C API Documentation

The C API is very simple and consist of these two basic parts:

- Lifecycle API
- Modem commands

The Typical application workflow is to:

- Create a DCE instance (using `esp_modem_new()`)
- Call specific functions to issue AT commands (Modem commands)
- Switch to the data mode (using `esp_modem_set_mode()`)
- Perform desired network operations (using standard networking API, unrelated to ESP-MODEM)
- Optionally switch back to command mode (again `esp_modem_set_mode()`)
- Destroy the DCE handle (sing `esp_modem_destroy()`)

Note the configuration structures for DTE and DCE, needed for creating the DCE instance, is documented in Configuration structures

Lifecycle API

These functions are used to create, destroy and set modem working mode.

- `esp_modem_new()`
- `esp_modem_destroy()`
- `esp_modem_set_mode()`

group ESP_MODEM_C_API

Set of basic C API for ESP-MODEM.

Typedefs

typedef enum esp_modem_dce_mode `esp_modem_dce_mode_t`

DCE mode: This enum is used to set desired operation mode of the DCE.

typedef enum esp_modem_dce_device `esp_modem_dce_device_t`

DCE devices: Enum list of supported devices.

typedef enum esp_modem_terminal_error `esp_modem_terminal_error_t`

Terminal errors.

typedef void (*`esp_modem_terminal_error_cbt`)(esp_modem_terminal_error_t)

Terminal error callback.

Enums

enum `esp_modem_dce_mode`

DCE mode: This enum is used to set desired operation mode of the DCE.

Values:

enumerator **`ESP_MODEM_MODE_COMMAND`**

Default mode after modem startup, used for sending AT commands

enumerator **`ESP_MODEM_MODE_DATA`**

Used for switching to PPP mode for the modem to connect to a network

enumerator **`ESP_MODEM_MODE_CMUX`**

Multiplexed terminal mode

enumerator **`ESP_MODEM_MODE_CMUX_MANUAL`**

CMUX manual mode

enumerator **`ESP_MODEM_MODE_CMUX_MANUAL_EXIT`**

Exit CMUX manual mode

enumerator **`ESP_MODEM_MODE_CMUX_MANUAL_SWAP`**

Swap terminals in CMUX manual mode

enumerator **`ESP_MODEM_MODE_CMUX_MANUAL_DATA`**

Set DATA mode in CMUX manual mode

enumerator **`ESP_MODEM_MODE_CMUX_MANUAL_COMMAND`**

Set COMMAND mode in CMUX manual mode

enumerator **`ESP_MODEM_MODE_DETECT`**

Detect the mode and resume it (if sucessfully detected)

enumerator **`ESP_MODEM_MODE_UNDEF`**

enum `esp_modem_dce_device`

DCE devices: Enum list of supported devices.

Values:

enumerator `ESP_MODEM_DCE_GENERIC`

The most generic device

enumerator `ESP_MODEM_DCE_SIM7600`

enumerator `ESP_MODEM_DCE_SIM7070`

enumerator `ESP_MODEM_DCE_SIM7000`

enumerator `ESP_MODEM_DCE_BG96`

enumerator `ESP_MODEM_DCE_EC20`

enumerator `ESP_MODEM_DCE_SIM800`

enumerator `ESP_MODEM_DCE_CUSTOM`

enum `esp_modem_terminal_error`

Terminal errors.

Values:

enumerator `ESP_MODEM_TERMINAL_BUFFER_OVERFLOW`

enumerator `ESP_MODEM_TERMINAL_CHECKSUM_ERROR`

enumerator `ESP_MODEM_TERMINAL_UNEXPECTED_CONTROL_FLOW`

enumerator `ESP_MODEM_TERMINAL_DEVICE_GONE`

enumerator `ESP_MODEM_TERMINAL_UNKNOWN_ERROR`

Functions

esp_modem_dce_t *`esp_modem_new`(*const* esp_modem_dte_config_t *dte_config, *const* esp_modem_dce_config_t *dce_config, esp_netif_t *netif)

Create a generic DCE handle for new modem API.

Parameters:
- **dte_config** – DTE configuration (UART config for now)
- **dce_config** – DCE configuration
- **netif** – Network interface handle for the data mode

Returns: DCE pointer on success, NULL on failure

esp_modem_dce_t ***esp_modem_new_dev**(esp_modem_dce_device_t module, *const* esp_modem_dte_config_t *dte_config, *const* esp_modem_dce_config_t *dce_config, esp_netif_t *netif)

Create a DCE handle using the supplied device.

Parameters:
- **module** – Specific device for creating this DCE
- **dte_config** – DTE configuration (UART config for now)
- **dce_config** – DCE configuration
- **netif** – Network interface handle for the data mode

Returns: DCE pointer on success, NULL on failure

void **esp_modem_destroy**(esp_modem_dce_t *dce)

Destroys modem's DCE handle.

