GigaDevice Semiconductor Inc.

GD32VW553 Network Application Development Guide

Application Note

AN185

Revision 1.1

(Mar. 2025)

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1. About this document

This document introduces how to use various SDK-integrated components to implement network application development based on the GD32VW553 series chip.

The GD32VW553 series chip is a 32-bit microcontroller (MCU) with RISC-V as the core, which contains Wi-Fi 4/Wi-Fi 6 and BLE 5.3 connection technologies. GD32VW553 Wi-Fi+BLE SDK integrates the Wi-Fi driver, BLE driver, LwIP TCP/IP protocol stack, MbedTLS, and other components, allowing developers to quickly develop IoT applications based on GD32VW553.

To quickly grasp the use of the GD32VW553 series chip, see the document "GD32VW553 Quick Development Guide". For how to develop Wi-Fi and BLE applications, see the documents "GD32VW553 Wi-Fi Development Guide" and "GD32VW553 BLE Development Guide".

2. LwIP Sockets application development

This chapter introduces how to use LwIP Sockets API to implement UDP Server/UDP Client/TCP Server/TCP Client.

The relevant LwIP Sockets function declaration is located in the following header file. LwIP is of Version 2.2.0.

MSDK\lwip\lwip-2.2.0\src\include\lwip\sockets.h

MSDK\lwip\lwip-2.2.0\src\include\lwip\priv\sockets_priv.h

2.1. Structure

2.1.1. sockaddr_in

Table 2-1 sockaddr_in structure

```
struct sockaddr_in {
   u8_t                  sin_len;
   sa_family_t           sin_family;
   in_port_t             sin_port;
   struct in_addr        sin_addr;
#define SIN_ZERO_LEN 8
   char                  sin_zero[SIN_ZERO_LEN];
};
```

2.1.2. sockaddr

Table 2-2 sockaddr structure

```
struct sockaddr {
   u8_t             sa_len;
   sa_family_t      sa_family;
   char             sa_data[14];
};
```

2.1.3. timeval

Table 2-3 timeval structure

```
struct timeval {
   long   tv_sec;    /* seconds */
   long   tv_usec;   /* and microseconds */
};
```

2.1.4. fd_set

Table 2-4 fd_set structure

```
/* Socket file descriptor set */
typedef struct fd_set
{
   unsigned char fd_bits [(FD_SETSIZE+7)/8];
} fd_set;
/* The following is the fd_set structure operator */
FD_ZERO(fd_set *fdset) //Clear the socket file descriptor set
FD_SET(int fd, fd_set *fdset) //Add a descriptor to the socket file descriptor set
FD_CLR(int fd, fd_set *fdset) //Delete a descriptor from the socket file descriptor set
FD_ISSET(int fd, fd_set *fdset) //Determine whether the descriptor is in the socket
file descriptor set
```

2.2. API

2.2.1. socket

Macro: #define socket(domain,type,protocol)   lwip_socket(domain,type,protocol)

Prototype: int lwip_socket(int domain, int type, int protocol)

Purpose: The function is used to apply for a socket.

Input parameter: domain, the protocol family used by the socket, where AF_INET corresponds to the IPv4 protocol and AF_INET6 to the IPv6 protocol.

type, the service type used by the socket, where SOCK_STREAM corresponds to TCP, SOCK_DGRAM to UDP, and SOCK_RAW to the original socket.

protocol, the specific protocol used by the socket, which generally uses the default value 0.

Output parameter: None.

Return value: Return a socket descriptor upon success and -1 upon failure.

2.2.2. bind

Macro: #define bind(s,name,namelen)   lwip_bind(s,name,namelen)

Prototype: int lwip_bind (int s, const struct sockaddr *name, socklen_t namelen)

Purpose: The function is used to bind the socket and NIC information on the server side.

Input parameter: s, the server-side socket descriptor to be bound.

name, a pointer to the sockaddr structure, which contains the IP address, port number, and other information of the NIC. This information is stored in the continuous 14 bytes (sa_data[14]) of the structure, which is not user-friendly. Therefore, the clearer structure sockaddr_ in is usually used. These two structures are equivalent, except that sockaddr_in re-divides sa_data[14] into sin_port, sin_addr, and other fields.

namelen, the length of the name structure.

Output parameter: None.

Return value: Return 0 upon success and -1 upon failure.

Table 2-5 bind () example

```
struct sockaddr_in server_addr;
/* Pad the sockaddr_in structure */
server_addr.sin_family = AF_INET;
server_addr.sin_len = sizeof(server_addr);
server_addr.sin_port = htons(server_port);
server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
/* Cast the sockaddr_in structure */
bind (server_fd, (struct sockaddr *) &server_addr, sizeof(server_addr));
```

2.2.3. connect

Macro: #define connect (s,name,namelen) lwip_connect(s,name,namelen)

Prototype: int lwip_connect (int s, const struct sockaddr *name, socklen_t namelen)

Purpose: The function is used to bind the socket and remote NIC information on the client. For UDP, only information is recorded; for TCP, in addition to information recording, a handshake process will be initiated and a TCP connection will be established.

Input parameter: s, the socket descriptor at the client.

name, a pointer to the sockaddr structure, which stores the remote

NIC information.

namelen, the length of the name structure.

Output parameter: None.

Return value: Return 0 upon success and -1 upon failure.

2.2.4. listen

Macro: #define listen(s,backlog) lwip_listen(s,backlog)

Prototype: int lwip_listen (int s, int backlog)

Purpose: The function is only used on the TCP server side, allowing the server to enter the listening status and wait for the remote connection request.

Input parameter: s, the server-side socket descriptor.

backlog, the size of the request queue, which is only valid when the macro TCP_LISTEN_BACKLOG==1.

Output parameter: None.

Return value: Return 0 upon success and -1 upon failure.

2.2.5. accept

Macro: #define accept(s,addr,addrlen)   lwip_accept(s,addr,addrlen)

Prototype: int lwip_accept (int s, struct sockaddr *addr, socklen_t *addrlen)

Purpose: The function is only used to wait for the remote connection request and establish a new TCP connection on the TCP server side. Before calling this function, call the listen () function to enter the listening status.

