Hi3861 V100 / Hi3861L V100 CoAP

Development Guide

Issue 01

Date 2020-04-30

Trademarks and Permissions

 , HISILICON, and other HiSilicon icons are trademarks of HiSilicon Technologies Co., Ltd. All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between HiSilicon and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided “AS IS” without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

HiSilicon (Shanghai) Technologies Co., Ltd.

Address: New R&D Center, 49 Wuhe Road,
Bantian, Longgang District,
Shenzhen 518129 P. R. China

Website: https://www.hisilicon.com/en/

Email: support@hisilicon.com

About This Document

Purpose

This document describes the libcoap-based Constrained Application Protocol (CoAP) development with examples and the libcoap APIs with thread security encapsulated based on the Lightweight TCP/IP stack (lwIP).

Related Versions

The following table lists the product versions related to this document.

Product Name	Version
Hi3861	V100
Hi3861L	V100

Intended Audience

This document is intended for:

- Software development engineers
- Technical support engineers

Symbol Conventions

The symbols that may be found in this document are defined as follows.

Symbol	Description
DANGER	Indicates a hazard with a high level of risk which, if not avoided, will result in death or serious injury.
WARNING	Indicates a hazard with a medium level of risk which, if not avoided, could result in death or serious injury.

Symbol	Description
⚠ CAUTION	Indicates a hazard with a low level of risk which, if not avoided, could result in minor or moderate injury.
NOTICE	Indicates a potentially hazardous situation which, if not avoided, could result in equipment damage, data loss, performance deterioration, or unanticipated results. NOTICE is used to address practices not related to personal injury.
🕮 NOTE	Supplements the important information in the main text. NOTE is used to address information not related to personal injury, equipment damage, and environment deterioration.

Change History

Issue	Date	Change Description
01	2020-04-30	This issue is the first official release. • Added coap_server_sessions_check_idle_lwip to **1.1 Overview**. • Added **1.10 coap_server_sessions_check_idle_lwip**.
00B01	2020-01-15	This issue is the first draft release.

Contents

1 API Description

1.1 Overview

The CoAP basic functions depend on the open-source library libcoap. To implement CoAP-based services, call the libcoap APIs. This document describes the newly provided APIs that adapt to the lwIP protocol stack only. For details about the open-source libcoap APIs, see the libcoap website.

The open-source libcoap APIs are as follows:

- coap_new_context
- coap_free_context
- coap_check_notify
- coap_send
- coap_send_ack
- coap_send_error
- coap_send_message_type
- coap_send_rst

The following APIs are encapsulated and implemented to avoid race condition with TCP/IP threads:

- coap_new_context ↔ coap_new_context_lwip
- coap_free_context ↔ coap_free_context_lwip
- coap_check_notify ↔ coap_check_notify_lwip
- coap_send ↔ coap_send_lwip
- coap_send_ack ↔ coap_send_ack_lwip
- coap_send_error ↔ coap_send_error_lwip
- coap_send_message_type ↔ coap_send_message_type_lwip
- coap_send_rst ↔ coap_send_rst_lwip
- coap_server_sessions_check_idle_lwip

1.2 coap_new_context_lwip

Prototype	coap_context_t *coap_new_context_lwip(const coap_address_t *listen_addr);
Description	This API has the same function as **coap_new_context()**. However, this API must be called in a non-TCP/IP thread. This API is used to create an object of the **coap_context_t** type to record the status of the CoAP protocol stack.
Argument	**listen_addr**: local listening address and port number (must not be NULL).
Return Values	• Success: pointer of the **coap_context_t** type • Failure: NULL
Error Codes	None
Availability Since	nStack_N500 1.0.2

1.3 coap_free_context_lwip

Prototype	void coap_free_context_lwip(coap_context_t *context);
Description	This API has the same function as **coap_free_context()**. However, this API must be called in a non-TCP/IP thread. The **coap_free_context[_lwip]** API is used to free the context of the CoAP protocol stack. This API clears all entries in the TX queue and RX queue, deletes resources registered with the context, and releases endpoints associated with the context.
Argument	**context**: pointer of the **coap_context_t** type