Parameters: **dce** – DCE to destroy

esp_err_t **esp_modem_set_error_cb**(esp_modem_dce_t *dce, esp_modem_terminal_error_cbt err_cb)

Set DTE's error callback.

Parameters:
- **dce** – Modem DCE handle
- **err_cb** – **[in]** Error callback

Returns: ESP_OK on success, ESP_FAIL on failure

esp_err_t **esp_modem_set_mode**(esp_modem_dce_t *dce, esp_modem_dce_mode_t mode)

Set operation mode for this DCE.

Parameters:
- **dce** – Modem DCE handle
- **mode** – Desired MODE

Returns: ESP_OK on success, ESP_FAIL on failure

esp_err_t **esp_modem_command**(esp_modem_dce_t *dce, *const* char *command, esp_err_t (*got_line_cb)(uint8_t *data, size_t len), uint32_t timeout_ms)

Convenient function to run arbitrary commands from C-API.

Parameters:
- **dce** – Modem DCE handle
- **command** – Command to send
- **got_line_cb** – Callback function which is called whenever we receive a line
- **timeout_ms** – Command timeout

Returns: ESP_OK on success, ESP_FAIL on failure

---

esp_err_t **`esp_modem_set_apn`**(esp_modem_dce_t *dce, *const* char *apn)

Sets the APN and configures it into the modem's PDP context.

Parameters:
- **dce** – Modem DCE handle
- **apn** – Access Point Name

Returns: ESP_OK on success

---

esp_err_t **`esp_modem_pause_net`**(esp_modem_dce_t *dce, bool pause)

This API provides support for temporarily pausing networking in order to send/receive AT commands and resume networking afterwards.

Note

This function does not switch modes, the modem is still in data mode.

Parameters:
- **dce** – Modem DCE handle
- **pause** – true to pause the network interface, false to resume networking

Returns: ESP_OK on success

---

esp_modem_dce_mode_t **`esp_modem_get_mode`**(esp_modem_dce_t *dce)

Modem commands

These functions are the actual commands to communicate with the modem using AT command interface.

Note that the functions which implement AT commands returning textual values use plain `char *` pointer as the return value. The API expects the output data to point to user allocated space of at least `CONFIG_ESP_MODEM_C_API_STR_MAX` (128 by default) bytes, it also truncates the output data to this size.

Functions

---

command_result **`esp_modem_sync`**()

Sends the initial AT sequence to sync up with the device.

Returns: OK, FAIL or TIMEOUT

---

command_result **`esp_modem_get_operator_name`**(char *name, int *act)

Reads the operator name.

Parameters:
- **name** – [out] operator name

- **act** – **[out]** access technology

Returns: OK, FAIL or TIMEOUT

---

command_result **`esp_modem_store_profile`**()

Stores current user profile.

Returns: OK, FAIL or TIMEOUT

---

command_result **`esp_modem_set_pin`**(*const* char *pin)

Sets the supplied PIN code.

Parameters: **pin** – **[in]** Pin

Returns: OK, FAIL or TIMEOUT

---

command_result **`esp_modem_at_raw`**(*const* char *cmd, char *out, *const* char *pass, *const* char *fail, int timeout)

Execute the supplied AT command in raw mode (doesn't append '\r' to command, returns everything)

Parameters:

- **cmd** – **[in]** String command that's send to DTE
- **out** – **[out]** Raw output from DTE
- **pass** – **[in]** Pattern in response for the API to return OK
- **fail** – **[in]** Pattern in response for the API to return FAIL
- **timeout** – **[in]** AT command timeout in milliseconds

Returns: OK, FAIL or TIMEOUT

---

command_result **`esp_modem_at`**(*const* char *cmd, char *out, int timeout)

Execute the supplied AT command.

Parameters:

- **cmd** – **[in]** AT command
- **out** – **[out]** Command output string
- **timeout** – **[in]** AT command timeout in milliseconds

Returns: OK, FAIL or TIMEOUT

---

command_result **`esp_modem_read_pin`**(bool *pin_ok)

Checks if the SIM needs a PIN.

Parameters: **pin_ok** – **[out]** true if the SIM card doesn't need a PIN to unlock

Returns: OK, FAIL or TIMEOUT

---

command_result **`esp_modem_set_echo`**(*const* bool echo_on)

Sets echo mode.