Input parameter: s, the server-side socket descriptor.

addrlen, the length of the addr structure.

Output parameter: addr, a pointer to the sockaddr structure, which stores the remote NIC information.

Return value: Return a socket descriptor that represents the remote end upon success and -1 upon failure.

2.2.6. sendto

Macro: #define sendto(s,dataptr,size,flags,to,tolen)   lwip_sendto(s,dataptr,size,flags,to,tolen)

Prototype: ssize_t lwip_sendto (int s, const void *dataptr, size_t size, int flags,

const struct sockaddr *to, socklen_t tolen)

Purpose: The function is used to transmit data via UDP and send UDP messages to

the remote end.

Input parameter: s, the socket descriptor.

dataptr, the start address of the data to be sent.

size, the length of the data.

flags, which is used to specify some processing when sending data.

For example, MSG_DONTWAIT (0x08) indicates that this transmission

Is non-blocking. It is usually set to 0. to, a pointer to the

sockaddr structure, which stores the remote NIC information.

tolen, the length of the "to" structure.

Output parameter: None.

Return value: Return the length of sent data upon success and -1 upon failure.

2.2.7. send

Macro: #define send(s,dataptr,size,flags)   lwip_send(s,dataptr,size,flags)

Prototype: ssize_t lwip_send (int s, const void *dataptr, size_t size, int flags)

Purpose: The function is used to transmit data via UDP and TCP and send data to

the remote end. Because "send" does not specify remote information,

this function needs to be used when the socket is connected.

Input parameter: s, the socket descriptor.

dataptr, the start address of the data to be sent.

size, the length of the data.

flags, which is used to specify some processing when sending data.

See descriptions of "sendto".

Output parameter: None.

Return value: Return the length of sent data upon success and -1 upon failure.

2.2.8. recvfrom

Macro:

#define recvfrom(s,mem,len,flags,from,fromlen) lwip_recvfrom(s,mem,len,flags,from,fromlen)

Prototype: ssize_t lwip_recvfrom (int s, void *mem, size_t len, int flags,

struct sockaddr *from, socklen_t *fromlen)

Purpose: The function is used to receive data via UDP and TCP.

Input parameter: s, the socket descriptor.

len, the maximum length of received data.

flags, which is used to specify some processing when sending data.

See descriptions of "sendto".

fromlen, the length of the "from" structure.

Output parameter: mem, the start address of the cache of received data.

from, a pointer to the sockaddr structure, which stores the remote

NIC information.

Return value: Return the length of received data upon success and -1 upon failure.

2.2.9. recv

Macro: #define recv(s,mem,len,flags) lwip_recv(s,mem,len,flags)

Prototype: ssize_t lwip_recv (int s, void *mem, size_t len, int flags)

Purpose: The function is used to receive data via UDP and TCP. It is practically

equivalent to the recvfrom function when "from" and "fromlen" are both NULL.

Input parameter: s, the socket descriptor.

len, the maximum length of received data.

flags, which is used to specify some processing when sending data.

See descriptions of "sendto".

Output parameter: mem, the start address of the cache of received data.

Return value: Return the length of received data upon success and -1 upon failure.

2.2.10. shutdown

Macro: #define shutdown(s,how)   lwip_shutdown(s,how)

Prototype: int lwip_shutdown (int s, int how)

Purpose: The function is used to close the connection, which only works for TCP, not for UDP.

Input parameter: s, the socket descriptor.

how, the disconnection method. SHUT_RD, which is used to disconnect the input stream; SHUT_WR, which is used to disconnect the output stream; SHUT_RDWR, which is used to disconnect the input and output streams.

Output parameter: None.

Return value: Return 0 upon success and -1 upon failure.

2.2.11. close

Macro: #define close(s) lwip_close(s)

Prototype: int lwip_close (int s)

Purpose: The function is used to close the socket.

Input parameter: s, the socket descriptor.

Output parameter: None.

Return value: Return 0 upon success and -1 upon failure.

2.2.12. setsockopt

Macro: #define setsockopt(s,level,optname,opval,optlen)   lwip_setsockopt(s,level, \
optname,opval,optlen)

Prototype: int lwip_setsockopt (int s, int level, int optname, const void *optval, socklen_t optlen)

Purpose: The function is used to set the socket option information.

Input parameter: s, the socket descriptor.

level, the option level. For example, SOL_SOCKET represents the socket level; IPPROTO_IP represents the IP level; IPPROTO_TCP represents the TCP level.

optname, the specific option name of the level. For example, at the TCP level, there are TCP_NODELAY (not using the

Nagle algorithm) and TCP_KEEPALIVE (setting the TCP keep-alive time); at the IP level, there are IP_TOS (setting the service type) and IP_TTL (setting the survival time); at the socket level, there are SO_REUSEADDR (allowing reuse of the local address), SO_RCVTIMEO (setting the data receiving timeout period), etc.

opval, the value set by the optname option.

optlen, the length of opval.

Output parameter: None.

Return value: Return 0 upon success and -1 upon failure.

2.2.13. getsockname

Macro: #define getsockopt(s,level,optname,opval,optlen) lwip_getsockopt(s,level, \
optname,opval,optlen)

Prototype: int lwip_getsockopt (int s, int level, int optname, void *optval, socklen_t *optlen)

Purpose: The function is used to obtain the socket option information.

Input parameter: s, the socket descriptor.

level, the option level. See the setsockopt function.

optname, the specific option name of the level. See the setsockopt function.

optlen, the length of opval.

Output parameter: opval, the obtained value set by the optname option.

Return value: Return 0 upon success and -1 upon failure.

2.2.14. fcntl

Macro: #define fcntl(s,cmd,val) lwip_fcntl(s,cmd,val)

Prototype: int lwip_fcntl (int s, int cmd, int val)

Purpose: The function is used to perform some socket operations.

Input parameter: s, the socket descriptor.

cmd, socket operations. F_GETFL is used to obtain the attributes of the socket; F_SETFL is used to set the attributes of the socket.

val, when cmd is F_GETFL. is invalid and can be set to 0,It represents the socket attributes to be set when cmd is F_SETFL. For example, O_NONBLOCK indicates that the socket is non-blocking.

