LwESP

Tilen MAJERLE

Apr 15, 2024

CONTENTS

Welcome to the documentation for version latest-develop.

LwESP is generic, platform independent, ESP-AT parser library to communicate with *ESP8266* or *ESP32* WiFi-based microcontrollers from *Espressif systems* using official AT Commands set running on ESP device. 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 Donate

CHAPTER

ONE

FEATURES

- Supports latest ESP32, ESP32-C2, ESP32-C3, ESP32-C6 & ESP8266 AT software from Espressif system
- 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) 2024 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

Getting started may be the most challenging part of every new library. This guide is describing how to start with the library quickly and effectively

5.1.1 Download library

Library is primarly hosted on Github.

You can get it by:

- Downloading latest release from releases area on Github
- Clone `main` branch for latest stable version
- Clone `develop` branch for latest development

Download from releases

All releases are available on Github releases area.

Clone from Github

First-time clone

This is used when you do not have yet local copy on your machine.

- Make sure `git` is installed.
- Open console and navigate to path in the system to clone repository to. Use command `cd your_path`
- Clone repository with one of available options below
  - Run `git clone --recurse-submodules https://github.com/MaJerle/lwesp` command to clone entire repository, including submodules
  - Run `git clone --recurse-submodules --branch develop https://github.com/MaJerle/lwesp` to clone *development* branch, including submodules
  - Run `git clone --recurse-submodules --branch main https://github.com/MaJerle/lwesp` 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 repository is located. Use command `cd your_path`
- Run `git pull origin main` command to get latest changes on `main` branch
- Run `git pull origin develop` command to get latest changes on `develop` branch
- Run `git submodule update --init --remote` to update submodules to latest version

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. Next step is to add the library to the project, by means of source files to compiler inputs and header files in search path.

CMake is the main supported build system. Package comes with the `CMakeLists.txt` and `library.cmake` files, both located in the `lwesp` directory:

- `CMakeLists.txt`: Is a wrapper and only includes `library.cmake` file. It is used if target application uses `add_subdirectory` and then uses `target_link_libraries` to include the library in the project
- `library.cmake`: It is a fully configured set of variables. User must use `include(path/to/library.cmake)` to include the library and must manually add files/includes to the final target

---

Tip: Open `library.cmake` file and manually analyze all the possible variables you can set for full functionality.

If you do not use the *CMake*, you can do the following:

- Copy `lwesp` folder to your project, it contains library files
- Add `lwesp/src/include` folder to *include path* of your toolchain. This is where *C/C++* compiler can find the files during compilation process. Usually using `-I` flag
- Add source files from `lwesp/src/` folder to toolchain build. These files are built by *C/C++* compiler. CMake configuration comes with the library, allows users to include library in the project as **subdirectory** and **library**.
- Copy `lwesp/src/include/lwesp/lwesp_opts_template.h` to project folder and rename it to `lwesp_opts.h`
- Build the project

5.1.3 Configuration file

Configuration file is used to overwrite default settings defined for the essential use case. Library comes with template config file, which can be modified according to the application needs. and it should be copied (or simply renamed in-place) and named `lwesp_opts.h`

Note: Default configuration template file location: `lwesp/src/include/lwesp/lwesp_opts_template.h`. File must be renamed to `lwesp_opts.h` first and then copied to the project directory where compiler include paths have

access to it by using `#include "lwesp_opts.h"`.

Tip: If you are using *CMake* build system, define the variable `LWESP_OPTS_FILE` before adding library's directory to the *CMake* project. Variable must contain the path to the user options file. If not provided and to avoid build error, one will be generated in the build directory.

Configuration options list is available available in the *Configuration* section. If any option is about to be modified, it should be done in configuration file

Listing 1: Template configuration file

```
/**
 * \file            lwesp_opts_template.h
 * \brief           Template config file
 */

/*
 * Copyright (c) 2024 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 LwESP - Lightweight ESP-AT parser library.
 *
 * Author:          Tilen MAJERLE <tilen@majerle.eu>
 * Version:         v1.1.2-dev
 */
#ifndef LWESP_OPTS_HDR_H
#define LWESP_OPTS_HDR_H

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

/*
 * Open "include/lwesp/lwesp_opt.h" and
```

(continues on next page)

(continued from previous page)

```
 * copy & replace here settings you want to change values
 */

#endif /* LWESP_OPTS_HDR_H */
```

Note: If you prefer to avoid using configuration file, application must define a global symbol `LWESP_IGNORE_USER_OPTS`, visible across entire application. This can be achieved with `-D` compiler option.

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.

Fig. 1: Typical application example with access to WiFi

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 for it. Now it is time to use *LwESP* you are reading this manual for.

LwESP 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 *LwESP* do not need to take care about proper command for specific task, they can call API functions, such as *`lwesp_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: *LwESP* 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 *LwESP* 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 middleware. 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 *LWESP_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 middleware 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 `lwesp_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 `LWESP_CFG_INPUT_USE_PROCESS` is enabled and `lwesp_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 `lwesp_evt_register()` function to register a new, custom, event function.

Tip: Implementation of *Netconn API* leverages `lwesp_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            lwesp_netconn.c
 * \brief           API functions for sequential calls
 */

/*
 * Copyright (c) 2024 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,
```

(continues on next page)

(continued from previous page)

```
 * 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 LwESP - Lightweight ESP-AT parser library.
 *
 * Author:          Tilen MAJERLE <tilen@majerle.eu>
 * Version:         v1.1.2-dev
 */
#include "lwesp/lwesp_netconn.h"
#include "lwesp/lwesp_conn.h"
#include "lwesp/lwesp_mem.h"
#include "lwesp/lwesp_private.h"

#if LWESP_CFG_NETCONN || __DOXYGEN__

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

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

/* Check for IP status */
#if LWESP_CFG_IPV6
#define NETCONN_IS_TCP(nc) ((nc)->type == LWESP_NETCONN_TYPE_TCP || (nc)->type == LWESP_
↪NETCONN_TYPE_TCPV6)
#define NETCONN_IS_SSL(nc) ((nc)->type == LWESP_NETCONN_TYPE_SSL || (nc)->type == LWESP_
↪NETCONN_TYPE_SSLV6)
#define NETCONN_IS_UDP(nc) ((nc)->type == LWESP_NETCONN_TYPE_UDP || (nc)->type == LWESP_
↪NETCONN_TYPE_UDPV6)
#else
#define NETCONN_IS_TCP(nc) ((nc)->type == LWESP_NETCONN_TYPE_TCP)
#define NETCONN_IS_SSL(nc) ((nc)->type == LWESP_NETCONN_TYPE_SSL)
#define NETCONN_IS_UDP(nc) ((nc)->type == LWESP_NETCONN_TYPE_UDP)
#endif /* LWESP_CFG_IPV6 */

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

    lwesp_netconn_type_t type;  /*!< Netconn type */
    lwesp_port_t listen_port;   /*!< Port on which we are listening */

```

(continues on next page)

(continued from previous page)

```
    size_t rcv_packets;         /*!< Number of received packets so far on this connection */
    lwesp_conn_p conn;          /*!< Pointer to actual connection */
    uint16_t conn_val_id; /*!< Connection validation ID that changes between every connection active/closed operation */

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

    lwesp_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 LWESP_CFG_NETCONN_RECEIVE_TIMEOUT || __DOXYGEN__
    uint32_t rcv_timeout; /*!< Receive timeout in unit of milliseconds */
#endif
} lwesp_netconn_t;

static uint8_t recv_closed = 0xFF, recv_not_present = 0xFF;
static lwesp_netconn_t* listen_api;   /*!< Main connection in listening mode */
static lwesp_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(lwesp_netconn_t* nc, uint8_t protect) {
    lwesp_pbuf_p pbuf;
    lwesp_netconn_t* new_nc;
    if (protect) {
        lwesp_core_lock();
    }
    if (lwesp_sys_mbox_isvalid(&nc->mbox_receive)) {
        while (lwesp_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) {
                LWESP_DEBUGF(LWESP_CFG_DBG_NETCONN | LWESP_DBG_TYPE_TRACE | LWESP_DBG_LVL_WARNING,
                             "[LWESP NETCONN] flush mboxes. Clearing pbuf 0x%p\r\n", (void*)pbuf);
                lwesp_pbuf_free_s(&pbuf); /* Free received data buffers */
```

(continues on next page)

(continued from previous page)

```
            }
        }
        lwesp_sys_mbox_delete(&nc->mbox_receive);  /* Delete message queue */
        lwesp_sys_mbox_invalid(&nc->mbox_receive); /* Invalid handle */
    }
    if (lwesp_sys_mbox_isvalid(&nc->mbox_accept)) {
        while (lwesp_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) {
                lwesp_netconn_close(new_nc); /* Close netconn connection */
            }
        }
        lwesp_sys_mbox_delete(&nc->mbox_accept);  /* Delete message queue */
        lwesp_sys_mbox_invalid(&nc->mbox_accept); /* Invalid handle */
    }
    if (protect) {
        lwesp_core_unlock();
    }
}

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

    conn = lwesp_conn_get_from_evt(evt); /* Get connection from event */
    switch (lwesp_evt_get_type(evt)) {
        /*
         * A new connection has been active
         * and should be handled by netconn API
         */
        case LWESP_EVT_CONN_ACTIVE: {               /* A new connection active is active␣
↪*/
            if (lwesp_conn_is_client(conn)) {       /* Was connection started by us? */
                nc = lwesp_conn_get_arg(conn);      /* Argument should be already set */
                if (nc != NULL) {
                    nc->conn = conn;                /* Save actual connection */
                    nc->conn_val_id = conn->val_id; /* Get value ID */
                } else {
                    close = 1;                      /* Close this connection, invalid␣
↪netconn */
                }

                /* Is the connection server type and we have known listening API? */
            } else if (lwesp_conn_is_server(conn) && listen_api != NULL) {
                /*
```

(continues on next page)

(continued from previous page)

```
                 * Create a new netconn structure
                 * and set it as connection argument.
                 */
                nc = lwesp_netconn_new(LWESP_NETCONN_TYPE_TCP); /* Create new API */
                LWESP_DEBUGW(LWESP_CFG_DBG_NETCONN | LWESP_DBG_TYPE_TRACE | LWESP_DBG_
↪LVL_WARNING, nc == NULL,
                             "[LWESP NETCONN] Cannot create new structure for incoming␣
↪server connection!\r\n");

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

                    /*
                     * In case there is no listening connection,
                     * simply close the connection
                     */
                    if (!lwesp_sys_mbox_isvalid(&listen_api->mbox_accept)
                        || !lwesp_sys_mbox_putnow(&listen_api->mbox_accept, nc)) {
                        LWESP_DEBUGF(LWESP_CFG_DBG_NETCONN | LWESP_DBG_TYPE_TRACE |␣
↪LWESP_DBG_LVL_WARNING,
                                     "[LWESP NETCONN] Accept MBOX is invalid or it␣
↪cannot insert new nc!\r\n");
                        close = 1;
                    }
                } else {
                    close = 1;
                }
            } else {
                LWESP_DEBUGW(LWESP_CFG_DBG_NETCONN | LWESP_DBG_TYPE_TRACE | LWESP_DBG_
↪LVL_WARNING, listen_api == NULL,
                             "[LWESP 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) {
                    lwesp_conn_set_arg(conn, NULL); /* Reset argument */
                    lwesp_netconn_delete(nc);       /* Free memory for API */
                }
                lwesp_conn_close(conn, 0);          /* Close the connection */
                close = 0;
            }
            break;
        }

        /*
         * We have a new data received which
         * should have netconn structure as argument
```

(continues on next page)

(continued from previous page)

```
         */
        case LWESP_EVT_CONN_RECV: {
            lwesp_pbuf_p pbuf;

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

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

            lwesp_pbuf_ref(pbuf);          /* Increase reference counter */
            LWESP_DEBUGW(LWESP_CFG_DBG_NETCONN | LWESP_DBG_TYPE_TRACE, nc == NULL,
                         "[LWESP NETCONN] Data receive -> netconn is NULL!\r\n");
            LWESP_DEBUGW(LWESP_CFG_DBG_NETCONN | LWESP_DBG_TYPE_TRACE, nc->conn_val_id !
↪= conn->val_id,
                         "[LWESP NETCONN] Connection validation ID does not match␣
↪connection val_id!\r\n");
            LWESP_DEBUGW(LWESP_CFG_DBG_NETCONN | LWESP_DBG_TYPE_TRACE, !lwesp_sys_mbox_
↪isvalid(&nc->mbox_receive),
                         "[LWESP NETCONN] Receive mbox is not valid!\r\n");
            if (nc == NULL || nc->conn_val_id != conn->val_id || !lwesp_sys_mbox_
↪isvalid(&nc->mbox_receive)
                || !lwesp_sys_mbox_putnow(&nc->mbox_receive, pbuf)) {
                LWESP_DEBUGF(LWESP_CFG_DBG_NETCONN,
                             "[LWESP NETCONN] Could not put receive packet. Ignoring␣
↪more data for receive!\r\n");
                lwesp_pbuf_free_s(&pbuf); /* Free pbuf */
                return lwespOKIGNOREMORE; /* Return OK to free the memory and ignore␣
↪further data */
            }
            ++nc->mbox_receive_entries;   /* Increase number of packets in receive mbox␣
↪*/
#if LWESP_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 >= (LWESP_CFG_NETCONN_RECEIVE_QUEUE_LEN - 1)) {
                conn->status.f.receive_blocked = 1; /* Block reading more data */
            }
#endif                                              /* LWESP_CFG_CONN_MANUAL_TCP_RECEIVE␣
↪*/

            ++nc->rcv_packets;                      /* Increase number of packets␣
↪received */
            LWESP_DEBUGF(LWESP_CFG_DBG_NETCONN | LWESP_DBG_TYPE_TRACE,
                         "[LWESP NETCONN] Received pbuf contains %d bytes. Handle␣
↪written to receive mbox\r\n",
                         (int)lwesp_pbuf_length(pbuf, 0));
            break;
        }

        /* Connection was just closed */
```

(continues on next page)

(continued from previous page)

```
        case LWESP_EVT_CONN_CLOSE: {
            nc = lwesp_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 && nc->conn_val_id == conn->val_id && lwesp_sys_mbox_isvalid(&
↪nc->mbox_receive)) {
                if (lwesp_sys_mbox_putnow(&nc->mbox_receive, (void*)&recv_closed)) {
                    ++nc->mbox_receive_entries;
                }
            }
            break;
        }
        default: return lwespERR;
    }
    return lwespOK;
}

/**
 * \brief           Global event callback function
 * \param[in]       evt: Callback information and data
 * \return          \ref lwespOK on success, member of \ref lwespr_t otherwise
 */
static lwespr_t
lwesp_evt(lwesp_evt_t* evt) {
    switch (lwesp_evt_get_type(evt)) {
        case LWESP_EVT_WIFI_DISCONNECTED: { /* Wifi disconnected event */
            if (listen_api != NULL) {       /* Check if listen API active */
                lwesp_sys_mbox_putnow(&listen_api->mbox_accept, &recv_closed);
            }
            break;
        }
        case LWESP_EVT_DEVICE_PRESENT: {                            /* Device present␣
↪event */
            if (listen_api != NULL && !lwesp_device_is_present()) { /* Check if device␣
↪present */
                lwesp_sys_mbox_putnow(&listen_api->mbox_accept, &recv_not_present);
            }
        }
        default: break;
    }
    return lwespOK;
}

/**
 * \brief           Create new netconn connection
 * \param[in]       type: Netconn connection type
 * \return          New netconn connection on success, `NULL` otherwise
 */
lwesp_netconn_p
```

(continues on next page)

(continued from previous page)

```
lwesp_netconn_new(lwesp_netconn_type_t type) {
    lwesp_netconn_t* a;
    static uint8_t first = 1;

    /* Register only once! */
    lwesp_core_lock();
    if (first) {
        first = 0;
        lwesp_evt_register(lwesp_evt); /* Register global event function */
    }
    lwesp_core_unlock();
    a = lwesp_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 (!lwesp_sys_mbox_create(&a->mbox_accept, LWESP_CFG_NETCONN_ACCEPT_QUEUE_LEN))
↪{
            LWESP_DEBUGF(LWESP_CFG_DBG_NETCONN | LWESP_DBG_TYPE_TRACE | LWESP_DBG_LVL_
↪DANGER,
                         "[LWESP NETCONN] Cannot create accept MBOX\r\n");
            goto free_ret;
        }
        if (!lwesp_sys_mbox_create(&a->mbox_receive, LWESP_CFG_NETCONN_RECEIVE_QUEUE_
↪LEN)) {
            LWESP_DEBUGF(LWESP_CFG_DBG_NETCONN | LWESP_DBG_TYPE_TRACE | LWESP_DBG_LVL_
↪DANGER,
                         "[LWESP NETCONN] Cannot create receive MBOX\r\n");
            goto free_ret;
        }
        lwesp_core_lock();
        a->next = netconn_list; /* Add it to beginning of the list */
        netconn_list = a;
        lwesp_core_unlock();
    }
    return a;
free_ret:
    if (lwesp_sys_mbox_isvalid(&a->mbox_accept)) {
        lwesp_sys_mbox_delete(&a->mbox_accept);
        lwesp_sys_mbox_invalid(&a->mbox_accept);
    }
    if (lwesp_sys_mbox_isvalid(&a->mbox_receive)) {
        lwesp_sys_mbox_delete(&a->mbox_receive);
        lwesp_sys_mbox_invalid(&a->mbox_receive);
    }
    if (a != NULL) {
        lwesp_mem_free_s((void**)&a);
    }
    return NULL;
}

/**
 * \brief           Delete netconn connection
```

(continues on next page)

(continued from previous page)

```
 * \param[in]       nc: Netconn handle
 * \return          \ref lwespOK on success, member of \ref lwespr_t enumeration otherwise
 */
lwespr_t
lwesp_netconn_delete(lwesp_netconn_p nc) {
    LWESP_ASSERT(nc != NULL);

    lwesp_core_lock();
    if (nc->conn != NULL) {
        /* No NC for any incoming connections or anything else... */
        lwesp_conn_set_arg(nc->conn, NULL);
    }
    flush_mboxes(nc, 0); /* Clear mboxes */

    /* Stop listening on netconn */
    if (nc == listen_api) {
        listen_api = NULL;
        lwesp_core_unlock();
        lwesp_set_server(0, nc->listen_port, 0, 0, NULL, NULL, NULL, 1);
        lwesp_core_lock();
    }

    /* Remove netconn from linkedlist */
    if (nc == netconn_list) {
        netconn_list = netconn_list->next; /* Remove first from linked list */
    } else if (netconn_list != NULL) {
        lwesp_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;
            }
        }
    }
    if (nc->conn != NULL) {
        /*
         * First delete the connection argument,
         * then close the connection.
         */
        if (lwesp_conn_is_active(nc->conn)) {
            lwesp_conn_close(nc->conn, 1);
        }
        nc->conn = NULL;
    }
    lwesp_core_unlock();

    lwesp_mem_free_s((void**)&nc);
    return lwespOK;
}
```

(continues on next page)

(continued from previous page)

```

/**
 * \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 lwespOK if successfully connected, member of \ref lwespr_t otherwise
 */
lwespr_t
lwesp_netconn_connect(lwesp_netconn_p nc, const char* host, lwesp_port_t port) {
    lwespr_t res;

    LWESP_ASSERT(nc != NULL);
    LWESP_ASSERT(host != NULL);
    LWESP_ASSERT(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 = lwesp_conn_start(NULL, (lwesp_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
 * \param[in]       mode: UDP mode
 * \return          \ref lwespOK if successfully connected, member of \ref lwespr_t otherwise
 */
lwespr_t
lwesp_netconn_connect_ex(lwesp_netconn_p nc, const char* host, lwesp_port_t port, uint16_t keep_alive,
                         const char* local_ip, lwesp_port_t local_port, uint8_t mode) {
    lwesp_conn_start_t cs = {0};
    lwespr_t res;

    LWESP_ASSERT(nc != NULL);
    LWESP_ASSERT(host != NULL);
```

(continues on next page)

(continued from previous page)

```
    LWESP_ASSERT(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 = (lwesp_conn_type_t)nc->type;
    cs.remote_host = host;
    cs.remote_port = port;
    cs.local_ip = local_ip;
    if (NETCONN_IS_TCP(nc) || NETCONN_IS_SSL(nc)) {
        cs.ext.tcp_ssl.keep_alive = keep_alive;
    } else {
        cs.ext.udp.local_port = local_port;
        cs.ext.udp.mode = mode;
    }
    res = lwesp_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 lwespOK on success, member of \ref lwespr_t enumeration otherwise
 */
lwespr_t
lwesp_netconn_bind(lwesp_netconn_p nc, lwesp_port_t port) {
    lwespr_t res = lwespOK;

    LWESP_ASSERT(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;

    return res;
}

/**
 * \brief           Set timeout value in units of seconds when connection is in listening mode
```

(continues on next page)

(continued from previous page)

```
 *                  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 \ref lwesp_netconn_listen
 * \param[in]       nc: Netconn handle used for listen mode
 * \param[in]       timeout: Time in units of seconds. Set to `0` to disable timeout feature
 * \return          \ref lwespOK on success, member of \ref lwespr_t otherwise
 */
lwespr_t
lwesp_netconn_set_listen_conn_timeout(lwesp_netconn_p nc, uint16_t timeout) {
    lwespr_t res = lwespOK;
    LWESP_ASSERT(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 reentrant in this case
     */

    nc->conn_timeout = timeout;

    return res;
}

/**
 * \brief           Listen on previously binded connection
 * \param[in]       nc: Netconn handle used to listen for new connections
 * \return          \ref lwespOK on success, member of \ref lwespr_t enumeration otherwise
 */
lwespr_t
lwesp_netconn_listen(lwesp_netconn_p nc) {
    return lwesp_netconn_listen_with_max_conn(nc, LWESP_CFG_MAX_CONNS);
}

/**
 * \brief           Listen on previously binded connection with max allowed connections at a time
 * \param[in]       nc: Netconn handle used to listen for new connections
 * \param[in]       max_connections: Maximal number of connections server can accept at a time
 *                      This parameter may not be larger than \ref LWESP_CFG_MAX_CONNS
 * \return          \ref lwespOK on success, member of \ref lwespr_t otherwise
 */
lwespr_t
lwesp_netconn_listen_with_max_conn(lwesp_netconn_p nc, uint16_t max_connections) {
    lwespr_t res;

    LWESP_ASSERT(nc != NULL);
```

(continues on next page)

(continued from previous page)

```
    LWESP_ASSERT(NETCONN_IS_TCP(nc));

    /* Enable server on port and set default netconn callback */
    if ((res = lwesp_set_server(1, nc->listen_port, LWESP_U16(LWESP_MIN(max_connections, LWESP_CFG_MAX_CONNS)),
                                nc->conn_timeout, netconn_evt, NULL, NULL, 1))
        == lwespOK) {
        lwesp_core_lock();
        listen_api = nc; /* Set current main API in listening state */
        lwesp_core_unlock();
    }
    return res;
}

/**
 * \brief           Accept a new connection
 * \param[in]       nc: Netconn handle used as base connection to accept new clients
 * \param[out]      client: Pointer to netconn handle to save new connection to
 * \return          \ref lwespOK on success, member of \ref lwespr_t enumeration, otherwise
 */
lwespr_t
lwesp_netconn_accept(lwesp_netconn_p nc, lwesp_netconn_p* client) {
    lwesp_netconn_t* tmp;
    uint32_t time;

    LWESP_ASSERT(nc != NULL);
    LWESP_ASSERT(client != NULL);
    LWESP_ASSERT(NETCONN_IS_TCP(nc));
    LWESP_ASSERT(nc == listen_api);

    *client = NULL;
    time = lwesp_sys_mbox_get(&nc->mbox_accept, (void**)&tmp, 0);
    if (time == LWESP_SYS_TIMEOUT) {
        return lwespTIMEOUT;
    }
    if ((uint8_t*)tmp == (uint8_t*)&recv_closed) {
        lwesp_core_lock();
        listen_api = NULL;              /* Disable listening at this point */
        lwesp_core_unlock();
        return lwespERRWIFINOTCONNECTED; /* Wifi disconnected */
    } else if ((uint8_t*)tmp == (uint8_t*)&recv_not_present) {
        lwesp_core_lock();
        listen_api = NULL;       /* Disable listening at this point */
        lwesp_core_unlock();
        return lwespERRNODEVICE; /* Device not present */
    }
    *client = tmp;               /* Set new pointer */
    return lwespOK;              /* We have a new connection */
}

/**
```

(continues on next page)

(continued from previous page)

```
 * \brief           Write data to connection output buffers
 * \note            This function may only be used on TCP or SSL connections
 * \param[in]       nc: Netconn handle used to write data to
 * \param[in]       data: Pointer to data to write
 * \param[in]       btw: Number of bytes to write
 * \return          \ref lwespOK on success, member of \ref lwespr_t enumeration␣
↪otherwise
 */
lwespr_t
lwesp_netconn_write(lwesp_netconn_p nc, const void* data, size_t btw) {
    size_t len, sent;
    const uint8_t* d = data;
    lwespr_t res;

    LWESP_ASSERT(nc != NULL);
    LWESP_ASSERT(NETCONN_IS_TCP(nc) || NETCONN_IS_SSL(nc));
    LWESP_ASSERT(lwesp_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 need 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 have impact on speed␣
↪and effectivenes)
     */

    /* Step 1 */
    if (nc->buff.buff != NULL) {                            /* Is there a write buffer␣
↪ready to accept more data? */
        len = LWESP_MIN(nc->buff.len - nc->buff.ptr, btw); /* Get number of bytes we can␣
↪write to buffer */
        if (len > 0) {
            LWESP_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 = lwesp_conn_send(nc->conn, nc->buff.buff, nc->buff.len, &sent, 1);

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

(continues on next page)

(continued from previous page)

```
        }
    }

    /* Step 2 */
    if (btw >= LWESP_CFG_CONN_MAX_DATA_LEN) {
        size_t rem;
        rem = btw % LWESP_CFG_CONN_MAX_DATA_LEN;                /* Get remaining bytes␣
↪for max data length */
        res = lwesp_conn_send(nc->conn, d, btw - rem, &sent, 1); /* Write data directly␣
↪*/
        if (res != lwespOK) {
            return res;
        }
        d += sent;   /* Advance in data pointer */
        btw -= sent; /* Decrease remaining data to send */
    }

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

    /* Step 3 */
    if (nc->buff.buff == NULL) {                    /* Check if we should allocate a new␣
↪buffer */
        nc->buff.buff = lwesp_mem_malloc(sizeof(*nc->buff.buff) * LWESP_CFG_CONN_MAX_
↪DATA_LEN);
        nc->buff.len = LWESP_CFG_CONN_MAX_DATA_LEN; /* Save buffer length */
        nc->buff.ptr = 0;                           /* Save buffer pointer */
    }

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

/**
 * \brief           Extended version of \ref lwesp_netconn_write with additional
 *                  option to set custom flags.
 *
 * \note            It is recommended to use this for full features support
 *
 * \param[in]       nc: Netconn handle used to write data to
 * \param[in]       data: Pointer to data to write
 * \param[in]       btw: Number of bytes to write
 * \param           flags: Bitwise-ORed set of flags for netconn.
```

(continues on next page)

(continued from previous page)

```
 *                      Flags start with \ref LWESP_NETCONN_FLAG_xxx
 * \return          \ref lwespOK on success, member of \ref lwespr_t enumeration␣
↪otherwise
 */
lwespr_t
lwesp_netconn_write_ex(lwesp_netconn_p nc, const void* data, size_t btw, uint16_t flags)
↪{
    lwespr_t res = lwesp_netconn_write(nc, data, btw);
    if (res == lwespOK) {
        if (flags & LWESP_NETCONN_FLAG_FLUSH) {
            res = lwesp_netconn_flush(nc);
        }
    }
    return res;
}

/**
 * \brief           Flush buffered data on netconn TCP/SSL connection
 * \note            This function may only be used on TCP/SSL connection
 * \param[in]       nc: Netconn handle to flush data
 * \return          \ref lwespOK on success, member of \ref lwespr_t enumeration␣
↪otherwise
 */
lwespr_t
lwesp_netconn_flush(lwesp_netconn_p nc) {
    LWESP_ASSERT(nc != NULL);
    LWESP_ASSERT(NETCONN_IS_TCP(nc) || NETCONN_IS_SSL(nc));
    LWESP_ASSERT(lwesp_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? */
            lwesp_conn_send(nc->conn, nc->buff.buff, nc->buff.ptr, NULL, 1); /* Send␣
↪data */
        }
        lwesp_mem_free_s((void**)&nc->buff.buff);
    }
    return lwespOK;
}

/**
 * \brief           Send data on 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 lwespOK on success, member of \ref lwespr_t enumeration␣
↪otherwise
```

(continues on next page)

(continued from previous page)

```
 */
lwespr_t
lwesp_netconn_send(lwesp_netconn_p nc, const void* data, size_t btw) {
    LWESP_ASSERT(nc != NULL);
    LWESP_ASSERT(nc->type == LWESP_NETCONN_TYPE_UDP);
    LWESP_ASSERT(lwesp_conn_is_active(nc->conn));

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

/**
 * \brief           Send data on 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 lwespOK on success, member of \ref lwespr_t enumeration␣
↪otherwise
 */
lwespr_t
lwesp_netconn_sendto(lwesp_netconn_p nc, const lwesp_ip_t* ip, lwesp_port_t port, const␣
↪void* data, size_t btw) {
    LWESP_ASSERT(nc != NULL);
    LWESP_ASSERT(nc->type == LWESP_NETCONN_TYPE_UDP);
    LWESP_ASSERT(lwesp_conn_is_active(nc->conn));

    return lwesp_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 lwespOK when new data ready
 * \return          \ref lwespCLOSED when connection closed by remote side
 * \return          \ref lwespTIMEOUT when receive timeout occurs
 * \return          Any other member of \ref lwespr_t otherwise
 */
lwespr_t
lwesp_netconn_receive(lwesp_netconn_p nc, lwesp_pbuf_p* pbuf) {
    LWESP_ASSERT(nc != NULL);
    LWESP_ASSERT(pbuf != NULL);

    *pbuf = NULL;
#if LWESP_CFG_NETCONN_RECEIVE_TIMEOUT
    /*
     * Wait for new received data for up to specific timeout
     * or throw error for timeout notification
```

(continues on next page)

(continued from previous page)

```
     */
    if (nc->rcv_timeout == LWESP_NETCONN_RECEIVE_NO_WAIT) {
        if (!lwesp_sys_mbox_getnow(&nc->mbox_receive, (void**)pbuf)) {
            return lwespTIMEOUT;
        }
    } else if (lwesp_sys_mbox_get(&nc->mbox_receive, (void**)pbuf, nc->rcv_timeout) == LWESP_SYS_TIMEOUT) {
        return lwespTIMEOUT;
    }
#else  /* LWESP_CFG_NETCONN_RECEIVE_TIMEOUT */
    /* Forever wait for new receive packet */
    lwesp_sys_mbox_get(&nc->mbox_receive, (void**)pbuf, 0);
#endif /* !LWESP_CFG_NETCONN_RECEIVE_TIMEOUT */

    lwesp_core_lock();
    if (nc->mbox_receive_entries > 0) {
        --nc->mbox_receive_entries;
    }
    lwesp_core_unlock();

    /* Check if connection closed */
    if ((uint8_t*)(*pbuf) == (uint8_t*)&recv_closed) {
        *pbuf = NULL; /* Reset pbuf */
        LWESP_DEBUGF(LWESP_CFG_DBG_NETCONN | LWESP_DBG_TYPE_TRACE | LWESP_DBG_LVL_WARNING,
                     "[LWESP NETCONN] netcon_receive: Got object handle for close event\r\n");
        return lwespCLOSED;
    }
#if LWESP_CFG_CONN_MANUAL_TCP_RECEIVE
    else {
        lwesp_core_lock();
        nc->conn->status.f.receive_blocked = 0; /* Resume reading more data */
        lwesp_conn_recved(nc->conn, *pbuf);     /* Notify stack about received data */
        lwesp_core_unlock();
    }
#endif /* LWESP_CFG_CONN_MANUAL_TCP_RECEIVE */
    LWESP_DEBUGF(LWESP_CFG_DBG_NETCONN | LWESP_DBG_TYPE_TRACE | LWESP_DBG_LVL_WARNING,
                 "[LWESP NETCONN] netcon_receive: Got pbuf object handle at 0x%p. Len/Tot_len: %u/%u\r\n", (void*)*pbuf,
                 (unsigned)lwesp_pbuf_length(*pbuf, 0), (unsigned)lwesp_pbuf_length(*pbuf, 1));
    return lwespOK; /* We have data available */
}

/**
 * \brief           Close a netconn connection
 * \param[in]       nc: Netconn handle to close
 * \return          \ref lwespOK on success, member of \ref lwespr_t enumeration otherwise
 */
lwespr_t
```

(continues on next page)

(continued from previous page)

```
lwesp_netconn_close(lwesp_netconn_p nc) {
    lwesp_conn_p conn;

    LWESP_ASSERT(nc != NULL);
    LWESP_ASSERT(nc->conn != NULL);
    LWESP_ASSERT(lwesp_conn_is_active(nc->conn));

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

    lwesp_conn_set_arg(conn, NULL); /* Reset argument */
    lwesp_conn_close(conn, 1);      /* Close the connection */
    flush_mboxes(nc, 1);            /* Flush message queues */
    return lwespOK;
}

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

#if LWESP_CFG_NETCONN_RECEIVE_TIMEOUT || __DOXYGEN__

/**
 * \brief           Set timeout value for receiving data.
 *
 * When enabled, \ref lwesp_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. Function blocks until data␣
↪receive or connection closed
 *                      Set to `> 0` to set maximum milliseconds to wait before timeout
 *                      Set to \ref LWESP_NETCONN_RECEIVE_NO_WAIT to enable non-blocking␣
↪receive
 */
void
lwesp_netconn_set_receive_timeout(lwesp_netconn_p nc, uint32_t timeout) {
    nc->rcv_timeout = timeout;
}

```

(continues on next page)

(continued from previous page)

```
/**
 * \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
lwesp_netconn_get_receive_timeout(lwesp_netconn_p nc) {
    return nc->rcv_timeout;
}

#endif /* LWESP_CFG_NETCONN_RECEIVE_TIMEOUT || __DOXYGEN__ */

/**
 * \brief           Get netconn connection handle
 * \param[in]       nc: Netconn handle
 * \return          ESP connection handle
 */
lwesp_conn_p
lwesp_netconn_get_conn(lwesp_netconn_p nc) {
    return nc->conn;
}

/**
 * \brief           Get netconn connection type
 * \param[in]       nc: Netconn handle
 * \return          ESP connection type
 */
lwesp_netconn_type_t
lwesp_netconn_get_type(lwesp_netconn_p nc) {
    return nc->type;
}

#endif /* LWESP_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 `LWESP_EVT_CONN_*`, such as `LWESP_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 *`lwesp_conn_start()`* function
- Application sets global event callback function when enabling server mode with *`lwesp_set_server()`* function

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

```c
#include "client.h"
#include "lwesp/lwesp.h"

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

static lwespr_t   conn_callback_func(lwesp_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) {
    lwespr_t res;

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

    /* Start 2 more */
    lwesp_conn_start(NULL, LWESP_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 LWESP_EVT_CONN_ERROR event should be triggered
     * in callback function processing
     */
    lwesp_conn_start(NULL, LWESP_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
```

(continues on next page)

(continued from previous page)

