\r\n"
"Connection: close\r\n"
"\r\n";

/**
 * \brief           Netconn client thread implementation
 * \param[in]       arg: User argument
 */
void
netconn_client_thread(void const* arg) {
    espr_t res;
    esp_pbuf_p pbuf;
    esp_netconn_p client;
    esp_sys_sem_t* sem = (void *)arg;

    /*
     * First create a new instance of netconn
     * connection and initialize system message boxes
     * to accept received packet buffers
     */
    client = esp_netconn_new(ESP_NETCONN_TYPE_TCP);
    if (client != NULL) {
        /*
         * Connect to external server as client
         * with custom NETCONN_CONN_HOST and CONN_PORT values
         *
         * Function will block thread until we are successfully connected (or not) to server
         */
        res = esp_netconn_connect(client, NETCONN_HOST, NETCONN_PORT);
        if (res == espOK) {                     /* Are we successfully connected? */
            printf("Connected to " NETCONN_HOST "\r\n");
            res = esp_netconn_write(client, request_header, sizeof(request_header) - 1);    /* Send data to server */
            if (res == espOK) {
                res = esp_netconn_flush(client);    /* Flush data to output */
            }
            if (res == espOK) {                 /* Were data sent? */
                printf("Data were successfully sent to server\r\n");

```

(continues on next page)

(continued from previous page)

```
                /*
                 * Since we sent HTTP request,
                 * we are expecting some data from server
                 * or at least forced connection close from remote side
                 */
                do {
                    /*
                     * Receive single packet of data
                     *
                     * Function will block thread until new packet
                     * is ready to be read from remote side
                     *
                     * After function returns, don't forgot the check value.
                     * Returned status will give you info in case connection
                     * was closed too early from remote side
                     */
                    res = esp_netconn_receive(client, &pbuf);
                    if (res == espCLOSED) {     /* Was the connection closed? This can be checked by return status of receive function */
                        printf("Connection closed by remote side...\r\n");
                        break;
                    } else if (res == espTIMEOUT) {
                        printf("Netconn timeout while receiving data. You may try multiple readings before deciding to close manually\r\n");
                    }

                    if (res == espOK && pbuf != NULL) { /* Make sure we have valid packet buffer */
                        /*
                         * At this point read and manipulate
                         * with received buffer and check if you expect more data
                         *
                         * After you are done using it, it is important
                         * you free the memory otherwise memory leaks will appear
                         */
                        printf("Received new data packet of %d bytes\r\n", (int)esp_pbuf_length(pbuf, 1));
                        esp_pbuf_free(pbuf);    /* Free the memory after usage */
                        pbuf = NULL;
                    }
                } while (1);
            } else {
                printf("Error writing data to remote host!\r\n");
            }

            /*
             * Check if connection was closed by remote server
             * and in case it wasn't, close it manually
             */
            if (res != espCLOSED) {
                esp_netconn_close(client);
            }
        } else {
            printf("Cannot connect to remote host %s:%d!\r\n", NETCONN_HOST, NETCONN_PORT);
        }
        esp_netconn_delete(client);             /* Delete netconn structure */
```

(continues on next page)

(continued from previous page)

```
    }

    if (esp_sys_sem_isvalid(sem)) {
        esp_sys_sem_release(sem);
    }
    esp_sys_thread_terminate(NULL);             /* Terminate current thread */
}
```

Netconn server

Fig. 10: Netconn API server block diagram

When netconn is configured in server mode, it is possible to accept new clients from remote side. Application creates *netconn server connection*, which can only accept *clients* and cannot send/receive any data. It configures server on dedicated port (selected by application) and listens on it.

When new client connects, *server callback function* is called with *new active connection event*. Newly accepted connection is then written to server structure netconn which is later read by application thread. At the same time, *netconn connection* structure (blue) is created to allow standard send/receive operation on active connection.

Note: Each connected client has its own *netconn connection* structure. When multiple clients connect to server at the same time, multiple entries are written to *connection accept* message queue and are ready to be processed by application thread.

From this point, program flow is the same as in case of *netconn client*.

This is basic example for netconn thread. It waits for client and processes it in blocking mode.

Warning: When multiple clients connect at the same time to netconn server, they are processed one-by-one, sequentially. This may introduce delay in response for other clients. Check netconn concurrency option to process multiple clients at the same time

Listing 26: Netconn server with single processing thread