Return Value	None
Error Code	None
Availability Since	nStack_N500 1.0.2

1.4 coap_check_notify_lwip

Prototype	void coap_check_notify_lwip(coap_context_t *context);
Description	This API has the same function as **coap_check_notify()**. However, this API must be called in a non-TCP/IP thread. This API is used to check if all known resources of a context are dirty and notify the subscription observer of the result.
Argument	**context**: pointer of the **coap_context_t** type
Return Value	None
Error Code	None
Availability Since	nStack_N500 1.0.2

1.5 coap_send_lwip

Prototype	coap_tid_t coap_send_lwip(coap_session_t *session, coap_pdu_t *pdu);
Description	This API has the same function as **coap_send()**. However, this API must be called in a non-TCP/IP thread. This API is used to send CoAP messages to the peer end. The memory allocated for the PDU will be freed by **coap_send_lwip()**. The caller must not use the PDU after **coap_send_lwip()** is called.
Argument	• **session**: CoAP session (must not be NULL) • **pdu**: PDU to be sent (must not be NULL)
Return Value	• Success: message ID of the PDU • Failure: **COAP_INVALID_TID**
Error Code	**COAP_INVALID_TID**: invalid argument or internal error
Availability Since	nStack_N500 1.0.2

1.6 coap_send_ack_lwip

Prototype	coap_tid_t coap_send_ack_lwip(coap_session_t *session, coap_pdu_t *request);
Description	This API has the same function as **coap_send_ack()**. However, this API must be called in a non-TCP/IP thread. This API is used to send an ACK message with code 0 to the endpoint that sends the **request** message.
Arguments	• **session**: CoAP session (must not be NULL) • **request**: PDU to be acknowledged by the local endpoint (must not be NULL)
Return Values	• Success: message ID of the ACK message • Failure: **COAP_INVALID_TID**
Error Codes	**COAP_INVALID_TID**: invalid argument or internal error
Availability Since	nStack_N500 1.0.2

1.7 coap_send_error_lwip

Prototype	coap_tid_t coap_send_error_lwip(coap_session_t *session, coap_pdu_t *request, unsigned char code, coap_opt_filter_t opts);
Description	This API has the same function as **coap_send_error()**. However, this API must be called in a non-TCP/IP thread. This API is used to send an error response message with a response **code** to the endpoint that sends the **request** message. A list of options with the **opts** flag will be carried.
Arguments	• **session**: CoAP session (must not be NULL) • **request**: request message to be responded to (must not be NULL) • **code**: response code • **opts**: filter that specifies the list of options to be copied from **request**
Return Values	• Success: message ID of the sent message • Failure: **COAP_INVALID_TID**
Error Codes	**COAP_INVALID_TID**: invalid argument or internal error

Availability Since	nStack_N500 1.0.2

1.8 coap_send_message_type_lwip

Prototype	coap_tid_t coap_send_message_type_lwip(coap_session_t *session, coap_pdu_t *request, unsigned char type);
Description	This API has the same function as **coap_send_message_type()**. However, this API must be called in a non-TCP/IP thread. This API is used to send a response of the **type** type to the endpoint that sends a **request** message.
Arguments	• **session**: CoAP session (must not be NULL) • **request**: request message to be responded to (must not be NULL) • **type**: message type (ACK or RST)
Return Values	• Success: message ID of the sent message • Failure: **COAP_INVALID_TID**
Error Codes	**COAP_INVALID_TID**: invalid argument or internal error
Availability Since	nStack_N500 1.0.2

1.9 coap_send_rst_lwip

Prototype	coap_tid_t coap_send_rst_lwip(coap_session_t *session, coap_pdu_t *request);
Description	This API has the same function as **coap_send_rst()**. However, this API must be called in a non-TCP/IP thread. This API is used to send a response of the RST type to the endpoint that sends a **request** message. This API actually calls **coap_send_message_type_lwip()** with **type** set to **COAP_MESSAGE_RST**.
Arguments	• **session**: CoAP session (must not be NULL) • **request**: request message to be responded to (must not be NULL)
Return Values	• Success: message ID of the PDU • Failure: **COAP_INVALID_TID**

Error Codes	**COAP_INVALID_TID**: invalid argument or internal error
Availability Since	nStack_N500 1.0.2

1.10 coap_server_sessions_check_idle_lwip

Prototype	void coap_server_sessions_check_idle_lwip(coap_context_t *ctx)
Description	This new API must be called in a non-TCP/IP thread. This API is used to detect whether any session of the server is idle because no packet is received or sent. The idle time can be set by **ctx > session_timeout** (unit: s). If **ctx->session_timeout** is set to **0**, the default time is **COAP_DEFAULT_SESSION_TIMEOUT**.
Argument	**cts**: CoAP session (must not be NULL)
Return Value	void
Error Code	N/A
Availability Since	nStack_N500 1.0.2

2 Development Guidance

2.1 Development Restrictions

2.1.1 Resource Configuration

The libcoap resources are pre-allocated in lwIP. Set the maximum number of supported nodes, contexts, endpoints, sessions, PDUs, and resources as required in the **lwippools.h** file using macros.