```
 * \return          \ref lwespOK on success, member of \ref lwespr_t otherwise
 */
static lwespr_t
conn_callback_func(lwesp_evt_t* evt) {
    lwesp_conn_p conn;
    lwespr_t res;
    uint8_t conn_num;

    conn = lwesp_conn_get_from_evt(evt);          /* Get connection handle from event */
    if (conn == NULL) {
        return lwespERR;
    }
    conn_num = lwesp_conn_getnum(conn);           /* Get connection number for␣
↪identification */
    switch (lwesp_evt_get_type(evt)) {
        case LWESP_EVT_CONN_ACTIVE: {             /* Connection just active */
            printf("Connection %d active!\r\n", (int)conn_num);
            res = lwesp_conn_send(conn, req_data, sizeof(req_data) - 1, NULL, 0); /*␣
↪Start sending data in non-blocking mode */
            if (res == lwespOK) {
                printf("Sending request data to server...\r\n");
            } else {
                printf("Cannot send request data to server. Closing connection manually..
↪.\r\n");
                lwesp_conn_close(conn, 0);        /* Close the connection */
            }
            break;
        }
        case LWESP_EVT_CONN_CLOSE: {              /* Connection closed */
            if (lwesp_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 LWESP_EVT_CONN_SEND: {               /* Data send event */
            lwespr_t res = lwesp_evt_conn_send_get_result(evt);
            if (res == lwespOK) {
                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 LWESP_EVT_CONN_RECV: {               /* Data received from remote side */
            lwesp_pbuf_p pbuf = lwesp_evt_conn_recv_get_buff(evt);
            lwesp_conn_recved(conn, pbuf);        /* Notify stack about received pbuf */
            printf("Received %d bytes on connection %d..\r\n", (int)lwesp_pbuf_
↪length(pbuf, 1), (int)conn_num);
            break;
```

(continues on next page)

(continued from previous page)

```
        }
        case LWESP_EVT_CONN_ERROR: {            /* Error connecting to server */
            const char* host = lwesp_evt_conn_error_get_host(evt);
            lwesp_port_t port = lwesp_evt_conn_error_get_port(evt);
            printf("Error connecting to %s:%d\r\n", host, (int)port);
            break;
        }
        default:
            break;
    }
    return lwespOK;
}
```

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

```
/*
 * This snippet shows how to use ESP's DNS module to
 * obtain IP address from domain name
 */
#include "dns.h"
#include "lwesp/lwesp.h"

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

/**
 * \brief           Variable to hold result of DNS resolver
 */
static lwesp_ip_t ip;

/**
 * \brief           Function to print actual resolved IP address
 */
static void
prv_print_ip(void) {
    if (0) {
```

(continues on next page)

(continued from previous page)

```
#if LWESP_CFG_IPV6
    } else if (ip.type == LWESP_IPTYPE_V6) {
        printf("IPv6: %04X:%04X:%04X:%04X:%04X:%04X:%04X:%04X\r\n",
            (unsigned)ip.addr.ip6.addr[0], (unsigned)ip.addr.ip6.addr[1], (unsigned)ip.addr.ip6.addr[2],
            (unsigned)ip.addr.ip6.addr[3], (unsigned)ip.addr.ip6.addr[4], (unsigned)ip.addr.ip6.addr[5],
            (unsigned)ip.addr.ip6.addr[6], (unsigned)ip.addr.ip6.addr[7]);
#endif /* LWESP_CFG_IPV6 */
    } else {
        printf("IPv4: %d.%d.%d.%d\r\n",
            (int)ip.addr.ip4.addr[0], (int)ip.addr.ip4.addr[1], (int)ip.addr.ip4.addr[2], (int)ip.addr.ip4.addr[3]);
    }
}

/**
 * \brief           Event callback function for API call,
 *                  called when API command finished with execution
 */
static void
prv_dns_resolve_evt(lwespr_t res, void* arg) {
    LWESP_UNUSED(arg);
    /* Check result of command */
    if (res == lwespOK) {
        /* Print actual resolved IP */
        prv_print_ip();
    }
}

/**
 * \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 (lwesp_dns_gethostbyname(DNS_HOST2, &ip, prv_dns_resolve_evt, DNS_HOST2, 0) == lwespOK) {
        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 (lwesp_dns_gethostbyname(DNS_HOST1, &ip, prv_dns_resolve_evt, DNS_HOST1, 1) == lwespOK) {
        /* Print actual resolved IP */
        prv_print_ip();
    } else {
        printf("Could not retrieve IP address for " DNS_HOST1 "\r\n");
```

(continues on next page)

(continued from previous page)

```
70      }
71  }
```

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 (lwesp_hostname_get(hostname, sizeof(hostname), NULL, NULL, 1 /* 1 means blocking␣
↪call */) == lwespOK) {
    /* 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 not 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 lwesp_hostname_get() function finishes */
void
hostname_fn(lwespr_t res, void* arg) {
    /* Check actual result from device */
    if (res == lwespOK) {
        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 (lwesp_hostname_get(hostname, sizeof(hostname), hostname_fn, NULL, 0 /* 0 means non-
↪blocking call */) == lwespOK) {
    /* 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");
```

(continues on next page)

(continued from previous page)

```
} 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 lwesp_hostname_get() function finishes */
void
hostname_fn(lwespr_t res, void* arg) {
    /* Check actual result from device */
    if (res == lwespOK) {
        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 (lwesp_hostname_get(hostname, sizeof(hostname), hostname_fn, NULL, 0 /* 0 means␣
↪non-blocking call */) == lwespOK) {
        /* 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");
    }
}
```

5.2.6 Porting guide

High level of *ESP-AT* library is platform independent, written in C (C11), 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 prepare all parts of *low-level* driver, application must take care of:

- Implementing `lwesp_ll_init()` and `lwesp_ll_deinit()` callback functions
- Implement and assign *send data* and optional *hardware reset* function callbacks
- Assign memory for allocation manager when using default allocator or use custom allocator
- Process received data from *ESP* device and send it 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 `lwesp_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 `lwesp/src/system/lwesp_sys_template.c` to the same folder and rename it to application port, eg. `lwesp_sys_win32.c`
- Open newly created file and implement all system functions
- Copy folder `lwesp/src/include/system/port/template/*` to the same folder and rename *folder name* to application port, eg. `cmsis_os`
- Open `lwesp_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 *lwesp_input_process()* or *lwesp_input()* functions, based on application configuration of *LWESP_CFG_INPUT_USE_PROCESS* parameter.
  - When *LWESP_CFG_INPUT_USE_PROCESS* is disabled, dedicated receive buffer is created by *ESP-AT* library and *lwesp_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 *LWESP_CFG_INPUT_USE_PROCESS* is enabled, *lwesp_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 `LWESP_MEM_SIZE` size
- It sets *send* and *reset* callback functions for *ESP-AT* library

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

```
/**
 * \file            lwesp_ll_win32.c
 * \brief           Low-level communication with ESP device for WIN32
 */

/*
 * Copyright (c) 2024 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:
 *
```

(continues on next page)

(continued from previous page)

```
 * 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 LwESP - Lightweight ESP-AT parser library.
 *
 * Author:          Tilen MAJERLE <tilen@majerle.eu>
 * Version:         v1.1.2-dev
 */
#include "lwesp/lwesp.h"
#include "lwesp/lwesp_input.h"
#include "lwesp/lwesp_mem.h"
#include "system/lwesp_ll.h"

#if !__DOXYGEN__

volatile uint8_t lwesp_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 !LWESP_CFG_AT_ECHO && 0
        const uint8_t* d = data;
        HANDLE hConsole;

        hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
        SetConsoleTextAttribute(hConsole, FOREGROUND_RED);
        for (DWORD i = 0; i < len; ++i) {
            printf("%c", d[i]);
        }
        SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE);
#endif /* !LWESP_CFG_AT_ECHO */
```

(continues on next page)

(continued from previous page)

```
67
68          WriteFile(com_port, data, len, &written, NULL);
69          FlushFileBuffers(com_port);
70          return written;
71      }
72      return 0;
73  }
74
75  /**
76   * \brief           Configure UART (USB to UART)
77   * \return          `1` if initialized, `0` otherwise
78   */
79  static uint8_t
80  configure_uart(uint32_t baudrate) {
81      size_t i;
82      DCB dcb = {.DCBlength = sizeof(dcb)};
83
84      /*
85       * List of COM ports to probe for ESP devices
86       * This may be different on your computer
87       */
88      static const char* com_port_names[] = {"\\\\.\\COM7", "\\\\.\\COM60", "\\\\.\\COM4",
    ↪"\\\\.\\COM8",
89                                             "\\\\.\\COM9", "\\\\.\\COM10", "\\\\.\\COM17"}
    ↪;
90
91      /* Try to open one of listed COM ports */
92      if (!initialized) {
93          printf("Initializing COM port first time\r\n");
94          for (i = 0; i < LWESP_ARRAYSIZE(com_port_names); ++i) {
95              printf("Trying to open COM port %s\r\n", com_port_names[i]);
96              com_port = CreateFileA(com_port_names[i], GENERIC_READ | GENERIC_WRITE, 0, 0,
    ↪ OPEN_EXISTING, 0, NULL);
97              if (GetCommState(com_port, &dcb)) {
98                  printf("Successfully received info for COM port %s. Using this one..\r\n
    ↪", com_port_names[i]);
99                  break;
100             } else {
101                 printf("Could not get info for COM port %s\r\n", com_port_names[i]);
102             }
103         }
104         if (i == LWESP_ARRAYSIZE(com_port_names)) {
105             printf("Could not get info for any COM port. Entering while loop\r\n");
106             while (1) {
107                 Sleep(1000);
108             }
109         }
110     }
111
112     /* Configure COM port parameters */
113     if (GetCommState(com_port, &dcb)) {
114         COMMTIMEOUTS timeouts;
```

(continues on next page)

(continued from previous page)

```
        /* Set port config */
        dcb.BaudRate = baudrate;
        dcb.ByteSize = 8;
        dcb.Parity = NOPARITY;
        dcb.StopBits = ONESTOPBIT;
        if (SetCommState(com_port, &dcb)) {
            /* Set timeouts config */
            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)) {
                    GetCommTimeouts(com_port, &timeouts);
                } else {
                    printf("[LWESP LL] Could not set port timeout config\r\n");
                }
            } else {
                printf("[LWESP LL] Could not get port timeout config\r\n");
            }
        } else {
            printf("[LWESP LL] Could not set port config\r\n");
        }
    } else {
        printf("[LWESP LL] Could not get port info\r\n");
    }

    /* On first function call, create a thread to read data from COM port */
    if (!initialized) {
        lwesp_sys_thread_create(&thread_handle, "lwesp_ll_thread", uart_thread, NULL, 0,␣
↪0);
    }
    return 1;
}

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

    LWESP_UNUSED(param);

    lwesp_sys_sem_create(&sem, 0); /* Create semaphore for delay functions */
    while (com_port == NULL) {
        lwesp_sys_sem_wait(&sem, 1); /* Add some delay with yield */
    }

```

(continues on next page)

(continued from previous page)

```
    fopen_s(&file, "log_file.txt", "w+"); /* Open debug file in write mode */
    while (1) {
        while (com_port == NULL) {
            lwesp_sys_sem_wait(&sem, 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);
#if 0
                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);
#endif

                if (lwesp_ll_win32_driver_ignore_data) {
                    printf("IGNORING..\r\n");
                    continue;
                }

                /* Send received data to input processing module */
#if LWESP_CFG_INPUT_USE_PROCESS
                lwesp_input_process(data_buffer, (size_t)bytes_read);
#else  /* LWESP_CFG_INPUT_USE_PROCESS */
                lwesp_input(data_buffer, (size_t)bytes_read);
#endif /* !LWESP_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 */
        lwesp_sys_sem_wait(&sem, 1);
    }
}

/**
 * \brief           Reset device GPIO management
 */
```

(continues on next page)

(continued from previous page)

```
static uint8_t
reset_device(uint8_t state) {
    LWESP_UNUSED(state);
    return 0; /* Hardware reset was not successful */
}

/**
 * \brief           Callback function called from initialization process
 */
lwespr_t
lwesp_ll_init(lwesp_ll_t* ll) {
#if !LWESP_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
     */
    lwesp_mem_region_t mem_regions[] = {{memory, sizeof(memory)}};
    if (!initialized) {
        lwesp_mem_assignmemory(mem_regions,
                               LWESP_ARRAYSIZE(mem_regions)); /* Assign memory for␣
↪allocations to ESP library */
    }
#endif /* !LWESP_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 */
    if (!configure_uart(ll->uart.baudrate)) { /* Initialize UART for communication */
        return lwespERR;
    }
    initialized = 1;
    return lwespOK;
}

/**
 * \brief           Callback function to de-init low-level communication part
 */
lwespr_t
lwesp_ll_deinit(lwesp_ll_t* ll) {
    LWESP_UNUSED(ll);
    if (thread_handle != NULL) {
        lwesp_sys_thread_terminate(&thread_handle);
        thread_handle = NULL;
```

(continues on next page)

(continued from previous page)

```
    }
    initialized = 0; /* Clear initialized flag */
    return lwespOK;
}

#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 *lwesp_input_process()* function to directly process received data without using intermediate receive buffer
- Memory manager has been assigned to 1 region of `LWESP_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            lwesp_ll_stm32.c
 * \brief           Generic STM32 driver, included in various STM32 driver variants
 */

/*
 * Copyright (c) 2024 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 LwESP - Lightweight ESP-AT parser library.
 *
```

(continues on next page)

(continued from previous page)

```
 * Author:          Tilen MAJERLE <tilen@majerle.eu>
 * Version:         v1.1.2-dev
 */

/*
 * How it works
 *
 * On first call to \ref lwesp_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 LWESP_CFG_INPUT_USE_PROCESS must be enabled in `lwesp_config.h` to use this driver.
 */
#include "lwesp/lwesp.h"
#include "lwesp/lwesp_input.h"
#include "lwesp/lwesp_mem.h"
#include "system/lwesp_ll.h"

#if !__DOXYGEN__

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

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

#if !defined(LWESP_MEM_SIZE)
#define LWESP_MEM_SIZE 0x4000
#endif /* !defined(LWESP_MEM_SIZE) */

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

/* USART memory */
static uint8_t usart_mem[LWESP_USART_DMA_RX_BUFF_SIZE];
static uint8_t is_running, initialized;
static size_t old_pos;

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

```

(continues on next page)

(continued from previous page)

```
/* Message queue */
static osMessageQueueId_t usart_ll_mbox_id;

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

    LWESP_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(LWESP_USART_DMA_RX_STREAM)
        pos = sizeof(usart_mem) - LL_DMA_GetDataLength(LWESP_USART_DMA, LWESP_USART_DMA_
↪RX_STREAM);
#else
        pos = sizeof(usart_mem) - LL_DMA_GetDataLength(LWESP_USART_DMA, LWESP_USART_DMA_
↪RX_CH);
#endif /* defined(LWESP_USART_DMA_RX_STREAM) */
        if (pos != old_pos && is_running) {
            if (pos > old_pos) {
                lwesp_input_process(&usart_mem[old_pos], pos - old_pos);
            } else {
                lwesp_input_process(&usart_mem[old_pos], sizeof(usart_mem) - old_pos);
                if (pos > 0) {
                    lwesp_input_process(&usart_mem[0], pos);
                }
            }
            old_pos = pos;
        }
    }
}

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

    if (!initialized) {
        /* Enable peripheral clocks */
        LWESP_USART_CLK;
```

(continues on next page)

(continued from previous page)

```
        LWESP_USART_DMA_CLK;
        LWESP_USART_TX_PORT_CLK;
        LWESP_USART_RX_PORT_CLK;

#if defined(LWESP_RESET_PIN)
        LWESP_RESET_PORT_CLK;
#endif /* defined(LWESP_RESET_PIN) */

#if defined(LWESP_GPIO0_PIN)
        LWESP_GPIO0_PORT_CLK;
#endif /* defined(LWESP_GPIO0_PIN) */

#if defined(LWESP_GPIO2_PIN)
        LWESP_GPIO2_PORT_CLK;
#endif /* defined(LWESP_GPIO2_PIN) */

#if defined(LWESP_CH_PD_PIN)
        LWESP_CH_PD_PORT_CLK;
#endif /* defined(LWESP_CH_PD_PIN) */

        /* Global pin configuration */
        LL_GPIO_StructInit(&gpio_init);
        gpio_init.OutputType = LL_GPIO_OUTPUT_PUSHPULL;
        gpio_init.Pull = LL_GPIO_PULL_UP;
        gpio_init.Speed = LL_GPIO_SPEED_FREQ_VERY_HIGH;
        gpio_init.Mode = LL_GPIO_MODE_OUTPUT;

#if defined(LWESP_RESET_PIN)
        /* Configure RESET pin */
        gpio_init.Pin = LWESP_RESET_PIN;
        LL_GPIO_Init(LWESP_RESET_PORT, &gpio_init);
#endif /* defined(LWESP_RESET_PIN) */

#if defined(LWESP_GPIO0_PIN)
        /* Configure GPIO0 pin */
        gpio_init.Pin = LWESP_GPIO0_PIN;
        LL_GPIO_Init(LWESP_GPIO0_PORT, &gpio_init);
        LL_GPIO_SetOutputPin(LWESP_GPIO0_PORT, LWESP_GPIO0_PIN);
#endif /* defined(LWESP_GPIO0_PIN) */

#if defined(LWESP_GPIO2_PIN)
        /* Configure GPIO2 pin */
        gpio_init.Pin = LWESP_GPIO2_PIN;
        LL_GPIO_Init(LWESP_GPIO2_PORT, &gpio_init);
        LL_GPIO_SetOutputPin(LWESP_GPIO2_PORT, LWESP_GPIO2_PIN);
#endif /* defined(LWESP_GPIO2_PIN) */

#if defined(LWESP_CH_PD_PIN)
        /* Configure CH_PD pin */
        gpio_init.Pin = LWESP_CH_PD_PIN;
        LL_GPIO_Init(LWESP_CH_PD_PORT, &gpio_init);
        LL_GPIO_SetOutputPin(LWESP_CH_PD_PORT, LWESP_CH_PD_PIN);
```

(continues on next page)

(continued from previous page)

```
#endif /* defined(LWESP_CH_PD_PIN) */

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

        /* TX PIN */
        gpio_init.Alternate = LWESP_USART_TX_PIN_AF;
        gpio_init.Pin = LWESP_USART_TX_PIN;
        LL_GPIO_Init(LWESP_USART_TX_PORT, &gpio_init);

        /* RX PIN */
        gpio_init.Alternate = LWESP_USART_RX_PIN_AF;
        gpio_init.Pin = LWESP_USART_RX_PIN;
        LL_GPIO_Init(LWESP_USART_RX_PORT, &gpio_init);

        /* Configure UART */
        LL_USART_DeInit(LWESP_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(LWESP_USART, &usart_init);

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

        /* Enable USART interrupts */
        NVIC_SetPriority(LWESP_USART_IRQ, NVIC_EncodePriority(NVIC_GetPriorityGrouping(),
↪ 0x07, 0x00));
        NVIC_EnableIRQ(LWESP_USART_IRQ);

        /* Configure DMA */
        is_running = 0;
#if defined(LWESP_USART_DMA_RX_STREAM)
        LL_DMA_DeInit(LWESP_USART_DMA, LWESP_USART_DMA_RX_STREAM);
#if defined(LWESP_USART_DMA_RX_CH)
        dma_init.Channel = LWESP_USART_DMA_RX_CH;
#else
        dma_init.PeriphRequest = LWESP_USART_DMA_RX_REQ_NUM;
#endif /* !defined(STM32F4xx) && !defined(STM32F7xx) && !defined(STM32F2xx) */
#else
        LL_DMA_DeInit(LWESP_USART_DMA, LWESP_USART_DMA_RX_CH);
        dma_init.PeriphRequest = LWESP_USART_DMA_RX_REQ_NUM;
#endif /* defined(LWESP_USART_DMA_RX_STREAM) */
        dma_init.PeriphOrM2MSrcAddress = (uint32_t)&LWESP_USART->LWESP_USART_RDR_NAME;
```

(continues on next page)

(continued from previous page)

```
        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(LWESP_USART_DMA_RX_STREAM)
        LL_DMA_Init(LWESP_USART_DMA, LWESP_USART_DMA_RX_STREAM, &dma_init);
#else
        LL_DMA_Init(LWESP_USART_DMA, LWESP_USART_DMA_RX_CH, &dma_init);
#endif /* defined(LWESP_USART_DMA_RX_STREAM) */

        /* Enable DMA interrupts */
#if defined(LWESP_USART_DMA_RX_STREAM)
        LL_DMA_EnableIT_HT(LWESP_USART_DMA, LWESP_USART_DMA_RX_STREAM);
        LL_DMA_EnableIT_TC(LWESP_USART_DMA, LWESP_USART_DMA_RX_STREAM);
        LL_DMA_EnableIT_TE(LWESP_USART_DMA, LWESP_USART_DMA_RX_STREAM);
        LL_DMA_EnableIT_FE(LWESP_USART_DMA, LWESP_USART_DMA_RX_STREAM);
        LL_DMA_EnableIT_DME(LWESP_USART_DMA, LWESP_USART_DMA_RX_STREAM);
#else
        LL_DMA_EnableIT_HT(LWESP_USART_DMA, LWESP_USART_DMA_RX_CH);
        LL_DMA_EnableIT_TC(LWESP_USART_DMA, LWESP_USART_DMA_RX_CH);
        LL_DMA_EnableIT_TE(LWESP_USART_DMA, LWESP_USART_DMA_RX_CH);
#endif /* defined(LWESP_USART_DMA_RX_STREAM) */

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

        old_pos = 0;
        is_running = 1;

        /* Start DMA and USART */
#if defined(LWESP_USART_DMA_RX_STREAM)
        LL_DMA_EnableStream(LWESP_USART_DMA, LWESP_USART_DMA_RX_STREAM);
#else
        LL_DMA_EnableChannel(LWESP_USART_DMA, LWESP_USART_DMA_RX_CH);
#endif /* defined(LWESP_USART_DMA_RX_STREAM) */
        LL_USART_Enable(LWESP_USART);
    } else {
        osDelay(10);
        LL_USART_Disable(LWESP_USART);
        usart_init.BaudRate = baudrate;
        LL_USART_Init(LWESP_USART, &usart_init);
        LL_USART_Enable(LWESP_USART);
    }

    /* Create mbox and start thread */
```

(continues on next page)

(continued from previous page)

```
    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 = 1536};
        usart_ll_thread_id = osThreadNew(usart_ll_thread, usart_ll_mbox_id, &attr);
    }
}

#if defined(LWESP_RESET_PIN)
/**
 * \brief           Hardware reset callback
 */
static uint8_t
prv_reset_device(uint8_t state) {
    if (state) { /* Activate reset line */
        LL_GPIO_ResetOutputPin(LWESP_RESET_PORT, LWESP_RESET_PIN);
    } else {
        LL_GPIO_SetOutputPin(LWESP_RESET_PORT, LWESP_RESET_PIN);
    }
    return 1;
}
#endif /* defined(LWESP_RESET_PIN) */

/**
 * \brief           Send data to ESP device
 * \param[in]       data: Pointer to data to send
 * \param[in]       len: Number of bytes to send
 * \return          Number of bytes sent
 */
static size_t
prv_send_data(const void* data, size_t len) {
    const uint8_t* d = data;

    for (size_t i = 0; i < len; ++i, ++d) {
        LL_USART_TransmitData8(LWESP_USART, *d);
        while (!LL_USART_IsActiveFlag_TXE(LWESP_USART)) {}
    }
    return len;
}

/**
 * \brief           Callback function called from initialization process
 */
lwespr_t
lwesp_ll_init(lwesp_ll_t* ll) {
#if !LWESP_CFG_MEM_CUSTOM
    static uint8_t memory[LWESP_MEM_SIZE];
    const lwesp_mem_region_t mem_regions[] = {{memory, sizeof(memory)}};

    if (!initialized) {
        lwesp_mem_assignmemory(mem_regions, LWESP_ARRAYSIZE(mem_regions)); /* Assign␣
```

(continues on next page)

(continued from previous page)

```
↪memory for allocations */
    }
#endif /* !LWESP_CFG_MEM_CUSTOM */

    if (!initialized) {
        ll->send_fn = prv_send_data; /* Set callback function to send data */
#if defined(LWESP_RESET_PIN)
        ll->reset_fn = prv_reset_device; /* Set callback for hardware reset */
#endif                                   /* defined(LWESP_RESET_PIN) */
    }

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

/**
 * \brief           Callback function to de-init low-level communication part
 */
lwespr_t
lwesp_ll_deinit(lwesp_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;
    LWESP_UNUSED(ll);
    return lwespOK;
}

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

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

(continues on next page)

(continued from previous page)

```

/**
 * \brief           UART DMA stream/channel handler
 */
void
LWESP_USART_DMA_RX_IRQHANDLER(void) {
    LWESP_USART_DMA_RX_CLEAR_TC;
    LWESP_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            lwesp_sys_port.h
 * \brief           WIN32 based system file implementation
 */

/*
 * Copyright (c) 2024 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 LwESP - Lightweight ESP-AT parser library.
 *
```

(continues on next page)

(continued from previous page)

```
 * Author:          Tilen MAJERLE <tilen@majerle.eu>
 * Version:         v1.1.2-dev
 */
#ifndef LWESP_SYSTEM_PORT_HDR_H
#define LWESP_SYSTEM_PORT_HDR_H

#include <stdint.h>
#include <stdlib.h>
#include "lwesp/lwesp_opt.h"
#include "windows.h"

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

#if LWESP_CFG_OS && !__DOXYGEN__

typedef HANDLE lwesp_sys_mutex_t;
typedef HANDLE lwesp_sys_sem_t;
typedef HANDLE lwesp_sys_mbox_t;
typedef HANDLE lwesp_sys_thread_t;
typedef int lwesp_sys_thread_prio_t;

#define LWESP_SYS_MBOX_NULL   ((HANDLE)0)
#define LWESP_SYS_SEM_NULL    ((HANDLE)0)
#define LWESP_SYS_MUTEX_NULL  ((HANDLE)0)
#define LWESP_SYS_TIMEOUT     (INFINITE)
#define LWESP_SYS_THREAD_PRIO (0)
#define LWESP_SYS_THREAD_SS   (1024)

#endif /* LWESP_CFG_OS && !__DOXYGEN__ */

#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif /* LWESP_SYSTEM_PORT_HDR_H */
```

Listing 11: Actual implementation of system functions for WIN32

```
/**
 * \file            lwesp_sys_win32.c
 * \brief           System dependant functions for WIN32
 */

/*
 * Copyright (c) 2024 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,
```

(continues on next page)

(continued from previous page)

```
 * 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 LwESP - Lightweight ESP-AT parser library.
 *
 * Author:          Tilen MAJERLE <tilen@majerle.eu>
 * Version:         v1.1.2-dev
 */
#include <stdlib.h>
#include <string.h>
#include "lwesp/lwesp_private.h"
#include "system/lwesp_sys.h"
#include "windows.h"

#if !__DOXYGEN__

/**
 * \brief           Custom message queue implementation for WIN32
 */
typedef struct {
    lwesp_sys_sem_t sem_not_empty; /*!< Semaphore indicates not empty */
    lwesp_sys_sem_t sem_not_full;  /*!< Semaphore indicates not full */
    lwesp_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 lwesp_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) {
```

(continues on next page)

(continued from previous page)

```
        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);
}

uint8_t
lwesp_sys_init(void) {
    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&sys_start_time);

    lwesp_sys_mutex_create(&sys_mutex);
    return 1;
}

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

#if LWESP_CFG_OS
uint8_t
lwesp_sys_protect(void) {
    lwesp_sys_mutex_lock(&sys_mutex);
    return 1;
}

```

(continues on next page)

(continued from previous page)

```
uint8_t
lwesp_sys_unprotect(void) {
    lwesp_sys_mutex_unlock(&sys_mutex);
    return 1;
}

uint8_t
lwesp_sys_mutex_create(lwesp_sys_mutex_t* p) {
    *p = CreateMutex(NULL, FALSE, NULL);
    return *p != NULL;
}

uint8_t
lwesp_sys_mutex_delete(lwesp_sys_mutex_t* p) {
    return CloseHandle(*p);
}

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

uint8_t
lwesp_sys_mutex_unlock(lwesp_sys_mutex_t* p) {
    return ReleaseMutex(*p);
}

uint8_t
lwesp_sys_mutex_isvalid(lwesp_sys_mutex_t* p) {
    return p != NULL && *p != NULL;
}

uint8_t
lwesp_sys_mutex_invalid(lwesp_sys_mutex_t* p) {
    *p = LWESP_SYS_MUTEX_NULL;
    return 1;
}

uint8_t
lwesp_sys_sem_create(lwesp_sys_sem_t* p, uint8_t cnt) {
    HANDLE h;
    h = CreateSemaphore(NULL, !!cnt, 1, NULL);
    *p = h;
    return *p != NULL;
}

uint8_t
```

(continues on next page)

(continued from previous page)

```
lwesp_sys_sem_delete(lwesp_sys_sem_t* p) {
    return CloseHandle(*p);
}

uint32_t
lwesp_sys_sem_wait(lwesp_sys_sem_t* p, uint32_t timeout) {
    DWORD ret;

    if (timeout == 0) {
        ret = WaitForSingleObject(*p, INFINITE);
        return 1;
    } else {
        ret = WaitForSingleObject(*p, timeout);
        if (ret == WAIT_OBJECT_0) {
            return 1;
        } else {
            return LWESP_SYS_TIMEOUT;
        }
    }
}

uint8_t
lwesp_sys_sem_release(lwesp_sys_sem_t* p) {
    return ReleaseSemaphore(*p, 1, NULL);
}

uint8_t
lwesp_sys_sem_isvalid(lwesp_sys_sem_t* p) {
    return p != NULL && *p != NULL;
}

uint8_t
lwesp_sys_sem_invalid(lwesp_sys_sem_t* p) {
    *p = LWESP_SYS_SEM_NULL;
    return 1;
}

uint8_t
lwesp_sys_mbox_create(lwesp_sys_mbox_t* b, size_t size) {
    win32_mbox_t* mbox;

    *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 */
        lwesp_sys_sem_create(&mbox->sem, 1);
        lwesp_sys_sem_create(&mbox->sem_not_empty, 0);
        lwesp_sys_sem_create(&mbox->sem_not_full, 0);
        *b = mbox;
```

(continues on next page)

(continued from previous page)

```
    }
    return *b != NULL;
}

uint8_t
lwesp_sys_mbox_delete(lwesp_sys_mbox_t* b) {
    win32_mbox_t* mbox = *b;
    lwesp_sys_sem_delete(&mbox->sem);
    lwesp_sys_sem_delete(&mbox->sem_not_full);
    lwesp_sys_sem_delete(&mbox->sem_not_empty);
    free(mbox);
    return 1;
}

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

    lwesp_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)) {
        lwesp_sys_sem_release(&mbox->sem);          /* Release semaphore */
        lwesp_sys_sem_wait(&mbox->sem_not_full, 0); /* Wait for semaphore indicating not␣
↪full */
        lwesp_sys_sem_wait(&mbox->sem, 0);          /* Wait availability again */
    }
    mbox->entries[mbox->in] = m;
    if (++mbox->in >= mbox->size) {
        mbox->in = 0;
    }
    lwesp_sys_sem_release(&mbox->sem_not_empty); /* Signal non-empty state */
    lwesp_sys_sem_release(&mbox->sem);           /* Release access for other threads */
    return osKernelSysTick() - time;
}

uint32_t
lwesp_sys_mbox_get(lwesp_sys_mbox_t* b, void** m, uint32_t timeout) {
    win32_mbox_t* mbox = *b;
    uint32_t time;

    time = osKernelSysTick();

    /* Get exclusive access to message queue */
    if (lwesp_sys_sem_wait(&mbox->sem, timeout) == LWESP_SYS_TIMEOUT) {
        return LWESP_SYS_TIMEOUT;
    }
```

(continues on next page)

(continued from previous page)

```
    while (mbox_is_empty(mbox)) {
        lwesp_sys_sem_release(&mbox->sem);
        if (lwesp_sys_sem_wait(&mbox->sem_not_empty, timeout) == LWESP_SYS_TIMEOUT) {
            return LWESP_SYS_TIMEOUT;
        }
        lwesp_sys_sem_wait(&mbox->sem, timeout);
    }
    *m = mbox->entries[mbox->out];
    if (++mbox->out >= mbox->size) {
        mbox->out = 0;
    }
    lwesp_sys_sem_release(&mbox->sem_not_full);
    lwesp_sys_sem_release(&mbox->sem);

    return osKernelSysTick() - time;
}

uint8_t
lwesp_sys_mbox_putnow(lwesp_sys_mbox_t* b, void* m) {
    win32_mbox_t* mbox = *b;

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

uint8_t
lwesp_sys_mbox_getnow(lwesp_sys_mbox_t* b, void** m) {
    win32_mbox_t* mbox = *b;

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

    *m = mbox->entries[mbox->out];
    if (++mbox->out >= mbox->size) {
        mbox->out = 0;
    }
    lwesp_sys_sem_release(&mbox->sem_not_full); /* Queue not full anymore */
```

(continues on next page)

(continued from previous page)

```
323     lwesp_sys_sem_release(&mbox->sem);          /* Release semaphore */
324     return 1;
325 }
326 
327 uint8_t
328 lwesp_sys_mbox_isvalid(lwesp_sys_mbox_t* b) {
329     return b != NULL && *b != NULL;
330 }
331 
332 uint8_t
333 lwesp_sys_mbox_invalid(lwesp_sys_mbox_t* b) {
334     *b = LWESP_SYS_MBOX_NULL;
335     return 1;
336 }
337 
338 uint8_t
339 lwesp_sys_thread_create(lwesp_sys_thread_t* t, const char* name, lwesp_sys_thread_fn␣
    ↪thread_func, void* const arg,
340                         size_t stack_size, lwesp_sys_thread_prio_t prio) {
341     HANDLE h;
342     DWORD id;
343 
344     LWESP_UNUSED(name);
345     LWESP_UNUSED(stack_size);
346     LWESP_UNUSED(prio);
347     h = CreateThread(0, 0, (LPTHREAD_START_ROUTINE)thread_func, arg, 0, &id);
348     if (t != NULL) {
349         *t = h;
350     }
351     return h != NULL;
352 }
353 
354 uint8_t
355 lwesp_sys_thread_terminate(lwesp_sys_thread_t* t) {
356     if (t == NULL) { /* Shall we terminate ourself? */
357         ExitThread(0);
358     } else {
359         /* We have known thread, find handle by looking at ID */
360         TerminateThread(*t, 0);
361     }
362     return 1;
363 }
364 
365 uint8_t
366 lwesp_sys_thread_yield(void) {
367     /* Not implemented */
368     return 1;
369 }
370 
371 #endif /* LWESP_CFG_OS */
372 #endif /* !__DOXYGEN__ */
```

Example: System functions for CMSIS-OS

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

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

/*
 * Copyright (c) 2024 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 LwESP - Lightweight ESP-AT parser library.
 *
 * Author:          Tilen MAJERLE <tilen@majerle.eu>
 * Version:         v1.1.2-dev
 */
#ifndef LWESP_SYSTEM_PORT_HDR_H
#define LWESP_SYSTEM_PORT_HDR_H

#include <stdint.h>
#include <stdlib.h>
#include "cmsis_os.h"
#include "lwesp/lwesp_opt.h"

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

#if LWESP_CFG_OS && !__DOXYGEN__

typedef osMutexId_t lwesp_sys_mutex_t;
```

(continues on next page)

(continued from previous page)

```
typedef osSemaphoreId_t lwesp_sys_sem_t;
typedef osMessageQueueId_t lwesp_sys_mbox_t;
typedef osThreadId_t lwesp_sys_thread_t;
typedef osPriority_t lwesp_sys_thread_prio_t;

#define LWESP_SYS_MUTEX_NULL  ((lwesp_sys_mutex_t)0)
#define LWESP_SYS_SEM_NULL    ((lwesp_sys_sem_t)0)
#define LWESP_SYS_MBOX_NULL   ((lwesp_sys_mbox_t)0)
#define LWESP_SYS_TIMEOUT     ((uint32_t)osWaitForever)
#define LWESP_SYS_THREAD_PRIO (osPriorityNormal)
#define LWESP_SYS_THREAD_SS   (1536)

#endif /* LWESP_CFG_OS && !__DOXYGEN__ */

#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif /* LWESP_SYSTEM_PORT_HDR_H */
```

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

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

/*
 * Copyright (c) 2024 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 LwESP - Lightweight ESP-AT parser library.
```

(continues on next page)

(continued from previous page)