Output parameter: None.

Return value: When cmd is F_SETFL, return 0 upon success and -1 upon failure.
When cmd is F_GETFL, return the attributes of the socket upon success and -1 upon failure.

Table 2-6 fcntl example

```
int nflags = -1;
nflags = fcntl(fd, F_GETFL, 0);
if (nflags < 0)
      return;
nflags |= O_NONBLOCK;
if (fcntl(fd, F_SETFL, nflags) < 0)
      return;
```

2.2.15. select

Macro: #define select (maxfdp1, readset, writeset,exceptset,timeout) lwip_select(maxfdp1, \ readset,writeset,exceptset,timeout)

Prototype: int lwip_select (int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset, struct timeval *timeout)

Purpose: The function is used to monitor the changes of the socket status, including reading, writing, and exceptions.

Input parameter: maxfdp1, the maximum socket descriptor value that needs to be monitored plus 1. The maximum socket descriptor value here refers to the maximum value in the three socket descriptor sets readset/writeset/exceptset.

readset, fd_set structure pointer, which points to the socket descriptor set whose readable status needs to be monitored. It can be NULL, indicating that the readable status is not monitored.

writeset, fd_set structure pointer, which points to the socket descriptor set whose writable status needs to be monitored. It can be NULL, indicating that the writable status is not monitored.

exceptset, fd_set structure pointer, which points to the socket descriptor set whose exception status needs to be monitored. It can be NULL, indicating that the exception status is not monitored.

timeout, timeval structure pointer, which points to the timeout value. When it is NULL, "select" is blocked and will not return a value until any socket in a monitored socket descriptor set changes; when the timeout value is set to 0, "select" is in the non-blocking status and will directly return a value regardless of whether any socket in a monitored socket descriptor set changes; when the timeout value is greater than 0, "select" is blocked within the timeout period and will return a value when any socket changes within the timeout period or the timeout period expires.

Output parameter: None.

Return value: negative value, indicating that "select" has an error; positive value, indicating that a change of the socket descriptor status is monitored; 0, indicating that it timed out or no change is monitored.

Table 2-7 "select" example

```
char recv_buf[128];
fd_set read_set;
struct timeval timeout;
int max_fd_num = 0;

timeout.tv_sec = 1;
timeout.tv_usec = 0;

while (1) {
/* The socket descriptor set will change after "select" is executed, so the monitored descriptors
need to be reinitialized and set */
    FD_ZERO(&read_set);
    FD_SET (fd, &read_set);
    max_fd_num = fd + 1;
```

```
        select (max_fd_num, &read_set, NULL, NULL, &timeout);
        if (!FD_ISSET(fd, &read_set))
            continue;
        sys_memset(recv_buf, 0, 128);
        recv (fd, recv_buf, 128, 0);
}
```

2.3. Errno error code

During the use of the above APIs, if an execution error occurs, the return value is usually -1, without providing specific error information. LwIP uses a global variable errno. When an API error occurs, errno will assign an error code through sock_set_errno(). This allows users to determine the specific cause of the error by reading errno at the API error location.

The error code is located in the MSDK\lwip\lwip-2.2.0\src\include\lwip\errno.h file. The example of use is as follows:

Table 2-8 errno example

```
int ret;
char recv_buf[128];
sys_memset (recv_buf, 0, 128);
ret = recv (fd, recv_buf, 128, MSG_DONTWAIT);
if (ret < 0) {
  if (errno != EAGAIN) {
    printf("recv error: %d.\r\n", errno);
  }
}
```

2.4. LwIP Sockets programming example

For LwIP Sockets programming example, see the MSDK\lwip\lwip-2.2.0\demo\lwip_sockets_demo.c file.

The example uses the LwIP Sockets API introduced in this chapter to implement UDP Server/UDP Client/TCP Server/TCP Client, which can all communicate with the remote end.

3. MbedTLS application development

This chapter introduces how to use the MbedTLS component to implement an HTTPS Client that can access and interact with the HTTPS Server.

The relevant MbedTLS interface function declaration is located in the following header file. MbedTLS is version 3.6.2.

MSDK\mbedtls\mbedtls-3.6.2\include\mbedtls\ssl.h

MSDK\mbedtls\mbedtls-3.6.2\include\mbedtls\net_sockets.h

MSDK\mbedtls\mbedtls-3.6.2\include\mbedtls\x509_crt.h

MSDK\mbedtls\mbedtls-3.6.2\include\mbedtls\pk.h

MSDK\mbedtls\mbedtls-3.6.2\include\mbedtls\debug.h

3.1. Related structures

3.1.1. mbedtls_net_context

/* TLS network context */

typedef struct mbedtls_net_context mbedtls_net_context;

3.1.2. mbedtls_ssl_context

/* SSL context */

typedef struct mbedtls_ssl_context mbedtls_ssl_context;

3.1.3. mbedtls_ssl_config

/* SSL configuration */

typedef struct mbedtls_ssl_config mbedtls_ssl_config;

3.1.4. mbedtls_x509_crt

/* x509 certificate structure */

typedef struct mbedtls_x509_crt mbedtls_x509_crt;

3.1.5. mbedtls_pk_context

/* Public key context */

typedef struct mbedtls_pk_context mbedtls_pk_context;

3.2. Initialization API

3.2.1. mbedtls_debug_set_threshold

Prototype: void mbedtls_debug_set_threshold (int threshold)

Purpose: The function is used to set the output level of the debug log.

Input parameter: threshold, the debug level to be set.

Output parameter: None.

Return value: None.

The function is used for debugging. To use it, open the macro MBEDTLS_DEBUG_C, which is located in MSDK\mbedtls\mbedtls-3.6.2\include\mbedtls\mbedtls_config.h.

3.2.2. mbedtls_net_init

Prototype: void mbedtls_net_init (mbedtls_net_context *ctx)

Purpose: The function is used to initialize the TLS network context.

Input parameter: ctx, mbedtls_net_context structure pointer, which points to
the network context object.

Output parameter: None.

Return value: None.