```
/*
 * Netconn server example is based on single thread
 * and it listens for single client only on port 23
 */
#include "netconn_server_1thread.h"
#include "esp/esp.h"

/**
 * \brief           Basic thread for netconn server to test connections
 * \param[in]       arg: User argument
 */
void
netconn_server_1thread_thread(void* arg) {
    espr_t res;
    esp_netconn_p server, client;
```

(continues on next page)

(continued from previous page)

```
    esp_pbuf_p p;

    /* Create netconn for server */
    server = esp_netconn_new(ESP_NETCONN_TYPE_TCP);
    if (server == NULL) {
        printf("Cannot create server netconn!\r\n");
    }

    /* Bind it to port 23 */
    res = esp_netconn_bind(server, 23);
    if (res != espOK) {
        printf("Cannot bind server\r\n");
        goto out;
    }

    /* Start listening for incoming connections with maximal 1 client */
    res = esp_netconn_listen_with_max_conn(server, 1);
    if (res != espOK) {
        goto out;
    }

    /* Unlimited loop */
    while (1) {
        /* Accept new client */
        res = esp_netconn_accept(server, &client);
        if (res != espOK) {
            break;
        }
        printf("New client accepted!\r\n");
        while (1) {
            /* Receive data */
            res = esp_netconn_receive(client, &p);
            if (res == espOK) {
                printf("Data received!\r\n");
                esp_pbuf_free(p);
            } else {
                printf("Netconn receive returned: %d\r\n", (int)res);
                if (res == espCLOSED) {
                    printf("Connection closed by client\r\n");
                    break;
                }
            }
        }
        /* Delete client */
        if (client != NULL) {
            esp_netconn_delete(client);
            client = NULL;
        }
    }
    /* Delete client */
    if (client != NULL) {
        esp_netconn_delete(client);
        client = NULL;
    }

out:
    printf("Terminating netconn thread!\r\n");
```

(continues on next page)

(continued from previous page)

```
    if (server != NULL) {
        esp_netconn_delete(server);
    }
    esp_sys_thread_terminate(NULL);
}
```

Netconn server concurrency

Fig. 11: Netconn API server concurrency block diagram

When compared to classic netconn server, concurrent netconn server mode allows multiple clients to be processed at the same time. This can drastically improve performance and response time on clients side, especially when many clients are connected to server at the same time.

Every time *server application thread* (green block) gets new client to process, it starts a new *processing* thread instead of doing it in accept thread.

- Server thread is only dedicated to accept clients and start threads
- Multiple processing thread can run in parallel to send/receive data from multiple clients
- No delay when multi clients are active at the same time
- Higher memory footprint is necessary as there are multiple threads active

Listing 27: Netconn server with multiple processing threads

```
/*
 * Netconn server example is based on single "user" thread
 * which listens for new connections and accepts them.
 *
 * When a new client is accepted by server,
 * separate thread for client is created where
 * data is read, processed and send back to user
 */
#include "netconn_server.h"
#include "esp/esp.h"

static void netconn_server_processing_thread(void* const arg);

/**
 * \brief           Main page response file
 */
static const uint8_t
resp_data_mainpage_top[] = ""
"HTTP/1.1 200 OK\r\n"
"Content-Type: text/html\r\n"
"\r\n"
"<html>"
"    <head>"
"        <link rel=\"stylesheet\" href=\"style.css\" type=\"text/css\" />"
"        <meta http-equiv=\"refresh\" content=\"1\" />"
"    </head>"
"    <body>"
"        <p>Netconn driven website!</p>"
```

(continues on next page)

(continued from previous page)

```
"       <p>Total system up time: <b>";

/**
 * \brief           Bottom part of main page
 */
static const uint8_t
resp_data_mainpage_bottom[] = ""
"       </b></p>"
"   </body>"
"</html>";

