
LwESP

Tilen MAJERLE

Sep 24, 2020

CONTENTS

1	Features	3
2	Requirements	5
3	Contribute	7
4	License	9
5	Table of contents	11
5.1	Getting started	11
5.2	User manual	13
5.3	API reference	67
5.4	Examples and demos	214
	Index	219

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](#)

FEATURES

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

REQUIREMENTS

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

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

LICENSE

MIT License

Copyright (c) 2020 Tilen MAJERLE

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

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

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

TABLE OF CONTENTS

5.1 Getting started

5.1.1 Download library

Library is primarily hosted on [Github](#).

- Download latest release from [releases area](#) on Github
- Clone *develop* branch for latest development

Download from releases

All releases are available on Github [releases area](#).

Clone from Github

First-time clone

- Download and install `git` if not already
- Open console and navigate to path in the system to clone repository to. Use command `cd your_path`
- Clone repository with one of available 3 options
 - Run `git clone --recurse-submodules https://github.com/MaJerle/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 master 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 resources repository is. Use command `cd your_path`
- Run `git pull origin master --recurse-submodules` command to pull latest changes and to fetch latest changes from submodules
- Run `git submodule foreach git pull origin master` to update & merge all submodules

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

5.1.2 Add library to project

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

- Copy `lwesp` folder to your project
- Add `lwesp/src/include` folder to *include path* of your toolchain
- Add port architecture `lwesp/src/include/system/port/_arch_` folder to *include path* of your toolchain
- Add source files from `lwesp/src/` folder to toolchain build
- Add source files from `lwesp/src/system/` folder to toolchain build for arch port
- 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

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

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 (or simply renamed in-place) where compiler include paths have access to it by using `#include "lwesp_opts.h"`.

Tip: Check *Configuration* section for possible configuration settings

Listing 1: Template options file

```
1  /**
2  * \file          lwesp_opts_template.h
3  * \brief        Template config file
4  */
5
```

(continues on next page)

(continued from previous page)

```

6  /*
7  * Copyright (c) 2020 Tilen MAJERLE
8  *
9  * Permission is hereby granted, free of charge, to any person
10 * obtaining a copy of this software and associated documentation
11 * files (the "Software"), to deal in the Software without restriction,
12 * including without limitation the rights to use, copy, modify, merge,
13 * publish, distribute, sublicense, and/or sell copies of the Software,
14 * and to permit persons to whom the Software is furnished to do so,
15 * subject to the following conditions:
16 *
17 * The above copyright notice and this permission notice shall be
18 * included in all copies or substantial portions of the Software.
19 *
20 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
21 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
22 * OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
23 * AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
24 * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
25 * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
26 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
27 * OTHER DEALINGS IN THE SOFTWARE.
28 *
29 * This file is part of LwESP - Lightweight ESP-AT parser library.
30 *
31 * Author:          Tilen MAJERLE <tilen@majerle.eu>
32 * Version:         $_version_$
33 */
34 #ifndef LWESP_HDR_OPTS_H
35 #define LWESP_HDR_OPTS_H
36
37 /* Rename this file to "lwesp_opts.h" for your application */
38
39 /*
40 * Open "include/lwesp/lwesp_opt.h" and
41 * copy & replace here settings you want to change values
42 */
43
44 #endif /* LWESP_HDR_OPTS_H */

```

5.2 User manual

5.2.1 Overview

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

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

Wifi connectivity is often part of bigger system with more powerful MCU. There is usually bigger MCU + Wifi transceiver (usually module) aside with UART/SPI communication. MCU handles application, such as display &

graphics, runs operating systems, drives motor and has additional external memories.

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 successfully 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 response 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 as well 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 excluding 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

```

1  /**
2   * \file          lwesp_netconn.c
3   * \brief         API functions for sequential calls
4   */
5
6  /*
7   * Copyright (c) 2020 Tilen MAJERLE
8   *
9   * Permission is hereby granted, free of charge, to any person
10  * obtaining a copy of this software and associated documentation
11  * files (the "Software"), to deal in the Software without restriction,
12  * including without limitation the rights to use, copy, modify, merge,
13  * publish, distribute, sublicense, and/or sell copies of the Software,
14  * and to permit persons to whom the Software is furnished to do so,
15  * subject to the following conditions:
16  *
17  * The above copyright notice and this permission notice shall be
18  * included in all copies or substantial portions of the Software.
19  *
20  * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
21  * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
22  * OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
23  * AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
24  * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
25  * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
26  * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
27  * OTHER DEALINGS IN THE SOFTWARE.
28  *
29  * This file is part of LwESP - Lightweight ESP-AT parser library.
30  */

```

(continues on next page)

(continued from previous page)

```

31  * Author:          Tilen MAJERLE <tilen@majerle.eu>
32  * Version:         $_version_$
33  */
34  #include "lwesp/lwesp_netconn.h"
35  #include "lwesp/lwesp_private.h"
36  #include "lwesp/lwesp_conn.h"
37  #include "lwesp/lwesp_mem.h"
38
39  #if LWESP_CFG_NETCONN || __DOXYGEN__
40
41  /* Check conditions */
42  #if LWESP_CFG_NETCONN_RECEIVE_QUEUE_LEN < 2
43  #error "LWESP_CFG_NETCONN_RECEIVE_QUEUE_LEN must be greater or equal to 2"
44  #endif /* LWESP_CFG_NETCONN_RECEIVE_QUEUE_LEN < 2 */
45
46  #if LWESP_CFG_NETCONN_ACCEPT_QUEUE_LEN < 2
47  #error "LWESP_CFG_NETCONN_ACCEPT_QUEUE_LEN must be greater or equal to 2"
48  #endif /* LWESP_CFG_NETCONN_ACCEPT_QUEUE_LEN < 2 */
49
50  /**
51   * \brief          Sequential API structure
52   */
53  typedef struct lwesp_netconn {
54      struct lwesp_netconn* next;                /*!< Linked list entry */
55
56      lwesp_netconn_type_t type;                /*!< Netconn type */
57      lwesp_port_t listen_port;                /*!< Port on which we are listening */
58
59      size_t rcv_packets;                      /*!< Number of received packets so_
↳far on this connection */
60      lwesp_conn_p conn;                      /*!< Pointer to actual connection */
61
62      lwesp_sys_mbox_t mbox_accept;           /*!< List of active connections_
↳waiting to be processed */
63      lwesp_sys_mbox_t mbox_receive;         /*!< Message queue for receive mbox */
64      size_t mbox_receive_entries;           /*!< Number of entries written to_
↳receive mbox */
65
66      lwesp_linbuff_t buff;                  /*!< Linear buffer structure */
67
68      uint16_t conn_timeout;                 /*!< Connection timeout in units of_
↳seconds when
69
↳mode.
70
↳closed if there is no
71
↳when timeout feature is disabled. */
72
73      #if LWESP_CFG_NETCONN_RECEIVE_TIMEOUT || __DOXYGEN__
74          uint32_t rcv_timeout;              /*!< Receive timeout in unit of_
↳milliseconds */
75      #endif
76  } lwesp_netconn_t;
77
78  static uint8_t rcv_closed = 0xFF, rcv_not_present = 0xFF;
79  static lwesp_netconn_t* listen_api;       /*!< Main connection in listening_
↳mode */

```

(continues on next page)

(continued from previous page)

```

80 static lwesp_netconn_t* netconn_list;          /*!< Linked list of netconn entries */
81
82 /**
83  * \brief          Flush all mboxes and clear possible used memories
84  * \param[in]     nc: Pointer to netconn to flush
85  * \param[in]     protect: Set to 1 to protect against multi-thread access
86  */
87 static void
88 flush_mboxes(lwesp_netconn_t* nc, uint8_t protect) {
89     lwesp_pbuf_p pbuf;
90     lwesp_netconn_t* new_nc;
91     if (protect) {
92         lwesp_core_lock();
93     }
94     if (lwesp_sys_mbox_isvalid(&nc->mbox_receive)) {
95         while (lwesp_sys_mbox_getnow(&nc->mbox_receive, (void**)&pbuf)) {
96             if (nc->mbox_receive_entries > 0) {
97                 --nc->mbox_receive_entries;
98             }
99             if (pbuf != NULL && (uint8_t*)pbuf != (uint8_t*)&recv_closed) {
100                 lwesp_pbuf_free(pbuf);          /* Free received data buffers */
101             }
102         }
103         lwesp_sys_mbox_delete(&nc->mbox_receive); /* Delete message queue */
104         lwesp_sys_mbox_invalid(&nc->mbox_receive); /* Invalid handle */
105     }
106     if (lwesp_sys_mbox_isvalid(&nc->mbox_accept)) {
107         while (lwesp_sys_mbox_getnow(&nc->mbox_accept, (void**)&new_nc)) {
108             if (new_nc != NULL
109                 && (uint8_t*)new_nc != (uint8_t*)&recv_closed
110                 && (uint8_t*)new_nc != (uint8_t*)&recv_not_present) {
111                 lwesp_netconn_close(new_nc);    /* Close netconn connection */
112             }
113         }
114         lwesp_sys_mbox_delete(&nc->mbox_accept); /* Delete message queue */
115         lwesp_sys_mbox_invalid(&nc->mbox_accept); /* Invalid handle */
116     }
117     if (protect) {
118         lwesp_core_unlock();
119     }
120 }
121
122 /**
123  * \brief          Callback function for every server connection
124  * \param[in]     evt: Pointer to callback structure
125  * \return        Member of \ref lwespr_t enumeration
126  */
127 static lwespr_t
128 netconn_evt(lwesp_evt_t* evt) {
129     lwesp_conn_p conn;
130     lwesp_netconn_t* nc = NULL;
131     uint8_t close = 0;
132
133     conn = lwesp_conn_get_from_evt(evt);        /* Get connection from event */
134     switch (lwesp_evt_get_type(evt)) {
135         /*
136          * A new connection has been active

```

(continues on next page)

(continued from previous page)

```

137     * and should be handled by netconn API
138     */
139     case LWESP_EVT_CONN_ACTIVE: {           /* A new connection active is active_
↪ */
140         if (lwesp_conn_is_client(conn)) { /* Was connection started by us? */
141             nc = lwesp_conn_get_arg(conn); /* Argument should be already set */
142             if (nc != NULL) {
143                 nc->conn = conn;         /* Save actual connection */
144             } else {
145                 close = 1;               /* Close this connection, invalid_
↪ netconn */
146             }
147
148             /* Is the connection server type and we have known listening API? */
149             } else if (lwesp_conn_is_server(conn) && listen_api != NULL) {
150                 /*
151                  * Create a new netconn structure
152                  * and set it as connection argument.
153                  */
154                 nc = lwesp_netconn_new(LWESP_NETCONN_TYPE_TCP); /* Create new API */
155                 LWESP_DEBUGW(LWESP_CFG_DBG_NETCONN | LWESP_DBG_TYPE_TRACE | LWESP_DBG_
↪ LVL_WARNING,
156                             nc == NULL, "[NETCONN] Cannot create new structure for_
↪ incoming server connection!\r\n");
157
158                 if (nc != NULL) {
159                     nc->conn = conn;     /* Set connection handle */
160                     lwesp_conn_set_arg(conn, nc); /* Set argument for connection */
161
162                     /*
163                      * In case there is no listening connection,
164                      * simply close the connection
165                      */
166                     if (!lwesp_sys_mbox_isvalid(&listen_api->mbox_accept)
167                         || !lwesp_sys_mbox_putnow(&listen_api->mbox_accept, nc)) {
168                         close = 1;
169                     }
170                 } else {
171                     close = 1;
172                 }
173             } else {
174                 LWESP_DEBUGW(LWESP_CFG_DBG_NETCONN | LWESP_DBG_TYPE_TRACE | LWESP_DBG_
↪ LVL_WARNING, listen_api == NULL,
175                             "[NETCONN] Closing connection as there is no listening API_
↪ in netconn!\r\n");
176                 close = 1;               /* Close the connection at this point_
↪ */
177             }
178
179             /* Decide if some events want to close the connection */
180             if (close) {
181                 if (nc != NULL) {
182                     lwesp_conn_set_arg(conn, NULL); /* Reset argument */
183                     lwesp_netconn_delete(nc); /* Free memory for API */
184                 }
185                 lwesp_conn_close(conn, 0); /* Close the connection */
186                 close = 0;

```

(continues on next page)

```

187     }
188     break;
189 }
190
191 /*
192  * We have a new data received which
193  * should have netconn structure as argument
194  */
195 case LWESP_EVT_CONN_RECV: {
196     lwesp_pbuf_p pbuf;
197
198     nc = lwesp_conn_get_arg(conn);      /* Get API from connection */
199     pbuf = lwesp_evt_conn_recv_get_buff(evt); /* Get received buff */
200
201 #if !LWESP_CFG_CONN_MANUAL_TCP_RECEIVE
202     lwesp_conn_recved(conn, pbuf);      /* Notify stack about received data */
203 #endif /* !LWESP_CFG_CONN_MANUAL_TCP_RECEIVE */
204
205     lwesp_pbuf_ref(pbuf);                /* Increase reference counter */
206     if (nc == NULL || !lwesp_sys_mbox_isvalid(&nc->mbox_receive)
207         || !lwesp_sys_mbox_putnow(&nc->mbox_receive, pbuf)) {
208         LWESP_DEBUGF(LWESP_CFG_DBG_NETCONN,
209             "[NETCONN] Ignoring more data for receive!\r\n");
210         lwesp_pbuf_free(pbuf);          /* Free pbuf */
211         return lwespOKIGNOREMORE;      /* Return OK to free the memory and
↳ ignore further data */
212     }
213     ++nc->mbox_receive_entries;          /* Increase number of packets in
↳ receive mbox */
214 #if LWESP_CFG_CONN_MANUAL_TCP_RECEIVE
215     /* Check against 1 less to still allow potential close event to be
↳ written to queue */
216     if (nc->mbox_receive_entries >= (LWESP_CFG_NETCONN_RECEIVE_QUEUE_LEN -
↳ 1)) {
217         conn->status.f.receive_blocked = 1; /* Block reading more data */
218     }
219 #endif /* LWESP_CFG_CONN_MANUAL_TCP_RECEIVE */
220
221     ++nc->rcv_packets;                  /* Increase number of packets
↳ received */
222     LWESP_DEBUGF(LWESP_CFG_DBG_NETCONN | LWESP_DBG_TYPE_TRACE,
223         "[NETCONN] Received pbuf contains %d bytes. Handle written to
↳ receive mbox\r\n",
224         (int) lwesp_pbuf_length(pbuf, 0));
225     break;
226 }
227
228 /* Connection was just closed */
229 case LWESP_EVT_CONN_CLOSE: {
230     nc = lwesp_conn_get_arg(conn);      /* Get API from connection */
231
232     /*
233     * In case we have a netconn available,
234     * simply write pointer to received variable to indicate closed state
235     */
236     if (nc != NULL && lwesp_sys_mbox_isvalid(&nc->mbox_receive)) {
237         if (lwesp_sys_mbox_putnow(&nc->mbox_receive, (void*)&recv_closed)) {

```

(continues on next page)

(continued from previous page)

```

238         ++nc->mbox_receive_entries;
239     }
240 }
241
242     break;
243 }
244     default:
245         return lwespERR;
246 }
247     return lwespOK;
248 }
249
250 /**
251  * \brief      Global event callback function
252  * \param[in]  evt: Callback information and data
253  * \return     \ref lwespOK on success, member of \ref lwespr_t otherwise
254  */
255 static lwespr_t
256 lwesp_evt(lwesp_evt_t* evt) {
257     switch (lwesp_evt_get_type(evt)) {
258         case LWESP_EVT_WIFI_DISCONNECTED: { /* Wifi disconnected event */
259             if (listen_api != NULL) { /* Check if listen API active */
260                 lwesp_sys_mbox_putnow(&listen_api->mbox_accept, &recv_closed);
261             }
262             break;
263         }
264         case LWESP_EVT_DEVICE_PRESENT: { /* Device present event */
265             if (listen_api != NULL && !lwesp_device_is_present()) { /* Check if_
↳device present */
266                 lwesp_sys_mbox_putnow(&listen_api->mbox_accept, &recv_not_present);
267             }
268         }
269         default:
270             break;
271     }
272     return lwespOK;
273 }
274
275 /**
276  * \brief      Create new netconn connection
277  * \param[in]  type: Netconn connection type
278  * \return     New netconn connection on success, `NULL` otherwise
279  */
280 lwesp_netconn_p
281 lwesp_netconn_new(lwesp_netconn_type_t type) {
282     lwesp_netconn_t* a;
283     static uint8_t first = 1;
284
285     /* Register only once! */
286     lwesp_core_lock();
287     if (first) {
288         first = 0;
289         lwesp_evt_register(lwesp_evt); /* Register global event function */
290     }
291     lwesp_core_unlock();
292     a = lwesp_mem_malloc(1, sizeof(*a)); /* Allocate memory for core object */
293     if (a != NULL) {

```

(continues on next page)

(continued from previous page)

```

294     a->type = type;                                     /* Save netconn type */
295     a->conn_timeout = 0;                               /* Default connection timeout */
296     if (!lwesp_sys_mbox_create(&a->mbox_accept, LWESP_CFG_NETCONN_ACCEPT_QUEUE_
↳LEN)) { /* Allocate memory for accepting message box */
297         LWESP_DEBUGF(LWESP_CFG_DBG_NETCONN | LWESP_DBG_TYPE_TRACE | LWESP_DBG_LVL_
↳DANGER,
298                     "[NETCONN] Cannot create accept MBOX\r\n");
299         goto free_ret;
300     }
301     if (!lwesp_sys_mbox_create(&a->mbox_receive, LWESP_CFG_NETCONN_RECEIVE_QUEUE_
↳LEN)) { /* Allocate memory for receiving message box */
302         LWESP_DEBUGF(LWESP_CFG_DBG_NETCONN | LWESP_DBG_TYPE_TRACE | LWESP_DBG_LVL_
↳DANGER,
303                     "[NETCONN] Cannot create receive MBOX\r\n");
304         goto free_ret;
305     }
306     lwesp_core_lock();
307     if (netconn_list == NULL) { /* Add new netconn to the existing_
↳list */
308         netconn_list = a;
309     } else {
310         a->next = netconn_list; /* Add it to beginning of the list */
311         netconn_list = a;
312     }
313     lwesp_core_unlock();
314 }
315 return a;
316 free_ret:
317 if (lwesp_sys_mbox_isvalid(&a->mbox_accept)) {
318     lwesp_sys_mbox_delete(&a->mbox_accept);
319     lwesp_sys_mbox_invalid(&a->mbox_accept);
320 }
321 if (lwesp_sys_mbox_isvalid(&a->mbox_receive)) {
322     lwesp_sys_mbox_delete(&a->mbox_receive);
323     lwesp_sys_mbox_invalid(&a->mbox_receive);
324 }
325 if (a != NULL) {
326     lwesp_mem_free_s((void*)&a);
327 }
328 return NULL;
329 }
330
331 /**
332  * \brief Delete netconn connection
333  * \param[in] nc: Netconn handle
334  * \return \ref lwespOK on success, member of \ref lwespr_t enumeration_
↳otherwise
335  */
336 lwespr_t
337 lwesp_netconn_delete(lwespr_t nc) {
338     LWESP_ASSERT("netconn != NULL", nc != NULL);
339
340     lwesp_core_lock();
341     flush_mboxes(nc, 0); /* Clear mboxs */
342
343     /* Stop listening on netconn */
344     if (nc == listen_api) {

```

(continues on next page)

(continued from previous page)

```

345     listen_api = NULL;
346     lwesp_core_unlock();
347     lwesp_set_server(0, nc->listen_port, 0, 0, NULL, NULL, NULL, 1);
348     lwesp_core_lock();
349 }
350
351 /* Remove netconn from linkedlist */
352 if (nc == netconn_list) {
353     netconn_list = netconn_list->next;      /* Remove first from linked list */
354 } else if (netconn_list != NULL) {
355     lwesp_netconn_p tmp, prev;
356     /* Find element on the list */
357     for (prev = netconn_list, tmp = netconn_list->next;
358         tmp != NULL; prev = tmp, tmp = tmp->next) {
359         if (nc == tmp) {
360             prev->next = tmp->next;      /* Remove tmp from linked list */
361             break;
362         }
363     }
364 }
365 lwesp_core_unlock();
366
367 lwesp_mem_free_s((void**) &nc);
368 return lwespOK;
369 }
370
371 /**
372  * \brief          Connect to server as client
373  * \param[in]     nc: Netconn handle
374  * \param[in]     host: Pointer to host, such as domain name or IP address in
375  * \param[in]     port: Target port to use
376  * \return        \ref lwespOK if successfully connected, member of \ref lwespr_t_
377  * \otherwise
378  */
379 lwespr_t
380 lwesp_netconn_connect(lwesp_netconn_p nc, const char* host, lwesp_port_t port) {
381     lwespr_t res;
382
383     LWESP_ASSERT("nc != NULL", nc != NULL);
384     LWESP_ASSERT("host != NULL", host != NULL);
385     LWESP_ASSERT("port > 0", port > 0);
386
387     /*
388      * Start a new connection as client and:
389      *
390      * - Set current netconn structure as argument
391      * - Set netconn callback function for connection management
392      * - Start connection in blocking mode
393      */
394     res = lwesp_conn_start(NULL, (lwesp_conn_type_t)nc->type, host, port, nc, netconn_
395     ↪evt, 1);
396     return res;
397 }
398
399 /**
400  * \brief          Connect to server as client, allow keep-alive option

```

(continues on next page)

(continued from previous page)

```

399 * \param[in]      nc: Netconn handle
400 * \param[in]      host: Pointer to host, such as domain name or IP address in_
↳string format
401 * \param[in]      port: Target port to use
402 * \param[in]      keep_alive: Keep alive period seconds
403 * \param[in]      local_ip: Local ip in connected command
404 * \param[in]      local_port: Local port address
405 * \param[in]      mode: UDP mode
406 * \return         \ref lwespOK if successfully connected, member of \ref lwespr_t_
↳otherwise
407 */
408 lwespr_t
409 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) {
410     lwesp_conn_start_t cs = {0};
411     lwespr_t res;
412
413     LWESP_ASSERT("nc != NULL", nc != NULL);
414     LWESP_ASSERT("host != NULL", host != NULL);
415     LWESP_ASSERT("port > 0", port > 0);
416
417     /*
418      * Start a new connection as client and:
419      *
420      * - Set current netconn structure as argument
421      * - Set netconn callback function for connection management
422      * - Start connection in blocking mode
423      */
424     cs.type = nc->type;
425     cs.remote_host = host;
426     cs.remote_port = port;
427     cs.local_ip = local_ip;
428     if (nc->type == LWESP_NETCONN_TYPE_TCP || nc->type == LWESP_NETCONN_TYPE_SSL) {
429         cs.ext.tcp_ssl.keep_alive = keep_alive;
430     } else {
431         cs.ext.udp.local_port = local_port;
432         cs.ext.udp.mode = mode;
433     }
434     res = lwesp_conn_startex(NULL, &cs, nc, netconn_evt, 1);
435     return res;
436 }
437
438 /**
439 * \brief          Bind a connection to specific port, can be only used for server_
↳connections
440 * \param[in]      nc: Netconn handle
441 * \param[in]      port: Port used to bind a connection to
442 * \return         \ref lwespOK on success, member of \ref lwespr_t enumeration_
↳otherwise
443 */
444 lwespr_t
445 lwesp_netconn_bind(lwesp_netconn_p nc, lwesp_port_t port) {
446     lwespr_t res = lwespOK;
447
448     LWESP_ASSERT("nc != NULL", nc != NULL);
449
450     /*

```

(continues on next page)

(continued from previous page)

```

451     * Protection is not needed as it is expected
452     * that this function is called only from single
453     * thread for single netconn connection,
454     * thus it is considered reentrant
455     */
456
457     nc->listen_port = port;
458
459     return res;
460 }
461
462 /**
463  * \brief          Set timeout value in units of seconds when connection is in_
↳listening mode
464  *                If new connection is accepted, it will be automatically closed_
↳after `seconds` elapsed
465  *                without any data exchange.
466  * \note          Call this function before you put connection to listen mode with \
↳ref lwesp_netconn_listen
467  * \param[in]     nc: Netconn handle used for listen mode
468  * \param[in]     timeout: Time in units of seconds. Set to `0` to disable timeout_
↳feature
469  * \return        \ref lwespOK on success, member of \ref lwespr_t otherwise
470  */
471 lwespr_t
472 lwesp_netconn_set_listen_conn_timeout(lwesp_netconn_p nc, uint16_t timeout) {
473     lwespr_t res = lwespOK;
474     LWESP_ASSERT("nc != NULL", nc != NULL);
475
476     /*
477      * Protection is not needed as it is expected
478      * that this function is called only from single
479      * thread for single netconn connection,
480      * thus it is reentrant in this case
481      */
482
483     nc->conn_timeout = timeout;
484
485     return res;
486 }
487
488 /**
489  * \brief          Listen on previously binded connection
490  * \param[in]     nc: Netconn handle used to listen for new connections
491  * \return        \ref lwespOK on success, member of \ref lwespr_t enumeration_
↳otherwise
492  */
493 lwespr_t
494 lwesp_netconn_listen(lwesp_netconn_p nc) {
495     return lwesp_netconn_listen_with_max_conn(nc, LWESP_CFG_MAX_CONNS);
496 }
497
498 /**
499  * \brief          Listen on previously binded connection with max allowed_
↳connections at a time
500  * \param[in]     nc: Netconn handle used to listen for new connections
501  * \param[in]     max_connections: Maximal number of connections server can accept_
↳at a time

```

(continues on next page)

(continued from previous page)

```

502  *           This parameter may not be larger than \ref LWESP_CFG_MAX_CONNS
503  * \return   \ref lwespOK on success, member of \ref lwespr_t otherwise
504  */
505 lwespr_t
506 lwesp_netconn_listen_with_max_conn(lwesp_netconn_p nc, uint16_t max_connections) {
507     lwespr_t res;
508
509     LWESP_ASSERT("nc != NULL", nc != NULL);
510     LWESP_ASSERT("nc->type must be TCP", nc->type == LWESP_NETCONN_TYPE_TCP);
511
512     /* Enable server on port and set default netconn callback */
513     if ((res = lwesp_set_server(1, nc->listen_port,
514                               LWESP_U16(LWESP_MIN(max_connections, LWESP_CFG_MAX_
515 ↪CONNS))),
516                               nc->conn_timeout, netconn_evt, NULL, NULL, 1)) == ↪
517 ↪lwespOK) {
518         lwesp_core_lock();
519         listen_api = nc;           /* Set current main API in listening ↪
520 ↪state */
521         lwesp_core_unlock();
522     }
523     return res;
524 }
525
526 /**
527  * \brief     Accept a new connection
528  * \param[in] nc: Netconn handle used as base connection to accept new clients
529  * \param[out] client: Pointer to netconn handle to save new connection to
530  * \return   \ref lwespOK on success, member of \ref lwespr_t enumeration ↪
531 ↪otherwise
532  */
533 lwespr_t
534 lwesp_netconn_accept(lwesp_netconn_p nc, lwesp_netconn_p* client) {
535     lwesp_netconn_t* tmp;
536     uint32_t time;
537
538     LWESP_ASSERT("nc != NULL", nc != NULL);
539     LWESP_ASSERT("client != NULL", client != NULL);
540     LWESP_ASSERT("nc->type must be TCP", nc->type == LWESP_NETCONN_TYPE_TCP);
541     LWESP_ASSERT("nc == listen_api", nc == listen_api);
542
543     *client = NULL;
544     time = lwesp_sys_mbox_get(&nc->mbox_accept, (void**) &tmp, 0);
545     if (time == LWESP_SYS_TIMEOUT) {
546         return lwespTIMEOUT;
547     }
548     if ((uint8_t*)tmp == (uint8_t*)&recv_closed) {
549         lwesp_core_lock();
550         listen_api = NULL;           /* Disable listening at this point */
551         lwesp_core_unlock();
552         return lwespERRWIFINOTCONNECTED; /* Wifi disconnected */
553     } else if ((uint8_t*)tmp == (uint8_t*)&recv_not_present) {
554         lwesp_core_lock();
555         listen_api = NULL;           /* Disable listening at this point */
556         lwesp_core_unlock();
557         return lwespERRNODEVICE;     /* Device not present */
558     }
559 }

```

(continues on next page)

(continued from previous page)

```

555     *client = tmp;                               /* Set new pointer */
556     return lwespOK;                               /* We have a new connection */
557 }
558
559 /**
560  * \brief      Write data to connection output buffers
561  * \note      This function may only be used on TCP or SSL connections
562  * \param[in] nc: Netconn handle used to write data to
563  * \param[in] data: Pointer to data to write
564  * \param[in] btw: Number of bytes to write
565  * \return     \ref lwespOK on success, member of \ref lwespr_t enumeration
↳otherwise
566  */
567 lwespr_t
568 lwesp_netconn_write(lwesp_netconn_p nc, const void* data, size_t btw) {
569     size_t len, sent;
570     const uint8_t* d = data;
571     lwespr_t res;
572
573     LWESP_ASSERT("nc != NULL", nc != NULL);
574     LWESP_ASSERT("nc->type must be TCP or SSL", nc->type == LWESP_NETCONN_TYPE_TCP ||
↳nc->type == LWESP_NETCONN_TYPE_SSL);
575     LWESP_ASSERT("nc->conn must be active", lwesp_conn_is_active(nc->conn));
576
577     /*
578      * Several steps are done in write process
579      *
580      * 1. Check if buffer is set and check if there is something to write to it.
581      *    1. In case buffer will be full after copy, send it and free memory.
582      *    2. Check how many bytes we can write directly without need to copy
583      *    3. Try to allocate a new buffer and copy remaining input data to it
584      *    4. In case buffer allocation fails, send data directly (may have impact on
↳speed and effectiveness)
585      */
586
587     /* Step 1 */
588     if (nc->buff.buff != NULL) {                  /* Is there a write buffer ready to
↳accept more data? */
589         len = LWESP_MIN(nc->buff.len - nc->buff.ptr, btw); /* Get number of bytes we
↳can write to buffer */
590         if (len > 0) {
591             LWESP_MEMCPY(&nc->buff.buff[nc->buff.ptr], data, len); /* Copy memory to
↳temporary write buffer */
592             d += len;
593             nc->buff.ptr += len;
594             btw -= len;
595         }
596
597         /* Step 1.1 */
598         if (nc->buff.ptr == nc->buff.len) {
599             res = lwesp_conn_send(nc->conn, nc->buff.buff, nc->buff.len, &sent, 1);
600
601             lwesp_mem_free_s((void**) &nc->buff.buff);
602             if (res != lwespOK) {
603                 return res;
604             }
605         } else {

```

(continues on next page)

(continued from previous page)

```

606         return lwespOK;                /* Buffer is not full yet */
607     }
608 }
609
610 /* Step 2 */
611 if (btw >= LWESP_CFG_CONN_MAX_DATA_LEN) {
612     size_t rem;
613     rem = btw % LWESP_CFG_CONN_MAX_DATA_LEN; /* Get remaining bytes for max data_
↳length */
614     res = lwesp_conn_send(nc->conn, d, btw - rem, &sent, 1); /* Write data_
↳directly */
615     if (res != lwespOK) {
616         return res;
617     }
618     d += sent;                /* Advance in data pointer */
619     btw -= sent;             /* Decrease remaining data to send */
620 }
621
622 if (btw == 0) {                /* Sent everything? */
623     return lwespOK;
624 }
625
626 /* Step 3 */
627 if (nc->buff.buff == NULL) {    /* Check if we should allocate a new_
↳buffer */
628     nc->buff.buff = lwesp_mem_malloc(sizeof(*nc->buff.buff) * LWESP_CFG_CONN_MAX_
↳DATA_LEN);
629     nc->buff.len = LWESP_CFG_CONN_MAX_DATA_LEN; /* Save buffer length */
630     nc->buff.ptr = 0;          /* Save buffer pointer */
631 }
632
633 /* Step 4 */
634 if (nc->buff.buff != NULL) {    /* Memory available? */
635     LWESP_MEMCPY(&nc->buff.buff[nc->buff.ptr], d, btw); /* Copy data to buffer */
636     nc->buff.ptr += btw;
637 } else {                        /* Still no memory available? */
638     return lwesp_conn_send(nc->conn, data, btw, NULL, 1); /* Simply send_
↳directly blocking */
639 }
640 return lwespOK;
641 }
642
643 /**
644  * \brief      Flush buffered data on netconn TCP/SSL connection
645  * \note       This function may only be used on TCP/SSL connection
646  * \param[in]  nc: Netconn handle to flush data
647  * \return     \ref lwespOK on success, member of \ref lwespr_t enumeration_
↳otherwise
648  */
649 lwespr_t
650 lwesp_netconn_flush(lwesp_netconn_p nc) {
651     LWESP_ASSERT("nc != NULL", nc != NULL);
652     LWESP_ASSERT("nc->type must be TCP or SSL", nc->type == LWESP_NETCONN_TYPE_TCP ||
↳nc->type == LWESP_NETCONN_TYPE_SSL);
653     LWESP_ASSERT("nc->conn must be active", lwesp_conn_is_active(nc->conn));
654
655     /*

```

(continues on next page)

(continued from previous page)

```

656     * In case we have data in write buffer,
657     * flush them out to network
658     */
659     if (nc->buff.buff != NULL) {                               /* Check remaining data */
660         if (nc->buff.ptr > 0) {                               /* Do we have data in current buffer? */
661             ↪ */
662                 lwesp_conn_send(nc->conn, nc->buff.buff, nc->buff.ptr, NULL, 1); /* Send
663             ↪ data */
664         }
665         lwesp_mem_free_s((void*)&nc->buff.buff);
666     }
667     return lwespOK;
668 }
669
670 /**
671  * \brief          Send data on UDP connection to default IP and port
672  * \param[in]     nc: Netconn handle used to send
673  * \param[in]     data: Pointer to data to write
674  * \param[in]     btw: Number of bytes to write
675  * \return        \ref lwespOK on success, member of \ref lwespr_t enumeration
676  ↪ otherwise
677  */
678 lwespr_t
679 lwesp_netconn_send(lwesp_netconn_p nc, const void* data, size_t btw) {
680     LWESP_ASSERT("nc != NULL", nc != NULL);
681     LWESP_ASSERT("nc->type must be UDP", nc->type == LWESP_NETCONN_TYPE_UDP);
682     LWESP_ASSERT("nc->conn must be active", lwesp_conn_is_active(nc->conn));
683
684     return lwesp_conn_send(nc->conn, data, btw, NULL, 1);
685 }
686
687 /**
688  * \brief          Send data on UDP connection to specific IP and port
689  * \note           Use this function in case of UDP type netconn
690  * \param[in]     nc: Netconn handle used to send
691  * \param[in]     ip: Pointer to IP address
692  * \param[in]     port: Port number used to send data
693  * \param[in]     data: Pointer to data to write
694  * \param[in]     btw: Number of bytes to write
695  * \return        \ref lwespOK on success, member of \ref lwespr_t enumeration
696  ↪ otherwise
697  */
698 lwespr_t
699 lwesp_netconn_sendto(lwesp_netconn_p nc, const lwesp_ip_t* ip, lwesp_port_t port,
700 ↪ const void* data, size_t btw) {
701     LWESP_ASSERT("nc != NULL", nc != NULL);
702     LWESP_ASSERT("nc->type must be UDP", nc->type == LWESP_NETCONN_TYPE_UDP);
703     LWESP_ASSERT("nc->conn must be active", lwesp_conn_is_active(nc->conn));
704
705     return lwesp_conn_sendto(nc->conn, ip, port, data, btw, NULL, 1);
706 }
707
708 /**
709  * \brief          Receive data from connection
710  * \param[in]     nc: Netconn handle used to receive from
711  * \param[in]     pbuf: Pointer to pointer to save new receive buffer to.
712  *               When function returns, user must check for valid pbuf value
713  ↪ `pbuf != NULL`

```

(continues on next page)

```

708 * \return          \ref lwespOK when new data ready
709 * \return          \ref lwespCLOSED when connection closed by remote side
710 * \return          \ref lwespTIMEOUT when receive timeout occurs
711 * \return          Any other member of \ref lwespr_t otherwise
712 */
713 lwespr_t
714 lwesp_netconn_receive(lwesp_netconn_p nc, lwesp_pbuf_p* pbuf) {
715     LWESP_ASSERT("nc != NULL", nc != NULL);
716     LWESP_ASSERT("pbuf != NULL", pbuf != NULL);
717
718     *pbuf = NULL;
719 #if LWESP_CFG_NETCONN_RECEIVE_TIMEOUT
720     /*
721      * Wait for new received data for up to specific timeout
722      * or throw error for timeout notification
723      */
724     if (nc->rcv_timeout == LWESP_NETCONN_RECEIVE_NO_WAIT) {
725         if (!lwesp_sys_mbox_getnow(&nc->mbox_receive, (void**)pbuf)) {
726             return lwespTIMEOUT;
727         }
728     } else if (lwesp_sys_mbox_get(&nc->mbox_receive, (void**)pbuf, nc->rcv_timeout)
729 → == LWESP_SYS_TIMEOUT) {
730         return lwespTIMEOUT;
731     }
732 #else /* LWESP_CFG_NETCONN_RECEIVE_TIMEOUT */
733     /* Forever wait for new receive packet */
734     lwesp_sys_mbox_get(&nc->mbox_receive, (void**)pbuf, 0);
735 #endif /* !LWESP_CFG_NETCONN_RECEIVE_TIMEOUT */
736
737     lwesp_core_lock();
738     if (nc->mbox_receive_entries > 0) {
739         --nc->mbox_receive_entries;
740     }
741     lwesp_core_unlock();
742
743     /* Check if connection closed */
744     if ((uint8_t*)(*pbuf) == (uint8_t*)&rcv_closed) {
745         *pbuf = NULL; /* Reset pbuf */
746         return lwespCLOSED;
747     }
748 #if LWESP_CFG_CONN_MANUAL_TCP_RECEIVE
749     else {
750         lwesp_core_lock();
751         nc->conn->status.f.receive_blocked = 0; /* Resume reading more data */
752         lwesp_conn_rcv(nc->conn, *pbuf); /* Notify stack about received data */
753         lwesp_core_unlock();
754     }
755 #endif /* LWESP_CFG_CONN_MANUAL_TCP_RECEIVE */
756     return lwespOK; /* We have data available */
757 }
758 /**
759 * \brief          Close a netconn connection
760 * \param[in]      nc: Netconn handle to close
761 * \return          \ref lwespOK on success, member of \ref lwespr_t enumeration,
762 → otherwise
763 */

```

(continues on next page)

(continued from previous page)

```

763 lwespr_t
764 lwesp_netconn_close(lwesp_netconn_p nc) {
765     lwesp_conn_p conn;
766
767     LWESP_ASSERT("nc != NULL", nc != NULL);
768     LWESP_ASSERT("nc->conn != NULL", nc->conn != NULL);
769     LWESP_ASSERT("nc->conn must be active", lwesp_conn_is_active(nc->conn));
770
771     lwesp_netconn_flush(nc);           /* Flush data and ignore result */
772     conn = nc->conn;
773     nc->conn = NULL;
774
775     lwesp_conn_set_arg(conn, NULL);   /* Reset argument */
776     lwesp_conn_close(conn, 1);       /* Close the connection */
777     flush_mboxes(nc, 1);             /* Flush message queues */
778     return lwespOK;
779 }
780
781 /**
782  * \brief          Get connection number used for netconn
783  * \param[in]     nc: Netconn handle
784  * \return        `-1` on failure, connection number between `0` and \ref LWESP_CFG_
785  *               ↪ MAX_CONNS otherwise
786  */
787 int8_t
788 lwesp_netconn_get_connum(lwesp_netconn_p nc) {
789     if (nc != NULL && nc->conn != NULL) {
790         return lwesp_conn_getnum(nc->conn);
791     }
792     return -1;
793 }
794 #if LWESP_CFG_NETCONN_RECEIVE_TIMEOUT || __DOXYGEN__
795
796 /**
797  * \brief          Set timeout value for receiving data.
798  *
799  * When enabled, \ref lwesp_netconn_receive will only block for up to
800  * `timeout` value and will return if no new data within this time
801  *
802  * \param[in]     nc: Netconn handle
803  * \param[in]     timeout: Timeout in units of milliseconds.
804  *                Set to `0` to disable timeout feature
805  *                Set to `> 0` to set maximum milliseconds to wait before_
806  *               ↪ timeout
807  *                Set to \ref LWESP_NETCONN_RECEIVE_NO_WAIT to enable non-
808  *               ↪ blocking receive
809  */
810 void
811 lwesp_netconn_set_receive_timeout(lwesp_netconn_p nc, uint32_t timeout) {
812     nc->rcv_timeout = timeout;
813 }
814
815 /**
816  * \brief          Get netconn receive timeout value
817  * \param[in]     nc: Netconn handle
818  * \return        Timeout in units of milliseconds.