3.2.3. mbedtls_ssl_init

Prototype: void mbedtls_ssl_init (mbedtls_ssl_context *ssl)

Purpose: The function is used to initialize the SSL context.

Input parameter: ssl, mbedtls_ssl_context structure pointer, which points to the SSL
context object.

Output parameter: None.

Return value: None.

3.2.4. mbedtls_ssl_config_init

Prototype: void mbedtls_ssl_config_init (mbedtls_ssl_config *conf)

Purpose: The function is used to initialize SSL configuration.

Input parameter: conf, mbedtls_ssl_config structure pointer, which points to SSL configuration.

Output parameter: None.

Return value: None.

3.2.5. mbedtls_x509_crt_init

Prototype: void mbedtls_x509_crt_init (mbedtls_x509_crt *crt)

Purpose: The function is used to initialize the root certificate linked list.

Input parameter: crt, mbedtls_x509_crt structure pointer, which points to the x509 certificate object.

Output parameter: None.

Return value: None.

3.2.6. mbedtls_pk_init

Prototype: void mbedtls_pk_init (mbedtls_pk_context *ctx)

Purpose: The function is used to initialize the public key context.

Input parameter: ctx, mbedtls_pk_context structure pointer, which points to the public key context object.

Output parameter: None.

Return value: None.

3.3. Configuration API

3.3.1. mbedtls_x509_crt_parse

Prototype: int mbedtls_x509_crt_parse (mbedtls_x509_crt *chain, const unsigned char *buf, size_t buflen)

Purpose: The function is used to parse one or more certificates in buf and add them to the root certificate linked list.

Input parameter: chain, mbedtls_x509_crt structure pointer, which points to the x509 certificate object.

buf, which points to the buffer that stores the root certificate.

buflen, the size of the buffer that stores the root certificate.

Output parameter: None.

Return value: 0 indicates successful parsing and a non-0 value indicates failure.

3.3.2. mbedtls_pk_parse_key

Prototype: int mbedtls_pk_parse_key (mbedtls_pk_context *ctx,

const unsigned char *key, size_t keylen,

const unsigned char *pwd, size_t pwdlen ,

int (*f_rng)(void *, unsigned char *, size_t), void *p_rng)

Purpose: The function is used to parse the public key and add it to the public key context.

Input parameter: ctx, mbedtls_pk_context structure pointer, which points to

the public key context object.

key, which points to the buffer that stores the public key.

keylen, the size of the buffer that stores the public key.

pwd, which points to the buffer that stores the decryption password.

It can be NULL, when it indicates that the public key is not encrypted.

pwdlen, the size of the buffer that stores the decryption password,

which is ignored when pwd is NULL.

f_rng, RNG function, must not be NULL.

p_rng, RNG parameter.

Output parameter: None.

Return value: 0 indicates successful parsing and a non-0 value indicates failure.

3.3.3. mbedtls_ssl_conf_own_cert

Prototype: int mbedtls_ssl_conf_own_cert (mbedtls_ssl_config *conf,

mbedtls_x509_crt *own_cert,

mbedtls_pk_context *pk_key)

Purpose: The function is used to associate its own certificate chain and public key.

Input parameter: conf, mbedtls_ssl_config structure pointer, which points to SSL configuration.

own_cert, mbedtls_x509_crt structure pointer, which points to the x509 certificate object.

pk_key, mbedtls_pk_context structure pointer, which points to

the public key context object.

Output parameter: None.

Return value: 0 indicates successful association and a non-0 value indicates failure.

3.3.4. mbedtls_ssl_config_defaults

Prototype: int mbedtls_ssl_config_defaults (mbedtls_ssl_config *conf,

int endpoint, int transport, int preset)

Purpose: The function is used to load the default SSL configuration.

Input parameter: conf, mbedtls_ssl_config structure pointer,

which points to SSL configuration. endpoint, which is used to set SSL

as client or server, MBEDTLS_SSL_IS_CLIENT or

MBEDTLS_SSL_IS_SERVER. transport,

TLS(MBEDTLS_SSL_TRANSPORT_STREAM)

or DTLS(MBEDTLS_SSL_TRANSPORT_DATAGRAM).

preset, MBEDTLS_SSL_PRESET_DEFAULT by default.

Output parameter: None.

Return value: 0 indicates successful configuration and a non-0 value indicates failure.

3.3.5. mbedtls_ssl_conf_rng

Prototype: void mbedtls_ssl_conf_rng (mbedtls_ssl_config *conf,

int (*f_rng) (void *, unsigned char *, size_t),

void *p_rng)

Purpose: The function is used to set the random number generator callback.

Input parameter: conf, mbedtls_ssl_config structure pointer, which points to SSL configuration.

f_rng, the random number generator function.

p_rng, parameters of the random number generator function.

Output parameter: None.

Return value: None.

For f_rng, see the my_random () function in the MSDK\mbedtls\demo\ssl_client.c file.

3.3.6. mbedtls_ssl_conf_dbg

Prototype: void mbedtls_ssl_conf_dbg (mbedtls_ssl_config *conf,
void (*f_dbg) (void *, int, const char *, int, const char *),
void *p_dbg)

Purpose: The function is used to set the debug callback.

Input parameter: conf, mbedtls_ssl_config structure pointer, which points to SSL configuration.
f_dbg, the debug function.
p_dbg, parameters of the debug function.

Output parameter: None.

Return value: None.

For f_dbg, see the my_debug () function in the MSDK\mbedtls\demo\ssl_client.c file.

3.3.7. mbedtls_ssl_conf_authmode

Prototype: void mbedtls_ssl_conf_authmode (mbedtls_ssl_config *conf, int authmode)

Purpose: The function is used to set the certificate verification mode.

Input parameter: conf, mbedtls_ssl_config structure pointer, which points to
SSL configuration. authmode, the certificate verification mode. As
follows, MBEDTLS_SSL_VERIFY_NONE: Do not check Do not check
the certificate, server default. If it is a client, the connection is not secure.
MBEDTLS_SSL_VERIFY_OPTIONAL: Check the peer certificate. However,
the handshake will continue even if the verification result is "fail".
MBEDTLS_SSL_VERIFY_REQUIRED: Check the peer certificate.
The handshake will terminate if verification result is "fail".
This is the client's default value.