/**
 * \brief           Style file response
 */
static const uint8_t
resp_data_style[] = ""
"HTTP/1.1 200 OK\r\n"
"Content-Type: text/css\r\n"
"\r\n"
"body { color: red; font-family: Tahoma, Arial; };";

/**
 * \brief           404 error response
 */
static const uint8_t
resp_error_404[] = ""
"HTTP/1.1 404 Not Found\r\n"
"\r\n"
"Error 404";

/**
 * \brief           Netconn server thread implementation
 * \param[in]       arg: User argument
 */
void
netconn_server_thread(void const* arg) {
    espr_t res;
    esp_netconn_p server, client;

    /*
     * First create a new instance of netconn
     * connection and initialize system message boxes
     * to accept clients and packet buffers
     */
    server = esp_netconn_new(ESP_NETCONN_TYPE_TCP);
    if (server != NULL) {
        printf("Server netconn created\r\n");

        /* Bind network connection to port 80 */
        res = esp_netconn_bind(server, 80);
        if (res == espOK) {
            printf("Server netconn listens on port 80\r\n");
            /*
             * Start listening for incoming connections
             * on previously binded port
             */
            res = esp_netconn_listen(server);
```

(continues on next page)

(continued from previous page)

```

            while (1) {
                /*
                 * Wait and accept new client connection
                 *
                 * Function will block thread until
                 * new client is connected to server
                 */
                res = esp_netconn_accept(server, &client);
                if (res == espOK) {
                    printf("Netconn new client connected. Starting new thread...\r\n
↪");
                    /*
                     * Start new thread for this request.
                     *
                     * Read and write back data to user in separated thread
                     * to allow processing of multiple requests at the same time
                     */
                    if (esp_sys_thread_create(NULL, "client", (esp_sys_thread_
↪fn)netconn_server_processing_thread, client, 512, ESP_SYS_THREAD_PRIO)) {
                        printf("Netconn client thread created\r\n");
                    } else {
                        printf("Netconn client thread creation failed!\r\n");

                        /* Force close & delete */
                        esp_netconn_close(client);
                        esp_netconn_delete(client);
                    }
                } else {
                    printf("Netconn connection accept error!\r\n");
                    break;
                }
            }
        } else {
            printf("Netconn server cannot bind to port\r\n");
        }
    } else {
        printf("Cannot create server netconn\r\n");
    }

    esp_netconn_delete(server);                 /* Delete netconn structure */
    esp_sys_thread_terminate(NULL);             /* Terminate current thread */
}

/**
 * \brief           Thread to process single active connection
 * \param[in]       arg: Thread argument
 */
static void
netconn_server_processing_thread(void* const arg) {
    esp_netconn_p client;
    esp_pbuf_p pbuf, p = NULL;
    espr_t res;
    char strt[20];

    client = arg;                               /* Client handle is passed to␣
↪argument */
```

(continues on next page)

(continued from previous page)

```
    printf("A new connection accepted!\r\n");    /* Print simple message */

    do {
        /*
         * Client was accepted, we are now
         * expecting client will send to us some data
         *
         * Wait for data and block thread for that time
         */
        res = esp_netconn_receive(client, &pbuf);

        if (res == espOK) {
            printf("Netconn data received, %d bytes\r\n", (int)esp_pbuf_length(pbuf,␣
↪1));
            /* Check reception of all header bytes */
            if (p == NULL) {
                p = pbuf;                        /* Set as first buffer */
            } else {
                esp_pbuf_cat(p, pbuf);           /* Concatenate buffers together */
            }
            if (esp_pbuf_strfind(pbuf, "\r\n\r\n", 0) != ESP_SIZET_MAX) {
                if (esp_pbuf_strfind(pbuf, "GET / ", 0) != ESP_SIZET_MAX) {
                    uint32_t now;
                    printf("Main page request\r\n");
                    now = esp_sys_now();         /* Get current time */
                    sprintf(strt, "%u ms; %d s", (unsigned)now, (unsigned)(now /␣
↪1000));
                    esp_netconn_write(client, resp_data_mainpage_top, sizeof(resp_
↪data_mainpage_top) - 1);
                    esp_netconn_write(client, strt, strlen(strt));
                    esp_netconn_write(client, resp_data_mainpage_bottom, sizeof(resp_
↪data_mainpage_bottom) - 1);
                } else if (esp_pbuf_strfind(pbuf, "GET /style.css ", 0) != ESP_SIZET_
↪MAX) {
                    printf("Style page request\r\n");
                    esp_netconn_write(client, resp_data_style, sizeof(resp_data_
↪style) - 1);
                } else {
                    printf("404 error not found\r\n");
                    esp_netconn_write(client, resp_error_404, sizeof(resp_error_404) -
↪ 1);
                }
                esp_netconn_close(client);       /* Close netconn connection */
                esp_pbuf_free(p);                /* Do not forget to free memory after␣
↪usage! */
                p = NULL;
                break;
            }
        }
    } while (res == espOK);