Parameters: **echo_on** – **[in]** true if echo mode on (repeats the commands)

Returns: OK, FAIL or TIMEOUT

command_result **`esp_modem_sms_txt_mode`**(*const* bool txt)

Sets the Txt or Pdu mode for SMS (only txt is supported)

Parameters: **txt** – **[in]** true if txt mode

Returns: OK, FAIL or TIMEOUT

command_result **`esp_modem_sms_character_set`**()

Sets the default (GSM) character set.

Returns: OK, FAIL or TIMEOUT

command_result **`esp_modem_send_sms`**(*const* char *number, *const* char *message)

Sends SMS message in txt mode.

Parameters:
- **number** – **[in]** Phone number to send the message to
- **message** – **[in]** Text message to be sent

Returns: OK, FAIL or TIMEOUT

command_result **`esp_modem_resume_data_mode`**()

Resumes data mode (Switches back to the data mode, which was temporarily suspended)

Returns: OK, FAIL or TIMEOUT

command_result **`esp_modem_set_pdp_context`**(esp_modem_PdpContext_t *p1)

Sets php context.

Parameters: **p1** – **[in]** PdP context struct to setup modem cellular connection

Returns: OK, FAIL or TIMEOUT

command_result **`esp_modem_set_command_mode`**()

Switches to the command mode.

Returns: OK, FAIL or TIMEOUT

command_result **`esp_modem_set_cmux`**()

Switches to the CMUX mode.

Returns: OK, FAIL or TIMEOUT

command_result **`esp_modem_get_imsi`**(char *imsi)

Reads the IMSI number.

Parameters: **imsi** – **[out]** Module’s IMSI number
Returns: OK, FAIL or TIMEOUT

command_result **esp_modem_get_imei**(char *imei)

Reads the IMEI number.

Parameters: **imei** – **[out]** Module’s IMEI number
Returns: OK, FAIL or TIMEOUT

command_result **esp_modem_get_module_name**(char *name)

Reads the module name.

Parameters: **name** – **[out]** module name
Returns: OK, FAIL or TIMEOUT

command_result **esp_modem_set_data_mode**()

Sets the modem to data mode.

Returns: OK, FAIL or TIMEOUT

command_result **esp_modem_get_signal_quality**(int *rssi, int *ber)

Get Signal quality.

Parameters:
- **rssi** – **[out]** signal strength indication
- **ber** – **[out]** channel bit error rate

Returns: OK, FAIL or TIMEOUT

command_result **esp_modem_set_flow_control**(int dce_flow, int dte_flow)

Sets HW control flow.

Parameters:
- **dce_flow** – **[in]** 0=none, 2=RTS hw flow control of DCE
- **dte_flow** – **[in]** 0=none, 2=CTS hw flow control of DTE

Returns: OK, FAIL or TIMEOUT

command_result **esp_modem_hang_up**()

Hangs up current data call.

Returns: OK, FAIL or TIMEOUT

command_result **esp_modem_get_battery_status**(int *voltage, int *bcs, int *bcl)

Get voltage levels of modem power up circuitry.

Parameters:
- **voltage** – **[out]** Current status in mV

- **bcs** – **[out]** charge status (-1-Not available, 0-Not charging, 1-Charging, 2-Charging done)
- **bcl** – **[out]** 1-100% battery capacity, -1-Not available

Returns: OK, FAIL or TIMEOUT

command_result `esp_modem_power_down`()

Power down the module.

Returns: OK, FAIL or TIMEOUT

command_result `esp_modem_reset`()

Reset the module.

Returns: OK, FAIL or TIMEOUT

command_result `esp_modem_set_baud`(int baud)

Configures the baudrate.

Parameters: **baud** – **[in]** Desired baud rate of the DTE

Returns: OK, FAIL or TIMEOUT

command_result `esp_modem_set_operator`(int mode, int format, *const* char *oper)

Force an attempt to connect to a specific operator.

Parameters:

- **mode** – **[in]** mode of attempt mode=0 - automatic mode=1 - manual mode=2 - deregister mode=3 - set format for read operation mode=4 - manual with fallback to automatic
- **format** – **[in]** what format the operator is given in format=0 - long format format=1 - short format format=2 - numeric
- **oper** – **[in]** the operator to connect to

Returns: OK, FAIL or TIMEOUT

command_result `esp_modem_set_network_attachment_state`(int state)

Attach or detach from the GPRS service.

Parameters: **state** – **[in]** 1-attach 0-detach

Returns: OK, FAIL or TIMEOUT

command_result `esp_modem_get_network_attachment_state`(int *state)

Get network attachment state.

Parameters: **state** – **[out]** 1-attached 0-detached

Returns: OK, FAIL or TIMEOUT

command_result **esp_modem_set_radio_state**(int state)

What mode the radio should be set to.

Parameters: **state** – **[in]** state 1-full 0-minimum ...

Returns: OK, FAIL or TIMEOUT

command_result **esp_modem_get_radio_state**(int *state)

Get current radio state.