Output parameter: None.

Return value: None.

3.3.8. mbedtls_ssl_conf_ca_chain

Prototype: void mbedtls_ssl_conf_ca_chain (mbedtls_ssl_config *conf,

mbedtls_x509_crt *ca_chain,

mbedtls_x509_crl *ca_crl)

Purpose: The function is used to set the data required to verify the certificate.

Input parameter: conf, mbedtls_ssl_config structure pointer, which points to SSL configuration.

ca_chain, the trusted CA certificate chain, which is stored in

the mbedtls_x509_crt structure.

ca_crl, the trusted CA CRLs, which are stored in

the mbedtls_x509_crl structure

Output parameter: None.

Return value: None.

3.3.9. mbedtls_ssl_conf_verify

Prototype: void mbedtls_ssl_conf_verify (mbedtls_ssl_config *conf,

int (*f_vrfy)(void *, mbedtls_x509_crt *, int, uint32_t *),

void *p_vrfy)

Purpose: The function is used to set the certificate verification callback.

Input parameter: conf, mbedtls_ssl_config structure pointer, which points to SSL configuration.

f_vrfy, the verification callback function.

p_vrfy, parameters of the verification callback function.

Output parameter: None.

Return value: None.

For f_vrfy, it is recommended to use the my_verify () function in the MSDK\mbedtls\demo\ssl_client.c file, and users can also modify it.

3.3.10. mbedtls_ssl_setup

Prototype: int mbedtls_ssl_setup (mbedtls_ssl_context *ssl,

const mbedtls_ssl_config *conf)

Purpose: The function is used to set the SSL configuration into the SSL context

and initialize handshake.

Input parameter: ssl, mbedtls_ssl_context structure pointer, which points to

the SSL context object.

conf, mbedtls_ssl_config structure pointer, which points to

SSL configuration.

Output parameter: None.

Return value: 0 indicates successful configuration and a non-0 value indicates failure.

3.3.11. mbedtls_ssl_set_hostname

Prototype: int mbedtls_ssl_set_hostname (mbedtls_ssl_context *ssl, const char *hostname)

Purpose: The function is used to set the host name.

Input parameter: ssl, mbedtls_ssl_context structure pointer, which points to

the SSL context object.

hostname, the host name, which must correspond to the server certificate.

Output parameter: None.

Return value: 0 indicates successful setting and a non-0 value indicates failure.

3.4. Connection and handshake APIs

3.4.1. mbedtls_net_connect

Prototype: int mbedtls_net_connect (mbedtls_net_context *ctx, const char *host,

const char *port, int proto)

Purpose: The function is used to establish a network connection based on

the specified host:port and protocol.

Input parameter: ctx, mbedtls_net_context structure pointer, which points to

the network context object.

host, the name of the host to be connected.

port, the port number of the host to be connected.

proto, the specified protocol type, UDP(MBEDTLS_NET_PROTO_UDP)

or TCP (MBEDTLS_NET_PROTO_TCP).

Output parameter: None.

Return value: 0 indicates successful connection establishment and a non-0
value indicates failure.

3.4.2. mbedtls_ssl_set_bio

Prototype: void mbedtls_ssl_set_bio (mbedtls_ssl_context *ssl,
void *p_bio,
mbedtls_ssl_send_t *f_send,
mbedtls_ssl_recv_t *f_recv,
mbedtls_ssl_recv_timeout_t *f_recv_timeout)

Purpose: The function is used to set the read and write functions for the network level.

Input parameter: ssl, mbedtls_ssl_context structure pointer, which points to
the SSL context object.
p_bio, parameters of the read and write functions.
f_send, the write callback function.
f_recv, the read callback function.
f_recv_timeout, the blocking callback function.

Output parameter: None.

Return value: None.

For TLS, provide either f_recv or f_recv_timeout. If both are available, f_recv_timeout is used by default. For DTLS, provide either f_recv_timeout or non-blocking f_recv.

There are three corresponding functions in MSDK\mbedtls\mbedtls-2.17.0-ssl\library\net_sockets.c, which are mbedtls_net_send(), mbedtls_net_recv(), and mbedtls_net_recv_timeout().

3.4.3. mbedtls_ssl_handshake

Prototype: int mbedtls_ssl_handshake (mbedtls_ssl_context *ssl)

Purpose: The function is used to perform SSL handshake.

Input parameter: ssl, mbedtls_ssl_context structure pointer, which points to
the SSL context object.

Output parameter: None.

Return value: 0 indicates successful handshake and a non-0 value indicates failure.

3.4.4. mbedtls_ssl_get_verify_result

Prototype: uint32_t mbedtls_ssl_get_verify_result (const mbedtls_ssl_context *ssl)

Purpose: The function is used to obtain the certificate verification result.

Input parameter: ssl, mbedtls_ssl_context structure pointer, which points to
the SSL context object.

Output parameter: None.

Return value: 0 indicates successful certificate verification and a non-0 value indicates failure.

3.4.5. mbedtls_x509_crt_verify_info

Prototype: int mbedtls_x509_crt_verify_info (char *buf, size_t size, const char *prefix,
uint32_t flags)

Purpose: The function is used to obtain the certificate verification status information.
Usually, certificate verification failure information is obtained
when the mbedtls_ssl_get_verify_result() function returns a non-0 value.

Input parameter: size, the size of the output buffer.
prefix, the line prefix.
flags, the return value of the mbedtls_ssl_get_verify_result() function.

Output parameter: buf, the buffer that stores the verification status information string.

Return value: The length of the verification status information string written to
the buffer (excluding the terminator) or a negative error code.

3.5. Read and write APIs

3.5.1. mbedtls_ssl_write

Prototype: int mbedtls_ssl_write (mbedtls_ssl_context *ssl, const unsigned char *buf,
size_t len)

Purpose: The function is used to write data with a length of up to 'len' bytes to SSL.

Input parameter: ssl, mbedtls_ssl_context structure pointer, which points to
the SSL context object.

buf, the buffer to which data is to be written.

len, the length of the data to be written.

Output parameter: None.