    if (p != NULL) {                             /* Free received data */
        esp_pbuf_free(p);
        p = NULL;
    }
    esp_netconn_delete(client);                  /* Destroy client memory */
```

(continues on next page)

(continued from previous page)

```
189     esp_sys_thread_terminate(NULL);             /* Terminate this thread */
190 }
```

Non-blocking receive

By default, netconn API is written to only work in separate application thread, dedicated for network connection processing. Because of that, by default every function is fully blocking. It will wait until result is ready to be used by application.

It is, however, possible to enable timeout feature for receiving data only. When this feature is enabled, *esp_netconn_receive()* will block for maximal timeout set with *esp_netconn_set_receive_timeout()* function.

When enabled, if there is no received data for timeout amount of time, function will return with timeout status and application needs to process it accordingly.

Tip: *ESP_CFG_NETCONN_RECEIVE_TIMEOUT* must be set to `1` to use this feature.

---

group **ESP_NETCONN**

Network connection.

Defines

ESP_NETCONN_RECEIVE_NO_WAIT

Receive data with no timeout.

Note Used with *esp_netconn_set_receive_timeout* function

Typedefs

typedef struct esp_netconn ***esp_netconn_p**

Netconn object structure.

Enums

enum esp_netconn_type_t

Netconn connection type.

Values:

ESP_NETCONN_TYPE_TCP = ESP_CONN_TYPE_TCP

TCP connection

ESP_NETCONN_TYPE_SSL = ESP_CONN_TYPE_SSL

SSL connection

ESP_NETCONN_TYPE_UDP = ESP_CONN_TYPE_UDP

UDP connection

Functions

esp_netconn_p **esp_netconn_new** (esp_netconn_type_t *type*)
Create new netconn connection.

Return New netconn connection on success, `NULL` otherwise

Parameters

- `[in] type`: Netconn connection type

espr_t **esp_netconn_delete** (*esp_netconn_p nc*)
Delete netconn connection.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] nc`: Netconn handle

espr_t **esp_netconn_bind** (*esp_netconn_p nc*, *esp_port_t port*)
Bind a connection to specific port, can be only used for server connections.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] nc`: Netconn handle
- `[in] port`: Port used to bind a connection to

espr_t **esp_netconn_connect** (*esp_netconn_p nc*, **const** char **host*, *esp_port_t port*)
Connect to server as client.

Return *espOK* if successfully connected, member of *espr_t* otherwise

Parameters

- `[in] nc`: Netconn handle
- `[in] host`: Pointer to host, such as domain name or IP address in string format
- `[in] port`: Target port to use

espr_t **esp_netconn_receive** (*esp_netconn_p nc*, *esp_pbuf_p *pbuf*)
Receive data from connection.

Return *espOK* when new data ready

Return *espCLOSED* when connection closed by remote side

Return *espTIMEOUT* when receive timeout occurs

Return Any other member of *espr_t* otherwise

Parameters

- `[in] nc`: Netconn handle used to receive from
- `[in] pbuf`: Pointer to pointer to save new receive buffer to. When function returns, user must check for valid pbuf value `pbuf != NULL`

espr_t **esp_netconn_close**(*esp_netconn_p nc*)
Close a netconn connection.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] nc`: Netconn handle to close

int8_t **esp_netconn_get_connnum**(*esp_netconn_p nc*)
Get connection number used for netconn.

Return `-1` on failure, connection number between `0` and *ESP_CFG_MAX_CONNS* otherwise

Parameters

- `[in] nc`: Netconn handle

esp_conn_p **esp_netconn_get_conn**(*esp_netconn_p nc*)
Get netconn connection handle.

Return ESP connection handle

Parameters

- `[in] nc`: Netconn handle

void **esp_netconn_set_receive_timeout**(*esp_netconn_p nc*, uint32_t *timeout*)
Set timeout value for receiving data.

When enabled, *esp_netconn_receive* will only block for up to *timeout* value and will return if no new data within this time

Parameters

- `[in] nc`: Netconn handle