```
 *
 * Author:          Tilen MAJERLE <tilen@majerle.eu>
 * Version:         v1.1.2-dev
 */
#include "cmsis_os.h"
#include "system/lwesp_sys.h"

#if !__DOXYGEN__

static osMutexId_t sys_mutex;

uint8_t
lwesp_sys_init(void) {
    lwesp_sys_mutex_create(&sys_mutex);
    return 1;
}

uint32_t
lwesp_sys_now(void) {
    return osKernelGetTickCount();
}

uint8_t
lwesp_sys_protect(void) {
    lwesp_sys_mutex_lock(&sys_mutex);
    return 1;
}

uint8_t
lwesp_sys_unprotect(void) {
    lwesp_sys_mutex_unlock(&sys_mutex);
    return 1;
}

uint8_t
lwesp_sys_mutex_create(lwesp_sys_mutex_t* p) {
    const osMutexAttr_t attr = {
        .attr_bits = osMutexRecursive,
        .name = "lwesp_mutex",
    };
    return (*p = osMutexNew(&attr)) != NULL;
}

uint8_t
lwesp_sys_mutex_delete(lwesp_sys_mutex_t* p) {
    return osMutexDelete(*p) == osOK;
}

uint8_t
lwesp_sys_mutex_lock(lwesp_sys_mutex_t* p) {
    return osMutexAcquire(*p, osWaitForever) == osOK;
}
```

(continues on next page)

(continued from previous page)

```

uint8_t
lwesp_sys_mutex_unlock(lwesp_sys_mutex_t* p) {
    return osMutexRelease(*p) == osOK;
}

uint8_t
lwesp_sys_mutex_isvalid(lwesp_sys_mutex_t* p) {
    return p != NULL && *p != NULL;
}

uint8_t
lwesp_sys_mutex_invalid(lwesp_sys_mutex_t* p) {
    *p = LWESP_SYS_MUTEX_NULL;
    return 1;
}

uint8_t
lwesp_sys_sem_create(lwesp_sys_sem_t* p, uint8_t cnt) {
    const osSemaphoreAttr_t attr = {
        .name = "lwesp_sem",
    };
    return (*p = osSemaphoreNew(1, cnt > 0 ? 1 : 0, &attr)) != NULL;
}

uint8_t
lwesp_sys_sem_delete(lwesp_sys_sem_t* p) {
    return osSemaphoreDelete(*p) == osOK;
}

uint32_t
lwesp_sys_sem_wait(lwesp_sys_sem_t* p, uint32_t timeout) {
    uint32_t tick = osKernelSysTick();
    return (osSemaphoreAcquire(*p, timeout == 0 ? osWaitForever : timeout) == osOK) ?␣
↪(osKernelSysTick() - tick)
                                                                                    :␣
↪LWESP_SYS_TIMEOUT;
}

uint8_t
lwesp_sys_sem_release(lwesp_sys_sem_t* p) {
    return osSemaphoreRelease(*p) == osOK;
}

uint8_t
lwesp_sys_sem_isvalid(lwesp_sys_sem_t* p) {
    return p != NULL && *p != NULL;
}

uint8_t
lwesp_sys_sem_invalid(lwesp_sys_sem_t* p) {
    *p = LWESP_SYS_SEM_NULL;
```

(continues on next page)

(continued from previous page)

```
    return 1;
}

uint8_t
lwesp_sys_mbox_create(lwesp_sys_mbox_t* b, size_t size) {
    const osMessageQueueAttr_t attr = {
        .name = "lwesp_mbox",
    };
    return (*b = osMessageQueueNew(size, sizeof(void*), &attr)) != NULL;
}

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

uint32_t
lwesp_sys_mbox_put(lwesp_sys_mbox_t* b, void* m) {
    uint32_t tick = osKernelSysTick();
    return osMessageQueuePut(*b, &m, 0, osWaitForever) == osOK ? (osKernelSysTick() -␣
↪tick) : LWESP_SYS_TIMEOUT;
}

uint32_t
lwesp_sys_mbox_get(lwesp_sys_mbox_t* b, void** m, uint32_t timeout) {
    uint32_t tick = osKernelSysTick();
    return (osMessageQueueGet(*b, m, NULL, timeout == 0 ? osWaitForever : timeout) ==␣
↪osOK) ? (osKernelSysTick() - tick)
                                                                                    ␣
↪    : LWESP_SYS_TIMEOUT;
}

uint8_t
lwesp_sys_mbox_putnow(lwesp_sys_mbox_t* b, void* m) {
    return osMessageQueuePut(*b, &m, 0, 0) == osOK;
}

uint8_t
lwesp_sys_mbox_getnow(lwesp_sys_mbox_t* b, void** m) {
    return osMessageQueueGet(*b, m, NULL, 0) == osOK;
}

uint8_t
lwesp_sys_mbox_isvalid(lwesp_sys_mbox_t* b) {
    return b != NULL && *b != NULL;
}

uint8_t
lwesp_sys_mbox_invalid(lwesp_sys_mbox_t* b) {
```

(continues on next page)

(continued from previous page)

```
    *b = LWESP_SYS_MBOX_NULL;
    return 1;
}

uint8_t
lwesp_sys_thread_create(lwesp_sys_thread_t* t, const char* name, lwesp_sys_thread_fn␣
↪thread_func, void* const arg,
                        size_t stack_size, lwesp_sys_thread_prio_t prio) {
    lwesp_sys_thread_t id;
    const osThreadAttr_t thread_attr = {.name = (char*)name,
                                        .priority = (osPriority)prio,
                                        .stack_size = stack_size > 0 ? stack_size :␣
↪LWESP_SYS_THREAD_SS};

    id = osThreadNew(thread_func, arg, &thread_attr);
    if (t != NULL) {
        *t = id;
    }
    return id != NULL;
}

uint8_t
lwesp_sys_thread_terminate(lwesp_sys_thread_t* t) {
    if (t != NULL) {
        osThreadTerminate(*t);
    } else {
        osThreadExit();
    }
    return 1;
}

uint8_t
lwesp_sys_thread_yield(void) {
    osThreadYield();
    return 1;
}

#endif /* !__DOXYGEN__ */
```

5.2.7 TCP connection SSL support

Warning: SSL support is currently in experimental mode. API changes may occur in the future.

ESP-AT binary, running on Espressif chips, supports SLL connection types. Such connections, to work properly, require client or server certificates to be loaded to Espressif device.

With the recent update, *July 29th, 2023*, library has been updated to support AT commands for flash and MFG operations, allowing host microcontroller to load required certificates to the Espressif device.

Note: Minimum required ESP-AT library running on ESP device is now `v3.2.0`, which supports new *AT+SYSMFG* command, that is required to load custom data to the device.

Note: SSL connections mentioned on this page are secure from Espressif device towards network. Data between host MCU and Espressif MCU is not protected and may be exposed to an attacker

Prepare the certificate

Assuming we would like to establish connection to another server with secure SSL connection, ESP device shall have up to 3 certificates loaded in its own system flash. These are:

- *client_ca* - Client root CA certificate - client uses this certificate to verify server. Example
- *client_cert* - Client certificate. Example
- *client_key* - Client private key. Example

ESP-AT website includes some test certificates, that could be used for test purposes: Description of all slots

LwESP repository contains aforementioned certificates in the *certificates* folder. There are *2* files for each certificate:

- Original *.crt* or *.key* file
- Original *.crt* or *.key* file converted to *.hex* array for easier include in the C project.

Load to ESP device

Loading can be done as part of custom AT firmware build, or by using AT commands. LwESP library has the support for system flash and manufacturing NVS data operation, that is required for certificate load.

All combined, steps to establish SSL connection is to:

- Have certificates loaded to ESP-AT device with Espressif format
- Have configured connections to use your certificates, if *SSL* type is used on them
- Have valid time in ESP device. *SNTP* module can help with that

Example

Below is the up-to-date netconn API example using SSL connection. Example file is located in `snippets/netconn_client_ssl.c`

Listing 14: Netconn example with SSL

```
/*
 * Netconn client demonstrates how to connect as a client to server
 * using sequential API from separate thread.
 *
 * it does not use callbacks to obtain connection status.
 *
 * Demo connects to NETCONN_HOST at NETCONN_PORT and sends GET request header,
 * then waits for respond and expects server to close the connection accordingly.
```

(continues on next page)

(continued from previous page)

```
 */
#include "lwesp/lwesp.h"
#include "lwesp/lwesp_netconn.h"
#include "netconn_client.h"

/* Certificates, ready to be loaded to the flash */
static uint8_t client_ca[] = {
#include "../certificates/client_ca_generated_atpki.hex"
};
static uint8_t client_cert[] = {
#include "../certificates/client_cert_generated_atpki.hex"
};
static uint8_t client_key[] = {
#include "../certificates/client_key_generated_atpki.hex"
};

/**
 * \brief           Host and port settings
 */
#define NETCONN_HOST "example.com"
#define NETCONN_PORT 443

/**
 * \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_ssl_thread(void const* arg) {
    lwespr_t res;
    lwesp_pbuf_p pbuf;
    lwesp_netconn_p client;
    lwesp_sys_sem_t* sem = (void*)arg;

    /* Make sure we are connected to access point first */
    while (!lwesp_sta_has_ip()) {
        lwesp_delay(1000);
    }

    /*
     * First create a new instance of netconn
     * connection and initialize system message boxes
     * to accept received packet buffers
     */
```

(continues on next page)

(continued from previous page)

```
    client = lwesp_netconn_new(LWESP_NETCONN_TYPE_SSL);
    if (client != NULL) {
        struct tm dt;
        uint32_t sntp_interval = 0;
        uint8_t sntp_en = 0;

        /* Write data to coresponding manuf NVS */
        res = lwesp_mfg_write(LWESP_MFG_NAMESPACE_CLIENT_CA, "client_ca.0", LWESP_MFG_
↪VALTYPE_BLOB, client_ca,
                              sizeof(client_ca), NULL, NULL, 1);
        res = lwesp_mfg_write(LWESP_MFG_NAMESPACE_CLIENT_CERT, "client_cert.0", LWESP_
↪MFG_VALTYPE_BLOB, client_cert,
                              sizeof(client_cert), NULL, NULL, 1);
        res = lwesp_mfg_write(LWESP_MFG_NAMESPACE_CLIENT_KEY, "client_key.0", LWESP_MFG_
↪VALTYPE_BLOB, client_key,
                              sizeof(client_key), NULL, NULL, 1);

        /* Configure SSL for all connections */
        for (size_t i = 0; i < LWESP_CFG_MAX_CONNS; ++i) {
            lwesp_conn_ssl_set_config(i, 1, 0, 0, NULL, NULL, 1);
        }

        /* Ensure SNTP is enabled, time is required for SSL */
        if (lwesp_sntp_get_config(&sntp_en, NULL, NULL, NULL, NULL, NULL, NULL, 1) ==␣
↪lwespOK) {
            if (!sntp_en) {
                lwesp_sntp_set_config(1, 2, NULL, NULL, NULL, NULL, NULL, 1);
            }
            lwesp_sntp_get_interval(&sntp_interval, NULL, NULL, 1);
            printf("SNTP interval: %u seconds\r\n", (unsigned)sntp_interval);
            do {
                lwesp_sntp_gettime(&dt, NULL, NULL, 1);
                if (dt.tm_year > 100) {
                    break;
                }
                lwesp_delay(1000);
            } while (1);
        }

        /*
         * 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 = lwesp_netconn_connect(client, NETCONN_HOST, NETCONN_PORT);
        if (res == lwespOK) { /* Are we successfully connected? */
            printf("Connected to " NETCONN_HOST "\r\n");
            res = lwesp_netconn_write(client, request_header, sizeof(request_header) -␣
↪1); /* Send data to server */
            if (res == lwespOK) {
```

(continues on next page)

(continued from previous page)

```
            res = lwesp_netconn_flush(client); /* Flush data to output */
        }
        if (res == lwespOK) { /* Were data sent? */
            printf("Data were successfully sent to server\r\n");

            /*
             * 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 = lwesp_netconn_receive(client, &pbuf);
                if (res
                    == lwespCLOSED) { /* 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 == lwespTIMEOUT) {
                    printf("Netconn timeout while receiving data. You may try multiple readings before deciding to "
                           "close manually\r\n");
                }

                if (res == lwespOK && 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, or memory leaks will appear
                     */
                    printf("Received new data packet of %d bytes\r\n", (int)lwesp_pbuf_length(pbuf, 1));
                    lwesp_pbuf_free_s(&pbuf); /* Free the memory after usage */
                }
            } while (1);
        } else {
            printf("Error writing data to remote host!\r\n");
        }

        /*
```

(continues on next page)

(continued from previous page)

```
155             * Check if connection was closed by remote server
156             * and in case it wasn't, close it manually
157             */
158            if (res != lwespCLOSED) {
159                lwesp_netconn_close(client);
160            }
161        } else {
162            printf("Cannot connect to remote host %s:%d!\r\n", NETCONN_HOST, NETCONN_
    ↪PORT);
163        }
164        lwesp_netconn_delete(client); /* Delete netconn structure */
165    }
166
167    printf("Terminating thread\r\n");
168    if (lwesp_sys_sem_isvalid(sem)) {
169        lwesp_sys_sem_release(sem);
170    }
171    lwesp_sys_thread_terminate(NULL); /* Terminate current thread */
172 }
```

5.3 API reference

List of all the modules:

5.3.1 LwESP

Access point

group **LWESP_AP**

Access point.

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

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

Functions

lwespr_t **lwesp_ap_getip**(*lwesp_ip_t* *ip, *lwesp_ip_t* *gw, *lwesp_ip_t* *nm, const *lwesp_api_cmd_evt_fn* evt_fn, void *const evt_arg, const uint32_t blocking)

Get IP of access point.

Parameters

- **ip** – **[out]** Pointer to variable to write IP address
- **gw** – **[out]** Pointer to variable to write gateway address
- **nm** – **[out]** Pointer to variable to write netmask address
- **evt_fn** – **[in]** Callback function called when command has finished. Set to `NULL` when not used

- **evt_arg** – **[in]** Custom argument for event callback function
- **blocking** – **[in]** Status whether command should be blocking or not

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_ap_setip**(const *lwesp_ip_t* *ip, const *lwesp_ip_t* *gw, const *lwesp_ip_t* *nm, const *lwesp_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.

Parameters

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

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_ap_getmac**(*lwesp_mac_t* *mac, const *lwesp_api_cmd_evt_fn* evt_fn, void *const evt_arg, const uint32_t blocking)

Get MAC of access point.

Parameters

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

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_ap_setmac**(const *lwesp_mac_t* *mac, const *lwesp_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.

Parameters

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

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_ap_get_config**(*lwesp_ap_conf_t* *ap_conf, const *lwesp_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 *lwesp_set_wifi_mode* for more information

---

Parameters

- **ap_conf** – **[out]** soft access point configuration
- **evt_fn** – **[in]** Callback function called when command has finished. Set to `NULL` when not used
- **evt_arg** – **[in]** Custom argument for event callback function
- **blocking** – **[in]** Status whether command should be blocking or not

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_ap_set_config**(const char *ssid, const char *pwd, uint8_t ch, *lwesp_ecn_t* ecn, uint8_t max_sta, uint8_t hid, const *lwesp_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 *lwesp_set_wifi_mode* for more information

---

Parameters

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

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_ap_list_sta**(*lwesp_sta_t* *sta, size_t stal, size_t *staf, const *lwesp_api_cmd_evt_fn* evt_fn, void *const evt_arg, const uint32_t blocking)

List stations connected to access point.

Parameters

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

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_ap_disconn_sta**(const *lwesp_mac_t* *mac, const *lwesp_api_cmd_evt_fn* evt_fn, void *const evt_arg, const uint32_t blocking)

Disconnects connected station from SoftAP access point.

Parameters

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

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

struct **lwesp_ap_t**

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

Public Members

lwesp_ecn_t **ecn**

Encryption mode

char **ssid**[LWESP_CFG_MAX_SSID_LENGTH]

Access point name

int16_t **rssi**

Received signal strength indicator

lwesp_mac_t **mac**

MAC physical address

uint8_t **ch**

WiFi channel used on access point

uint8_t **scan_type**

Scan type, 0 = active, 1 = passive

uint16_t **scan_time_min**

Minimum active scan time per channel in units of milliseconds

uint16_t **scan_time_max**

Maximum active scan time per channel in units of milliseconds

int16_t **freq_offset**

Frequency offset

int16_t **freq_cal**

Frequency calibration

lwesp_ap_cipher_t **pairwise_cipher**

Pairwise cipher mode

lwesp_ap_cipher_t **group_cipher**

Group cipher mode

uint8_t **bgn**

Information about 802.11[b|g|n] support

uint8_t **wps**

Status if WPS function is supported

struct **lwesp_sta_info_ap_t**

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

Public Members

char **ssid**[LWESP_CFG_MAX_SSID_LENGTH]

Access point name

int16_t **rssi**

RSSI

lwesp_mac_t **mac**

MAC address

uint8_t **ch**

Channel information

struct **lwesp_ap_conf_t**

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

Public Members

char **ssid**[LWESP_CFG_MAX_SSID_LENGTH]

Access point name

char **pwd**[LWESP_CFG_MAX_PWD_LENGTH]

Access point password/passphrase

uint8_t **ch**

WiFi channel used on access point

lwesp_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

Bluetooth Low Energy

group **LWESP_BLE**

Bluetooth Low Energy.

Functions to manage BLE protocol on some of ESP devices (if hardware supports it)

LWESP_CFG_BLE must be enabled to use this feature.

Bluetooth Classic

group **LWESP_BT**

Bluetooth Classic.

Functions to manage Bluetooth Classic protocol on some of ESP devices (if hardware supports it)

LWESP_CFG_BT must be enabled to use this feature.

Ring buffer

group **LWESP_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 **lwesp_buff_init**(*lwesp_buff_t* *buff, size_t size)

Initialize buffer.

Parameters

- **buff** – **[in]** Pointer to buffer structure
- **size** – **[in]** Size of buffer in units of bytes

Returns

1 on success, 0 otherwise

void **lwesp_buff_free**(*lwesp_buff_t* *buff)

Free dynamic allocation if used on memory.

Parameters

buff – **[in]** Pointer to buffer structure

void **lwesp_buff_reset**(*lwesp_buff_t* *buff)

Resets buffer to default values. Buffer size is not modified.

Parameters

buff – **[in]** Buffer handle

size_t **lwesp_buff_write**(*lwesp_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.

Parameters

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

Returns

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

size_t **lwesp_buff_read**(*lwesp_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.

Parameters

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

Returns

Number of bytes read and copied to data array

size_t **lwesp_buff_peek**(*lwesp_buff_t* *buff, size_t skip_count, void *data, size_t btp)

Read from buffer without changing read pointer (peek only)

Parameters

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

Returns

Number of bytes peeked and written to output array

size_t **lwesp_buff_get_free**(*lwesp_buff_t* *buff)

Get number of bytes in buffer available to write.

Parameters

buff – **[in]** Buffer handle

Returns

Number of free bytes in memory

size_t **lwesp_buff_get_full**(*lwesp_buff_t* *buff)

Get number of bytes in buffer available to read.

Parameters

buff – **[in]** Buffer handle

Returns

Number of bytes ready to be read

void ***lwesp_buff_get_linear_block_read_address**(*lwesp_buff_t* *buff)

Get linear address for buffer for fast read.

Parameters

buff – **[in]** Buffer handle

Returns

Linear buffer start address

size_t **lwesp_buff_get_linear_block_read_length**(*lwesp_buff_t* *buff)

Get length of linear block address before it overflows for read operation.

Parameters

buff – **[in]** Buffer handle

Returns
Linear buffer size in units of bytes for read operation

size_t **lwesp_buff_skip**(*lwesp_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

Parameters

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

Returns
Number of bytes skipped

void ***lwesp_buff_get_linear_block_write_address**(*lwesp_buff_t* *buff)

Get linear address for buffer for fast read.

Parameters
buff – **[in]** Buffer handle

Returns
Linear buffer start address

size_t **lwesp_buff_get_linear_block_write_length**(*lwesp_buff_t* *buff)

Get length of linear block address before it overflows for write operation.

Parameters
buff – **[in]** Buffer handle

Returns
Linear buffer size in units of bytes for write operation

size_t **lwesp_buff_advance**(*lwesp_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

Parameters

- **buff** – **[in]** Buffer handle
- **len** – **[in]** Number of bytes to advance

Returns
Number of bytes advanced for write operation

struct **lwesp_buff_t**

#include <lwesp_types.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 15: Client connection minimum example

```
1  #include "client.h"
2  #include "lwesp/lwesp.h"
3
4  /* Host parameter */
5  #define CONN_HOST           "example.com"
6  #define CONN_PORT           80
7
8  static lwespr_t   conn_callback_func(lwesp_evt_t* evt);
9
```

(continues on next page)

(continued from previous page)

```
/**
 * \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) {
    lwespr_t res;

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

    /* Start 2 more */
    lwesp_conn_start(NULL, LWESP_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 LWESP_EVT_CONN_ERROR event should be triggered
     * in callback function processing
     */
    lwesp_conn_start(NULL, LWESP_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 lwespOK on success, member of \ref lwespr_t otherwise
 */
static lwespr_t
conn_callback_func(lwesp_evt_t* evt) {
    lwesp_conn_p conn;
    lwespr_t res;
    uint8_t conn_num;

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

(continues on next page)

(continued from previous page)

```
    }
    conn_num = lwesp_conn_getnum(conn);           /* Get connection number for identification */
    switch (lwesp_evt_get_type(evt)) {
        case LWESP_EVT_CONN_ACTIVE: {             /* Connection just active */
            printf("Connection %d active!\r\n", (int)conn_num);
            res = lwesp_conn_send(conn, req_data, sizeof(req_data) - 1, NULL, 0); /* Start sending data in non-blocking mode */
            if (res == lwespOK) {
                printf("Sending request data to server...\r\n");
            } else {
                printf("Cannot send request data to server. Closing connection manually..\r\n");
                lwesp_conn_close(conn, 0);        /* Close the connection */
            }
            break;
        }
        case LWESP_EVT_CONN_CLOSE: {              /* Connection closed */
            if (lwesp_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 LWESP_EVT_CONN_SEND: {               /* Data send event */
            lwespr_t res = lwesp_evt_conn_send_get_result(evt);
            if (res == lwespOK) {
                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 LWESP_EVT_CONN_RECV: {               /* Data received from remote side */
            lwesp_pbuf_p pbuf = lwesp_evt_conn_recv_get_buff(evt);
            lwesp_conn_recved(conn, pbuf);        /* Notify stack about received pbuf */
            printf("Received %d bytes on connection %d..\r\n", (int)lwesp_pbuf_length(pbuf, 1), (int)conn_num);
            break;
        }
        case LWESP_EVT_CONN_ERROR: {              /* Error connecting to server */
            const char* host = lwesp_evt_conn_error_get_host(evt);
            lwesp_port_t port = lwesp_evt_conn_error_get_port(evt);
            printf("Error connecting to %s:%d\r\n", host, (int)port);
            break;
        }
        default:
            break;
    }
    return lwespOK;
```

(continues on next page)

(continued from previous page)

```
}
```

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 *lwesp_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 16: Write data to connection output buffer

```
size_t rem_len;
lwesp_conn_p conn;
lwespr_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 = lwesp_conn_write(conn, "My string", 9, 0, &rem_len);
if (rem_len == 0) {
    printf("No more memory available for next write!\r\n");
}
res = lwesp_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
 *
```

(continues on next page)

(continued from previous page)

```
 * It will send out together 20 bytes
 */
lwesp_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 *lwesp_conn_send()* or *lwesp_conn_sendto()* functions.

group **LWESP_CONN**

Connection API functions.

Typedefs

typedef struct lwesp_conn ***lwesp_conn_p**

Pointer to *lwesp_conn_t* structure.

Enums

enum **lwesp_conn_type_t**

List of possible connection types.

Values:

enumerator **LWESP_CONN_TYPE_TCP**

Connection type is TCP

enumerator **LWESP_CONN_TYPE_UDP**

Connection type is UDP

enumerator **LWESP_CONN_TYPE_SSL**

Connection type is SSL

enumerator **LWESP_CONN_TYPE_TCPV6**

Connection type is TCP over IPv6

enumerator **LWESP_CONN_TYPE_UDPV6**

Connection type is UDP over IPv6

enumerator **LWESP_CONN_TYPE_SSLV6**

Connection type is SSL over IPv6

Functions

lwespr_t **lwesp_conn_start**(*lwesp_conn_p* *conn, *lwesp_conn_type_t* type, const char *const remote_host, *lwesp_port_t* remote_port, void *const arg, *lwesp_evt_fn* conn_evt_fn, const uint32_t blocking)

Start a new connection of specific type.

Parameters

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

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_conn_startex**(*lwesp_conn_p* *conn, *lwesp_conn_start_t* *start_struct, void *const arg, *lwesp_evt_fn* conn_evt_fn, const uint32_t blocking)

Start a new connection of specific type in extended mode.

Parameters

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

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_conn_close**(*lwesp_conn_p* conn, const uint32_t blocking)

Close specific or all connections.

Parameters

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

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_conn_send**(*lwesp_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.

Parameters

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

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_conn_sendto**(*lwesp_conn_p* conn, const *lwesp_ip_t* *const ip, *lwesp_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)

Parameters

- **conn** – **[in]** Connection handle to send data
- **ip** – **[in]** Remote IP address for UDP connection
- **port** – **[in]** Remote port connection
- **data** – **[in]** Pointer to data to send
- **btw** – **[in]** Number of bytes to send
- **bw** – **[out]** Pointer to output variable to save number of sent data when successfully sent
- **blocking** – **[in]** Status whether command should be blocking or not

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_conn_set_arg**(*lwesp_conn_p* conn, void *const arg)

Set argument variable for connection.

See also:

lwesp_conn_get_arg

Parameters

- **conn** – **[in]** Connection handle to set argument
- **arg** – **[in]** Pointer to argument

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

void *lwesp_conn_get_arg(*lwesp_conn_p* conn)

Get user defined connection argument.

See also:

lwesp_conn_set_arg

Parameters
conn – **[in]** Connection handle to get argument

Returns
User argument

uint8_t lwesp_conn_is_client(*lwesp_conn_p* conn)

Check if connection type is client.

Parameters
conn – **[in]** Pointer to connection to check for status

Returns
1 on success, 0 otherwise

uint8_t lwesp_conn_is_server(*lwesp_conn_p* conn)

Check if connection type is server.

Parameters
conn – **[in]** Pointer to connection to check for status

Returns
1 on success, 0 otherwise

uint8_t lwesp_conn_is_active(*lwesp_conn_p* conn)

Check if connection is active.

Parameters
conn – **[in]** Pointer to connection to check for status

Returns
1 on success, 0 otherwise

uint8_t lwesp_conn_is_closed(*lwesp_conn_p* conn)

Check if connection is closed.

Parameters
conn – **[in]** Pointer to connection to check for status

Returns
1 on success, 0 otherwise

int8_t lwesp_conn_getnum(*lwesp_conn_p* conn)

Get the number from connection.

Parameters
conn – **[in]** Connection pointer

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

lwespr_t **lwesp_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

Parameters

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

Returns

lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_get_conns_status**(const uint32_t blocking)

Gets connections status.

Parameters

blocking – **[in]** Status whether command should be blocking or not

Returns

lwespOK on success, member of *lwespr_t* enumeration otherwise

lwesp_conn_p **lwesp_conn_get_from_evt**(*lwesp_evt_t* *evt)

Get connection from connection based event.

Parameters

evt – **[in]** Event which happened for connection

Returns

Connection pointer on success, `NULL` otherwise

lwespr_t **lwesp_conn_write**(*lwesp_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)

Parameters

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

Returns

lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_conn_recved**(*lwesp_conn_p* conn, *lwesp_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

Parameters

- **conn** – **[in]** Connection handle
- **pbuf** – **[in]** Packet buffer received on connection

Returns

lwespOK on success, member of *lwespr_t* enumeration otherwise

size_t **lwesp_conn_get_total_recved_count**(*lwesp_conn_p* conn)

Get total number of bytes ever received on connection and sent to user.

Parameters

conn – **[in]** Connection handle

Returns

Total number of received bytes on connection

uint8_t **lwesp_conn_get_remote_ip**(*lwesp_conn_p* conn, *lwesp_ip_t* *ip)

Get connection remote IP address.

Parameters

- **conn** – **[in]** Connection handle
- **ip** – **[out]** Pointer to IP output handle

Returns

1 on success, 0 otherwise

lwesp_port_t **lwesp_conn_get_remote_port**(*lwesp_conn_p* conn)

Get connection remote port number.

Parameters

conn – **[in]** Connection handle

Returns

Port number on success, 0 otherwise

lwesp_port_t **lwesp_conn_get_local_port**(*lwesp_conn_p* conn)

Get connection local port number.

Parameters

conn – **[in]** Connection handle

Returns

Port number on success, 0 otherwise

lwespr_t **lwesp_conn_ssl_set_config**(uint8_t link_id, uint8_t auth_mode, uint8_t pki_number, uint8_t ca_number, const *lwesp_api_cmd_evt_fn* evt_fn, void *const evt_arg, const uint32_t blocking)

Configure SSL parameters.

Parameters

- **link_id** – **[in]** ID of the connection (0~max), for multiple connections, if the value is max, it means all connections. By default, max is *LWESP_CFG_MAX_CONNS*.
- **auth_mode** – **[in]** 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
- **pki_number** – **[in]** The index of cert and private key, if only one cert and private key, the value should be `0`.
- **ca_number** – **[in]** The index of CA, if only one CA, the value should be 0.
- **evt_fn** – **[in]** Callback function called when command has finished. Set to `NULL` when not used
- **evt_arg** – **[in]** Custom argument for event callback function
- **blocking** – **[in]** Status whether command should be blocking or not

Returns

lwespOK on success, member of *lwespr_t* enumeration otherwise

struct **lwesp_conn_start_t**

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

Public Members

lwesp_conn_type_t **type**

Connection type

const char ***remote_host**

Host name or IP address in string format

lwesp_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 *lwesp_conn_start_t*::[anonymous]::[anonymous] **tcp_ssl**

TCP/SSL specific features

lwesp_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 *lwesp_conn_start_t*::[anonymous]::[anonymous] **udp**

UDP specific features

union *lwesp_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 *LWESP_CFG_DBG_OUT* configuration.

For successful debugging, application must:

- Enable global debugging by setting *LWESP_CFG_DBG* to *LWESP_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 *LWESP_DBG_ON*

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

An example code with config and latter usage:

Listing 17: Debug configuration setup

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

/* Enable global debug */
#define LWESP_CFG_DBG               LWESP_DBG_ON

/*
 * Enable debug types.
 * Application may use bitwise OR | to use multiple types:
 *    LWESP_DBG_TYPE_TRACE | LWESP_DBG_TYPE_STATE
 */
#define LWESP_CFG_DBG_TYPES_ON      LWESP_DBG_TYPE_TRACE

```

(continues on next page)

(continued from previous page)

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

Listing 18: Debug usage within middleware

```
#include "lwesp/lwesp_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.
 */
LWESP_DEBUGF(MY_DBG_MODULE | LWESP_DBG_TYPE_TRACE, "This is trace message on my program\
↪r\n");
LWESP_DEBUGF(MY_DBG_MODULE | LWESP_DBG_TYPE_STATE, "This is state message on my program\
↪r\n");
```

group **LWESP_DEBUG**

Debug support module to track library flow.

Unnamed Group

LWESP_DBG_ON

Indicates debug is enabled

LWESP_DBG_OFF

Indicates debug is disabled

Defines

LWESP_DEBUGF(c, fmt, ...)

Print message to the debug "window" if enabled.

Parameters

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

LWESP_DEBUGW(c, cond, fmt, ...)

Print message to the debug "window" if enabled when specific condition is met.

Parameters

- **c** – **[in]** Condition if debug of specific type is enabled
- **cond** – **[in]** Debug only if this condition is true
- **fmt** – **[in]** Formatted string for debug
- **...** – **[in]** Variable parameters for formatted string

Dynamic Host Configuration Protocol

group **LWESP_DHCP**

DHCP config.

Functions

lwespr_t **lwesp_dhcp_set_config**(uint8_t sta, uint8_t ap, uint8_t en, const *lwesp_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.

Parameters

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

Returns

lwespOK on success, member of *lwespr_t* enumeration otherwise

Domain Name System

group **LWESP_DNS**

Domain name server.

Functions

lwespr_t **lwesp_dns_gethostbyname**(const char *host, *lwesp_ip_t* *const ip, const *lwesp_api_cmd_evt_fn* evt_fn, void *const evt_arg, const uint32_t blocking)

Get IP address from host name.

Parameters

- **host** – **[in]** Pointer to host name to get IP for
- **ip** – **[out]** Pointer to *lwesp_ip_t* variable to save IP
- **evt_fn** – **[in]** Callback function called when command has finished. Set to NULL when not used
- **evt_arg** – **[in]** Custom argument for event callback function
- **blocking** – **[in]** Status whether command should be blocking or not

Returns

lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_dns_get_config**(*lwesp_ip_t* *s1, *lwesp_ip_t* *s2, const *lwesp_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.

Parameters

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

Returns

lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_dns_set_config**(uint8_t en, const char *s1, const char *s2, const *lwesp_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.

Parameters

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

Returns

lwespOK on success, member of *lwespr_t* enumeration otherwise

Event management

group **LWESP_EVT**

Event helper functions.

Reset detected

Event helper functions for *LWESP_EVT_RESET_DETECTED* event

uint8_t **lwesp_evt_reset_detected_is_forced**(*lwesp_evt_t* *cc)

Check if detected reset was forced by user.

Parameters
cc – **[in]** Event handle

Returns
1 if forced by user, 0 otherwise

Reset event

Event helper functions for *LWESP_EVT_RESET* event

lwespr_t **lwesp_evt_reset_get_result**(*lwesp_evt_t* *cc)

Get reset sequence operation status.

Parameters
cc – **[in]** Event data

Returns
Member of *lwespr_t* enumeration

Restore event

Event helper functions for *LWESP_EVT_RESTORE* event

lwespr_t **lwesp_evt_restore_get_result**(*lwesp_evt_t* *cc)

Get restore sequence operation status.

Parameters
cc – **[in]** Event data

Returns
Member of *lwespr_t* enumeration

Access point or station IP or MAC

Event helper functions for *LWESP_EVT_AP_IP_STA* event

lwesp_mac_t ***lwesp_evt_ap_ip_sta_get_mac**(*lwesp_evt_t* *cc)

Get MAC address from station.

Parameters
cc – **[in]** Event handle

Returns
MAC address

lwesp_ip_t *lwesp_evt_ap_ip_sta_get_ip(*lwesp_evt_t* *cc)

Get IP address from station.

Parameters
cc – **[in]** Event handle

Returns
IP address

Connected station to access point

Event helper functions for *LWESP_EVT_AP_CONNECTED_STA* event

lwesp_mac_t *lwesp_evt_ap_connected_sta_get_mac(*lwesp_evt_t* *cc)

Get MAC address from connected station.

Parameters
cc – **[in]** Event handle

Returns
MAC address

Disconnected station from access point

Event helper functions for *LWESP_EVT_AP_DISCONNECTED_STA* event

lwesp_mac_t *lwesp_evt_ap_disconnected_sta_get_mac(*lwesp_evt_t* *cc)

Get MAC address from disconnected station.

Parameters
cc – **[in]** Event handle

Returns
MAC address

Connection data received

Event helper functions for *LWESP_EVT_CONN_RECV* event

lwesp_pbuf_p lwesp_evt_conn_recv_get_buff(*lwesp_evt_t* *cc)

Get buffer from received data.

Parameters
cc – **[in]** Event handle

Returns
Buffer handle

lwesp_conn_p lwesp_evt_conn_recv_get_conn(*lwesp_evt_t* *cc)

Get connection handle for receive.

Parameters
cc – **[in]** Event handle

Returns
Connection handle

Connection data send

Event helper functions for *LWESP_EVT_CONN_SEND* event

lwesp_conn_p **lwesp_evt_conn_send_get_conn**(*lwesp_evt_t* *cc)
Get connection handle for data sent event.

