

# CMD 库数据手册

修订原因

| 版本   | 日期         | 原因                   |
|------|------------|----------------------|
| V1.0 | 2025/12/16 | 首版发布                 |
| V2.0 | 2025/12/25 | 更名为 CMD, 并且可以实现多实例框架 |

# 1. CMD 库概述

## 1.1. 库简介

[CMD 库] 是一款专为嵌入式设备设计的命令交互开发库，采用分层架构设计，封装了硬件接口、数据收发、网络连接、协议交互等核心功能，旨在简化基于命令交互的模块（如 4G、DTU、WiFi 模块）开发流程，降低开发难度，提高项目复用性和稳定性。

批注：带“选配”字样的参数可以传入 NULL，内部自动适配；

## 1.2. 特性

- 分层解耦：核心层、网络层、协议层独立划分，适配不同硬件平台和协议需求；
- 核心层：自定义缓冲区大小，内嵌接收帧队列+ DMA 环形缓冲区处理，串行口接收超时状态机，帮助开发者管理底层数据通道，无任何中断资源占用，支持多实例；
- 网络层：自定义网络指令，自定义校验，可在运行过程中灵活添加(删除)网络指令，可自定义起始连接指令，内嵌网络状态机，支持多实例；
- 协议层：自定义协议指令，自定义解析，自动搜寻协议指令，自动发送协议指令，可发送超长数据包，支持连包处理，多包头处理，内嵌接收状态机，发送状态机，任务队列，支持多实例；

## 1.3. 适用场景

- 嵌入式设备与指令模块的通信开发（4G, DTU 等）；
- 基于命令指令交互的网络连接、数据上报、远程控制等场景；
- 需自定义协议与云端 / 服务器交互的嵌入式项目。

## 1.4. 依赖环境

- 硬件环境：支持 DMA、定时器、通信总线，的嵌入式 MCU；
- 软件环境：支持标准 C 语言的嵌入式开发环境（如 Keil MDK、IAR、GCC 等）；
- 底层依赖：需用户实现硬件外设（通信总线、DMA, MODULE）的初始化代码。

## 2. CMD 库架构

2.1.库采用自上而下的分层架构，各层职责明确，层间通过接口交互，实现解耦：

| 层级  | 核心职责                           | 依赖关系 | 对外头文件         |
|-----|--------------------------------|------|---------------|
| 核心层 | 接收帧队列 + DMA 环形缓冲区管理、状态机扫描等底层功能 | /    | AT_Core.h     |
| 网络层 | 管理模块联网状态，实现联网流程控制、指令交互         | 核心层  | AT_Network.h  |
| 协议层 | 支持自定义协议指令的构建、发送队列、解析与校验        | 核心层  | AT_Protocol.h |

(1) 每一层都有一个句柄，用户需要通过接口函数去注册句柄，根据接口函数的返回值判断句柄是否注册成功；

(2) 核心层句柄：该层相当于一条 MCU 与 Module 交互的通道，后续网络层，协议层都是基于这条通道进行交互；

(3) 网络层句柄：该层是基于核心层句柄进行网络数据交互，故网络层句柄会包含核心层句柄；

(4) 协议层句柄：该层是基于核心层句柄进行协议数据交互，故协议层句柄会包含核心层句柄；

(5) 由上所述，网络层，协议层并不需要知道在哪交互数据，只做自身该做的任务，所以用户可以创建一个网络层句柄用于 TCP 连接，另一个网络层句柄用于 MQTT 连接，然后在调用网络层接口函数的时候，传入不同的网络层句柄，实现不同平台的交互，如果用户模块可以保存链路，可以实现单模块同事连接多平台

## 3. 硬件层简介

### 3.1. 库内依赖的硬件资源

(1) 本库依赖 DMA 控制器, 下面给出初始化流程;

|          |                      |
|----------|----------------------|
| 外设基地址:   | 与模块通信的接收             |
| 存储器基地址:  | 用户自行定义的缓冲区           |
| 外设数据宽度:  | 由通信接口的数据帧格式决定        |
| 存储器数据宽度: | 由通信接口的数据帧格式决定        |
| 外设地址增量:  | 由通信接口的数据寄存器决定(一般不增量) |
| 存储器地址增量: | 一般为增量(除非用户知道自己在做什么)  |
| 传输方向:    | 外设---存储器             |
| 传输计数器:   | DMA(部分芯片可能会有这部分电路)   |
| 传输计数器值:  | 根据用户定义的缓冲区决定         |
| 循环模式:    | 开启                   |
| 存储器切换:   | 关闭                   |
| 通道优先级:   | 用户根据项目要求决定           |