- `[in] timeout`: Timeout in units of milliseconds. Set to `0` to disable timeout feature Set to `> 0` to set maximum milliseconds to wait before timeout Set to *ESP_NETCONN_RECEIVE_NO_WAIT* to enable non-blocking receive

uint32_t **esp_netconn_get_receive_timeout**(*esp_netconn_p nc*)
Get netconn receive timeout value.

Return Timeout in units of milliseconds. If value is `0`, timeout is disabled (wait forever)

Parameters

- `[in] nc`: Netconn handle

espr_t **esp_netconn_connect_ex**(*esp_netconn_p nc*, **const** char **host*, *esp_port_t port*, uint16_t *keep_alive*, **const** char **local_ip*, *esp_port_t local_port*, uint8_t *mode*)
Connect to server as client, allow keep-alive option.

Return *espOK* if successfully connected, member of *espr_t* otherwise

Parameters

- `[in] nc`: Netconn handle

- `[in] host`: Pointer to host, such as domain name or IP address in string format
- `[in] port`: Target port to use
- `[in] keep_alive`: Keep alive period seconds
- `[in] local_ip`: Local ip in connected command
- `[in] local_port`: Local port address
- `[in] mode`: UDP mode

espr_t **esp_netconn_listen**(*esp_netconn_p nc*)
Listen on previously binded connection.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] nc`: Netconn handle used to listen for new connections

espr_t **esp_netconn_listen_with_max_conn**(*esp_netconn_p nc*, uint16_t *max_connections*)
Listen on previously binded connection with max allowed connections at a time.

Return *espOK* on success, member of *espr_t* otherwise

Parameters

- `[in] nc`: Netconn handle used to listen for new connections
- `[in] max_connections`: Maximal number of connections server can accept at a time This parameter may not be larger than *ESP_CFG_MAX_CONNS*

espr_t **esp_netconn_set_listen_conn_timeout**(*esp_netconn_p nc*, uint16_t *timeout*)
Set timeout value in units of seconds when connection is in listening mode If new connection is accepted, it will be automatically closed after `seconds` elapsed without any data exchange.

Note Call this function before you put connection to listen mode with *esp_netconn_listen*

Return *espOK* on success, member of *espr_t* otherwise

Parameters

- `[in] nc`: Netconn handle used for listen mode
- `[in] timeout`: Time in units of seconds. Set to `0` to disable timeout feature

espr_t **esp_netconn_accept**(*esp_netconn_p nc*, *esp_netconn_p* **client*)
Accept a new connection.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] nc`: Netconn handle used as base connection to accept new clients
- `[out] client`: Pointer to netconn handle to save new connection to

espr_t **esp_netconn_write**(*esp_netconn_p nc*, **const** void **data*, size_t *btw*)
Write data to connection output buffers.

Note This function may only be used on TCP or SSL connections

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] nc`: Netconn handle used to write data to
- `[in] data`: Pointer to data to write
- `[in] btw`: Number of bytes to write

espr_t **esp_netconn_flush**(*esp_netconn_p nc*)
Flush buffered data on netconn *TCP/SSL* connection.

Note This function may only be used on *TCP/SSL* connection

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] nc`: Netconn handle to flush data

espr_t **esp_netconn_send**(*esp_netconn_p nc*, **const** void **data*, size_t *btw*)
Send data on *UDP* connection to default IP and port.

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] nc`: Netconn handle used to send
- `[in] data`: Pointer to data to write
- `[in] btw`: Number of bytes to write

espr_t **esp_netconn_sendto**(*esp_netconn_p nc*, **const** *esp_ip_t *ip*, *esp_port_t port*, **const** void **data*, size_t *btw*)
Send data on *UDP* connection to specific IP and port.

Note Use this function in case of UDP type netconn

Return *espOK* on success, member of *espr_t* enumeration otherwise

Parameters

- `[in] nc`: Netconn handle used to send
- `[in] ip`: Pointer to IP address
- `[in] port`: Port number used to send data
- `[in] data`: Pointer to data to write
- `[in] btw`: Number of bytes to write

5.3.5 Command line interface

CLI Configuration