Parameters
cc – **[in]** Event handle

Returns
Connection handle

size_t **lwesp_evt_conn_send_get_length**(*lwesp_evt_t* *cc)
Get number of bytes sent on connection.

Parameters
cc – **[in]** Event handle

Returns
Number of bytes sent

lwespr_t **lwesp_evt_conn_send_get_result**(*lwesp_evt_t* *cc)
Check if connection send was successful.

Parameters
cc – **[in]** Event handle

Returns
Member of *lwespr_t* enumeration

Connection active

Event helper functions for *LWESP_EVT_CONN_ACTIVE* event

lwesp_conn_p **lwesp_evt_conn_active_get_conn**(*lwesp_evt_t* *cc)
Get connection handle.

Parameters
cc – **[in]** Event handle

Returns
Connection handle

uint8_t **lwesp_evt_conn_active_is_client**(*lwesp_evt_t* *cc)
Check if new connection is client.

Parameters
cc – **[in]** Event handle

Returns
1 if client, 0 otherwise

Connection close event

Event helper functions for *LWESP_EVT_CONN_CLOSE* event

lwesp_conn_p **lwesp_evt_conn_close_get_conn**(*lwesp_evt_t* *cc)

Get connection handle.

Parameters
cc – **[in]** Event handle

Returns
Connection handle

uint8_t **lwesp_evt_conn_close_is_client**(*lwesp_evt_t* *cc)

Check if just closed connection was client.

Parameters
cc – **[in]** Event handle

Returns
1 if client, 0 otherwise

uint8_t **lwesp_evt_conn_close_is_forced**(*lwesp_evt_t* *cc)

Check if connection close even was forced by user.

Parameters
cc – **[in]** Event handle

Returns
1 if forced, 0 otherwise

lwespr_t **lwesp_evt_conn_close_get_result**(*lwesp_evt_t* *cc)

Get connection close event result.

Parameters
cc – **[in]** Event handle

Returns
Member of *lwespr_t* enumeration

Connection poll

Event helper functions for *LWESP_EVT_CONN_POLL* event

lwesp_conn_p **lwesp_evt_conn_poll_get_conn**(*lwesp_evt_t* *cc)

Get connection handle.

Parameters
cc – **[in]** Event handle

Returns
Connection handle

Connection error

Event helper functions for *LWESP_EVT_CONN_ERROR* event

lwespr_t **lwesp_evt_conn_error_get_error**(*lwesp_evt_t* *cc)

Get connection error type.

Parameters
cc – **[in]** Event handle

Returns
Member of *lwespr_t* enumeration

lwesp_conn_type_t **lwesp_evt_conn_error_get_type**(*lwesp_evt_t* *cc)

Get connection type.

Parameters
cc – **[in]** Event handle

Returns
Member of *lwespr_t* enumeration

const char ***lwesp_evt_conn_error_get_host**(*lwesp_evt_t* *cc)

Get connection host.

Parameters
cc – **[in]** Event handle

Returns
Host name for connection

lwesp_port_t **lwesp_evt_conn_error_get_port**(*lwesp_evt_t* *cc)

Get connection port.

Parameters
cc – **[in]** Event handle

Returns
Host port number

void ***lwesp_evt_conn_error_get_arg**(*lwesp_evt_t* *cc)

Get user argument.

Parameters
cc – **[in]** Event handle

Returns
User argument

List access points

Event helper functions for *LWESP_EVT_STA_LIST_AP* event

lwespr_t **lwesp_evt_sta_list_ap_get_result**(*lwesp_evt_t* *cc)

Get command success result.

Parameters
cc – **[in]** Event handle

Returns
Member of *lwespr_t* enumeration

lwesp_ap_t ***lwesp_evt_sta_list_ap_get_aps**(*lwesp_evt_t* *cc)

Get access points.

Parameters
cc – **[in]** Event handle

Returns
Pointer to *lwesp_ap_t* with first access point description

size_t **lwesp_evt_sta_list_ap_get_length**(*lwesp_evt_t* *cc)

Get number of access points found.

Parameters
cc – **[in]** Event handle

Returns
Number of access points found

Join access point

Event helper functions for *LWESP_EVT_STA_JOIN_AP* event

lwespr_t **lwesp_evt_sta_join_ap_get_result**(*lwesp_evt_t* *cc)

Get command success result.

Parameters
cc – **[in]** Event handle

Returns
Member of *lwespr_t* enumeration

Get access point info

Event helper functions for *LWESP_EVT_STA_INFO_AP* event

lwespr_t **lwesp_evt_sta_info_ap_get_result**(*lwesp_evt_t* *cc)

Get command result.

Parameters
cc – **[in]** Event handle

Returns
Member of *lwespr_t* enumeration

const char ***lwesp_evt_sta_info_ap_get_ssid**(*lwesp_evt_t* *cc)

Get current AP name.

Parameters
cc – **[in]** Event handle

Returns
AP name

lwesp_mac_t **lwesp_evt_sta_info_ap_get_mac**(*lwesp_evt_t* *cc)

Get current AP MAC address.

Parameters
cc – **[in]** Event handle

Returns
AP MAC address

uint8_t **lwesp_evt_sta_info_ap_get_channel**(*lwesp_evt_t* *cc)

Get current AP channel.

Parameters
cc – **[in]** Event handle

Returns
AP channel

int16_t **lwesp_evt_sta_info_ap_get_rssi**(*lwesp_evt_t* *cc)

Get current AP rssi.

Parameters
cc – **[in]** Event handle

Returns
AP rssi

Get host address by name

Event helper functions for *LWESP_EVT_DNS_HOSTBYNAME* event

lwespr_t **lwesp_evt_dns_hostbyname_get_result**(*lwesp_evt_t* *cc)

Get resolve result.

Parameters
cc – **[in]** Event handle

Returns
Member of *lwespr_t* enumeration

const char ***lwesp_evt_dns_hostbyname_get_host**(*lwesp_evt_t* *cc)

Get hostname used to resolve IP address.

Parameters
cc – **[in]** Event handle

Returns
Hostname

lwesp_ip_t ***lwesp_evt_dns_hostbyname_get_ip**(*lwesp_evt_t* *cc)

Get IP address from DNS function.

Parameters
cc – **[in]** Event handle

Returns
IP address

Ping

Event helper functions for *LWESP_EVT_PING* event

lwespr_t **lwesp_evt_ping_get_result**(*lwesp_evt_t* *cc)

Get ping status.

Parameters
cc – **[in]** Event handle

Returns
Member of *lwespr_t* enumeration

const char ***lwesp_evt_ping_get_host**(*lwesp_evt_t* *cc)

Get hostname used to ping.

Parameters
cc – **[in]** Event handle

Returns
Hostname

uint32_t **lwesp_evt_ping_get_time**(*lwesp_evt_t* *cc)

Get time required for ping.

Parameters
cc – **[in]** Event handle

Returns
Ping time

Simple Network Time Protocol

Event helper functions for *LWESP_EVT_SNTP_TIME* event

lwespr_t **lwesp_evt_sntp_time_get_result**(*lwesp_evt_t* *cc)

Get command success result.

Parameters
cc – **[in]** Event handle

Returns
Member of *lwespr_t* enumeration

const struct tm *`lwesp_evt_sntp_time_get_datetime`(*lwesp_evt_t* *cc)

Get date time pointer with data.

Parameters
cc – **[in]** Event handle

Returns
pointer to read-only structure with datetime

Web Server

Event helper functions for *LWESP_EVT_WEBSERVER* event

uint8_t `lwesp_evt_webserver_get_status`(*lwesp_evt_t* *cc)

Get web server status.

Parameters
cc – **[in]** Event handle

Returns
Web server status code

Server

Event helper functions for *LWESP_EVT_SERVER*

lwespr_t `lwesp_evt_server_get_result`(*lwesp_evt_t* *cc)

Get server command result.

Parameters
cc – **[in]** Event handle

Returns
Member of *lwespr_t* enumeration

lwesp_port_t `lwesp_evt_server_get_port`(*lwesp_evt_t* *cc)

Get port for server operation.

Parameters
cc – **[in]** Event handle

Returns
Server port

uint8_t `lwesp_evt_server_is_enable`(*lwesp_evt_t* *cc)

Check if operation was to enable or disable server.

Parameters
cc – **[in]** Event handle

Returns
1 if enable, 0 otherwise

Typedefs

typedef *lwespr_t* (*`lwesp_evt_fn`)(struct lwesp_evt *evt)

Event function prototype.

Param evt
[in] Callback event data

Return
lwespOK on success, member of *lwespr_t* otherwise

Enums

enum `lwesp_evt_type_t`

List of possible callback types received to user.

Values:

enumerator `LWESP_EVT_INIT_FINISH`

Initialization has been finished at this point

enumerator `LWESP_EVT_RESET_DETECTED`

Device reset detected

enumerator `LWESP_EVT_RESET`

Device reset operation finished

enumerator `LWESP_EVT_RESTORE`

Device restore operation finished

enumerator `LWESP_EVT_CMD_TIMEOUT`

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

enumerator `LWESP_EVT_DEVICE_PRESENT`

Notification when device present status changes

enumerator `LWESP_EVT_AT_VERSION_NOT_SUPPORTED`

Library does not support firmware version on ESP device.

enumerator `LWESP_EVT_CONN_RECV`

Connection data received

enumerator `LWESP_EVT_CONN_SEND`

Connection data send

enumerator **LWESP_EVT_CONN_ACTIVE**

Connection just became active

enumerator **LWESP_EVT_CONN_ERROR**

Client connection start was not successful

enumerator **LWESP_EVT_CONN_CLOSE**

Connection close event. Check status if successful

enumerator **LWESP_EVT_CONN_POLL**

Poll for connection if there are any changes

enumerator **LWESP_EVT_SERVER**

Server status changed

enumerator **LWESP_EVT_KEEP_ALIVE**

Generic keep-alive event type, used as periodic timeout. Optionally enabled with *LWESP_CFG_KEEP_ALIVE*

enumerator **LWESP_EVT_WIFI_CONNECTED**

Station just connected to access point. When received, station may not have yet valid IP hence new connections cannot be started in this mode

enumerator **LWESP_EVT_WIFI_GOT_IP**

Station has valid IP. When this event is received to application, ESP has got IP from access point, but no IP has been read from device and at this moment it is still being unknown to application. Stack will proceed with IP read from device and will later send *LWESP_EVT_WIFI_IP_ACQUIRED* event.

Note: When IPv6 is enabled, this event may be called multiple times during single connection to access point, as device may report "got IP" several times. Application must take care when starting new conection from this event, not to start it multiple times

enumerator **LWESP_EVT_WIFI_DISCONNECTED**

Station just disconnected from access point

enumerator **LWESP_EVT_WIFI_IP_ACQUIRED**

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

enumerator **LWESP_EVT_STA_LIST_AP**

Station listed APs event

enumerator **LWESP_EVT_STA_JOIN_AP**

Join to access point

enumerator **LWESP_EVT_STA_INFO_AP**

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

enumerator **LWESP_EVT_AP_CONNECTED_STA**

New station just connected to ESP's access point

enumerator **LWESP_EVT_AP_DISCONNECTED_STA**

New station just disconnected from ESP's access point

enumerator **LWESP_EVT_AP_IP_STA**

New station just received IP from ESP's access point

enumerator **LWESP_EVT_DNS_HOSTBYNAME**

DNS domain service finished

enumerator **LWESP_EVT_PING**

PING service finished

enumerator **LWESP_EVT_WEBSERVER**

Web server events

enumerator **LWESP_EVT_SNTP_TIME_UPDATED**

SNTP core inside ESP device has updated the time. lwesp_sntp_* can be used to actually read the data from device. This event is just a notification, but does not contain any data. Alternatively, user can enable *LWESP_CFG_SNTP_AUTO_READ_TIME_ON_UPDATE* to request data automatically when event is received

enumerator **LWESP_EVT_SNTP_TIME**

SNTP event with date and time

enumerator **LWESP_CFG_END**

Functions

lwespr_t **lwesp_evt_register**(*lwesp_evt_fn* fn)

Register event function for global (non-connection based) events.

Parameters
fn – **[in]** Callback function to call on specific event

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_evt_unregister**(*lwesp_evt_fn* fn)

Unregister callback function for global (non-connection based) events.

Note: Function must be first registered using *lwesp_evt_register*

Parameters
fn – **[in]** Callback function to remove from event list

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwesp_evt_type_t **lwesp_evt_get_type**(*lwesp_evt_t* *cc)

Get event type.

Parameters
cc – **[in]** Event handle

Returns
Event type. Member of *lwesp_evt_type_t* enumeration

struct **lwesp_evt_t**

#include <lwesp_types.h> Global callback structure to pass as parameter to callback function.

Public Members

lwesp_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 *lwesp_evt_t*::[anonymous]::[anonymous] **reset_detected**

Reset occurred. Use with *LWESP_EVT_RESET_DETECTED* event

lwespr_t **res**

Reset operation result

Restore operation result

Send data result

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

Status of command

Result of command

struct *lwesp_evt_t*::[anonymous]::[anonymous] **reset**

Reset sequence finish. Use with *LWESP_EVT_RESET* event

struct *lwesp_evt_t*::[anonymous]::[anonymous] **restore**

Restore sequence finish. Use with *LWESP_EVT_RESTORE* event

lwesp_conn_p **conn**

Connection where data were received

Connection where data were sent

Pointer to connection

Set connection pointer

lwesp_pbuf_p **buff**

Pointer to received data

struct *lwesp_evt_t*::[anonymous]::[anonymous] **conn_data_recv**

Network data received. Use with *LWESP_EVT_CONN_RECV* event

size_t **sent**

Number of bytes sent on connection

struct *lwesp_evt_t*::[anonymous]::[anonymous] **conn_data_send**

Data send. Use with *LWESP_EVT_CONN_SEND* event

const char ***host**

Host to use for connection

Host name for DNS lookup

Host name for ping

lwesp_port_t **port**

Remote port used for connection

Server port number

lwesp_conn_type_t **type**

Connection type

void ***arg**

Connection user argument

lwespr_t **err**

Error value

struct *lwesp_evt_t*::[anonymous]::[anonymous] **conn_error**

Client connection start error. Use with *LWESP_EVT_CONN_ERROR* event

uint8_t **client**

Set to 1 if connection is/was client mode

struct *lwesp_evt_t*::[anonymous]::[anonymous] **conn_active_close**

Process active and closed statuses at the same time. Use with *LWESP_EVT_CONN_ACTIVE* or *LWESP_EVT_CONN_CLOSE* events

struct *lwesp_evt_t*::[anonymous]::[anonymous] **conn_poll**

Polling active connection to check for timeouts. Use with *LWESP_EVT_CONN_POLL* event

uint8_t **en**

Status to enable/disable server

struct *lwesp_evt_t*::[anonymous]::[anonymous] **server**

Server change event. Use with *LWESP_EVT_SERVER* event

lwesp_ap_t ***aps**

Pointer to access points

size_t **len**

Number of access points found

struct *lwesp_evt_t*::[anonymous]::[anonymous] **sta_list_ap**

Station list access points. Use with *LWESP_EVT_STA_LIST_AP* event

struct *lwesp_evt_t*::[anonymous]::[anonymous] **sta_join_ap**

Join to access point. Use with *LWESP_EVT_STA_JOIN_AP* event

lwesp_sta_info_ap_t ***info**

AP info of current station

struct *lwesp_evt_t*::[anonymous]::[anonymous] **sta_info_ap**

Current AP informations. Use with *LWESP_EVT_STA_INFO_AP* event

lwesp_mac_t ***mac**

Station MAC address

struct *lwesp_evt_t*::[anonymous]::[anonymous] **ap_conn_disconn_sta**

A new station connected or disconnected to ESP's access point. Use with *LWESP_EVT_AP_CONNECTED_STA* or *LWESP_EVT_AP_DISCONNECTED_STA* events

lwesp_ip_t ***ip**

Station IP address

Pointer to IP result

struct *lwesp_evt_t*::[anonymous]::[anonymous] **ap_ip_sta**

Station got IP address from ESP's access point. Use with *LWESP_EVT_AP_IP_STA* event

struct *lwesp_evt_t*::[anonymous]::[anonymous] **dns_hostbyname**

DNS domain service finished. Use with *LWESP_EVT_DNS_HOSTBYNAME* event

uint32_t **time**

Time required for ping. Valid only if operation succedded

struct *lwesp_evt_t*::[anonymous]::[anonymous] **ping**

Ping finished. Use with *LWESP_EVT_PING* event

const struct tm ***dt**

Pointer to datetime structure

struct *lwesp_evt_t*::[anonymous]::[anonymous] **cip_sntp_time**

SNTP time finished. Use with *LWESP_EVT_SNTP_TIME* event

uint8_t **code**

Result of command

struct *lwesp_evt_t*::[anonymous]::[anonymous] **ws_status**

Ping finished. Use with *LWESP_EVT_PING* event

union *lwesp_evt_t*::[anonymous] **evt**

Callback event union

System Flash

group **LWESP_FLASH**

System flash API.

Functions

lwespr_t **lwesp_flash_erase**(lwesp_flash_partition_t partition, uint32_t offset, uint32_t length, const *lwesp_api_cmd_evt_fn* evt_fn, void *const evt_arg, const uint32_t blocking)

Erase flash block.

Parameters

- **partition** – **[in]** Partition to do erase operation on
- **offset** – **[in]** Offset from start of partition. Must be `4kB` aligned when used. Set to `0` to erase full partition
- **length** – **[in]** Size to erase. Must be `4kB` aligned when used. Set to `0` to erase full partition
- **evt_fn** – **[in]** Callback function called when command has finished. Set to `NULL` when not used
- **evt_arg** – **[in]** Custom argument for event callback function
- **blocking** – **[in]** Status whether command should be blocking or not

Returns

lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_flash_write**(lwesp_flash_partition_t partition, uint32_t offset, const void *data, uint32_t length, const *lwesp_api_cmd_evt_fn* evt_fn, void *const evt_arg, const uint32_t blocking)

Write data to flash partition.

Parameters

- **partition** – **[in]** Partition to write to
- **offset** – **[in]** Offset from start of partition to start writing at
- **data** – **[in]** Actual data to write. Must not be NULL
- **length** – **[in]** Number of bytes to write
- **evt_fn** – **[in]** Callback function called when command has finished. Set to NULL when not used
- **evt_arg** – **[in]** Custom argument for event callback function
- **blocking** – **[in]** Status whether command should be blocking or not

Returns

lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_mfg_erase**(lwesp_mfg_namespace_t mfgns, const char *key, uint32_t offset, uint32_t length, const *lwesp_api_cmd_evt_fn* evt_fn, void *const evt_arg, const uint32_t blocking)

lwespr_t **lwesp_mfg_write**(lwesp_mfg_namespace_t mfgns, const char *key, lwesp_mfg_valtype_t valtype, const void *data, uint32_t length, const *lwesp_api_cmd_evt_fn* evt_fn, void *const evt_arg, const uint32_t blocking)

Write key-value pair into user MFG area.

Note: When writing into this section, no need to previously erase the data System is smart enough to do this for us, if absolutely necessary

---

Parameters

- **mfgns** – **[in]** User namespace option
- **key** – **[in]** Key to write
- **valtype** – **[in]** Value type to follow
- **data** – **[in]** Pointer to data to write. If value type is primitive type, then pointer is copied to the local structure. This means even for non-blocking calls, user can safely use local variables for data pointers.
- **length** – **[in]** Length of data to write. It only makes sense for string and binary data types, otherwise it is derived from value type parameter and can be set to 0 by user
- **evt_fn** – **[in]** Callback function called when command has finished. Set to NULL when not used
- **evt_arg** – **[in]** Custom argument for event callback function
- **blocking** – **[in]** Status whether command should be blocking or not

Returns

lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_mfg_read**(lwesp_mfg_namespace_t mfgns, const char *key, void *data, uint32_t btr, uint32_t offset, uint32_t *br, const *lwesp_api_cmd_evt_fn* evt_fn, void *const evt_arg, const uint32_t blocking)

Read key-value pair into user MFG area.

Note: When writing into this section, no need to previously erase the data System is smart enough to do this for us, if absolutely necessary

Parameters

- **mfgns** – **[in]** User namespace option
- **key** – **[in]** Key to ead
- **data** – **[in]** Pointer to data to write received data to
- **btr** – **[in]** Number of bytes to read
- **offset** – **[in]** Offset from partition start to read data from
- **br** – **[out]** Pointer to output variable to write number of bytes read
- **evt_fn** – **[in]** Callback function called when command has finished. Set to `NULL` when not used
- **evt_arg** – **[in]** Custom argument for event callback function
- **blocking** – **[in]** Status whether command should be blocking or not

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

Hostname

group **LWESP_HOSTNAME**

Hostname API.

Functions

lwespr_t **lwesp_hostname_set**(const char *hostname, const *lwesp_api_cmd_evt_fn* evt_fn, void *const evt_arg, const uint32_t blocking)

Set hostname of WiFi station.

Parameters

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

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_hostname_get**(char *hostname, size_t size, const *lwesp_api_cmd_evt_fn* evt_fn, void *const evt_arg, const uint32_t blocking)

Get hostname of WiFi station.

Parameters

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

Returns

lwespOK on success, member of *lwespr_t* enumeration otherwise

Input module

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

- Indirect processing with *`lwesp_input()`* (default mode)
- Direct processing with *`lwesp_input_process()`*

Tip: Direct or indirect processing mode is select by setting *`LWESP_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 *`lwesp_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 *`lwesp_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 **LWESP_INPUT**

Input function for received data.

Functions

lwespr_t **lwesp_input**(const void *data, size_t len)

Write data to input buffer.

Note: *LWESP_CFG_INPUT_USE_PROCESS* must be disabled to use this function

Parameters

- **data** – **[in]** Pointer to data to write
- **len** – **[in]** Number of data elements in units of bytes

Returns

lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_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: *LWESP_CFG_INPUT_USE_PROCESS* must be enabled to use this function

Parameters

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

Returns

lwespOK on success, member of *lwespr_t* enumeration otherwise

Multicast DNS

group **LWESP_MDNS**

mDNS function

Functions

lwespr_t **lwesp_mdns_set_config**(uint8_t en, const char *host, const char *server, *lwesp_port_t* port, const *lwesp_api_cmd_evt_fn* evt_fn, void *const evt_arg, const uint32_t blocking)

Configure mDNS parameters with hostname and server.

Parameters

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

Returns

lwespOK on success, member of *lwespr_t* enumeration otherwise

Memory manager

group **LWESP_MEM**

Dynamic memory manager.

Functions

uint8_t **lwesp_mem_assignmemory**(const *lwesp_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

Note: Function is not available when *LWESP_CFG_MEM_CUSTOM* is `1`

Parameters

- **regions** – **[in]** Pointer to list of regions to use for allocations
- **len** – **[in]** Number of regions to use

Returns

`1` on success, `0` otherwise

void *lwesp_mem_malloc(size_t size)

Allocate memory of specific size.

Note: Function is not available when *LWESP_CFG_MEM_CUSTOM* is 1 and must be implemented by user

Parameters

size – **[in]** Number of bytes to allocate

Returns

Memory address on success, NULL otherwise

void *lwesp_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

Note: Function is not available when *LWESP_CFG_MEM_CUSTOM* is 1 and must be implemented by user

Parameters

- **ptr** – **[in]** Pointer to current allocated memory to resize, returned using *lwesp_mem_malloc*, *lwesp_mem_calloc* or *lwesp_mem_realloc* functions
- **size** – **[in]** Number of bytes to allocate on new memory

Returns

Memory address on success, NULL otherwise

void *lwesp_mem_calloc(size_t num, size_t size)

Allocate memory of specific size and set memory to zero.

Note: Function is not available when *LWESP_CFG_MEM_CUSTOM* is 1 and must be implemented by user

Parameters

- **num** – **[in]** Number of elements to allocate
- **size** – **[in]** Size of each element

Returns

Memory address on success, NULL otherwise

void lwesp_mem_free(void *ptr)

Free memory.

Note: Function is not available when *LWESP_CFG_MEM_CUSTOM* is 1 and must be implemented by user

Parameters
ptr – **[in]** Pointer to memory previously returned using *lwesp_mem_malloc*, *lwesp_mem_calloc* or *lwesp_mem_realloc* functions

uint8_t **lwesp_mem_free_s**(void **ptr)

Free memory in safe way by invalidating pointer after freeing.

Parameters
ptr – **[in]** Pointer to pointer to allocated memory to free

Returns
1 on success, 0 otherwise

struct **lwesp_mem_region_t**

#include <lwesp_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 `lwesp_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 `lwesp_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 *lwesp_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 19: Packet buffer concat example

```
lwesp_pbuf_p a, b;

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

/* Link them together with concat operation */
/* Reference on b will stay as is, won't be increased */
lwesp_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
 */
lwesp_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 *lwesp_pbuf_chain()*, it also has to manually free its reference using *lwesp_pbuf_free()* function. Forgetting to free pbuf invokes memory leak

An example code showing proper usage of chain operation:

Listing 20: Packet buffer chain example

```
lwesp_pbuf_p a, b;

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

/* Chain both pbufs together */
/* This will increase reference on b as 2 variables now point to it */
lwesp_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 */
lwesp_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
 */
lwesp_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 21: Packet buffer data extraction

```
const void* data;
size_t pos, len;
lwesp_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 = lwesp_pbuf_new(strlen(str_a));
b = lwesp_pbuf_new(strlen(str_b));
c = lwesp_pbuf_new(strlen(str_c));

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

/* Connect pbufs together */
lwesp_pbuf_chain(a, b);
lwesp_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 = lwesp_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;
}

```

(continues on next page)

(continued from previous page)

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

group **LWESP_PBUF**

Packet buffer manager.

Typedefs

typedef struct lwesp_pbuf ***lwesp_pbuf_p**

Pointer to *lwesp_pbuf_t* structure.

Functions

lwesp_pbuf_p **lwesp_pbuf_new**(size_t len)

Allocate packet buffer for network data of specific size.

Parameters
len – **[in]** Length of payload memory to allocate

Returns
Pointer to allocated memory, NULL otherwise

size_t **lwesp_pbuf_free**(*lwesp_pbuf_p* pbuf)

Free previously allocated packet buffer.

See also:

lwesp_pbuf_free_s

Note: Application must not use reference to pbuf after the call to this function. It is advised to immediately set pointer to NULL or to call. Alternatively, call *lwesp_pbuf_free_s*, which will reset the pointer after free operation has been completed

Parameters
pbuf – **[in]** Packet buffer to free

Returns
Number of freed pbufs from head

size_t **lwesp_pbuf_free_s**(*lwesp_pbuf_p* *pbuf)

Free previously allocated packet buffer in safe way. Function accepts pointer to pointer and will set the pointer to NULL after the successful allocation.

Parameters
pbuf_ptr – **[inout]** Pointer to pointer to packet buffer

Returns
Number of packet buffers freed in the chain

void *lwesp_pbuf_data(const *lwesp_pbuf_p* pbuf)

Get data pointer from packet buffer.

Parameters
pbuf – **[in]** Packet buffer

Returns
Pointer to data buffer on success, NULL otherwise

size_t lwesp_pbuf_length(const *lwesp_pbuf_p* pbuf, uint8_t tot)

Get length of packet buffer.

Parameters

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

Returns
Length of data in units of bytes

uint8_t lwesp_pbuf_set_length(*lwesp_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

Parameters

- **pbuf** – **[in]** Pbuf to make it smaller
- **new_len** – **[in]** New length in units of bytes

Returns
1 on success, 0 otherwise

lwespr_t lwesp_pbuf_take(*lwesp_pbuf_p* pbuf, const void *data, size_t len, size_t offset)

Copy user data to chain of pbufs.

Parameters

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

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

size_t lwesp_pbuf_copy(*lwesp_pbuf_p* pbuf, void *data, size_t len, size_t offset)

Copy memory from pbuf to user linear memory.

Parameters

- **pbuf** – **[in]** Pbuf to copy from

- **data** – **[out]** User linear memory to copy to
- **len** – **[in]** Length of data in units of bytes
- **offset** – **[in]** Possible start offset in pbuf

Returns
Number of bytes copied

lwespr_t **lwesp_pbuf_cat**(*lwesp_pbuf_p* head, const *lwesp_pbuf_p* tail)

Concatenate 2 packet buffers together to one big packet.

See also:

lwesp_pbuf_cat_s

See also:

lwesp_pbuf_chain

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

Parameters

- **head** – **[in]** Head packet buffer to append new pbuf to
- **tail** – **[in]** Tail packet buffer to append to head pbuf

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_pbuf_cat_s**(*lwesp_pbuf_p* head, *lwesp_pbuf_p* *tail)

Concatenate 2 packet buffers together to one big packet with safe pointer management.

See also:

lwesp_pbuf_cat

See also:

lwesp_pbuf_chain

Note: After `tail` pbuf has been added to `head` pbuf chain, `tail` pointer will be set to `NULL`

Parameters

- **head** – **[in]** Head packet buffer to append new pbuf to
- **tail** – **[in]** Pointer to pointer to tail packet buffer to append to head pbuf. Pointed memory will be set to `NULL` after successful concatenation

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_pbuf_chain**(*lwesp_pbuf_p* head, *lwesp_pbuf_p* tail)

Chain 2 pbufs together. Similar to *lwesp_pbuf_cat* but now new reference is done from head pbuf to tail pbuf.

See also:

lwesp_pbuf_cat

See also:

lwesp_pbuf_cat_s

See also:

lwesp_pbuf_chain_s

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

Parameters

- **head** – **[in]** Head packet buffer to append new pbuf to
- **tail** – **[in]** Tail packet buffer to append to head pbuf

Returns

lwespOK on success, member of *lwespr_t* enumeration otherwise

lwesp_pbuf_p **lwesp_pbuf_unchain**(*lwesp_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`)

Parameters

head – **[in]** First pbuf in chain to remove from chain

Returns

Next pbuf after `head`

lwespr_t **lwesp_pbuf_ref**(*lwesp_pbuf_p* pbuf)

Increment reference count on pbuf.

Parameters

pbuf – **[in]** pbuf to increase reference

Returns

lwespOK on success, member of *lwespr_t* enumeration otherwise

uint8_t **lwesp_pbuf_get_at**(const *lwesp_pbuf_p* pbuf, size_t pos, uint8_t *el)

Get value from pbuf at specific position.

Parameters

- **pbuf** – **[in]** Pbuf used to get data from

- **pos** – **[in]** Position at which to get element
- **el** – **[out]** Output variable to save element value at desired position

Returns
1 on success, 0 otherwise

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

See also:

lwesp_pbuf_strcmp

Note: Compare is done on entire pbuf chain

Parameters

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

Returns
0 if equal, LWESP_SIZET_MAX if memory/offset too big or anything between if not equal

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

See also:

lwesp_pbuf_memcmp

Note: Compare is done on entire pbuf chain

Parameters

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

Returns
0 if equal, LWESP_SIZET_MAX if memory/offset too big or anything between if not equal

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

See also:

lwesp_pbuf_strfind

Parameters

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

Returns
LWESP_SIZET_MAX if no match or position where in pbuf we have a match

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

See also:

lwesp_pbuf_memfind

Parameters

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

Returns
LWESP_SIZET_MAX if no match or position where in pbuf we have a match

uint8_t **lwesp_pbuf_advance**(*lwesp_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

Parameters

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

Returns
1 on success, 0 otherwise

lwesp_pbuf_p **lwesp_pbuf_skip**(*lwesp_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

Parameters

- **pbuf** – **[in]** Start of pbuf chain
- **offset** – **[in]** Offset in units of bytes to skip

- **new_offset** – **[out]** Pointer to output variable to save new offset in returned pbuf

Returns
New pbuf on success, NULL otherwise

void *lwesp_pbuf_get_linear_addr(const *lwesp_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

Parameters

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

Returns
Pointer to memory on success, NULL otherwise

void **lwesp_pbuf_set_ip**(*lwesp_pbuf_p* pbuf, const *lwesp_ip_t* *ip, *lwesp_port_t* port)
Set IP address and port number for received data.

Parameters

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

void **lwesp_pbuf_dump**(*lwesp_pbuf_p* p, uint8_t seq)
Dump and debug pbuf chain.

Parameters

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

struct **lwesp_pbuf_t**
#include <lwesp_private.h> Packet buffer structure.

Public Members

struct lwesp_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

lwesp_ip_t **ip**

Remote address for received IPD data

lwesp_port_t **port**

Remote port for received IPD data

Ping support

group **LWESP_PING**

Ping server and get response time.

Functions

lwespr_t **lwesp_ping**(const char *host, uint32_t *time, const *lwesp_api_cmd_evt_fn* evt_fn, void *const evt_arg, const uint32_t blocking)

Ping server and get response time from it.

Parameters

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

Returns

lwespOK on success, member of *lwespr_t* enumeration otherwise

Server

group **LWESP_SERVER**

Server mode.

Functions

lwespr_t **lwesp_set_server**(uint8_t en, *lwesp_port_t* port, uint16_t max_conn, uint16_t timeout, *lwesp_evt_fn* cb, const *lwesp_api_cmd_evt_fn* evt_fn, void *const evt_arg, const uint32_t blocking)

Enables or disables server mode.

Parameters

- **en** – **[in]** Set to 1 to enable server, 0 otherwise
- **port** – **[in]** Port number used to listen on. Must also be used when disabling server mode
- **max_conn** – **[in]** Number of maximal connections populated by server
- **timeout** – **[in]** Time used to automatically close the connection in units of seconds. Set to 0 to disable timeout feature (not recommended)
- **server_evt_fn** – **[in]** Connection callback function for new connections started as server
- **evt_fn** – **[in]** Callback function called when command has finished. Set to NULL when not used
- **evt_arg** – **[in]** Custom argument for event callback function
- **blocking** – **[in]** Status whether command should be blocking or not

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

Smart config

group **LWESP_SMART**

SMART function on ESP device.

Functions

lwespr_t **lwesp_smart_set_config**(uint8_t en, const *lwesp_api_cmd_evt_fn* evt_fn, void *const evt_arg, const uint32_t blocking)

Configure SMART function on ESP device.

Parameters

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

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

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 22: Minimum SNTP example

```
/*
 * A simple example to get current time using SNTP protocol
 * thanks to AT commands being supported by Espressif
 */
#include "sntp.h"
#include "lwesp/lwesp.h"

/**
 * \brief           Run SNTP
 */
void
sntp_gettime(void) {
    struct tm dt;

    /* Enable SNTP with default configuration for NTP servers */
    if (lwesp_sntp_set_config(1, 1, NULL, NULL, NULL, NULL, NULL, 1) == lwespOK) {
        lwesp_delay(5000);

        /* Get actual time and print it */
        if (lwesp_sntp_gettime(&dt, NULL, NULL, 1) == lwespOK) {
            printf("Date & time: %d.%d.%d, %d:%d:%d\r\n", (int)dt.tm_mday, (int)(dt.tm_
↪mon + 1),
                   (int)(dt.tm_year + 1900), (int)dt.tm_hour, (int)dt.tm_min, (int)dt.tm_
↪sec);
        }
    }
}
```

group **LWESP_SNTP**

Simple network time protocol supported by AT commands.

Functions

lwespr_t **lwesp_sntp_set_config**(uint8_t en, int16_t tz, const char *h1, const char *h2, const char *h3, const *lwesp_api_cmd_evt_fn* evt_fn, void *const evt_arg, const uint32_t blocking)

Configure SNTP mode parameters. It must be called prior any *lwesp_sntp_gettime* can be used, otherwise wrong data will be received back.

Parameters

- **en** – **[in]** Status whether SNTP mode is enabled or disabled on ESP device
- **tz** – **[in]** Timezone to use when SNTP acquires time, between `-12` and `14`
- **h1** – **[in]** Optional first SNTP server for time. Set to `NULL` if not used
- **h2** – **[in]** Optional second SNTP server for time. Set to `NULL` if not used

- **h3** – **[in]** Optional third SNTP server for time. Set to NULL if not used
- **evt_fn** – **[in]** Callback function called when command has finished. Set to NULL when not used
- **evt_arg** – **[in]** Custom argument for event callback function
- **blocking** – **[in]** Status whether command should be blocking or not

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_sntp_get_config**(uint8_t *en, int16_t *tz, char *h1, char *h2, char *h3, const *lwesp_api_cmd_evt_fn* evt_fn, void *const evt_arg, const uint32_t blocking)

Get current SNTP configuration.