```

(continues on next page)

(continued from previous page)

```

817 *           If value is `0`, timeout is disabled (wait forever)
818 */
819 uint32_t
820 lwesp_netconn_get_receive_timeout(lwesp_netconn_p nc) {
821     return nc->rcv_timeout;
822 }
823
824 #endif /* LWESP_CFG_NETCONN_RECEIVE_TIMEOUT || __DOXYGEN__ */
825
826 /**
827 * \brief           Get netconn connection handle
828 * \param[in]       nc: Netconn handle
829 * \return          ESP connection handle
830 */
831 lwesp_conn_p
832 lwesp_netconn_get_conn(lwesp_netconn_p nc) {
833     return nc->conn;
834 }
835
836 #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

```

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
10 /**
11 * \brief           Request data for connection
12 */
13 static const
14 uint8_t req_data[] = ""
15             "GET / HTTP/1.1\r\n"

```

(continues on next page)

(continued from previous page)

```

16         "Host: " CONN_HOST "\r\n"
17         "Connection: close\r\n"
18         "\r\n";
19
20 /**
21  * \brief      Start a new connection(s) as client
22  */
23 void
24 client_connect(void) {
25     lwespr_t res;
26
27     /* Start a new connection as client in non-blocking mode */
28     if ((res = lwesp_conn_start(NULL, LWESP_CONN_TYPE_TCP, "example.com", 80, NULL,
↳conn_callback_func, 0)) == lwespOK) {
29         printf("Connection to " CONN_HOST " started...\r\n");
30     } else {
31         printf("Cannot start connection to " CONN_HOST "!\r\n");
32     }
33
34     /* Start 2 more */
35     lwesp_conn_start(NULL, LWESP_CONN_TYPE_TCP, CONN_HOST, CONN_PORT, NULL, conn_
↳callback_func, 0);
36
37     /*
38      * An example of connection which should fail in connecting.
39      * When this is the case, \ref LWESP_EVT_CONN_ERROR event should be triggered
40      * in callback function processing
41      */
42     lwesp_conn_start(NULL, LWESP_CONN_TYPE_TCP, CONN_HOST, 10, NULL, conn_callback_
↳func, 0);
43 }
44
45 /**
46  * \brief      Event callback function for connection-only
47  * \param[in]  evt: Event information with data
48  * \return     \ref lwespOK on success, member of \ref lwespr_t otherwise
49  */
50 static lwespr_t
51 conn_callback_func(lwesp_evt_t* evt) {
52     lwesp_conn_p conn;
53     lwespr_t res;
54     uint8_t conn_num;
55
56     conn = lwesp_conn_get_from_evt(evt);          /* Get connection handle from event_
↳*/
57     if (conn == NULL) {
58         return lwespERR;
59     }
60     conn_num = lwesp_conn_getnum(conn);          /* Get connection number for_
↳identification */
61     switch (lwesp_evt_get_type(evt)) {
62         case LWESP_EVT_CONN_ACTIVE: {           /* Connection just active */
63             printf("Connection %d active!\r\n", (int) conn_num);
64             res = lwesp_conn_send(conn, req_data, sizeof(req_data) - 1, NULL, 0); /*_
↳Start sending data in non-blocking mode */
65             if (res == lwespOK) {
66                 printf("Sending request data to server...\r\n");

```

(continues on next page)

```

67         } else {
68             printf("Cannot send request data to server. Closing connection_
↳ manually...\r\n");
69             lwesp_conn_close(conn, 0);          /* Close the connection */
70         }
71         break;
72     }
73     case LWESP_EVT_CONN_CLOSE: {              /* Connection closed */
74         if (lwesp_evt_conn_close_is_forced(evt)) {
75             printf("Connection %d closed by client!\r\n", (int)conn_num);
76         } else {
77             printf("Connection %d closed by remote side!\r\n", (int)conn_num);
78         }
79         break;
80     }
81     case LWESP_EVT_CONN_SEND: {              /* Data send event */
82         lwesp_r_t res = lwesp_evt_conn_send_get_result(evt);
83         if (res == lwespOK) {
84             printf("Data sent successfully on connection %d...waiting to receive_
↳ data from remote side...\r\n", (int)conn_num);
85         } else {
86             printf("Error while sending data on connection %d!\r\n", (int)conn_
↳ num);
87         }
88         break;
89     }
90     case LWESP_EVT_CONN_RECV: {              /* Data received from remote side */
91         lwesp_pbuf_p pbuf = lwesp_evt_conn_recv_get_buff(evt);
92         lwesp_conn_recved(conn, pbuf);        /* Notify stack about received pbuf_
↳ */
93         printf("Received %d bytes on connection %d...\r\n", (int)lwesp_pbuf_
↳ length(pbuf, 1), (int)conn_num);
94         break;
95     }
96     case LWESP_EVT_CONN_ERROR: {             /* Error connecting to server */
97         const char* host = lwesp_evt_conn_error_get_host(evt);
98         lwesp_port_t port = lwesp_evt_conn_error_get_port(evt);
99         printf("Error connecting to %s:%d\r\n", host, (int)port);
100        break;
101    }
102    default:
103        break;
104    }
105    return lwespOK;
106 }

```

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 distinguish domain name (when multiple domains are to be resolved).

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

```

1  #include "dns.h"
2  #include "lwesp/lwesp.h"
3
4  /* Host to resolve */
5  #define DNS_HOST1      "example.com"
6  #define DNS_HOST2      "example.net"
7
8  /**
9   * \brief          Variable to hold result of DNS resolver
10  */
11  static lwesp_ip_t ip;
12
13  /**
14   * \brief          Event callback function for API call,
15   *                called when API command finished with execution
16  */
17  static void
18  dns_resolve_evt(lwespr_t res, void* arg) {
19      /* Check result of command */
20      if (res == lwespOK) {
21          /* DNS resolver has IP address */
22          printf("DNS record for %s (from API callback): %d.%d.%d.%d\r\n",
23              (const char*)arg, (int)ip.ip[0], (int)ip.ip[1], (int)ip.ip[2], (int)ip.
↪ip[3]);
24      }
25  }
26
27  /**
28   * \brief          Start DNS resolver
29  */
30  void
31  dns_start(void) {
32      /* Use DNS protocol to get IP address of domain name */
33
34      /* Get IP with non-blocking mode */
35      if (lwesp_dns_gethostbyname(DNS_HOST2, &ip, dns_resolve_evt, DNS_HOST2, 0) ==_
↪lwespOK) {
36          printf("Request for DNS record for " DNS_HOST2 " has started\r\n");
37      } else {
38          printf("Could not start command for DNS\r\n");
39      }

```

(continues on next page)

(continued from previous page)

```

40
41     /* Get IP with blocking mode */
42     if (lwesp_dns_gethostbyname(DNS_HOST1, &ip, dns_resolve_evt, DNS_HOST1, 1) ==
↳lwespOK) {
43         printf("DNS record for " DNS_HOST1 " (from lin code): %d.%d.%d.%d\r\n",
44             (int)ip.ip[0], (int)ip.ip[1], (int)ip.ip[2], (int)ip.ip[3]);
45     } else {
46         printf("Could not retrieve IP address for " DNS_HOST1 "\r\n");
47     }
48 }

```

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

```

1  char hostname[20];
2
3  /* Somewhere in thread function */
4
5  /* Get device hostname in blocking mode */
6  /* Function returns actual result */
7  if (lwesp_hostname_get(hostname, sizeof(hostname), NULL, NULL, 1 /* 1 means blocking_
↳call */) == lwespOK) {
8      /* At this point we have valid result and parameters from API function */
9      printf("ESP hostname is %s\r\n", hostname);
10 } else {
11     printf("Error reading ESP hostname..\r\n");
12 }

```

Non-blocking mode

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

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

Example code:

Listing 6: Non-blocking command example

```

1 char hostname[20];
2
3 /* Hostname event function, called when lwesp_hostname_get() function finishes */
4 void
5 hostname_fn(lwespr_t res, void* arg) {
6     /* Check actual result from device */
7     if (res == lwespOK) {
8         printf("ESP hostname is %s\r\n", hostname);
9     } else {
10        printf("Error reading ESP hostname...\r\n");
11    }
12 }
13
14 /* Somewhere in thread and/or other ESP event function */
15
16 /* Get device hostname in non-blocking mode */
17 /* Function now returns if command has been sent to internal message queue */
18 if (lwesp_hostname_get(hostname, sizeof(hostname), hostname_fn, NULL, 0 /* 0 means_
19 ↪non-blocking call */) == lwespOK) {
20     /* At this point application knows that command has been sent to queue */
21     /* But it does not have yet valid data in "hostname" variable */
22     printf("ESP hostname get command sent to queue.\r\n");
23 } else {
24     /* Error writing message to queue */
25     printf("Cannot send hostname get command to queue.\r\n");
26 }

```

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

```

1 char hostname[20];
2
3 /* Hostname event function, called when lwesp_hostname_get() function finishes */
4 void
5 hostname_fn(lwespr_t res, void* arg) {
6     /* Check actual result from device */

```

(continues on next page)

```

7     if (res == lwespOK) {
8         printf("ESP hostname is %s\r\n", hostname);
9     } else {
10        printf("Error reading ESP hostname...\r\n");
11    }
12 }
13
14 /* Check hostname */
15 void
16 check_hostname(void) {
17     char hostname[20];
18
19     /* Somewhere in thread and/or other ESP event function */
20
21     /* Get device hostname in non-blocking mode */
22     /* Function now returns if command has been sent to internal message queue */
23     /* Function will use local "hostname" variable and will write to undefined memory.
↳ */
24     if (lwesp_hostname_get(hostname, sizeof(hostname), hostname_fn, NULL, 0 /* 0
↳ means non-blocking call */) == lwespOK) {
25         /* At this point application knows that command has been sent to queue */
26         /* But it does not have yet valid data in "hostname" variable */
27         printf("ESP hostname get command sent to queue.\r\n");
28     } else {
29         /* Error writing message to queue */
30         printf("Cannot send hostname get command to queue.\r\n");
31     }
32 }

```

5.2.6 Porting guide

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

Porting consists of:

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

Implement low-level driver

To successfully 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

```

1  /**
2   * \file          lwesp_ll_win32.c
3   * \brief        Low-level communication with ESP device for WIN32
4   */
5
6  /*
7   * Copyright (c) 2020 Tilen MAJERLE
8   *
9   * Permission is hereby granted, free of charge, to any person
10  * obtaining a copy of this software and associated documentation
11  * files (the "Software"), to deal in the Software without restriction,
12  * including without limitation the rights to use, copy, modify, merge,
13  * publish, distribute, sublicense, and/or sell copies of the Software,
14  * and to permit persons to whom the Software is furnished to do so,
15  * subject to the following conditions:
16  *
17  * The above copyright notice and this permission notice shall be
18  * included in all copies or substantial portions of the Software.
19  *
20  * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
21  * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
22  * OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
23  * AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
24  * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
25  * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
26  * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
27  * OTHER DEALINGS IN THE SOFTWARE.
28  *
29  * This file is part of LwESP - Lightweight ESP-AT parser library.
30  *
31  * Author:         Tilen MAJERLE <tilen@majerle.eu>
32  * Version:        $_version_$
33  */
34 #include "system/lwesp_ll.h"
35 #include "lwesp/lwesp.h"

```

(continues on next page)

(continued from previous page)

```

36 #include "lwesp/lwesp_mem.h"
37 #include "lwesp/lwesp_input.h"
38
39 #if !__DOXYGEN__
40
41 volatile uint8_t lwesp_ll_win32_driver_ignore_data;
42 static uint8_t initialized = 0;
43 static HANDLE thread_handle;
44 static volatile HANDLE com_port;           /*!< COM port handle */
45 static uint8_t data_buffer[0x1000];       /*!< Received data array */
46
47 static void uart_thread(void* param);
48
49 /**
50  * \brief      Send data to ESP device, function called from ESP stack when we
51  *             have data to send
52  */
53 static size_t
54 send_data(const void* data, size_t len) {
55     DWORD written;
56     if (com_port != NULL) {
57 #if !LWESP_CFG_AT_ECHO
58         const uint8_t* d = data;
59         HANDLE hConsole;
60
61         hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
62         SetConsoleTextAttribute(hConsole, FOREGROUND_RED);
63         for (DWORD i = 0; i < len; ++i) {
64             printf("%c", d[i]);
65         }
66         SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN |
67         FOREGROUND_BLUE);
68 #endif /* !LWESP_CFG_AT_ECHO */
69
70         WriteFile(com_port, data, len, &written, NULL);
71         FlushFileBuffers(com_port);
72         return written;
73     }
74     return 0;
75 }
76
77 /**
78  * \brief      Configure UART (USB to UART)
79  */
80 static void
81 configure_uart(uint32_t baudrate) {
82     DCB dcb = { 0 };
83     dcb.DCBlength = sizeof(dcb);
84
85     /*
86      * On first call,
87      * create virtual file on selected COM port and open it
88      * as generic read and write
89      */
90     if (!initialized) {
91         com_port = CreateFile(L"\\\\.\\COM4",
92                             GENERIC_READ | GENERIC_WRITE,

```

(continues on next page)

```

91         0,
92         0,
93         OPEN_EXISTING,
94         0,
95         NULL
96     );
97 }
98
99 /* Configure COM port parameters */
100 if (GetCommState(com_port, &dcb)) {
101     COMMTIMEOUTS timeouts;
102
103     dcb.BaudRate = baudrate;
104     dcb.ByteSize = 8;
105     dcb.Parity = NOPARITY;
106     dcb.StopBits = ONESTOPBIT;
107
108     if (!SetCommState(com_port, &dcb)) {
109         printf("Cannot set COM PORT info\r\n");
110     }
111     if (GetCommTimeouts(com_port, &timeouts)) {
112         /* Set timeout to return immediately from ReadFile function */
113         timeouts.ReadIntervalTimeout = MAXDWORD;
114         timeouts.ReadTotalTimeoutConstant = 0;
115         timeouts.ReadTotalTimeoutMultiplier = 0;
116         if (!SetCommTimeouts(com_port, &timeouts)) {
117             printf("Cannot set COM PORT timeouts\r\n");
118         }
119         GetCommTimeouts(com_port, &timeouts);
120     } else {
121         printf("Cannot get COM PORT timeouts\r\n");
122     }
123 } else {
124     printf("Cannot get COM PORT info\r\n");
125 }
126
127 /* On first function call, create a thread to read data from COM port */
128 if (!initialized) {
129     lwesp_sys_thread_create(&thread_handle, "lwesp_ll_thread", uart_thread, NULL,
130 ↪0, 0);
131 }
132
133 /**
134  * \brief          UART thread
135  */
136 static void
137 uart_thread(void* param) {
138     DWORD bytes_read;
139     lwesp_sys_sem_t sem;
140     FILE* file = NULL;
141
142     lwesp_sys_sem_create(&sem, 0);           /* Create semaphore for delay_
143 ↪functions */
144
145     while (com_port == NULL) {
146         lwesp_sys_sem_wait(&sem, 1);       /* Add some delay with yield */

```

(continues on next page)

(continued from previous page)

```

146     }
147
148     fopen_s(&file, "log_file.txt", "w+");           /* Open debug file in write mode */
149     while (1) {
150         /*
151          * Try to read data from COM port
152          * and send it to upper layer for processing
153          */
154         do {
155             ReadFile(com_port, data_buffer, sizeof(data_buffer), &bytes_read, NULL);
156             if (bytes_read > 0) {
157                 HANDLE hConsole;
158                 hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
159                 SetConsoleTextAttribute(hConsole, FOREGROUND_GREEN);
160                 for (DWORD i = 0; i < bytes_read; ++i) {
161                     printf("%c", data_buffer[i]);
162                 }
163                 SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN |
↳FOREGROUND_BLUE);
164
165                 if (lwesp_ll_win32_driver_ignore_data) {
166                     printf("IGNORING..\r\n");
167                     continue;
168                 }
169
170                 /* Send received data to input processing module */
171 #if LWESP_CFG_INPUT_USE_PROCESS
172                 lwesp_input_process(data_buffer, (size_t)bytes_read);
173 #else /* LWESP_CFG_INPUT_USE_PROCESS */
174                 lwesp_input(data_buffer, (size_t)bytes_read);
175 #endif /* !LWESP_CFG_INPUT_USE_PROCESS */
176
177                 /* Write received data to output debug file */
178                 if (file != NULL) {
179                     fwrite(data_buffer, 1, bytes_read, file);
180                     fflush(file);
181                 }
182             }
183         } while (bytes_read == (DWORD)sizeof(data_buffer));
184
185         /* Implement delay to allow other tasks processing */
186         lwesp_sys_sem_wait(&sem, 1);
187     }
188 }
189
190 /**
191  * \brief          Reset device GPIO management
192  */
193 static uint8_t
194 reset_device(uint8_t state) {
195     return 0;                                     /* Hardware reset was not successful_
↳*/
196 }
197
198 /**
199  * \brief          Callback function called from initialization process
200  */

```

(continues on next page)

```

201 lwespr_t
202 lwesp_ll_init(lwesp_ll_t* ll) {
203 #if !LWESP_CFG_MEM_CUSTOM
204     /* Step 1: Configure memory for dynamic allocations */
205     static uint8_t memory[0x10000];          /* Create memory for dynamic_
↳allocations with specific size */
206
207     /*
208      * Create memory region(s) of memory.
209      * If device has internal/external memory available,
210      * multiple memories may be used
211      */
212     lwesp_mem_region_t mem_regions[] = {
213         { memory, sizeof(memory) }
214     };
215     if (!initialized) {
216         lwesp_mem_assignmemory(mem_regions, LWESP_ARRAYSIZE(mem_regions)); /* Assign_
↳memory for allocations to ESP library */
217     }
218 #endif /* !LWESP_CFG_MEM_CUSTOM */
219
220     /* Step 2: Set AT port send function to use when we have data to transmit */
221     if (!initialized) {
222         ll->send_fn = send_data;             /* Set callback function to send data_
↳*/
223
224         ll->reset_fn = reset_device;
225     }
226
227     /* Step 3: Configure AT port to be able to send/receive data to/from ESP device */
228     configure_uart(ll->uart.baudrate);      /* Initialize UART for communication_
↳*/
229     initialized = 1;
230     return lwespOK;
231 }
232
233 /**
234  * \brief          Callback function to de-init low-level communication part
235  */
236 lwespr_t
237 lwesp_ll_deinit(lwesp_ll_t* ll) {
238     if (thread_handle != NULL) {
239         lwesp_sys_thread_terminate(&thread_handle);
240         thread_handle = NULL;
241     }
242     initialized = 0;                         /* Clear initialized flag */
243     return lwespOK;
244 }
245 #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

```

1  /**
2   * \file          lwesp_ll_stm32.c
3   * \brief        Generic STM32 driver, included in various STM32 driver variants
4   */
5
6  /**
7   * Copyright (c) 2020 Tilen MAJERLE
8   *
9   * Permission is hereby granted, free of charge, to any person
10  * obtaining a copy of this software and associated documentation
11  * files (the "Software"), to deal in the Software without restriction,
12  * including without limitation the rights to use, copy, modify, merge,
13  * publish, distribute, sublicense, and/or sell copies of the Software,
14  * and to permit persons to whom the Software is furnished to do so,
15  * subject to the following conditions:
16  *
17  * The above copyright notice and this permission notice shall be
18  * included in all copies or substantial portions of the Software.
19  *
20  * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
21  * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
22  * OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
23  * AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
24  * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
25  * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
26  * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
27  * OTHER DEALINGS IN THE SOFTWARE.
28  *
29  * This file is part of LwESP - Lightweight ESP-AT parser library.
30  *
31  * Author:         Tilen MAJERLE <tilen@majerle.eu>
32  * Version:        $_version_$
33  */
34
35  /**
36   * How it works
37   *
38   * On first call to \ref lwesp_ll_init, new thread is created and processed in usart_
39   * ↪ll_thread function.
40   * USART is configured in RX DMA mode and any incoming bytes are processed inside_
41   * ↪thread function.
42   * DMA and USART implement interrupt handlers to notify main thread about new data_
43   * ↪ready to send to upper layer.

```

(continues on next page)

```

41  *
42  * More about UART + RX DMA: https://github.com/MaJerle/stm32-usart-dma-rx-tx
43  *
44  * \ref LWESP_CFG_INPUT_USE_PROCESS must be enabled in `lwesp_config.h` to use this_
  ↪ driver.
45  */
46  #include "lwesp/lwesp.h"
47  #include "lwesp/lwesp_mem.h"
48  #include "lwesp/lwesp_input.h"
49  #include "system/lwesp_ll.h"
50
51  #if !__DOXYGEN__
52
53  #if !LWESP_CFG_INPUT_USE_PROCESS
54  #error "LWESP_CFG_INPUT_USE_PROCESS must be enabled in `lwesp_config.h` to use this_
  ↪ driver."
55  #endif /* LWESP_CFG_INPUT_USE_PROCESS */
56
57  #if !defined(LWESP_USART_DMA_RX_BUFF_SIZE)
58  #define LWESP_USART_DMA_RX_BUFF_SIZE      0x1000
59  #endif /* !defined(LWESP_USART_DMA_RX_BUFF_SIZE) */
60
61  #if !defined(LWESP_MEM_SIZE)
62  #define LWESP_MEM_SIZE                    0x1000
63  #endif /* !defined(LWESP_MEM_SIZE) */
64
65  #if !defined(LWESP_USART_RDR_NAME)
66  #define LWESP_USART_RDR_NAME              RDR
67  #endif /* !defined(LWESP_USART_RDR_NAME) */
68
69  /* USART memory */
70  static uint8_t    usart_mem[LWESP_USART_DMA_RX_BUFF_SIZE];
71  static uint8_t    is_running, initialized;
72  static size_t     old_pos;
73
74  /* USART thread */
75  static void usart_ll_thread(void* arg);
76  static osThreadId_t usart_ll_thread_id;
77
78  /* Message queue */
79  static osMessageQueueId_t usart_ll_mbox_id;
80
81  /**
82   * \brief          USART data processing
83   */
84  static void
85  usart_ll_thread(void* arg) {
86      size_t pos;
87
88      LWESP_UNUSED(arg);
89
90      while (1) {
91          void* d;
92          /* Wait for the event message from DMA or USART */
93          osMessageQueueGet(usart_ll_mbox_id, &d, NULL, osWaitForever);
94
95          /* Read data */

```

(continues on next page)

(continued from previous page)

```

96 #if defined(LWESP_USART_DMA_RX_STREAM)
97     pos = sizeof(usart_mem) - LL_DMA_GetDataLength(LWESP_USART_DMA, LWESP_USART_
↳DMA_RX_STREAM);
98 #else
99     pos = sizeof(usart_mem) - LL_DMA_GetDataLength(LWESP_USART_DMA, LWESP_USART_
↳DMA_RX_CH);
100 #endif /* defined(LWESP_USART_DMA_RX_STREAM) */
101     if (pos != old_pos && is_running) {
102         if (pos > old_pos) {
103             lwesp_input_process(&usart_mem[old_pos], pos - old_pos);
104         } else {
105             lwesp_input_process(&usart_mem[old_pos], sizeof(usart_mem) - old_pos);
106             if (pos > 0) {
107                 lwesp_input_process(&usart_mem[0], pos);
108             }
109         }
110         old_pos = pos;
111         if (old_pos == sizeof(usart_mem)) {
112             old_pos = 0;
113         }
114     }
115 }
116 }
117
118 /**
119  * \brief          Configure UART using DMA for receive in double buffer mode and
↳IDLE line detection
120  */
121 static void
122 configure_uart(uint32_t baudrate) {
123     static LL_USART_InitTypeDef usart_init;
124     static LL_DMA_InitTypeDef dma_init;
125     LL_GPIO_InitTypeDef gpio_init;
126
127     if (!initialized) {
128         /* Enable peripheral clocks */
129         LWESP_USART_CLK;
130         LWESP_USART_DMA_CLK;
131         LWESP_USART_TX_PORT_CLK;
132         LWESP_USART_RX_PORT_CLK;
133
134         #if defined(LWESP_RESET_PIN)
135             LWESP_RESET_PORT_CLK;
136         #endif /* defined(LWESP_RESET_PIN) */
137
138         #if defined(LWESP_GPIO0_PIN)
139             LWESP_GPIO0_PORT_CLK;
140         #endif /* defined(LWESP_GPIO0_PIN) */
141
142         #if defined(LWESP_GPIO2_PIN)
143             LWESP_GPIO2_PORT_CLK;
144         #endif /* defined(LWESP_GPIO2_PIN) */
145
146         #if defined(LWESP_CH_PD_PIN)
147             LWESP_CH_PD_PORT_CLK;
148         #endif /* defined(LWESP_CH_PD_PIN) */
149

```

(continues on next page)

```

150     /* Global pin configuration */
151     LL_GPIO_StructInit(&gpio_init);
152     gpio_init.OutputType = LL_GPIO_OUTPUT_PUSHPULL;
153     gpio_init.Pull = LL_GPIO_PULL_UP;
154     gpio_init.Speed = LL_GPIO_SPEED_FREQ_VERY_HIGH;
155     gpio_init.Mode = LL_GPIO_MODE_OUTPUT;
156
157     #if defined(LWESP_RESET_PIN)
158         /* Configure RESET pin */
159         gpio_init.Pin = LWESP_RESET_PIN;
160         LL_GPIO_Init(LWESP_RESET_PORT, &gpio_init);
161     #endif /* defined(LWESP_RESET_PIN) */
162
163     #if defined(LWESP_GPIO0_PIN)
164         /* Configure GPIO0 pin */
165         gpio_init.Pin = LWESP_GPIO0_PIN;
166         LL_GPIO_Init(LWESP_GPIO0_PORT, &gpio_init);
167         LL_GPIO_SetOutputPin(LWESP_GPIO0_PORT, LWESP_GPIO0_PIN);
168     #endif /* defined(LWESP_GPIO0_PIN) */
169
170     #if defined(LWESP_GPIO2_PIN)
171         /* Configure GPIO2 pin */
172         gpio_init.Pin = LWESP_GPIO2_PIN;
173         LL_GPIO_Init(LWESP_GPIO2_PORT, &gpio_init);
174         LL_GPIO_SetOutputPin(LWESP_GPIO2_PORT, LWESP_GPIO2_PIN);
175     #endif /* defined(LWESP_GPIO2_PIN) */
176
177     #if defined(LWESP_CH_PD_PIN)
178         /* Configure CH_PD pin */
179         gpio_init.Pin = LWESP_CH_PD_PIN;
180         LL_GPIO_Init(LWESP_CH_PD_PORT, &gpio_init);
181         LL_GPIO_SetOutputPin(LWESP_CH_PD_PORT, LWESP_CH_PD_PIN);
182     #endif /* defined(LWESP_CH_PD_PIN) */
183
184     /* Configure USART pins */
185     gpio_init.Mode = LL_GPIO_MODE_ALTERNATE;
186
187     /* TX PIN */
188     gpio_init.Alternate = LWESP_USART_TX_PIN_AF;
189     gpio_init.Pin = LWESP_USART_TX_PIN;
190     LL_GPIO_Init(LWESP_USART_TX_PORT, &gpio_init);
191
192     /* RX PIN */
193     gpio_init.Alternate = LWESP_USART_RX_PIN_AF;
194     gpio_init.Pin = LWESP_USART_RX_PIN;
195     LL_GPIO_Init(LWESP_USART_RX_PORT, &gpio_init);
196
197     /* Configure UART */
198     LL_USART_DeInit(LWESP_USART);
199     LL_USART_StructInit(&usart_init);
200     usart_init.BaudRate = baudrate;
201     usart_init.DataWidth = LL_USART_DATAWIDTH_8B;
202     usart_init.HardwareFlowControl = LL_USART_HWCONTROL_NONE;
203     usart_init.OverSampling = LL_USART_OVERSAMPLING_16;
204     usart_init.Parity = LL_USART_PARITY_NONE;
205     usart_init.StopBits = LL_USART_STOPBITS_1;
206     usart_init.TransferDirection = LL_USART_DIRECTION_TX_RX;

```

(continues on next page)

(continued from previous page)

```

207     LL_USART_Init(LWESP_USART, &usart_init);
208
209     /* Enable USART interrupts and DMA request */
210     LL_USART_EnableIT_IDLE(LWESP_USART);
211     LL_USART_EnableIT_PE(LWESP_USART);
212     LL_USART_EnableIT_ERROR(LWESP_USART);
213     LL_USART_EnableDMAReq_RX(LWESP_USART);
214
215     /* Enable USART interrupts */
216     NVIC_SetPriority(LWESP_USART_IRQ, NVIC_EncodePriority(NVIC_
↳GetPriorityGrouping(), 0x07, 0x00));
217     NVIC_EnableIRQ(LWESP_USART_IRQ);
218
219     /* Configure DMA */
220     is_running = 0;
221 #if defined(LWESP_USART_DMA_RX_STREAM)
222     LL_DMA_DeInit(LWESP_USART_DMA, LWESP_USART_DMA_RX_STREAM);
223     dma_init.Channel = LWESP_USART_DMA_RX_CH;
224 #else
225     LL_DMA_DeInit(LWESP_USART_DMA, LWESP_USART_DMA_RX_CH);
226     dma_init.PeriphRequest = LWESP_USART_DMA_RX_REQ_NUM;
227 #endif /* defined(LWESP_USART_DMA_RX_STREAM) */
228     dma_init.PeriphOrM2MSrcAddress = (uint32_t)&LWESP_USART->LWESP_USART_RDR_NAME;
229     dma_init.MemoryOrM2MDstAddress = (uint32_t)usart_mem;
230     dma_init.Direction = LL_DMA_DIRECTION_PERIPH_TO_MEMORY;
231     dma_init.Mode = LL_DMA_MODE_CIRCULAR;
232     dma_init.PeriphOrM2MSrcIncMode = LL_DMA_PERIPH_NOINCREMENT;
233     dma_init.MemoryOrM2MDstIncMode = LL_DMA_MEMORY_INCREMENT;
234     dma_init.PeriphOrM2MSrcDataSize = LL_DMA_PDATAALIGN_BYTE;
235     dma_init.MemoryOrM2MDstDataSize = LL_DMA_MDATAALIGN_BYTE;
236     dma_init.NbData = sizeof(usart_mem);
237     dma_init.Priority = LL_DMA_PRIORITY_MEDIUM;
238 #if defined(LWESP_USART_DMA_RX_STREAM)
239     LL_DMA_Init(LWESP_USART_DMA, LWESP_USART_DMA_RX_STREAM, &dma_init);
240 #else
241     LL_DMA_Init(LWESP_USART_DMA, LWESP_USART_DMA_RX_CH, &dma_init);
242 #endif /* defined(LWESP_USART_DMA_RX_STREAM) */
243
244     /* Enable DMA interrupts */
245 #if defined(LWESP_USART_DMA_RX_STREAM)
246     LL_DMA_EnableIT_HT(LWESP_USART_DMA, LWESP_USART_DMA_RX_STREAM);
247     LL_DMA_EnableIT_TC(LWESP_USART_DMA, LWESP_USART_DMA_RX_STREAM);
248     LL_DMA_EnableIT_TE(LWESP_USART_DMA, LWESP_USART_DMA_RX_STREAM);
249     LL_DMA_EnableIT_FE(LWESP_USART_DMA, LWESP_USART_DMA_RX_STREAM);
250     LL_DMA_EnableIT_DME(LWESP_USART_DMA, LWESP_USART_DMA_RX_STREAM);
251 #else
252     LL_DMA_EnableIT_HT(LWESP_USART_DMA, LWESP_USART_DMA_RX_CH);
253     LL_DMA_EnableIT_TC(LWESP_USART_DMA, LWESP_USART_DMA_RX_CH);
254     LL_DMA_EnableIT_TE(LWESP_USART_DMA, LWESP_USART_DMA_RX_CH);
255 #endif /* defined(LWESP_USART_DMA_RX_STREAM) */
256
257     /* Enable DMA interrupts */
258     NVIC_SetPriority(LWESP_USART_DMA_RX_IRQ, NVIC_EncodePriority(NVIC_
↳GetPriorityGrouping(), 0x07, 0x00));
259     NVIC_EnableIRQ(LWESP_USART_DMA_RX_IRQ);
260
261     old_pos = 0;

```

(continues on next page)

```

262     is_running = 1;
263
264     /* Start DMA and USART */
265 #if defined(LWESP_USART_DMA_RX_STREAM)
266     LL_DMA_EnableStream(LWESP_USART_DMA, LWESP_USART_DMA_RX_STREAM);
267 #else
268     LL_DMA_EnableChannel(LWESP_USART_DMA, LWESP_USART_DMA_RX_CH);
269 #endif /* defined(LWESP_USART_DMA_RX_STREAM) */
270     LL_USART_Enable(LWESP_USART);
271 } else {
272     osDelay(10);
273     LL_USART_Disable(LWESP_USART);
274     usart_init.BaudRate = baudrate;
275     LL_USART_Init(LWESP_USART, &usart_init);
276     LL_USART_Enable(LWESP_USART);
277 }
278
279 /* Create mbox and start thread */
280 if (usart_ll_mbox_id == NULL) {
281     usart_ll_mbox_id = osMessageQueueNew(10, sizeof(void*), NULL);
282 }
283 if (usart_ll_thread_id == NULL) {
284     const osThreadAttr_t attr = {
285         .stack_size = 1024
286     };
287     usart_ll_thread_id = osThreadNew(usart_ll_thread, usart_ll_mbox_id, &attr);
288 }
289 }
290
291 #if defined(LWESP_RESET_PIN)
292 /**
293  * \brief      Hardware reset callback
294  */
295 static uint8_t
296 reset_device(uint8_t state) {
297     if (state) { /* Activate reset line */
298         LL_GPIO_ResetOutputPin(LWESP_RESET_PORT, LWESP_RESET_PIN);
299     } else {
300         LL_GPIO_SetOutputPin(LWESP_RESET_PORT, LWESP_RESET_PIN);
301     }
302     return 1;
303 }
304 #endif /* defined(LWESP_RESET_PIN) */
305
306 /**
307  * \brief      Send data to ESP device
308  * \param[in]  data: Pointer to data to send
309  * \param[in]  len: Number of bytes to send
310  * \return     Number of bytes sent
311  */
312 static size_t
313 send_data(const void* data, size_t len) {
314     const uint8_t* d = data;
315
316     for (size_t i = 0; i < len; ++i, ++d) {
317         LL_USART_TransmitData8(LWESP_USART, *d);
318         while (!LL_USART_IsActiveFlag_TXE(LWESP_USART)) {}