(2) 本库的核心层 (CMD\_Core.c) 依赖 DMA 目的缓冲区的指针, 还需要用户自行定义一个和 DMA 目的缓冲区大小一致的线性缓冲区, 防止数据断帧, 同时还需要用户提供 DMA 通道传输计数器获取的函数;

(3) 本库的网络层只依赖核心层句柄;

(4) 本库的协议层依赖核心层句柄, 同时还依赖 2 个发送函数, 如下

格式符发送函数: 要求此函数可以发送格式符数据, 用在发布阶段, 例如移远的 "AT+QISEND=0,%d\r\n", 用于发送格式符数据(DTU 模块可不用);

定长数据发送函数: 要求此函数可以发送定长数据, 所以这个函数最起码要有俩个参数, 一个是地址指针, 一个是长度;

批注: 如果用户要使用的模块可以不需要发布功能, 那么只需要给格式符发送函数传入

NULL 即可，内部自动适配，用户可以在协议层头文件看到一些选配参数;

## 4. CMD\_Core.h 核心层简介

### 4.1.核心层快速使用说明

- (1) 用户自行声明一个 CMD\_Core\_Handle\_t 类型的全局变量;
- (2) 用户自行声明一个 CMD\_Core\_SerialParam\_t 类型的变量，对这个结构体变量里的成员进行配置;
- (3) 调用 CMD\_Core\_SerialHandle\_Create 函数，将 (1), (2) 处的变量传递进行，根据返回值判断句柄创建成功还是失败(返回值查看函数说明);
- (4) 将 CMD\_Core\_SerialRecvTimeout\_Task 此函数放入周期 1ms(具体由用户决定)的任务区，传入注册成功的核心层句柄，并校验该函数的返回值 (返回值说明在头文件，或者本文档的函数说明处);
- (5) 1 个核心层句柄对应 1 块硬件，你可以创建多个网络层句柄，用于不同场景，但都依赖 1 块硬件(1 个核心层句柄);

### 4.2.类型简介

#### 4.2.01. CMD\_Core\_Handle\_t 句柄

typedef void \* CMD\_Core\_Handle\_t; 核心层句柄;

#### 4.2.02. CMD\_Core\_State\_t 枚举类型

|                        |         |
|------------------------|---------|
| CMD_CORE_OKCMD_CORE_OK | 通用的成功状态 |
|------------------------|---------|

|                     |      |
|---------------------|------|
| CMD_CORE_ERR_PARAM  | 错误参数 |
| CMD_CORE_ERR_FREE   | 重复删除 |
| CMD_CORE_STATE_FULL | 状态满  |

该类型是核心层所有接口函数的返回值;

#### 4.2.03. CMD\_Core\_SerialParam\_t 结构体类型

|                               |                                   |
|-------------------------------|-----------------------------------|
| char *recv_buf                | DMA 目的缓冲区首地址指针                    |
| char *recv_line_buf           | DMA 环形缓冲区首地址指针                    |
| uint16_t recv_buf_size        | DMA 搬运目的缓冲区, 环形缓冲区长度(这俩个缓冲区大小需一致) |
| uint32_t scan_threshold       | 串行口接收超时扫描阈值                       |
| uint32_t<br>timeout_threshold | 串行口接收超时阈值                         |

该类型用在创建核心层句柄函数中;

### 4.3.函数简介

4.3.01. CMD\_Core\_State\_t **CMD\_Core\_SerialHandle\_Create**(CMD\_Core\_Handle\_t \*handle, CMD\_Core\_SerialParam\_t \*cfg);

|                           |  |
|---------------------------|--|
| CMD_Core_State_t          | CMD_CORE_OK<br>CMD_CORE_ERR_PARAM<br>CMD_CORE_STATE_FULL |
| CMD_Core_Handle_t *handle | CMD_Core_Handle_t 类型的指针                                  |

|                             |                              |
|-----------------------------|------------------------------|
| CMD_Core_SerialParam_t *cfg | CMD_Core_SerialParam_t 类型的指针 |
|-----------------------------|------------------------------|