Todo:

Parse response for hostnames, which is not done at the moment

Parameters

- **en** – **[in]** Pointer to status variable
- **tz** – **[in]** Pointer to timezone
- **h1** – **[in]** Optional first SNTP server for time. Set to NULL if not used, otherwise value is copied into the pointer. Must be sufficient enough
- **h2** – **[in]** Optional second SNTP server for time. Set to NULL if not used otherwise value is copied into the pointer. Must be sufficient enough
- **h3** – **[in]** Optional third SNTP server for time. Set to NULL if not used otherwise value is copied into the pointer. Must be sufficient enough
- **evt_fn** – **[in]** Callback function called when command has finished. Set to NULL when not used
- **evt_arg** – **[in]** Custom argument for event callback function
- **blocking** – **[in]** Status whether command should be blocking or not

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_sntp_set_interval**(uint32_t interval, const *lwesp_api_cmd_evt_fn* evt_fn, void *const evt_arg, const uint32_t blocking)

Set SNTP synchronization interval on Espressif device SNTP must be configured using *lwesp_sntp_set_config* before you can use this function.

Note: This command is not available for all Espressif devices using AT commands and will return error when this is the case.

Parameters

- **interval** – **[in]** Synchronization interval in units of seconds. Value can be set between 15 and 4294967 included

- **evt_fn** – **[in]** Callback function called when command has finished. Set to NULL when not used
- **evt_arg** – **[in]** Custom argument for event callback function
- **blocking** – **[in]** Status whether command should be blocking or not

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_sntp_get_interval**(uint32_t *interval, const *lwesp_api_cmd_evt_fn* evt_fn, void *const evt_arg, const uint32_t blocking)

Get SNTP synchronization interval on Espressif device SNTP must be configured using *lwesp_sntp_set_config* before you can use this function.

Note: This command is not available for all Espressif devices using AT commands and will return error when this is the case.

Parameters

- **interval** – **[in]** Pointer to variable to write interval. It is value in seconds. It must not be NULL
- **evt_fn** – **[in]** Callback function called when command has finished. Set to NULL when not used
- **evt_arg** – **[in]** Custom argument for event callback function
- **blocking** – **[in]** Status whether command should be blocking or not

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_sntp_gettime**(struct tm *dt, const *lwesp_api_cmd_evt_fn* evt_fn, void *const evt_arg, const uint32_t blocking)

Get time from SNTP servers SNTP must be configured using *lwesp_sntp_set_config* before you can use this function.

Parameters

- **dt** – **[out]** Pointer to struct tm structure to fill with date and time values
- **evt_fn** – **[in]** Callback function called when command has finished. Set to NULL when not used
- **evt_arg** – **[in]** Custom argument for event callback function
- **blocking** – **[in]** Status whether command should be blocking or not

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

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 23: Station manager used with all examples

```
/*
 * Station manager to connect station to access point.
 *
 * It is consider as a utility module, simple set of helper functions
 * to quickly connect to access point.
 *
 * It utilizes 2 different modes, sequential or asynchronous.
 *
 * Sequential:
 * ==========
 * Call station_manager_connect_to_preferred_access_point function to connect to access point
 * in blocking mode until being ready to move forward.
 *
 * Asynchronous:
 * ============
 * Call station_manager_connect_to_access_point_async_init to initialize
 * asynchronous connect mode and activity will react upon received LwESP events to application.
 *
 * Define list of access points:
 * ============================
 * Have a look at "ap_list_preferred" variable and define
 * list of preferred access point's SSID and password.
 * Ordered by "most preferred" at the lower array index.
 */
#include "station_manager.h"
#include "lwesp/lwesp.h"
#include "utils.h"

/**
 * \brief           Private access-point and station management system
 *
 * This is used for asynchronous connection to access point
 */
typedef struct {
    size_t index_preferred_list; /*!< Current index position of preferred array */
    size_t index_scanned_list;   /*!< Current index position in array of scanned APs */

    uint8_t command_is_running; /*!< Indicating if command is currently in progress */
} prv_ap_data_t;

/* Arguments for callback function */
#define ARG_SCAN    (void*)1
```

(continues on next page)

(continued from previous page)

```c
#define ARG_CONNECT (void*)2

/* Function declaration */
static void prv_cmd_event_fn(lwespr_t status, void* arg);
static void prv_try_next_access_point(void);

/*
 * 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
 */
static const ap_entry_t ap_list_preferred[] = {
    //{ .ssid = "SSID name", .pass = "SSID password" },
    {.ssid = "TilenM_ST", .pass = "its private"},
    {.ssid = "404WiFiNotFound", .pass = "its private"},
    {.ssid = "Majerle WIFI", .pass = "majerle_internet_private"},
    {.ssid = "Majerle AMIS", .pass = "majerle_internet_private"},
};
static lwesp_ap_t ap_list_scanned[100]; /* Scanned access points information */
static size_t ap_list_scanned_len = 0;  /* Number of scanned access points */
static prv_ap_data_t ap_async_data;     /* Asynchronous data structure */

/* Command to execute to start scanning access points */
#define prv_scan_ap_command_ex(blocking)                                        \
    lwesp_sta_list_ap(NULL, ap_list_scanned, LWESP_ARRAYSIZE(ap_list_scanned), &ap_list_scanned_len, NULL, NULL,       \
                      (blocking))
#define prv_scan_ap_command()                                                   \
    do {                                                                        \
        if (!ap_async_data.command_is_running) {                                \
            ap_async_data.command_is_running =                                  \
                lwesp_sta_list_ap(NULL, ap_list_scanned, LWESP_ARRAYSIZE(ap_list_scanned), &ap_list_scanned_len,       \
                                  prv_cmd_event_fn, ARG_SCAN, 0)                \
                == lwespOK;                                                     \
        }                                                                       \
    } while (0)

/**
 * \brief           Every internal command execution callback
 * \param[in]       status: Execution status result
 * \param[in]       arg: Custom user argument
```

(continues on next page)

(continued from previous page)

```
 */
static void
prv_cmd_event_fn(lwespr_t status, void* arg) {
    LWESP_UNUSED(status);
    /*
     * Command has now successfully finish
     * and callbacks have been properly processed
     */
    ap_async_data.command_is_running = 0;

    if (arg == ARG_SCAN) {
        /* Immediately try to connect to access point after successful scan */
        prv_try_next_access_point();
    }
}

/**
 * \brief           Try to connect to next access point on a list
 */
static void
prv_try_next_access_point(void) {
    uint8_t tried = 0;

    /* No action to be done if command is currently in progress or already connected to␣
↪network */
    if (ap_async_data.command_is_running || lwesp_sta_has_ip()) {
        return;
    }

    /*
     * Process complete list and try to find suitable match
     *
     * Use global variable for indexes to be able to call function multiple times
     * and continue where it finished previously
     */

    /* List all preferred access points */
    for (; ap_async_data.index_preferred_list < LWESP_ARRAYSIZE(ap_list_preferred);
         ap_async_data.index_preferred_list++, ap_async_data.index_scanned_list = 0) {

        /* List all scanned access points */
        for (; ap_async_data.index_scanned_list < ap_list_scanned_len; ap_async_data.
↪index_scanned_list++) {

            /* Find a match if available */
            if (strncmp(ap_list_scanned[ap_async_data.index_scanned_list].ssid,
                        ap_list_preferred[ap_async_data.index_preferred_list].ssid,
                        strlen(ap_list_preferred[ap_async_data.index_preferred_list].
↪ssid))
                == 0) {

                /* Try to connect to the network */
```

(continues on next page)

(continued from previous page)

```
                if (!ap_async_data.command_is_running
                    && lwesp_sta_join(ap_list_preferred[ap_async_data.index_preferred_
↪list].ssid,
                                      ap_list_preferred[ap_async_data.index_preferred_
↪list].pass, NULL,
                                      prv_cmd_event_fn, ARG_CONNECT, 0)
                            == lwespOK) {
                    ap_async_data.command_is_running = 1;

                    /* Go to next index for sub-for loop and exit */
                    ap_async_data.index_scanned_list++;
                    tried = 1;
                    goto stp;
                } else {
                    /* We have a problem, needs to resume action in next run */
                }
            }
        }
    }

    /* Restart scan operation if there was no try to connect and station has no IP */
    if (!tried && !lwesp_sta_has_ip()) {
        prv_scan_ap_command();
    }
stp:
    return;
}

/**
 * \brief           Private event function for asynchronous scanning
 * \param[in]       evt: Event information
 * \return          \ref lwespOK on success, member of \ref lwespr_t otherwise
 */
static lwespr_t
prv_evt_fn(lwesp_evt_t* evt) {
    switch (evt->type) {
        case LWESP_EVT_KEEP_ALIVE:
        case LWESP_EVT_WIFI_DISCONNECTED: {
            /* Try to connect to next access point */
            prv_try_next_access_point();
            break;
        }
        case LWESP_EVT_STA_LIST_AP: {
            /*
             * After scanning gets completed
             * manually reset all indexes for comparison purposes
             */
            ap_async_data.index_scanned_list = 0;
            ap_async_data.index_preferred_list = 0;

            /* Actual connection try is done in function callback */
            break;
```

(continues on next page)

(continued from previous page)

```
        }
        default: break;
    }
    return lwespOK;
}

/**
 * \brief           Initialize asynchronous mode to connect to preferred access point
 *
 * Asynchronous mode relies on system events received by the application,
 * to determine current device status if station is being, or not, connected to access point.
 *
 * When used, async acts only upon station connection change through callbacks,
 * therefore it does not require additional system thread or user code,
 * to be able to properly handle preferred access points.
 * This certainly decreases memory consumption of the complete system.
 *
 * \ref LWESP_CFG_KEEP_ALIVE feature must be enable to properly handle all events
 * \return          \ref lwespOK on success, member of \ref lwespr_t otherwise
 */
lwespr_t
station_manager_connect_to_access_point_async_init(void) {
    /* Register system event function */
    lwesp_evt_register(prv_evt_fn);

    /*
     * Start scanning process in non-blocking mode
     *
     * This is the only command being executed from non-callback mode,
     * therefore it must be protected against other threads trying to access the same core
     */
    lwesp_core_lock();
    prv_scan_ap_command();
    lwesp_core_unlock();

    /* Return all good, things will progress (from now-on) asynchronously */
    return lwespOK;
}

/**
 * \brief           Connect to preferred access point in blocking mode
 *
 * This functionality can only be used if non-blocking approach is not used
 *
 * \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 lwespOK on success, member of \ref lwespr_t enumeration otherwise
 */
```

(continues on next page)

(continued from previous page)

```
lwespr_t
station_manager_connect_to_preferred_access_point(uint8_t unlimited) {
    lwespr_t eres;
    uint8_t tried;

    /*
     * Scan for network access points
     * In case we have access point,
     * try to connect to known AP
     */
    do {
        if (lwesp_sta_has_ip()) {
            return lwespOK;
        }

        /* Scan for access points visible to ESP device */
        printf("Scanning access points...\r\n");
        if ((eres = prv_scan_ap_command_ex(1)) == lwespOK) {
            tried = 0;

            /* Print all access points found by ESP */
            for (size_t i = 0; i < ap_list_scanned_len; i++) {
                printf("AP found: %s, CH: %d, RSSI: %d\r\n", ap_list_scanned[i].ssid, ap_
↪list_scanned[i].ch,
                       ap_list_scanned[i].rssi);
            }

            /* Process array of preferred access points with array of found points */
            for (size_t j = 0; j < LWESP_ARRAYSIZE(ap_list_preferred); j++) {

                /* Go through all scanned list */
                for (size_t i = 0; i < ap_list_scanned_len; i++) {

                    /* Try to find a match between preferred and scanned */
                    if (strncmp(ap_list_scanned[i].ssid, ap_list_preferred[j].ssid,␣
↪strlen(ap_list_scanned[i].ssid))
                        == 0) {
                        tried = 1;
                        printf("Connecting to \"%s\" network...\r\n", ap_list_
↪preferred[j].ssid);

                        /* Try to join to access point */
                        if ((eres = lwesp_sta_join(ap_list_preferred[j].ssid, ap_list_
↪preferred[j].pass, NULL, NULL,
                                                   NULL, 1))
                            == lwespOK) {
                            lwesp_ip_t ip;
                            uint8_t is_dhcp;

                            printf("Connected to %s network!\r\n", ap_list_preferred[j].
↪ssid);

```

(continues on next page)

(continued from previous page)

```
                            lwesp_sta_copy_ip(&ip, NULL, NULL, &is_dhcp);
                            utils_print_ip("Station IP address: ", &ip, "\r\n");
                            printf("; Is DHCP: %d\r\n", (int)is_dhcp);
                            return lwespOK;
                        } else {
                            printf("Connection error: %d\r\n", (int)eres);
                        }
                    }
                }
            }
            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 == lwespERRNODEVICE) {
            printf("Device is not present!\r\n");
            break;
        } else {
            printf("Error on WIFI scan procedure!\r\n");
        }
        if (!unlimited) {
            break;
        }
    } while (1);
    return lwespERR;
}
```

group **LWESP_STA**

Station API.

Functions

lwespr_t **lwesp_sta_join**(const char *name, const char *pass, const *lwesp_mac_t* *mac, const *lwesp_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.

Parameters

- **name** – **[in]** SSID of access point to connect to
- **pass** – **[in]** Password of access point. Use `NULL` if AP does not have password
- **mac** – **[in]** 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
- **evt_fn** – **[in]** Callback function called when command has finished. Set to `NULL` when not used
- **evt_arg** – **[in]** Custom argument for event callback function
- **blocking** – **[in]** Status whether command should be blocking or not

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_sta_quit**(const *lwesp_api_cmd_evt_fn* evt_fn, void *const evt_arg, const uint32_t blocking)

Quit (disconnect) from access point.

Parameters

- **evt_fn** – **[in]** Callback function called when command has finished. Set to `NULL` when not used
- **evt_arg** – **[in]** Custom argument for event callback function
- **blocking** – **[in]** Status whether command should be blocking or not

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_sta_autojoin**(uint8_t en, const *lwesp_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 *lwesp_sta_join* for more information

---

Parameters

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

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_sta_reconnect_set_config**(uint16_t interval, uint16_t rep_cnt, const *lwesp_api_cmd_evt_fn* evt_fn, void *const evt_arg, const uint32_t blocking)

Set reconnect interval and maximum tries when connection drops.

Parameters

- **interval** – **[in]** Interval in units of seconds. Valid numbers are `1-7200` or `0` to disable reconnect feature
- **rep_cnt** – **[in]** Repeat counter. Number of maximum tries for reconnect. Valid entries are `1-1000` or `0` to always try. This parameter is only valid if interval is not `0`
- **evt_fn** – **[in]** Callback function called when command has finished. Set to `NULL` when not used
- **evt_arg** – **[in]** Custom argument for event callback function
- **blocking** – **[in]** Status whether command should be blocking or not

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_sta_getip**(*lwesp_ip_t* *ip, *lwesp_ip_t* *gw, *lwesp_ip_t* *nm, const *lwesp_api_cmd_evt_fn* evt_fn, void *const evt_arg, const uint32_t blocking)

Get station IP address.

Parameters

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

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_sta_setip**(const *lwesp_ip_t* *ip, const *lwesp_ip_t* *gw, const *lwesp_ip_t* *nm, const *lwesp_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, *LWESP_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

---

Parameters

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

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_sta_getmac**(*lwesp_mac_t* *mac, const *lwesp_api_cmd_evt_fn* evt_fn, void *const evt_arg, const uint32_t blocking)

Get station MAC address.

Parameters

- **mac** – **[out]** Pointer to output variable to save MAC address
- **evt_fn** – **[in]** Callback function called when command has finished. Set to `NULL` when not used

- **evt_arg** – **[in]** Custom argument for event callback function
- **blocking** – **[in]** Status whether command should be blocking or not

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_sta_setmac**(const *lwesp_mac_t* *mac, const *lwesp_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.

Parameters

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

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

uint8_t **lwesp_sta_has_ip**(void)

Check if ESP got IP from access point.

Returns
1 on success, `0` otherwise

uint8_t **lwesp_sta_is_joined**(void)

Check if station is connected to WiFi network.

Returns
1 on success, `0` otherwise

lwespr_t **lwesp_sta_copy_ip**(*lwesp_ip_t* *ip, *lwesp_ip_t* *gw, *lwesp_ip_t* *nm, uint8_t *is_dhcp)

Copy IP address from internal value to user variable.

Note: Use *lwesp_sta_getip* to refresh actual IP value from device

---

Parameters

- **ip** – **[out]** Pointer to output IP variable. Set to `NULL` if not interested in IP address
- **gw** – **[out]** Pointer to output gateway variable. Set to `NULL` if not interested in gateway address
- **nm** – **[out]** Pointer to output netmask variable. Set to `NULL` if not interested in netmask address
- **is_dhcp** – **[out]** Pointer to output DHCP status variable. Set to `NULL` if not interested

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_sta_list_ap**(const char *ssid, *lwesp_ap_t* *aps, size_t apsl, size_t *apf, const *lwesp_api_cmd_evt_fn* evt_fn, void *const evt_arg, const uint32_t blocking)

List for available access points ESP can connect to.

Parameters

- **ssid** – **[in]** Optional SSID name to search for. Set to `NULL` to disable filter
- **aps** – **[in]** Pointer to array of available access point parameters
- **apsl** – **[in]** Length of aps array
- **apf** – **[out]** Pointer to output variable to save number of access points found
- **evt_fn** – **[in]** Callback function called when command has finished. Set to `NULL` when not used
- **evt_arg** – **[in]** Custom argument for event callback function
- **blocking** – **[in]** Status whether command should be blocking or not

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_sta_get_ap_info**(*lwesp_sta_info_ap_t* *info, const *lwesp_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

Parameters

- **info** – **[in]** Pointer to connected access point information
- **evt_fn** – **[in]** Callback function called when command has finished. Set to `NULL` when not used
- **evt_arg** – **[in]** Custom argument for event callback function
- **blocking** – **[in]** Status whether command should be blocking or not

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

uint8_t **lwesp_sta_is_ap_802_11b**(*lwesp_ap_t* *ap)

Check if access point is `802.11b` compatible.

Parameters
ap – **[in]** Access point detailes acquired by *lwesp_sta_list_ap*

Returns
1 on success, `0` otherwise

uint8_t **lwesp_sta_is_ap_802_11g**(*lwesp_ap_t* *ap)

Check if access point is `802.11g` compatible.

Parameters
ap – **[in]** Access point detailes acquired by *lwesp_sta_list_ap*

Returns
1 on success, `0` otherwise

uint8_t **lwesp_sta_is_ap_802_11n**(*lwesp_ap_t* *ap)

Check if access point is 802.11n compatible.

Parameters
ap – **[in]** Access point detailes acquired by *lwesp_sta_list_ap*

Returns
1 on success, 0 otherwise

uint8_t **lwesp_sta_has_ipv6_local**(void)

Check if station has local IPV6 IP Local IP is used between station and router.

Note: Defined as macro with 0 constant if *LWESP_CFG_IPV6* is disabled

Returns
1 if local IPv6 is available, 0 otherwise

uint8_t **lwesp_sta_has_ipv6_global**(void)

Check if station has global IPV6 IP Global IP is used router and outside network.

Note: Defined as macro with 0 constant if *LWESP_CFG_IPV6* is disabled

Returns
1 if global IPv6 is available, 0 otherwise

struct **lwesp_sta_t**

#include <lwesp_types.h> Station data structure.

Public Members

lwesp_ip_t **ip**

IP address of connected station

lwesp_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 **LWESP_TIMEOUT**

Timeout manager.

Typedefs

typedef void (***lwesp_timeout_fn**)(void *arg)

Timeout callback function prototype.

Param arg
[in] Custom user argument

Functions

lwespr_t **lwesp_timeout_add**(uint32_t time, *lwesp_timeout_fn* fn, void *arg)

Add new timeout to processing list.

Parameters

- **time** – **[in]** Time in units of milliseconds for timeout execution
- **fn** – **[in]** Callback function to call when timeout expires
- **arg** – **[in]** Pointer to user specific argument to call when timeout callback function is executed

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_timeout_remove**(*lwesp_timeout_fn* fn)

Remove callback from timeout list.

Parameters
fn – **[in]** Callback function to identify timeout to remove

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

struct **lwesp_timeout_t**

#include <lwesp_types.h> Timeout structure.

Public Members

struct lwesp_timeout *__next__

Pointer to next timeout entry

uint32_t **time**

Time difference from previous entry

void ***arg**

Argument to pass to callback function

lwesp_timeout_fn **fn**

Callback function for timeout

Structures and enumerations

group **LWESP_TYPES**

List of core structures and enumerations.

Typedefs

typedef uint16_t **lwesp_port_t**

Port variable.

typedef void (***lwesp_api_cmd_evt_fn**)(*lwespr_t* res, void *arg)

Function declaration for API function command event callback function.

Param res
[in] Operation result, member of *lwespr_t* enumeration

Param arg
[in] Custom user argument

Enums

enum **lwesp_cmd_t**

List of possible messages.

Values:

enumerator **LWESP_CMD_IDLE** = 0

IDLE mode

enumerator **LWESP_CMD_RESET**

Reset device

enumerator **LWESP_CMD_ATE0**

Disable ECHO mode on AT commands

enumerator **LWESP_CMD_ATE1**

Enable ECHO mode on AT commands

enumerator **LWESP_CMD_GMR**

Get AT commands version

enumerator **LWESP_CMD_CMD**
List support AT commands

enumerator **LWESP_CMD_GSLP**
Set ESP to sleep mode

enumerator **LWESP_CMD_RESTORE**
Restore ESP internal settings to default values

enumerator **LWESP_CMD_UART**

enumerator **LWESP_CMD_SLEEP**

enumerator **LWESP_CMD_WAKEUPGPIO**

enumerator **LWESP_CMD_RFPOWER**

enumerator **LWESP_CMD_RFVDD**

enumerator **LWESP_CMD_RFAUTOTRACE**

enumerator **LWESP_CMD_SYSRAM**

enumerator **LWESP_CMD_SYSADC**

enumerator **LWESP_CMD_SYSMSG**

enumerator **LWESP_CMD_SYSLOG**

enumerator **LWESP_CMD_SYSFLASH_WRITE**
Write flash operation

enumerator **LWESP_CMD_SYSFLASH_READ**
Read flash operation

enumerator **LWESP_CMD_SYSFLASH_ERASE**
Erase flash operation

enumerator **LWESP_CMD_SYSFLASH_GET**
Get flash partitions

enumerator **LWESP_CMD_SYSMFG_WRITE**
Write manufacturing NVS data

enumerator **LWESP_CMD_SYSMFG_READ**
Read manufacturing NVS data

enumerator **LWESP_CMD_SYSMFG_ERASE**
Erase manufacturing NVS data

enumerator **LWESP_CMD_SYSMFG_GET**
Get manufacturing user partitions

enumerator **LWESP_CMD_WIFI_CWMODE**
Set wifi mode

enumerator **LWESP_CMD_WIFI_CWMODE_GET**
Get wifi mode

enumerator **LWESP_CMD_WIFI_CWLAPOPT**
Configure what is visible on CWLAP response

enumerator **LWESP_CMD_WIFI_IPV6**
Configure IPv6 support

enumerator **LWESP_CMD_WIFI_CWJAP**
Connect to access point

enumerator **LWESP_CMD_WIFI_CWRECONNCFG**
Setup reconnect interval and maximum tries

enumerator **LWESP_CMD_WIFI_CWJAP_GET**
Info of the connected access point

enumerator **LWESP_CMD_WIFI_CWQAP**
Disconnect from access point

enumerator **LWESP_CMD_WIFI_CWLAP**
List available access points

enumerator **LWESP_CMD_WIFI_CIPSTAMAC_GET**
Get MAC address of ESP station

enumerator **LWESP_CMD_WIFI_CIPSTAMAC_SET**
Set MAC address of ESP station

enumerator **LWESP_CMD_WIFI_CIPSTA_GET**
Get IP address of ESP station

enumerator **LWESP_CMD_WIFI_CIPSTA_SET**
Set IP address of ESP station

enumerator **LWESP_CMD_WIFI_CWAUTOCONN**
Configure auto connection to access point

enumerator **LWESP_CMD_WIFI_CWDHCP_SET**
Set DHCP config

enumerator **LWESP_CMD_WIFI_CWDHCP_GET**
Get DHCP config

enumerator **LWESP_CMD_WIFI_CWDHCPS_SET**
Set DHCP SoftAP IP config

enumerator **LWESP_CMD_WIFI_CWDHCPS_GET**
Get DHCP SoftAP IP config

enumerator **LWESP_CMD_WIFI_CWSAP_GET**
Get software access point configuration

enumerator **LWESP_CMD_WIFI_CWSAP_SET**
Set software access point configuration

enumerator **LWESP_CMD_WIFI_CIPAPMAC_GET**
Get MAC address of ESP access point

enumerator **LWESP_CMD_WIFI_CIPAPMAC_SET**
Set MAC address of ESP access point

enumerator **LWESP_CMD_WIFI_CIPAP_GET**
Get IP address of ESP access point

enumerator **LWESP_CMD_WIFI_CIPAP_SET**
Set IP address of ESP access point

enumerator **LWESP_CMD_WIFI_CWLIF**
Get connected stations on access point

enumerator **LWESP_CMD_WIFI_CWQIF**
Discnnect station from SoftAP

enumerator **LWESP_CMD_WIFI_WPS**
Set WPS option

enumerator **LWESP_CMD_WIFI_MDNS**
Configure MDNS function

enumerator **LWESP_CMD_WIFI_CWHOSTNAME_SET**
Set device hostname

enumerator **LWESP_CMD_WIFI_CWHOSTNAME_GET**
Get device hostname

enumerator **LWESP_CMD_TCPIP_CIPDOMAIN**
Get IP address from domain name = DNS function

enumerator **LWESP_CMD_TCPIP_CIPDNS_SET**
Configure user specific DNS servers

enumerator **LWESP_CMD_TCPIP_CIPDNS_GET**
Get DNS configuration

enumerator **LWESP_CMD_TCPIP_CIPSTATUS**
Get status of connections (deprecated, used on ESP8266 devices)

enumerator **LWESP_CMD_TCPIP_CIPSTATE**
Obtain connection state and information

enumerator **LWESP_CMD_TCPIP_CIPSTART**
Start client connection

enumerator **LWESP_CMD_TCPIP_CIPSEND**
Send network data

enumerator **LWESP_CMD_TCPIP_CIPCLOSE**
Close active connection

enumerator **LWESP_CMD_TCPIP_CIPSSLSIZE**
Set SSL buffer size for SSL connection

enumerator **LWESP_CMD_TCPIP_CIPSSLCCONF**
Set the SSL configuration

enumerator **LWESP_CMD_TCPIP_CIFSR**
Get local IP

enumerator **LWESP_CMD_TCPIP_CIPMUX**
Set single or multiple connections

enumerator **LWESP_CMD_TCPIP_CIPSERVER**

Enables/Disables server mode

enumerator **LWESP_CMD_TCPIP_CIPSERVERMAXCONN**

Sets maximal number of connections allowed for server population

enumerator **LWESP_CMD_TCPIP_CIPMODE**

Transmission mode, either transparent or normal one

enumerator **LWESP_CMD_TCPIP_CIPSTO**

Sets connection timeout

enumerator **LWESP_CMD_TCPIP_CIPRECVMODE**

Sets mode for TCP data receive (manual or automatic)

enumerator **LWESP_CMD_TCPIP_CIPRECVDATA**

Manually reads TCP data from device

enumerator **LWESP_CMD_TCPIP_CIPRECVLEN**

Gets number of available bytes in connection to be read

enumerator **LWESP_CMD_TCPIP_CIUPDATE**

Perform self-update

enumerator **LWESP_CMD_TCPIP_CIPSNTPCFG**

Configure SNTP servers

enumerator **LWESP_CMD_TCPIP_CIPSNTPCFG_GET**

Get SNTP config

enumerator **LWESP_CMD_TCPIP_CIPSNTPTIME**

Get current time using SNTP

enumerator **LWESP_CMD_TCPIP_CIPSNTPINTV**

Set the SNTP time synchronization interval

enumerator **LWESP_CMD_TCPIP_CIPSNTPINTV_GET**

Query the SNTP time synchronization interval

enumerator **LWESP_CMD_TCPIP_CIPDINFO**

Configure what data are received on +IPD statement

enumerator **LWESP_CMD_TCPIP_PING**

Ping domain

enumerator **LWESP_CMD_WIFI_SMART_START**

Start smart config

enumerator **LWESP_CMD_WIFI_SMART_STOP**

Stop smart config

enumerator **LWESP_CMD_WEBSERVER**

Start or Stop Web Server

enumerator **LWESP_CMD_BLEINIT_GET**

Get BLE status

enum **lwespr_t**

Result enumeration used across application functions.

Values:

enumerator **lwespOK** = 0

Function succeeded

enumerator **lwespOKIGNOREMORE**

Function succedded, should continue as lwespOK but ignore sending more data. This result is possible on connection data receive callback

enumerator **lwespERR**

General error

enumerator **lwespERRPAR**

Wrong parameters on function call

enumerator **lwespERRMEM**

Memory error occurred

enumerator **lwespTIMEOUT**

Timeout occurred on command

enumerator **lwespCONT**

There is still some command to be processed in current command

enumerator **lwespCLOSED**

Connection just closed

enumerator **lwespINPROG**

Operation is in progress

enumerator **lwespERRNOIP**

Station does not have IP address

enumerator **lwespERRNOFREECONN**

There is no free connection available to start

enumerator **lwespERRCONNTIMEOUT**

Timeout received when connection to access point

enumerator **lwespERRPASS**

Invalid password for access point

enumerator **lwespERRNOAP**

No access point found with specific SSID and MAC address

enumerator **lwespERRCONNFAIL**

Connection failed to access point

enumerator **lwespERRWIFINOTCONNECTED**

Wifi not connected to access point

enumerator **lwespERRNODEVICE**

Device is not present

enumerator **lwespERRBLOCKING**

Blocking mode command is not allowed

enumerator **lwespERRCMDNOTSUPPORTED**

Command is not supported error received by device, or when command is not supported in the stack itself

enum **lwesp_device_t**

List of support ESP devices by firmware.

Values:

enumerator **LWESP_DEVICE_UNKNOWN** = 0x00

Device is unknown by default

enumerator **LWESP_DEVICE_ESP8266**

Device is ESP8266

enumerator **LWESP_DEVICE_ESP32**

Device is ESP32

enumerator **LWESP_DEVICE_ESP32_C2**

Device is ESP32-C2

enumerator **LWESP_DEVICE_ESP32_C3**

Device is ESP32-C3

enumerator **LWESP_DEVICE_ESP32_C6**

Device is ESP32-C6

enumerator **LWESP_DEVICE_END**

End of the list

enum **lwesp_ecn_t**

List of encryptions of access point.

Values:

enumerator **LWESP_ECN_OPEN** = 0x00

No encryption on access point

enumerator **LWESP_ECN_WEP** = 0x01

WEP (Wired Equivalent Privacy) encryption

enumerator **LWESP_ECN_WPA_PSK** = 0x02

WPA (Wifi Protected Access) encryption

enumerator **LWESP_ECN_WPA2_PSK** = 0x03

WPA2 (Wifi Protected Access 2) encryption

enumerator **LWESP_ECN_WPA_WPA2_PSK** = 0x04

WPA/2 (Wifi Protected Access 1/2) encryption

enumerator **LWESP_ECN_WPA2_Enterprise** = 0x05

Enterprise encryption.

Note: ESP8266 is not able to connect to such device

---

enumerator **LWESP_ECN_WPA3_PSK** = 0x06

WPA3 (Wifi Protected Access 3) encryption

enumerator **LWESP_ECN_WPA2_WPA3_PSK** = 0x07

WPA2/3 (Wifi Protected Access 2/3) encryption

enumerator **LWESP_ECN_WAPI_PSK** = 0x08

WAPI PSK encryption mode

enumerator **LWESP_ECN_OWE** = 0x08

Opportunistic Wifi Encryption for end-to-end encryption

enumerator **LWESP_ECN_END**

Last entry

enum **lwesp_iptype_t**

IP type.

Values:

enumerator **LWESP_IPTYPE_V4** = 0x00

IP type is V4

enumerator **LWESP_IPTYPE_V6**

IP type is V6

enum **lwesp_mode_t**

List of possible WiFi modes.

Values:

enumerator **LWESP_MODE_NONE** = 0

Wifi RF IP disabled

enumerator **LWESP_MODE_STA** = 1

Set WiFi mode to station only

enumerator **LWESP_MODE_AP** = 2

Set WiFi mode to access point only

enumerator **LWESP_MODE_STA_AP** = 3

Set WiFi mode to station and access point

enum **lwesp_http_method_t**

List of possible HTTP methods.

Values:

enumerator **LWESP_HTTP_METHOD_GET**

HTTP method GET

enumerator **LWESP_HTTP_METHOD_HEAD**

HTTP method HEAD

enumerator **LWESP_HTTP_METHOD_POST**

HTTP method POST

enumerator **LWESP_HTTP_METHOD_PUT**
HTTP method PUT

enumerator **LWESP_HTTP_METHOD_DELETE**
HTTP method DELETE

enumerator **LWESP_HTTP_METHOD_CONNECT**
HTTP method CONNECT

enumerator **LWESP_HTTP_METHOD_OPTIONS**
HTTP method OPTIONS

enumerator **LWESP_HTTP_METHOD_TRACE**
HTTP method TRACE

enumerator **LWESP_HTTP_METHOD_PATCH**
HTTP method PATCH

enumerator **LWESP_HTTP_METHOD_END**

enum **lwesp_blocking_t**
API calls blocking or non-blocking type.
Values:

enumerator **LWESP_NON_BLOCKING** = 0
Blocking call

enumerator **LWESP_BLOCKING** = 1
Non-blocking call

struct **lwesp_conn_t**
#include <lwesp_private.h> Connection structure.

Public Members

lwesp_conn_type_t **type**
Connection type

uint8_t **num**
Connection number

lwesp_ip_t **remote_ip**
Remote IP address

lwesp_port_t **remote_port**

Remote port number

lwesp_port_t **local_port**

Local IP address

lwesp_evt_fn **evt_func**

Callback function for connection

void ***arg**

User custom argument

uint16_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.

lwesp_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**

Status whether connection is active

uint8_t **client**

Status whether connection is in client mode

uint8_t **data_received**

Status whether first data were received on connection

uint8_t **in_closing**

Status if connection is in closing mode. When in closing mode, ignore any possible received data from function

uint8_t **receive_blocked**

Status whether we should block manual receive for some time

uint8_t **receive_is_command_queued**

Status whether manual read command is in the queue already

struct *lwesp_conn_t*::[anonymous]::[anonymous] **f**

Connection flags

union *lwesp_conn_t*::[anonymous] **status**

Connection status union with flag bits

struct **lwesp_pbuf_t**

#include <lwesp_private.h> Packet buffer structure.

Public Members

struct lwesp_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

lwesp_ip_t **ip**

Remote address for received IPD data

lwesp_port_t **port**

Remote port for received IPD data

struct **lwesp_ipd_t**

#include <lwesp_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

lwesp_conn_p **conn**

Pointer to connection for network data

lwesp_ip_t **ip**

Remote IP address on from IPD data

lwesp_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

lwesp_pbuf_p **buff**

Pointer to data buffer used for receiving data

struct **lwesp_msg_t**

#include <lwesp_private.h> Message queue structure to share between threads.