```

(continues on next page)

(continued from previous page)

```

319     }
320     return len;
321 }
322
323 /**
324  * \brief          Callback function called from initialization process
325  */
326 lwespr_t
327 lwesp_ll_init(lwesp_ll_t* ll) {
328 #if !LWESP_CFG_MEM_CUSTOM
329     static uint8_t memory[LWESP_MEM_SIZE];
330     lwesp_mem_region_t mem_regions[] = {
331         { memory, sizeof(memory) }
332     };
333
334     if (!initialized) {
335         lwesp_mem_assignmemory(mem_regions, LWESP_ARRAYSIZE(mem_regions)); /* Assign_
↪memory for allocations */
336     }
337 #endif /* !LWESP_CFG_MEM_CUSTOM */
338
339     if (!initialized) {
340         ll->send_fn = send_data; /* Set callback function to send data_
↪*/
341 #if defined(LWESP_RESET_PIN)
342         ll->reset_fn = reset_device; /* Set callback for hardware reset */
343 #endif /* defined(LWESP_RESET_PIN) */
344     }
345
346     configure_uart(ll->uart.baudrate); /* Initialize UART for communication_
↪*/
347     initialized = 1;
348     return lwespOK;
349 }
350
351 /**
352  * \brief          Callback function to de-init low-level communication part
353  */
354 lwespr_t
355 lwesp_ll_deinit(lwesp_ll_t* ll) {
356     if (usart_ll_mbox_id != NULL) {
357         osMessageQueueId_t tmp = usart_ll_mbox_id;
358         usart_ll_mbox_id = NULL;
359         osMessageQueueDelete(tmp);
360     }
361     if (usart_ll_thread_id != NULL) {
362         osThreadId_t tmp = usart_ll_thread_id;
363         usart_ll_thread_id = NULL;
364         osThreadTerminate(tmp);
365     }
366     initialized = 0;
367     LWESP_UNUSED(ll);
368     return lwespOK;
369 }
370
371 /**
372  * \brief          UART global interrupt handler

```

(continues on next page)

(continued from previous page)

```

373  */
374  void
375  LWESP_USART_IRQHANDLER(void) {
376      LL_USART_ClearFlag_IDLE(LWESP_USART);
377      LL_USART_ClearFlag_PE(LWESP_USART);
378      LL_USART_ClearFlag_FE(LWESP_USART);
379      LL_USART_ClearFlag_ORE(LWESP_USART);
380      LL_USART_ClearFlag_NE(LWESP_USART);
381
382      if (usart_ll_mbox_id != NULL) {
383          void* d = (void*)1;
384          osMessageQueuePut(usart_ll_mbox_id, &d, 0, 0);
385      }
386  }
387
388  /**
389   * \brief          UART DMA stream/channel handler
390   */
391  void
392  LWESP_USART_DMA_RX_IRQHANDLER(void) {
393      LWESP_USART_DMA_RX_CLEAR_TC;
394      LWESP_USART_DMA_RX_CLEAR_HT;
395
396      if (usart_ll_mbox_id != NULL) {
397          void* d = (void*)1;
398          osMessageQueuePut(usart_ll_mbox_id, &d, 0, 0);
399      }
400  }
401
402  #endif /* !__DOXYGEN__ */

```

Example: System functions for WIN32

Listing 10: Actual header implementation of system functions for WIN32

```

1  /**
2   * \file          lwesp_sys_port.h
3   * \brief        WIN32 based system file implementation
4   */
5
6  /**
7   * Copyright (c) 2020 Tilen MAJERLE
8   *
9   * Permission is hereby granted, free of charge, to any person
10  * obtaining a copy of this software and associated documentation
11  * files (the "Software"), to deal in the Software without restriction,
12  * including without limitation the rights to use, copy, modify, merge,
13  * publish, distribute, sublicense, and/or sell copies of the Software,
14  * and to permit persons to whom the Software is furnished to do so,
15  * subject to the following conditions:
16  *
17  * The above copyright notice and this permission notice shall be
18  * included in all copies or substantial portions of the Software.
19  *

```

(continues on next page)

(continued from previous page)

```

20 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
21 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
22 * OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
23 * AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
24 * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
25 * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
26 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
27 * OTHER DEALINGS IN THE SOFTWARE.
28 *
29 * This file is part of LwESP - Lightweight ESP-AT parser library.
30 *
31 * Author:          Tilen MAJERLE <tilen@majerle.eu>
32 * Version:         $_version_$
33 */
34 #ifndef LWESP_HDR_SYSTEM_PORT_H
35 #define LWESP_HDR_SYSTEM_PORT_H
36
37 #include <stdint.h>
38 #include <stdlib.h>
39 #include "lwesp/lwesp_opt.h"
40 #include "windows.h"
41
42 #ifdef __cplusplus
43 extern "C" {
44 #endif /* __cplusplus */
45
46 #if LWESP_CFG_OS && !__DOXYGEN__
47
48 typedef HANDLE          lwesp_sys_mutex_t;
49 typedef HANDLE          lwesp_sys_sem_t;
50 typedef HANDLE          lwesp_sys_mbox_t;
51 typedef HANDLE          lwesp_sys_thread_t;
52 typedef int             lwesp_sys_thread_prio_t;
53
54 #define LWESP_SYS_MBOX_NULL          ((HANDLE)0)
55 #define LWESP_SYS_SEM_NULL           ((HANDLE)0)
56 #define LWESP_SYS_MUTEX_NULL        ((HANDLE)0)
57 #define LWESP_SYS_TIMEOUT            (INFINITE)
58 #define LWESP_SYS_THREAD_PRIO       (0)
59 #define LWESP_SYS_THREAD_SS         (1024)
60
61 #endif /* LWESP_CFG_OS && !__DOXYGEN__ */
62
63 #ifdef __cplusplus
64 }
65 #endif /* __cplusplus */
66
67 #endif /* LWESP_HDR_SYSTEM_PORT_H */

```

Listing 11: Actual implementation of system functions for WIN32

```

1 /**
2  * \file          lwesp_sys_win32.c
3  * \brief        System dependant functions for WIN32
4  */
5

```

(continues on next page)

```

6  /*
7  * Copyright (c) 2020 Tilen MAJERLE
8  *
9  * Permission is hereby granted, free of charge, to any person
10 * obtaining a copy of this software and associated documentation
11 * files (the "Software"), to deal in the Software without restriction,
12 * including without limitation the rights to use, copy, modify, merge,
13 * publish, distribute, sublicense, and/or sell copies of the Software,
14 * and to permit persons to whom the Software is furnished to do so,
15 * subject to the following conditions:
16 *
17 * The above copyright notice and this permission notice shall be
18 * included in all copies or substantial portions of the Software.
19 *
20 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
21 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
22 * OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
23 * AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
24 * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
25 * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
26 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
27 * OTHER DEALINGS IN THE SOFTWARE.
28 *
29 * This file is part of LwESP - Lightweight ESP-AT parser library.
30 *
31 * Author:          Tilen MAJERLE <tilen@majerle.eu>
32 * Version:         $_version_$
33 */
34 #include <string.h>
35 #include <stdlib.h>
36 #include "system/lwesp_sys.h"
37 #include "windows.h"
38
39 #if !__DOXYGEN__
40
41 /**
42  * \brief          Custom message queue implementation for WIN32
43  */
44 typedef struct {
45     lwesp_sys_sem_t sem_not_empty;          /*!< Semaphore indicates not empty */
46     lwesp_sys_sem_t sem_not_full;         /*!< Semaphore indicates not full */
47     lwesp_sys_sem_t sem;                  /*!< Semaphore to lock access */
48     size_t in, out, size;
49     void* entries[1];
50 } win32_mbox_t;
51
52 static LARGE_INTEGER freq, sys_start_time;
53 static lwesp_sys_mutex_t sys_mutex;       /* Mutex ID for main protection */
54
55 /**
56  * \brief          Check if message box is full
57  * \param[in]     m: Message box handle
58  * \return        1 if full, 0 otherwise
59  */
60 static uint8_t
61 mbox_is_full(win32_mbox_t* m) {
62     size_t size = 0;

```

(continues on next page)

(continued from previous page)

```

63     if (m->in > m->out) {
64         size = (m->in - m->out);
65     } else if (m->out > m->in) {
66         size = m->size - m->out + m->in;
67     }
68     return size == m->size - 1;
69 }
70
71 /**
72  * \brief      Check if message box is empty
73  * \param[in]  m: Message box handle
74  * \return     1 if empty, 0 otherwise
75  */
76 static uint8_t
77 mbox_is_empty(win32_mbox_t* m) {
78     return m->in == m->out;
79 }
80
81 /**
82  * \brief      Get current kernel time in units of milliseconds
83  */
84 static uint32_t
85 osKernelSysTick(void) {
86     LONGLONG ret;
87     LARGE_INTEGER now;
88
89     QueryPerformanceFrequency(&freq);          /* Get frequency */
90     QueryPerformanceCounter(&now);             /* Get current time */
91     ret = now.QuadPart - sys_start_time.QuadPart;
92     return (uint32_t)((ret) * 1000) / freq.QuadPart;
93 }
94
95 uint8_t
96 lwesp_sys_init(void) {
97     QueryPerformanceFrequency(&freq);
98     QueryPerformanceCounter(&sys_start_time);
99
100     lwesp_sys_mutex_create(&sys_mutex);
101     return 1;
102 }
103
104 uint32_t
105 lwesp_sys_now(void) {
106     return osKernelSysTick();
107 }
108
109 #if LWESP_CFG_OS
110 uint8_t
111 lwesp_sys_protect(void) {
112     lwesp_sys_mutex_lock(&sys_mutex);
113     return 1;
114 }
115
116 uint8_t
117 lwesp_sys_unprotect(void) {
118     lwesp_sys_mutex_unlock(&sys_mutex);
119     return 1;

```

(continues on next page)

```
120 }
121
122 uint8_t
123 lwesp_sys_mutex_create(lwesp_sys_mutex_t* p) {
124     *p = CreateMutex(NULL, FALSE, NULL);
125     return *p != NULL;
126 }
127
128 uint8_t
129 lwesp_sys_mutex_delete(lwesp_sys_mutex_t* p) {
130     return CloseHandle(*p);
131 }
132
133 uint8_t
134 lwesp_sys_mutex_lock(lwesp_sys_mutex_t* p) {
135     DWORD ret;
136     ret = WaitForSingleObject(*p, INFINITE);
137     if (ret != WAIT_OBJECT_0) {
138         return 0;
139     }
140     return 1;
141 }
142
143 uint8_t
144 lwesp_sys_mutex_unlock(lwesp_sys_mutex_t* p) {
145     return ReleaseMutex(*p);
146 }
147
148 uint8_t
149 lwesp_sys_mutex_isvalid(lwesp_sys_mutex_t* p) {
150     return p != NULL && *p != NULL;
151 }
152
153 uint8_t
154 lwesp_sys_mutex_invalid(lwesp_sys_mutex_t* p) {
155     *p = LWESP_SYS_MUTEX_NULL;
156     return 1;
157 }
158
159 uint8_t
160 lwesp_sys_sem_create(lwesp_sys_sem_t* p, uint8_t cnt) {
161     HANDLE h;
162     h = CreateSemaphore(NULL, !!cnt, 1, NULL);
163     *p = h;
164     return *p != NULL;
165 }
166
167 uint8_t
168 lwesp_sys_sem_delete(lwesp_sys_sem_t* p) {
169     return CloseHandle(*p);
170 }
171
172 uint32_t
173 lwesp_sys_sem_wait(lwesp_sys_sem_t* p, uint32_t timeout) {
174     DWORD ret;
175     uint32_t tick = osKernelSysTick();
176
```

(continues on next page)

(continued from previous page)

```

177     if (timeout == 0) {
178         ret = WaitForSingleObject(*p, INFINITE);
179         return 1;
180     } else {
181         ret = WaitForSingleObject(*p, timeout);
182         if (ret == WAIT_OBJECT_0) {
183             return 1;
184         } else {
185             return LWESP_SYS_TIMEOUT;
186         }
187     }
188 }
189
190 uint8_t
191 lwesp_sys_sem_release(lwesp_sys_sem_t* p) {
192     return ReleaseSemaphore(*p, 1, NULL);
193 }
194
195 uint8_t
196 lwesp_sys_sem_isvalid(lwesp_sys_sem_t* p) {
197     return p != NULL && *p != NULL;
198 }
199
200 uint8_t
201 lwesp_sys_sem_invalid(lwesp_sys_sem_t* p) {
202     *p = LWESP_SYS_SEM_NULL;
203     return 1;
204 }
205
206 uint8_t
207 lwesp_sys_mbox_create(lwesp_sys_mbox_t* b, size_t size) {
208     win32_mbox_t* mbox;
209
210     *b = 0;
211
212     mbox = malloc(sizeof(*mbox) + size * sizeof(void*));
213     if (mbox != NULL) {
214         memset(mbox, 0x00, sizeof(*mbox));
215         mbox->size = size + 1;           /* Set it to 1 more as cyclic buffer_
↳has only one less than size */
216         lwesp_sys_sem_create(&mbox->sem, 1);
217         lwesp_sys_sem_create(&mbox->sem_not_empty, 0);
218         lwesp_sys_sem_create(&mbox->sem_not_full, 0);
219         *b = mbox;
220     }
221     return *b != NULL;
222 }
223
224 uint8_t
225 lwesp_sys_mbox_delete(lwesp_sys_mbox_t* b) {
226     win32_mbox_t* mbox = *b;
227     lwesp_sys_sem_delete(&mbox->sem);
228     lwesp_sys_sem_delete(&mbox->sem_not_full);
229     lwesp_sys_sem_delete(&mbox->sem_not_empty);
230     free(mbox);
231     return 1;
232 }

```

(continues on next page)

```

233
234 uint32_t
235 lwesp_sys_mbox_put(lwesp_sys_mbox_t* b, void* m) {
236     win32_mbox_t* mbox = *b;
237     uint32_t time = osKernelSysTick();           /* Get start time */
238
239     lwesp_sys_sem_wait(&mbox->sem, 0);           /* Wait for access */
240
241     /*
242      * Since function is blocking until ready to write something to queue,
243      * wait and release the semaphores to allow other threads
244      * to process the queue before we can write new value.
245      */
246     while (mbox_is_full(mbox)) {
247         lwesp_sys_sem_release(&mbox->sem);       /* Release semaphore */
248         lwesp_sys_sem_wait(&mbox->sem_not_full, 0); /* Wait for semaphore indicating_
↳not full */
249         lwesp_sys_sem_wait(&mbox->sem, 0);       /* Wait availability again */
250     }
251     mbox->entries[mbox->in] = m;
252     if (++mbox->in >= mbox->size) {
253         mbox->in = 0;
254     }
255     lwesp_sys_sem_release(&mbox->sem_not_empty); /* Signal non-empty state */
256     lwesp_sys_sem_release(&mbox->sem);           /* Release access for other threads */
257     return osKernelSysTick() - time;
258 }
259
260 uint32_t
261 lwesp_sys_mbox_get(lwesp_sys_mbox_t* b, void** m, uint32_t timeout) {
262     win32_mbox_t* mbox = *b;
263     uint32_t time;
264
265     time = osKernelSysTick();
266
267     /* Get exclusive access to message queue */
268     if (lwesp_sys_sem_wait(&mbox->sem, timeout) == LWESP_SYS_TIMEOUT) {
269         return LWESP_SYS_TIMEOUT;
270     }
271     while (mbox_is_empty(mbox)) {
272         lwesp_sys_sem_release(&mbox->sem);
273         if (lwesp_sys_sem_wait(&mbox->sem_not_empty, timeout) == LWESP_SYS_TIMEOUT) {
274             return LWESP_SYS_TIMEOUT;
275         }
276         lwesp_sys_sem_wait(&mbox->sem, timeout);
277     }
278     *m = mbox->entries[mbox->out];
279     if (++mbox->out >= mbox->size) {
280         mbox->out = 0;
281     }
282     lwesp_sys_sem_release(&mbox->sem_not_full);
283     lwesp_sys_sem_release(&mbox->sem);
284
285     return osKernelSysTick() - time;
286 }
287
288 uint8_t

```

(continues on next page)

(continued from previous page)

```

289 lwesp_sys_mbox_putnow(lwesp_sys_mbox_t* b, void* m) {
290     win32_mbox_t* mbox = *b;
291
292     lwesp_sys_sem_wait(&mbox->sem, 0);
293     if (mbox_is_full(mbox)) {
294         lwesp_sys_sem_release(&mbox->sem);
295         return 0;
296     }
297     mbox->entries[mbox->in] = m;
298     if (mbox->in == mbox->out) {
299         lwesp_sys_sem_release(&mbox->sem_not_empty);
300     }
301     if (++mbox->in >= mbox->size) {
302         mbox->in = 0;
303     }
304     lwesp_sys_sem_release(&mbox->sem);
305     return 1;
306 }
307
308 uint8_t
309 lwesp_sys_mbox_getnow(lwesp_sys_mbox_t* b, void** m) {
310     win32_mbox_t* mbox = *b;
311
312     lwesp_sys_sem_wait(&mbox->sem, 0);           /* Wait exclusive access */
313     if (mbox->in == mbox->out) {
314         lwesp_sys_sem_release(&mbox->sem);       /* Release access */
315         return 0;
316     }
317
318     *m = mbox->entries[mbox->out];
319     if (++mbox->out >= mbox->size) {
320         mbox->out = 0;
321     }
322     lwesp_sys_sem_release(&mbox->sem_not_full); /* Queue not full anymore */
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, size_t stack_size, lwesp_sys_thread_prio_t prio) {
340     HANDLE h;
341     DWORD id;
342     h = CreateThread(0, 0, (LPTHREAD_START_ROUTINE)thread_func, arg, 0, &id);
343     if (t != NULL) {
344         *t = h;

```

(continues on next page)

(continued from previous page)

```

345     }
346     return h != NULL;
347 }
348
349 uint8_t
350 lwesp_sys_thread_terminate(lwesp_sys_thread_t* t) {
351     HANDLE h = NULL;
352
353     if (t == NULL) {                /* Shall we terminate ourselves? */
354         h = GetCurrentThread();      /* Get current thread handle */
355     } else {                          /* We have known thread, find handle_
↳by looking at ID */
356         h = *t;
357     }
358     TerminateThread(h, 0);
359     return 1;
360 }
361
362 uint8_t
363 lwesp_sys_thread_yield(void) {
364     /* Not implemented */
365     return 1;
366 }
367
368 #endif /* LWESP_CFG_OS */
369 #endif /* !__DOXYGEN__ */

```

Example: System functions for CMSIS-OS

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

```

1  /**
2   * \file          lwesp_sys_port.h
3   * \brief        CMSIS-OS based system file
4   */
5
6  /**
7   * Copyright (c) 2020 Tilen MAJERLE
8   *
9   * Permission is hereby granted, free of charge, to any person
10  * obtaining a copy of this software and associated documentation
11  * files (the "Software"), to deal in the Software without restriction,
12  * including without limitation the rights to use, copy, modify, merge,
13  * publish, distribute, sublicense, and/or sell copies of the Software,
14  * and to permit persons to whom the Software is furnished to do so,
15  * subject to the following conditions:
16  *
17  * The above copyright notice and this permission notice shall be
18  * included in all copies or substantial portions of the Software.
19  *
20  * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
21  * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
22  * OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
23  * AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT

```

(continues on next page)

(continued from previous page)

```

24  * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
25  * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
26  * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
27  * OTHER DEALINGS IN THE SOFTWARE.
28  *
29  * This file is part of LwESP - Lightweight ESP-AT parser library.
30  *
31  * Author:          Tilen MAJERLE <tilen@majerle.eu>
32  * Version:         $_version_$
33  */
34  #ifndef LWESP_HDR_SYSTEM_PORT_H
35  #define LWESP_HDR_SYSTEM_PORT_H
36
37  #include <stdint.h>
38  #include <stdlib.h>
39  #include "lwesp/lwesp_opt.h"
40  #include "cmsis_os.h"
41
42  #ifdef __cplusplus
43  extern "C" {
44  #endif /* __cplusplus */
45
46  #if LWESP_CFG_OS && !__DOXYGEN__
47
48  typedef osMutexId_t          lwesp_sys_mutex_t;
49  typedef osSemaphoreId_t     lwesp_sys_sem_t;
50  typedef osMessageQueueId_t  lwesp_sys_mbox_t;
51  typedef osThreadId_t        lwesp_sys_thread_t;
52  typedef osPriority_t         lwesp_sys_thread_prio_t;
53
54  #define LWESP_SYS_MUTEX_NULL          ((lwesp_sys_mutex_t)0)
55  #define LWESP_SYS_SEM_NULL           ((lwesp_sys_sem_t)0)
56  #define LWESP_SYS_MBOX_NULL          ((lwesp_sys_mbox_t)0)
57  #define LWESP_SYS_TIMEOUT            ((uint32_t)osWaitForever)
58  #define LWESP_SYS_THREAD_PRIO       (osPriorityNormal)
59  #define LWESP_SYS_THREAD_SS         (512)
60
61  #endif /* LWESP_CFG_OS && !__DOXYGEN__ */
62
63  #ifdef __cplusplus
64  }
65  #endif /* __cplusplus */
66
67  #endif /* LWESP_HDR_SYSTEM_PORT_H */

```

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

```

1  /**
2  * \file          lwesp_sys_cmsis_os.c
3  * \brief        System dependent functions for CMSIS based operating system
4  */
5
6  /**
7  * Copyright (c) 2020 Tilen MAJERLE
8  */

```

(continues on next page)

```
9  * Permission is hereby granted, free of charge, to any person
10 * obtaining a copy of this software and associated documentation
11 * files (the "Software"), to deal in the Software without restriction,
12 * including without limitation the rights to use, copy, modify, merge,
13 * publish, distribute, sublicense, and/or sell copies of the Software,
14 * and to permit persons to whom the Software is furnished to do so,
15 * subject to the following conditions:
16 *
17 * The above copyright notice and this permission notice shall be
18 * included in all copies or substantial portions of the Software.
19 *
20 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
21 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
22 * OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
23 * AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
24 * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
25 * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
26 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
27 * OTHER DEALINGS IN THE SOFTWARE.
28 *
29 * This file is part of LwESP - Lightweight ESP-AT parser library.
30 *
31 * Author:          Tilen MAJERLE <tilen@majerle.eu>
32 * Version:         $_version_$
33 */
34 #include "system/lwesp_sys.h"
35 #include "cmsis_os.h"
36
37 #if !__DOXYGEN__
38
39 static osMutexId_t sys_mutex;
40
41 uint8_t
42 lwesp_sys_init(void) {
43     lwesp_sys_mutex_create(&sys_mutex);
44     return 1;
45 }
46
47 uint32_t
48 lwesp_sys_now(void) {
49     return osKernelSysTick();
50 }
51
52 uint8_t
53 lwesp_sys_protect(void) {
54     lwesp_sys_mutex_lock(&sys_mutex);
55     return 1;
56 }
57
58 uint8_t
59 lwesp_sys_unprotect(void) {
60     lwesp_sys_mutex_unlock(&sys_mutex);
61     return 1;
62 }
63
64 uint8_t
65 lwesp_sys_mutex_create(lwesp_sys_mutex_t* p) {
```

(continues on next page)

(continued from previous page)

```

66     const osMutexAttr_t attr = {
67         .attr_bits = osMutexRecursive
68     };
69     *p = osMutexNew(&attr);
70     return *p != NULL;
71 }
72
73 uint8_t
74 lwesp_sys_mutex_delete(lwesp_sys_mutex_t* p) {
75     return osMutexDelete(*p) == osOK;
76 }
77
78 uint8_t
79 lwesp_sys_mutex_lock(lwesp_sys_mutex_t* p) {
80     return osMutexAcquire(*p, osWaitForever) == osOK;
81 }
82
83 uint8_t
84 lwesp_sys_mutex_unlock(lwesp_sys_mutex_t* p) {
85     return osMutexRelease(*p) == osOK;
86 }
87
88 uint8_t
89 lwesp_sys_mutex_isvalid(lwesp_sys_mutex_t* p) {
90     return p != NULL && *p != NULL;
91 }
92
93 uint8_t
94 lwesp_sys_mutex_invalid(lwesp_sys_mutex_t* p) {
95     *p = LWESP_SYS_MUTEX_NULL;
96     return 1;
97 }
98
99 uint8_t
100 lwesp_sys_sem_create(lwesp_sys_sem_t* p, uint8_t cnt) {
101     return (*p = osSemaphoreNew(1, cnt > 0 ? 1 : 0, NULL)) != NULL;
102 }
103
104 uint8_t
105 lwesp_sys_sem_delete(lwesp_sys_sem_t* p) {
106     return osSemaphoreDelete(*p) == osOK;
107 }
108
109 uint32_t
110 lwesp_sys_sem_wait(lwesp_sys_sem_t* p, uint32_t timeout) {
111     uint32_t tick = osKernelSysTick();
112     return (osSemaphoreAcquire(*p, timeout == 0 ? osWaitForever : timeout) == osOK) ?
113     ↪ (osKernelSysTick() - tick) : LWESP_SYS_TIMEOUT;
114 }
115
116 uint8_t
117 lwesp_sys_sem_release(lwesp_sys_sem_t* p) {
118     return osSemaphoreRelease(*p) == osOK;
119 }
120
121 uint8_t
122 lwesp_sys_sem_isvalid(lwesp_sys_sem_t* p) {

```

(continues on next page)

```

122     return p != NULL && *p != NULL;
123 }
124
125 uint8_t
126 lwesp_sys_sem_invalid(lwesp_sys_sem_t* p) {
127     *p = LWESP_SYS_SEM_NULL;
128     return 1;
129 }
130
131 uint8_t
132 lwesp_sys_mbox_create(lwesp_sys_mbox_t* b, size_t size) {
133     return (*b = osMessageQueueNew(size, sizeof(void*), NULL)) != NULL;
134 }
135
136 uint8_t
137 lwesp_sys_mbox_delete(lwesp_sys_mbox_t* b) {
138     if (osMessageQueueGetCount(*b) > 0) {
139         return 0;
140     }
141     return osMessageQueueDelete(*b) == osOK;
142 }
143
144 uint32_t
145 lwesp_sys_mbox_put(lwesp_sys_mbox_t* b, void* m) {
146     uint32_t tick = osKernelSysTick();
147     return osMessageQueuePut(*b, &m, 0, osWaitForever) == osOK ? (osKernelSysTick() -
148 ↪tick) : LWESP_SYS_TIMEOUT;
149 }
150
151 uint32_t
152 lwesp_sys_mbox_get(lwesp_sys_mbox_t* b, void** m, uint32_t timeout) {
153     uint32_t tick = osKernelSysTick();
154     return (osMessageQueueGet(*b, m, NULL, timeout == 0 ? osWaitForever : timeout) ==
155 ↪osOK) ? (osKernelSysTick() - tick) : LWESP_SYS_TIMEOUT;
156 }
157
158 uint8_t
159 lwesp_sys_mbox_putnow(lwesp_sys_mbox_t* b, void* m) {
160     return osMessageQueuePut(*b, &m, 0, 0) == osOK;
161 }
162
163 uint8_t
164 lwesp_sys_mbox_getnow(lwesp_sys_mbox_t* b, void** m) {
165     return osMessageQueueGet(*b, m, NULL, 0) == osOK;
166 }
167
168 uint8_t
169 lwesp_sys_mbox_isvalid(lwesp_sys_mbox_t* b) {
170     return b != NULL && *b != NULL;
171 }
172
173 uint8_t
174 lwesp_sys_mbox_invalid(lwesp_sys_mbox_t* b) {
175     *b = LWESP_SYS_MBOX_NULL;
176     return 1;
177 }

```

(continues on next page)

(continued from previous page)

```

177 uint8_t
178 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) {
179     lwesp_sys_thread_t id;
180     const osThreadAttr_t thread_attr = {
181         .name = (char*)name,
182         .priority = (osPriority)prio,
183         .stack_size = stack_size > 0 ? stack_size : LWESP_SYS_THREAD_SS
184     };
185
186     id = osThreadNew(thread_func, arg, &thread_attr);
187     if (t != NULL) {
188         *t = id;
189     }
190     return id != NULL;
191 }
192
193 uint8_t
194 lwesp_sys_thread_terminate(lwesp_sys_thread_t* t) {
195     if (t != NULL) {
196         osThreadTerminate(*t);
197     } else {
198         osThreadExit();
199     }
200     return 1;
201 }
202
203 uint8_t
204 lwesp_sys_thread_yield(void) {
205     osThreadYield();
206     return 1;
207 }
208
209 #endif /* !__DOXYGEN__ */

```

5.3 API reference

List of all the modules:

5.3.1 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.

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

Parameters

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

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.

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

Parameters

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

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.

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

Parameters

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

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.

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

Parameters

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

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

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

Parameters

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

lwespr_t **lwesp_ap_configure** (**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

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

Parameters

- [in] *ssid*: SSID name of access point
- [in] *pwd*: Password for network. Either set it to NULL or less than 64 characters
- [in] *ch*: Wifi RF channel
- [in] *ecn*: Encryption type. Valid options are OPEN, WPA_PSK, WPA2_PSK and WPA_WPA2_PSK
- [in] *max_sta*: Maximal number of stations access point can accept. Valid between 1 and 10 stations

- [in] hid: Set to 1 to hide access point from public access
- [in] evt_fn: Callback function called when command has finished. Set to NULL when not used
- [in] evt_arg: Custom argument for event callback function
- [in] blocking: Status whether command should be blocking or not

```
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.

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

Parameters

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

```
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.

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

Parameters

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

```
struct lwesp_ap_t
```

```
    #include <lwesp_typedefs.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 **rss**

Received signal strength indicator

lwesp_mac_t **mac**

MAC physical address

uint8_t **ch**

WiFi channel used on access point

uint8_t **bgn**

Information about 802.11[b/g/n] support

struct lwesp_sta_info_ap_t*#include <lwesp_typedefs.h>* Access point information on which station is connected to.**Public Members**char **ssid**[LWESP_CFG_MAX_SSID_LENGTH]

Access point name

int16_t **rss**

RSSI

lwesp_mac_t **mac**

MAC address

uint8_t **ch**

Channel information

struct lwesp_ap_conf_t*#include <lwesp_typedefs.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

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.

Return 1 on success, 0 otherwise

Parameters

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

`void lwesp_buff_free (lwesp_buff_t *buff)`
Free dynamic allocation if used on memory.

Parameters

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

`void lwesp_buff_reset (lwesp_buff_t *buff)`
Resets buffer to default values. Buffer size is not modified.

Parameters

- [in] `buff`: 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.

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

Parameters

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

`size_t 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.

Return Number of bytes read and copied to data array

Parameters

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

`size_t 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)

Return Number of bytes peeked and written to output array

Parameters

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

`size_t lwesp_buff_get_free (lwesp_buff_t *buff)`

Get number of bytes in buffer available to write.

Return Number of free bytes in memory

Parameters

- [in] `buff`: Buffer handle

`size_t lwesp_buff_get_full (lwesp_buff_t *buff)`

Get number of bytes in buffer available to read.

Return Number of bytes ready to be read

Parameters

- [in] `buff`: Buffer handle

`void *lwesp_buff_get_linear_block_read_address (lwesp_buff_t *buff)`

Get linear address for buffer for fast read.

Return Linear buffer start address

Parameters

- [in] `buff`: Buffer handle

`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.

Return Linear buffer size in units of bytes for read operation

Parameters

- [in] buff: Buffer handle

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

Return Number of bytes skipped

Parameters

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

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

Get linear address for buffer for fast read.

Return Linear buffer start address

Parameters

- [in] buff: Buffer handle

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.

Return Linear buffer size in units of bytes for write operation

Parameters

- [in] buff: Buffer handle

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

Return Number of bytes advanced for write operation

Parameters

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

struct lwesp_buff_t

#include <lwesp_typedefs.h> Buffer structure.

Public Members

`uint8_t *buff`

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

`size_t size`

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

`size_t r`

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

`size_t w`

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

Connections

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

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

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

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

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

Example below shows *bare minimum* implementation to:

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

Listing 14: Client connection minimum example

```

1 #include "client.h"
2 #include "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
10 /**
11  * \brief          Request data for connection
12  */
13 static const
14 uint8_t req_data[] = ""

```

(continues on next page)

```

15         "GET / HTTP/1.1\r\n"
16         "Host: " CONN_HOST "\r\n"
17         "Connection: close\r\n"
18         "\r\n";
19
20 /**
21  * \brief      Start a new connection(s) as client
22  */
23 void
24 client_connect(void) {
25     lwespr_t res;
26
27     /* Start a new connection as client in non-blocking mode */
28     if ((res = lwesp_conn_start(NULL, LWESP_CONN_TYPE_TCP, "example.com", 80, NULL,
↳conn_callback_func, 0)) == lwespOK) {
29         printf("Connection to " CONN_HOST " started...\r\n");
30     } else {
31         printf("Cannot start connection to " CONN_HOST "!\r\n");
32     }
33
34     /* Start 2 more */
35     lwesp_conn_start(NULL, LWESP_CONN_TYPE_TCP, CONN_HOST, CONN_PORT, NULL, conn_
↳callback_func, 0);
36
37     /*
38      * An example of connection which should fail in connecting.
39      * When this is the case, \ref LWESP_EVT_CONN_ERROR event should be triggered
40      * in callback function processing
41      */
42     lwesp_conn_start(NULL, LWESP_CONN_TYPE_TCP, CONN_HOST, 10, NULL, conn_callback_
↳func, 0);
43 }
44
45 /**
46  * \brief      Event callback function for connection-only
47  * \param[in]  evt: Event information with data
48  * \return     \ref lwespOK on success, member of \ref lwespr_t otherwise
49  */
50 static lwespr_t
51 conn_callback_func(lwesp_evt_t* evt) {
52     lwesp_conn_p conn;
53     lwespr_t res;
54     uint8_t conn_num;
55
56     conn = lwesp_conn_get_from_evt(evt);          /* Get connection handle from event_
↳*/
57     if (conn == NULL) {
58         return lwespERR;
59     }
60     conn_num = lwesp_conn_getnum(conn);          /* Get connection number for_
↳identification */
61     switch (lwesp_evt_get_type(evt)) {
62         case LWESP_EVT_CONN_ACTIVE: {           /* Connection just active */
63             printf("Connection %d active!\r\n", (int) conn_num);
64             res = lwesp_conn_send(conn, req_data, sizeof(req_data) - 1, NULL, 0); /*
↳Start sending data in non-blocking mode */
65             if (res == lwespOK) {

```

(continues on next page)

(continued from previous page)

```

66     printf("Sending request data to server...\r\n");
67     } else {
68     printf("Cannot send request data to server. Closing connection_
↳ manually...\r\n");
69     lwesp_conn_close(conn, 0);          /* Close the connection */
70     }
71     break;
72     }
73     case LWESP_EVT_CONN_CLOSE: {          /* Connection closed */
74     if (lwesp_evt_conn_close_is_forced(evt)) {
75     printf("Connection %d closed by client!\r\n", (int)conn_num);
76     } else {
77     printf("Connection %d closed by remote side!\r\n", (int)conn_num);
78     }
79     break;
80     }
81     case LWESP_EVT_CONN_SEND: {          /* Data send event */
82     lwespr_t res = lwesp_evt_conn_send_get_result(evt);
83     if (res == lwespOK) {
84     printf("Data sent successfully on connection %d...waiting to receive_
↳ data from remote side...\r\n", (int)conn_num);
85     } else {
86     printf("Error while sending data on connection %d!\r\n", (int)conn_
↳ num);
87     }
88     break;
89     }
90     case LWESP_EVT_CONN_RECV: {          /* Data received from remote side */
91     lwesp_pbuf_p pbuf = lwesp_evt_conn_recv_get_buff(evt);
92     lwesp_conn_recved(conn, pbuf);      /* Notify stack about received pbuf_
↳ */
93     printf("Received %d bytes on connection %d...\r\n", (int)lwesp_pbuf_
↳ length(pbuf, 1), (int)conn_num);
94     break;
95     }
96     case LWESP_EVT_CONN_ERROR: {          /* Error connecting to server */
97     const char* host = lwesp_evt_conn_error_get_host(evt);
98     lwesp_port_t port = lwesp_evt_conn_error_get_port(evt);
99     printf("Error connecting to %s:%d\r\n", host, (int)port);
100    break;
101    }
102    default:
103    break;
104    }
105    return lwespOK;
106 }

```

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 15: Write data to connection output buffer

```

1  size_t rem_len;
2  lwesp_conn_p conn;
3  lwespr_t res;
4
5  /* ... other tasks to make sure connection is established */
6
7  /* We are connected to server at this point! */
8  /*
9   * Call write function to write data to memory
10  * and do not send immediately unless buffer is full after this write
11  *
12  * rem_len will give us response how much bytes
13  * is available in memory after write
14  */
15  res = lwesp_conn_write(conn, "My string", 9, 0, &rem_len);
16  if (rem_len == 0) {
17      printf("No more memory available for next write!\r\n");
18  }
19  res = lwesp_conn_write(conn, "example.com", 11, 0, &rem_len);
20
21  /*
22  * Data will stay in buffer until buffer is full,
23  * except if user wants to force send,
24  * call write function with flush mode enabled
25  *
26  * It will send out together 20 bytes
27  */
28  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

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.