核心层句柄创建函数;

4.3.02. CMD\_Core\_State\_t **CMD\_Core\_SerialHandle\_Delete**(CMD\_Core\_Handle\_t \*handle);

|                           |  |
|---------------------------|--|
| CMD_Core_State_t          | CMD_CORE_OK<br>CMD_CORE_ERR_PARAM<br>CMD_CORE_ERR_FREE |
| CMD_Core_Handle_t *handle | CMD_Core_Handle_t 类型的指针                                |

核心层句柄删除函数;

4.3.03.CMD\_Core\_State\_t

**CMD\_Core\_SerialRecvTimeout\_Task**(CMD\_Core\_Handle\_t handle);

|                          |   |
|--------------------------|---|
| CMD_Core_State_t         | CMD_CORE_OK<br>CMD_CORE_ERR_PARAM<br>CMD_CORE_STATE_FULL(接收帧队列满会返回) |
| CMD_Core_Handle_t handle | CMD_Core_Handle_t 类型的变量   |

核心层串行口接收超时任务;

#### 4.4.宏说明

|                                       |                  |
|---------------------------------------|------------------|
| CMD_CORE_SERIAL_MEMPOOL_ELEMENT_COUNT | 核心层句柄数量<br>默认 3  |
| CMD_CORE_RECV_FRAME_QUEUE_COUNT       | 接收帧队列长度<br>默认 15 |

特殊需求用户, 将该宏填写好后, 发给库开发者, 进行资源分配;

## 5. CMD\_Ntwork.h 网络层简介

### 5.1.网络层快速使用说明

- (1) 用户自行声明一个 CMD\_Network\_Handle\_t 类型的全局变量;
- (2) 用户自行声明一个 CMD\_Network\_Param\_t 类型的变量, 对这个结构体变量里的成员进行赋值;
- (3) 调用 CMD\_Network\_Handle\_Create 函数, 将 (1), (2) 处的变量传递进行, 根据返回值判断句柄创建成功还是失败(返回值查看函数说明);
- (4) 用户自行创建一个发送命令函数, 如下  

```
void TT_Send_BEAT(void) { TT_HW_Serial_SendString("AT\r\n"); }
```

该函数内, 只做发送的任务;
- (5) 用户自行创建一个接收校验函数, 如下  

```
char *TT_Check_BEAT(char *frame) { return strstr(frame, "OK"); }
```

该函数内, 只做校验的任务, frame 是每一帧数据的起始地址指针, 用户可直接用例使用, 该函数内校验成功返回 !NULL, 校验失败返回 NULL 即可;
- (6) 调用 CMD\_Network\_Directive\_Add 函数, 将 (3) 处创建成功的网络层句柄, 网络指令唯一标识(建议用户使用枚举管理), (4), (5) 处的变量传入, 根据返回值判断是否添加成功(返回值查看函数说明);
- (7) 用户根据需求添加相应的指令数量, 每个句柄默认支持 40 条指令, 若已添加过标识 1, 用户不可在添加指令的时候再次传入 1, 需要先调用删除指令函数, 在重新添加;



- (8) 用户添加完网络指令后, 调用 `CMD_Network_StartLinkDirective_Set` 函数, 传入 (3) 处创建成功的网络层句柄, 网络指令唯一标识, 设置起始连接指令;
- (9) 网络指令的执行顺序是根据用户添加的顺序进行的, 不过用户可以调用 `CMD_Network_CurrentLinkDirective_Set` 设置当前网络连接指令;
- (10) 定时调用 `CMD_Network_Task` 函数, 需传入对应的网络层句柄, 由该函数扫描周期和 `CMD_Network_Param_t` 中的 `speed_threshold` 一同决定, 建议 300ms 以上;
- (11) 用户如果想实现多个网络任务, 可能在回到步骤 1, 创建新的网络层句柄, 实现多实例;