The following is a configuration example.

```
#define MEMP_NUM_COAPCONTEXT 1 /*Maximum number of contexts*/
#define MEMP_NUM_COAPENDPOINT 1 /*Maximum number of endpoints*/
#define MEMP_NUM_COAPPACKET 1 /*Maximum number of packets to be processed at the same time*/
#define MEMP_NUM_COAPNODE 4 /*Maximum number of CoAP nodes*/
#define MEMP_NUM_COAPPDU MEMP_NUM_COAPNODE /*Maximum number of messages that can be sent and received at the same time */
#define MEMP_NUM_COAPSESSION 4 /*Maximum number of sessions*/
#define MEMP_NUM_COAP_SUBSCRIPTION 1 /*Maximum number of subscribers */
#define MEMP_NUM_COAPRESOURCE 1 /*Maximum number of resources*/
#define MEMP_NUM_COAPRESOURCEATTR 2 /*Maximum number of resource features*/
#define MEMP_NUM_COAPOPTLIST 1 /*Maximum number of options*/
#define MEMP_LEN_COAPOPTLIST 12 /*Maximum length of a single option*/
#define MEMP_NUM_COAPSTRING 2 /*Maximum number of CoAP strings*/
#define MEMP_LEN_COAPSTRING 16 /*Maximum length of a single CoAP string*/
```

2.1.2 Session Creation

Extra code is needed to ensure the functionality of creating client sessions, because lwIP APIs instead of POSIX APIs are used.

```
session = coap_new_client_session(context, &local_addr, dst_addr, COAP_PROTO_UDP);
session->sock.pcb = context->endpoint->sock.pcb;
LL_PREPEND(context->endpoint->sessions, session);
```


After a session is created, you need to configure the UDP program control block (PCB) associated with the session and extend the session to the session list of the endpoint, so that the session can receive and send packets normally.

2.1.3 Interface Bounding

In this version, no network interface is bound by default. To bind a network interface, define the **WITH_LWIP_SPECIFIC_IFINDEX** macro in **coap_config.h** and specify the network interface to be bound to the session after the session is created.

```
session->ifindex = netif->ifindex;
```