Return value: A non-negative number indicates the actual length of written data,

and a negative number indicates other SSL specified error codes.

3.5.2. mbedtls_ssl_read

Prototype: int mbedtls_ssl_read (mbedtls_ssl_context *ssl, unsigned char *buf, size_t len)

Purpose: The function is used to read data with a length of up to 'len' bytes from SSL.

Input parameter: ssl, mbedtls_ssl_context structure pointer, which points to

the SSL context object.

buf, the buffer that receives the read data.

len, the length of the data to be read.

Output parameter: None.

Return value: A positive number indicates the length of read data; 0 indicates that the terminator is read; a negative number indicates other SSL-specified error codes.

3.6. Disconnection and resource release APIs

3.6.1. mbedtls_ssl_close_notify

Prototype: int mbedtls_ssl_close_notify (mbedtls_ssl_context *ssl)

Purpose: The function is used to notify the counterparty that the connection is being closed.

Input parameter: ssl, mbedtls_ssl_context structure pointer, which points to

the SSL context object.

Output parameter: None.

Return value: 0 indicates success and a non-0 value indicates failure.

3.6.2. mbedtls_net_free

Prototype: void mbedtls_net_free (mbedtls_net_context *ctx)

Purpose: The function is used to disconnect from the counterparty and

release related resources.

Input parameter: ctx, mbedtls_net_context structure pointer, which points to

the network context object.

Output parameter: None.

Return value: None.

3.6.3. mbedtls_x509_crt_free

Prototype: void mbedtls_x509_crt_free (mbedtls_x509_crt *crt)

Purpose: The function is used to release certificate data.

Input parameter: crt, mbedtls_x509_crt structure pointer, which points to

the x509 certificate object.

Output parameter: None.

Return value: None.

3.6.4. mbedtls_pk_free

Prototype: void mbedtls_pk_free (mbedtls_pk_context *ctx)

Purpose: The function is used to release public key data.

Input parameter: ctx, mbedtls_pk_context structure pointer, which points to

the public key context object.

Output parameter: None.

Return value: None.

3.6.5. mbedtls_ssl_free

Prototype: void mbedtls_ssl_free (mbedtls_ssl_context *ssl)

Purpose: The function is used to release the SSL context.

Input parameter: ssl, mbedtls_ssl_context structure pointer, which points to

the SSL context object.

Output parameter: None.

Return value: None.

3.6.6. mbedtls_ssl_config_free

Prototype: void mbedtls_ssl_config_free (mbedtls_ssl_config *conf)

Purpose: The function is used to release SSL configuration.

Input parameter: conf, mbedtls_ssl_config structure pointer, which points to SSL configuration.

Output parameter: None.

Return value: None.

3.7. Code example

For HTTPS Client examples, see MSDK\mbedtls\demo\ssl_client.c and MSDK\mbedtls\demo\ ssl_certs.c.

The certificate verification mode is used in the examples, which supports two certificate verification levels and is configured through the mbedtls_ssl_conf_authmode() function. One level is MBEDTLS_SSL_VERIFY_NONE, at which the validity of certificate is not verified, so it is not secure and not recommended. The other level is MBEDTLS_SSL_VERIFY_REQUIRED, at which the certificate verification must be successful before the next step can be done.

At the same time, the examples also provide several HTTPS Request methods such as GET, HEAD, and POST to interact with the server.

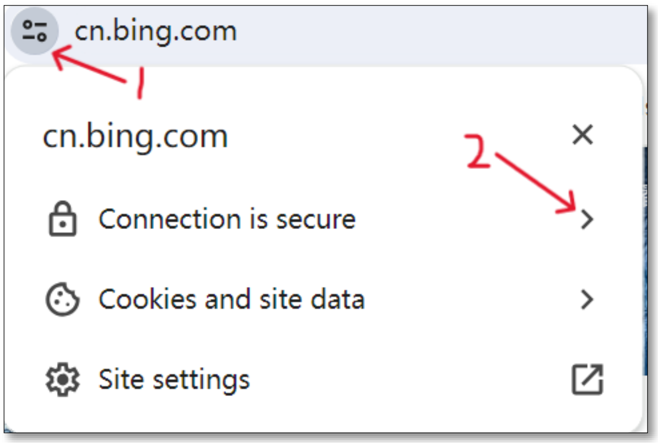
3.8. Certificate acquisition

3.8.1. Server certificate

If users build their own server, they can use OpenSSL to create and generate a certificate.

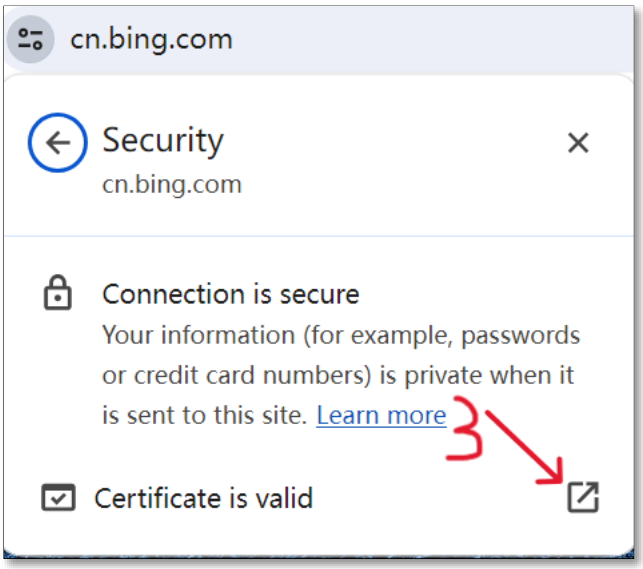
If the service provider provides the server, users can directly contact the service provider to obtain the base64 or x.509-encoded certificate file in PEM format, or export it from the service provider's website. The following is an example of using the Chrome browser to view the Bing website certificate.