Return `lwespOK` on success, member of `lwespr_t` enumeration otherwise

Parameters

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

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.

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

Parameters

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

lwespr_t **lwesp_conn_close** (*lwesp_conn_p* conn, const uint32_t blocking)
Close specific or all connections.

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

Parameters

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

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.

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

Parameters

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

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)

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

Parameters

- [in] conn: Connection handle to send data
- [in] ip: Remote IP address for UDP connection

- [in] port: Remote port connection
- [in] data: Pointer to data to send
- [in] btw: Number of bytes to send
- [out] bw: Pointer to output variable to save number of sent data when successfully sent
- [in] blocking: Status whether command should be blocking or not

lwespr_t **lwesp_conn_set_arg** (*lwesp_conn_p* conn, void *const arg)
Set argument variable for connection.

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

See *lwesp_conn_get_arg*

Parameters

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

void ***lwesp_conn_get_arg** (*lwesp_conn_p* conn)
Get user defined connection argument.

Return User argument

See *lwesp_conn_set_arg*

Parameters

- [in] conn: Connection handle to get argument

uint8_t **lwesp_conn_is_client** (*lwesp_conn_p* conn)
Check if connection type is client.

Return 1 on success, 0 otherwise

Parameters

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

uint8_t **lwesp_conn_is_server** (*lwesp_conn_p* conn)
Check if connection type is server.

Return 1 on success, 0 otherwise

Parameters

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

uint8_t **lwesp_conn_is_active** (*lwesp_conn_p* conn)
Check if connection is active.

Return 1 on success, 0 otherwise

Parameters

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

`uint8_t lwesp_conn_is_closed (lwesp_conn_p conn)`

Check if connection is closed.

Return 1 on success, 0 otherwise

Parameters

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

`int8_t lwesp_conn_getnum (lwesp_conn_p conn)`

Get the number from connection.

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

Parameters

- [in] `conn`: Connection pointer

`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

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

Parameters

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

`lwespr_t lwesp_get_conns_status (const uint32_t blocking)`

Gets connections status.

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

Parameters

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

`lwesp_conn_p lwesp_conn_get_from_evt (lwesp_evt_t *evt)`

Get connection from connection based event.

Return Connection pointer on success, NULL otherwise

Parameters

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

`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)

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

Parameters

- [in] `conn`: Connection to write
- [in] `data`: Data to copy to write buffer

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

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

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

Parameters

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

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.

Return Total number of received bytes on connection

Parameters

- [in] conn: Connection handle

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

Get connection remote IP address.

Return 1 on success, 0 otherwise

Parameters

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

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

Get connection remote port number.

Return Port number on success, 0 otherwise

Parameters

- [in] conn: Connection handle

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

Get connection local port number.

Return Port number on success, 0 otherwise

Parameters

- [in] `conn`: Connection handle

lwespr_t **lwesp_conn_ssl_configure** (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.

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

Parameters

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

struct *lwesp_conn_start_t*

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

Public Members

lwespr_t **type**

Connection type

const char ***remote_host**

Host name or IP address in string format

lwespr_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

lwespr_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
    UPD 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 16: Debug configuration setup

```

1  /* Modifications of lwesp_opts.h file for configuration */
2
3  /* Enable global debug */
4  #define LWESP_CFG_DBG                LWESP_DBG_ON
5
6  /*
7   * Enable debug types.
8   * Application may use bitwise OR | to use multiple types:
9   *     LWESP_DBG_TYPE_TRACE | LWESP_DBG_TYPE_STATE
10  */
11 #define LWESP_CFG_DBG_TYPES_ON       LWESP_DBG_TYPE_TRACE
12
13 /* Enable debug on custom module */
14 #define MY_DBG_MODULE                LWESP_DBG_ON

```

Listing 17: Debug usage within middleware

```

1  #include "lwesp/lwesp_debug.h"
2
3  /*
4   * Print debug message to the screen
5   * Trace message will be printed as it is enabled in types
6   * while state message will not be printed.

```

(continues on next page)

```
7  */
8  LWESP_DEBUGF (MY_DBG_MODULE | LWESP_DBG_TYPE_TRACE, "This is trace message on my_
↳program\r\n" );
9  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.

Debug levels

List of debug levels

LWESP_DBG_LVL_ALL

Print all messages of all types

LWESP_DBG_LVL_WARNING

Print warning and upper messages

LWESP_DBG_LVL_DANGER

Print danger errors

LWESP_DBG_LVL_SEVERE

Print severe problems affecting program flow

LWESP_DBG_LVL_MASK

Mask for getting debug level

Debug types

List of debug types

LWESP_DBG_TYPE_TRACE

Debug trace messages for program flow

LWESP_DBG_TYPE_STATE

Debug state messages (such as state machines)

LWESP_DBG_TYPE_ALL

All debug types

Defines

LWESP_DBG_ON

Indicates debug is enabled

LWESP_DBG_OFF

Indicates debug is disabled

LWESP_DEBUGF (*c*, *fmt*, ...)

Print message to the debug “window” if enabled.

Parameters

- [*in*] *c*: Condition if debug of specific type is enabled
- [*in*] *fmt*: Formatted string for debug

- [in] ...: Variable parameters for formatted string

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

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

Parameters

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

Dynamic Host Configuration Protocol

group **LWESP_DHCP**

DHCP config.

Functions

lwespr_t **lwesp_dhcp_configure** (*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.

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

Parameters

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

Domain Name System

group **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.

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

Parameters

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

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.

Retrieve configuration saved in the NVS area of ESP device.

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

Parameters

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

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.

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

Parameters

- [in] *en*: Set to 1 to enable, 0 to disable custom DNS configuration. When disabled, default DNS servers are used as proposed by ESP AT commands firmware
- [in] *s1*: First server IP address in string format, set to NULL if not used
- [in] *s2*: Second server IP address in string format, set to NULL if not used. Address *s1* cannot be the same as *s2*

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

Event management

group 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.

Return 1 if forced by user, 0 otherwise

Parameters

- [in] `cc`: Event handle

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.

Return Member of `lwespr_t` enumeration

Parameters

- [in] `cc`: Event data

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.

Return Member of `lwespr_t` enumeration

Parameters

- [in] `cc`: Event data

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.

Return MAC address

Parameters

- [in] *cc*: Event handle

*lwesp_ip_t****lwesp_evt_ap_ip_sta_get_ip** (*lwesp_evt_t***cc*)

Get IP address from station.

Return IP address

Parameters

- [in] *cc*: Event handle

Connected station to access point

Event helper functions for 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.

Return MAC address

Parameters

- [in] *cc*: Event handle

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.

Return MAC address

Parameters

- [in] *cc*: Event handle

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.

Return Buffer handle

Parameters

- [in] cc: Event handle

lwesp_conn_p **lwesp_evt_conn_recv_get_conn** (*lwesp_evt_t* *cc)
Get connection handle for receive.

Return Connection handle

Parameters

- [in] cc: Event 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.

Return Connection handle

Parameters

- [in] cc: Event handle

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

Return Number of bytes sent

Parameters

- [in] cc: Event handle

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

Return Member of *lwespr_t* enumeration

Parameters

- [in] cc: Event handle

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.

Return Connection handle

Parameters

- [in] cc: Event handle

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

Return 1 if client, 0 otherwise

Parameters

- [in] cc: Event handle

Connection close event

Event helper functions for LWESP_EVT_CONN_CLOSE event

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

Return Connection handle

Parameters

- [in] cc: Event handle

uint8_t **lwesp_evt_conn_close_is_client** (*lwesp_evt_t* *cc)
Check if just closed connection was client.

Return 1 if client, 0 otherwise

Parameters

- [in] cc: Event handle

uint8_t **lwesp_evt_conn_close_is_forced** (*lwesp_evt_t* *cc)
Check if connection close even was forced by user.

Return 1 if forced, 0 otherwise

Parameters

- [in] cc: Event handle

lwespr_t **lwesp_evt_conn_close_get_result** (*lwesp_evt_t* *cc)
Get connection close event result.

Return Member of *lwespr_t* enumeration

Parameters

- [in] cc: Event handle

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.

Return Connection handle

Parameters

- [in] cc: Event 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.

Return Member of *lwespr_t* enumeration

Parameters

- [in] cc: Event handle

lwesp_conn_type_t **lwesp_evt_conn_error_get_type** (*lwesp_evt_t* *cc)
Get connection type.

Return Member of *lwespr_t* enumeration

Parameters

- [in] cc: Event handle

const char ***lwesp_evt_conn_error_get_host** (*lwesp_evt_t* *cc)
Get connection host.

Return Host name for connection

Parameters

- [in] cc: Event handle

lwesp_port_t **lwesp_evt_conn_error_get_port** (*lwesp_evt_t* *cc)
Get connection port.

Return Host port number

Parameters

- [in] cc: Event handle

void ***lwesp_evt_conn_error_get_arg** (*lwesp_evt_t* *cc)
Get user argument.

Return User argument

Parameters

- [in] cc: Event handle

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.

Return Member of *lwespr_t* enumeration

Parameters

- [in] cc: Event handle

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

Get access points.

Return Pointer to *lwesp_ap_t* with first access point description

Parameters

- [in] cc: Event handle

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

Get number of access points found.

Return Number of access points found

Parameters

- [in] cc: Event handle

Join access point

Event helper functions for LWESP_EVT_STA_JOIN_AP event

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

Get command success result.

Return Member of *lwespr_t* enumeration

Parameters

- [in] cc: Event handle

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.

Return Member of *lwespr_t* enumeration

Parameters

- [in] cc: Event handle

const char ***lwesp_evt_sta_info_ap_get_ssid** (*lwesp_evt_t* *cc)
Get current AP name.

Return AP name

Parameters

- [in] cc: Event handle

lwesp_mac_t **lwesp_evt_sta_info_ap_get_mac** (*lwesp_evt_t* *cc)
Get current AP MAC address.

Return AP MAC address

Parameters

- [in] cc: Event handle

uint8_t **lwesp_evt_sta_info_ap_get_channel** (*lwesp_evt_t* *cc)
Get current AP channel.

Return AP channel

Parameters

- [in] cc: Event handle

int16_t **lwesp_evt_sta_info_ap_get_rssi** (*lwesp_evt_t* *cc)
Get current AP rssi.

Return AP rssi

Parameters

- [in] cc: Event handle

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.

Return Member of *lwespr_t* enumeration

Parameters

- [in] cc: Event handle

const char ***lwesp_evt_dns_hostbyname_get_host** (*lwesp_evt_t* *cc)
Get hostname used to resolve IP address.

Return Hostname

Parameters

- [in] cc: Event handle

lwesp_ip_t ***lwesp_evt_dns_hostbyname_get_ip** (*lwesp_evt_t* *cc)
Get IP address from DNS function.

Return IP address

Parameters

- [in] cc: Event handle

Ping

Event helper functions for LWESP_EVT_PING event

lwespr_t **lwesp_evt_ping_get_result** (*lwesp_evt_t* *cc)
Get ping status.

Return Member of *lwespr_t* enumeration

Parameters

- [in] cc: Event handle

const char ***lwesp_evt_ping_get_host** (*lwesp_evt_t* *cc)
Get hostname used to ping.

Return Hostname

Parameters

- [in] cc: Event handle

uint32_t **lwesp_evt_ping_get_time** (*lwesp_evt_t* *cc)
Get time required for ping.

Return Ping time

Parameters

- [in] cc: Event handle

Server

Event helper functions for LWESP_EVT_SERVER event

lwespr_t **lwesp_evt_server_get_result** (*lwesp_evt_t* *cc)
Get server command result.

Return Member of *lwespr_t* enumeration

Parameters

- [in] cc: Event handle

lwesp_port_t **lwesp_evt_server_get_port** (*lwesp_evt_t* *cc)
Get port for server operation.

Return Server port

Parameters

- [in] cc: Event handle

uint8_t **lwesp_evt_server_is_enable** (*lwesp_evt_t* *cc)
Check if operation was to enable or disable server.

Return 1 if enable, 0 otherwise

Parameters

- [in] cc: Event handle

Typedefs

typedef *lwespr_t* (***lwesp_evt_fn**) (**struct** lwesp_evt *evt)
Event function prototype.

Return *lwespOK* on success, member of *lwespr_t* otherwise

Parameters

- [in] evt: Callback event data

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_WIFI_CONNECTED**
Station just connected to AP
- enumerator LWESP_EVT_WIFI_GOT_IP**
Station has valid IP. When this event is received to application, no IP has been read from device. Stack will proceed with IP read from device and will later send *LWESP_EVT_WIFI_IP_ACQUIRED* event
- enumerator LWESP_EVT_WIFI_DISCONNECTED**
Station just disconnected from AP
- 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

Functions

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

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

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

Parameters

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

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*

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

Parameters

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

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

Get event type.

Return Event type. Member of *lwesp_evt_type_t* enumeration

Parameters

- [in] *cc*: Event handle

struct lwesp_evt_t

#include <lwesp_typedefs.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 succeeded

struct lwesp_evt_t::[anonymous]::[anonymous] ping
    Ping finished. Use with LWESP_EVT_PING event

union lwesp_evt_t::[anonymous] evt
    Callback event union

```

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.

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

Parameters

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

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.

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

Parameters

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

Input module

Input module is used to input received data from *ESP* device to *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

Return `lwespOK` on success, member of *lwespr_t* enumeration otherwise

Parameters

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

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

Return `lwespOK` on success, member of *lwespr_t* enumeration otherwise

Parameters

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

Multicast DNS

group **LWESP_MDNS**
mDNS function

Functions

lwespr_t **lwesp_mdns_configure** (*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.

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

Parameters

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

Memory manager

group **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

Return 1 on success, 0 otherwise

Note Function is not available when *LWESP_CFG_MEM_CUSTOM* is 1

Parameters

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

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

Return Memory address on success, NULL otherwise

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

Parameters

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

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

Return Memory address on success, NULL otherwise

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

Parameters

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

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

Return Memory address on success, NULL otherwise

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

Parameters

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

void **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

- [in] *ptr*: 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.

Return 1 on success, 0 otherwise

Parameters

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

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	<i>Block 2</i>	150	550	1
Block 2	<i>Block 3</i>	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 concatenating 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 18: Packet buffer concat example

```

1 lwesp_pbuf_p a, b;
2
3 /* Create 2 pbufs of different sizes */
4 a = lwesp_pbuf_new(10);
5 b = lwesp_pbuf_new(20);
6
7 /* Link them together with concat operation */
8 /* Reference on b will stay as is, won't be increased */
9 lwesp_pbuf_cat(a, b);
10
11 /*
12  * Operating with b variable has from now on undefined behavior,
13  * application shall stop using variable b to access pbuf.
14  *
15  * The best way would be to set b reference to NULL
16  */
17 b = NULL;
18
19 /*
20  * When application doesn't need pbufs anymore,
21  * free a and it will also free b
22  */
23 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 19: Packet buffer chain example

```

1 lwesp_pbuf_p a, b;
2
3 /* Create 2 pbufs of different sizes */
4 a = lwesp_pbuf_new(10);
5 b = lwesp_pbuf_new(20);
6
7 /* Chain both pbufs together */
8 /* This will increase reference on b as 2 variables now point to it */
9 lwesp_pbuf_chain(a, b);
10

```

(continues on next page)

(continued from previous page)

```

11 /*
12  * When application does not need a anymore, it may free it
13
14  * This will free only pbuf a, as pbuf b has now 2 references:
15  * - one from pbuf a
16  * - one from variable b
17  */
18
19 /* If application calls this, it will free only first pbuf */
20 /* As there is link to b pbuf somewhere */
21 lwesp_pbuf_free(a);
22
23 /* Reset a variable, not used anymore */
24 a = NULL;
25
26 /*
27  * At this point, b is still valid memory block,
28  * but when application doesn't need it anymore,
29  * it should free it, otherwise memory leak appears
30  */
31 lwesp_pbuf_free(b);
32
33 /* Reset b variable */
34 b = NULL;

```

Extract pbuf data

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

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

Listing 20: Packet buffer data extraction

```

1  const void* data;
2  size_t pos, len;
3  lwesp_pbuf_p a, b, c;
4
5  const char str_a[] = "This is one long";
6  const char str_b[] = "string. We want to save";
7  const char str_c[] = "chain of pbufs to file";
8
9  /* Create pbufs to hold these strings */
10 a = lwesp_pbuf_new(strlen(str_a));
11 b = lwesp_pbuf_new(strlen(str_b));
12 c = lwesp_pbuf_new(strlen(str_c));
13
14 /* Write data to pbufs */
15 lwesp_pbuf_take(a, str_a, strlen(str_a), 0);
16 lwesp_pbuf_take(b, str_b, strlen(str_b), 0);
17 lwesp_pbuf_take(c, str_c, strlen(str_c), 0);
18
19 /* Connect pbufs together */
20 lwesp_pbuf_chain(a, b);
21 lwesp_pbuf_chain(a, c);

```

(continues on next page)

```

22
23 /*
24  * pbuf a now contains chain of b and c together
25  * and at this point application wants to print (or save) data from chained pbuf
26  *
27  * Process pbuf by pbuf with code below
28  */
29
30 /*
31  * Get linear address of current pbuf at specific offset
32  * Function will return pointer to memory address at specific position
33  * and `len` will hold length of data block
34  */
35 pos = 0;
36 while ((data = lwesp_pbuf_get_linear_addr(a, pos, &len)) != NULL) {
37     /* Custom process function... */
38     /* Process data with data pointer and block length */
39     process_data(data, len);
40     printf("Str: %.*s", len, data);
41
42     /* Increase offset position for next block */
43     pos += len;
44 }
45
46 /* Call free only on a pbuf. Since it is chained, b and c will be freed too */
47 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.

Return Pointer to allocated memory, NULL otherwise

Parameters

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

size_t **lwesp_pbuf_free** (*lwesp_pbuf_p* pbuf)
Free previously allocated packet buffer.

Return Number of freed pbufs from head

Parameters

- [in] pbuf: Packet buffer to free

void *lwesp_pbuf_data (const lwesp_pbuf_p pbuf)

Get data pointer from packet buffer.

Return Pointer to data buffer on success, NULL otherwise

Parameters

- [in] pbuf: Packet buffer

size_t lwesp_pbuf_length (const lwesp_pbuf_p pbuf, uint8_t tot)

Get length of packet buffer.

Return Length of data in units of bytes

Parameters

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

uint8_t 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

Return 1 on success, 0 otherwise

Parameters

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

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.

Return lwespOK on success, member of lwespr_t enumeration otherwise

Parameters

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

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

Copy memory from pbuf to user linear memory.

Return Number of bytes copied

Parameters

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

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

Concatenate 2 packet buffers together to one big packet.

Note After tail pbuf has been added to head pbuf chain, it must not be referenced by user anymore as it is now 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.

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

See *lwesp_pbuf_chain*

Parameters

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

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.

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)

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

See *lwesp_pbuf_cat*

Parameters

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

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)

Return Next pbuf after head

Parameters

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

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

Increment reference count on pbuf.

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

Parameters

- [in] pbuf: pbuf to increase reference

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

Get value from pbuf at specific position.

Return 1 on success, 0 otherwise

Parameters

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

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

Note Compare is done on entire pbuf chain

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

See *lwesp_pbuf_strcmp*

Parameters

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

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

Note Compare is done on entire pbuf chain

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

See *lwesp_pbuf_memcmp*

Parameters

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

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

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

See *lwesp_pbuf_strfind*

Parameters

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

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

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

See *lwesp_pbuf_memfind*

Parameters

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

uint8_t **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

Return 1 on success, 0 otherwise

Parameters

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

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

Return New pbuf on success, NULL otherwise

Parameters

- [in] pbuf: Start of pbuf chain
- [in] offset: Offset in units of bytes to skip
- [out] new_offset: Pointer to output variable to save new offset in returned pbuf

void ***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

Return Pointer to memory on success, NULL otherwise

Parameters

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

void **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

- [in] pbuf: Packet buffer
- [in] ip: IP to assign to packet buffer

- [in] port: Port number to assign to packet buffer

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

Parameters

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

struct 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.

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

Parameters

- [in] host: Host name to ping
- [out] time: Pointer to output variable to save ping time in units of milliseconds
- [in] evt_fn: Callback function called when command has finished. Set to NULL when not used

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

Smart config

group **LWESP_SMART**

SMART function on ESP device.

Functions

lwespr_t **lwesp_smart_configure** (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.

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

Parameters

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

Simple Network Time Protocol

ESP has built-in support for *Simple Network Time Protocol (SNTP)*. It is support through middleware API calls for configuring servers and reading actual date and time.

Listing 21: Minimum SNTP example

```

1  #include "sntp.h"
2  #include "lwesp/lwesp.h"
3
4  /**
5   * \brief          Run SNTP
6   */
7  void
8  sntp_gettime(void) {
9      lwesp_datetime_t dt;
10
11     /* Enable SNTP with default configuration for NTP servers */
12     if (lwesp_sntp_configure(1, 1, NULL, NULL, NULL, NULL, NULL, 1) == lwespOK) {
13         lwesp_delay(5000);
14
15         /* Get actual time and print it */
16         if (lwesp_sntp_gettime(&dt, NULL, NULL, 1) == lwespOK) {
17             printf("Date & time: %d.%d.%d, %d:%d:%d\r\n",
18                 (int)dt.date, (int)dt.month, (int)dt.year,
19                 (int)dt.hours, (int)dt.minutes, (int)dt.seconds);
20         }
21     }
22 }

```

group LWESP_Sntp

Simple network time protocol supported by AT commands.

Functions

lwespr_t **lwesp_sntp_configure** (*uint8_t* *en*, *int8_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.

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

Parameters

- [in] *en*: Status whether SNTP mode is enabled or disabled on ESP device
- [in] *tz*: Timezone to use when SNTP acquires time, between -11 and 13
- [in] *h1*: Optional first SNTP server for time. Set to NULL if not used
- [in] *h2*: Optional second SNTP server for time. Set to NULL if not used
- [in] *h3*: Optional third SNTP server for time. Set to NULL if not used
- [in] *evt_fn*: Callback function called when command has finished. Set to NULL when not used
- [in] *evt_arg*: Custom argument for event callback function
- [in] *blocking*: Status whether command should be blocking or not

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

Get time from SNTP servers.

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

Parameters

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

Station API

Station API is used to work with *ESP* acting in station mode. It allows to join other access point, scan for available access points or simply disconnect from it.

An example below is showing how all examples (coming with this library) scan for access point and then try to connect to AP from list of preferred one.

Listing 22: Station manager used with all examples

```

1 #include "station_manager.h"
2 #include "lwesp/lwesp.h"
3
4 /*
5  * List of preferred access points for ESP device
6  * SSID and password
7  *
8  * ESP will try to scan for access points
9  * and then compare them with the one on the list below
10 */
11 ap_entry_t
12 ap_list[] = {
13     /*{ "SSID name", "SSID password" },
14     { "TilenM_ST", "its private" },
15     { "Majerle WIFI", "majerle_internet_private" },
16     { "Majerle AMIS", "majerle_internet_private" },
17 };
18
19 /**
20  * \brief      List of access points found by ESP device
21  */
22 static
23 lwesp_ap_t aps[100];
24
25 /**
26  * \brief      Number of valid access points in \ref aps array
27  */
28 static
29 size_t apf;
30
31 /**
32  * \brief      Connect to preferred access point
33  *
34  * \note      List of access points should be set by user in \ref ap_list_
35  * \structure
36  * \param[in] unlimited: When set to 1, function will block until SSID is found_
37  * \and connected
38  * \return     \ref lwespOK on success, member of \ref lwespr_t enumeration_
39  * \otherwise
40  */
41 lwespr_t
42 connect_to_preferred_access_point(uint8_t unlimited) {
43     lwespr_t eres;
44     uint8_t tried;
45
46     /*
47      * Scan for network access points
48      * In case we have access point,
49      * try to connect to known AP
50      */
51     do {
52         if (lwesp_sta_has_ip()) {
53             return lwespOK;
54         }
55
56         /* Scan for access points visible to ESP device */

```

(continues on next page)

(continued from previous page)

```

54     printf("Scanning access points...\r\n");
55     if ((eres = lwesp_sta_list_ap(NULL, aps, LWESP_ARRAYSIZE(aps), &apf, NULL,
↪NULL, 1)) == lwespOK) {
56         tried = 0;
57         /* Print all access points found by ESP */
58         for (size_t i = 0; i < apf; i++) {
59             printf("AP found: %s, CH: %d, RSSI: %d\r\n", aps[i].ssid, aps[i].ch,
↪aps[i].rssi);
60         }
61
62         /* Process array of preferred access points with array of found points */
63         for (size_t j = 0; j < LWESP_ARRAYSIZE(ap_list); j++) {
64             for (size_t i = 0; i < apf; i++) {
65                 if (!strcmp(aps[i].ssid, ap_list[j].ssid)) {
66                     tried = 1;
67                     printf("Connecting to \"%s\" network...\r\n", ap_list[j].
↪ssid);
68
69                     /* Try to join to access point */
70                     if ((eres = lwesp_sta_join(ap_list[j].ssid, ap_list[j].pass,
↪NULL, NULL, NULL, 1)) == lwespOK) {
71                         lwesp_ip_t ip;
72                         uint8_t is_dhcp;
73                         lwesp_sta_copy_ip(&ip, NULL, NULL, &is_dhcp);
74
75                         printf("Connected to %s network!\r\n", ap_list[j].ssid);
76                         printf("Station IP address: %d.%d.%d.%d; Is DHCP: %d\r\n",
↪(int)ip.ip[0], (int)ip.ip[1], (int)ip.ip[2],
↪(int)ip.ip[3], (int)is_dhcp);
77                         return lwespOK;
78                     } else {
79                         printf("Connection error: %d\r\n", (int)eres);
80                     }
81                 }
82             }
83         }
84         if (!tried) {
85             printf("No access points available with preferred SSID!\r\nPlease
↪check station_manager.c file and edit preferred SSID access points!\r\n");
86         }
87         } else if (eres == lwespERRNODEVICE) {
88             printf("Device is not present!\r\n");
89             break;
90         } else {
91             printf("Error on WIFI scan procedure!\r\n");
92         }
93         if (!unlimited) {
94             break;
95         }
96     } while (1);
97     return lwespERR;
98 }

```

group **LWESP_STA**
Station API.

Functions

lwespr_t **lwespr_sta_join** (**const** char *name, **const** char *pass, **const** *lwespr_mac_t* *mac, **const** *lwespr_api_cmd_evt_fn* evt_fn, void ***const** evt_arg, **const** uint32_t blocking)

Join as station to access point.

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

Return *lwesprOK* on success, member of *lwespr_t* enumeration otherwise

Parameters

- [in] name: SSID of access point to connect to
- [in] pass: Password of access point. Use NULL if AP does not have password
- [in] mac: Pointer to MAC address of AP. If multiple APs with same name exist, MAC may help to select proper one. Set to NULL if not needed
- [in] evt_fn: Callback function called when command has finished. Set to NULL when not used
- [in] evt_arg: Custom argument for event callback function
- [in] blocking: Status whether command should be blocking or not

lwespr_t **lwespr_sta_quit** (**const** *lwespr_api_cmd_evt_fn* evt_fn, void ***const** evt_arg, **const** uint32_t blocking)

Quit (disconnect) from access point.

Return *lwesprOK* on success, member of *lwespr_t* enumeration otherwise

Parameters

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

lwespr_t **lwespr_sta_autojoin** (uint8_t en, **const** *lwespr_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 *lwespr_sta_join* for more information

Return *lwesprOK* on success, member of *lwespr_t* enumeration otherwise

Parameters

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

```
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.

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

Parameters

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

```
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

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

Parameters

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

```
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.

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

Parameters

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

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

`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.

Return `lwespOK` on success, member of `lwespr_t` enumeration otherwise

Parameters

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

`uint8_t lwesp_sta_has_ip` (void)

Check if ESP got IP from access point.

Return 1 on success, 0 otherwise

`uint8_t lwesp_sta_is_joined` (void)

Check if station is connected to WiFi network.

Return 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

Return `lwespOK` on success, member of `lwespr_t` enumeration otherwise

Parameters

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

`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.

Return `lwespOK` on success, member of `lwespr_t` enumeration otherwise

Parameters

- [in] `ssid`: Optional SSID name to search for. Set to `NULL` to disable filter
- [in] `aps`: Pointer to array of available access point parameters

- [in] `apsl`: Length of `aps` array
- [out] `apf`: Pointer to output variable to save number of access points found
- [in] `evt_fn`: Callback function called when command has finished. Set to `NULL` when not used
- [in] `evt_arg`: Custom argument for event callback function
- [in] `blocking`: Status whether command should be blocking or not

```
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

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

Parameters

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

```
uint8_t lwesp_sta_is_ap_802_11b (lwesp_ap_t *ap)
Check if access point is 802.11b compatible.
```

Return 1 on success, 0 otherwise

Parameters

- [in] `ap`: Access point details acquired by *lwesp_sta_list_ap*

```
uint8_t lwesp_sta_is_ap_802_11g (lwesp_ap_t *ap)
Check if access point is 802.11g compatible.
```

Return 1 on success, 0 otherwise

Parameters

- [in] `ap`: Access point details acquired by *lwesp_sta_list_ap*

```
uint8_t lwesp_sta_is_ap_802_11n (lwesp_ap_t *ap)
Check if access point is 802.11n compatible.
```

Return 1 on success, 0 otherwise

Parameters

- [in] `ap`: Access point details acquired by *lwesp_sta_list_ap*

```
struct lwesp_sta_t
#include <lwesp_typedefs.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.

Parameters

- [in] *arg*: Custom user argument

Functions

lwespr_t **lwesp_timeout_add** (uint32_t *time*, *lwesp_timeout_fn* *fn*, void **arg*)
Add new timeout to processing list.

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

Parameters

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

lwespr_t **lwesp_timeout_remove** (*lwesp_timeout_fn* *fn*)
Remove callback from timeout list.

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

Parameters

- [in] *fn*: Callback function to identify timeout to remove

```
struct lwesp_timeout_t
    #include <lwesp_typedefs.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_TYPEDEFS**
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.
```

Parameters

- [in] *res*: Operation result, member of *lwespr_t* enumeration
- [in] *arg*: Custom user argument

Enums

```
enum lwesp_cmd_t
    List of possible messages.

Values:

enumerator LWESP_CMD_IDLE
    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_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_SYSMMSG

enumerator LWESP_CMD_SYSLOG

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_CWJAP
Connect to access point

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

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_CIUUPDATE
Perform self-update

enumerator LWESP_CMD_TCPIP_CIPSNTPCFG
Configure SNTP servers

enumerator LWESP_CMD_TCPIP_CIPSNTPTIME
Get current time using SNTP

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_BLEINIT_GET
Get BLE status

enum lwespr_t
Result enumeration used across application functions.

Values:

enumerator lwespOK
Function succeeded

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

enumerator lwespERR

enumerator lwespPARERR
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

enum lwesp_device_t
List of support ESP devices by firmware.

Values:

enumerator LWESP_DEVICE_ESP8266
Device is ESP8266

enumerator LWESP_DEVICE_ESP32
Device is ESP32

enumerator LWESP_DEVICE_UNKNOWN

Unknown device

enum lwesp_ecn_t

List of encryptions of access point.

Values:

enumerator LWESP_ECN_OPEN

No encryption on access point

enumerator LWESP_ECN_WEP

WEP (Wired Equivalent Privacy) encryption

enumerator LWESP_ECN_WPA_PSK

WPA (Wifi Protected Access) encryption

enumerator LWESP_ECN_WPA2_PSK

WPA2 (Wifi Protected Access 2) encryption

enumerator LWESP_ECN_WPA_WPA2_PSK

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

enumerator LWESP_ECN_WPA2_Enterprise

Enterprise encryption.

Note ESP is currently not able to connect to access point of this encryption type

enum lwesp_mode_t

List of possible WiFi modes.

Values:

enumerator LWESP_MODE_STA

Set WiFi mode to station only

enumerator LWESP_MODE_AP

Set WiFi mode to access point only

enumerator LWESP_MODE_STA_AP

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

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

uint8_t val_id
Validation ID number. It is increased each time a new connection is established. It protects sending data to wrong connection in case we have data in send queue, and connection was closed and active again in between.