## 5.2.类型简介

### 5.2.01. `CMD_Network_Handle_t` 句柄

`typedef void * CMD_Network_Handle_t;` 网络层句柄;

### 5.2.02. `CMD_Network_State_t` 枚举类型

|                                      |          |
|--------------------------------------|----------|
| <code>CMD_NETWORK_OK</code>          | 通用的成功状态  |
| <code>CMD_NETWORK_ERR_PARAM</code>   | 错误参数     |
| <code>CMD_NETWORK_ERR_FREE</code>    | 重复删除     |
| <code>CMD_NETWORK_ERR_REPEAT</code>  | 指令唯一标识重复 |
| <code>CMD_NETWORK_ERR_INDEX</code>   | 错误的唯一标识  |
| <code>CMD_NETWORK_ERR_TIMEOUT</code> | 错误超时     |
| <code>CMD_NETWORK_STATE_FULL</code>  | 状态满      |

该类似是网络层所有接口函数的返回值;

5.2.03. CMD\_Network\_Param\_t 结构体

|                               |              |
|-------------------------------|--------------|
| CMD_Core_Handle_t core_handle | 网络层使用到的核心层句柄 |
| uint32_t speed_threshold      | 联网速度阈值       |
| uint8_t retry_threshold       | 重试计数器阈值      |

该类型用在创建网络层句柄函数中，其中阈值的周期都是 CMD\_Network\_Task 函数的周期;

5.3.函数简介

5.3.01. CMD\_Network\_State\_t

CMD\_Network\_Handle\_Create(CMD\_Network\_Handle\_t \*handle,  
CMD\_Network\_Param\_t \*cfg);

|                              |   |
|------------------------------|---|
| CMD_Network_State_t          | CMD_NETWORK_OK<br>CMD_NETWORK_ERR_PARAM<br>CMD_NETWORK_STATE_FULL |
| CMD_Network_Handle_t *handle | CMD_Network_Handle_t 类型的指针  |
| CMD_Network_Param_t *cfg     | CMD_Network_Param_t 类型的指针   |

网络层句柄创建函数;

5.3.02. CMD\_Network\_State\_t

CMD\_Network\_Handle\_Delete(CMD\_Network\_Handle\_t \*handle);

|                     |   |
|---------------------|---|
| CMD_Network_State_t | CMD_NETWORK_OK<br>CMD_NETWORK_ERR_PARAM |
|---------------------|---|

|                              |                            |
|------------------------------|----------------------------|
|                              | CMD_NETWORK_ERR_FREE       |
| CMD_Network_Handle_t *handle | CMD_Network_Handle_t 类型的指针 |

网络层句柄删除函数;

5.3.03. CMD\_Network\_State\_t **CMD\_Network\_Task**(CMD\_Network\_Handle\_t handle);

|                             |   |
|-----------------------------|---|
| CMD_Network_State_t         | CMD_NETWORK_OK<br>CMD_NETWORK_ERR_TIMEOUT |
| CMD_Network_Handle_t handle | CMD_Network_Handle_t 类型                   |

网络层任务函数;

5.3.04. CMD\_Network\_State\_t

**CMD\_Network\_Directive\_Add**(CMD\_Network\_Handle\_t handle, uint32\_t directive\_index, void(\*send\_command)(void), char \*(recv\_command)(void));

|                                  |   |
|----------------------------------|---|
| CMD_Network_State_t              | CMD_NETWORK_OK<br>CMD_NETWORK_ERR_PARAM<br>CMD_NETWORK_ERR_REPEAT<br>CMD_NETWORK_STATE_FULL |
| CMD_Network_Handle_t handle      | CMD_Network_Handle_t 类型   |
| uint32_t directive_logo          | 网络指令唯一标识  |
| void(*send_func)(void)           | 用户创建的发送网络指令   |
| char *(*check_func)(char *frame) | 用户创建的校验网络指令函数, 其中 frame 是每一帧数据的起始地址指针, 用  |

|  |           |
|--|-----------|
|  | 户可以直接用来使用 |
|--|-----------|

网络指令添加函数;