Public Members

lwesp_cmd_t **cmd_def**

Default message type received from queue

lwesp_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

lwesp_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

lwespr_t **res**

Result of message operation

lwespr_t **res_err_code**

Result from "ERR CODE" received by AT command execution

lwespr_t (***fn**)(struct lwesp_msg*)

Processing callback function to process packet

uint32_t **delay**

Delay in units of milliseconds before executing first RESET command

struct *lwesp_msg_t*::[anonymous]::[anonymous] **reset**

Reset device

uint32_t **baudrate**

Baudrate for AT port

struct *lwesp_msg_t*::[anonymous]::[anonymous] **uart**

UART configuration

lwesp_mode_t **mode**

Mode of operation

lwesp_mode_t ***mode_get**

Get mode

struct *lwesp_msg_t*::[anonymous]::[anonymous] **wifi_mode**

When message type *LWESP_CMD_WIFI_CWMODE* is used

const char ***name**

AP name

const char ***pass**

AP password

const *lwesp_mac_t* ***mac**

Specific MAC address to use when connecting to AP

uint8_t **error_num**

Error number on connecting

struct *lwesp_msg_t*::[anonymous]::[anonymous] **sta_join**

Message for joining to access point

uint16_t **interval**

Interval in units of seconds

uint16_t **rep_cnt**

Repetition counter

struct *lwesp_msg_t*::[anonymous]::[anonymous] **sta_reconn_set**

Reconnect setup

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

Enable/Disable web server status

struct *lwesp_msg_t*::[anonymous]::[anonymous] **sta_autojoin**

Message for auto join procedure

lwesp_sta_info_ap_t ***info**

Information structure

struct *lwesp_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

lwesp_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 *lwesp_msg_t*::[anonymous]::[anonymous] **ap_list**

List for available access points to connect to

const char ***pwd**

Password of access point

lwesp_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 *lwesp_msg_t*::[anonymous]::[anonymous] **ap_conf**

Parameters to configure access point

lwesp_ap_conf_t ***ap_conf**

AP configuration

struct *lwesp_msg_t*::[anonymous]::[anonymous] **ap_conf_get**

Get the soft AP configuration

lwesp_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 *lwesp_msg_t*::[anonymous]::[anonymous] **sta_list**

List for stations connected to SoftAP

uint8_t **use_mac**

Status if specific MAC is to be used

lwesp_mac_t **mac**

MAC address to disconnect from access point

Pointer to MAC variable

struct *lwesp_msg_t*::[anonymous]::[anonymous] **ap_disconn_sta**

Disconnect station from access point

lwesp_ip_t ***ip**

Pointer to IP variable

lwesp_ip_t ***gw**

Pointer to gateway variable

lwesp_ip_t ***nm**

Pointer to netmask variable

struct *lwesp_msg_t*::[anonymous]::[anonymous] **sta_ap_getip**

Message for reading station or access point IP

lwesp_mac_t ***mac**

Pointer to MAC variable

struct *lwesp_msg_t*::[anonymous]::[anonymous] **sta_ap_getmac**

Message for reading station or access point MAC address

lwesp_ip_t **ip**

IP variable

lwesp_ip_t **gw**

Gateway variable

lwesp_ip_t **nm**

Netmask variable

struct *lwesp_msg_t*::[anonymous]::[anonymous] **sta_ap_setip**

Message for setting station or access point IP

struct *lwesp_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 *lwesp_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 *lwesp_msg_t*::[anonymous]::[anonymous] **wifi_hostname**

Set or get hostname structure

lwesp_conn_t ****conn**

Pointer to pointer to save connection used

const char ***remote_host**

Host to use for connection

lwesp_port_t **remote_port**

Remote port used for connection

Remote port address for UDP connection

lwesp_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

lwesp_port_t **udp_local_port**

UDP local port

void *arg

Connection custom argument

lwesp_evt_fn evt_func

Callback function to use on connection

uint8_t success

Status if connection AT+CIPSTART succedded

uint8_t ssl_auth

SSL authentication mode

uint8_t ssl_pki_num

SSL PKI number

uint8_t ssl_ca_num

SSL CA number

struct *lwesp_msg_t*::[anonymous]::[anonymous] conn_start

Structure for starting new connection

lwesp_conn_t *conn

Pointer to connection to close

Pointer to connection to send data

uint16_t val_id

Connection current validation ID when command was sent to queue

struct *lwesp_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 *lwesp_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 *lwesp_msg_t*::[anonymous]::[anonymous] **conn_send**

Structure to send data on connection

lwesp_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

lwesp_evt_fn **cb**

Server default callback function

struct *lwesp_msg_t*::[anonymous]::[anonymous] **tcpip_server**

Server configuration

size_t **size**

Size for SSL in uints of bytes

struct *lwesp_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 *lwesp_msg_t*::[anonymous]::[anonymous] **tcpip_ping**

Pinging structure

int16_t **tz**

Timezone setup

const char ***h1**

Optional server 1

const char ***h2**

Optional server 2

const char ***h3**

Optional server 3

struct *lwesp_msg_t*::[anonymous]::[anonymous] **tcpip_sntp_cfg**

SNTP configuration

uint8_t ***en**

Status if SNTP is enabled or not

int16_t ***tz**

Timezone setup

char ***h1**

Optional server 1

char ***h2**

Optional server 2

char ***h3**

Optional server 3

struct *lwesp_msg_t*::[anonymous]::[anonymous] **tcpip_sntp_cfg_get**

SNTP configuration read

uint32_t **interval**

Time in units of seconds

struct *lwesp_msg_t*::[anonymous]::[anonymous] **tcpip_sntp_intv**

SNTP interval configuration

uint32_t ***interval**

Pointer to write time to

struct *lwesp_msg_t*::[anonymous]::[anonymous] **tcpip_sntp_intv_get**

SNTP interval configuration

struct tm ***dt**

Pointer to datetime structure

struct *lwesp_msg_t*::[anonymous]::[anonymous] **tcpip_sntp_time**

SNTP get time

lwesp_ecn_t **min_ecn**

Minimum ECN level WPS will look for

struct *lwesp_msg_t*::[anonymous]::[anonymous] **wps_cfg**

WPS configuration

const char ***server**

mDNS server

struct *lwesp_msg_t*::[anonymous]::[anonymous] **mdns**

mDNS configuration

uint8_t **timeout**

Connection timeout

struct *lwesp_msg_t*::[anonymous]::[anonymous] **web_server**

Web Server 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 *lwesp_msg_t*::[anonymous]::[anonymous] **tcpip_ssl_cfg**

SSl configuration for connection

union *lwesp_msg_t*::[anonymous] **msg**

Group of different message contents

struct **lwesp_ip_mac_t**

#include <lwesp_private.h> IP and MAC structure with netmask and gateway addresses.

Public Members

lwesp_ip_t **ip**

IP address

lwesp_ip_t **gw**

Gateway address

lwesp_ip_t **nm**

Netmask address

lwesp_mac_t **mac**

MAC address

uint8_t **dhcp**

Flag indicating DHCP is enabled

uint8_t **has_ip**

Flag indicating IP is available

uint8_t **is_connected**

Flag indicating ESP is connected to wifi

struct *lwesp_ip_mac_t*::[anonymous] **f**

Flags structure

struct **lwesp_link_conn_t**

#include <lwesp_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

lwesp_conn_type_t **type**

Connection type

lwesp_ip_t **remote_ip**

Remote IP address

lwesp_port_t **remote_port**

Remote port

lwesp_port_t **local_port**

Local port number

struct **lwesp_evt_func_t**

#include <lwesp_private.h> Callback function linked list prototype.

Public Members

struct lwesp_evt_func ***next**

Next function in the list

lwesp_evt_fn **fn**

Function pointer itself

struct **lwesp_modules_t**

#include <lwesp_private.h> ESP modules structure.

Public Members

lwesp_device_t **device**

ESP device type

lwesp_sw_version_t **version_at**

Version of AT command software on ESP device

lwesp_sw_version_t **version_sdk**

Version of SDK used to build AT software

uint32_t **active_conns**

Bit field of currently active connections,

Todo:

: In case user has more than 32 connections, single variable is not enough

uint32_t **active_conns_last**

The same as previous but status before last check

lwesp_link_conn_t **link_conn**

Link connection handle

lwesp_ipd_t **ipd**

Connection incoming data structure

lwesp_conn_t **conns**[LWESP_CFG_MAX_CONNS]

Array of all connection structures

lwesp_ip_mac_t **sta**

Station IP and MAC addressed

lwesp_ip_mac_t **ap**

Access point IP and MAC addressed

struct tm **sntp_dt**

Data & time structure, used for automatic read request from the module, if feature enabled.

struct **lwesp_t**

#include <lwesp_private.h> ESP global structure.

Public Members

size_t **locked_cnt**

Counter how many times (recursive) stack is currently locked

lwesp_sys_sem_t **sem_sync**

Synchronization semaphore between threads

lwesp_sys_mbox_t **mbox_producer**

Producer message queue handle

lwesp_sys_mbox_t **mbox_process**

Consumer message queue handle

lwesp_sys_thread_t **thread_produce**

Producer thread handle

lwesp_sys_thread_t **thread_process**

Processing thread handle

lwesp_buff_t **buff**

Input processing buffer

lwesp_ll_t **ll**

Low level functions

lwesp_msg_t ***msg**

Pointer to current user message being executed

lwesp_evt_t **evt**

Callback processing structure

lwesp_evt_func_t ***evt_func**

Callback function linked list

lwesp_evt_fn **evt_server**

Default callback function for server connections

lwesp_modules_t **m**

All modules. When resetting, reset structure

uint8_t **initialized**

Flag indicating ESP library is initialized

uint8_t **dev_present**

Flag indicating if physical device is connected to host device

struct *lwesp_t*::[anonymous]::[anonymous] **f**

Flags structure

union *lwesp_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 **lwesp_esp_device_desc_t**

#include <lwesp_private.h> Physical device descriptor data.

This is used for library internal reasons

Public Members

lwesp_device_t **device**

Device identification

const char ***gmr_strid_1**

AT+GMR string identification option 1

const char ***gmr_strid_2**

AT+GMR string identification option 2

uint32_t **min_at_version**

Minimum Espressif official AT version for the module

struct **lwesp_ip4_addr_t**

#include <lwesp_types.h> IPv4 address structure.

Public Members

uint8_t **addr**[4]

IP address data

struct **lwesp_ip6_addr_t**

#include <lwesp_types.h> IPv6 address structure.

Public Members

uint16_t **addr**[8]

IP address data

struct **lwesp_ip_t**

#include <lwesp_types.h> IP structure.

Public Members

lwesp_ip4_addr_t **ip4**

IPv4 address

lwesp_ip6_addr_t **ip6**

IPv6 address

union *lwesp_ip_t*::[anonymous] **addr**

Actual IP address

lwesp_iptype_t **type**

IP type, either V4 or V6

struct **lwesp_mac_t**

#include <lwesp_types.h> MAC address.

Public Members

uint8_t **mac**[6]

MAC address

struct **lwesp_sw_version_t**

#include <lwesp_types.h> SW version handle object.

Format is (major << 24 | minor << 16 | patch << 8 | 0)

Public Members

uint32_t **version**

Version in single hex format

struct **lwesp_sta_ssid_pass_pair_t**

#include <lwesp_types.h> Simple helper structure for application purpose.

<>

User can define array of structure objects and set its preferred WIFI options, then trying to iterate through all and connect to first available

Note: This structure is not used by the LwESP library

Public Members

const char ***ssid**

SSID to connect to

const char ***pass**

Password for SSID

struct **lwesp_linbuff_t**

#include <lwesp_types.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 **LWESP_UNICODE**

Unicode support manager.

Functions

lwespr_t **lwespi_unicode_decode**(*lwesp_unicode_t* *uni, uint8_t ch)

Decode single character for unicode (UTF-8 only) format.

Parameters

- **s** – **[inout]** Pointer to unicode decode control structure
- **c** – **[in]** UTF-8 character sequence to test for device

Returns
lwespOK Function succedded, there is a valid UTF-8 sequence

Returns
lwespINPROG Function continues well but expects some more data to finish sequence

Returns
lwespERR Error in UTF-8 sequence

struct **lwesp_unicode_t**

#include <lwesp_types.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

lwespr_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 **LWESP_UTILS**

Utilities.

Defines

LWESP_ASSERT(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 *lwespr_t* enumeration

Parameters

- **c** – **[in]** Condition to test

LWESP_ASSERT0(c)

Assert an input parameter if in valid range, return 0 from function on failure.

Note: Since this is a macro, it may only be used on a functions where return status is of type *lwespr_t* enumeration

Parameters

- **c** – **[in]** Condition to test

LWESP_MEM_ALIGN(x)

Align `x` value to specific number of bytes, provided by *LWESP_CFG_MEM_ALIGNMENT* configuration.

Parameters

- **x** – **[in]** Input value to align

Returns

Input value aligned to specific number of bytes

LWESP_MIN(x, y)

Get minimal value between `x` and `y` inputs.

Parameters

- **x** – **[in]** First input to test
- **y** – **[in]** Second input to test

Returns

Minimal value between `x` and `y` parameters

LWESP_MAX(x, y)

Get maximal value between `x` and `y` inputs.

Parameters

- **x** – **[in]** First input to test
- **y** – **[in]** Second input to test

Returns

Maximal value between `x` and `y` parameters

LWESP_ARRAYSIZE(x)

Get size of statically declared array.

Parameters

- **x** – **[in]** Input array

Returns

Number of array elements

LWESP_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

- **x** – **[in]** Variable which is not used

LWESP_U32(x)

Get input value casted to `unsigned 32-bit` value.

Parameters

- **x** – **[in]** Input value

LWESP_U16(x)

Get input value casted to `unsigned 16-bit` value.

Parameters

- **x** – **[in]** Input value

LWESP_U8(x)

Get input value casted to `unsigned 8-bit` value.

Parameters

- **x** – **[in]** Input value

LWESP_I32(x)

Get input value casted to `signed 32-bit` value.

Parameters

- **x** – **[in]** Input value

LWESP_I16(x)

Get input value casted to `signed 16-bit` value.

Parameters

- **x** – **[in]** Input value

LWESP_I8(x)

Get input value casted to `signed 8-bit` value.

Parameters

- **x** – **[in]** Input value

LWESP_SZ(x)

Get input value casted to size_t value.

Parameters

- **x** – **[in]** Input value

lwesp_u32_to_str(num, out)

Convert unsigned 32-bit number to string.

Parameters

- **num** – **[in]** Number to convert
- **out** – **[out]** Output variable to save string

Returns
Pointer to output variable

lwesp_u32_to_hex_str(num, out, w)

Convert unsigned 32-bit number to HEX string.

Parameters

- **num** – **[in]** Number to convert
- **out** – **[out]** Output variable to save string
- **w** – **[in]** Width of output string. When number is shorter than width, leading 0 characters will apply

Returns
Pointer to output variable

lwesp_i32_to_str(num, out)

Convert signed 32-bit number to string.

Parameters

- **num** – **[in]** Number to convert
- **out** – **[out]** Output variable to save string

Returns
Pointer to output variable

lwesp_u16_to_str(num, out)

Convert unsigned 16-bit number to string.

Parameters

- **num** – **[in]** Number to convert
- **out** – **[out]** Output variable to save string

Returns
Pointer to output variable

lwesp_u16_to_hex_str(num, out, w)

Convert unsigned 16-bit number to HEX string.

Parameters

- **num** – **[in]** Number to convert
- **out** – **[out]** Output variable to save string

- **w** – **[in]** Width of output string. When number is shorter than width, leading `0` characters will apply.

Returns
Pointer to output variable

`lwesp_i16_to_str`(num, out)

Convert `signed 16-bit` number to string.

Parameters

- **num** – **[in]** Number to convert
- **out** – **[out]** Output variable to save string

Returns
Pointer to output variable

`lwesp_u8_to_str`(num, out)

Convert `unsigned 8-bit` number to string.

Parameters

- **num** – **[in]** Number to convert
- **out** – **[out]** Output variable to save string

Returns
Pointer to output variable

`lwesp_u8_to_hex_str`(num, out, w)

Convert `unsigned 16-bit` number to HEX string.

Parameters

- **num** – **[in]** Number to convert
- **out** – **[out]** Output variable to save string
- **w** – **[in]** Width of output string. When number is shorter than width, leading `0` characters will apply.

Returns
Pointer to output variable

`lwesp_i8_to_str`(num, out)

Convert `signed 8-bit` number to string.

Parameters

- **num** – **[in]** Number to convert
- **out** – **[out]** Output variable to save string

Returns
Pointer to output variable

Functions

char *lwesp_u32_to_gen_str(uint32_t num, char *out, uint8_t is_hex, uint8_t padding)

Convert unsigned 32-bit number to string.

Parameters

- **num** – **[in]** Number to convert
- **out** – **[out]** Output variable to save string
- **is_hex** – **[in]** Set to 1 to output hex, 0 otherwise
- **width** – **[in]** Width of output string. When number is shorter than width, leading 0 characters will apply. This parameter is valid only when formatting hex numbers

Returns

Pointer to output variable

char *lwesp_i32_to_gen_str(int32_t num, char *out)

Convert signed 32-bit number to string.

Parameters

- **num** – **[in]** Number to convert
- **out** – **[out]** Output variable to save string

Returns

Pointer to output variable

Web Server

Use ESP-AT's built-in web server feature to help WiFi provisioning and/or Firmware Over-the-Air update.

Note: Web Server is not enabled in ESP-AT by default. Refer to ESP-AT User Guide to build a custom image from source.

---

group LWESP_WEBSERVER

Web Server function.

Functions

lwespr_t lwesp_set_webserver(uint8_t en, *lwesp_port_t* port, uint16_t timeout, const *lwesp_api_cmd_evt_fn* evt_fn, void *const evt_arg, const uint32_t blocking)

Enables or disables Web Server.

Parameters

- **en** – **[in]** Set to 1 to enable web server, 0 to disable web server.
- **port** – **[in]** The web server port number.
- **timeout** – **[in]** The timeout for the every connection. Unit: second. Range:[21,60].
- **evt_fn** – **[in]** Callback function called when command has finished. Set to NULL when not used

- **evt_arg** – **[in]** Custom argument for event callback function
- **blocking** – **[in]** Status whether command should be blocking or not

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

Wi-Fi Protected Setup

group **LWESP_WPS**

WPS function on ESP device.

Functions

lwespr_t **lwesp_wps_set_config**(uint8_t en, *lwesp_ecn_t* min_ecn, const *lwesp_api_cmd_evt_fn* evt_fn, void *const evt_arg, const uint32_t blocking)

Configure WPS function on ESP device.

Note: WPS does not support WEP encryption

Parameters

- **en** – **[in]** Set to 1 to enable WPS or 0 to disable WPS
- **min_ecn** – **[in]** Minimum security level ESP will look for. It will not connect to the level below selected parameter
- **evt_fn** – **[in]** Callback function called when command has finished. Set to NULL when not used
- **evt_arg** – **[in]** Custom argument for event callback function
- **blocking** – **[in]** Status whether command should be blocking or not

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

group **LWESP**

Lightweight ESP-AT parser.

Defines

lwesp_set_fw_version(v, version_)

Set and format major, minor and patch values to firmware version.

Parameters

- **v** – **[in]** Version output, pointer to *lwesp_sw_version_t* structure
- **version_** – **[in]** Version in 32-bit format

Functions

lwespr_t **lwesp_init**(*lwesp_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 *LWESP_CFG_RESET_ON_INIT* is enabled, reset sequence will be sent to device otherwise manual call to *lwesp_reset* is required to setup device
- When *LWESP_CFG_RESTORE_ON_INIT* is enabled, restore sequence will be sent to device.

Parameters

- **evt_func** – **[in]** Global event callback function for all major events
- **blocking** – **[in]** Status whether command should be blocking or not. Used when *LWESP_CFG_RESET_ON_INIT* or *LWESP_CFG_RESTORE_ON_INIT* are enabled.

Returns

lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_reset**(const *lwesp_api_cmd_evt_fn* evt_fn, void *const evt_arg, const uint32_t blocking)

Execute reset and send default commands.

Parameters

- **evt_fn** – **[in]** Callback function called when command has finished. Set to `NULL` when not used
- **evt_arg** – **[in]** Custom argument for event callback function
- **blocking** – **[in]** Status whether command should be blocking or not

Returns

lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_reset_with_delay**(uint32_t delay, const *lwesp_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.

Parameters

- **delay** – **[in]** Number of milliseconds to wait before initiating first command to device
- **evt_fn** – **[in]** Callback function called when command has finished. Set to `NULL` when not used
- **evt_arg** – **[in]** Custom argument for event callback function
- **blocking** – **[in]** Status whether command should be blocking or not

Returns

lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_restore**(const *lwesp_api_cmd_evt_fn* evt_fn, void *const evt_arg, const uint32_t blocking)

Execute restore command and set module to default values.

Parameters

- **evt_fn** – **[in]** Callback function called when command has finished. Set to NULL when not used
- **evt_arg** – **[in]** Custom argument for event callback function
- **blocking** – **[in]** Status whether command should be blocking or not

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_set_at_baudrate**(uint32_t baud, const *lwesp_api_cmd_evt_fn* evt_fn, void *const evt_arg, const uint32_t blocking)

Sets baudrate of AT port (usually UART)

Parameters

- **baud** – **[in]** Baudrate in units of bits per second
- **evt_fn** – **[in]** Callback function called when command has finished. Set to NULL when not used
- **evt_arg** – **[in]** Custom argument for event callback function
- **blocking** – **[in]** Status whether command should be blocking or not

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_set_wifi_mode**(*lwesp_mode_t* mode, const *lwesp_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.

Parameters

- **mode** – **[in]** Mode of operation. This parameter can be a value of *lwesp_mode_t* enumeration
- **evt_fn** – **[in]** Callback function called when command has finished. Set to NULL when not used
- **evt_arg** – **[in]** Custom argument for event callback function
- **blocking** – **[in]** Status whether command should be blocking or not

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_get_wifi_mode**(*lwesp_mode_t* *mode, const *lwesp_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.

Parameters

- **mode** – **[in]** point to space of Mode to get. This parameter can be a pointer of *lwesp_mode_t* enumeration
- **evt_fn** – **[in]** Callback function called when command has finished. Set to NULL when not used
- **evt_arg** – **[in]** Custom argument for event callback function
- **blocking** – **[in]** Status whether command should be blocking or not

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_update_sw**(const *lwesp_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

Parameters

- **evt_fn** – **[in]** Callback function called when command has finished. Set to `NULL` when not used
- **evt_arg** – **[in]** Custom argument for event callback function
- **blocking** – **[in]** Status whether command should be blocking or not

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_core_lock**(void)

Lock stack from multi-thread access, enable atomic access to core.

If lock was `0` prior function call, lock is enabled and increased

Note: Function may be called multiple times to increase locks. Application must take care to call *lwesp_core_unlock* the same amount of time to make sure lock gets back to `0`

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_core_unlock**(void)

Unlock stack for multi-thread access.

Used in conjunction with *lwesp_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

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_device_set_present**(uint8_t present, const *lwesp_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

Parameters

- **present** – **[in]** Flag indicating device is present
- **evt_fn** – **[in]** Callback function called when command has finished. Set to `NULL` when not used

- **evt_arg** – **[in]** Custom argument for event callback function
- **blocking** – **[in]** Status whether command should be blocking or not

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

uint8_t **lwesp_device_is_present**(void)

Check if device is present.

Returns
1 on success, 0 otherwise

uint8_t **lwesp_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

Parameters
ms – **[in]** Milliseconds to delay

Returns
1 on success, 0 otherwise

lwespr_t **lwesp_get_current_at_fw_version**(*lwesp_sw_version_t* *const version)

Get current AT firmware version of connected Espressif device. It copies version from internal buffer to user variable, and is valid only if reset/restore operation is successful.

Parameters
version – **[out]** Output version variable

Returns
1 on success, 0 otherwise

lwespr_t **lwesp_get_min_at_fw_version**(*lwesp_sw_version_t* *const version)

Get minimal AT version required to run on Espressif device, to be well supported by LwESP library and to ensure proper compatibility and correct operation.

Parameters
version – **[out]** Version output, pointer to *lwesp_sw_version_t* structure

Returns
lwespOK on success, member of *lwespr_t* otherwise

lwesp_device_t **lwesp_device_get_device**(void)

Get currently connected Espressif device to AT port.

Returns
Member of *lwesp_device_t* enumeration

uint8_t **lwesp_device_is_device**(*lwesp_device_t* device)

Checks if connected device to the AT host is the one as requested as parameter check.

Parameters
device – Device type to check against

Returns
1 on success, 0 otherwise

uint8_t **lwesp_device_is_esp8266**(void)

Check if modem device is ESP8266.

Deprecated:

Use *lwesp_device_is_device* instead

Note: Function is only available if *LWESP_CFG_ESP8266* is enabled, otherwise it is defined as macro and evaluated to `0`

Returns

`1` on success, `0` otherwise

uint8_t **lwesp_device_is_esp32**(void)

Check if modem device is ESP32.

Deprecated:

Use *lwesp_device_is_device* instead

Note: Function is only available if *LWESP_CFG_ESP32* is enabled, otherwise it is defined as macro and evaluated to `0`

Returns

`1` on success, `0` otherwise

uint8_t **lwesp_device_is_esp32_c3**(void)

Check if modem device is ESP32-C3.

Deprecated:

Use *lwesp_device_is_device* instead

Note: Function is only available if *LWESP_CFG_ESP32_C3* is enabled, otherwise it is defined as macro and evaluated to `0`

Returns

`1` on success, `0` otherwise

5.3.2 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 `lwesp_opts.h` file.

Note: Check *Getting started* for guidelines on how to create and use configuration file.

group **LWESP_OPT**

ESP-AT options.

Defines

LWESP_CFG_ESP8266

Enables `1` or disables `0` support for ESP8266 AT commands.

LWESP_CFG_ESP32

Enables `1` or disables `0` support for ESP32 AT commands.

LWESP_CFG_ESP32_C2

Enables `1` or disables `0` support for ESP32-C2 AT commands.

LWESP_CFG_ESP32_C3

Enables `1` or disables `0` support for ESP32-C3 AT commands.

LWESP_CFG_ESP32_C6

Enables `1` or disables `0` support for ESP32-C6 AT commands.

LWESP_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

LWESP_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 *lwesp_mem_malloc*, *lwesp_mem_calloc*, *lwesp_mem_realloc* and *lwesp_mem_free*

Note: Function declaration follows standard C functions `malloc, calloc, realloc, free`. Declaration is available in `lwesp/lwesp_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

LWESP_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`

LWESP_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`.

LWESP_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.

LWESP_CFG_AT_PORT_BAUDRATE

Default baudrate used for AT port.

Note: User may call API function to change to desired baudrate if necessary

LWESP_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

LWESP_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

LWESP_CFG_ACCESS_POINT_STRUCT_FULL_FIELDS

Enables 1 or disables 0 full data info in *lwesp_ap_t* structure.

When enabled, advanced information is stored, and as a consequence, structure size is increased. Information such as scan type, min scan time, max scan time, frequency offset, frequency calibration are added

LWESP_CFG_KEEP_ALIVE

Enables 1 or disables 0 periodic keep-alive events to registered callbacks.

LWESP_CFG_KEEP_ALIVE_TIMEOUT

Timeout periodic time to trigger keep alive events to registered callbacks.

Feature must be enabled with *LWESP_CFG_KEEP_ALIVE*

LWESP_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 *LWESP_CFG_INPUT_USE_PROCESS* is enabled

---

LWESP_CFG_RESET_ON_INIT

Enables 1 or disables 0 reset sequence after *lwesp_init* call.

Note: When this functionality is disabled, user must manually call *lwesp_reset* to send reset sequence to ESP device.

LWESP_CFG_RESTORE_ON_INIT

Enables 1 or disables 0 device restore after *lwesp_init* call.

Note: When this feature is enabled, it will automatically restore and clear any settings stored as *default* in ESP device

---

LWESP_CFG_RESET_ON_DEVICE_PRESENT

Enables 1 or disables 0 reset sequence after *lwesp_device_set_present* call.

Note: When this functionality is disabled, user must manually call *lwesp_reset* to send reset sequence to ESP device.

LWESP_CFG_RESET_DELAY_DEFAULT

Default delay (milliseconds unit) before sending first AT command on reset sequence.

LWESP_CFG_MAX_SSID_LENGTH

Maximum length of SSID for access point scan.

Note: This parameter must include trailling zero

LWESP_CFG_MAX_PWD_LENGTH

Maximum length of PWD for access point.

Note: This parameter must include trailling zero

LWESP_CFG_LIST_CMD

Enables `1` or disables `0` listing all available CMDs during reset/restore operation.

Connection settings.

Defines

LWESP_CFG_IPV6

Enables `1` or disables `0` support for IPv6.

LWESP_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. Exception is UDP connection type, which can be controlled, with option *LWESP_CFG_CONN_ALLOW_FRAGMENTED_UDP_SEND*

LWESP_CFG_CONN_ALLOW_FRAGMENTED_UDP_SEND

Enables `1` or disables `0` support for fragmented send of UDP packets.

When connection type is UDP and packet length longer than maximal transmission unit, it can be split into multiple packets and sent over the network.

When this feature is disabled, max length of UDP packet is defined with *LWESP_CFG_CONN_MAX_DATA_LEN* option

LWESP_CFG_MAX_CONNS

Maximal number of connections AT software can support on ESP device.

Note: In case of official ESP-AT software, leave this on default value (`5`)

LWESP_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

LWESP_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/SSL` connections.

LWESP_CFG_CONN_MIN_DATA_LEN

Minimal buffer in bytes for connection receive allocation.

```
            Allocation will always start with (up to) \ref LWESP_CFG_CONN_MAX_
↪DATA_LEN
            and will continue with trial down to this setting up until␣
↪allocating is successful.
```

Note: This feature is used together with *LWESP_CFG_CONN_MANUAL_TCP_RECEIVE*

LWESP_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

LWESP_CFG_CONN_ALLOW_START_STATION_NO_IP

Enables (1) or disabled (1) option to start connection event if station does not have valid IP address (is not connected to another access point)

When enabled, starting a connection as a client can be successful even, if ESP-AT station isn't connected to another access point. This feature is only used if ESP is in access point mode and another station connects to it.

Note: Value is set to `0` to keep backward compatibility.

Debugging configurations.

Defines

LWESP_CFG_DBG

Set global debug support.

Possible values are *LWESP_DBG_ON* or *LWESP_DBG_OFF*

Note: Set to *LWESP_DBG_OFF* to globally disable all debugs

LWESP_CFG_DBG_OUT(fmt, ...)

Debugging output function.

Called with format and optional parameters for printf-like debug

LWESP_CFG_DBG_LVL_MIN

Minimal debug level.

Check LWESP_DBG_LVL for possible values

LWESP_CFG_DBG_TYPES_ON

Enabled debug types.

When debug is globally enabled with *LWESP_CFG_DBG* parameter, user must enable debug types such as TRACE or STATE messages.

Check LWESP_DBG_TYPE for possible options. Separate values with `bitwise OR` operator

LWESP_CFG_DBG_INIT

Set debug level for init function.

Possible values are *LWESP_DBG_ON* or *LWESP_DBG_OFF*

LWESP_CFG_DBG_MEM

Set debug level for memory manager.

Possible values are *LWESP_DBG_ON* or *LWESP_DBG_OFF*

LWESP_CFG_DBG_INPUT

Set debug level for input module.

Possible values are *LWESP_DBG_ON* or *LWESP_DBG_OFF*

LWESP_CFG_DBG_THREAD

Set debug level for ESP threads.

Possible values are *LWESP_DBG_ON* or *LWESP_DBG_OFF*

LWESP_CFG_DBG_ASSERT

Set debug level for asserting of input variables.

Possible values are *LWESP_DBG_ON* or *LWESP_DBG_OFF*

LWESP_CFG_DBG_IPD

Set debug level for incoming data received from device.

Possible values are *LWESP_DBG_ON* or *LWESP_DBG_OFF*

LWESP_CFG_DBG_NETCONN

Set debug level for netconn sequential API.

Possible values are *LWESP_DBG_ON* or *LWESP_DBG_OFF*

LWESP_CFG_DBG_PBUF

Set debug level for packet buffer manager.

Possible values are *LWESP_DBG_ON* or *LWESP_DBG_OFF*

LWESP_CFG_DBG_CONN

Set debug level for connections.

Possible values are *LWESP_DBG_ON* or *LWESP_DBG_OFF*

LWESP_CFG_DBG_VAR

Set debug level for dynamic variable allocations.

Possible values are *LWESP_DBG_ON* or *LWESP_DBG_OFF*

LWESP_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

Operating system dependant configuration.

Defines

LWESP_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

LWESP_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

LWESP_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 *lwesp_input_process* function

If this mode is not enabled, then user have to send every received byte via *lwesp_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 *LWESP_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

LWESP_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.

LWESP_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.

LWESP_CFG_THREADX_CUSTOM_MEM_BYTE_POOL

Enables `1` or disables `0` custom memory byte pool extension for ThreadX port.

When enabled, user must manually set byte pool at run-time, before *lwesp_init* is called

LWESP_CFG_THREADX_IDLE_THREAD_EXTENSION

Enables `1` or disables `0` idle thread extensions feature of ThreadX.

When enabled, user must manually configure idle thread and setup additional thread handle extension fields. By default ThreadX doesn't support self-thread cleanup when thread memory is dynamically allocated & thread terminated, hence another thread is mandatory to do the cleanup process instead.

This configuration does not create idle-thread, rather only sets additional TX_THREAD fields, indicating thread handle and thread stack are dynamically allocated.

Have a look at System-ThreadX port for implementation

Configuration of specific modules.

Defines

LWESP_CFG_DNS

Enables 1 or disables 0 support for DNS functions.

LWESP_CFG_WPS

Enables 1 or disables 0 support for WPS functions.

LWESP_CFG_SNTP

Enables 1 or disables 0 support for SNTP protocol with AT commands.

LWESP_CFG_SNTP_AUTO_READ_TIME_ON_UPDATE

Enables 1 or disables 0 automatic time read from the device when time gets updated.

Latest version of ESP-AT, starting from v3.0 supports, when enabled, to receive +TIME_UPDATED notification, when ESP device got new time via SNTP protocol.

When this option is enabled, command will be send to the ESP device requesting new time for each new TIME UPDATED event.

Note: *LWESP_CFG_SNTP* shall be enabled and SNTP configured on ESP device

LWESP_CFG_HOSTNAME

Enables 1 or disables 0 support for hostname with AT commands.

LWESP_CFG_FLASH

Enables 1 or disables 0 support for system flash with AT commands.

LWESP_CFG_PING

Enables 1 or disables 0 support for ping functions.

LWESP_CFG_MDNS

Enables 1 or disables 0 support for mDNS.

LWESP_CFG_SMART

Enables 1 or disables 0 support for SMART config.

LWESP_CFG_WEBSERVER

Enables 1 or disables 0 support for Web Server feature.

LWESP_CFG_BLE

Enables 1 or disables 0 support for Bluetooth Low Energy.

Note: This feature only works for some of Espressif devices, that support AT BLE commands

LWESP_CFG_BT

Enables 1 or disables 0 support for Bluetooth Classic.

Note: This feature only works for some of Espressif devices, that support AT BT commands

Configuration of netconn API module.

Defines

LWESP_CFG_NETCONN

Enables 1 or disables 0 NETCONN sequential API support for OS systems.

See also:

LWESP_CFG_OS

Note: To use this feature, OS support is mandatory.

LWESP_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.

LWESP_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

LWESP_CFG_NETCONN_RECEIVE_QUEUE_LEN

Receive queue length for pbuf entries.

Defines maximal number of pbuf data packet references for receive

Configuration of MQTT and MQTT API client modules.

Defines

LWESP_CFG_MQTT_MAX_REQUESTS

Maximal number of open MQTT requests at a time.

LWESP_CFG_MQTT_API_MBOX_SIZE

Size of MQTT API message queue for received messages.

LWESP_CFG_DBG_MQTT

Set debug level for MQTT client module.

Possible values are *LWESP_DBG_ON* or *LWESP_DBG_OFF*

LWESP_CFG_DBG_MQTT_API

Set debug level for MQTT API client module.

Possible values are *LWESP_DBG_ON* or *LWESP_DBG_OFF*

Configuration of Bluetooth Low Energy.

Configuration of Bluetooth Classic.