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 (*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

uint8_t **en**
Status to enable/disable auto join feature
Enable/disable DHCP settings
Enable/Disable server status
Status if SNTP is enabled or not
Status if WPS is enabled or not
Set to 1 to enable or 0 to disable

struct *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 **aps1**
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**
Ecrption 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

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**
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_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 num
Connection number used for start

uint8_t success
Status if connection AT+CIPSTART succeeded

```

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

uint8_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

int8_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

lwesp_datetime_t *dt
Pointer to datetime structure

struct lwesp_msg_t::[anonymous]::[anonymous] tcpip_sntp_time
SNTP get time

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 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
SSI 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 ESP has IP

uint8_t is_connected
 Flag indicating ESP is connected to wifi

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,

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 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_unicode_t
#include <lwesp_private.h> Unicode support structure.

Public Members

uint8_t ch[4]
 UTF-8 max characters

uint8_t t
 Total expected length in UTF-8 sequence

uint8_t r
 Remaining bytes in UTF-8 sequence

lwespr_t **res**
 Current result of processing

struct lwesp_ip_t
#include <lwesp_typedefs.h> IP structure.

Public Members

`uint8_t ip[4]`
IPv4 address

struct lwesp_mac_t
#include <lwesp_typedefs.h> MAC address.

Public Members

`uint8_t mac[6]`
MAC address

struct lwesp_sw_version_t
#include <lwesp_typedefs.h> SW version in semantic versioning format.

Public Members

`uint8_t major`
Major version

`uint8_t minor`
Minor version

`uint8_t patch`
Patch version

struct lwesp_datetime_t
#include <lwesp_typedefs.h> Date and time structure.

Public Members

`uint8_t date`
Day in a month, from 1 to up to 31

`uint8_t month`
Month in a year, from 1 to 12

`uint16_t year`
Year

`uint8_t day`
Day in a week, from 1 to 7

`uint8_t hours`
Hours in a day, from 0 to 23

`uint8_t minutes`
Minutes in a hour, from 0 to 59

`uint8_t seconds`
Seconds in a minute, from 0 to 59

struct lwesp_linbuff_t
#include <lwesp_typedefs.h> Linear buffer structure.

Public Members

`uint8_t *buff`
 Pointer to buffer data array

`size_t len`
 Length of buffer array

`size_t ptr`
 Current buffer pointer

Unicode

Unicode decoder block. It can decode sequence of *UTF-8* characters, between 1 and 4 bytes long.

Note: This is simple implementation and does not support string encoding.

group **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.

Return *lwespOK* Function succeeded, there is a valid UTF-8 sequence

Return *lwespINPROG* Function continues well but expects some more data to finish sequence

Return *lwespERR* Error in UTF-8 sequence

Parameters

- [*inout*] *s*: Pointer to unicode decode control structure
- [*in*] *c*: UTF-8 character sequence to test for device

struct **lwesp_unicode_t**
#include <*lwesp_private.h*> Unicode support structure.

Public Members

`uint8_t ch[4]`
 UTF-8 max characters

`uint8_t t`
 Total expected length in UTF-8 sequence

`uint8_t r`
 Remaining bytes in UTF-8 sequence

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 (*msg, c*)

Assert an input parameter if in valid range.

Note Since this is a macro, it may only be used on a functions where return status is of type *lwespr_t* enumeration

Parameters

- [in] *msg*: message to print to debug if test fails
- [in] *c*: Condition to test

LWESP_MEM_ALIGN (*x*)

Align *x* value to specific number of bytes, provided by *LWESP_CFG_MEM_ALIGNMENT* configuration.

Return Input value aligned to specific number of bytes

Parameters

- [in] *x*: Input value to align

LWESP_MIN (*x, y*)

Get minimal value between *x* and *y* inputs.

Return Minimal value between *x* and *y* parameters

Parameters

- [in] *x*: First input to test
- [in] *y*: Second input to test

LWESP_MAX (*x, y*)

Get maximal value between *x* and *y* inputs.

Return Maximal value between *x* and *y* parameters

Parameters

- [in] *x*: First input to test
- [in] *y*: Second input to test

LWESP_ARRAYSIZE (*x*)

Get size of statically declared array.

Return Number of array elements

Parameters

- [in] x: Input array

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

- [in] x: Variable which is not used

LWESP_U32 (*x*)

Get input value casted to unsigned 32-bit value.

Parameters

- [in] x: Input value

LWESP_U16 (*x*)

Get input value casted to unsigned 16-bit value.

Parameters

- [in] x: Input value

LWESP_U8 (*x*)

Get input value casted to unsigned 8-bit value.

Parameters

- [in] x: Input value

LWESP_I32 (*x*)

Get input value casted to signed 32-bit value.

Parameters

- [in] x: Input value

LWESP_I16 (*x*)

Get input value casted to signed 16-bit value.

Parameters

- [in] x: Input value

LWESP_I8 (*x*)

Get input value casted to signed 8-bit value.

Parameters

- [in] x: Input value

LWESP_SZ (*x*)

Get input value casted to `size_t` value.

Parameters

- [in] x: Input value

lwesp_u32_to_str (*num, out*)

Convert unsigned 32-bit number to string.

Return Pointer to output variable

Parameters

- [in] num: Number to convert
- [out] out: Output variable to save string

lwesp_u32_to_hex_str (*num, out, w*)

Convert unsigned 32-bit number to HEX string.

Return Pointer to output variable

Parameters

- [in] num: Number to convert
- [out] out: Output variable to save string
- [in] w: Width of output string. When number is shorter than width, leading 0 characters will apply

lwesp_i32_to_str (*num, out*)

Convert signed 32-bit number to string.

Return Pointer to output variable

Parameters

- [in] num: Number to convert
- [out] out: Output variable to save string

lwesp_u16_to_str (*num, out*)

Convert unsigned 16-bit number to string.

Return Pointer to output variable

Parameters

- [in] num: Number to convert
- [out] out: Output variable to save string

lwesp_u16_to_hex_str (*num, out, w*)

Convert unsigned 16-bit number to HEX string.

Return Pointer to output variable

Parameters

- [in] num: Number to convert
- [out] out: Output variable to save string
- [in] w: Width of output string. When number is shorter than width, leading 0 characters will apply.

lwesp_i16_to_str (*num, out*)

Convert signed 16-bit number to string.

Return Pointer to output variable

Parameters

- [in] num: Number to convert
- [out] out: Output variable to save string

lwesp_u8_to_str (*num, out*)

Convert unsigned 8-bit number to string.

Return Pointer to output variable

Parameters

- [in] num: Number to convert
- [out] out: Output variable to save string

lwesp_u8_to_hex_str (*num, out, w*)

Convert unsigned 16-bit number to HEX string.

Return Pointer to output variable

Parameters

- [in] num: Number to convert
- [out] out: Output variable to save string
- [in] w: Width of output string. When number is shorter than width, leading 0 characters will apply.

lwesp_i8_to_str (*num, out*)

Convert signed 8-bit number to string.

Return Pointer to output variable

Parameters

- [in] num: Number to convert
- [out] out: Output variable to save string

Functions

char ***lwesp_u32_to_gen_str** (uint32_t *num*, char **out*, uint8_t *is_hex*, uint8_t *padding*)

Convert unsigned 32-bit number to string.

Return Pointer to output variable

Parameters

- [in] num: Number to convert
- [out] out: Output variable to save string
- [in] is_hex: Set to 1 to output hex, 0 otherwise

- [in] width: Width of output string. When number is shorter than width, leading 0 characters will apply. This parameter is valid only when formatting hex numbers

char ***lwesp_i32_to_gen_str** (int32_t *num*, char **out*)
Convert signed 32-bit number to string.

Return Pointer to output variable

Parameters

- [in] num: Number to convert
- [out] out: Output variable to save string

Wi-Fi Protected Setup

group **LWESP_WPS**
WPS function on ESP device.

Functions

lwespr_t **lwesp_wps_configure** (uint8_t *en*, const *lwesp_api_cmd_evt_fn* *evt_fn*, void *const *evt_arg*, const uint32_t *blocking*)
Configure WPS function on ESP device.

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

Parameters

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

group **LWESP**
Lightweight ESP-AT parser.

Defines

lwesp_set_fw_version (*v*, *major_*, *minor_*, *patch_*)
Set and format major, minor and patch values to firmware version.

Parameters

- [in] v: Version output, pointer to *lwesp_sw_version_t* structure
- [in] major_: Major version
- [in] minor_: Minor version
- [in] patch_: Patch version

lwesp_get_min_at_fw_version (*v*)
Get minimal AT version supported by library.

Parameters

- [out] v: Version output, pointer to *lwesp_sw_version_t* structure

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.

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

Parameters

- [in] *evt_func*: Global event callback function for all major events
- [in] *blocking*: Status whether command should be blocking or not. Used when *LWESP_CFG_RESET_ON_INIT* or *LWESP_CFG_RESTORE_ON_INIT* are enabled.

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.

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

Parameters

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

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.

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

Parameters

- [in] *delay*: Number of milliseconds to wait before initiating first command to device
- [in] *evt_fn*: Callback function called when command has finished. Set to NULL when not used
- [in] *evt_arg*: Custom argument for event callback function
- [in] *blocking*: Status whether command should be blocking or not

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.

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

Parameters

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

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)

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

Parameters

- [in] *baud*: Baudrate in units of bits per second
- [in] *evt_fn*: Callback function called when command has finished. Set to NULL when not used
- [in] *evt_arg*: Custom argument for event callback function
- [in] *blocking*: Status whether command should be blocking or not

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.

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

Parameters

- [in] *mode*: Mode of operation. This parameter can be a value of *lwesp_mode_t* enumeration
- [in] *evt_fn*: Callback function called when command has finished. Set to NULL when not used
- [in] *evt_arg*: Custom argument for event callback function
- [in] *blocking*: Status whether command should be blocking or not

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.

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

Parameters

- [in] *mode*: point to space of Mode to get. This parameter can be a pointer of *lwesp_mode_t* enumeration
- [in] *evt_fn*: Callback function called when command has finished. Set to NULL when not used
- [in] *evt_arg*: Custom argument for event callback function
- [in] *blocking*: Status whether command should be blocking or not

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.

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

Parameters

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

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

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

Parameters

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

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

Return *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

Return *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

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

Parameters

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

uint8_t **lwesp_device_is_present** (void)

Check if device is present.

Return 1 on success, 0 otherwise

uint8_t **lwesp_device_is_esp8266** (void)

Check if modem device is ESP8266.

Return 1 on success, 0 otherwise

uint8_t **lwesp_device_is_esp32** (void)

Check if modem device is ESP32.

Return 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

Return 1 on success, 0 otherwise

Parameters

- [in] *ms*: Milliseconds to delay

uint8_t **lwesp_get_current_at_fw_version** (*lwesp_sw_version_t* *const *version*)

Get current AT firmware version of connected device.

Return 1 on success, 0 otherwise

Parameters

- [out] *version*: Output version variable

5.3.2 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_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 `esp/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 don'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_CONNS

Maximal number of connections AT software can support on ESP device.

Note In case of official AT software, leave this on default value (5)

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_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_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

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_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 trailing zero

LWESP_CFG_MAX_PWD_LENGTH

Maximum length of PWD for access point.

Note This parameter must include trailing zero

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_MANUAL_TCP_RECEIVE

Enables 1 or disables 0 manual TCP data receive from ESP device.

Normally ESP automatically sends received TCP data to host device in async mode. When host device is slow or if there is memory constrain, it may happen that processing cannot handle all received data.

When feature is enabled, ESP will notify host device about new data available for read and then user may start read process

Note This feature is only available for TCP connections.

LWESP_MIN_AT_VERSION_MAJOR_ESP8266

Minimal major version for ESP8266

LWESP_MIN_AT_VERSION_MINOR_ESP8266

Minimal minor version for ESP8266

LWESP_MIN_AT_VERSION_PATCH_ESP8266

Minimal patch version for ESP8266

LWESP_MIN_AT_VERSION_MAJOR_ESP32

Minimal major version for ESP32

LWESP_MIN_AT_VERSION_MINOR_ESP32

Minimal minor version for ESP32

LWESP_MIN_AT_VERSION_PATCH_ESP32

Minimal patch version for ESP32

group **LWESP_OPT_DBG**
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

group **LWESP_OPT_OS**

Operating system dependant configuration.

Defines**LWESP_CFG_THREAD_PRODUCER_MBOX_SIZE**

Set number of message queue entries for producer 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.

group LWESP_OPT_STD_LIB

Standard C library configuration.

Configuration allows you to overwrite default C language function in case of better implementation with hardware (for example DMA for data copy).

Defines**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);
```

Return Destination memory start address

Parameters

- [in] *dst*: Destination memory start address
- [in] *src*: Source memory start address
- [in] *len*: Number of bytes to copy

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);
```

Return Destination memory start address

Parameters

- [in] *dst*: Destination memory start address
- [in] *b*: Value (byte) to set in memory
- [in] *len*: Number of bytes to set

group LWESP_OPT_MODULES

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_HOSTNAME

Enables 1 or disables 0 support for hostname 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.

group **LWESP_OPT_MODULES_NETCONN**

Configuration of netconn API module.

Defines**LWESP_CFG_NETCONN**

Enables 1 or disables 0 NETCONN sequential API support for OS systems.

Note To use this feature, OS support is mandatory.

See *LWESP_CFG_OS*

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

group **LWESP_OPT_MODULES_MQTT**

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_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*

group **LWESP_OPT_MODULES_CAYENNE**
Configuration of Cayenne MQTT client.

Defines

LWESP_CFG_DBG_CAYENNE
Set debug level for Cayenne MQTT client module.
Possible values are *LWESP_DBG_ON* or *LWESP_DBG_OFF*

group **LWESP_OPT_APP_HTTP**
Configuration of HTTP server app.

Defines

LWESP_CFG_DBG_SERVER
Server debug default setting.

HTTP_SSI_TAG_START
SSI tag start string

HTTP_SSI_TAG_START_LEN
SSI tag start length

HTTP_SSI_TAG_END
SSI tag end string

HTTP_SSI_TAG_END_LEN
SSI tag end length

HTTP_SSI_TAG_MAX_LEN
Maximal length of tag name excluding start and end parts of tag.

HTTP_SUPPORT_POST
Enables 1 or disables 0 support for POST request.

HTTP_MAX_URI_LEN
Maximal length of allowed uri length including parameters in format */uri/sub/path?param=value*

HTTP_MAX_PARAMS
Maximal number of parameters in URI.

HTTP_USE_METHOD_NOTALLOWED_RESP
Enables 1 or disables 0 method not allowed response.
Response is used in case user makes HTTP request with method which is not on the list of allowed methods.
See *http_req_method_t*

Note When disabled, connection will be closed without response

HTTP_USE_DEFAULT_STATIC_FILES
Enables 1 or disables 1 default static files.

To allow fast startup of server development, several static files are included by default:

- */index.html*
- */index.shtml*
- */js/style.css*

- /js/js.js

HTTP_DYNAMIC_HEADERS

Enables 1 or disables 0 dynamic headers support.

With dynamic headers enabled, script will try to detect most common file extensions and will try to response with:

- HTTP response code as first line
- Server name as second line
- Content type as third line including end of headers (empty line)

HTTP_DYNAMIC_HEADERS_CONTENT_LEN

Enables 1 or disables 0 content length header for response.

If response has fixed length without SSI tags, dynamic headers will try to include “Content-Length” header as part of response message sent to client

Note In order to use this, *HTTP_DYNAMIC_HEADERS* must be enabled

HTTP_SERVER_NAME

Default server name for `Server: x` response dynamic header.

5.3.3 Platform specific

List of all the modules:

Low-Level functions

Low-level module consists of callback-only functions, which are called by middleware and must be implemented by final application.

Tip: Check *Porting guide* for actual implementation

group **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.

Return Number of bytes sent

Parameters

- [*in*] `data`: Pointer to data to send. This parameter can be set to `NULL`
- [*in*] `len`: Number of bytes to send. This parameter can be set to 0 to indicate that internal buffer can be flushed to stream. This is implementation defined and feature might be ignored

typedef `uint8_t (*lwesp_ll_reset_fn) (uint8_t state)`

Function prototype for hardware reset of ESP device.

Return 1 on successful action, 0 otherwise

Parameters

- [in] *state*: State indicating reset. When set to 1, reset must be active (usually pin active low), or set to 0 when reset is cleared

Functions

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.

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

Parameters

- [inout] *ll*: Pointer to *lwesp_ll_t* structure to fill data for communication functions

lwespr_t **lwesp_ll_deinit** (*lwesp_ll_t *ll*)

Callback function to de-init low-level communication part.

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

Parameters

- [inout] *ll*: Pointer to *lwesp_ll_t* structure to fill data for communication functions

struct *lwesp_ll_t*

#include <lwesp_typedefs.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.

Return 1 on success, 0 otherwise

uint32_t **lwesp_sys_now** (void)

Get current time in units of milliseconds.

Return 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.

Return 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.

Return 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

Return 1 on success, 0 otherwise

Parameters

- [out] p: Pointer to mutex structure to allocate

`uint8_t lwesp_sys_mutex_delete (lwesp_sys_mutex_t *p)`
Delete recursive mutex from system.

Return 1 on success, 0 otherwise

Parameters

- [in] p: Pointer to mutex structure

`uint8_t lwesp_sys_mutex_lock (lwesp_sys_mutex_t *p)`
Lock recursive mutex, wait forever to lock.

Return 1 on success, 0 otherwise

Parameters

- [in] p: Pointer to mutex structure

`uint8_t lwesp_sys_mutex_unlock (lwesp_sys_mutex_t *p)`
Unlock recursive mutex.

Return 1 on success, 0 otherwise

Parameters

- [in] p: Pointer to mutex structure

`uint8_t lwesp_sys_mutex_isvalid (lwesp_sys_mutex_t *p)`
Check if mutex structure is valid system.

Return 1 on success, 0 otherwise

Parameters

- [in] p: Pointer to mutex structure

`uint8_t lwesp_sys_mutex_invalid (lwesp_sys_mutex_t *p)`
Set recursive mutex structure as invalid.

Return 1 on success, 0 otherwise

Parameters

- [in] p: Pointer to mutex structure

Semaphores

`uint8_t 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

Return 1 on success, 0 otherwise

Parameters

- [out] p: Pointer to semaphore structure to fill with result
- [in] cnt: Count indicating default semaphore state: 0: Take semaphore token immediately 1: Keep token available

`uint8_t lwesp_sys_sem_delete (lwesp_sys_sem_t *p)`

Delete binary semaphore.

Return 1 on success, 0 otherwise

Parameters

- [in] p: Pointer to semaphore structure

`uint32_t lwesp_sys_sem_wait (lwesp_sys_sem_t *p, uint32_t timeout)`

Wait for semaphore to be available.

Return Number of milliseconds waited for semaphore to become available or `LWESP_SYS_TIMEOUT` if not available within given time

Parameters

- [in] p: Pointer to semaphore structure
- [in] timeout: Timeout to wait in milliseconds. When 0 is applied, wait forever

`uint8_t lwesp_sys_sem_release (lwesp_sys_sem_t *p)`

Release semaphore.

Return 1 on success, 0 otherwise

Parameters

- [in] p: Pointer to semaphore structure

`uint8_t lwesp_sys_sem_isvalid (lwesp_sys_sem_t *p)`

Check if semaphore is valid.

Return 1 on success, 0 otherwise

Parameters

- [in] p: Pointer to semaphore structure

`uint8_t lwesp_sys_sem_invalid (lwesp_sys_sem_t *p)`

Invalid semaphore.

Return 1 on success, 0 otherwise

Parameters

- [in] p: Pointer to semaphore structure

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 *`

Return 1 on success, 0 otherwise

Parameters

- [out] b: Pointer to message queue structure
- [in] size: Number of entries for message queue to hold

`uint8_t lwesp_sys_mbox_delete (lwesp_sys_mbox_t *b)`
Delete message queue.

Return 1 on success, 0 otherwise

Parameters

- [in] b: Pointer to message queue structure

`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.

Return Time in units of milliseconds needed to put a message to queue

Parameters

- [in] b: Pointer to message queue structure
- [in] m: Pointer to entry to insert to message queue

`uint32_t lwesp_sys_mbox_get (lwesp_sys_mbox_t *b, void **m, uint32_t timeout)`
Get a new entry from message queue with timeout.

Return Time in units of milliseconds needed to put a message to queue or `LWESP_SYS_TIMEOUT` if it was not successful

Parameters

- [in] b: Pointer to message queue structure
- [in] m: Pointer to pointer to result to save value from message queue to
- [in] timeout: Maximal timeout to wait for new message. When 0 is applied, wait for unlimited time

`uint8_t lwesp_sys_mbox_putnow (lwesp_sys_mbox_t *b, void *m)`
Put a new entry to message queue without timeout (now or fail)

Return 1 on success, 0 otherwise

Parameters

- [in] b: Pointer to message queue structure
- [in] m: Pointer to message to save to queue

`uint8_t lwesp_sys_mbox_getnow (lwesp_sys_mbox_t *b, void **m)`

Get an entry from message queue immediately.

Return 1 on success, 0 otherwise

Parameters

- [in] b: Pointer to message queue structure
- [in] m: Pointer to pointer to result to save value from message queue to

`uint8_t lwesp_sys_mbox_isvalid (lwesp_sys_mbox_t *b)`

Check if message queue is valid.

Return 1 on success, 0 otherwise

Parameters

- [in] b: Pointer to message queue structure

`uint8_t lwesp_sys_mbox_invalid (lwesp_sys_mbox_t *b)`

Invalid message queue.

Return 1 on success, 0 otherwise

Parameters

- [in] b: Pointer to message queue structure

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.

Return 1 on success, 0 otherwise

Parameters

- [out] t: Pointer to thread identifier if create was successful. It may be set to NULL
- [in] name: Name of a new thread
- [in] thread_func: Thread function to use as thread body
- [in] arg: Thread function argument
- [in] stack_size: Size of thread stack in uints of bytes. If set to 0, reserve default stack size
- [in] prio: Thread priority

`uint8_t lwesp_sys_thread_terminate (lwesp_sys_thread_t *t)`

Terminate thread (shut it down and remove)

Return 1 on success, 0 otherwise

Parameters

- [in] t: Pointer to thread handle to terminate. If set to NULL, terminate current thread (thread from where function is called)

`uint8_t lwesp_sys_thread_yield (void)`
Yield current thread.

Return 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 prioer 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 osMutexId_t *lwesp_sys_mutex_t*
System mutex type.

It is used by middleware as base type of mutex.

typedef osSemaphoreId_t *lwesp_sys_sem_t*
System semaphore type.

It is used by middleware as base type of mutex.

typedef osMessageQueueId_t *lwesp_sys_mbox_t*
System message queue type.

It is used by middleware as base type of mutex.

typedef osThreadId_t *lwesp_sys_thread_t*
System thread ID type.

typedef osPriority **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

group **LWESP_APP_CAYENNE_API**
MQTT client API for Cayenne.

Defines

LWESP_CAYENNE_API_VERSION
Cayenne API version in string.

LWESP_CAYENNE_HOST
Cayenne host server.

LWESP_CAYENNE_PORT
Cayenne port number.

LWESP_CAYENNE_NO_CHANNEL
No channel macro

LWESP_CAYENNE_ALL_CHANNELS
All channels macro

Typedefs

typedef *lwespr_t* (***lwesp_cayenne_evt_fn**) (**struct** lwesp_cayenne *c, *lwesp_cayenne_evt_t* *evt)

Cayenne event callback function.

Return *lwespOK* on success, member of *lwespr_t* otherwise

Parameters

- [in] c: Cayenne handle
- [in] evt: Event handle

Enums

enum **lwesp_cayenne_topic_t**
List of possible cayenne topics.

Values:

enumerator **LWESP_CAYENNE_TOPIC_DATA**
Data topic

enumerator **LWESP_CAYENNE_TOPIC_COMMAND**
Command topic

enumerator **LWESP_CAYENNE_TOPIC_CONFIG**

```
enumerator LWESP_CAYENNE_TOPIC_RESPONSE
enumerator LWESP_CAYENNE_TOPIC_SYS_MODEL
enumerator LWESP_CAYENNE_TOPIC_SYS_VERSION
enumerator LWESP_CAYENNE_TOPIC_SYS_CPU_MODEL
enumerator LWESP_CAYENNE_TOPIC_SYS_CPU_SPEED
enumerator LWESP_CAYENNE_TOPIC_DIGITAL
enumerator LWESP_CAYENNE_TOPIC_DIGITAL_COMMAND
enumerator LWESP_CAYENNE_TOPIC_DIGITAL_CONFIG
enumerator LWESP_CAYENNE_TOPIC_ANALOG
enumerator LWESP_CAYENNE_TOPIC_ANALOG_COMMAND
enumerator LWESP_CAYENNE_TOPIC_ANALOG_CONFIG
enumerator LWESP_CAYENNE_TOPIC_END
```

Last entry

```
enum lwesp_cayenne_rlwesp_t
```

Cayenne response types.

Values:

```
enumerator LWESP_CAYENNE_RLWESP_OK
```

Response OK

```
enumerator LWESP_CAYENNE_RLWESP_ERROR
```

Response error

```
enum lwesp_cayenne_evt_type_t
```

Cayenne events.

Values:

```
enumerator LWESP_CAYENNE_EVT_CONNECT
```

Connect to Cayenne event

```
enumerator LWESP_CAYENNE_EVT_DISCONNECT
```

Disconnect from Cayenne event

```
enumerator LWESP_CAYENNE_EVT_DATA
```

Data received event

Functions

```
lwespr_t lwesp_cayenne_create (lwesp_cayenne_t *c, const lwesp_mqtt_client_info_t
                               *client_info, lwesp_cayenne_evt_fn evt_fn)
```

Create new instance of cayenne MQTT connection.

Note Each call to this functions starts new thread for async receive processing. Function will block until thread is created and successfully started

Return *lwesprOK* on success, member of *lwespr_t* otherwise

Parameters

- [in] c: Cayenne empty handle

- [in] `client_info`: MQTT client info with username, password and id
- [in] `evt_fn`: Event function

lwespr_t **lwesp_cayenne_subscribe** (*lwesp_cayenne_t* *c, *lwesp_cayenne_topic_t* topic, uint16_t channel)

Subscribe to cayenne based topics and channels.

Return *lwespOK* on success, member of *lwespr_t* otherwise

Parameters

- [in] `c`: Cayenne handle
- [in] `topic`: Cayenne topic
- [in] `channel`: Optional channel number. Use *LWESP_CAYENNE_NO_CHANNEL* when channel is not needed or *LWESP_CAYENNE_ALL_CHANNELS* to subscribe to all channels

lwespr_t **lwesp_cayenne_publish_data** (*lwesp_cayenne_t* *c, *lwesp_cayenne_topic_t* topic, uint16_t channel, const char *type, const char *unit, const char *data)

lwespr_t **lwesp_cayenne_publish_float** (*lwesp_cayenne_t* *c, *lwesp_cayenne_topic_t* topic, uint16_t channel, const char *type, const char *unit, float f)

lwespr_t **lwesp_cayenne_publish_response** (*lwesp_cayenne_t* *c, *lwesp_cayenne_msg_t* *msg, *lwesp_cayenne_rlwesp_t* resp, const char *message)

Publish response message to command.

Return *lwespOK* on success, member of *lwespr_t* otherwise

Parameters

- [in] `c`: Cayenne handle
- [in] `msg`: Received message with command topic
- [in] `resp`: Response type, either OK or ERROR
- [in] `message`: Message text in case of error to be displayed to Cayenne dashboard

struct lwesp_cayenne_key_value_t
#include <lwesp_cayenne.h> Key/Value pair structure.

Public Members

const char *key
Key string

const char *value
Value string

struct lwesp_cayenne_msg_t
#include <lwesp_cayenne.h> Cayenne message.

Public Members

lwesp_cayenne_topic_t **topic**
Message topic

uint16_t **channel**
Message channel, optional, based on topic type

const char ***seq**
Sequence string on command

lwesp_cayenne_key_value_t **values**[2]
Key/Value pair of values

size_t **values_count**
Count of valid pairs in values member

struct lwesp_cayenne_evt_t
#include <lwesp_cayenne.h> Cayenne event.

Public Members

lwesp_cayenne_evt_type_t **type**
Event type

lwesp_cayenne_msg_t ***msg**
Pointer to data message

struct lwesp_cayenne_evt_t::[anonymous]::[anonymous] data
Data event, used with *LWESP_CAYENNE_EVT_DATA* event

union lwesp_cayenne_evt_t::[anonymous] evt
Event union

struct lwesp_cayenne_t
#include <lwesp_cayenne.h> Cayenne handle.

Public Members

lwesp_mqtt_client_api_p **api_c**
MQTT API client

const *lwesp_mqtt_client_info_t* ***info_c**
MQTT Client info structure

lwesp_cayenne_msg_t **msg**
Received data message

lwesp_cayenne_evt_t **evt**
Event handle

lwesp_cayenne_evt_fn **evt_fn**
Event callback function

lwesp_sys_thread_t **thread**
Cayenne thread handle

lwesp_sys_sem_t **sem**
Sync semaphore handle

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.

Note May only be called from SSI callback function

Return Number of bytes written to output

See *lwesp_http_server_write*

Parameters

- [in] *hs*: HTTP handle
- [in] *str*: String to write

Typedefs

typedef char **(*http_cgi_fn)** (*http_param_t* *params, size_t params_len)

CGI callback function.

Return Function must return a new URI which is used later as response string, such as “/index.html” or similar

Parameters

- [in] *params*: Pointer to list of parameteres and their values
- [in] *params_len*: Number of parameters

typedef *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.

Return *lwespOK* on success, member of *lwespr_t* otherwise

Parameters

- [in] *hs*: HTTP state
- [in] *uri*: POST request URI
- [in] *content_length*: Total content length (Content-Length HTTP parameter) in units of bytes

typedef *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

Return *lwespOK* on success, member of *lwespr_t* otherwise

Parameters

- [in] *hs*: HTTP state
- [in] *pbuf*: Packet buffer with received data

typedef *lwespr_t* (***http_post_end_fn**) (**struct** http_state **hs*)
End of POST data request function prototype.

Return *lwespOK* on success, member of *lwespr_t* otherwise

Parameters

- [in] *hs*: HTTP state

typedef size_t (***http_ssi_fn**) (**struct** http_state **hs*, **const** char **tag_name*, size_t *tag_len*)
SSI (Server Side Includes) callback function prototype.

Note User can use server write functions to directly write to connection output

Parameters

- [in] *hs*: HTTP state
- [in] *tag_name*: Name of TAG to replace with user content
- [in] *tag_len*: Length of TAG

Return Value

- 1: Everything was written on this tag
- 0: There are still data to write to output which means callback will be called again for user to process all the data

typedef uint8_t (***http_fs_open_fn**) (**struct** http_fs_file **file*, **const** char **path*)
File system open file function Function is called when user file system (FAT or similar) should be invoked to open a file from specific path.

Return 1 if file is opened, 0 otherwise

Parameters

- [in] *file*: Pointer to file where user has to set length of file if opening was successful
- [in] *path*: Path of file to open

typedef uint32_t (***http_fs_read_fn**) (**struct** http_fs_file **file*, void **buff*, size_t *btr*)
File system read file function Function may be called for 2 purposes. First is to read data and second to get remaining length of file to read.

Return Number of bytes read or number of bytes available to read

Parameters

- [in] *file*: File pointer to read content
- [in] *buff*: Buffer to read data to. When parameter is set to NULL, number of remaining bytes available to read should be returned

- [in] `btr`: Number of bytes to read from file. This parameter has no meaning when `buff` is `NULL`

typedef `uint8_t(*http_fs_close_fn)(struct http_fs_file *file)`
Close file callback function.

Return 1 on success, 0 otherwise

Parameters

- [in] `file`: File to close

Enums

enum `http_req_method_t`

Request method type.

Values:

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`
Waiting beginning of tag

enumerator `HTTP_SSI_STATE_BEGIN`
Beginning detected, parsing it

enumerator `HTTP_SSI_STATE_TAG`
Parsing TAG value

enumerator `HTTP_SSI_STATE_END`
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.

Return *lwespOK* on success, member of *lwespr_t* otherwise

Parameters

- [in] `init`: Initialization structure for server
- [in] `port`: Port for HTTP server, usually 80

`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

Return Number of bytes written

Parameters

- [in] *hs*: HTTP state
- [in] *data*: Data to write
- [in] *len*: Length of bytes to write

struct http_param_t

#include <lwesp_http_server.h> HTTP parameters on http URI in format ?
param1=value1¶m2=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 function for post data

http_post_end_fn post_end_fn
Callback function 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.

Return 1 on success, 0 otherwise

Parameters

- [in] file: File structure to fill if file is successfully open
- [in] path: File path to open in format “/js/scripts.js” or “/index.html”

uint32_t **http_fs_read** (*http_fs_file_t* *file, void *buff, size_t btr)

Read a file content.

Return Number of bytes read or number of bytes available to read

Parameters

- [in] file: File handle to read
- [out] buff: Buffer to read data to. When set to NULL, function should return remaining available data to read
- [in] btr: Number of bytes to read. Has no meaning when buff = NULL

uint8_t **http_fs_close** (*http_fs_file_t* *file)

Close a file handle.

Return 1 on success, 0 otherwise

Parameters

- [in] file: File handle

MQTT Client

MQTT client v3.1.1 implementation, based on callback (non-netconn) connection API.

Listing 23: MQTT application example code

```

1  /*
2   * MQTT client example with ESP device.
3   *
4   * Once device is connected to network,
5   * it will try to connect to mosquitto test server and start the MQTT.
6   *
7   * If successfully connected, it will publish data to "esp8266_mqtt_topic" topic_
↳every x seconds.
8   *
9   * To check if data are sent, you can use mqtt-spy PC software to inspect
10  * test.mosquitto.org server and subscribe to publishing topic
11  */
12
13 #include "lwesp/apps/lwesp_mqtt_client.h"
14 #include "lwesp/lwesp.h"
15 #include "lwesp/lwesp_timeout.h"
16 #include "mqtt_client.h"
17
18 /**
19  * \brief          MQTT client structure
20  */
21 static lwesp_mqtt_client_p
22 mqtt_client;
23
24 /**
25  * \brief          Client ID is structured from ESP station MAC address
26  */
27 static char
28 mqtt_client_id[13];
29
30 /**
31  * \brief          Connection information for MQTT CONNECT packet
32  */
33 static const lwesp_mqtt_client_info_t
34 mqtt_client_info = {
35     .id = mqtt_client_id,                /* The only required field for_
↳connection! */
36
37     .keep_alive = 10,
38     // .user = "test_username",
39     // .pass = "test_password",
40 };
41
42 static void mqtt_cb(lwesp_mqtt_client_p client, lwesp_mqtt_evt_t* evt);
43 static void example_do_connect(lwesp_mqtt_client_p client);
44 static uint32_t retries = 0;
45
46 /**
47  * \brief          Custom callback function for ESP events
48  */
49 static lwespr_t
50 mqtt_lwesp_cb(lwesp_evt_t* evt) {

```

(continues on next page)

(continued from previous page)

```

51     switch (lwesp_evt_get_type(evt)) {
52 #if LWESP_CFG_MODE_STATION
53     case LWESP_EVT_WIFI_GOT_IP: {
54         example_do_connect(mqtt_client);    /* Start connection after we have a
↳connection to network client */
55         break;
56     }
57 #endif /* LWESP_CFG_MODE_STATION */
58     default:
59         break;
60     }
61     return lwespOK;
62 }
63
64 /**
65  * \brief          MQTT client thread
66  * \param[in]      arg: User argument
67  */
68 void
69 mqtt_client_thread(void const* arg) {
70     lwesp_mac_t mac;
71
72     lwesp_evt_register(mqtt_lwesp_cb);      /* Register new callback for
↳general events from ESP stack */
73
74     /* Get station MAC to format client ID */
75     if (lwesp_sta_getmac(&mac, NULL, NULL, 1) == lwespOK) {
76         snprintf(mqtt_client_id, sizeof(mqtt_client_id), "%02X%02X%02X%02X%02X",
77                 (unsigned)mac.mac[0], (unsigned)mac.mac[1], (unsigned)mac.mac[2],
78                 (unsigned)mac.mac[3], (unsigned)mac.mac[4], (unsigned)mac.mac[5]
79                 );
80     } else {
81         strcpy(mqtt_client_id, "unknown");
82     }
83     printf("MQTT Client ID: %s\r\n", mqtt_client_id);
84
85     /*
86      * Create a new client with 256 bytes of RAW TX data
87      * and 128 bytes of RAW incoming data
88      */
89     mqtt_client = lwesp_mqtt_client_new(256, 128); /* Create new MQTT client */
90     if (lwesp_sta_is_joined()) {                /* If ESP is already joined to
↳network */
91         example_do_connect(mqtt_client);    /* Start connection to MQTT server */
92     }
93
94     /* Make dummy delay of thread */
95     while (1) {
96         lwesp_delay(1000);
97     }
98 }
99
100 /**
101  * \brief          Timeout callback for MQTT events
102  * \param[in]      arg: User argument
103  */
104 void

```

(continues on next page)

```

105 mqtt_timeout_cb(void* arg) {
106     static uint32_t num = 10;
107     lwesp_mqtt_client_p client = arg;
108     lwespr_t res;
109
110     static char tx_data[20];
111
112     if (lwesp_mqtt_client_is_connected(client)) {
113         sprintf(tx_data, "R: %u, N: %u", (unsigned)retries, (unsigned)num);
114         if ((res = lwesp_mqtt_client_publish(client, "esp8266_mqtt_topic", tx_data,
↳LWESP_U16(strlen(tx_data)), LWESP_MQTT_QOS_EXACTLY_ONCE, 0, (void*)num)) ==
↳lwespOK) {
115             printf("Publishing %d...\r\n", (int)num);
116             num++;
117         } else {
118             printf("Cannot publish...: %d\r\n", (int)res);
119         }
120     }
121     lwesp_timeout_add(10000, mqtt_timeout_cb, client);
122 }
123
124 /**
125  * \brief          MQTT event callback function
126  * \param[in]      client: MQTT client where event occurred
127  * \param[in]      evt: Event type and data
128  */
129 static void
130 mqtt_cb(lwesp_mqtt_client_p client, lwesp_mqtt_evt_t* evt) {
131     switch (lwesp_mqtt_client_evt_get_type(client, evt)) {
132         /*
133          * Connect event
134          * Called if user successfully connected to MQTT server
135          * or even if connection failed for some reason
136          */
137         case LWESP_MQTT_EVT_CONNECT: { /* MQTT connect event occurred */
138             lwesp_mqtt_conn_status_t status = lwesp_mqtt_client_evt_connect_get_
↳status(client, evt);
139
140             if (status == LWESP_MQTT_CONN_STATUS_ACCEPTED) {
141                 printf("MQTT accepted!\r\n");
142                 /*
143                  * Once we are accepted by server,
144                  * it is time to subscribe to different topics
145                  * We will subscribe to "mqtt_lwesp_example_topic" topic,
146                  * and will also set the same name as subscribe argument for callback.
↳later
147                  */
148                 lwesp_mqtt_client_subscribe(client, "esp8266_mqtt_topic", LWESP_MQTT_
↳QOS_EXACTLY_ONCE, "esp8266_mqtt_topic");
149
150                 /* Start timeout timer after 5000ms and call mqtt_timeout_cb function.
↳*/
151                 lwesp_timeout_add(5000, mqtt_timeout_cb, client);
152             } else {
153                 printf("MQTT server connection was not successful: %d\r\n",
↳ (int)status);
154

```

(continues on next page)

(continued from previous page)

```

155         /* Try to connect all over again */
156         example_do_connect(client);
157     }
158     break;
159 }
160
161 /*
162  * Subscribe event just happened.
163  * Here it is time to check if it was successful or failed attempt
164  */
165 case LWESP_MQTT_EVT_SUBSCRIBE: {
166     const char* arg = lwesp_mqtt_client_evt_subscribe_get_argument(client,
167 ↪ evt); /* Get user argument */
168     lwespr_t res = lwesp_mqtt_client_evt_subscribe_get_result(client, evt); /
169 ↪ * Get result of subscribe event */
170
171     if (res == lwespOK) {
172         printf("Successfully subscribed to %s topic\r\n", arg);
173         if (!strcmp(arg, "esp8266_mqtt_topic")) { /* Check topic name we
174 ↪ were subscribed */
175             /* Subscribed to "esp8266_mqtt_topic" topic */
176
177             /*
178              * Now publish an even on example topic
179              * and set QoS to minimal value which does not guarantee message
180 ↪ delivery to received
181              */
182             lwesp_mqtt_client_publish(client, "esp8266_mqtt_topic", "test_data
183 ↪ ", 9, LWESP_MQTT_QOS_AT_MOST_ONCE, 0, (void*)1);
184         }
185     }
186     break;
187 }
188
189 /* Message published event occurred */
190 case LWESP_MQTT_EVT_PUBLISH: {
191     uint32_t val = (uint32_t)lwesp_mqtt_client_evt_publish_get_
192 ↪ argument(client, evt); /* Get user argument, which is in fact our custom number */
193
194     printf("Publish event, user argument on message was: %d\r\n", (int)val);
195     break;
196 }
197
198 /*
199  * A new message was published to us
200  * and now it is time to read the data
201  */
202 case LWESP_MQTT_EVT_PUBLISH_RECV: {
203     const char* topic = lwesp_mqtt_client_evt_publish_recv_get_topic(client,
204 ↪ evt);
205     size_t topic_len = lwesp_mqtt_client_evt_publish_recv_get_topic_
206 ↪ len(client, evt);
207     const uint8_t* payload = lwesp_mqtt_client_evt_publish_recv_get_
208 ↪ payload(client, evt);
209     size_t payload_len = lwesp_mqtt_client_evt_publish_recv_get_payload_
210 ↪ len(client, evt);

```

(continues on next page)