group **CLI_CONFIG**
Default CLI configuration.

Configuration for command line interface (CLI).

Defines

CLI_PROMPT
CLI promet, printed on every NL.

CLI_NL
CLI NL, default is NL and CR.

CLI_MAX_CMD_LENGTH
Max CLI command length.

CLI_CMD_HISTORY
Max sotred CLI commands to history.

CLI_MAX_NUM_OF_ARGS
Max CLI arguments in a single command.

CLI_MAX_MODULES
Max modules for CLI.

CLI Input module

group **CLI_INPUT**
Command line interface helper funcions for paring input data.

Functions to parse incoming data for command line interface (CLI).

Functions

void **cli_in_data**(*cli_printf cliprintf*, char *ch*)
parse new characters to the CLI

Parameters

- `[in] cliprintf`: Pointer to CLI printf function
- `[in] ch`: new character to CLI

group **CLI**
Command line interface.

Functions to initialize everything needed for command line interface (CLI).

Typedefs

typedef void **cli_printf**(**const** char **format*, ...)
Printf handle for CLI.

Parameters

- `[in] format`: string format

typedef void **cli_function**(*cli_printf cliprintf*, int *argc*, char ***argv*)
CLI entriy function.

Parameters

- `[in] cliprintf`: Printf handle callback
- `[in] argc`: Number of arguments
- `[in] argv`: Pointer to pointer to arguments

Functions

const *cli_command_t* ***cli_lookup_command**(char **command*)
Find the CLI command that matches the input string.

Return pointer of the command if we found a match, else NULL

Parameters

- `[in] command`: pointer to command string for which we are searching

void **cli_tab_auto_complete**(*cli_printf cliprintf*, char **cmd_buffer*, uint32_t **cmd_pos*, bool *print_options*)
CLI auto completion function.

Parameters

- `[in] cliprintf`: Pointer to CLI printf function
- `[in] cmd_buffer`: CLI command buffer
- `[in] cmd_pos`: pointer to current courser postion in command buffer
- `[in] print_options`: additional prints in case of double tab

bool **cli_register_commands**(**const** *cli_command_t* **commands*, size_t *num_of_commands*)
Register new CLI commands.

Return true when new commands where succesfully added, else false

Parameters

- `[in] commands`: Pointer to commands table
- `[in] num_of_commands`: Number of new commands

void **cli_init**(void)
CLI Init function for adding basic CLI commands.

struct cli_command_t
#include <*cli.h*> CLI command structure.

Public Members

const char ***name**
Command name

const char ***help**
Command help

cli_function ***func**
Command function

struct cli_commands_t
#include <*cli.h*> List of commands.

Public Members

const *cli_command_t* ***commands**
Pointer to commands

size_t **num_of_commands**
Total number of commands

5.4 Examples and demos

Various examples are provided for fast library evaluation on embedded systems. These are optimized prepared and maintained for 2 platforms, but could be easily extended to more platforms:

- WIN32 examples, prepared as Visual Studio Community projects
- ARM Cortex-M examples for STM32, prepared as STM32CubeIDE GCC projects

Warning: Library is platform independent and can be used on any platform.

5.4.1 Example architectures

There are many platforms available today on a market, however supporting them all would be tough task for single person. Therefore it has been decided to support (for purpose of examples) 2 platforms only, *WIN32* and *STM32*.

WIN32

Examples for *WIN32* are prepared as Visual Studio Community projects. You can directly open project in the IDE, compile & debug.

Application opens *COM* port, set in the low-level driver. External USB to UART converter (FTDI-like device) is necessary in order to connect to *ESP* device.

Note: *ESP* device is connected with *USB to UART converter* only by *RX* and *TX* pins.

Device driver is located in `/esp_at_lib/src/system/esp_ll_win32.c`

STM32

Embedded market is supported by many vendors and STMicroelectronics is, with their STM32 series of microcontrollers, one of the most important players. There are numerous amount of examples and topics related to this architecture.