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

LWESP_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);
```

Parameters

- **dst** – **[in]** Destination memory start address
- **src** – **[in]** Source memory start address
- **len** – **[in]** Number of bytes to copy

Returns
Destination memory start address

LWESP_MEMSET(dst, b, len)

Memory set function declaration.

Function prototype must be similar to:

```
void *  my_memset(void* dst, int b, size_t len);
```

Parameters

- **dst** – **[in]** Destination memory start address
- **b** – **[in]** Value (byte) to set in memory
- **len** – **[in]** Number of bytes to set

Returns
Destination memory start address

Minimum AT versions needed for Espressif devices to run properly with LwESP.

Defines

LWESP_MIN_AT_VERSION_ESP8266

LWESP_MIN_AT_VERSION_ESP32

LWESP_MIN_AT_VERSION_ESP32_C2

LWESP_MIN_AT_VERSION_ESP32_C3

LWESP_MIN_AT_VERSION_ESP32_C6

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 **LWESP_LL**

Low-level communication functions.

Typedefs

typedef size_t (*`lwesp_ll_send_fn`)(const void *data, size_t len)

Function prototype for AT output data.

Param data
[in] Pointer to data to send. This parameter can be set to `NULL`, indicating to the low-level that (if used) DMA could be started to transmit data to the device

Param len
[in] 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

Return
Number of bytes sent

typedef uint8_t (*`lwesp_ll_reset_fn`)(uint8_t state)

Function prototype for hardware reset of ESP device.

Param state
[in] State indicating reset. When set to `1`, reset must be active (usually pin active low), or set to `0` when reset is cleared

Return
1 on successful action, `0` otherwise

Functions

lwespr_t `lwesp_ll_init`(*lwesp_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 *LWESP_CFG_INPUT_USE_PROCESS* is set to `1`, this function may be called from user UART thread.

Parameters
`ll` – **[inout]** Pointer to *lwesp_ll_t* structure to fill data for communication functions

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t `lwesp_ll_deinit`(*lwesp_ll_t* *ll)

Callback function to de-init low-level communication part.

Parameters
`ll` – **[inout]** Pointer to *lwesp_ll_t* structure to fill data for communication functions

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

struct **lwesp_ll_t**

#include <lwesp_types.h> Low level user specific functions.

Public Members

lwesp_ll_send_fn **send_fn**

Callback function to transmit data

lwesp_ll_reset_fn **reset_fn**

Reset callback function

uint32_t **baudrate**

UART baudrate value

struct *lwesp_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 **LWESP_SYS**

System based function for OS management, timings, etc.

Main

uint8_t **lwesp_sys_init**(void)

Init system dependant parameters.

After this function is called, all other system functions must be fully ready.

Returns

1 on success, `0` otherwise

uint32_t **lwesp_sys_now**(void)

Get current time in units of milliseconds.

Returns
Current time in units of milliseconds

uint8_t **lwesp_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.

Returns
1 on success, 0 otherwise

uint8_t **lwesp_sys_unprotect**(void)

Unprotect middleware core.

This function must follow number of calls of *lwesp_sys_protect* and unlock access only when counter reached back zero.

Note: Most operating systems support recursive mutexes.

Returns
1 on success, 0 otherwise

Mutex

uint8_t **lwesp_sys_mutex_create**(*lwesp_sys_mutex_t* *p)

Create new recursive mutex.

Note: Recursive mutex has to be created as it may be locked multiple times before unlocked

Parameters
p – **[out]** Pointer to mutex structure to allocate

Returns
1 on success, 0 otherwise

uint8_t **lwesp_sys_mutex_delete**(*lwesp_sys_mutex_t* *p)

Delete recursive mutex from system.

Parameters
p – **[in]** Pointer to mutex structure

Returns
1 on success, 0 otherwise

uint8_t **lwesp_sys_mutex_lock**(*lwesp_sys_mutex_t* *p)

Lock recursive mutex, wait forever to lock.

Parameters
p – **[in]** Pointer to mutex structure

Returns
1 on success, 0 otherwise

uint8_t **lwesp_sys_mutex_unlock**(*lwesp_sys_mutex_t* *p)

Unlock recursive mutex.

Parameters
p – **[in]** Pointer to mutex structure

Returns
1 on success, 0 otherwise

uint8_t **lwesp_sys_mutex_isvalid**(*lwesp_sys_mutex_t* *p)

Check if mutex structure is valid system.

Parameters
p – **[in]** Pointer to mutex structure

Returns
1 on success, 0 otherwise

uint8_t **lwesp_sys_mutex_invalid**(*lwesp_sys_mutex_t* *p)

Set recursive mutex structure as invalid.

Parameters
p – **[in]** Pointer to mutex structure

Returns
1 on success, 0 otherwise

Semaphores

uint8_t **lwesp_sys_sem_create**(*lwesp_sys_sem_t* *p, uint8_t cnt)

Create a new binary semaphore and set initial state.

Note: Semaphore may only have 1 token available

Parameters

- **p** – **[out]** Pointer to semaphore structure to fill with result
- **cnt** – **[in]** Count indicating default semaphore state: 0: Take semaphore token immediately 1: Keep token available

Returns
1 on success, 0 otherwise

uint8_t **lwesp_sys_sem_delete**(*lwesp_sys_sem_t* *p)

Delete binary semaphore.

Parameters
p – **[in]** Pointer to semaphore structure

Returns
1 on success, `0` otherwise

uint32_t **lwesp_sys_sem_wait**(*lwesp_sys_sem_t* *p, uint32_t timeout)

Wait for semaphore to be available.

Parameters

- **p** – **[in]** Pointer to semaphore structure
- **timeout** – **[in]** Timeout to wait in milliseconds. When `0` is applied, wait forever

Returns
Number of milliseconds waited for semaphore to become available or *LWESP_SYS_TIMEOUT* if not available within given time

uint8_t **lwesp_sys_sem_release**(*lwesp_sys_sem_t* *p)

Release semaphore.

Parameters
p – **[in]** Pointer to semaphore structure

Returns
1 on success, `0` otherwise

uint8_t **lwesp_sys_sem_isvalid**(*lwesp_sys_sem_t* *p)

Check if semaphore is valid.

Parameters
p – **[in]** Pointer to semaphore structure

Returns
1 on success, `0` otherwise

uint8_t **lwesp_sys_sem_invalid**(*lwesp_sys_sem_t* *p)

Invalid semaphore.

Parameters
p – **[in]** Pointer to semaphore structure

Returns
1 on success, `0` otherwise

Message queues

uint8_t **lwesp_sys_mbox_create**(*lwesp_sys_mbox_t* *b, size_t size)

Create a new message queue with entry type of `void *`

Parameters

- **b** – **[out]** Pointer to message queue structure
- **size** – **[in]** Number of entries for message queue to hold

Returns
1 on success, `0` otherwise

uint8_t **lwesp_sys_mbox_delete**(*lwesp_sys_mbox_t* *b)

Delete message queue.

Parameters
b – **[in]** Pointer to message queue structure

Returns
1 on success, `0` otherwise

uint32_t **lwesp_sys_mbox_put**(*lwesp_sys_mbox_t* *b, void *m)

Put a new entry to message queue and wait until memory available.

Parameters

- **b** – **[in]** Pointer to message queue structure
- **m** – **[in]** Pointer to entry to insert to message queue

Returns
Time in units of milliseconds needed to put a message to queue

uint32_t **lwesp_sys_mbox_get**(*lwesp_sys_mbox_t* *b, void **m, uint32_t timeout)

Get a new entry from message queue with timeout.

Parameters

- **b** – **[in]** Pointer to message queue structure
- **m** – **[in]** Pointer to pointer to result to save value from message queue to
- **timeout** – **[in]** Maximal timeout to wait for new message. When `0` is applied, wait for unlimited time

Returns
Time in units of milliseconds needed to put a message to queue or *LWESP_SYS_TIMEOUT* if it was not successful

uint8_t **lwesp_sys_mbox_putnow**(*lwesp_sys_mbox_t* *b, void *m)

Put a new entry to message queue without timeout (now or fail)

Parameters

- **b** – **[in]** Pointer to message queue structure
- **m** – **[in]** Pointer to message to save to queue

Returns
1 on success, `0` otherwise

uint8_t **lwesp_sys_mbox_getnow**(*lwesp_sys_mbox_t* *b, void **m)

Get an entry from message queue immediately.

Parameters

- **b** – **[in]** Pointer to message queue structure
- **m** – **[in]** Pointer to pointer to result to save value from message queue to

Returns
1 on success, `0` otherwise

uint8_t **lwesp_sys_mbox_isvalid**(*lwesp_sys_mbox_t* *b)

Check if message queue is valid.

Parameters
b – **[in]** Pointer to message queue structure

Returns
1 on success, 0 otherwise

uint8_t **lwesp_sys_mbox_invalid**(*lwesp_sys_mbox_t* *b)

Invalid message queue.

Parameters
b – **[in]** Pointer to message queue structure

Returns
1 on success, 0 otherwise

Threads

uint8_t **lwesp_sys_thread_create**(*lwesp_sys_thread_t* *t, const char *name, *lwesp_sys_thread_fn* thread_func, void *const arg, size_t stack_size, *lwesp_sys_thread_prio_t* prio)

Create a new thread.

Parameters

- **t** – **[out]** Pointer to thread identifier if create was successful. It may be set to NULL
- **name** – **[in]** Name of a new thread
- **thread_func** – **[in]** Thread function to use as thread body
- **arg** – **[in]** Thread function argument
- **stack_size** – **[in]** Size of thread stack in uints of bytes. If set to 0, reserve default stack size
- **prio** – **[in]** Thread priority

Returns
1 on success, 0 otherwise

uint8_t **lwesp_sys_thread_terminate**(*lwesp_sys_thread_t* *t)

Terminate thread (shut it down and remove)

Parameters
t – **[in]** Pointer to thread handle to terminate. If set to NULL, terminate current thread (thread from where function is called)

Returns
1 on success, 0 otherwise

uint8_t **lwesp_sys_thread_yield**(void)

Yield current thread.

Returns
1 on success, 0 otherwise

Defines

LWESP_SYS_MUTEX_NULL

Mutex invalid value.

Value assigned to *lwesp_sys_mutex_t* type when it is not valid.

LWESP_SYS_SEM_NULL

Semaphore invalid value.

Value assigned to *lwesp_sys_sem_t* type when it is not valid.

LWESP_SYS_MBOX_NULL

Message box invalid value.

Value assigned to *lwesp_sys_mbox_t* type when it is not valid.

LWESP_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

LWESP_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 proper operation.

LWESP_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 (***lwesp_sys_thread_fn**)(void*)

Thread function prototype.

typedef void ***lwesp_sys_mutex_t**

System mutex type.

It is used by middleware as base type of mutex.

typedef void ***lwesp_sys_sem_t**

System semaphore type.

It is used by middleware as base type of mutex.

typedef void *lwesp_sys_mbox_t

System message queue type.

It is used by middleware as base type of mutex.

typedef void *lwesp_sys_thread_t

System thread ID type.

typedef int lwesp_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

Warning: doxygengroup: Cannot find group "LWESP_APP_CAYENNE_API" in doxygen xml output for project "lwesp" from directory: _build/xml/

HTTP Server

group LWESP_APP_HTTP_SERVER

HTTP server based on callback API.

Defines

HTTP_MAX_HEADERS

Maximal number of headers we can control.

lwesp_http_server_write_string(hs, str)

Write string to HTTP server output.

See also:

lwesp_http_server_write

Note: May only be called from SSI callback function

Parameters

- **hs** – **[in]** HTTP handle
- **str** – **[in]** String to write

Returns

Number of bytes written to output

Typedefs

typedef char *(*`http_cgi_fn`)(*http_param_t* *params, size_t params_len)

CGI callback function.

Param params
[in] Pointer to list of parameteres and their values

Param params_len
[in] Number of parameters

Return
Function must return a new URI which is used later as response string, such as "/index.html" or similar

typedef *lwespr_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.

Param hs
[in] HTTP state

Param uri
[in] POST request URI

Param content_length
[in] Total content length (Content-Length HTTP parameter) in units of bytes

Return
lwespOK on success, member of *lwespr_t* otherwise

typedef *lwespr_t* (*`http_post_data_fn`)(struct http_state *hs, *lwesp_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

---

Param hs
[in] HTTP state

Param pbuf
[in] Packet buffer wit reciveed data

Return
lwespOK on success, member of *lwespr_t* otherwise

typedef *lwespr_t* (*`http_post_end_fn`)(struct http_state *hs)

End of POST data request function prototype.

Param hs
[in] HTTP state

Return
lwespOK on success, member of *lwespr_t* otherwise

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

Param hs
[in] HTTP state

Param tag_name
[in] Name of TAG to replace with user content

Param tag_len
[in] Length of TAG

Retval 1
Everything was written on this tag

Retval 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.

Param file
[in] Pointer to file where user has to set length of file if opening was successful

Param path
[in] Path of file to open

Return
1 if file is opened, 0 otherwise

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.

Param file
[in] File pointer to read content

Param buff
[in] Buffer to read data to. When parameter is set to NULL, number of remaining bytes available to read should be returned

Param btr
[in] Number of bytes to read from file. This parameter has no meaning when buff is NULL

Return
Number of bytes read or number of bytes available to read

typedef uint8_t (*http_fs_close_fn)(struct http_fs_file *file)

Close file callback function.

Param file
[in] File to close

Return
1 on success, 0 otherwise

Enums

enum **http_req_method_t**

Request method type.

Values:

enumerator **HTTP_METHOD_NOTALLOWED**

HTTP method is not allowed

enumerator **HTTP_METHOD_GET**

HTTP request method GET

enumerator **HTTP_METHOD_POST**

HTTP request method POST

enum **http_ssi_state_t**

List of SSI TAG parsing states.

Values:

enumerator **HTTP_SSI_STATE_WAIT_BEGIN** = 0x00

Waiting beginning of tag

enumerator **HTTP_SSI_STATE_BEGIN** = 0x01

Beginning detected, parsing it

enumerator **HTTP_SSI_STATE_TAG** = 0x02

Parsing TAG value

enumerator **HTTP_SSI_STATE_END** = 0x03

Parsing end of TAG

Functions

lwespr_t **lwesp_http_server_init**(const *http_init_t* *init, *lwesp_port_t* port)

Initialize HTTP server at specific port.

Parameters

- **init** – **[in]** Initialization structure for server
- **port** – **[in]** Port for HTTP server, usually 80

Returns
lwespOK on success, member of *lwespr_t* otherwise

size_t **lwesp_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

Parameters

- **hs** – **[in]** HTTP state
- **data** – **[in]** Data to write
- **len** – **[in]** Length of bytes to write

Returns
Number of bytes written

struct **http_param_t**

#include <*lwesp_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 <*lwesp_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 <*lwesp_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 <lwesp_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 <lwesp_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 <*lwesp_http_server.h*> HTTP state structure.

Public Members

lwesp_conn_p conn

Connection handle

lwesp_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 **rlwesp_file**

Response file structure

uint8_t **rlwesp_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 **LWESP_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.

Parameters

- **file** – **[in]** File structure to fill if file is successfully open
- **path** – **[in]** File path to open in format "/js/scripts.js" or "/index.html"

Returns

1 on success, 0 otherwise

uint32_t **http_fs_read**(*http_fs_file_t* *file, void *buff, size_t btr)

Read a file content.

Parameters

- **file** – **[in]** File handle to read
- **buff** – **[out]** Buffer to read data to. When set to NULL, function should return remaining available data to read
- **btr** – **[in]** Number of bytes to read. Has no meaning when buff = NULL

Returns

Number of bytes read or number of bytes available to read

uint8_t **http_fs_close**(*http_fs_file_t* *file)

Close a file handle.

Parameters

file – **[in]** File handle

Returns
1 on success, 0 otherwise

MQTT Client

MQTT client v3.1.1 implementation, based on callback (non-netconn) connection API.

Listing 24: MQTT application example code

```
/*
 * MQTT client example with ESP device using asynchronous callbacks
 *
 * 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 "lwesp_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 "lwesp/apps/lwesp_mqtt_client.h"
#include "lwesp/lwesp.h"
#include "lwesp/lwesp_timeout.h"
#include "mqtt_client.h"

static lwesp_mqtt_client_p  mqtt_client;    /*!< MQTT client structure */
static char mqtt_client_id[13];             /*!< Client ID is structured from ESP␣
↪station MAC address */

/**
 * \brief           Connection information for MQTT CONNECT packet
 */
static const lwesp_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 prv_mqtt_cb(lwesp_mqtt_client_p client, lwesp_mqtt_evt_t* evt);
static void prv_example_do_connect(lwesp_mqtt_client_p client);
static uint32_t retries = 0;

/**
 * \brief           Custom callback function for ESP events
 * \param[in]       evt: ESP event callback function
 */
static lwespr_t
prv_mqtt_lwesp_cb(lwesp_evt_t* evt) {
```

(continues on next page)

(continued from previous page)

```
    switch (lwesp_evt_get_type(evt)) {
#if LWESP_CFG_MODE_STATION
        case LWESP_EVT_WIFI_GOT_IP: {
            prv_example_do_connect(mqtt_client);/* Start connection after we have a␣
↪connection to network client */
            break;
        }
#endif /* LWESP_CFG_MODE_STATION */
        default:
            break;
    }
    return lwespOK;
}

/**
 * \brief           MQTT client thread
 * \param[in]       arg: User argument
 */
void
mqtt_client_thread(void const* arg) {
    lwesp_mac_t mac;

    LWESP_UNUSED(arg);

    /* Register new callback for general events from ESP stack */
    lwesp_evt_register(prv_mqtt_lwesp_cb);

    /* Get station MAC to format client ID */
    if (lwesp_sta_getmac(&mac, NULL, NULL, 1) == lwespOK) {
        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
     *
     * If station is already connected to access point,
     * try to connect immediately, otherwise it
     * will get connected from callback function instead
     */
    mqtt_client = lwesp_mqtt_client_new(256, 128);  /* Create new MQTT client */
    if (lwesp_sta_is_joined()) {                /* If ESP is already joined to network */
        prv_example_do_connect(mqtt_client);    /* Start connection to MQTT server */
    }

    /* Make dummy delay of thread */
```

(continues on next page)

(continued from previous page)

```
    while (1) {
        lwesp_delay(1000);
    }
}

/**
 * \brief           Timeout callback for MQTT events
 * \param[in]       arg: User argument
 */
static void
prv_mqtt_timeout_cb(void* arg) {
    static char tx_data[20];
    static uint32_t num = 10;
    lwesp_mqtt_client_p client = arg;
    lwespr_t res;

    if (lwesp_mqtt_client_is_connected(client)) {
        sprintf(tx_data, "R: %u, N: %u", (unsigned)retries, (unsigned)num);
        if ((res = lwesp_mqtt_client_publish(client, "lwesp_topic", tx_data, LWESP_
↪U16(strlen(tx_data)), LWESP_MQTT_QOS_EXACTLY_ONCE, 0, (void*)((uintptr_t)num))) ==␣
↪lwespOK) {
            printf("Publishing %d...\r\n", (int)num);
            num++;
        } else {
            printf("Cannot publish...: %d\r\n", (int)res);
        }
    }
    lwesp_timeout_add(10000, prv_mqtt_timeout_cb, arg);
}

/**
 * \brief           MQTT event callback function
 * \param[in]       client: MQTT client where event occurred
 * \param[in]       evt: Event type and data
 */
static void
prv_mqtt_cb(lwesp_mqtt_client_p client, lwesp_mqtt_evt_t* evt) {
    switch (lwesp_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 LWESP_MQTT_EVT_CONNECT: {            /* MQTT connect event occurred */
            lwesp_mqtt_conn_status_t status = lwesp_mqtt_client_evt_connect_get_
↪status(client, evt);

            if (status == LWESP_MQTT_CONN_STATUS_ACCEPTED) {
                printf("MQTT accepted!\r\n");
                /*
                 * Once we are accepted by server,
                 * it is time to subscribe to different topics
```

(continues on next page)

(continued from previous page)

```
                 * We will subscrive to "mqtt_lwesp_example_topic" topic,
                 * and will also set the same name as subscribe argument for callback later
                 */
                lwesp_mqtt_client_subscribe(client, "lwesp_topic", LWESP_MQTT_QOS_EXACTLY_ONCE, "lwesp_topic");

                /* Start timeout timer after 5000ms and call mqtt_timeout_cb function */
                lwesp_timeout_add(5000, prv_mqtt_timeout_cb, client);
            } else {
                printf("MQTT server connection was not successful: %d\r\n", (int)status);

                /* Try to connect all over again */
                prv_example_do_connect(client);
            }
            break;
        }

        /*
         * Subscribe event just happened.
         * Here it is time to check if it was successful or failed attempt
         */
        case LWESP_MQTT_EVT_SUBSCRIBE: {
            const char* arg = lwesp_mqtt_client_evt_subscribe_get_argument(client, evt); /* Get user argument */
            lwespr_t res = lwesp_mqtt_client_evt_subscribe_get_result(client, evt); /* Get result of subscribe event */

            if (res == lwespOK) {
                printf("Successfully subscribed to %s topic\r\n", arg);
                if (!strcmp(arg, "lwesp_topic")) {   /* Check topic name we were subscribed */
                    /* Subscribed to "lwesp_topic" topic */

                    /*
                     * Now publish an even on example topic
                     * and set QoS to minimal value which does not guarantee message delivery to received
                     */
                    lwesp_mqtt_client_publish(client, "lwesp_topic", "test_data", 9, LWESP_MQTT_QOS_AT_MOST_ONCE, 0, (void*)1);
                }
            }
            break;
        }

        /* Message published event occurred */
        case LWESP_MQTT_EVT_PUBLISH: {
            uint32_t val = (uint32_t)(uintptr_t)lwesp_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);
```

(continues on next page)

(continued from previous page)

```
            break;
        }

        /*
         * A new message was published to us
         * and now it is time to read the data
         */
        case LWESP_MQTT_EVT_PUBLISH_RECV: {
            const char* topic = lwesp_mqtt_client_evt_publish_recv_get_topic(client, evt);
            size_t topic_len = lwesp_mqtt_client_evt_publish_recv_get_topic_len(client, evt);
            const uint8_t* payload = lwesp_mqtt_client_evt_publish_recv_get_payload(client, evt);
            size_t payload_len = lwesp_mqtt_client_evt_publish_recv_get_payload_len(client, evt);

            LWESP_UNUSED(payload);
            LWESP_UNUSED(payload_len);
            LWESP_UNUSED(topic);
            LWESP_UNUSED(topic_len);
            break;
        }

        /* Client is fully disconnected from MQTT server */
        case LWESP_MQTT_EVT_DISCONNECT: {
            printf("MQTT client disconnected!\r\n");
            prv_example_do_connect(client);         /* Connect to server all over again */
            break;
        }

        default:
            break;
    }
}

/**
 * \brief           Make a connection to MQTT server in non-blocking mode
 * Act only if client ready to connect and not already connected
 */
static void
prv_example_do_connect(lwesp_mqtt_client_p client) {
    if (client == NULL
        || lwesp_mqtt_client_is_connected(client)) {
        return;
    }
    printf("Trying to connect to MQTT server\r\n");

    /*
     * Start a simple connection to open source
     * MQTT server on mosquitto.org
```

(continues on next page)

(continued from previous page)

```
     */
    retries++;
    lwesp_timeout_remove(prv_mqtt_timeout_cb);
    lwesp_mqtt_client_connect(mqtt_client, "test.mosquitto.org", 1883, prv_mqtt_cb, &
↪mqtt_client_info);
}
```

group **LWESP_APP_MQTT_CLIENT**

MQTT client.

Typedefs

typedef struct lwesp_mqtt_client ***lwesp_mqtt_client_p**

Pointer to lwesp_mqtt_client_t structure.

typedef void (***lwesp_mqtt_evt_fn**)(*lwesp_mqtt_client_p* client, *lwesp_mqtt_evt_t* *evt)

MQTT event callback function.

Param client
[in] MQTT client

Param evt
[in] MQTT event with type and related data

Enums

enum **lwesp_mqtt_qos_t**

Quality of service enumeration.

Values:

enumerator **LWESP_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

enumerator **LWESP_MQTT_QOS_AT_LEAST_ONCE** = 0x01

Delivery is quaranteed `at least once`, but it may be delivered multiple times with the same content

enumerator **LWESP_MQTT_QOS_EXACTLY_ONCE** = 0x02

Delivery is quaranteed `exactly once` = very critical packets such as billing informations or similar

enum **lwesp_mqtt_state_t**

State of MQTT client.

Values:

enumerator **LWESP_MQTT_CONN_DISCONNECTED** = 0x00

Connection with server is not established

enumerator **LWESP_MQTT_CONN_CONNECTING**

Client is connecting to server

enumerator **LWESP_MQTT_CONN_DISCONNECTING**

Client connection is disconnecting from server

enumerator **LWESP_MQTT_CONNECTING**

MQTT client is connecting… CONNECT command has been sent to server

enumerator **LWESP_MQTT_CONNECTED**

MQTT is fully connected and ready to send data on topics

enum **lwesp_mqtt_evt_type_t**

MQTT event types.

Values:

enumerator **LWESP_MQTT_EVT_CONNECT**

MQTT client connect event

enumerator **LWESP_MQTT_EVT_SUBSCRIBE**

MQTT client subscribed to specific topic

enumerator **LWESP_MQTT_EVT_UNSUBSCRIBE**

MQTT client unsubscribed from specific topic

enumerator **LWESP_MQTT_EVT_PUBLISH**

MQTT client publish message to server event.

Note: When publishing packet with quality of service *LWESP_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 = LWESP_MQTT_QOS_AT_MOST_ONCE`

---

enumerator **LWESP_MQTT_EVT_PUBLISH_RECV**

MQTT client received a publish message from server

enumerator **LWESP_MQTT_EVT_DISCONNECT**

MQTT client disconnected from MQTT server

enumerator **LWESP_MQTT_EVT_KEEP_ALIVE**

MQTT keep-alive event. It gets invoked after client and server exchange successful "keep-alive message", defined by MQTT protocol

enumerator **LWESP_MQTT_EVT_CONN_POLL**

Local ESP connection poll event. When connection is active, stack periodically sends polling events to user. This event is propagated to user MQTT space

enum **lwesp_mqtt_conn_status_t**

List of possible results from MQTT server when executing connect command.

Values:

enumerator **LWESP_MQTT_CONN_STATUS_ACCEPTED** = 0x00

Connection accepted and ready to use

enumerator **LWESP_MQTT_CONN_STATUS_REFUSED_PROTOCOL_VERSION** = 0x01

Connection Refused, unacceptable protocol version

enumerator **LWESP_MQTT_CONN_STATUS_REFUSED_ID** = 0x02

Connection refused, identifier rejected

enumerator **LWESP_MQTT_CONN_STATUS_REFUSED_SERVER** = 0x03

Connection refused, server unavailable

enumerator **LWESP_MQTT_CONN_STATUS_REFUSED_USER_PASS** = 0x04

Connection refused, bad user name or password

enumerator **LWESP_MQTT_CONN_STATUS_REFUSED_NOT_AUTHORIZED** = 0x05

Connection refused, not authorized

enumerator **LWESP_MQTT_CONN_STATUS_TCP_FAILED** = 0x100

TCP connection to server was not successful

Functions

lwesp_mqtt_client_p **lwesp_mqtt_client_new**(size_t tx_buff_len, size_t rx_buff_len)

Allocate a new MQTT client structure.

Parameters

- **tx_buff_len** – **[in]** Length of raw data output buffer
- **rx_buff_len** – **[in]** Length of raw data input buffer

Returns

Pointer to new allocated MQTT client structure or `NULL` on failure

void **lwesp_mqtt_client_delete**(*lwesp_mqtt_client_p* client)

Delete MQTT client structure.

Note: MQTT client must be disconnected first

Parameters
client – **[in]** MQTT client

lwespr_t **lwesp_mqtt_client_connect**(*lwesp_mqtt_client_p* client, const char *host, *lwesp_port_t* port, *lwesp_mqtt_evt_fn* evt_fn, const *lwesp_mqtt_client_info_t* *info)

Connect to MQTT server in non-blocking mode. Function returns immediately and does not wait for server to be connected.

Note: After TCP connection is established, CONNECT packet is automatically sent to server. Application must rely on events coming to event function, passed at connect stage

Parameters

- **client** – **[in]** MQTT client
- **host** – **[in]** Host address for server
- **port** – **[in]** Host port number
- **evt_fn** – **[in]** Callback function for all events on this MQTT client
- **info** – **[in]** Information structure for connection. It is used after connection is successfully established. Variable must not be a local or changes will be lost with potential faulty operation

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_mqtt_client_disconnect**(*lwesp_mqtt_client_p* client)

Disconnect from MQTT server.

Parameters
client – **[in]** MQTT client

Returns
lwespOK if request sent to queue or member of *lwespr_t* otherwise

uint8_t **lwesp_mqtt_client_is_connected**(*lwesp_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

Parameters
client – **[in]** MQTT client

Returns
1 on success, 0 otherwise

lwespr_t **lwesp_mqtt_client_subscribe**(*lwesp_mqtt_client_p* client, const char *topic, *lwesp_mqtt_qos_t* qos, void *arg)

Subscribe to MQTT topic.

Parameters

- **client** – **[in]** MQTT client

- **topic** – **[in]** Topic name to subscribe to
- **qos** – **[in]** Quality of service. This parameter can be a value of *lwesp_mqtt_qos_t*
- **arg** – **[in]** User custom argument used in callback

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_mqtt_client_unsubscribe**(*lwesp_mqtt_client_p* client, const char *topic, void *arg)

Unsubscribe from MQTT topic.

Parameters

- **client** – **[in]** MQTT client
- **topic** – **[in]** Topic name to unsubscribe from
- **arg** – **[in]** User custom argument used in callback

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_mqtt_client_publish**(*lwesp_mqtt_client_p* client, const char *topic, const void *payload, uint16_t len, *lwesp_mqtt_qos_t* qos, uint8_t retain, void *arg)

Publish a new message on specific topic.

Parameters

- **client** – **[in]** MQTT client
- **topic** – **[in]** Topic to send message to
- **payload** – **[in]** Message data
- **payload_len** – **[in]** Length of payload data
- **qos** – **[in]** Quality of service. This parameter can be a value of *lwesp_mqtt_qos_t* enumeration
- **retain** – **[in]** Retian parameter value
- **arg** – **[in]** User custom argument used in callback

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

void ***lwesp_mqtt_client_get_arg**(*lwesp_mqtt_client_p* client)

Get user argument on client.

Parameters
client – **[in]** MQTT client handle

Returns
User argument

void **lwesp_mqtt_client_set_arg**(*lwesp_mqtt_client_p* client, void *arg)

Set user argument on client.

Parameters

- **client** – **[in]** MQTT client handle
- **arg** – **[in]** User argument

struct **lwesp_mqtt_client_info_t**

#include <lwesp_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

lwesp_mqtt_qos_t **will_qos**

Will topic quality of service

uint8_t **use_ssl**

Connect to server using SSL connection with AT commands

struct **lwesp_mqtt_request_t**

#include <lwesp_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 **lwesp_mqtt_evt_t**

#include <lwesp_mqtt_client.h> MQTT event structure for callback function.

Public Members

lwesp_mqtt_evt_type_t **type**

Event type

lwesp_mqtt_conn_status_t **status**

Connection status with MQTT

struct *lwesp_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 *lwesp_mqtt_evt_t*::[anonymous]::[anonymous] **disconnect**

Event for disconnecting from server

void ***arg**

User argument for callback function

lwespr_t **res**

Response status

struct *lwesp_mqtt_evt_t*::[anonymous]::[anonymous] **sub_unsub_scribed**

Event for (un)subscribe to/from topics

struct *lwesp_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

lwesp_mqtt_qos_t qos

Received packet quality of service

struct *lwesp_mqtt_evt_t*::[anonymous]::[anonymous] publish_recv

Publish received event

union *lwesp_mqtt_evt_t*::[anonymous] evt

Event data parameters

group LWESP_APP_MQTT_CLIENT_EVT

Event helper functions.

Connect event

Note: Use these functions on *LWESP_MQTT_EVT_CONNECT* event

lwesp_mqtt_client_evt_connect_get_status(client, evt)

Get connection status.

Parameters

- client – **[in]** MQTT client
- evt – **[in]** Event handle

Returns

Connection status. Member of *lwesp_mqtt_conn_status_t*

Disconnect event

Note: Use these functions on *LWESP_MQTT_EVT_DISCONNECT* event

`lwesp_mqtt_client_evt_disconnect_is_accepted`(client, evt)

Check if MQTT client was accepted by server when disconnect event occurred.

Parameters

- **client** – **[in]** MQTT client
- **evt** – **[in]** Event handle

Returns

1 on success, `0` otherwise

Subscribe/unsubscribe event

Note: Use these functions on *LWESP_MQTT_EVT_SUBSCRIBE* or *LWESP_MQTT_EVT_UNSUBSCRIBE* events

`lwesp_mqtt_client_evt_subscribe_get_argument`(client, evt)

Get user argument used on *lwesp_mqtt_client_subscribe*.

Parameters

- **client** – **[in]** MQTT client
- **evt** – **[in]** Event handle

Returns

User argument

`lwesp_mqtt_client_evt_subscribe_get_result`(client, evt)

Get result of subscribe event.

Parameters

- **client** – **[in]** MQTT client
- **evt** – **[in]** Event handle

Returns

lwespOK on success, member of *lwespr_t* otherwise

`lwesp_mqtt_client_evt_unsubscribe_get_argument`(client, evt)

Get user argument used on *lwesp_mqtt_client_unsubscribe*.

Parameters

- **client** – **[in]** MQTT client

- **evt** – **[in]** Event handle

Returns
User argument

lwesp_mqtt_client_evt_unsubscribe_get_result(client, evt)

Get result of unsubscribe event.

Parameters

- **client** – **[in]** MQTT client
- **evt** – **[in]** Event handle

Returns
lwespOK on success, member of *lwespr_t* otherwise

Publish receive event

Note: Use these functions on *LWESP_MQTT_EVT_PUBLISH_RECV* event

lwesp_mqtt_client_evt_publish_recv_get_topic(client, evt)

Get topic from received publish packet.

Parameters

- **client** – **[in]** MQTT client
- **evt** – **[in]** Event handle

Returns
Topic name

lwesp_mqtt_client_evt_publish_recv_get_topic_len(client, evt)

Get topic length from received publish packet.

Parameters

- **client** – **[in]** MQTT client
- **evt** – **[in]** Event handle

Returns
Topic length

lwesp_mqtt_client_evt_publish_recv_get_payload(client, evt)

Get payload from received publish packet.

Parameters

- **client** – **[in]** MQTT client
- **evt** – **[in]** Event handle

Returns
Packet payload

`lwesp_mqtt_client_evt_publish_recv_get_payload_len`(client, evt)

Get payload length from received publish packet.

Parameters

- **client** – **[in]** MQTT client
- **evt** – **[in]** Event handle

Returns
Payload length

`lwesp_mqtt_client_evt_publish_recv_is_duplicate`(client, evt)

Check if packet is duplicated.

Parameters

- **client** – **[in]** MQTT client
- **evt** – **[in]** Event handle

Returns
1 if duplicated, 0 otherwise

`lwesp_mqtt_client_evt_publish_recv_get_qos`(client, evt)

Get received quality of service.

Parameters

- **client** – **[in]** MQTT client
- **evt** – **[in]** Event handle

Returns
Member of *lwesp_mqtt_qos_t* enumeration

Publish event

Note: Use these functions on *LWESP_MQTT_EVT_PUBLISH* event

`lwesp_mqtt_client_evt_publish_get_argument`(client, evt)

Get user argument used on *lwesp_mqtt_client_publish*.

Parameters

- **client** – **[in]** MQTT client
- **evt** – **[in]** Event handle

Returns
User argument

`lwesp_mqtt_client_evt_publish_get_result`(client, evt)

Get result of publish event.

Parameters

- **client** – **[in]** MQTT client

- **evt** – **[in]** Event handle

Returns
lwespOK on success, member of *lwespr_t* otherwise

Defines

lwesp_mqtt_client_evt_get_type(client, evt)
Get MQTT event type.

Parameters

- **client** – **[in]** MQTT client
- **evt** – **[in]** Event handle

Returns
MQTT Event type, value of *lwesp_mqtt_evt_type_t* enumeration

MQTT Client API

MQTT Client API provides sequential API built on top of *MQTT Client*.

Listing 25: MQTT API application example code