```

202     LWESP_UNUSED(payload);
203     LWESP_UNUSED(payload_len);
204     LWESP_UNUSED(topic);
205     LWESP_UNUSED(topic_len);
206     break;
207 }
208
209 /* Client is fully disconnected from MQTT server */
210 case LWESP_MQTT_EVT_DISCONNECT: {
211     printf("MQTT client disconnected!\r\n");
212     example_do_connect(client); /* Connect to server all over again */
213     break;
214 }
215
216 default:
217     break;
218 }
219 }
220
221 /** Make a connection to MQTT server in non-blocking mode */
222 static void
223 example_do_connect(lwesp_mqtt_client_p client) {
224     if (client == NULL) {
225         return;
226     }
227
228     /*
229      * Start a simple connection to open source
230      * MQTT server on mosquitto.org
231      */
232     retries++;
233     lwesp_timeout_remove(mqtt_timeout_cb);
234     lwesp_mqtt_client_connect(mqtt_client, "test.mosquitto.org", 1883, mqtt_cb, &mqtt_
↪client_info);
235 }

```

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.

Parameters

- [in] client: MQTT client
- [in] evt: 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

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

Delivery is guaranteed at least once, but it may be delivered multiple times with the same content

enumerator LWESP_MQTT_QOS_EXACTLY_ONCE

Delivery is guaranteed 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

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 sent to server and reply received

enum lwesp_mqtt_conn_status_t

List of possible results from MQTT server when executing connect command.

Values:

enumerator LWESP_MQTT_CONN_STATUS_ACCEPTED
Connection accepted and ready to use

enumerator LWESP_MQTT_CONN_STATUS_REFUSED_PROTOCOL_VERSION
Connection Refused, unacceptable protocol version

enumerator LWESP_MQTT_CONN_STATUS_REFUSED_ID
Connection refused, identifier rejected

enumerator LWESP_MQTT_CONN_STATUS_REFUSED_SERVER
Connection refused, server unavailable

enumerator LWESP_MQTT_CONN_STATUS_REFUSED_USER_PASS
Connection refused, bad user name or password

enumerator LWESP_MQTT_CONN_STATUS_REFUSED_NOT_AUTHORIZED
Connection refused, not authorized

enumerator LWESP_MQTT_CONN_STATUS_TCP_FAILED
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.

Return Pointer to new allocated MQTT client structure or NULL on failure

Parameters

- [in] *tx_buff_len*: Length of raw data output buffer
- [in] *rx_buff_len*: Length of raw data input buffer

void **lwesp_mqtt_client_delete** (*lwesp_mqtt_client_p client*)
Delete MQTT client structure.

Note MQTT client must be disconnected first

Parameters

- [in] *client*: MQTT client

lwesp_r_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.

Note After TCP connection is established, CONNECT packet is automatically sent to server

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

Parameters

- [in] `client`: MQTT client
- [in] `host`: Host address for server
- [in] `port`: Host port number
- [in] `evt_fn`: Callback function for all events on this MQTT client
- [in] `info`: Information structure for connection

lwespr_t **lwesp_mqtt_client_disconnect** (*lwesp_mqtt_client_p* client)
Disconnect from MQTT server.

Return *lwespOK* if request sent to queue or member of *lwespr_t* otherwise

Parameters

- [in] `client`: MQTT client

`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

Return 1 on success, 0 otherwise

Parameters

- [in] `client`: MQTT client

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.

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

Parameters

- [in] `client`: MQTT client
- [in] `topic`: Topic name to subscribe to
- [in] `qos`: Quality of service. This parameter can be a value of *lwesp_mqtt_qos_t*
- [in] `arg`: User custom argument used in callback

lwespr_t **lwesp_mqtt_client_unsubscribe** (*lwesp_mqtt_client_p* client, **const** char *topic, void *arg)
Unsubscribe from MQTT topic.

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

Parameters

- [in] `client`: MQTT client
- [in] `topic`: Topic name to unsubscribe from
- [in] `arg`: User custom argument used in callback

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.

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

Parameters

- [in] client: MQTT client
- [in] topic: Topic to send message to
- [in] payload: Message data
- [in] payload_len: Length of payload data
- [in] qos: Quality of service. This parameter can be a value of *lwesp_mqtt_qos_t* enumeration
- [in] retain: Retian parameter value
- [in] arg: User custom argument used in callback

void ***lwesp_mqtt_client_get_arg** (*lwesp_mqtt_client_p* client)

Get user argument on client.

Return User argument

Parameters

- [in] client: MQTT client handle

void **lwesp_mqtt_client_set_arg** (*lwesp_mqtt_client_p* client, void *arg)

Set user argument on client.

Parameters

- [in] client: MQTT client handle
- [in] arg: 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

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.

Return Connection status. Member of `lwesp_mqtt_conn_status_t`

Parameters

- [in] *client*: MQTT client
- [in] *evt*: Event handle

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.

Return 1 on success, 0 otherwise

Parameters

- [in] *client*: MQTT client
- [in] *evt*: Event handle

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*.

Return User argument

Parameters

- [in] *client*: MQTT client
- [in] *evt*: Event handle

lwesp_mqtt_client_evt_subscribe_get_result (*client, evt*)

Get result of subscribe event.

Return *lwespOK* on success, member of *lwespr_t* otherwise

Parameters

- [in] *client*: MQTT client
- [in] *evt*: Event handle

lwesp_mqtt_client_evt_unsubscribe_get_argument (*client, evt*)

Get user argument used on *lwesp_mqtt_client_unsubscribe*.

Return User argument

Parameters

- [in] *client*: MQTT client
- [in] *evt*: Event handle

lwesp_mqtt_client_evt_unsubscribe_get_result (*client, evt*)

Get result of unsubscribe event.

Return *lwespOK* on success, member of *lwespr_t* otherwise

Parameters

- [in] *client*: MQTT client
- [in] *evt*: Event handle

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.

Return Topic name

Parameters

- [in] `client`: MQTT client
- [in] `evt`: Event handle

`lwesp_mqtt_client_evt_publish_rcv_get_topic_len` (*client, evt*)

Get topic length from received publish packet.

Return Topic length

Parameters

- [in] `client`: MQTT client
- [in] `evt`: Event handle

`lwesp_mqtt_client_evt_publish_rcv_get_payload` (*client, evt*)

Get payload from received publish packet.

Return Packet payload

Parameters

- [in] `client`: MQTT client
- [in] `evt`: Event handle

`lwesp_mqtt_client_evt_publish_rcv_get_payload_len` (*client, evt*)

Get payload length from received publish packet.

Return Payload length

Parameters

- [in] `client`: MQTT client
- [in] `evt`: Event handle

`lwesp_mqtt_client_evt_publish_rcv_is_duplicate` (*client, evt*)

Check if packet is duplicated.

Return 1 if duplicated, 0 otherwise

Parameters

- [in] `client`: MQTT client
- [in] `evt`: Event handle

`lwesp_mqtt_client_evt_publish_rcv_get_qos` (*client, evt*)

Get received quality of service.

Return Member of `lwesp_mqtt_qos_t` enumeration

Parameters

- [in] `client`: MQTT client
- [in] `evt`: Event handle

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*.

Return User argument

Parameters

- [in] *client*: MQTT client
- [in] *evt*: Event handle

lwesp_mqtt_client_evt_publish_get_result (*client, evt*)
Get result of publish event.

Return *lwespOK* on success, member of *lwespr_t* otherwise

Parameters

- [in] *client*: MQTT client
- [in] *evt*: Event handle

Defines

lwesp_mqtt_client_evt_get_type (*client, evt*)
Get MQTT event type.

Return MQTT Event type, value of *lwesp_mqtt_evt_type_t* enumeration

Parameters

- [in] *client*: MQTT client
- [in] *evt*: Event handle

MQTT Client API

MQTT Client API provides sequential API built on top of *MQTT Client*.

Listing 24: MQTT API application example code

```

1  /*
2  * MQTT client API example with ESP device.
3  *
4  * Once device is connected to network,
5  * it will try to connect to mosquitto test server and start the MQTT.
6  *
7  * If successfully connected, it will publish data to "lwesp_mqtt_topic" topic every_
  ↪ x seconds.
8  *
9  * To check if data are sent, you can use mqtt-spy PC software to inspect
10 * test.mosquitto.org server and subscribe to publishing topic
11 */
12

```

(continues on next page)

```

13 #include "lwesp/apps/lwesp_mqtt_client_api.h"
14 #include "mqtt_client_api.h"
15 #include "lwesp/lwesp_mem.h"
16
17 /**
18  * \brief      Connection information for MQTT CONNECT packet
19  */
20 static const lwesp_mqtt_client_info_t
21 mqtt_client_info = {
22     .keep_alive = 10,
23
24     /* Server login data */
25     .user = "8a215f70-a644-11e8-ac49-e932ed599553",
26     .pass = "26aa943f702e5e780f015cd048a91e8fb54cca28",
27
28     /* Device identifier address */
29     .id = "869f5a20-af9c-11e9-b01f-db5cf74e7fb7",
30 };
31
32 /**
33  * \brief      Memory for temporary topic
34  */
35 static char
36 mqtt_topic_str[256];
37
38 /**
39  * \brief      Generate random number and write it to string
40  * \param[out] str: Output string with new number
41  */
42 void
43 generate_random(char* str) {
44     static uint32_t random_beg = 0x8916;
45     random_beg = random_beg * 0x00123455 + 0x85654321;
46     sprintf(str, "%u", (unsigned)((random_beg >> 8) & 0xFFFF));
47 }
48
49 /**
50  * \brief      MQTT client API thread
51  */
52 void
53 mqtt_client_api_thread(void const* arg) {
54     lwesp_mqtt_client_api_p client;
55     lwesp_mqtt_conn_status_t conn_status;
56     lwesp_mqtt_client_api_buf_p buf;
57     lwespr_t res;
58     char random_str[10];
59
60     /* Create new MQTT API */
61     client = lwesp_mqtt_client_api_new(256, 128);
62     if (client == NULL) {
63         goto terminate;
64     }
65
66     while (1) {
67         /* Make a connection */
68         printf("Joining MQTT server\r\n");
69

```

(continues on next page)

(continued from previous page)

```

70     /* Try to join */
71     conn_status = lwesp_mqtt_client_api_connect(client, "mqtt.mydevices.com", ↵
↵1883, &mqtt_client_info);
72     if (conn_status == LWESP_MQTT_CONN_STATUS_ACCEPTED) {
73         printf("Connected and accepted!\r\n");
74         printf("Client is ready to subscribe and publish to new messages\r\n");
75     } else {
76         printf("Connect API response: %d\r\n", (int)conn_status);
77         lwesp_delay(5000);
78         continue;
79     }
80
81     /* Subscribe to topics */
82     sprintf(mqtt_topic_str, "v1/%s/things/%s/cmd/#", mqtt_client_info.user, mqtt_
↵client_info.id);
83     if (lwesp_mqtt_client_api_subscribe(client, mqtt_topic_str, LWESP_MQTT_QOS_AT_
↵LEAST_ONCE) == lwespOK) {
84         printf("Subscribed to topic\r\n");
85     } else {
86         printf("Problem subscribing to topic!\r\n");
87     }
88
89     while (1) {
90         /* Receive MQTT packet with 1000ms timeout */
91         res = lwesp_mqtt_client_api_receive(client, &buf, 5000);
92         if (res == lwespOK) {
93             if (buf != NULL) {
94                 printf("Publish received!\r\n");
95                 printf("Topic: %s, payload: %s\r\n", buf->topic, buf->payload);
96                 lwesp_mqtt_client_api_buf_free(buf);
97                 buf = NULL;
98             }
99             } else if (res == lwespCLOSED) {
100                 printf("MQTT connection closed!\r\n");
101                 break;
102             } else if (res == lwespTIMEOUT) {
103                 printf("Timeout on MQTT receive function. Manually publishing.\r\n");
104
105                 /* Publish data on channel 1 */
106                 generate_random(random_str);
107                 sprintf(mqtt_topic_str, "v1/%s/things/%s/data/1", mqtt_client_info.
↵user, mqtt_client_info.id);
108                 lwesp_mqtt_client_api_publish(client, mqtt_topic_str, random_str, ↵
↵strlen(random_str), LWESP_MQTT_QOS_AT_LEAST_ONCE, 0);
109             }
110         }
111         //goto terminate;
112     }
113
114 terminate:
115     lwesp_mqtt_client_api_delete(client);
116     printf("MQTT client thread terminate\r\n");
117     lwesp_sys_thread_terminate(NULL);
118 }

```

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.

Return Client handle on success, NULL otherwise

Parameters

- [in] tx_buff_len: Maximal TX buffer for maximal packet length
- [in] rx_buff_len: Maximal RX buffer

void **lwesp_mqtt_client_api_delete** (lwesp_mqtt_client_api_p client)
Delete client from memory.

Parameters

- [in] client: 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.

Return *LWESP_MQTT_CONN_STATUS_ACCEPTED* on success, member of *lwesp_mqtt_conn_status_t* otherwise

Parameters

- [in] client: MQTT API client handle
- [in] host: TCP host
- [in] port: TCP port
- [in] info: MQTT client info

lwespr_t **lwesp_mqtt_client_api_close** (lwesp_mqtt_client_api_p client)
Close MQTT connection.

Return *lwespOK* on success, member of *lwespr_t* otherwise

Parameters

- [in] client: MQTT API client handle

lwespr_t **lwesp_mqtt_client_api_subscribe** (lwesp_mqtt_client_api_p client, **const** char *topic, *lwesp_mqtt_qos_t* qos)

Subscribe to topic.

Return *lwespOK* on success, member of *lwespr_t* otherwise

Parameters

- [in] `client`: MQTT API client handle
- [in] `topic`: Topic to subscribe on
- [in] `qos`: Quality of service. This parameter can be a value of *lwesp_mqtt_qos_t*

lwespr_t **lwesp_mqtt_client_api_unsubscribe** (*lwesp_mqtt_client_api_p* `client`, **const** char `*topic`)

Unsubscribe from topic.

Return *lwespOK* on success, member of *lwespr_t* otherwise

Parameters

- [in] `client`: MQTT API client handle
- [in] `topic`: Topic to unsubscribe from

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.

Return *lwespOK* on success, member of *lwespr_t* otherwise

Parameters

- [in] `client`: MQTT API client handle
- [in] `topic`: Topic to publish on
- [in] `data`: Data to send
- [in] `btw`: Number of bytes to send for data parameter
- [in] `qos`: Quality of service. This parameter can be a value of *lwesp_mqtt_qos_t*
- [in] `retain`: Set to 1 for retain flag, 0 otherwise

`uint8_t` **lwesp_mqtt_client_api_is_connected** (*lwesp_mqtt_client_api_p* `client`)
Check if client MQTT connection is active.

Return 1 on success, 0 otherwise

Parameters

- [in] `client`: MQTT API client handle

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

Return *lwespOK* on success, *lwespCLOSED* if MQTT is closed, *lwespTIMEOUT* on timeout

Parameters

- [in] `client`: MQTT API client handle
- [in] `p`: Pointer to output buffer

- [in] `timeout`: Maximal time to wait before function returns timeout

void **lwesp_mqtt_client_api_buf_free** (*lwesp_mqtt_client_api_buf_p*)
Free buffer memory after usage.

Parameters

- [in] `p`: 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.

<p>Warning: Netconn API are designed to be called from application threads ONLY. It is not allowed to call any of <i>netconn API</i> functions from within interrupt or callback event functions.</p>
--

Netconn client

Fig. 9: Netconn API client block diagram

Above block diagram shows basic architecture of netconn client application. There is always one application thread (in green) which calls *netconn API* functions to interact with connection API in synchronous mode.

Every netconn connection uses dedicated structure to handle message queue for data received packet buffers. Each time new packet is received (red block, *data received event*), reference to it is written to message queue of netconn structure, while application thread reads new entries from the same queue to get packets.

Listing 25: Netconn client example

```

1  #include "netconn_client.h"
2  #include "lwesp/lwesp.h"
3
4  /**
5   * \brief      Host and port settings
6   */
7  #define NETCONN_HOST      "example.com"
8  #define NETCONN_PORT     80
9
10 /**
11  * \brief      Request header to send on successful connection
12  */
13  static const char
14  request_header[] = "
15                    "GET / HTTP/1.1\r\n"
16                    "Host: " NETCONN_HOST "\r\n"
17                    "Connection: close\r\n"
18                    "\r\n";
19
20 /**
21  * \brief      Netconn client thread implementation
22  * \param[in]  arg: User argument
23  */
24  void
25  netconn_client_thread(void const* arg) {
26      lwespr_t res;
27      lwesp_pbuf_p pbuf;
28      lwesp_netconn_p client;
29      lwesp_sys_sem_t* sem = (void*)arg;
30
31      /*
32       * First create a new instance of netconn
33       * connection and initialize system message boxes
34       * to accept received packet buffers
35       */
36      client = lwesp_netconn_new(LWESP_NETCONN_TYPE_TCP);
37      if (client != NULL) {
38          /*
39           * Connect to external server as client
40           * with custom NETCONN_CONN_HOST and CONN_PORT values
41           */
42           * Function will block thread until we are successfully connected (or not) to
43           ↪ server

```

(continues on next page)

(continued from previous page)

```

43     */
44     res = lwesp_netconn_connect(client, NETCONN_HOST, NETCONN_PORT);
45     if (res == lwespOK) { /* Are we successfully connected? */
46         printf("Connected to " NETCONN_HOST "\r\n");
47         res = lwesp_netconn_write(client, request_header, sizeof(request_header) -
↳ 1); /* Send data to server */
48         if (res == lwespOK) {
49             res = lwesp_netconn_flush(client); /* Flush data to output */
50         }
51         if (res == lwespOK) { /* Were data sent? */
52             printf("Data were successfully sent to server\r\n");
53
54             /*
55              * Since we sent HTTP request,
56              * we are expecting some data from server
57              * or at least forced connection close from remote side
58              */
59             do {
60                 /*
61                  * Receive single packet of data
62                  *
63                  * Function will block thread until new packet
64                  * is ready to be read from remote side
65                  *
66                  * After function returns, don't forgot the check value.
67                  * Returned status will give you info in case connection
68                  * was closed too early from remote side
69                  */
70                 res = lwesp_netconn_receive(client, &pbuf);
71                 if (res == lwespCLOSED) { /* Was the connection closed? This
↳ can be checked by return status of receive function */
72                     printf("Connection closed by remote side...\r\n");
73                     break;
74                 } else if (res == lwespTIMEOUT) {
75                     printf("Netconn timeout while receiving data. You may try
↳ multiple readings before deciding to close manually\r\n");
76                 }
77
78                 if (res == lwespOK && pbuf != NULL) { /* Make sure we have valid
↳ packet buffer */
79                     /*
80                      * At this point read and manipulate
81                      * with received buffer and check if you expect more data
82                      *
83                      * After you are done using it, it is important
84                      * you free the memory otherwise memory leaks will appear
85                      */
86                     printf("Received new data packet of %d bytes\r\n", (int)lwesp_
↳ pbuf_length(pbuf, 1));
87                     lwesp_pbuf_free(pbuf); /* Free the memory after usage */
88                     pbuf = NULL;
89                 }
90             } while (1);
91         } else {
92             printf("Error writing data to remote host!\r\n");
93         }
94     }

```

(continues on next page)

(continued from previous page)

```

95     /*
96     * Check if connection was closed by remote server
97     * and in case it wasn't, close it manually
98     */
99     if (res != lwespCLOSED) {
100         lwesp_netconn_close(client);
101     }
102     } else {
103         printf("Cannot connect to remote host %s:%d!\r\n", NETCONN_HOST, NETCONN_
↪PORT);
104     }
105     lwesp_netconn_delete(client);           /* Delete netconn structure */
106 }
107
108 if (lwesp_sys_sem_isvalid(sem)) {
109     lwesp_sys_sem_release(sem);
110 }
111 lwesp_sys_thread_terminate(NULL);        /* Terminate current thread */
112 }

```

Netconn server

Fig. 10: Netconn API server block diagram

When netconn is configured in server mode, it is possible to accept new clients from remote side. Application creates *netconn server connection*, which can only accept *clients* and cannot send/receive any data. It configures server on dedicated port (selected by application) and listens on it.

When new client connects, *server callback function* is called with *new active connection event*. Newly accepted connection is then written to server structure netconn which is later read by application thread. At the same time, *netconn connection* structure (blue) is created to allow standard send/receive operation on active connection.

Note: Each connected client has its own *netconn connection* structure. When multiple clients connect to server at the same time, multiple entries are written to *connection accept* message queue and are ready to be processed by application thread.

From this point, program flow is the same as in case of *netconn client*.

This is basic example for netconn thread. It waits for client and processes it in blocking mode.

Warning: When multiple clients connect at the same time to netconn server, they are processed one-by-one, sequentially. This may introduce delay in response for other clients. Check netconn concurrency option to process multiple clients at the same time

Listing 26: Netconn server with single processing thread

```

1  /*
2  * Netconn server example is based on single thread
3  * and it listens for single client only on port 23

```

(continues on next page)

```

4  */
5  #include "netconn_server_lthread.h"
6  #include "lwesp/lwesp.h"
7
8  /**
9   * \brief      Basic thread for netconn server to test connections
10  * \param[in]  arg: User argument
11  */
12  void
13  netconn_server_lthread_thread(void* arg) {
14      lwespr_t res;
15      lwesp_netconn_p server, client;
16      lwesp_pbuf_p p;
17
18      /* Create netconn for server */
19      server = lwesp_netconn_new(LWESP_NETCONN_TYPE_TCP);
20      if (server == NULL) {
21          printf("Cannot create server netconn!\r\n");
22      }
23
24      /* Bind it to port 23 */
25      res = lwesp_netconn_bind(server, 23);
26      if (res != lwespOK) {
27          printf("Cannot bind server\r\n");
28          goto out;
29      }
30
31      /* Start listening for incoming connections with maximal 1 client */
32      res = lwesp_netconn_listen_with_max_conn(server, 1);
33      if (res != lwespOK) {
34          goto out;
35      }
36
37      /* Unlimited loop */
38      while (1) {
39          /* Accept new client */
40          res = lwesp_netconn_accept(server, &client);
41          if (res != lwespOK) {
42              break;
43          }
44          printf("New client accepted!\r\n");
45          while (1) {
46              /* Receive data */
47              res = lwesp_netconn_receive(client, &p);
48              if (res == lwespOK) {
49                  printf("Data received!\r\n");
50                  lwesp_pbuf_free(p);
51              } else {
52                  printf("Netconn receive returned: %d\r\n", (int)res);
53                  if (res == lwespCLOSED) {
54                      printf("Connection closed by client\r\n");
55                      break;
56                  }
57              }
58          }
59          /* Delete client */
60          if (client != NULL) {

```

(continues on next page)

(continued from previous page)

```

61     lwesp_netconn_delete(client);
62     client = NULL;
63 }
64 }
65 /* Delete client */
66 if (client != NULL) {
67     lwesp_netconn_delete(client);
68     client = NULL;
69 }
70
71 out:
72 printf("Terminating netconn thread!\r\n");
73 if (server != NULL) {
74     lwesp_netconn_delete(server);
75 }
76 lwesp_sys_thread_terminate(NULL);
77 }

```

Netconn server concurrency

Fig. 11: Netconn API server concurrency block diagram

When compared to classic netconn server, concurrent netconn server mode allows multiple clients to be processed at the same time. This can drastically improve performance and response time on clients side, especially when many clients are connected to server at the same time.

Every time *server application thread* (green block) gets new client to process, it starts a new *processing* thread instead of doing it in accept thread.

- Server thread is only dedicated to accept clients and start threads
- Multiple processing thread can run in parallel to send/receive data from multiple clients
- No delay when multi clients are active at the same time
- Higher memory footprint is necessary as there are multiple threads active

Listing 27: Netconn server with multiple processing threads

```

1  /*
2  * Netconn server example is based on single "user" thread
3  * which listens for new connections and accepts them.
4  *
5  * When a new client is accepted by server,
6  * separate thread for client is created where
7  * data is read, processed and send back to user
8  */
9  #include "netconn_server.h"
10 #include "lwesp/lwesp.h"
11
12 static void netconn_server_processing_thread(void* const arg);
13
14 /**
15  * \brief      Main page response file
16  */

```

(continues on next page)

```

17 static const uint8_t
18 rlwesp_data_mainpage_top[] = ""
19     "HTTP/1.1 200 OK\r\n"
20     "Content-Type: text/html\r\n"
21     "\r\n"
22     "<html>"
23     "    <head>"
24     "        <link rel=\"stylesheet\" href=\"style.css\" type=\
↳"text/css\" />"
25     "        <meta http-equiv=\"refresh\" content=\"1\" />"
26     "    </head>"
27     "    <body>"
28     "        <p>Netconn driven website!</p>"
29     "        <p>Total system up time: <b>;"
30
31 /**
32  * \brief      Bottom part of main page
33  */
34 static const uint8_t
35 rlwesp_data_mainpage_bottom[] = ""
36     "        </b></p>"
37     "    </body>"
38     "</html>";
39
40 /**
41  * \brief      Style file response
42  */
43 static const uint8_t
44 rlwesp_data_style[] = ""
45     "HTTP/1.1 200 OK\r\n"
46     "Content-Type: text/css\r\n"
47     "\r\n"
48     "body { color: red; font-family: Tahoma, Arial; };";
49
50 /**
51  * \brief      404 error response
52  */
53 static const uint8_t
54 rlwesp_error_404[] = ""
55     "HTTP/1.1 404 Not Found\r\n"
56     "\r\n"
57     "Error 404";
58
59 /**
60  * \brief      Netconn server thread implementation
61  * \param[in]  arg: User argument
62  */
63 void
64 netconn_server_thread(void const* arg) {
65     lwespr_t res;
66     lwesp_netconn_p server, client;
67
68     /*
69     * First create a new instance of netconn
70     * connection and initialize system message boxes
71     * to accept clients and packet buffers
72     */

```

(continues on next page)

(continued from previous page)

```

73 server = lwesp_netconn_new(LWESP_NETCONN_TYPE_TCP);
74 if (server != NULL) {
75     printf("Server netconn created\r\n");
76
77     /* Bind network connection to port 80 */
78     res = lwesp_netconn_bind(server, 80);
79     if (res == lwespOK) {
80         printf("Server netconn listens on port 80\r\n");
81         /*
82          * Start listening for incoming connections
83          * on previously binded port
84          */
85         res = lwesp_netconn_listen(server);
86
87         while (1) {
88             /*
89              * Wait and accept new client connection
90              *
91              * Function will block thread until
92              * new client is connected to server
93              */
94             res = lwesp_netconn_accept(server, &client);
95             if (res == lwespOK) {
96                 printf("Netconn new client connected. Starting new thread...\r\n
↪");
97
98                 /*
99                  * Start new thread for this request.
100                 *
101                 * Read and write back data to user in separated thread
102                 * to allow processing of multiple requests at the same time
103                 */
104                 if (lwesp_sys_thread_create(NULL, "client", (lwesp_sys_thread_
↪fn)netconn_server_processing_thread, client, 512, LWESP_SYS_THREAD_PRIO)) {
105                     printf("Netconn client thread created\r\n");
106                 } else {
107                     printf("Netconn client thread creation failed!\r\n");
108
109                     /* Force close & delete */
110                     lwesp_netconn_close(client);
111                     lwesp_netconn_delete(client);
112                 }
113             } else {
114                 printf("Netconn connection accept error!\r\n");
115                 break;
116             }
117         } else {
118             printf("Netconn server cannot bind to port\r\n");
119         }
120     } else {
121         printf("Cannot create server netconn\r\n");
122     }
123
124     lwesp_netconn_delete(server); /* Delete netconn structure */
125     lwesp_sys_thread_terminate(NULL); /* Terminate current thread */
126 }
127

```

(continues on next page)

```

128 /**
129  * \brief      Thread to process single active connection
130  * \param[in]  arg: Thread argument
131  */
132 static void
133 netconn_server_processing_thread(void* const arg) {
134     lwesp_netconn_p client;
135     lwesp_pbuf_p pbuf, p = NULL;
136     lwespr_t res;
137     char strt[20];
138
139     client = arg;                                /* Client handle is passed to_
↳argument */
140
141     printf("A new connection accepted!\r\n"); /* Print simple message */
142
143     do {
144         /*
145          * Client was accepted, we are now
146          * expecting client will send to us some data
147          *
148          * Wait for data and block thread for that time
149          */
150         res = lwesp_netconn_receive(client, &pbuf);
151
152         if (res == lwespOK) {
153             printf("Netconn data received, %d bytes\r\n", (int)lwesp_pbuf_length(pbuf,
↳ 1));
154             /* Check reception of all header bytes */
155             if (p == NULL) {
156                 p = pbuf;                                /* Set as first buffer */
157             } else {
158                 lwesp_pbuf_cat(p, pbuf);                /* Concatenate buffers together */
159             }
160             if (lwesp_pbuf_strfind(pbuf, "\r\n\r\n", 0) != LWESP_SIZET_MAX) {
161                 if (lwesp_pbuf_strfind(pbuf, "GET / ", 0) != LWESP_SIZET_MAX) {
162                     uint32_t now;
163                     printf("Main page request\r\n");
164                     now = lwesp_sys_now();                /* Get current time */
165                     sprintf(strt, "%u ms; %d s", (unsigned)now, (unsigned)(now /_
↳1000));
166                     lwesp_netconn_write(client, rlwesp_data_mainpage_top,
↳sizeof(rlwesp_data_mainpage_top) - 1);
167                     lwesp_netconn_write(client, strt, strlen(strt));
168                     lwesp_netconn_write(client, rlwesp_data_mainpage_bottom,
↳sizeof(rlwesp_data_mainpage_bottom) - 1);
169                 } else if (lwesp_pbuf_strfind(pbuf, "GET /style.css ", 0) != LWESP_
↳SIZET_MAX) {
170                     printf("Style page request\r\n");
171                     lwesp_netconn_write(client, rlwesp_data_style, sizeof(rlwesp_data_
↳style) - 1);
172                 } else {
173                     printf("404 error not found\r\n");
174                     lwesp_netconn_write(client, rlwesp_error_404, sizeof(rlwesp_error_
↳404) - 1);
175                 }
176                 lwesp_netconn_close(client);            /* Close netconn connection */

```

(continues on next page)

(continued from previous page)

```

177         lwesp_pbuf_free(p);           /* Do not forget to free memory_
↳after usage! */
178         p = NULL;
179         break;
180     }
181 }
182 } while (res == lwespOK);
183
184 if (p != NULL) {                       /* Free received data */
185     lwesp_pbuf_free(p);
186     p = NULL;
187 }
188 lwesp_netconn_delete(client);          /* Destroy client memory */
189 lwesp_sys_thread_terminate(NULL);     /* Terminate this thread */
190 }

```

Non-blocking receive

By default, netconn API is written to only work in separate application thread, dedicated for network connection processing. Because of that, by default every function is fully blocking. It will wait until result is ready to be used by application.

It is, however, possible to enable timeout feature for receiving data only. When this feature is enabled, `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

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

TCP connection

enumerator LWESP_NETCONN_TYPE_SSL

SSL connection

enumerator LWESP_NETCONN_TYPE_UDP

UDP connection

Functions

lwesp_netconn_p **lwesp_netconn_new** (*lwesp_netconn_type_t* type)

Create new netconn connection.

Return New netconn connection on success, NULL otherwise

Parameters

- [in] type: Netconn connection type

lwespr_t **lwesp_netconn_delete** (*lwesp_netconn_p* nc)

Delete netconn connection.

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

Parameters

- [in] nc: Netconn handle

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.

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

Parameters

- [in] nc: Netconn handle
- [in] port: Port used to bind a connection to

lwespr_t **lwesp_netconn_connect** (*lwesp_netconn_p* nc, **const** char *host, *lwesp_port_t* port)

Connect to server as client.

Return *lwespOK* if successfully connected, member of *lwespr_t* otherwise

Parameters

- [in] nc: Netconn handle
- [in] host: Pointer to host, such as domain name or IP address in string format
- [in] port: Target port to use

lwespr_t **lwesp_netconn_receive** (*lwesp_netconn_p nc, lwesp_pbuf_p *pbuf*)

Receive data from connection.

Return *lwespOK* when new data ready

Return *lwespCLOSED* when connection closed by remote side

Return *lwespTIMEOUT* when receive timeout occurs

Return Any other member of *lwespr_t* otherwise

Parameters

- [in] *nc*: Netconn handle used to receive from
- [in] *pbuf*: Pointer to pointer to save new receive buffer to. When function returns, user must check for valid *pbuf* value *pbuf != NULL*

lwespr_t **lwesp_netconn_close** (*lwesp_netconn_p nc*)

Close a netconn connection.

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

Parameters

- [in] *nc*: Netconn handle to close

int8_t **lwesp_netconn_get_connum** (*lwesp_netconn_p nc*)

Get connection number used for netconn.

Return -1 on failure, connection number between 0 and *LWESP_CFG_MAX_CONNS* otherwise

Parameters

- [in] *nc*: Netconn handle

lwesp_conn_p **lwesp_netconn_get_conn** (*lwesp_netconn_p nc*)

Get netconn connection handle.

Return ESP connection handle

Parameters

- [in] *nc*: Netconn handle

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

- [in] *nc*: Netconn handle
- [in] *timeout*: Timeout in units of milliseconds. Set to 0 to disable timeout feature Set to > 0 to set maximum milliseconds to wait before timeout Set to *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.

Return Timeout in units of milliseconds. If value is 0, timeout is disabled (wait forever)

Parameters

- [in] nc: Netconn handle

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.

Return *lwespOK* if successfully connected, member of *lwespr_t* otherwise

Parameters

- [in] nc: Netconn handle
- [in] host: Pointer to host, such as domain name or IP address in string format
- [in] port: Target port to use
- [in] keep_alive: Keep alive period seconds
- [in] local_ip: Local ip in connected command
- [in] local_port: Local port address
- [in] mode: UDP mode

lwespr_t **lwesp_netconn_listen** (*lwesp_netconn_p* nc)

Listen on previously binded connection.

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

Parameters

- [in] nc: Netconn handle used to listen for new connections

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.

Return *lwespOK* on success, member of *lwespr_t* otherwise

Parameters

- [in] nc: Netconn handle used to listen for new connections
- [in] max_connections: Maximal number of connections server can accept at a time This parameter may not be larger than *LWESP_CFG_MAX_CONNS*

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*

Return *lwespOK* on success, member of *lwespr_t* otherwise

Parameters

- [in] nc: Netconn handle used for listen mode
- [in] timeout: Time in units of seconds. Set to 0 to disable timeout feature

lwespr_t **lwesp_netconn_accept** (*lwesp_netconn_p* nc, *lwesp_netconn_p* *client)
Accept a new connection.

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

Parameters

- [in] nc: Netconn handle used as base connection to accept new clients
- [out] client: Pointer to netconn handle to save new connection to

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

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

Parameters

- [in] nc: Netconn handle used to write data to
- [in] data: Pointer to data to write
- [in] btw: Number of bytes to write

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

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

Parameters

- [in] nc: Netconn handle to flush data

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.

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

Parameters

- [in] nc: Netconn handle used to send
- [in] data: Pointer to data to write
- [in] btw: Number of bytes to write

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

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

Parameters

- [in] nc: Netconn handle used to send
- [in] ip: Pointer to IP address

- [in] port: Port number used to send data
- [in] data: Pointer to data to write
- [in] btw: Number of bytes to write

5.3.5 Command line interface

CLI Input module

group **CLI_INPUT**

Command line interface helper functions for paring input data.

Functions to parse incoming data for command line interface (CLI).

Functions

void **cli_in_data** (*cli_printf* cliprintf, char ch)
parse new characters to the CLI

Parameters

- [in] cliprintf: Pointer to CLI printf function
- [in] ch: new character to CLI

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.

Parameters

- [in] *format*: string format

typedef void **cli_function** (*cli_printf* *cliprintf*, int *argc*, char ***argv*)
CLI entry function.

Parameters

- [in] *cliprintf*: Printf handle callback
- [in] *argc*: Number of arguments
- [in] *argv*: Pointer to pointer to arguments

Functions

const *cli_command_t* ***cli_lookup_command** (char **command*)
Find the CLI command that matches the input string.

Return pointer of the command if we found a match, else NULL

Parameters

- [in] *command*: pointer to command string for which we are searching

void **cli_tab_auto_complete** (*cli_printf* *cliprintf*, char **cmd_buffer*, uint32_t **cmd_pos*, bool
print_options)
CLI auto completion function.

Parameters

- [in] *cliprintf*: Pointer to CLI printf function
- [in] *cmd_buffer*: CLI command buffer
- [in] *cmd_pos*: pointer to current cursor position in command buffer
- [in] *print_options*: additional prints in case of double tab

bool **cli_register_commands** (const *cli_command_t* **commands*, size_t *num_of_commands*)
Register new CLI commands.

Return true when new commands were successfully added, else false

Parameters

- [in] *commands*: Pointer to commands table
- [in] *num_of_commands*: Number of new commands

void **cli_init** (void)
CLI Init function for adding basic CLI commands.