Figure 3-1. Certificate exported from the service provider' website (1)



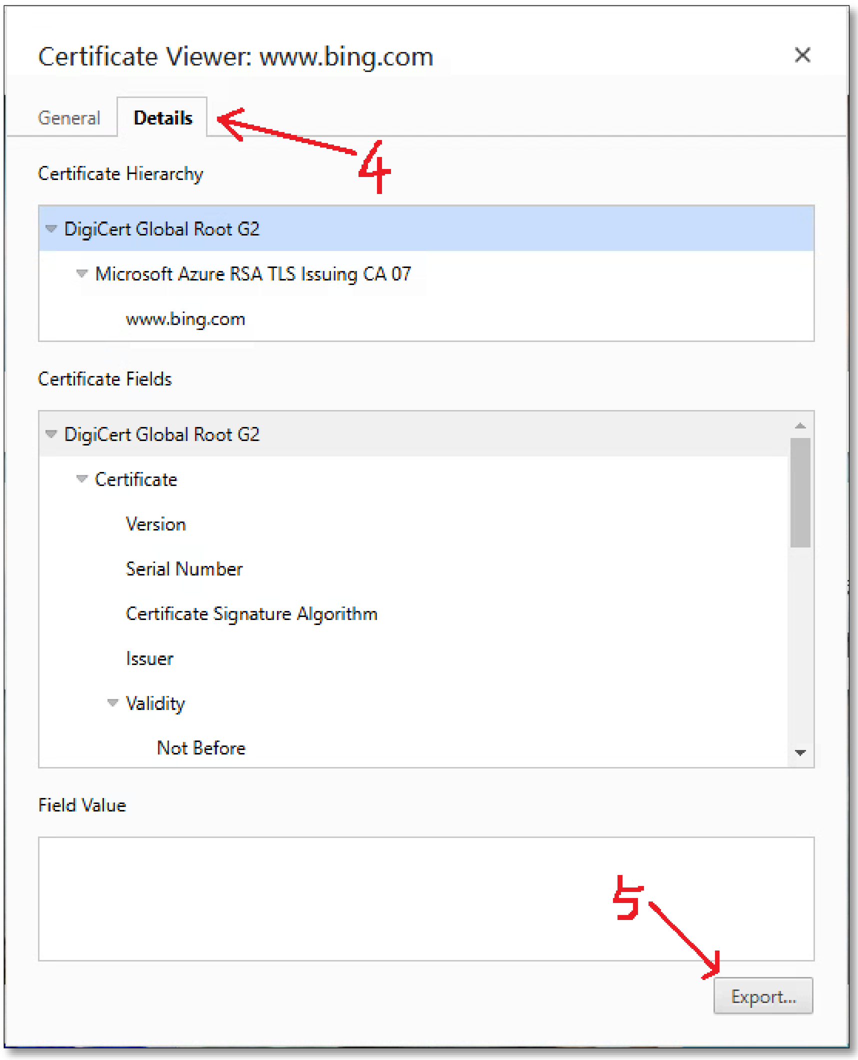
First click the "lock" symbol on the left side of the address bar, and then click the "arrow" symbol on the right side of the "Connection is secure" column to enter the ***Figure 3-2. Certificate exported from the service provider' website (2)*** interface.

Figure 3-2. Certificate exported from the service provider' website (2)


cn.bing.com
Security
cn.bing.com
Connection is secure
Your information (for example, passwords or credit card numbers) is private when it is sent to this site. Learn more
3
Certificate is valid

Click the "arrow" symbol on the right side of the "Certificate is valid" column to enter the ***Figure 3-3. Certificate exported from the service provider' website (3)*** interface.

Figure 3-3. Certificate exported from the service provider' website (3)

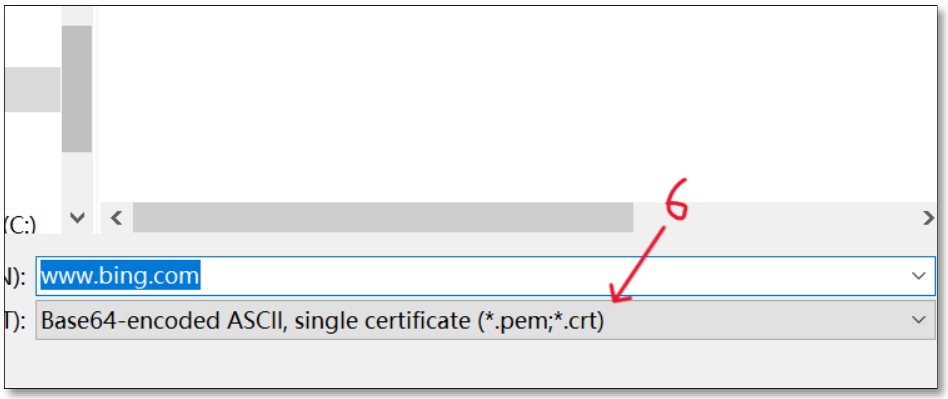

Certificate Viewer: www.bing.com

- DigiCert Global Root G2
  - Microsoft Azure RSA TLS Issuing CA 07
    - www.bing.com

- DigiCert Global Root G2
  - Certificate
    - Version
    - Serial Number
    - Certificate Signature Algorithm
    - Issuer
    - Validity
      - Not Before

First click "Details" at the top of the interface, and then click "Export" at the bottom right to enter the ***Figure 3-4. Certificate exported from the service provider' website (4)*** interface.