5.3.05. CMD\_Network\_State\_t

**CMD\_Network\_Directive\_Delete**(CMD\_Network\_Handle\_t handle, uint32\_t directive\_index);

|                             |  |
|-----------------------------|--|
| CMD_Network_State_t         | CMD_NETWORK_OK<br>CMD_NETWORK_ERR_PARAM<br>CMD_NETWORK_ERR_INDEX |
| CMD_Network_Handle_t handle | CMD_Network_Handle_t 类型  |
| uint32_t directive_index    | 网络指令唯一标识   |

网络指令删除函数;

5.3.06. CMD\_Network\_State\_t

**CMD\_Network\_StartLinkDirective\_Set**(CMD\_Network\_Handle\_t handle, uint32\_t start);

|                             |   |
|-----------------------------|---|
| CMD_Network_State_t         | CMD_NETWORK_OK<br>CMD_NETWORK_ERR_INDEX |
| CMD_Network_Handle_t handle | CMD_Network_Handle_t 类型                 |
| uint32_t start              | 网络指令唯一标识                                |

网络起始连接指令设置;

5.3.07. CMD\_Network\_State\_t

```
CMD_Network_CurrentLinkDirective_Set(CMD_Network_Handle_t handle,  
  
uint32_t current);
```

|                             |   |
|-----------------------------|---|
| CMD_Network_State_t         | CMD_NETWORK_OK<br>CMD_NETWORK_ERR_INDEX |
| CMD_Network_Handle_t handle | CMD_Network_Handle_t 类型                 |
| uint32_t current            | 网络指令唯一标识                                |

网络当前连接指令设置;

5.4.宏简介

|  |                            |
|--|----------------------------|
| CMD_NETWORK_MEMPOOL_ELEMENT_COUNT      | 网络层句柄数量<br>默认: 3           |
| CMD_NETWORK_MENU_MEMPOOL_ELEMENT_COUNT | 每个句柄最多支持几条<br>指令<br>默认: 40 |

特殊需求用户, 将该宏填写好后, 发给库开发者, 进行资源分配;

6. CMD\_Protocol.h 协议层简介

6.1.协议层快速使用说明

- (1) 用户自行声明一个 CMD\_Protocol\_Handle\_t 类型的全局变量;
- (2) 用户自行声明一个 CMD\_Protocol\_Param\_t 类型的变量, 对这个结构体变量里的成员进行赋值;

(3) 调用 `CMD_Protocol_Handle_Create` 函数, 将 (1), (2) 处的变量传递进行, 根据返回值判断句柄创建成功还是失败(返回值查看函数说明);

(4) 用户自行创建一个构建协议指令数据包函数, 如下

```
void TT_Login_Build(CMD_Protocol_SendQueue_Arg_t assign_data)
```

```
{
```

```
...构建代码
```

将数据包起始地址,长度传给发送任务状态机,构建的任务就完成了(该值不可以是局部的);

```
CMD_Protocol_SendPacket_Write(数据包起始地址指针, 数据包长度);
```

```
}
```

注意: `assin_data` 是用户调用 协议发送队列入队函数中的第 3 个参数

(5) 用户自行创建一个接收协议指令, 解析函数, 如下

```
char *TT_Login_Recv(char * directive_tag_addr) { 解析代码.. return 成功或者失败; } "选配"
```

注意: `directive_tag_addr` 是 `CMD_Protocol_Directive_Add` 函数的第 5 个参数  
`directive_tag` 在缓冲区中首次出现的地址指针,每次执行到用户的解析函数就是代表出现了一次 `directive_tag`, 假设出现 3 次相同的 `directive_tag`, 那么库内会连续调用 3 次用户的解析函数, 用户需要防止数据覆盖;

(6) 调用 `CMD_Protocol_Directive_Add` 函数, 传入 (3) 处创建成功的协议层句柄, 指令唯一索引(相同的索引可以直接覆盖), (4), (5) 处变量, `directive_tag` 是指令标签, 例如 `EBCharge#1#...` 这条指令, 那么你可以传入 `"EBCharge"`, 根据返回值判断是否添加成功(返回值查看函数说明);