```
struct cli_command_t  
    #include <cli.h> CLI command structure.
```

Public Members

```
const char *name  
    Command name
```

```
const char *help  
    Command help
```

```
cli_function *func  
    Command function
```

```
struct cli_commands_t  
    #include <cli.h> List of commands.
```

Public Members

```
const cli_command_t *commands  
    Pointer to commands
```

```
size_t num_of_commands  
    Total number of commands
```

5.4 Examples and demos

Various examples are provided for fast library evaluation on embedded systems. These are prepared and maintained for 2 platforms, but could be easily extended to more platforms:

- WIN32 examples, prepared as [Visual Studio Community](#) projects
- ARM Cortex-M examples for STM32, prepared as [STM32CubeIDE](#) GCC projects

Warning: Library is platform independent and can be used on any platform.
--

5.4.1 Example architectures

There are many platforms available today on a market, however supporting them all would be tough task for single person. Therefore it has been decided to support (for purpose of examples) 2 platforms only, *WIN32* and *STM32*.

WIN32

Examples for *WIN32* are prepared as [Visual Studio Community](#) projects. You can directly open project in the IDE, compile & debug.

Application opens *COM* port, set in the low-level driver. External USB to UART converter (FTDI-like device) is necessary in order to connect to *ESP* device.

Note: *ESP* device is connected with *USB to UART converter* only by *RX* and *TX* pins.

Device driver is located in `/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
STM32F745 Discovery	UART5	PC12	PD2	PJ14	.	.	.	USART1	PA9	PA10
STM32F713 Discovery	UART5	PC12	PD2	PG14	.	PD6	PD3	USART6	PC6	PC7
STM32L466 Discovery	UART1	PB6	PG10	PB2	PH2	PA0	PA4	USART2	PA2	PD6
STM32L412 Nucleo	UART1	PA9	PA10	PA12	PA7	PA6	PB0	USART2	PA2	PA3
STM32F412 Nucleo	UART2	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

B

BUF_PREF (*C macro*), 72

C

CLI_CMD_HISTORY (*C macro*), 212

cli_command_t (*C++ struct*), 213

cli_command_t::func (*C++ member*), 214

cli_command_t::help (*C++ member*), 214

cli_command_t::name (*C++ member*), 214

cli_commands_t (*C++ struct*), 214

cli_commands_t::commands (*C++ member*), 214

cli_commands_t::num_of_commands (*C++ member*), 214

cli_function (*C++ type*), 213

cli_in_data (*C++ function*), 212

cli_init (*C++ function*), 213

cli_lookup_command (*C++ function*), 213

CLI_MAX_CMD_LENGTH (*C macro*), 212

CLI_MAX_MODULES (*C macro*), 212

CLI_MAX_NUM_OF_ARGS (*C macro*), 212

CLI_NL (*C macro*), 212

cli_printf (*C++ type*), 213

CLI_PROMPT (*C macro*), 212

cli_register_commands (*C++ function*), 213

cli_tab_auto_complete (*C++ function*), 213

H

http_cgi_fn (*C++ type*), 173

http_cgi_t (*C++ struct*), 176

http_cgi_t::fn (*C++ member*), 176

http_cgi_t::uri (*C++ member*), 176

HTTP_DYNAMIC_HEADERS (*C macro*), 161

HTTP_DYNAMIC_HEADERS_CONTENT_LEN (*C macro*), 161

http_fs_close (*C++ function*), 179

http_fs_close_fn (*C++ type*), 175

http_fs_file_t (*C++ struct*), 177

http_fs_file_t::arg (*C++ member*), 177

http_fs_file_t::data (*C++ member*), 177

http_fs_file_t::fptr (*C++ member*), 177

http_fs_file_t::is_static (*C++ member*), 177

http_fs_file_t::rem_open_files (*C++ member*), 177

http_fs_file_t::size (*C++ member*), 177

http_fs_file_table_t (*C++ struct*), 177

http_fs_file_table_t::data (*C++ member*), 177

http_fs_file_table_t::path (*C++ member*), 177

http_fs_file_table_t::size (*C++ member*), 177

http_fs_open (*C++ function*), 179

http_fs_open_fn (*C++ type*), 174

http_fs_read (*C++ function*), 179

http_fs_read_fn (*C++ type*), 174

http_init_t (*C++ struct*), 176

http_init_t::cgi (*C++ member*), 176

http_init_t::cgi_count (*C++ member*), 176

http_init_t::fs_close (*C++ member*), 177

http_init_t::fs_open (*C++ member*), 176

http_init_t::fs_read (*C++ member*), 177

http_init_t::post_data_fn (*C++ member*), 176

http_init_t::post_end_fn (*C++ member*), 176

http_init_t::post_start_fn (*C++ member*), 176

http_init_t::ssi_fn (*C++ member*), 176

HTTP_MAX_HEADERS (*C macro*), 173

HTTP_MAX_PARAMS (*C macro*), 160

HTTP_MAX_URI_LEN (*C macro*), 160

http_param_t (*C++ struct*), 176

http_param_t::name (*C++ member*), 176

http_param_t::value (*C++ member*), 176

http_post_data_fn (*C++ type*), 173

http_post_end_fn (*C++ type*), 174

http_post_start_fn (*C++ type*), 173

http_req_method_t (*C++ enum*), 175

http_req_method_t::HTTP_METHOD_GET (*C++ enumerator*), 175

http_req_method_t::HTTP_METHOD_NOTALLOWED (*C++ enumerator*), 175

http_req_method_t::HTTP_METHOD_POST (*C++ enumerator*), 175

- HTTP_SERVER_NAME (*C macro*), 161
 http_ssi_fn (*C++ type*), 174
 http_ssi_state_t (*C++ enum*), 175
 http_ssi_state_t::HTTP_SSI_STATE_BEGIN (*C++ enumerator*), 175
 http_ssi_state_t::HTTP_SSI_STATE_END (*C++ enumerator*), 175
 http_ssi_state_t::HTTP_SSI_STATE_TAG (*C++ enumerator*), 175
 http_ssi_state_t::HTTP_SSI_STATE_WAIT_BEGIN (*C++ enumerator*), 175
 HTTP_SSI_TAG_END (*C macro*), 160
 HTTP_SSI_TAG_END_LEN (*C macro*), 160
 HTTP_SSI_TAG_MAX_LEN (*C macro*), 160
 HTTP_SSI_TAG_START (*C macro*), 160
 HTTP_SSI_TAG_START_LEN (*C macro*), 160
 http_state_t (*C++ struct*), 177
 http_state_t::arg (*C++ member*), 178
 http_state_t::buff (*C++ member*), 178
 http_state_t::buff_len (*C++ member*), 178
 http_state_t::buff_ptr (*C++ member*), 178
 http_state_t::conn (*C++ member*), 178
 http_state_t::conn_mem_available (*C++ member*), 178
 http_state_t::content_length (*C++ member*), 178
 http_state_t::content_received (*C++ member*), 178
 http_state_t::dyn_hdr_cnt_len (*C++ member*), 178
 http_state_t::dyn_hdr_idx (*C++ member*), 178
 http_state_t::dyn_hdr_pos (*C++ member*), 178
 http_state_t::dyn_hdr_strs (*C++ member*), 178
 http_state_t::headers_received (*C++ member*), 178
 http_state_t::is_ssi (*C++ member*), 178
 http_state_t::p (*C++ member*), 178
 http_state_t::process_resp (*C++ member*), 178
 http_state_t::req_method (*C++ member*), 178
 http_state_t::rlwesp_file (*C++ member*), 178
 http_state_t::rlwesp_file_opened (*C++ member*), 178
 http_state_t::sent_total (*C++ member*), 178
 http_state_t::ssi_state (*C++ member*), 178
 http_state_t::ssi_tag_buff (*C++ member*), 179
 http_state_t::ssi_tag_buff_ptr (*C++ member*), 179
 http_state_t::ssi_tag_buff_written (*C++ member*), 179
 http_state_t::ssi_tag_len (*C++ member*), 179
 http_state_t::ssi_tag_process_more (*C++ member*), 179
 http_state_t::written_total (*C++ member*), 178
 HTTP_SUPPORT_POST (*C macro*), 160
 HTTP_USE_DEFAULT_STATIC_FILES (*C macro*), 160
 HTTP_USE_METHOD_NOTALLOWED_RESP (*C macro*), 160
- ## L
- lwesp_ap_conf_t (*C++ struct*), 71
 lwesp_ap_conf_t::ch (*C++ member*), 71
 lwesp_ap_conf_t::ecn (*C++ member*), 71
 lwesp_ap_conf_t::hidden (*C++ member*), 71
 lwesp_ap_conf_t::max_cons (*C++ member*), 71
 lwesp_ap_conf_t::pwd (*C++ member*), 71
 lwesp_ap_conf_t::ssid (*C++ member*), 71
 lwesp_ap_configure (*C++ function*), 69
 lwesp_ap_disconn_sta (*C++ function*), 70
 lwesp_ap_get_config (*C++ function*), 69
 lwesp_ap_getip (*C++ function*), 68
 lwesp_ap_getmac (*C++ function*), 68
 lwesp_ap_list_sta (*C++ function*), 70
 lwesp_ap_setip (*C++ function*), 68
 lwesp_ap_setmac (*C++ function*), 68
 lwesp_ap_t (*C++ struct*), 70
 lwesp_ap_t::bgn (*C++ member*), 71
 lwesp_ap_t::ch (*C++ member*), 71
 lwesp_ap_t::ecn (*C++ member*), 71
 lwesp_ap_t::mac (*C++ member*), 71
 lwesp_ap_t::rssi (*C++ member*), 71
 lwesp_ap_t::ssid (*C++ member*), 71
 lwesp_api_cmd_evt_fn (*C++ type*), 125
 LWESP_ARRAYSIZE (*C macro*), 144
 LWESP_ASSERT (*C macro*), 144
 lwesp_buff_advance (*C++ function*), 74
 lwesp_buff_free (*C++ function*), 72
 lwesp_buff_get_free (*C++ function*), 73
 lwesp_buff_get_full (*C++ function*), 73
 lwesp_buff_get_linear_block_read_address (*C++ function*), 73
 lwesp_buff_get_linear_block_read_length (*C++ function*), 73
 lwesp_buff_get_linear_block_write_address (*C++ function*), 74
 lwesp_buff_get_linear_block_write_length (*C++ function*), 74
 lwesp_buff_init (*C++ function*), 72
 lwesp_buff_peek (*C++ function*), 73
 lwesp_buff_read (*C++ function*), 72

- lwesp_buff_reset (C++ function), 72
 lwesp_buff_skip (C++ function), 74
 lwesp_buff_t (C++ struct), 74
 lwesp_buff_t::buff (C++ member), 75
 lwesp_buff_t::r (C++ member), 75
 lwesp_buff_t::size (C++ member), 75
 lwesp_buff_t::w (C++ member), 75
 lwesp_buff_write (C++ function), 72
 LWESP_CAYENNE_ALL_CHANNELS (C macro), 169
 LWESP_CAYENNE_API_VERSION (C macro), 169
 lwesp_cayenne_create (C++ function), 170
 lwesp_cayenne_evt_fn (C++ type), 169
 lwesp_cayenne_evt_t (C++ struct), 172
 lwesp_cayenne_evt_t::data (C++ member), 172
 lwesp_cayenne_evt_t::evt (C++ member), 172
 lwesp_cayenne_evt_t::msg (C++ member), 172
 lwesp_cayenne_evt_t::type (C++ member), 172
 lwesp_cayenne_evt_type_t (C++ enum), 170
 lwesp_cayenne_evt_type_t::LWESP_CAYENNE_EVT_CONNECTED (C++ enumerator), 170
 lwesp_cayenne_evt_type_t::LWESP_CAYENNE_EVT_DATA (C++ enumerator), 169
 lwesp_cayenne_evt_type_t::LWESP_CAYENNE_EVT_DISCONNECTED (C++ enumerator), 170
 LWESP_CAYENNE_HOST (C macro), 169
 lwesp_cayenne_key_value_t (C++ struct), 171
 lwesp_cayenne_key_value_t::key (C++ member), 171
 lwesp_cayenne_key_value_t::value (C++ member), 171
 lwesp_cayenne_msg_t (C++ struct), 171
 lwesp_cayenne_msg_t::channel (C++ member), 172
 lwesp_cayenne_msg_t::seq (C++ member), 172
 lwesp_cayenne_msg_t::topic (C++ member), 172
 lwesp_cayenne_msg_t::values (C++ member), 172
 lwesp_cayenne_msg_t::values_count (C++ member), 172
 LWESP_CAYENNE_NO_CHANNEL (C macro), 169
 LWESP_CAYENNE_PORT (C macro), 169
 lwesp_cayenne_publish_data (C++ function), 171
 lwesp_cayenne_publish_float (C++ function), 171
 lwesp_cayenne_publish_response (C++ function), 171
 lwesp_cayenne_rlwsesp_t (C++ enum), 170
 lwesp_cayenne_rlwsesp_t::LWESP_CAYENNE_RLWSESP_ERROR (C++ enumerator), 170
 lwesp_cayenne_rlwsesp_t::LWESP_CAYENNE_RLWSESP_OK (C++ enumerator), 170
 (C++ enumerator), 170
 lwesp_cayenne_subscribe (C++ function), 171
 lwesp_cayenne_t (C++ struct), 172
 lwesp_cayenne_t::api_c (C++ member), 172
 lwesp_cayenne_t::evt (C++ member), 172
 lwesp_cayenne_t::evt_fn (C++ member), 172
 lwesp_cayenne_t::info_c (C++ member), 172
 lwesp_cayenne_t::msg (C++ member), 172
 lwesp_cayenne_t::sem (C++ member), 172
 lwesp_cayenne_t::thread (C++ member), 172
 lwesp_cayenne_topic_t (C++ enum), 169
 lwesp_cayenne_topic_t::LWESP_CAYENNE_TOPIC_ANALOG (C++ enumerator), 170
 lwesp_cayenne_topic_t::LWESP_CAYENNE_TOPIC_ANALOG_CONFIG (C++ enumerator), 170
 lwesp_cayenne_topic_t::LWESP_CAYENNE_TOPIC_COMMAND (C++ enumerator), 169
 lwesp_cayenne_topic_t::LWESP_CAYENNE_TOPIC_CONFIG (C++ enumerator), 169
 lwesp_cayenne_topic_t::LWESP_CAYENNE_TOPIC_DATA (C++ enumerator), 169
 lwesp_cayenne_topic_t::LWESP_CAYENNE_TOPIC_DIGITAL (C++ enumerator), 170
 lwesp_cayenne_topic_t::LWESP_CAYENNE_TOPIC_DIGITAL_CONFIG (C++ enumerator), 170
 lwesp_cayenne_topic_t::LWESP_CAYENNE_TOPIC_DIGITAL_DATA (C++ enumerator), 170
 lwesp_cayenne_topic_t::LWESP_CAYENNE_TOPIC_DIGITAL_END (C++ enumerator), 170
 lwesp_cayenne_topic_t::LWESP_CAYENNE_TOPIC_DIGITAL_RESPONSE (C++ enumerator), 169
 lwesp_cayenne_topic_t::LWESP_CAYENNE_TOPIC_SYS_CPU_LOAD (C++ enumerator), 170
 lwesp_cayenne_topic_t::LWESP_CAYENNE_TOPIC_SYS_CPU_LOAD_CONFIG (C++ enumerator), 170
 lwesp_cayenne_topic_t::LWESP_CAYENNE_TOPIC_SYS_MODEM (C++ enumerator), 170
 lwesp_cayenne_topic_t::LWESP_CAYENNE_TOPIC_SYS_VERSION (C++ enumerator), 170
 LWESP_CFG_AT_ECHO (C macro), 157
 LWESP_CFG_AT_PORT_BAUDRATE (C macro), 154
 LWESP_CFG_CONN_MANUAL_TCP_RECEIVE (C macro), 155
 LWESP_CFG_CONN_MAX_DATA_LEN (C macro), 154
 LWESP_CFG_CONN_MAX_RECV_BUFF_SIZE (C macro), 154
 LWESP_CFG_CONN_POLL_INTERVAL (C macro), 155
 LWESP_CFG_DBG (C macro), 156
 LWESP_CFG_DBG_ASSERT (C macro), 156
 LWESP_CFG_DBG_CAYENNE (C macro), 160
 LWESP_CFG_DBG_CONN (C macro), 157
 LWESP_CFG_DBG_INIT (C macro), 156

- LWESP_CFG_DBG_INPUT (*C macro*), 156
- LWESP_CFG_DBG_IPD (*C macro*), 156
- LWESP_CFG_DBG_LVL_MIN (*C macro*), 156
- LWESP_CFG_DBG_MEM (*C macro*), 156
- LWESP_CFG_DBG_MQTT (*C macro*), 159
- LWESP_CFG_DBG_MQTT_API (*C macro*), 159
- LWESP_CFG_DBG_NETCONN (*C macro*), 156
- LWESP_CFG_DBG_OUT (*C macro*), 156
- LWESP_CFG_DBG_PBUF (*C macro*), 156
- LWESP_CFG_DBG_SERVER (*C macro*), 160
- LWESP_CFG_DBG_THREAD (*C macro*), 156
- LWESP_CFG_DBG_TYPES_ON (*C macro*), 156
- LWESP_CFG_DBG_VAR (*C macro*), 157
- LWESP_CFG_DNS (*C macro*), 158
- LWESP_CFG_ESP32 (*C macro*), 153
- LWESP_CFG_ESP8266 (*C macro*), 153
- LWESP_CFG_HOSTNAME (*C macro*), 158
- LWESP_CFG_INPUT_USE_PROCESS (*C macro*), 157
- LWESP_CFG_MAX_CONNS (*C macro*), 153
- LWESP_CFG_MAX_PWD_LENGTH (*C macro*), 155
- LWESP_CFG_MAX_SEND_RETRIES (*C macro*), 154
- LWESP_CFG_MAX_SSID_LENGTH (*C macro*), 155
- LWESP_CFG_MDNS (*C macro*), 159
- LWESP_CFG_MEM_ALIGNMENT (*C macro*), 153
- LWESP_CFG_MEM_CUSTOM (*C macro*), 153
- LWESP_CFG_MODE_ACCESS_POINT (*C macro*), 154
- LWESP_CFG_MODE_STATION (*C macro*), 154
- LWESP_CFG_MQTT_MAX_REQUESTS (*C macro*), 159
- LWESP_CFG_NETCONN (*C macro*), 159
- LWESP_CFG_NETCONN_ACCEPT_QUEUE_LEN (*C macro*), 159
- LWESP_CFG_NETCONN_RECEIVE_QUEUE_LEN (*C macro*), 159
- LWESP_CFG_NETCONN_RECEIVE_TIMEOUT (*C macro*), 159
- LWESP_CFG_OS (*C macro*), 153
- LWESP_CFG_PING (*C macro*), 159
- LWESP_CFG_RCV_BUFF_SIZE (*C macro*), 154
- LWESP_CFG_RESET_DELAY_DEFAULT (*C macro*), 155
- LWESP_CFG_RESET_ON_DEVICE_PRESENT (*C macro*), 155
- LWESP_CFG_RESET_ON_INIT (*C macro*), 154
- LWESP_CFG_RESTORE_ON_INIT (*C macro*), 154
- LWESP_CFG_SMART (*C macro*), 159
- LWESP_CFG_SNTP (*C macro*), 158
- LWESP_CFG_THREAD_PROCESS_MBOX_SIZE (*C macro*), 157
- LWESP_CFG_THREAD_PRODUCER_MBOX_SIZE (*C macro*), 157
- LWESP_CFG_USE_API_FUNC_EVT (*C macro*), 153
- LWESP_CFG_WPS (*C macro*), 158
- lwesp_cmd_t (*C++ enum*), 125
- lwesp_cmd_t::LWESP_CMD_ATE0 (*C++ enumerator*), 125
- lwesp_cmd_t::LWESP_CMD_ATE1 (*C++ enumerator*), 125
- lwesp_cmd_t::LWESP_CMD_BLEINIT_GET (*C++ enumerator*), 128
- lwesp_cmd_t::LWESP_CMD_GMR (*C++ enumerator*), 125
- lwesp_cmd_t::LWESP_CMD_GSLP (*C++ enumerator*), 125
- lwesp_cmd_t::LWESP_CMD_IDLE (*C++ enumerator*), 125
- lwesp_cmd_t::LWESP_CMD_RESET (*C++ enumerator*), 125
- lwesp_cmd_t::LWESP_CMD_RESTORE (*C++ enumerator*), 126
- lwesp_cmd_t::LWESP_CMD_RFAUTOTRACE (*C++ enumerator*), 126
- lwesp_cmd_t::LWESP_CMD_RFPOWER (*C++ enumerator*), 126
- lwesp_cmd_t::LWESP_CMD_RFVDD (*C++ enumerator*), 126
- lwesp_cmd_t::LWESP_CMD_SLEEP (*C++ enumerator*), 126
- lwesp_cmd_t::LWESP_CMD_SYSADC (*C++ enumerator*), 126
- lwesp_cmd_t::LWESP_CMD_SYSLOG (*C++ enumerator*), 126
- lwesp_cmd_t::LWESP_CMD_SYSMMSG (*C++ enumerator*), 126
- lwesp_cmd_t::LWESP_CMD_SYSRAM (*C++ enumerator*), 126
- lwesp_cmd_t::LWESP_CMD_TCPIP_CIFSR (*C++ enumerator*), 128
- lwesp_cmd_t::LWESP_CMD_TCPIP_CIPCLOSE (*C++ enumerator*), 127
- lwesp_cmd_t::LWESP_CMD_TCPIP_CIPDINFO (*C++ enumerator*), 128
- lwesp_cmd_t::LWESP_CMD_TCPIP_CIPDNS_GET (*C++ enumerator*), 127
- lwesp_cmd_t::LWESP_CMD_TCPIP_CIPDNS_SET (*C++ enumerator*), 127
- lwesp_cmd_t::LWESP_CMD_TCPIP_CIPDOMAIN (*C++ enumerator*), 127
- lwesp_cmd_t::LWESP_CMD_TCPIP_CIPMODE (*C++ enumerator*), 128
- lwesp_cmd_t::LWESP_CMD_TCPIP_CIPMUX (*C++ enumerator*), 128
- lwesp_cmd_t::LWESP_CMD_TCPIP_CIPRECVDATA (*C++ enumerator*), 128
- lwesp_cmd_t::LWESP_CMD_TCPIP_CIPRECVLEN (*C++ enumerator*), 128
- lwesp_cmd_t::LWESP_CMD_TCPIP_CIPRECVMODE (*C++ enumerator*), 128

<code>lwesp_cmd_t::LWESP_CMD_TCPIP_CIPSEND</code> (C++ enumerator), 127	<code>lwesp_cmd_t::LWESP_CMD_WIFI_CWHOSTNAME_GET</code> (C++ enumerator), 127
<code>lwesp_cmd_t::LWESP_CMD_TCPIP_CIPSERVER</code> (C++ enumerator), 128	<code>lwesp_cmd_t::LWESP_CMD_WIFI_CWHOSTNAME_SET</code> (C++ enumerator), 127
<code>lwesp_cmd_t::LWESP_CMD_TCPIP_CIPSERVERMAXCONN</code> (C++ enumerator), 128	<code>lwesp_cmd_t::LWESP_CMD_WIFI_CWJAP</code> (C++ enumerator), 126
<code>lwesp_cmd_t::LWESP_CMD_TCPIP_CIPSNTPCFG</code> (C++ enumerator), 128	<code>lwesp_cmd_t::LWESP_CMD_WIFI_CWJAP_GET</code> (C++ enumerator), 126
<code>lwesp_cmd_t::LWESP_CMD_TCPIP_CIPSNTPTIME</code> (C++ enumerator), 128	<code>lwesp_cmd_t::LWESP_CMD_WIFI_CWLAP</code> (C++ enumerator), 126
<code>lwesp_cmd_t::LWESP_CMD_TCPIP_CIPSSLCCONF</code> (C++ enumerator), 128	<code>lwesp_cmd_t::LWESP_CMD_WIFI_CWLAPOPT</code> (C++ enumerator), 126
<code>lwesp_cmd_t::LWESP_CMD_TCPIP_CIPSSLSIZE</code> (C++ enumerator), 128	<code>lwesp_cmd_t::LWESP_CMD_WIFI_CWLIF</code> (C++ enumerator), 127
<code>lwesp_cmd_t::LWESP_CMD_TCPIP_CIPSTART</code> (C++ enumerator), 127	<code>lwesp_cmd_t::LWESP_CMD_WIFI_CWMODE</code> (C++ enumerator), 126
<code>lwesp_cmd_t::LWESP_CMD_TCPIP_CIPSTATUS</code> (C++ enumerator), 127	<code>lwesp_cmd_t::LWESP_CMD_WIFI_CWMODE_GET</code> (C++ enumerator), 126
<code>lwesp_cmd_t::LWESP_CMD_TCPIP_CIPSTO</code> (C++ enumerator), 128	<code>lwesp_cmd_t::LWESP_CMD_WIFI_CWQAP</code> (C++ enumerator), 126
<code>lwesp_cmd_t::LWESP_CMD_TCPIP_CIUUPDATE</code> (C++ enumerator), 128	<code>lwesp_cmd_t::LWESP_CMD_WIFI_CWQIF</code> (C++ enumerator), 127
<code>lwesp_cmd_t::LWESP_CMD_TCPIP_PING</code> (C++ enumerator), 128	<code>lwesp_cmd_t::LWESP_CMD_WIFI_CWSAP_GET</code> (C++ enumerator), 127
<code>lwesp_cmd_t::LWESP_CMD_UART</code> (C++ <i>enumera-</i> <i>tor</i>), 126	<code>lwesp_cmd_t::LWESP_CMD_WIFI_CWSAP_SET</code> (C++ enumerator), 127
<code>lwesp_cmd_t::LWESP_CMD_WAKEUPGPIO</code> (C++ enumerator), 126	<code>lwesp_cmd_t::LWESP_CMD_WIFI_MDNS</code> (C++ enumerator), 127
<code>lwesp_cmd_t::LWESP_CMD_WIFI_CIPAP_GET</code> (C++ enumerator), 127	<code>lwesp_cmd_t::LWESP_CMD_WIFI_SMART_START</code> (C++ enumerator), 128
<code>lwesp_cmd_t::LWESP_CMD_WIFI_CIPAP_SET</code> (C++ enumerator), 127	<code>lwesp_cmd_t::LWESP_CMD_WIFI_SMART_STOP</code> (C++ enumerator), 128
<code>lwesp_cmd_t::LWESP_CMD_WIFI_CIPAPMAC_GET</code> (C++ enumerator), 127	<code>lwesp_cmd_t::LWESP_CMD_WIFI_WPS</code> (C++ enumerator), 127
<code>lwesp_cmd_t::LWESP_CMD_WIFI_CIPAPMAC_SET</code> (C++ enumerator), 127	<code>lwesp_conn_close</code> (C++ <i>function</i>), 80
<code>lwesp_cmd_t::LWESP_CMD_WIFI_CIPSTA_GET</code> (C++ enumerator), 126	<code>lwesp_conn_get_arg</code> (C++ <i>function</i>), 81
<code>lwesp_cmd_t::LWESP_CMD_WIFI_CIPSTA_SET</code> (C++ enumerator), 126	<code>lwesp_conn_get_from_evt</code> (C++ <i>function</i>), 82
<code>lwesp_cmd_t::LWESP_CMD_WIFI_CIPSTAMAC_GET</code> (C++ enumerator), 126	<code>lwesp_conn_get_local_port</code> (C++ <i>function</i>), 83
<code>lwesp_cmd_t::LWESP_CMD_WIFI_CIPSTAMAC_SET</code> (C++ enumerator), 126	<code>lwesp_conn_get_remote_ip</code> (C++ <i>function</i>), 83
<code>lwesp_cmd_t::LWESP_CMD_WIFI_CWAUTOCONN</code> (C++ enumerator), 126	<code>lwesp_conn_get_remote_port</code> (C++ <i>function</i>), 83
<code>lwesp_cmd_t::LWESP_CMD_WIFI_CWDHCP_GET</code> (C++ enumerator), 126	<code>lwesp_conn_get_total_recved_count</code> (C++ <i>function</i>), 83
<code>lwesp_cmd_t::LWESP_CMD_WIFI_CWDHCP_SET</code> (C++ enumerator), 126	<code>lwesp_conn_getnum</code> (C++ <i>function</i>), 82
<code>lwesp_cmd_t::LWESP_CMD_WIFI_CWDHCP_GET</code> (C++ enumerator), 126	<code>lwesp_conn_is_active</code> (C++ <i>function</i>), 81
<code>lwesp_cmd_t::LWESP_CMD_WIFI_CWDHCP_SET</code> (C++ enumerator), 126	<code>lwesp_conn_is_client</code> (C++ <i>function</i>), 81
<code>lwesp_cmd_t::LWESP_CMD_WIFI_CWDHCP_GET</code> (C++ enumerator), 127	<code>lwesp_conn_is_closed</code> (C++ <i>function</i>), 81
<code>lwesp_cmd_t::LWESP_CMD_WIFI_CWDHCP_SET</code> (C++ enumerator), 127	<code>lwesp_conn_is_server</code> (C++ <i>function</i>), 81
<code>lwesp_cmd_t::LWESP_CMD_WIFI_CWDHCP_GET</code> (C++ enumerator), 127	<code>lwesp_conn_p</code> (C++ <i>type</i>), 79
<code>lwesp_cmd_t::LWESP_CMD_WIFI_CWDHCP_SET</code> (C++ enumerator), 127	<code>lwesp_conn_recved</code> (C++ <i>function</i>), 83
	<code>lwesp_conn_send</code> (C++ <i>function</i>), 80
	<code>lwesp_conn_sendto</code> (C++ <i>function</i>), 80
	<code>lwesp_conn_set_arg</code> (C++ <i>function</i>), 81
	<code>lwesp_conn_set_ssl_buffersize</code> (C++ <i>func-</i>

- tion), 82
- lwesp_conn_ssl_configure (C++ function), 84
- lwesp_conn_start (C++ function), 79
- lwesp_conn_start_t (C++ struct), 84
- lwesp_conn_start_t::ext (C++ member), 85
- lwesp_conn_start_t::keep_alive (C++ member), 84
- lwesp_conn_start_t::local_ip (C++ member), 84
- lwesp_conn_start_t::local_port (C++ member), 84
- lwesp_conn_start_t::mode (C++ member), 84
- lwesp_conn_start_t::remote_host (C++ member), 84
- lwesp_conn_start_t::remote_port (C++ member), 84
- lwesp_conn_start_t::tcp_ssl (C++ member), 84
- lwesp_conn_start_t::type (C++ member), 84
- lwesp_conn_start_t::udp (C++ member), 84
- lwesp_conn_startex (C++ function), 79
- lwesp_conn_t (C++ struct), 131
- lwesp_conn_t::active (C++ member), 131
- lwesp_conn_t::arg (C++ member), 131
- lwesp_conn_t::buff (C++ member), 131
- lwesp_conn_t::client (C++ member), 131
- lwesp_conn_t::data_received (C++ member), 131
- lwesp_conn_t::evt_func (C++ member), 131
- lwesp_conn_t::f (C++ member), 132
- lwesp_conn_t::in_closing (C++ member), 132
- lwesp_conn_t::local_port (C++ member), 131
- lwesp_conn_t::num (C++ member), 131
- lwesp_conn_t::receive_blocked (C++ member), 132
- lwesp_conn_t::receive_is_command_queued (C++ member), 132
- lwesp_conn_t::remote_ip (C++ member), 131
- lwesp_conn_t::remote_port (C++ member), 131
- lwesp_conn_t::status (C++ member), 132
- lwesp_conn_t::tcp_available_bytes (C++ member), 131
- lwesp_conn_t::tcp_not_ack_bytes (C++ member), 131
- lwesp_conn_t::total_recved (C++ member), 131
- lwesp_conn_t::type (C++ member), 131
- lwesp_conn_t::val_id (C++ member), 131
- lwesp_conn_type_t (C++ enum), 79
- lwesp_conn_type_t::LWESP_CONN_TYPE_SSL (C++ enumerator), 79
- lwesp_conn_type_t::LWESP_CONN_TYPE_TCP (C++ enumerator), 79
- lwesp_conn_type_t::LWESP_CONN_TYPE_UDP (C++ enumerator), 79
- lwesp_conn_write (C++ function), 82
- lwesp_core_lock (C++ function), 151
- lwesp_core_unlock (C++ function), 151
- lwesp_datetime_t (C++ struct), 142
- lwesp_datetime_t::date (C++ member), 142
- lwesp_datetime_t::day (C++ member), 142
- lwesp_datetime_t::hours (C++ member), 142
- lwesp_datetime_t::minutes (C++ member), 142
- lwesp_datetime_t::month (C++ member), 142
- lwesp_datetime_t::seconds (C++ member), 142
- lwesp_datetime_t::year (C++ member), 142
- LWESP_DBG_LVL_ALL (C macro), 86
- LWESP_DBG_LVL_DANGER (C macro), 86
- LWESP_DBG_LVL_MASK (C macro), 86
- LWESP_DBG_LVL_SEVERE (C macro), 86
- LWESP_DBG_LVL_WARNING (C macro), 86
- LWESP_DBG_OFF (C macro), 86
- LWESP_DBG_ON (C macro), 86
- LWESP_DBG_TYPE_ALL (C macro), 86
- LWESP_DBG_TYPE_STATE (C macro), 86
- LWESP_DBG_TYPE_TRACE (C macro), 86
- LWESP_DEBUGF (C macro), 86
- LWESP_DEBUGW (C macro), 87
- lwesp_delay (C++ function), 152
- lwesp_device_is_esp32 (C++ function), 152
- lwesp_device_is_esp8266 (C++ function), 152
- lwesp_device_is_present (C++ function), 152
- lwesp_device_set_present (C++ function), 151
- lwesp_device_t (C++ enum), 129
- lwesp_device_t::LWESP_DEVICE_ESP32 (C++ enumerator), 129
- lwesp_device_t::LWESP_DEVICE_ESP8266 (C++ enumerator), 129
- lwesp_device_t::LWESP_DEVICE_UNKNOWN (C++ enumerator), 129
- lwesp_dhcp_configure (C++ function), 87
- lwesp_dns_get_config (C++ function), 88
- lwesp_dns_gethostbyname (C++ function), 88
- lwesp_dns_set_config (C++ function), 88
- lwesp_ecn_t (C++ enum), 130
- lwesp_ecn_t::LWESP_ECN_OPEN (C++ enumerator), 130
- lwesp_ecn_t::LWESP_ECN_WEP (C++ enumerator), 130
- lwesp_ecn_t::LWESP_ECN_WPA2_Enterprise (C++ enumerator), 130
- lwesp_ecn_t::LWESP_ECN_WPA2_PSK (C++ enumerator), 130
- lwesp_ecn_t::LWESP_ECN_WPA_PSK (C++ enumerator), 130

lwesp_ecn_t::LWESP_ECN_WPA_WPA2_PSK
 (C++ enumerator), 130
 lwesp_evt_ap_connected_sta_get_mac (C++
 function), 90
 lwesp_evt_ap_disconnected_sta_get_mac
 (C++ function), 90
 lwesp_evt_ap_ip_sta_get_ip (C++ function),
 90
 lwesp_evt_ap_ip_sta_get_mac (C++ function),
 90
 lwesp_evt_conn_active_get_conn (C++ func-
 tion), 92
 lwesp_evt_conn_active_is_client (C++
 function), 92
 lwesp_evt_conn_close_get_conn (C++ func-
 tion), 92
 lwesp_evt_conn_close_get_result (C++
 function), 92
 lwesp_evt_conn_close_is_client (C++ func-
 tion), 92
 lwesp_evt_conn_close_is_forced (C++ func-
 tion), 92
 lwesp_evt_conn_error_get_arg (C++ func-
 tion), 93
 lwesp_evt_conn_error_get_error (C++ func-
 tion), 93
 lwesp_evt_conn_error_get_host (C++ func-
 tion), 93
 lwesp_evt_conn_error_get_port (C++ func-
 tion), 93
 lwesp_evt_conn_error_get_type (C++ func-
 tion), 93
 lwesp_evt_conn_poll_get_conn (C++ func-
 tion), 93
 lwesp_evt_conn_recv_get_buff (C++ func-
 tion), 91
 lwesp_evt_conn_recv_get_conn (C++ func-
 tion), 91
 lwesp_evt_conn_send_get_conn (C++ func-
 tion), 91
 lwesp_evt_conn_send_get_length (C++ func-
 tion), 91
 lwesp_evt_conn_send_get_result (C++ func-
 tion), 91
 lwesp_evt_dns_hostbyname_get_host (C++
 function), 96
 lwesp_evt_dns_hostbyname_get_ip (C++
 function), 96
 lwesp_evt_dns_hostbyname_get_result
 (C++ function), 96
 lwesp_evt_fn (C++ type), 97
 lwesp_evt_func_t (C++ struct), 139
 lwesp_evt_func_t::fn (C++ member), 140
 lwesp_evt_func_t::next (C++ member), 140
 lwesp_evt_get_type (C++ function), 99
 lwesp_evt_ping_get_host (C++ function), 96
 lwesp_evt_ping_get_result (C++ function), 96
 lwesp_evt_ping_get_time (C++ function), 96
 lwesp_evt_register (C++ function), 99
 lwesp_evt_reset_detected_is_forced (C++
 function), 89
 lwesp_evt_reset_get_result (C++ function),
 89
 lwesp_evt_restore_get_result (C++ func-
 tion), 89
 lwesp_evt_server_get_port (C++ function), 97
 lwesp_evt_server_get_result (C++ function),
 97
 lwesp_evt_server_is_enable (C++ function),
 97
 lwesp_evt_sta_info_ap_get_channel (C++
 function), 95
 lwesp_evt_sta_info_ap_get_mac (C++ func-
 tion), 95
 lwesp_evt_sta_info_ap_get_result (C++
 function), 95
 lwesp_evt_sta_info_ap_get_rssi (C++ func-
 tion), 95
 lwesp_evt_sta_info_ap_get_ssid (C++ func-
 tion), 95
 lwesp_evt_sta_join_ap_get_result (C++
 function), 94
 lwesp_evt_sta_list_ap_get_aps (C++ func-
 tion), 94
 lwesp_evt_sta_list_ap_get_length (C++
 function), 94
 lwesp_evt_sta_list_ap_get_result (C++
 function), 94
 lwesp_evt_t (C++ struct), 99
 lwesp_evt_t::ap_conn_disconn_sta (C++
 member), 101
 lwesp_evt_t::ap_ip_sta (C++ member), 101
 lwesp_evt_t::aps (C++ member), 101