2.2 Sample Code

2.2.1 Server

```
#define TEST_IPV4 1
static u32_t coap_test_taskid = -1;
static int serv_running = 0;
static coap_context_t *serv_ctx = NULL;

/*
 * The resource 'hello' GET method handler
 */
static void
hello_handler(coap_context_t  *ctx, struct coap_resource_t *resource,
              coap_session_t *session,
              coap_pdu_t *request, coap_binary_t *token,
              coap_string_t *query,
              coap_pdu_t *response)
{
    unsigned char buf[3];
    /* response with text "Hello World!" */
    const char* response_data     = "Hello World!";
    size_t len = 0;
    unsigned char *data = NULL;

    (void)ctx;
    (void)resource;
    (void)session;
    (void)token;
    (void)query;

    response->code          = COAP_RESPONSE_CODE(205);
    coap_add_option(response, COAP_OPTION_CONTENT_TYPE, coap_encode_var_safe(buf, 3,
COAP_MEDIATYPE_TEXT_PLAIN), buf);
    coap_add_data(response, strlen(response_data), (unsigned char *)response_data);

    if (coap_get_data(request, &len, &data)) {
        printf("[%s][%d] len : %d, data : %.*s\n", __FUNCTION__, __LINE__, len, len, data);
    }
}

void coap_server_thread(UINT32 uwParam1,
                        UINT32 uwParam2,
                        UINT32 uwParam3,
                        UINT32 uwParam4)
```

```
{
    coap_context_t*  ctx;

    (void)uwParam2;
    (void)uwParam3;
    (void)uwParam4;

    printf("[%s][%d] thread running\n", __FUNCTION__, __LINE__);
    ctx = (coap_context_t*)uwParam1;
    while (serv_running == 1) {
        // printf("coap_server sleep 1s\n");
        hi_sleep(1000);
        coap_check_notify_lwip(ctx);
    }
    if (serv_ctx != NULL) {
        coap_free_context_lwip(serv_ctx);
        serv_ctx = NULL;
    }
    printf("[%s][%d] thread exit\n", __FUNCTION__, __LINE__);
    return;
}

int coap_server_start(int argc, char* argv[])
{
    TSK_INIT_PARAM_S stappTask;
    UINT32 ret;

    (void)argc;
    (void)argv;

    coap_address_t    serv_addr;
    coap_resource_t* hello_resource;

    if (serv_running == 1) {
       return 0;
    }
    serv_running = 1;
    /* Prepare the CoAP server socket */
    coap_address_init(&serv_addr);
#if TEST_IPV4
    ip_addr_set_any(false, &(serv_addr.addr));
#else
    ip_addr_set_any(true, &(serv_addr.addr));
#endif
    serv_addr.port        = COAP_DEFAULT_PORT;
    serv_ctx              = coap_new_context_lwip(&serv_addr);
    if (!serv_ctx) {
        return -1;
    }
    /* Initialize the hello resource */
    hello_resource = coap_resource_init(coap_make_str_const("hello"), 0);
    coap_register_handler(hello_resource, COAP_REQUEST_GET, hello_handler);
    coap_add_resource(serv_ctx, hello_resource);

    /* create a thread task */
    stappTask.pfnTaskEntry = (TSK_ENTRY_FUNC)coap_server_thread;
    stappTask.uwStackSize  = 10*LOSCFG_BASE_CORE_TSK_DEFAULT_STACK_SIZE;
    stappTask.pcName = "coap_test_task";
    stappTask.usTaskPrio = 11;
    stappTask.uwResved   = LOS_TASK_STATUS_DETACHED;
    stappTask.auwArgs[0] = (UINT32)serv_ctx;
```



```
    printf("task create coap_server_thread\n");
    ret = LOS_TaskCreate(&coap_test_taskid, &stappTask);
    if (0 != ret ) {
        dprintf("coap_server_thread create failed ! \n");
        return -1;
    }

    return 0;
}

void coap_server_stop(void)
{
    if (serv_running == 0) {
       printf("[%s][%d] not running\n", __FUNCTION__, __LINE__);
       return;
    }
    if (serv_ctx != NULL) {
        coap_free_context_lwip(serv_ctx);
        serv_ctx = NULL;
    }
    serv_running = 0;
    printf("[%s][%d] stopped\n", __FUNCTION__, __LINE__);
    return;
}
```