(7) 主循环调用 `CMD_Protocol_Task` 函数, 传入对应的协议层句柄, 用户可以根据该函数的返回值判断模块是否发送超时(返回值查看函数说明);

(8) 如果需要执行发送的任务, 只需要调用 `CMD_Protocol_SendQueue_Enqueue` 函数,

传入对应的协议层句柄, 指令唯一标识, 指定数据, 用户可以指定端口等, 返回值为 OK 则代表入队成功, 内部自动发送(返回值查看函数说明);

(9) 如果用户想要创建多个协议任务, 可以在回到步骤 1, 创建一个新的协议层句柄, 实现多实例;

6.2.类型简介

6.2.01. CMD\_Protocol\_Handle\_t 句柄

typedef void \* CMD\_Protocol\_Handle\_t; 协议层句柄;

6.2.02. CMD\_Protocol\_State\_t 枚举类型

|                          |         |
|--------------------------|---------|
| CMD_PROTOCOL_OK          | 通用的成功状态 |
| CMD_PROTOCOL_ERR_PARAM   | 错误参数    |
| CMD_PROTOCOL_ERR_FREE    | 重复删除    |
| CMD_PROTOCOL_ERR_TIMEOUT | 错误超时    |
| CMD_PROTOCOL_STATE_FULL  | 状态满     |
| CMD_PROTOCOL_STATE_IDLE  | 状态空闲    |
| CMD_PROTOCOL_STATE_BUSY  | 状态忙     |

该类型是协议层所有接口函数的返回值;

6.2.03. CMD\_Protocol\_Param\_t 结构体类型

|                                  |              |
|----------------------------------|--------------|
| CMD_Core_Handle_t<br>core_handle | 协议层使用到的核心层句柄 |
|----------------------------------|--------------|

|  |   |
|--|---|
| uint32_t retry_threshold   | 协议指令发送超时阈值，周期是<br>CMD_Protocol_Task 函数的执行速度   |
| uint8_t packet_head_switch   | 1.当服务器下发的指令长度会超过模块接收上限<br>2.模块是以<CR><LF>响应<br>满足上述俩种条件, packet_head_switch = 1,<br>否则 packet_head_switch = 0                        |
| const char *module_rcv_urc   | 1.当服务器下发的指令长度会超过模块接收上限<br>2.模块是以<CR><LF>响应<br>满足上述俩种条件,<br>module_rcv_urc = 消息头, 例如 "+QIURC: \\"rcv\\""<br>否则 module_rcv_urc = NULL |
| const char<br>*module_send_done  | 模块将用户的数据发送完成后响应的数据, 例如<br>模块返回的 "SEND OK"   |
| uint16_t<br>module_send_max_len  | 模块单次发送的最大数据量  |
| void<br>(*module_publish_func)<br>(const char *string, uint32_t<br>length) | 发送定长数据函数  |
| const char<br>*module_publish_cmd  | 模块发送数据前的发布指令, 例如<br>"AT+QISEND=0,%d\r\n", "选配"  |
| const char<br>*module_publish_prompt                                       | 模块内部发送窗口打开响应的数据, 例如 用户发<br>送完 "AT+QISEND=0,%d\r\n", 模块响应 '>', "选<br>配"  |
| void (*module_publish_func)<br>(const char *string, ...)                   | 格式符函数, 内部用来发送格式符数据, 例如<br>MQTT 主题, "选配"   |
| void (*user_func)<br>(char *frame);  | 用户可自定义处理数据的函数, 参数 frame 是每<br>一帧数据的起始地址指针 "选配"  |

该函数用在协议层创建句柄函数中, 带 "选配" 的参数可以传入 NULL, 内部自动适配;



#### 6.2.04. CMD\_Protocol\_SendQueue\_Arg\_t 联合体

|                           |            |
|---------------------------|------------|
| uint32_t value_u32        | 无符号 32 位整型 |
| float value_f             | 浮点数        |
| void *value_ptr           | 指针         |
| void (*value_func) (void) | 函数指针       |

使用方法:

- (1) 先声明一个联合体变量: `CMD_Protocol_SendQueue_Arg_t assign_data;`
- (2) 对其中一个成员进行操作: `assign_data.value_u32 = 1;`
- (3) 在发送队列入队: `CMD_Protocol_SendQueue_Enqueue(协议层句柄, 协议菜单中对应的指令标识, assign_data);`
- (4) 这个 `assign_data` 会传递到用户的 `build_func` 函数中, 它可以是指定端口, 也可以是其他类型的数据;
- (5) 用户在 `build_func` 函数中可以直接 `assign_data.value_u32` 使用, 也可以先提前到一个变量中, `uint32_t port_bit = assign_data.value_u32`, 如果你的构造函数无需而外参数, 那么直接传递 `assign_data` 即可;
- (6) 如果传递的是指针类型, 用户需要注意生命周期, 因为不是调用了发送队列入队函数就立马开始发送的;

注意: 结合体的成员共用一片内存区域, 假设你在传递的时候使用的是 `value_u32`, 那么你在你的构造函数中也必须使用 `value_u32`, 如果你在构造函数中使用了 `value_f`, `value_ptr` 会发生未定义行为;

6.3.函数简介

6.3.01. CMD\_Protocol\_State\_t

CMD\_Protocol\_Handle\_Create(CMD\_Protocol\_Handle\_t \*handle,  
CMD\_Protocol\_Param\_t \*cfg);

|                               |  |
|-------------------------------|--|
| CMD_Protocol_State_t          | CMD_PROTOCOL_OK<br>CMD_PROTOCOL_ERR_PARAM<br>CMD_PROTOCOL_STATE_FULL |
| CMD_Protocol_Handle_t *handle | CMD_Protocol_Handle_t 类型的指针  |
| CMD_Protocol_Param_t *cfg     | CMD_Protocol_Param_t 类型的指针   |

协议层句柄创建函数;

6.3.02. CMD\_Protocol\_State\_t

CMD\_Protocol\_Handle\_Delete(CMD\_Protocol\_Handle\_t \*handle);

|                               |  |
|-------------------------------|--|
| CMD_Protocol_State_t          | CMD_PROTOCOL_OK<br>CMD_PROTOCOL_ERR_PARAM<br>CMD_PROTOCOL_ERR_FREE |
| CMD_Protocol_Handle_t *handle | CMD_Protocol_Handle_t 类型的指针  |

协议层句柄删除函数;

6.3.03. CMD\_Protocol\_State\_t CMD\_Protocol\_Task(CMD\_Protocol\_Handle\_t  
handle);

|                      |  |
|----------------------|--|
| CMD_Protocol_State_t | CMD_PROTOCOL_OK<br>CMD_PROTOCOL_ERR_PARAM<br>CMD_PROTOCOL_STATE_BUSY |
|----------------------|--|

|                              |   |
|------------------------------|---|
|                              | CMD_PROTOCOL_STATE_IDLE<br>CMD_PROTOCOL_ERR_TIMEOUT |
| CMD_Protocol_Handle_t handle | CMD_Protocol_Handle_t 类型                            |

协议层任务函数, 可以传入不同的协议层句柄执行不同的任务;

#### 6..3.04. CMD\_Protocol\_State\_t

**CMD\_Protocol\_Directive\_Add**(CMD\_Protocol\_Handle\_t handle, uint32\_t directive\_idx, void(\*build\_func)(uint8\_t assign\_data), char \*(\*parse\_func)(char \* directive\_tag\_addr), char \*directive\_tag);

|  |  |
|--|--|
| CMD_Protocol_State_t   | CMD_PROTOCOL_OK<br>CMD_PROTOCOL_ERR_PARAM<br>CMD_PROTOCOL_STATE_FULL |
| CMD_Protocol_Handle_t handle                                     | CMD_Protocol_Handle_t 类型   |
| uint32_t directive_idx   | 指令唯一标识(协议层的指令唯一标识是索引, 所以可以直接覆盖)                                      |
| void(*build_func)<br>( CMD_Protocol_SendQueue_Arg_t assign_data) | 用户的构建协议指令数据包函数, assign_data 是发送队列入队函数中 用户传入的第 3 个参数;                 |
| char *(*parse_func)(char *directive_tag_addr)                    | 用户的接收解析函数, directive_tag_addr 是 directive_tag 在缓冲区中第一次出现的地方          |
| char *directive_tag  | 每条指令的唯一标识, 例如 “EBCharge”;  |