```
/*
 * MQTT client API example with ESP device to test server.
 * It utilizes sequential mode without callbacks in one user thread
 *
 * 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 "lwesp_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 "mqtt_client_api.h"
#include "lwesp/apps/lwesp_mqtt_client_api.h"
#include "lwesp/lwesp_mem.h"

/**
 * \brief           Connection information for MQTT CONNECT packet
 */
static const lwesp_mqtt_client_info_t mqtt_client_info = {
    .keep_alive = 10,

    /* Server login data */
    .user = "8a215f70-a644-11e8-ac49-e932ed599553",
    .pass = "26aa943f702e5e780f015cd048a91e8fb54cca28",

    /* Device identifier address */
```

(continues on next page)

(continued from previous page)

```
    .id = "869f5a20-af9c-11e9-b01f-db5cf74e7fb7",
};

static char mqtt_topic_str[256];  /*!< Topic string */
static char mqtt_topic_data[256]; /*!< Data string */

/**
 * \brief           Generate random number and write it to string
 * It utilizes simple pseudo random generator, super simple one
 * \param[out]      str: Output string with new number
 */
static void
prv_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
 * \param[in]       arg: User argument
 */
void
lwesp_mqtt_client_api_thread(void const* arg) {
    lwesp_mqtt_client_api_p client;
    lwesp_mqtt_conn_status_t conn_status;
    lwesp_mqtt_client_api_buf_p buf;
    lwespr_t res;
    char random_str[10];

    LWESP_UNUSED(arg);

    /* Create new MQTT API */
    if ((client = lwesp_mqtt_client_api_new(256, 128)) == NULL) {
        goto terminate;
    }

    while (1) {
        /* Make a connection */
        printf("Joining MQTT server\r\n");

        /* Try to join */
        conn_status = lwesp_mqtt_client_api_connect(client, "mqtt.mydevices.com", 1883, &
↪mqtt_client_info);
        if (conn_status == LWESP_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);
            lwesp_delay(5000);
            continue;
        }
```

(continues on next page)

(continued from previous page)

```

        /* Subscribe to topics */
        sprintf(mqtt_topic_str, "v1/%s/things/%s/cmd/#", mqtt_client_info.user, mqtt_
↪client_info.id);
        if (lwesp_mqtt_client_api_subscribe(client, mqtt_topic_str, LWESP_MQTT_QOS_AT_
↪LEAST_ONCE) == lwespOK) {
            printf("Subscribed to topic\r\n");
        } else {
            printf("Problem subscribing to topic!\r\n");
        }

        while (1) {
            /* Receive MQTT packet with 1000ms timeout */
            if ((res = lwesp_mqtt_client_api_receive(client, &buf, 5000)) == lwespOK) {
                if (buf != NULL) {
                    printf("Publish received!\r\n");
                    printf("Topic: %s, payload: %s\r\n", buf->topic, buf->payload);
                    lwesp_mqtt_client_api_buf_free(buf);
                    buf = NULL;
                }
            } else if (res == lwespCLOSED) {
                printf("MQTT connection closed!\r\n");
                break;
            } else if (res == lwespTIMEOUT) {
                printf("Timeout on MQTT receive function. Manually publishing.\r\n");

                /* Publish data on channel 1 */
                prv_generate_random(random_str);
                sprintf(mqtt_topic_str, "v1/%s/things/%s/data/1", mqtt_client_info.user,␣
↪mqtt_client_info.id);
                sprintf(mqtt_topic_data, "temp,c=%s", random_str);
                lwesp_mqtt_client_api_publish(client, mqtt_topic_str, mqtt_topic_data,␣
↪strlen(mqtt_topic_data),
                                              LWESP_MQTT_QOS_AT_LEAST_ONCE, 0);
            }
        }
        //goto terminate;
    }

terminate:
    lwesp_mqtt_client_api_delete(client);
    printf("MQTT client thread terminate\r\n");
    lwesp_sys_thread_terminate(NULL);
}
```

group **LWESP_APP_MQTT_CLIENT_API**

Sequential, single thread MQTT client API.

Typedefs

typedef struct lwesp_mqtt_client_api_buf *__lwesp_mqtt_client_api_buf_p__

Pointer to *lwesp_mqtt_client_api_buf_t* structure.

Functions

lwesp_mqtt_client_api_p __lwesp_mqtt_client_api_new__(size_t tx_buff_len, size_t rx_buff_len)

Create new MQTT client API.

Parameters

- **tx_buff_len** – **[in]** Maximal TX buffer for maximal packet length
- **rx_buff_len** – **[in]** Maximal RX buffer

Returns

Client handle on success, `NULL` otherwise

void __lwesp_mqtt_client_api_delete__(lwesp_mqtt_client_api_p client)

Delete client from memory.

Parameters

client – **[in]** MQTT API client handle

lwesp_mqtt_conn_status_t __lwesp_mqtt_client_api_connect__(lwesp_mqtt_client_api_p client, const char *host, *lwesp_port_t* port, const *lwesp_mqtt_client_info_t* *info)

Connect to MQTT broker.

Parameters

- **client** – **[in]** MQTT API client handle
- **host** – **[in]** TCP host
- **port** – **[in]** TCP port
- **info** – **[in]** MQTT client info

Returns

LWESP_MQTT_CONN_STATUS_ACCEPTED on success, member of *lwesp_mqtt_conn_status_t* otherwise

lwespr_t __lwesp_mqtt_client_api_close__(lwesp_mqtt_client_api_p client)

Close MQTT connection.

Parameters

client – **[in]** MQTT API client handle

Returns

lwespOK on success, member of *lwespr_t* otherwise

lwespr_t __lwesp_mqtt_client_api_subscribe__(lwesp_mqtt_client_api_p client, const char *topic, *lwesp_mqtt_qos_t* qos)

Subscribe to topic.

Parameters

- **client** – **[in]** MQTT API client handle

- **topic** – **[in]** Topic to subscribe on
- **qos** – **[in]** Quality of service. This parameter can be a value of *lwesp_mqtt_qos_t*

Returns
lwespOK on success, member of *lwespr_t* otherwise

lwespr_t **lwesp_mqtt_client_api_unsubscribe**(lwesp_mqtt_client_api_p client, const char *topic)

Unsubscribe from topic.

Parameters

- **client** – **[in]** MQTT API client handle
- **topic** – **[in]** Topic to unsubscribe from

Returns
lwespOK on success, member of *lwespr_t* otherwise

lwespr_t **lwesp_mqtt_client_api_publish**(lwesp_mqtt_client_api_p client, const char *topic, const void *data, size_t btw, *lwesp_mqtt_qos_t* qos, uint8_t retain)

Publish new packet to MQTT network.

Parameters

- **client** – **[in]** MQTT API client handle
- **topic** – **[in]** Topic to publish on
- **data** – **[in]** Data to send
- **btw** – **[in]** Number of bytes to send for data parameter
- **qos** – **[in]** Quality of service. This parameter can be a value of *lwesp_mqtt_qos_t*
- **retain** – **[in]** Set to 1 for retain flag, 0 otherwise

Returns
lwespOK on success, member of *lwespr_t* otherwise

uint8_t **lwesp_mqtt_client_api_is_connected**(lwesp_mqtt_client_api_p client)

Check if client MQTT connection is active.

Parameters
client – **[in]** MQTT API client handle

Returns
1 on success, 0 otherwise

lwespr_t **lwesp_mqtt_client_api_receive**(lwesp_mqtt_client_api_p client, *lwesp_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

---

Parameters

- **client** – **[in]** MQTT API client handle
- **p** – **[in]** Pointer to output buffer
- **timeout** – **[in]** Maximal time to wait before function returns timeout

Returns
lwespOK on success, *lwespCLOSED* if MQTT is closed, *lwespTIMEOUT* on timeout

void **lwesp_mqtt_client_api_buf_free**(*lwesp_mqtt_client_api_buf_p* p)

Free buffer memory after usage.

Parameters
p – **[in]** Buffer to free

struct **lwesp_mqtt_client_api_buf_t**

#include <*lwesp_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

lwesp_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 26: Netconn client example

```
/*
 * Netconn client demonstrates how to connect as a client to server
 * using sequential API from separate thread.
 *
 * it does not use callbacks to obtain connection status.
 *
 * Demo connects to NETCONN_HOST at NETCONN_PORT and sends GET request header,
 * then waits for respond and expects server to close the connection accordingly.
 */
#include "netconn_client.h"
#include "lwesp/lwesp.h"
#include "lwesp/lwesp_netconn.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) {
    lwespr_t res;
    lwesp_pbuf_p pbuf;
    lwesp_netconn_p client;
    lwesp_sys_sem_t* sem = (void*)arg;

    /* Make sure we are connected to access point first */
```

(continues on next page)

(continued from previous page)

```
    while (!lwesp_sta_has_ip()) {
        lwesp_delay(1000);
    }

    /*
     * First create a new instance of netconn
     * connection and initialize system message boxes
     * to accept received packet buffers
     */
    client = lwesp_netconn_new(LWESP_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 = lwesp_netconn_connect(client, NETCONN_HOST, NETCONN_PORT);
        if (res == lwespOK) { /* Are we successfully connected? */
            printf("Connected to " NETCONN_HOST "\r\n");
            res = lwesp_netconn_write(client, request_header, sizeof(request_header) - 1); /* Send data to server */
            if (res == lwespOK) {
                res = lwesp_netconn_flush(client); /* Flush data to output */
            }
            if (res == lwespOK) { /* Were data sent? */
                printf("Data were successfully sent to server\r\n");

                /*
                 * 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 = lwesp_netconn_receive(client, &pbuf);
                    if (res
                        == lwespCLOSED) { /* Was the connection closed? This can be checked by return status of receive function */
                        printf("Connection closed by remote side...\r\n");
                        break;
```

(continues on next page)

(continued from previous page)

```
                    } else if (res == lwespTIMEOUT) {
                        printf("Netconn timeout while receiving data. You may try multiple readings before deciding to "
                                "close manually\r\n");
                    }

                    if (res == lwespOK && 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, or memory leaks will appear
                         */
                        printf("Received new data packet of %d bytes\r\n", (int)lwesp_pbuf_length(pbuf, 1));
                        lwesp_pbuf_free_s(&pbuf); /* Free the memory after usage */
                    }
                } 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 != lwespCLOSED) {
                lwesp_netconn_close(client);
            }
        } else {
            printf("Cannot connect to remote host %s:%d!\r\n", NETCONN_HOST, NETCONN_PORT);
        }
        lwesp_netconn_delete(client); /* Delete netconn structure */
    }

    printf("Terminating thread\r\n");
    if (lwesp_sys_sem_isvalid(sem)) {
        lwesp_sys_sem_release(sem);
    }
    lwesp_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 27: Netconn server with single processing thread

```
/*
 * Netconn server example is based on single thread
 * and it listens for single client only on port 23.
 *
 * When new client connects, application processes client in the same thread.
 * When multiple clients get connected at the same time,
 * each of them waits all previous to be processed first, hence it may
 * introduce latency, in some cases even clearly visible in (for example) user browser
 */
#include "netconn_server_1thread.h"
#include "lwesp/lwesp_netconn.h"
#include "lwesp/lwesp.h"

/**
 * \brief           Basic thread for netconn server to test connections
 * \param[in]       arg: User argument
 */
void
netconn_server_1thread_thread(void* arg) {
    lwespr_t res;
    lwesp_netconn_p server, client;
    lwesp_pbuf_p p;

    LWESP_UNUSED(arg);
```

(continues on next page)

(continued from previous page)

```

    /* Create netconn for server */
    server = lwesp_netconn_new(LWESP_NETCONN_TYPE_TCP);
    if (server == NULL) {
        printf("Cannot create server netconn!\r\n");
    }

    /* Bind it to port 23 */
    res = lwesp_netconn_bind(server, 23);
    if (res != lwespOK) {
        printf("Cannot bind server\r\n");
        goto out;
    }

    /* Start listening for incoming connections with maximal 1 client */
    res = lwesp_netconn_listen_with_max_conn(server, 1);
    if (res != lwespOK) {
        goto out;
    }

    /* Unlimited loop */
    while (1) {
        /* Accept new client */
        res = lwesp_netconn_accept(server, &client);
        if (res != lwespOK) {
            break;
        }
        printf("New client accepted!\r\n");
        while (1) {
            /* Receive data */
            res = lwesp_netconn_receive(client, &p);
            if (res == lwespOK) {
                printf("Data received!\r\n");
                lwesp_pbuf_free_s(&p);
            } else {
                printf("Netconn receive returned: %d\r\n", (int)res);
                if (res == lwespCLOSED) {
                    printf("Connection closed by client\r\n");
                    break;
                }
            }
        }
        /* Delete client */
        if (client != NULL) {
            lwesp_netconn_delete(client);
            client = NULL;
        }
    }
    /* Delete client */
    if (client != NULL) {
        lwesp_netconn_delete(client);
        client = NULL;
```

(continues on next page)

(continued from previous page)

```
    }

out:
    printf("Terminating netconn thread!\r\n");
    if (server != NULL) {
        lwesp_netconn_delete(server);
    }
    lwesp_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 28: 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,
 * a new thread gets spawned and processes client request
 * separately. When multiple users are connected,
 * they can be processed simultaneously, hence no such latency as in single thread mode.
 *
 * As a drawback, more memory is consumed for multiple parallel threads being potentially
 * used at the same period of time.
 */
#include "netconn_server.h"
#include "lwesp/lwesp.h"
#include "lwesp/lwesp_netconn.h"

static void netconn_server_processing_thread(void* const arg);

/**
 * \brief           Main page response file
 */
```

(continues on next page)

(continued from previous page)

```
static const uint8_t rlwesp_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>"
    "       <p>Total system up time: <b>";

/**
 * \brief           Bottom part of main page
 */
static const uint8_t rlwesp_data_mainpage_bottom[] = ""
                                                     "       </b></p>"
                                                     "   </body>"
                                                     "</html>";

/**
 * \brief           Style file response
 */
static const uint8_t rlwesp_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 rlwesp_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) {
    lwespr_t res;
    lwesp_netconn_p server, client;

    LWESP_UNUSED(arg);

    /*
```

(continues on next page)

(continued from previous page)

```
     * First create a new instance of netconn
     * connection and initialize system message boxes
     * to accept clients and packet buffers
     */
    server = lwesp_netconn_new(LWESP_NETCONN_TYPE_TCP);
    if (server != NULL) {
        printf("Server netconn created\r\n");

        /* Bind network connection to port 80 */
        res = lwesp_netconn_bind(server, 80);
        if (res == lwespOK) {
            printf("Server netconn listens on port 80\r\n");
            /*
             * Start listening for incoming connections
             * on previously binded port
             */
            res = lwesp_netconn_listen(server);

            while (1) {
                /*
                 * Wait and accept new client connection
                 *
                 * Function will block thread until
                 * new client is connected to server
                 */
                res = lwesp_netconn_accept(server, &client);
                if (res == lwespOK) {
                    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 (lwesp_sys_thread_create(NULL, "client", (lwesp_sys_thread_
↪fn)netconn_server_processing_thread,
                                                client, 512, LWESP_SYS_THREAD_PRIO)) {
                        printf("Netconn client thread created\r\n");
                    } else {
                        printf("Netconn client thread creation failed!\r\n");

                        /* Force close & delete */
                        lwesp_netconn_close(client);
                        lwesp_netconn_delete(client);
                    }
                } else {
                    printf("Netconn connection accept error!\r\n");
                    break;
                }
            }
        } else {
            printf("Netconn server cannot bind to port\r\n");
```

(continues on next page)

(continued from previous page)

```
        }
    } else {
        printf("Cannot create server netconn\r\n");
    }

    printf("Terminating thread\r\n");
    lwesp_netconn_delete(server);     /* Delete netconn structure */
    lwesp_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) {
    lwesp_netconn_p client = arg;
    lwesp_pbuf_p pbuf, p = NULL;
    lwespr_t res;
    char strt[20];

    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 = lwesp_netconn_receive(client, &pbuf);

        if (res == lwespOK) {
            printf("Netconn data received, %d bytes\r\n", (int)lwesp_pbuf_length(pbuf,␣
↪1));
            /* Check reception of all header bytes */
            if (p == NULL) {
                p = pbuf; /* Set as first buffer */
            } else {
                lwesp_pbuf_cat(p, pbuf); /* Concatenate buffers together */
            }
            /*
             * Search for end of request section, that is supposed
             * to end with line, followed by another fully empty line.
             */
            if (lwesp_pbuf_strfind(pbuf, "\r\n\r\n", 0) != LWESP_SIZET_MAX) {
                if (lwesp_pbuf_strfind(pbuf, "GET / ", 0) != LWESP_SIZET_MAX) {
                    uint32_t now;
                    printf("Main page request\r\n");
                    now = lwesp_sys_now(); /* Get current time */
                    sprintf(strt, "%u ms; %d s", (unsigned)now, (unsigned)(now / 1000));
                    lwesp_netconn_write(client, rlwesp_data_mainpage_top, sizeof(rlwesp_
```

(continues on next page)

(continued from previous page)

```
↪data_mainpage_top) - 1);
                    lwesp_netconn_write(client, strt, strlen(strt));
                    lwesp_netconn_write(client, rlwesp_data_mainpage_bottom,␣
↪sizeof(rlwesp_data_mainpage_bottom) - 1);
                } else if (lwesp_pbuf_strfind(pbuf, "GET /style.css ", 0) != LWESP_SIZET_
↪MAX) {
                    printf("Style page request\r\n");
                    lwesp_netconn_write(client, rlwesp_data_style, sizeof(rlwesp_data_
↪style) - 1);
                } else {
                    printf("404 error not found\r\n");
                    lwesp_netconn_write(client, rlwesp_error_404, sizeof(rlwesp_error_
↪404) - 1);
                }
                lwesp_netconn_close(client); /* Close netconn connection */
                lwesp_pbuf_free_s(&p);       /* Do not forget to free memory after usage!
↪ */
                break;
            }
        }
    } while (res == lwespOK);

    if (p != NULL) { /* Free received data */
        lwesp_pbuf_free_s(&p);
    }
    lwesp_netconn_delete(client);     /* Destroy client memory */
    lwesp_sys_thread_terminate(NULL); /* Terminate this thread */
}
```

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, *lwesp_netconn_receive()* will block for maximal timeout set with *lwesp_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: *LWESP_CFG_NETCONN_RECEIVE_TIMEOUT* must be set to 1 to use this feature.

group **LWESP_NETCONN**

Network connection.

Defines

LWESP_NETCONN_RECEIVE_NO_WAIT

Receive data with no timeout.

Note: Used with *lwesp_netconn_set_receive_timeout* function

LWESP_NETCONN_FLAG_FLUSH

Immediate flush after netconn write

Typedefs

typedef struct lwesp_netconn ***lwesp_netconn_p**

Netconn object structure.

Enums

enum **lwesp_netconn_type_t**

Netconn connection type.

Values:

enumerator **LWESP_NETCONN_TYPE_TCP** = LWESP_CONN_TYPE_TCP

TCP connection

enumerator **LWESP_NETCONN_TYPE_SSL** = LWESP_CONN_TYPE_SSL

SSL connection

enumerator **LWESP_NETCONN_TYPE_UDP** = LWESP_CONN_TYPE_UDP

UDP connection

enumerator **LWESP_NETCONN_TYPE_TCPV6** = LWESP_CONN_TYPE_TCPV6

TCP connection over IPv6

enumerator **LWESP_NETCONN_TYPE_SSLV6** = LWESP_CONN_TYPE_SSLV6

SSL connection over IPv6

enumerator **LWESP_NETCONN_TYPE_UDPV6** = LWESP_CONN_TYPE_UDPV6

UDP connection over IPv6

Functions

lwesp_netconn_p **lwesp_netconn_new**(*lwesp_netconn_type_t* type)

Create new netconn connection.

Parameters
type – **[in]** Netconn connection type

Returns
New netconn connection on success, NULL otherwise

lwespr_t **lwesp_netconn_delete**(*lwesp_netconn_p* nc)

Delete netconn connection.

Parameters
nc – **[in]** Netconn handle

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_netconn_bind**(*lwesp_netconn_p* nc, *lwesp_port_t* port)

Bind a connection to specific port, can be only used for server connections.

Parameters

- **nc** – **[in]** Netconn handle
- **port** – **[in]** Port used to bind a connection to

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_netconn_connect**(*lwesp_netconn_p* nc, const char *host, *lwesp_port_t* port)

Connect to server as client.

Parameters

- **nc** – **[in]** Netconn handle
- **host** – **[in]** Pointer to host, such as domain name or IP address in string format
- **port** – **[in]** Target port to use

Returns
lwespOK if successfully connected, member of *lwespr_t* otherwise

lwespr_t **lwesp_netconn_receive**(*lwesp_netconn_p* nc, *lwesp_pbuf_p* *pbuf)

Receive data from connection.

Parameters

- **nc** – **[in]** Netconn handle used to receive from
- **pbuf** – **[in]** Pointer to pointer to save new receive buffer to. When function returns, user must check for valid pbuf value `pbuf != NULL`

Returns
lwespOK when new data ready

Returns
lwespCLOSED when connection closed by remote side

Returns
lwespTIMEOUT when receive timeout occurs

Returns
Any other member of *lwespr_t* otherwise

lwespr_t **lwesp_netconn_close**(*lwesp_netconn_p* nc)

Close a netconn connection.

Parameters
nc – **[in]** Netconn handle to close

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

int8_t **lwesp_netconn_get_connnum**(*lwesp_netconn_p* nc)

Get connection number used for netconn.

Parameters
nc – **[in]** Netconn handle

Returns
-1 on failure, connection number between 0 and *LWESP_CFG_MAX_CONNS* otherwise

lwesp_conn_p **lwesp_netconn_get_conn**(*lwesp_netconn_p* nc)

Get netconn connection handle.

Parameters
nc – **[in]** Netconn handle

Returns
ESP connection handle

lwesp_netconn_type_t **lwesp_netconn_get_type**(*lwesp_netconn_p* nc)

Get netconn connection type.

Parameters
nc – **[in]** Netconn handle

Returns
ESP connection type

void **lwesp_netconn_set_receive_timeout**(*lwesp_netconn_p* nc, uint32_t timeout)

Set timeout value for receiving data.

When enabled, *lwesp_netconn_receive* will only block for up to *timeout* value and will return if no new data within this time

Parameters

- **nc** – **[in]** Netconn handle
- **timeout** – **[in]** Timeout in units of milliseconds. Set to 0 to disable timeout feature. Function blocks until data receive or connection closed Set to > 0 to set maximum milliseconds to wait before timeout Set to *LWESP_NETCONN_RECEIVE_NO_WAIT* to enable non-blocking receive

uint32_t **lwesp_netconn_get_receive_timeout**(*lwesp_netconn_p* nc)

Get netconn receive timeout value.

Parameters
nc – **[in]** Netconn handle

Returns
Timeout in units of milliseconds. If value is 0, timeout is disabled (wait forever)

lwespr_t **lwesp_netconn_connect_ex**(*lwesp_netconn_p* nc, const char *host, *lwesp_port_t* port, uint16_t keep_alive, const char *local_ip, *lwesp_port_t* local_port, uint8_t mode)

Connect to server as client, allow keep-alive option.

Parameters

- **nc** – **[in]** Netconn handle
- **host** – **[in]** Pointer to host, such as domain name or IP address in string format
- **port** – **[in]** Target port to use
- **keep_alive** – **[in]** Keep alive period seconds
- **local_ip** – **[in]** Local ip in connected command
- **local_port** – **[in]** Local port address
- **mode** – **[in]** UDP mode

Returns

lwespOK if successfully connected, member of *lwespr_t* otherwise

lwespr_t **lwesp_netconn_listen**(*lwesp_netconn_p* nc)

Listen on previously binded connection.

Parameters

nc – **[in]** Netconn handle used to listen for new connections

Returns

lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_netconn_listen_with_max_conn**(*lwesp_netconn_p* nc, uint16_t max_connections)

Listen on previously binded connection with max allowed connections at a time.

Parameters

- **nc** – **[in]** Netconn handle used to listen for new connections
- **max_connections** – **[in]** Maximal number of connections server can accept at a time This parameter may not be larger than *LWESP_CFG_MAX_CONNS*

Returns

lwespOK on success, member of *lwespr_t* otherwise

lwespr_t **lwesp_netconn_set_listen_conn_timeout**(*lwesp_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 *lwesp_netconn_listen*

Parameters

- **nc** – **[in]** Netconn handle used for listen mode
- **timeout** – **[in]** Time in units of seconds. Set to `0` to disable timeout feature

Returns

lwespOK on success, member of *lwespr_t* otherwise

lwespr_t **lwesp_netconn_accept**(*lwesp_netconn_p* nc, *lwesp_netconn_p* *client)

Accept a new connection.

Parameters

- **nc** – **[in]** Netconn handle used as base connection to accept new clients
- **client** – **[out]** Pointer to netconn handle to save new connection to

Returns

lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_netconn_write**(*lwesp_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

Parameters

- **nc** – **[in]** Netconn handle used to write data to
- **data** – **[in]** Pointer to data to write
- **btw** – **[in]** Number of bytes to write

Returns

lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_netconn_write_ex**(*lwesp_netconn_p* nc, const void *data, size_t btw, uint16_t flags)

Extended version of *lwesp_netconn_write* with additional option to set custom flags.

Note: It is recommended to use this for full features support

Parameters

- **nc** – **[in]** Netconn handle used to write data to
- **data** – **[in]** Pointer to data to write
- **btw** – **[in]** Number of bytes to write
- **flags** – Bitwise-ORed set of flags for netconn. Flags start with LWESP_NETCONN_FLAG_xxx

Returns

lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_netconn_flush**(*lwesp_netconn_p* nc)

Flush buffered data on netconn TCP/SSL connection.

Note: This function may only be used on TCP/SSL connection

Parameters

nc – **[in]** Netconn handle to flush data

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_netconn_send**(*lwesp_netconn_p* nc, const void *data, size_t btw)

Send data on UDP connection to default IP and port.

Parameters

- **nc** – **[in]** Netconn handle used to send
- **data** – **[in]** Pointer to data to write
- **btw** – **[in]** Number of bytes to write

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

lwespr_t **lwesp_netconn_sendto**(*lwesp_netconn_p* nc, const *lwesp_ip_t* *ip, *lwesp_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

Parameters

- **nc** – **[in]** Netconn handle used to send
- **ip** – **[in]** Pointer to IP address
- **port** – **[in]** Port number used to send data
- **data** – **[in]** Pointer to data to write
- **btw** – **[in]** Number of bytes to write

Returns
lwespOK on success, member of *lwespr_t* enumeration otherwise

5.3.5 Command line interface

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

- **cliprintf** – **[in]** Pointer to CLI printf function
- **ch** – **[in]** new character to CLI

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.

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.

Param format
[in] string format

typedef void **cli_function**(*cli_printf* cliprintf, int argc, char **argv)

CLI entriy function.

Param cliprintf
[in] Printf handle callback

Param argc
[in] Number of arguments

Param argv
[in] Pointer to pointer to arguments

Functions

const *cli_command_t* ***cli_lookup_command**(char *command)

Find the CLI command that matches the input string.

Parameters
command – **[in]** pointer to command string for which we are searching

Returns
pointer of the command if we found a match, else NULL

void **cli_tab_auto_complete**(*cli_printf* cliprintf, char *cmd_buffer, uint32_t *cmd_pos, bool print_options)

CLI auto completion function.

Parameters

- **cliprintf** – **[in]** Pointer to CLI printf function
- **cmd_buffer** – **[in]** CLI command buffer
- **cmd_pos** – **[in]** pointer to current courser postion in command buffer
- **print_options** – **[in]** 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.

Parameters

- **commands** – **[in]** Pointer to commands table
- **num_of_commands** – **[in]** Number of new commands

Returns
true when new commands where succesfully added, else false

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 prepared and maintained for 2 platforms, but could be easily extended to more platforms:

- WIN32 examples, prepared as *CMake* projects, ready for *MSYS2 GCC compiler*
- ARM Cortex-M examples for STM32, prepared as STM32CubeIDE GCC projects. These are also supported in *Visual Studio Code* through *CMake* and *ninja* build system. Dedicated tutorial is available to get started in *VSCode*.

Note: Library is platform agnostic and can be used on many different products

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 CMake-ready and *VSCode*-ready. It utilizes CMake-presets feature to let you select the example and compile it directly.

- Make sure you have installed GCC compiler and is in env path (you can get it through MSYS2 packet manager)
- Install ninja and cmake and make them available in the path (you can get all through MSYS2 packet manager)
- Go to *examples win32* folder, open vscode there or run cmd: `cmake --preset <project name>` to configure cmake and later `cmake --build --preset <project name>` to compile the project

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 `/lwesp/src/system/lwesp_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

5.5 Update ESP AT firmware

LwESP is developed to match latest releases of official Espressif AT releases for various ESP devices.

Have a look in their AT firmware documentation pages to find out more.

5.6 Changelog

```
# Changelog

## Develop

- Change license year to 2022
- MQTT: Improve module implementation
- MQTT: Add optional SSL connection type
```

(continues on next page)

(continued from previous page)

```
- MQTT: Add cayenne async demo, publish-mode only through ring buffer
- MQTT CAYENNE: Completely reworked with asynchronous MQTT instead. Improves performance␣
↪to transmit more data in one shot
- MQTT client: Add poll periodic event for event callback
- Port: Improve ThreadX port
- CONN: Enable manual TCP receive by default, to improve system stability
- Timeout: module returns ERRMEM if no memory to allocate block
- Add esp_at_binaries from Espressif, used for library verification (official AT␣
↪firmware)
- Add optional `AT+CMD?` command at reset/restore process, for debug purpose for the␣
↪moment, only
- Add function to get ESP device used for AT command communication
- Fix `lwesp_get_min_at_fw_version` to return min AT version for detected ESP device
- SNTP: Improve module comments, change timezone variable to `int16_t`
- SNTP: Implement global callback when command is to obtain current time
- SNTP: Add synchronization interval config, available with ESP AT `2.3.0` or later␣
↪(ESP32-C3 only for the moment)
- SNTP: Add option for readin current SNTP configuration
- SNTP: Add option to automatically read SNTP data on `+TIME_UPDATED` event (requires␣
↪ESP-AT v3.x or newer)
- ERR: Add option to get response to `ERR CODE:` message if command doesn't exist and␣
↪put it to result of command execution
- Fix min at version for ESP32 to `2.2.0`
- Add `LWESP ` prefix for debug messages
- Update code style with astyle
- Add `.clang-format` draft - remove astyle support
- SSL: Added experimental support
- FS: Added support for erase and write operation
- Code improvement: Change multiple local variables to single structure
- Date&Time `lwesp_datetime_t` and use generic `struct tm` instead
- CONN: Add validation counter to ensure netconn object matches connection object and␣
↪that there was no connection close/re-open in between
- Minimum supported AT version is now `v3.2.0 (ESP32, ESP32-C3)` to support new MFG␣
↪write operations
- Added support for `ESP32-C2` and `ESP32-C6` AT commands
- SYSFLASH: Split System flash and Manufacturing data to separate commands, following␣
↪new breaking changes for ESP-AT firmware
- WPS: Break API compatibility to configure the feature, by adding minimum security␣
↪level parameter in the `lwesp_wps_set_config` function

## 1.1.2-dev

- Add POSIX-compliant low-level driver (thanks to community to implement it)
- Prohibit transmission of too long UDP packets (default), can be disabled with␣
↪configuration option
- Split CMakeLists.txt files between library and executable
- Move `esp_set_server` function to separate file `lwesp_server.c`
- Use `AT+GMR` command just after reset/restore to determine ESP device being connected␣
↪on AT port
- Minimum required AT binaries are now `2.3.0` for `ESP32/ESP32C3` and `2.2.1` for␣
↪`ESP8266`
- Connection status is acquired with `AT+CIPSTATE` or `AT+CIPSTATUS`, depends on␣
```

(continues on next page)

(continued from previous page)

```
↪Espressif connected device
- Add optional full fields for access point scan with `LWESP_CFG_ACCESS_POINT_STRUCT_
↪FULL_FIELDS` config option
- Add optional keep-alive periodic timeout to system event callback functions. Can be␣
↪used to act as generic timeout event
- Improve station manager snippet with asynchronous mode

## v1.1.1-dev

- Update to support library.json for Platform.IO

## v1.1.0-dev

- Add support for SDK v2.2
- Extend number of information received on AP scan
  - Add option for `WPA3` and `WPA2_WPA3_PSK` authentication modes
  - Add bgn and wps readings
- Add support for IPv6
- Add option to disconnect all stations from Soft-AP
- TODO: Add DNS for IPv6 support (Optional)
- TODO: Add support for WIFI GOT IP to parse IPv6
- Update CMSIS OS driver to support FreeRTOS aware kernel

## v1.0.0

- First stable release
- Works with *esp-at* version `v2.1.0`
- Implements all basic functionality for ESP8266 and ESP32
- Added operating system-based sequential API
- Other bug fixes and docs updates

## v0.6.1

- Fixed inadequate MQTT RX data handling causing possible overflow of memory
- Added support for zero-copy MQTT RX data

## v0.6.0

- Added support for ESP32 & ESP8266
- Official support for ESP32 AT firmware v1.2 & ESP8266 AT firmware v2.0
- Added examples to main repository
- Preparation for BLE support in ESP32
- Removed AT commands with `_CUR` and `_DEF` suffixes
- Renamed some event names, such as `ESP_EVT_CONN_CLOSE` instead of `ESP_EVT_CONN_CLOSED`
- Added DHCP/static IP support
- Added CMSIS-OS v2 support
- Added LwMEM port for dynamic memory allocation
- Other bug fixes

## v0.5.0

- Remove `_t` for every struct/enum name
```

(continues on next page)

(continued from previous page)

```
- Fully use ESP_MEMCPY instead of memcpy
- When connection is in closing mode, stop sending any new data and return with error
- Remove `_data` part from event helper function for connection receive
- Implement semaphores in internal threads
- Add driver for NUCLEO-F429
- Implement timeout callback for major events when device does not reply in given time
- Add callback function support to every API function which directly interacts with␣
↪device
- Replace all files to CRLF ending
- Replace `ESP_EVT_RESET` to `ESP_EVT_RESET_DETECTED`
- Replace `ESP_EVT_RESET_FINISH` to ESP_EVT_RESET`
- Replace all header files guards with ESP_HDR_ prefix
- Add espERRBLOCKING return when function is called in blocking mode when not allowed
- Other bug fixes to stabilize AT communication

## v0.4.0

- Add sizeof for every memory allocation
- Function typedefs suffix has been renamed to `_fn` instead of `_t`
- Merge events for connection data send and data send error
- Send callback if sending data is not successful in any case (timeout, ERROR, etc)
- Add functions for IP/port retrieval on connections
- Remove goto statements and use deep if statements
- Fix MQTT problems with username and password
- Make consistent variable types across library

## v1.3.0

- Rename all cb annotations with evt, replacing callbacks with events,
- Replace built-in memcpy and memset functions with `ESP_MEMCPY` and `ESP_MEMSET` to␣
↪allow users to do custom implementation
- Added example for Server RTOS
- Added API to unchain first pbuf in pbuf chain
- Implemented first prototype for manual TCP receive functionality.

## v0.2.0

- Fixed netconn issue with wrong data type for OS semaphore
- Added support for asynchronous reset
- Added support for tickless sleep for modern RTOS systems

## v0.1.0

- Initial release
```

5.7 Authors

List of authors and contributors to the library

```
Tilen Majerle <tilen.majerle@gmail.com>
Adrian Carpenter <adrian.carpenter@me.com>
Miha Cesnik <cesnik.91@gmail.com>
Evgeny Ermakov <evgeny.v.ermakov@gmail.com>
Michal Převrátil <michprev@gmail.com>
Evgeny Ermakov <>
Tom van der Geer <t.vandergeer@sping.nl>
Tilen Majerle <tilen@majerle.eu>
turmary <turmary@126.com>
Bert <mail@bertlammers.com>
niedong <niedong0816@126.com>
neo <xiongyu0523@gmail.com>
TakashiKusachi <aisiars@gmail.com>
imi415 <imi415.public@gmail.com>
lisekt84 <lisek84@interia.pl>
```

INDEX

L