 lwesp_evt_t::arg (C++ member), 100
 lwesp_evt_t::buff (C++ member), 100
 lwesp_evt_t::client (C++ member), 100
 lwesp_evt_t::conn (C++ member), 100
 lwesp_evt_t::conn_active_close (C++ mem-
 ber), 100
 lwesp_evt_t::conn_data_recv (C++ member),
 100
 lwesp_evt_t::conn_data_send (C++ member),
 100
 lwesp_evt_t::conn_error (C++ member), 100
 lwesp_evt_t::conn_poll (C++ member), 101
 lwesp_evt_t::dns_hostbyname (C++ member),
 101
 lwesp_evt_t::en (C++ member), 101

- lwesp_evt_t::err (C++ member), 100
 lwesp_evt_t::evt (C++ member), 101
 lwesp_evt_t::forced (C++ member), 99
 lwesp_evt_t::host (C++ member), 100
 lwesp_evt_t::info (C++ member), 101
 lwesp_evt_t::ip (C++ member), 101
 lwesp_evt_t::len (C++ member), 101
 lwesp_evt_t::mac (C++ member), 101
 lwesp_evt_t::ping (C++ member), 101
 lwesp_evt_t::port (C++ member), 100
 lwesp_evt_t::res (C++ member), 99
 lwesp_evt_t::reset (C++ member), 100
 lwesp_evt_t::reset_detected (C++ member), 99
 lwesp_evt_t::restore (C++ member), 100
 lwesp_evt_t::sent (C++ member), 100
 lwesp_evt_t::server (C++ member), 101
 lwesp_evt_t::sta_info_ap (C++ member), 101
 lwesp_evt_t::sta_join_ap (C++ member), 101
 lwesp_evt_t::sta_list_ap (C++ member), 101
 lwesp_evt_t::time (C++ member), 101
 lwesp_evt_t::type (C++ member), 99, 100
 lwesp_evt_type_t (C++ enum), 97
 lwesp_evt_type_t::LWESP_EVT_AP_CONNECTED (C++ enumerator), 98
 lwesp_evt_type_t::LWESP_EVT_AP_DISCONNECTED (C++ enumerator), 98
 lwesp_evt_type_t::LWESP_EVT_AP_IP_STA (C++ enumerator), 99
 lwesp_evt_type_t::LWESP_EVT_AT_VERSION_NOT_SUPPORTED (C++ enumerator), 98
 lwesp_evt_type_t::LWESP_EVT_CMD_TIMEOUT (C++ enumerator), 98
 lwesp_evt_type_t::LWESP_EVT_CONN_ACTIVE (C++ enumerator), 98
 lwesp_evt_type_t::LWESP_EVT_CONN_CLOSE (C++ enumerator), 98
 lwesp_evt_type_t::LWESP_EVT_CONN_ERROR (C++ enumerator), 98
 lwesp_evt_type_t::LWESP_EVT_CONN_POLL (C++ enumerator), 98
 lwesp_evt_type_t::LWESP_EVT_CONN_RECV (C++ enumerator), 98
 lwesp_evt_type_t::LWESP_EVT_CONN_SEND (C++ enumerator), 98
 lwesp_evt_type_t::LWESP_EVT_DEVICE_PRESENT (C++ enumerator), 98
 lwesp_evt_type_t::LWESP_EVT_DNS_HOSTBYNAME (C++ enumerator), 99
 lwesp_evt_type_t::LWESP_EVT_INIT_FINISH (C++ enumerator), 97
 lwesp_evt_type_t::LWESP_EVT_PING (C++ enumerator), 99
 lwesp_evt_type_t::LWESP_EVT_RESET (C++ enumerator), 97
 lwesp_evt_type_t::LWESP_EVT_RESET_DETECTED (C++ enumerator), 97
 lwesp_evt_type_t::LWESP_EVT_RESTORE (C++ enumerator), 98
 lwesp_evt_type_t::LWESP_EVT_SERVER (C++ enumerator), 98
 lwesp_evt_type_t::LWESP_EVT_STA_INFO_AP (C++ enumerator), 98
 lwesp_evt_type_t::LWESP_EVT_STA_JOIN_AP (C++ enumerator), 98
 lwesp_evt_type_t::LWESP_EVT_STA_LIST_AP (C++ enumerator), 98
 lwesp_evt_type_t::LWESP_EVT_WIFI_CONNECTED (C++ enumerator), 98
 lwesp_evt_type_t::LWESP_EVT_WIFI_DISCONNECTED (C++ enumerator), 98
 lwesp_evt_type_t::LWESP_EVT_WIFI_GOT_IP (C++ enumerator), 98
 lwesp_evt_type_t::LWESP_EVT_WIFI_IP_ACQUIRED (C++ enumerator), 98
 lwesp_evt_unregister (C++ function), 99
 lwesp_get_conns_status (C++ function), 82
 lwesp_get_current_at_fw_version (C++ function), 152
 lwesp_get_min_at_fw_version (C macro), 148
 lwesp_get_wifi_mode (C++ function), 150
 lwesp_hostname_get (C++ function), 102
 lwesp_hostname_set (C++ function), 102
 lwesp_http_method_t (C++ enum), 130
 lwesp_http_method_t::LWESP_HTTP_METHOD_CONNECT (C++ enumerator), 130
 lwesp_http_method_t::LWESP_HTTP_METHOD_DELETE (C++ enumerator), 130
 lwesp_http_method_t::LWESP_HTTP_METHOD_GET (C++ enumerator), 130
 lwesp_http_method_t::LWESP_HTTP_METHOD_HEAD (C++ enumerator), 130
 lwesp_http_method_t::LWESP_HTTP_METHOD_OPTIONS (C++ enumerator), 130
 lwesp_http_method_t::LWESP_HTTP_METHOD_PATCH (C++ enumerator), 131
 lwesp_http_method_t::LWESP_HTTP_METHOD_POST (C++ enumerator), 130
 lwesp_http_method_t::LWESP_HTTP_METHOD_PUT (C++ enumerator), 130
 lwesp_http_method_t::LWESP_HTTP_METHOD_TRACE (C++ enumerator), 131
 lwesp_http_server_init (C++ function), 175
 lwesp_http_server_write (C++ function), 175
 lwesp_http_server_write_string (C macro), 173
 LWESP_I16 (C macro), 145
 lwesp_il6_to_str (C macro), 146

- LWESP_I32 (*C macro*), 145
 lwesp_i32_to_gen_str (*C++ function*), 148
 lwesp_i32_to_str (*C macro*), 146
 LWESP_I8 (*C macro*), 145
 lwesp_i8_to_str (*C macro*), 147
 lwesp_init (*C++ function*), 149
 lwesp_input (*C++ function*), 103
 lwesp_input_process (*C++ function*), 103
 lwesp_ip_mac_t (*C++ struct*), 139
 lwesp_ip_mac_t::dhcp (*C++ member*), 139
 lwesp_ip_mac_t::gw (*C++ member*), 139
 lwesp_ip_mac_t::has_ip (*C++ member*), 139
 lwesp_ip_mac_t::ip (*C++ member*), 139
 lwesp_ip_mac_t::is_connected (*C++ member*), 139
 lwesp_ip_mac_t::mac (*C++ member*), 139
 lwesp_ip_mac_t::nm (*C++ member*), 139
 lwesp_ip_t (*C++ struct*), 141
 lwesp_ip_t::ip (*C++ member*), 142
 lwesp_ipd_t (*C++ struct*), 132
 lwesp_ipd_t::buff (*C++ member*), 133
 lwesp_ipd_t::buff_ptr (*C++ member*), 133
 lwesp_ipd_t::conn (*C++ member*), 132
 lwesp_ipd_t::ip (*C++ member*), 132
 lwesp_ipd_t::port (*C++ member*), 133
 lwesp_ipd_t::read (*C++ member*), 132
 lwesp_ipd_t::rem_len (*C++ member*), 132
 lwesp_ipd_t::tot_len (*C++ member*), 132
 lwesp_linbuff_t (*C++ struct*), 142
 lwesp_linbuff_t::buff (*C++ member*), 143
 lwesp_linbuff_t::len (*C++ member*), 143
 lwesp_linbuff_t::ptr (*C++ member*), 143
 lwesp_link_conn_t (*C++ struct*), 139
 lwesp_link_conn_t::failed (*C++ member*), 139
 lwesp_link_conn_t::is_server (*C++ member*), 139
 lwesp_link_conn_t::local_port (*C++ member*), 139
 lwesp_link_conn_t::num (*C++ member*), 139
 lwesp_link_conn_t::remote_ip (*C++ member*), 139
 lwesp_link_conn_t::remote_port (*C++ member*), 139
 lwesp_link_conn_t::type (*C++ member*), 139
 lwesp_ll_deinit (*C++ function*), 162
 lwesp_ll_init (*C++ function*), 162
 lwesp_ll_reset_fn (*C++ type*), 161
 lwesp_ll_send_fn (*C++ type*), 161
 lwesp_ll_t (*C++ struct*), 162
 lwesp_ll_t::baudrate (*C++ member*), 162
 lwesp_ll_t::reset_fn (*C++ member*), 162
 lwesp_ll_t::send_fn (*C++ member*), 162
 lwesp_ll_t::uart (*C++ member*), 162
 lwesp_mac_t (*C++ struct*), 142
 lwesp_mac_t::mac (*C++ member*), 142
 LWESP_MAX (*C macro*), 144
 lwesp_mdns_configure (*C++ function*), 104
 LWESP_MEM_ALIGN (*C macro*), 144
 lwesp_mem_assignmemory (*C++ function*), 104
 lwesp_mem_calloc (*C++ function*), 105
 lwesp_mem_free (*C++ function*), 105
 lwesp_mem_free_s (*C++ function*), 105
 lwesp_mem_malloc (*C++ function*), 104
 lwesp_mem_realloc (*C++ function*), 105
 lwesp_mem_region_t (*C++ struct*), 105
 lwesp_mem_region_t::size (*C++ member*), 106
 lwesp_mem_region_t::start_addr (*C++ member*), 106
 LWESP_MEMCPY (*C macro*), 158
 LWESP_MEMSET (*C macro*), 158
 LWESP_MIN (*C macro*), 144
 LWESP_MIN_AT_VERSION_MAJOR_ESP32 (*C macro*), 155
 LWESP_MIN_AT_VERSION_MAJOR_ESP8266 (*C macro*), 155
 LWESP_MIN_AT_VERSION_MINOR_ESP32 (*C macro*), 155
 LWESP_MIN_AT_VERSION_MINOR_ESP8266 (*C macro*), 155
 LWESP_MIN_AT_VERSION_PATCH_ESP32 (*C macro*), 155
 LWESP_MIN_AT_VERSION_PATCH_ESP8266 (*C macro*), 155
 lwesp_mode_t (*C++ enum*), 130
 lwesp_mode_t::LWESP_MODE_AP (*C++ enumerator*), 130
 lwesp_mode_t::LWESP_MODE_STA (*C++ enumerator*), 130
 lwesp_mode_t::LWESP_MODE_STA_AP (*C++ enumerator*), 130
 lwesp_modules_t (*C++ struct*), 140
 lwesp_modules_t::active_conns (*C++ member*), 140
 lwesp_modules_t::active_conns_last (*C++ member*), 140
 lwesp_modules_t::ap (*C++ member*), 140
 lwesp_modules_t::conns (*C++ member*), 140
 lwesp_modules_t::device (*C++ member*), 140
 lwesp_modules_t::ipd (*C++ member*), 140
 lwesp_modules_t::link_conn (*C++ member*), 140
 lwesp_modules_t::sta (*C++ member*), 140
 lwesp_modules_t::version_at (*C++ member*), 140
 lwesp_modules_t::version_sdk (*C++ member*), 140

- lwesp_mqtt_client_api_buf_free (C++ *function*), 198
- lwesp_mqtt_client_api_buf_p (C++ *type*), 196
- lwesp_mqtt_client_api_buf_t (C++ *struct*), 198
- lwesp_mqtt_client_api_buf_t::payload (C++ *member*), 198
- lwesp_mqtt_client_api_buf_t::payload_len (C++ *member*), 198
- lwesp_mqtt_client_api_buf_t::qos (C++ *member*), 198
- lwesp_mqtt_client_api_buf_t::topic (C++ *member*), 198
- lwesp_mqtt_client_api_buf_t::topic_len (C++ *member*), 198
- lwesp_mqtt_client_api_close (C++ *function*), 196
- lwesp_mqtt_client_api_connect (C++ *function*), 196
- lwesp_mqtt_client_api_delete (C++ *function*), 196
- lwesp_mqtt_client_api_is_connected (C++ *function*), 197
- lwesp_mqtt_client_api_new (C++ *function*), 196
- lwesp_mqtt_client_api_publish (C++ *function*), 197
- lwesp_mqtt_client_api_receive (C++ *function*), 197
- lwesp_mqtt_client_api_subscribe (C++ *function*), 196
- lwesp_mqtt_client_api_unsubscribe (C++ *function*), 197
- lwesp_mqtt_client_connect (C++ *function*), 186
- lwesp_mqtt_client_delete (C++ *function*), 186
- lwesp_mqtt_client_disconnect (C++ *function*), 187
- lwesp_mqtt_client_evt_connect_get_status (C *macro*), 190
- lwesp_mqtt_client_evt_disconnect_is_accepted (C *macro*), 190
- lwesp_mqtt_client_evt_get_type (C *macro*), 193
- lwesp_mqtt_client_evt_publish_get_argument (C *macro*), 193
- lwesp_mqtt_client_evt_publish_get_result (C *macro*), 193
- lwesp_mqtt_client_evt_publish_rcv_get_payload (C *macro*), 192
- lwesp_mqtt_client_evt_publish_rcv_get_payload_len (C *macro*), 192
- lwesp_mqtt_client_evt_publish_rcv_get_qos (C *macro*), 192
- lwesp_mqtt_client_evt_publish_rcv_get_topic (C *macro*), 191
- lwesp_mqtt_client_evt_publish_rcv_get_topic_len (C *macro*), 192
- lwesp_mqtt_client_evt_publish_rcv_is_duplicate (C *macro*), 192
- lwesp_mqtt_client_evt_subscribe_get_argument (C *macro*), 191
- lwesp_mqtt_client_evt_subscribe_get_result (C *macro*), 191
- lwesp_mqtt_client_evt_unsubscribe_get_argument (C *macro*), 191
- lwesp_mqtt_client_evt_unsubscribe_get_result (C *macro*), 191
- lwesp_mqtt_client_get_arg (C++ *function*), 188
- lwesp_mqtt_client_info_t (C++ *struct*), 188
- lwesp_mqtt_client_info_t::id (C++ *member*), 188
- lwesp_mqtt_client_info_t::keep_alive (C++ *member*), 188
- lwesp_mqtt_client_info_t::pass (C++ *member*), 188
- lwesp_mqtt_client_info_t::user (C++ *member*), 188
- lwesp_mqtt_client_info_t::will_message (C++ *member*), 188
- lwesp_mqtt_client_info_t::will_qos (C++ *member*), 189
- lwesp_mqtt_client_info_t::will_topic (C++ *member*), 188
- lwesp_mqtt_client_is_connected (C++ *function*), 187
- lwesp_mqtt_client_new (C++ *function*), 186
- lwesp_mqtt_client_p (C++ *type*), 184
- lwesp_mqtt_client_publish (C++ *function*), 187
- lwesp_mqtt_client_set_arg (C++ *function*), 188
- lwesp_mqtt_client_subscribe (C++ *function*), 187
- lwesp_mqtt_client_unsubscribe (C++ *function*), 187
- lwesp_mqtt_conn_status_t (C++ *enum*), 186
- lwesp_mqtt_conn_status_t::LWESP_MQTT_CONN_STATUS_ACCEPTED (C++ *enumerator*), 186
- lwesp_mqtt_conn_status_t::LWESP_MQTT_CONN_STATUS_REJECTED (C++ *enumerator*), 186
- lwesp_mqtt_conn_status_t::LWESP_MQTT_CONN_STATUS_REJECTED_WITH_REASON (C++ *enumerator*), 186
- lwesp_mqtt_conn_status_t::LWESP_MQTT_CONN_STATUS_REJECTED_WITH_REASON_CODE (C++ *enumerator*), 186
- lwesp_mqtt_conn_status_t::LWESP_MQTT_CONN_STATUS_REJECTED_WITH_REASON_CODE_CODE (C++ *enumerator*), 186

lwesp_mqtt_conn_status_t::LWESP_MQTT_CONN_STATUS_ACCEPTED (C++ enumerator), 186
 lwesp_mqtt_conn_status_t::LWESP_MQTT_CONN_STATUS_REFUSED_USER (C++ struct), 189
 lwesp_mqtt_conn_status_t::LWESP_MQTT_CONN_STATUS_TCP_FAILED (C++ enumerator), 186
 lwesp_mqtt_evt_fn (C++ type), 184
 lwesp_mqtt_evt_t (C++ struct), 189
 lwesp_mqtt_evt_t::arg (C++ member), 189
 lwesp_mqtt_evt_t::connect (C++ member), 189
 lwesp_mqtt_evt_t::disconnect (C++ member), 189
 lwesp_mqtt_evt_t::dup (C++ member), 190
 lwesp_mqtt_evt_t::evt (C++ member), 190
 lwesp_mqtt_evt_t::is_accepted (C++ member), 189
 lwesp_mqtt_evt_t::payload (C++ member), 190
 lwesp_mqtt_evt_t::payload_len (C++ member), 190
 lwesp_mqtt_evt_t::publish (C++ member), 189
 lwesp_mqtt_evt_t::publish_recv (C++ member), 190
 lwesp_mqtt_evt_t::qos (C++ member), 190
 lwesp_mqtt_evt_t::res (C++ member), 189
 lwesp_mqtt_evt_t::status (C++ member), 189
 lwesp_mqtt_evt_t::sub_unsub_scribed (C++ member), 189
 lwesp_mqtt_evt_t::topic (C++ member), 189
 lwesp_mqtt_evt_t::topic_len (C++ member), 189
 lwesp_mqtt_evt_t::type (C++ member), 189
 lwesp_mqtt_evt_type_t (C++ enum), 185
 lwesp_mqtt_evt_type_t::LWESP_MQTT_EVT_CONNECTED (C++ enumerator), 185
 lwesp_mqtt_evt_type_t::LWESP_MQTT_EVT_DISCONNECTED (C++ enumerator), 185
 lwesp_mqtt_evt_type_t::LWESP_MQTT_EVT_KEEP_ALIVE (C++ enumerator), 185
 lwesp_mqtt_evt_type_t::LWESP_MQTT_EVT_PUBLISH (C++ enumerator), 185
 lwesp_mqtt_evt_type_t::LWESP_MQTT_EVT_PUBLISH_RECV (C++ enumerator), 185
 lwesp_mqtt_evt_type_t::LWESP_MQTT_EVT_SUBSCRIBE (C++ enumerator), 185
 lwesp_mqtt_evt_type_t::LWESP_MQTT_EVT_UNSUBSCRIBE (C++ enumerator), 185
 lwesp_mqtt_qos_t (C++ enum), 185
 lwesp_mqtt_qos_t::LWESP_MQTT_QOS_AT_LEAST_ONCE (C++ enumerator), 185
 lwesp_mqtt_qos_t::LWESP_MQTT_QOS_AT_MOST_ONCE (C++ enumerator), 185
 lwesp_mqtt_qos_t::LWESP_MQTT_QOS_EXACTLY_ONCE (C++ enumerator), 185
 lwesp_mqtt_request_t::arg (C++ member), 189
 lwesp_mqtt_request_t::expected_sent_len (C++ member), 189
 lwesp_mqtt_request_t::packet_id (C++ member), 189
 lwesp_mqtt_request_t::status (C++ member), 189
 lwesp_mqtt_request_t::timeout_start_time (C++ member), 189
 lwesp_mqtt_state_t (C++ enum), 185
 lwesp_mqtt_state_t::LWESP_MQTT_CONN_CONNECTING (C++ enumerator), 185
 lwesp_mqtt_state_t::LWESP_MQTT_CONN_DISCONNECTED (C++ enumerator), 185
 lwesp_mqtt_state_t::LWESP_MQTT_CONN_DISCONNECTING (C++ enumerator), 185
 lwesp_mqtt_state_t::LWESP_MQTT_CONNECTED (C++ enumerator), 185
 lwesp_mqtt_state_t::LWESP_MQTT_CONNECTING (C++ enumerator), 185
 lwesp_msg_t (C++ struct), 133
 lwesp_msg_t::ap (C++ member), 136
 lwesp_msg_t::ap_conf (C++ member), 135
 lwesp_msg_t::ap_conf_get (C++ member), 135
 lwesp_msg_t::ap_disconn_sta (C++ member), 135
 lwesp_msg_t::ap_list (C++ member), 134
 lwesp_msg_t::apf (C++ member), 134
 lwesp_msg_t::aps (C++ member), 134
 lwesp_msg_t::apsi (C++ member), 134
 lwesp_msg_t::apsl (C++ member), 134
 lwesp_msg_t::arg (C++ member), 136
 lwesp_msg_t::auth_mode (C++ member), 138
 lwesp_msg_t::baudrate (C++ member), 133
 lwesp_msg_t::block_time (C++ member), 133
 lwesp_msg_t::btw (C++ member), 137
 lwesp_msg_t::bw (C++ member), 137
 lwesp_msg_t::ca_number (C++ member), 138
 lwesp_msg_t::cb (C++ member), 137
 lwesp_msg_t::ch (C++ member), 134
 lwesp_msg_t::cmd (C++ member), 133
 lwesp_msg_t::cmd_def (C++ member), 133
 lwesp_msg_t::conn (C++ member), 136, 137
 lwesp_msg_t::conn_close (C++ member), 137
 lwesp_msg_t::conn_send (C++ member), 137
 lwesp_msg_t::conn_start (C++ member), 136
 lwesp_msg_t::data (C++ member), 137
 lwesp_msg_t::delay (C++ member), 133
 lwesp_msg_t::dt (C++ member), 138
 lwesp_msg_t::ecn (C++ member), 134
 lwesp_msg_t::en (C++ member), 134
 lwesp_msg_t::error_num (C++ member), 134

- lwesp_msg_t::evt_func (C++ member), 136
 lwesp_msg_t::fau (C++ member), 137
 lwesp_msg_t::fn (C++ member), 133
 lwesp_msg_t::gw (C++ member), 135
 lwesp_msg_t::h1 (C++ member), 138
 lwesp_msg_t::h2 (C++ member), 138
 lwesp_msg_t::h3 (C++ member), 138
 lwesp_msg_t::hid (C++ member), 135
 lwesp_msg_t::host (C++ member), 138
 lwesp_msg_t::hostname_get (C++ member), 136
 lwesp_msg_t::hostname_set (C++ member), 136
 lwesp_msg_t::i (C++ member), 133
 lwesp_msg_t::info (C++ member), 134
 lwesp_msg_t::ip (C++ member), 135
 lwesp_msg_t::is_blocking (C++ member), 133
 lwesp_msg_t::length (C++ member), 136
 lwesp_msg_t::link_id (C++ member), 138
 lwesp_msg_t::local_ip (C++ member), 136
 lwesp_msg_t::mac (C++ member), 134, 135
 lwesp_msg_t::max_conn (C++ member), 137
 lwesp_msg_t::max_sta (C++ member), 134
 lwesp_msg_t::mdns (C++ member), 138
 lwesp_msg_t::mode (C++ member), 133
 lwesp_msg_t::mode_get (C++ member), 133
 lwesp_msg_t::msg (C++ member), 138
 lwesp_msg_t::name (C++ member), 133
 lwesp_msg_t::nm (C++ member), 135
 lwesp_msg_t::num (C++ member), 136
 lwesp_msg_t::pass (C++ member), 134
 lwesp_msg_t::pki_number (C++ member), 138
 lwesp_msg_t::port (C++ member), 137
 lwesp_msg_t::ptr (C++ member), 137
 lwesp_msg_t::pwd (C++ member), 134
 lwesp_msg_t::remote_host (C++ member), 136
 lwesp_msg_t::remote_ip (C++ member), 137
 lwesp_msg_t::remote_port (C++ member), 136
 lwesp_msg_t::res (C++ member), 133
 lwesp_msg_t::reset (C++ member), 133
 lwesp_msg_t::sem (C++ member), 133
 lwesp_msg_t::sent (C++ member), 137
 lwesp_msg_t::sent_all (C++ member), 137
 lwesp_msg_t::server (C++ member), 138
 lwesp_msg_t::size (C++ member), 137
 lwesp_msg_t::ssid (C++ member), 134
 lwesp_msg_t::sta (C++ member), 136
 lwesp_msg_t::sta_ap_getip (C++ member), 135
 lwesp_msg_t::sta_ap_getmac (C++ member), 135
 lwesp_msg_t::sta_ap_setip (C++ member), 135
 lwesp_msg_t::sta_ap_setmac (C++ member), 136
 lwesp_msg_t::sta_autojoin (C++ member), 134
 lwesp_msg_t::sta_info_ap (C++ member), 134
 lwesp_msg_t::sta_join (C++ member), 134
 lwesp_msg_t::sta_list (C++ member), 135
 lwesp_msg_t::staf (C++ member), 135
 lwesp_msg_t::stai (C++ member), 135
 lwesp_msg_t::stal (C++ member), 135
 lwesp_msg_t::stas (C++ member), 135
 lwesp_msg_t::success (C++ member), 136
 lwesp_msg_t::tcp_ssl_keep_alive (C++ member), 136
 lwesp_msg_t::tcPIP_ping (C++ member), 138
 lwesp_msg_t::tcPIP_server (C++ member), 137
 lwesp_msg_t::tcPIP_snTP_cfg (C++ member), 138
 lwesp_msg_t::tcPIP_snTP_time (C++ member), 138
 lwesp_msg_t::tcPIP_ssl_cfg (C++ member), 138
 lwesp_msg_t::tcPIP_sslsize (C++ member), 138
 lwesp_msg_t::time (C++ member), 138
 lwesp_msg_t::time_out (C++ member), 138
 lwesp_msg_t::timeout (C++ member), 137
 lwesp_msg_t::tries (C++ member), 137
 lwesp_msg_t::type (C++ member), 136
 lwesp_msg_t::tz (C++ member), 138
 lwesp_msg_t::uart (C++ member), 133
 lwesp_msg_t::udp_local_port (C++ member), 136
 lwesp_msg_t::udp_mode (C++ member), 136
 lwesp_msg_t::val_id (C++ member), 137
 lwesp_msg_t::wait_send_ok_err (C++ member), 137
 lwesp_msg_t::wifi_cWdhCP (C++ member), 136
 lwesp_msg_t::wifi_hostname (C++ member), 136
 lwesp_msg_t::wifi_mode (C++ member), 133
 lwesp_msg_t::wps_cfg (C++ member), 138
 lwesp_netconn_accept (C++ function), 211
 lwesp_netconn_bind (C++ function), 208
 lwesp_netconn_close (C++ function), 209
 lwesp_netconn_connect (C++ function), 208
 lwesp_netconn_connect_ex (C++ function), 210
 lwesp_netconn_delete (C++ function), 208
 lwesp_netconn_flush (C++ function), 211
 lwesp_netconn_get_conn (C++ function), 209
 lwesp_netconn_get_connnum (C++ function), 209

- lwesp_netconn_get_receive_timeout (C++ function), 209
 lwesp_netconn_listen (C++ function), 210
 lwesp_netconn_listen_with_max_conn (C++ function), 210
 lwesp_netconn_new (C++ function), 208
 lwesp_netconn_p (C++ type), 207
 lwesp_netconn_receive (C++ function), 208
 LWESP_NETCONN_RECEIVE_NO_WAIT (C macro), 207
 lwesp_netconn_send (C++ function), 211
 lwesp_netconn_sendto (C++ function), 211
 lwesp_netconn_set_listen_conn_timeout (C++ function), 210
 lwesp_netconn_set_receive_timeout (C++ function), 209
 lwesp_netconn_type_t (C++ enum), 208
 lwesp_netconn_type_t::LWESP_NETCONN_TYPE_SOCKET (C++ enumerator), 208
 lwesp_netconn_type_t::LWESP_NETCONN_TYPE_TCP (C++ enumerator), 208
 lwesp_netconn_type_t::LWESP_NETCONN_TYPE_UDP (C++ enumerator), 208
 lwesp_netconn_write (C++ function), 211
 lwesp_pbuf_advance (C++ function), 114
 lwesp_pbuf_cat (C++ function), 112
 lwesp_pbuf_chain (C++ function), 112
 lwesp_pbuf_copy (C++ function), 111
 lwesp_pbuf_data (C++ function), 110
 lwesp_pbuf_dump (C++ function), 115
 lwesp_pbuf_free (C++ function), 110
 lwesp_pbuf_get_at (C++ function), 112
 lwesp_pbuf_get_linear_addr (C++ function), 114
 lwesp_pbuf_length (C++ function), 111
 lwesp_pbuf_memcmp (C++ function), 113
 lwesp_pbuf_memfind (C++ function), 113
 lwesp_pbuf_new (C++ function), 110
 lwesp_pbuf_p (C++ type), 110
 lwesp_pbuf_ref (C++ function), 112
 lwesp_pbuf_set_ip (C++ function), 114
 lwesp_pbuf_set_length (C++ function), 111
 lwesp_pbuf_skip (C++ function), 114
 lwesp_pbuf_strcmp (C++ function), 113
 lwesp_pbuf_strfind (C++ function), 113
 lwesp_pbuf_t (C++ struct), 115, 132
 lwesp_pbuf_t::ip (C++ member), 115, 132
 lwesp_pbuf_t::len (C++ member), 115, 132
 lwesp_pbuf_t::next (C++ member), 115, 132
 lwesp_pbuf_t::payload (C++ member), 115, 132
 lwesp_pbuf_t::port (C++ member), 115, 132
 lwesp_pbuf_t::ref (C++ member), 115, 132
 lwesp_pbuf_t::tot_len (C++ member), 115, 132
 lwesp_pbuf_take (C++ function), 111
 lwesp_pbuf_unchain (C++ function), 112
 lwesp_ping (C++ function), 115
 lwesp_port_t (C++ type), 125
 lwesp_reset (C++ function), 149
 lwesp_reset_with_delay (C++ function), 149
 lwesp_restore (C++ function), 149
 lwesp_set_at_baudrate (C++ function), 150
 lwesp_set_fw_version (C macro), 148
 lwesp_set_server (C++ function), 150
 lwesp_set_wifi_mode (C++ function), 150
 lwesp_smart_configure (C++ function), 116
 lwesp_sntp_configure (C++ function), 117
 lwesp_sntp_gettime (C++ function), 117
 lwesp_sta_autojoin (C++ function), 120
 lwesp_sta_copy_ip (C++ function), 122
 lwesp_sta_get_ap_info (C++ function), 123
 lwesp_sta_getip (C++ function), 120
 lwesp_sta_getmac (C++ function), 121
 lwesp_sta_has_ip (C++ function), 122
 lwesp_sta_info_ap_t (C++ struct), 71
 lwesp_sta_info_ap_t::ch (C++ member), 71
 lwesp_sta_info_ap_t::mac (C++ member), 71
 lwesp_sta_info_ap_t::rssi (C++ member), 71
 lwesp_sta_info_ap_t::ssid (C++ member), 71
 lwesp_sta_is_ap_802_11b (C++ function), 123
 lwesp_sta_is_ap_802_11g (C++ function), 123
 lwesp_sta_is_ap_802_11n (C++ function), 123
 lwesp_sta_is_joined (C++ function), 122
 lwesp_sta_join (C++ function), 120
 lwesp_sta_list_ap (C++ function), 122
 lwesp_sta_quit (C++ function), 120
 lwesp_sta_setip (C++ function), 121
 lwesp_sta_setmac (C++ function), 122
 lwesp_sta_t (C++ struct), 123
 lwesp_sta_t::ip (C++ member), 124
 lwesp_sta_t::mac (C++ member), 124
 lwesp_sw_version_t (C++ struct), 142
 lwesp_sw_version_t::major (C++ member), 142
 lwesp_sw_version_t::minor (C++ member), 142
 lwesp_sw_version_t::patch (C++ member), 142
 lwesp_sys_init (C++ function), 163
 lwesp_sys_mbox_create (C++ function), 166
 lwesp_sys_mbox_delete (C++ function), 166
 lwesp_sys_mbox_get (C++ function), 166
 lwesp_sys_mbox_getnow (C++ function), 166
 lwesp_sys_mbox_invalid (C++ function), 167
 lwesp_sys_mbox_isvalid (C++ function), 167
 LWESP_SYS_MBOX_NULL (C macro), 168
 lwesp_sys_mbox_put (C++ function), 166
 lwesp_sys_mbox_putnow (C++ function), 166
 lwesp_sys_mbox_t (C++ type), 168

- lwesp_sys_mutex_create (C++ function), 164
- lwesp_sys_mutex_delete (C++ function), 164
- lwesp_sys_mutex_invalid (C++ function), 164
- lwesp_sys_mutex_isvalid (C++ function), 164
- lwesp_sys_mutex_lock (C++ function), 164
- LWESP_SYS_MUTEX_NULL (C macro), 168
- lwesp_sys_mutex_t (C++ type), 168
- lwesp_sys_mutex_unlock (C++ function), 164
- lwesp_sys_now (C++ function), 163
- lwesp_sys_protect (C++ function), 163
- lwesp_sys_sem_create (C++ function), 165
- lwesp_sys_sem_delete (C++ function), 165
- lwesp_sys_sem_invalid (C++ function), 165
- lwesp_sys_sem_isvalid (C++ function), 165
- LWESP_SYS_SEM_NULL (C macro), 168
- lwesp_sys_sem_release (C++ function), 165
- lwesp_sys_sem_t (C++ type), 168
- lwesp_sys_sem_wait (C++ function), 165
- lwesp_sys_thread_create (C++ function), 167
- lwesp_sys_thread_fn (C++ type), 168
- LWESP_SYS_THREAD_PRIO (C macro), 168
- lwesp_sys_thread_prio_t (C++ type), 168
- LWESP_SYS_THREAD_SS (C macro), 168
- lwesp_sys_thread_t (C++ type), 168
- lwesp_sys_thread_terminate (C++ function), 167
- lwesp_sys_thread_yield (C++ function), 167
- LWESP_SYS_TIMEOUT (C macro), 168
- lwesp_sys_unprotect (C++ function), 163
- LWESP_SZ (C macro), 145
- lwesp_t (C++ struct), 140
- lwesp_t::buff (C++ member), 141
- lwesp_t::conn_val_id (C++ member), 141
- lwesp_t::dev_present (C++ member), 141
- lwesp_t::evt (C++ member), 141
- lwesp_t::evt_func (C++ member), 141
- lwesp_t::evt_server (C++ member), 141
- lwesp_t::f (C++ member), 141
- lwesp_t::initialized (C++ member), 141
- lwesp_t::ll (C++ member), 141
- lwesp_t::locked_cnt (C++ member), 140
- lwesp_t::m (C++ member), 141
- lwesp_t::mbox_process (C++ member), 140
- lwesp_t::mbox_producer (C++ member), 140
- lwesp_t::msg (C++ member), 141
- lwesp_t::sem_sync (C++ member), 140
- lwesp_t::status (C++ member), 141
- lwesp_t::thread_process (C++ member), 141
- lwesp_t::thread_produce (C++ member), 140
- LWESP_THREAD_PROCESS_HOOK (C macro), 157
- LWESP_THREAD_PRODUCER_HOOK (C macro), 157
- lwesp_timeout_add (C++ function), 124
- lwesp_timeout_fn (C++ type), 124
- lwesp_timeout_remove (C++ function), 124
- lwesp_timeout_t (C++ struct), 124
- lwesp_timeout_t::arg (C++ member), 125
- lwesp_timeout_t::fn (C++ member), 125
- lwesp_timeout_t::next (C++ member), 125
- lwesp_timeout_t::time (C++ member), 125
- LWESP_U16 (C macro), 145
- lwesp_u16_to_hex_str (C macro), 146
- lwesp_u16_to_str (C macro), 146
- LWESP_U32 (C macro), 145
- lwesp_u32_to_gen_str (C++ function), 147
- lwesp_u32_to_hex_str (C macro), 146
- lwesp_u32_to_str (C macro), 146
- LWESP_U8 (C macro), 145
- lwesp_u8_to_hex_str (C macro), 147
- lwesp_u8_to_str (C macro), 147
- lwesp_unicode_t (C++ struct), 141, 143
- lwesp_unicode_t::ch (C++ member), 141, 143
- lwesp_unicode_t::r (C++ member), 141, 143
- lwesp_unicode_t::res (C++ member), 141, 143
- lwesp_unicode_t::t (C++ member), 141, 143
- LWESP_UNUSED (C macro), 145
- lwesp_update_sw (C++ function), 151
- lwesp_wps_configure (C++ function), 148
- lwespi_unicode_decode (C++ function), 143
- lwespr_t (C++ enum), 128
- lwespr_t::lwespCLOSED (C++ enumerator), 129
- lwespr_t::lwespCONT (C++ enumerator), 129
- lwespr_t::lwespERR (C++ enumerator), 129
- lwespr_t::lwespERRBLOCKING (C++ enumerator), 129
- lwespr_t::lwespERRCONNFAIL (C++ enumerator), 129
- lwespr_t::lwespERRCONNTIMEOUT (C++ enumerator), 129
- lwespr_t::lwespERRMEM (C++ enumerator), 129
- lwespr_t::lwespERRNOAP (C++ enumerator), 129
- lwespr_t::lwespERRNODEVICE (C++ enumerator), 129
- lwespr_t::lwespERRNOFREECONN (C++ enumerator), 129
- lwespr_t::lwespERRNOIP (C++ enumerator), 129
- lwespr_t::lwespERRPASS (C++ enumerator), 129
- lwespr_t::lwespERRWIFINOTCONNECTED (C++ enumerator), 129
- lwespr_t::lwespINPROG (C++ enumerator), 129
- lwespr_t::lwespOK (C++ enumerator), 128
- lwespr_t::lwespOKIGNOREMORE (C++ enumerator), 129
- lwespr_t::lwespPARERR (C++ enumerator), 129
- lwespr_t::lwespTIMEOUT (C++ enumerator), 129