协议指令添加函数;

6.3.05. CMD\_Protocol\_State\_t

**CMD\_Protocol\_SendQueue\_Enqueue**(CMD\_Protocol\_Handle\_t handle, uint32\_t directive\_idx, CMD\_Protocol\_SendQueue\_Arg\_t assign\_data);

|  |  |
|--|--|
| CMD_Protocol_State_t                     | CMD_PROTOCOL_OK<br>CMD_PROTOCOL_ERR_PARAM<br>CMD_PROTOCOL_STATE_FULL |
| CMD_Protocol_Handle_t handle             | CMD_Protocol_Handle_t 类型   |
| uint32_t directive_idx                   | 指令唯一索引   |
| CMD_Protocol_SendQueue_Arg_t assign_data | 指定数据, 可以是端口等(详见类型说明)   |

协议发送队列入队函数;

6.3.06. CMD\_Protocol\_State\_t

**CMD\_Protocol\_SendPacket\_Write**(CMD\_Protocol\_Handle\_t handle, char \*packet\_addr, uint32\_t packet\_len);

|                              |   |
|------------------------------|---|
| CMD_Protocol_State_t         | CMD_PROTOCOL_OK<br>CMD_PROTOCOL_ERR_PARAM |
| CMD_Protocol_Handle_t handle | CMD_Protocol_Handle_t 类型                  |
| char *packet_addr            | 数据包起始地址指针                                 |
| uint32_t packet_len          | 数据包长度                                     |

协议发送数据包写入函数, 该函数在用户的构建协议指令函数中使用, 构建完协议指令数据包后, 调用;

6.4.宏简介

|                                    |                       |
|------------------------------------|-----------------------|
| CMD_PROTOCOL_MEMPOOL_ELEMENT_COUNT | 协议层句柄数量<br>默认: 3      |
| CMD_PROTOCOL_MENU_ELEMENT_COUNT    | 每个句柄最大支持多少条协议指令默认: 15 |
| CMD_PROTOCOL_LINK_BAG_MAX          | 每个句柄协议连包处理最大次数默认: 3   |
| CMD_PROTOCOL_SEND_TASK_QUEUE_MAX   | 每个句柄协议发送任务队列大小默认: 15  |

特殊需求用户将该宏填写好后，发给库开发者进行资源分配;

7. 多实例快速使用说明

批注: 多实例场景下，用户需要自行管理句柄，因为调用每个接口函数都需要传入对应的句柄;

7.1.单模块多实例说明

- (1) 创建好 A 模块的核心层句柄;
- (2) 在 CMD\_Network\_Param\_t 类型中配置好参数，基与(1)核心层句柄创建一个 socket 网络层句柄，然后再调用接口函数添加网络指令，添加完成后，传 socket 句柄给网络层任务函数，定时调用即可;
- (3) 在 CMD\_Network\_Param\_t 类型中配置好参数，基与(1)核心层句柄创建一个 mqtt 网络层句柄，然后再调用接口函数添加网络指令，添加完成后，传 mqtt 句柄给网络层任务函数，定时调用即可;

- (4) 这种基于单模块的多实例，俩个不同的网络层句柄不可以同一时间调用，否则模块会响应异常，需要顺序执行；

## 7.2.多模块多实例说明

- (1) 创建模块 A 的核心层句柄；
- (2) 创建模块 B 的核心层句柄；
- (3) 在 CMD\_Network\_Param\_t 类型中配置好参数，基与(1)核心层句柄创建一个 socket 网络层句柄，然后再调用接口函数添加网络指令，添加完成后，传 socket 句柄给网络层任务函数，定时调用即可；
- (4) 在 CMD\_Network\_Param\_t 类型中配置好参数，基与(2)核心层句柄创建一个 mqtt 网络层句柄，然后再调用接口函数添加网络指令，添加完成后，传 mqtt 句柄给网络层任务函数，定时调用即可；
- (5) 多模块多实例，由于模块是分开的，故同时调用不会对模块产生冲突，所以可以并行执行；

编写人：王土成