Examples for *STM32* are natively supported with STM32CubeIDE, an official development IDE from STMicroelectronics.

You can run examples on one of official development boards, available in repository examples.

Table 3: Supported development boards

Board name	ESP settings							Debug settings		
	UART	MTX	MRX	RST	GP0	GP2	CHPD	UART	MDTX	MDRX
STM32F769I-Discovery	UART5	PC12	PD2	PJ14	•	•	•	USART1	PA9	PA10
STM32F723E-Discovery	UART5	PC12	PD2	PG14	•	PD6	PD3	USART6	PC6	PC7
STM32L496G-Discovery	USART1	PB6	PG10	PB2	PH2	PA0	PA4	USART2	PA2	PD6
STM32L432KC-Nucleo	USART1	PA9	PA10	PA12	PA7	PA6	PB0	USART2	PA2	PA3
STM32F429ZI-Nucleo	USART2	PD5	PD6	PD1	PD4	PD7	PD3	USART3	PD8	PD9

Pins to connect with ESP device:

- *MTX*: MCU TX pin, connected to ESP RX pin
- *MRX*: MCU RX pin, connected to ESP TX pin
- *RST*: MCU output pin to control reset state of ESP device
- *GP0*: *GPIO0* pin of ESP8266, connected to MCU, configured as output at MCU side
- *GP2*: *GPIO2* pin of ESP8266, connected to MCU, configured as output at MCU side
- *CHPD*: *CH_PD* pin of ESP8266, connected to MCU, configured as output at MCU side

---

Note: *GP0*, *GP2*, *CH_PD* pins are not always necessary for *ESP* device to work properly. When not used, these pins must be tied to fixed values as explained in *ESP* datasheet.

Other pins are for your information and are used for debugging purposes on board.

- MDTX: MCU Debug TX pin, connected via on-board ST-Link to PC
- MDRX: MCU Debug RX pin, connected via on-board ST-Link to PC
- Baudrate is always set to `921600` bauds

5.4.2 Examples list

Here is a list of all examples coming with this library.

Tip: Examples are located in `/examples/` folder in downloaded package. Check *Download library* section to get your package.

Warning: Several examples need to connect to access point first, then they may start client connection or pinging server. Application needs to modify file `/snippets/station_manager.c` and update `ap_list` variable with preferred access points, in order to allow *ESP* to connect to home/local network

Access point

ESP device is configured as software access point, allowing stations to connect to it. When station connects to access point, it will output its *MAC* and *IP* addresses.

Client

Application tries to connect to custom server with classic, event-based API. It starts concurrent connections and processes data in its event callback function.

Server

It starts server on port `80` in event based connection mode. Every client is processed in callback function.

When *ESP* is successfully connected to access point, it is possible to connect to it using its assigned IP address.

Domain name server

ESP tries to get domain name from specific domain name, `example.com` as an example. It needs to be connected to access point to have access to global internet.

MQTT Client

This example demonstrates raw MQTT connection to mosquitto test server. A new application thread is started after *ESP* successfully connects to access point. MQTT application starts by initiating a new TCP connection.

This is event-based example as there is no linear code.

MQTT Client API

Similar to *MQTT Client* examples, but it uses separate thread to process events in blocking mode. Application does not use events to process data, rather it uses blocking API to receive packets

Netconn client

Netconn client is based on sequential API. It starts connection to server, sends initial request and then waits to receive data.

Processing is in separate thread and fully sequential, no callbacks or events.

Netconn server

Netconn server is based on sequential API. It starts server on specific port (see example details) and it waits for new client in separate threads. Once new client has been accepted, it waits for client request and processes data accordingly by sending reply message back.

Tip: Server may accept multiple clients at the same time

INDEX

E

F

G

H

I

K

L

M

N

P

Q

R

S

T

U

V

W

Y