Parameters: **state** – **[out]** 1-full 0-minimum ...

Returns: OK, FAIL or TIMEOUT

command_result **esp_modem_set_network_mode**(int mode)

Set network mode.

Parameters: **mode** – **[in]** preferred mode

Returns: OK, FAIL or TIMEOUT

command_result **esp_modem_set_preferred_mode**(int mode)

Preferred network mode (CAT-M and/or NB-IoT)

Parameters: **mode** – **[in]** preferred selection

Returns: OK, FAIL or TIMEOUT

command_result **esp_modem_set_network_bands**(*const* char *mode, *const* int *bands, int size)

Set network bands for CAT-M or NB-IoT.

Parameters:

- **mode** – **[in]** CAT-M or NB-IoT
- **bands** – **[in]** bitmap in hex representing bands
- **size** – **[in]** size of teh bands bitmap

Returns: OK, FAIL or TIMEOUT

command_result **esp_modem_get_network_system_mode**(int *mode)

Show network system mode.

Parameters: **mode** – **[out]** current network mode

Returns: OK, FAIL or TIMEOUT

command_result **esp_modem_set_gnss_power_mode**(int mode)

GNSS power control.

Parameters: **mode** – **[out]** power mode (0 - off, 1 - on)

Returns: OK, FAIL or TIMEOUT

command_result **`esp_modem_get_gnss_power_mode`**(int *mode)

GNSS power control.

Parameters: **mode** – **[out]** power mode (0 - off, 1 - on)

Returns: OK, FAIL or TIMEOUT

Configuration structures

group ESP_MODEM_CONFIG

Configuration structures for DTE and DCE.

Defines

`ESP_MODEM_DEFAULT_UART_CLK`

`ESP_MODEM_DTE_DEFAULT_CONFIG`()

ESP Modem DTE Default Configuration.

`ESP_MODEM_DCE_DEFAULT_CONFIG`(APN)

ESP Modem DCE Default Configuration.

Typedefs

typedef struct esp_modem_dte_config **`esp_modem_dte_config_t`**

typedef struct esp_modem_dce_config **`esp_modem_dce_config_t`**

Enums

enum **`esp_modem_flow_ctrl_t`**

Modem flow control type.

Values:

enumerator **`ESP_MODEM_FLOW_CONTROL_NONE`**

enumerator **`ESP_MODEM_FLOW_CONTROL_SW`**

enumerator **`ESP_MODEM_FLOW_CONTROL_HW`**

struct **`esp_modem_uart_term_config`**

#include <esp_modem_config.h>
UART configuration structure.

Public Members

uart_port_t **`port_num`**

UART port number

uart_word_length_t **`data_bits`**

Data bits of UART

uart_stop_bits_t **`stop_bits`**

Stop bits of UART

uart_parity_t **`parity`**

Parity type

esp_modem_flow_ctrl_t **`flow_control`**

Flow control type

uart_sclk_t **`source_clk`**

UART source clock selection

int **`baud_rate`**

Communication baud rate

int **`tx_io_num`**

TXD Pin Number

int **`rx_io_num`**

RXD Pin Number

int **`rts_io_num`**

RTS Pin Number

int **`cts_io_num`**

CTS Pin Number

int **`rx_buffer_size`**

UART RX Buffer Size

int **`tx_buffer_size`**

UART TX Buffer Size

int **`event_queue_size`**

UART Event Queue Size, set to 0 if no event queue needed

struct **`esp_modem_vfs_term_config`**

#include <esp_modem_config.h>
VFS configuration structure.

Public Members

int **`fd`**

Already created file descriptor

void (***`deleter`**)(int, *struct* esp_modem_vfs_resource*)

Custom close function for the fd

struct esp_modem_vfs_resource ***`resource`**

Resource attached to the VFS (need for clenaup)

struct **`esp_modem_dte_config`**

#include <esp_modem_config.h>
Complete DTE configuration structure.

Note that the generic part is common for DTE and its SW resources The following portions for config is dedicated to the chosen HW resource used as a communication terminal for this DTE

Public Members

size_t **`dte_buffer_size`**

DTE buffer size

uint32_t **`task_stack_size`**

Terminal task stack size

unsigned **`task_priority`**

Terminal task priority

struct esp_modem_uart_term_config **`uart_config`**

Configuration for UART Terminal

struct esp_modem_vfs_term_config **`vfs_config`**

Configuration for VFS Terminal

void *__`extension_config`__

Configuration for app specific Terminal

struct **`esp_modem_dce_config`**

#include <esp_modem_dce_config.h>

DCE configuration structure.

Public Members

const char *__`apn`__

APN: Logical name of the Access point