2.2.2 Client

```
#define TEST_IPV4 1
#define DHCP_COAP_TOKEN_LEN (4)
static coap_context_t*  cli_ctx = NULL;

/*
 * The response handler
 */
static void
message_handler(struct coap_context_t *ctx,
                coap_session_t *session,
                coap_pdu_t *sent,
                coap_pdu_t *received,
                const coap_tid_t id)
{
    unsigned char* data;
    size_t         data_len;

    (void)ctx;
    (void)sent;
    (void)received;
    (void)id;

    if (COAP_RESPONSE_CLASS(received->code) == 2)
    {
        if (coap_get_data(received, &data_len, &data))
        {
            printf("Received: %.*s\n", data_len, data);
        }
    }
}

int coap_client_start(int argc, char* argv[])
{
    coap_address_t    src_addr;
```

```
    (void)argc;
    (void)argv;

    if (cli_ctx != NULL) {
        return 0;
    }
    /* Prepare coap socket*/
    coap_address_init(&src_addr);
#if TEST_IPV4
    ip_addr_set_any(false, &(src_addr.addr));
#else
    ip_addr_set_any(true, &(src_addr.addr));
#endif
    src_addr.port        = 23456;

    cli_ctx = coap_new_context_lwip(&src_addr);
    if (!cli_ctx) {
        return -1;
    }

    /* Set the response handler*/
    coap_register_response_handler(cli_ctx, message_handler);

  return 0;
}

void coap_client_stop(void)
{
    if (cli_ctx != NULL) {
        coap_free_context_lwip(cli_ctx);
        cli_ctx = NULL;
    }
    printf("[%s][%d] stopped\n", __FUNCTION__, __LINE__);
    return;
}

/* to create a new token value depanding on time */
s32_t coap_new_token(u16_t msg_id, u8_t *token, u8_t token_len)
{
  u32_t now_ms;

  if ((token == NULL) || (token_len < DHCP_COAP_TOKEN_LEN)) {
    return -1;
  }

  now_ms = sys_now();

  token[0] = (u8_t)(msg_id);
  token[1] = (u8_t)(msg_id >> 8);
  token[2] = (u8_t)(now_ms);
  token[3] = (u8_t)(now_ms >> 8);

  return 0;
}

int coap_client_send_msg(char* dst)
{
    coap_address_t    dst_addr, listen_addr;
    static coap_uri_t uri;
    coap_pdu_t*       request;
    coap_session_t *session = NULL;
    char              server_uri[128] = {0};
```

```
    u8_t temp_token[DHCP_COAP_TOKEN_LEN] = {0};
    unsigned char     get_method = COAP_REQUEST_GET;

    /* The destination endpoint */
    coap_address_init(&dst_addr);
    printf("[%s][%d] server : %s\n", __FUNCTION__, __LINE__, dst);
#if TEST_IPV4
    if (!ipaddr_aton(dst, &(dst_addr.addr))) {
        printf("invalid ip4 addr\n");
        return -1;
    }
#else
    if (!ip6addr_aton(dst, &(dst_addr.addr.u_addr.ip6))) {
        printf("invalid ip6 addr\n");
        return -1;
    }
    IP_SET_TYPE_VAL(dst_addr.addr, IPADDR_TYPE_V6);
#endif
    dst_addr.port        = COAP_DEFAULT_PORT;

    /* try to reuse existed session */
    session = coap_session_get_by_peer(cli_ctx, &dst_addr, 0);
    if ((session != NULL) && !(coap_address_equals(&dst_addr, &dst_addr))) {
        printf("[%s][%d] session dst changed\n", __FUNCTION__, __LINE__);
        LL_DELETE(cli_ctx->sessions, session);
        LL_DELETE(cli_ctx->endpoint->sessions, session);
        coap_session_mfree(session);
        coap_free_type(COAP_SESSION, session);
        session = NULL;
      }
    if (session == NULL) {
        coap_address_init(&listen_addr);
#if TEST_IPV4
        ip_addr_set_any(false, &(listen_addr.addr));
#else
        ip_addr_set_any(true, &(listen_addr.addr));
#endif
        listen_addr.port = 23456;
        session = coap_new_client_session(cli_ctx, &listen_addr, &dst_addr, COAP_PROTO_UDP);
        if (session == NULL) {
            printf("[%s][%d] new client session failed\n", __FUNCTION__, __LINE__);
            return -1;
        }
        session->sock.pcb = cli_ctx->endpoint->sock.pcb;
        LL_PREPEND(cli_ctx->endpoint->sessions, session);
    }

    /* Prepare the request */
    strcpy(server_uri, "/hello");
    coap_split_uri((unsigned char *)server_uri, strlen(server_uri), &uri);
    request           = coap_new_pdu(session);
    if (request == NULL) {
        printf("[%s][%d] get pdu failed\n", __FUNCTION__, __LINE__);
        return -1;
    }
    request->type = COAP_MESSAGE_CON;
    request->tid   = coap_new_message_id(session);
    (void)coap_new_token(request->tid, temp_token, DHCP_COAP_TOKEN_LEN);
    if (coap_add_token(request, DHCP_COAP_TOKEN_LEN, temp_token) == 0) {
        printf("[%s][%d] add token failed\n", __FUNCTION__, __LINE__);
    }
    request->code = get_method;
```



```
    coap_add_option(request, COAP_OPTION_URI_PATH, uri.path.length, uri.path.s);

    char request_data[64] = {0};
    (void)snprintf_s(request_data, sizeof(request_data), sizeof(request_data)-1, "%s", "luminais");

    coap_add_data(request, 4+strlen((const char *)(request_data+4)), (unsigned char
*)request_data);
    coap_send_lwip(session, request);

    return 0;
}

int coap_client_send(int argc, char* argv[])
{
    (void)argc;
    (void)argv;
    if (argc < 2) {
        printf("coap client {start | stop | send server_addr}\n");
        return -1;
    }
    if (cli_ctx == NULL) {
        return -1;
    }

    /* argv[0] is server_addr */
    coap_client_send_msg(argv[0]);
    return 0;
}
```