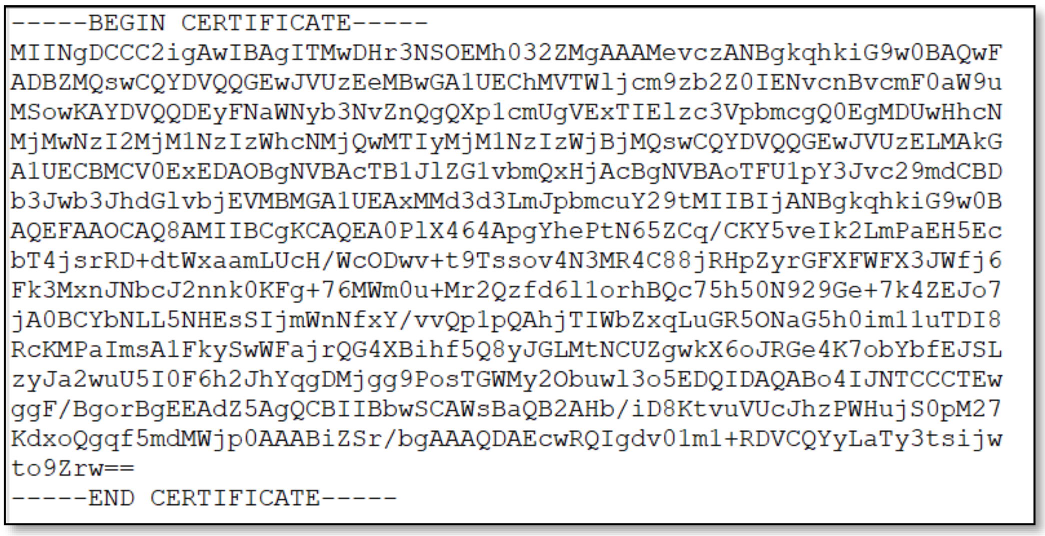
Figure 3-4. Certificate exported from the service provider' website (4)



N): www.bing.com
T): Base64-encoded ASCII, single certificate (*.pem;*.crt)

Select the "Base64 encoded" format as the saving type, and then save it. Open the saved file with a text editor. The content is as shown in ***Figure 3-5. Certificate exported from the service provider' website (5)***, starting with "-----BEGIN CERTIFICATE-----" and ending with "-----END CERTIFICATE-----". Note that the certificate in the figure has deleted content and cannot be used directly.

Figure 3-5. Certificate exported from the service provider' website (5)



```
-----BEGIN CERTIFICATE-----
MIINgDCCC2igAwIBAgITMwDHr3NSOEMh032ZMgAAAMevczANBgkqhkiG9w0BAQwF
ADBZMQswCQYDVQQGEwJVUzEeMBwGA1UEChMVTWljcm9zb2Z0IENvcnBvcmF0aW9u
MSowKAYDVQQDEyFNaWNyb3NvZnQgQXplcmUgVExTIElzc3VpbmcgQ0EgMDUwHhcN
MjMwNzI2MjM1NzIzWhcNMjQwMTIyMjM1NzIzWjBjMQswCQYDVQQGEwJVUzELMAkG
A1UECBMCV0ExEDAOBgNVBAcTB1JlZG1vbmQxHjAcBgNVBAoTFU1pY3Jvc29mdCBD
b3Jwb3JhdGlvbjEVMBMGA1UEAxMMd3d3LmJpbmcuY29tMIIBIjANBgkqhkiG9w0B
AQEFAAOCAQ8AMIIBCgKCAQEA0PlX464ApgYhePtN65ZCq/CKY5veIk2LmPaEH5Ec
bT4jsrRD+dtWxaamLUcH/WcODwv+t9Tssov4N3MR4C88jRHpZyrGFXFWFX3JWfj6
Fk3MxnJNbcJ2nnk0KFg+76MWm0u+Mr2Qzfd6llorhBQc75h50N929Ge+7k4ZEJo7
jA0BCYbNLL5NHEsSIjmWnNfxY/vvQplpQAhjTIWbZxqLuGR5ONaG5h0imlluTDI8
RcKMPaImsAlFkySwWFajrQG4XBihf5Q8yJGLMtNCUZgwkX6oJRGe4K7obYbfEJSL
zyJa2wuU5I0F6h2JhYqgDMjgg9PosTGWMy2Obuwl3o5EDQIDAQABo4IJNTCCCTEw
ggF/BgorBgEEAdZ5AgQCBIIBbwSCAWsBaQB2AHb/iD8KtvuVUcJhzPWHujS0pM27
KdxoQgqf5mdMWjp0AAABiZSr/bgAAAQDAEcwRQIgdv01m1+RDVCQYyLaTy3tsijw
to9Zrw==
-----END CERTIFICATE-----
```

3.8.2. Client certificate

If users build their own client, they can use OpenSSL to create and generate a certificate.

Note that the certificate generated by using OpenSSL is for testing only.

4. MQTT

This chapter provides development guidance of MQTT, which is a lightweight message transmission protocol based on the publish/subscribe mode.

Currently, mqtt3.1.1 and mqtt5.0 are supported. The underlying interfaces are located in the following files.

\lwip\lwip-2.2.0\src\apps\mqtt.c

\lwip\lwip-2.2.0\src\apps\mqtt5.c

4.1. Initialization of configuration parameters

1. To enable MQTT, enable the macro CONFIG_MQTT in app_cfg.h.

2. To enable transmission with SSL encryption, enable the macro LWIP_SSL_MQTT (currently enabled by default).

The SDK encryption mode uses the SSL encryption suite that comes with LwIP. The relevant interface functions are located in the following files.

lwip\lwip-2.2.0\src\apps\altcp_tls\altcp_tls_mbedtls.c

lwip\lwip-2.2.0\src\apps\altcp_tls\altcp_tls_mbedtls_mem.c

lwip\lwip-2.2.0\src\apps\altcp_tls\altcp_tls_mbedtls_mem.h

lwip\lwip-2.2.0\src\apps\altcp_tls\altcp_tls_mbedtls_structs.h

The certificates and related client configurations are located in the following files.

MSDK\app\mqtt_app\mqtt_ssl_config.c

3. The configuration interface functions of the client are located in the following files.

MSDK\app\mqtt_app\mqtt_client_config.c

MSDK\app\mqtt_app\mqtt5_client_config.c

mqtt_client_config.c is mainly the basic configurations of the MQTT client, such as client ID, username and password, heartbeat, and other general MQTT client configurations. To use SSL secure encrypted transmission, users also need to configure the root certificate of the proxy service. mqtt5_client_config.c is mainly used to configure new features related to the MQTT5.0 version.

Note: The current software only provides the basic features of MQTT3.1.1 and MQTT5.0. For the configuration of enhanced features of some versions, the software writes fixed values in the corresponding locations in the two configuration files. In the later stage, users can add interfaces by themselves as needed and in combination with the protocols to flexibly modify the configurations so as to meet corresponding business needs.

4.2. Related structures

4.2.1. mqtt_connect_client_info_t

/* Basic configuration of client information and connection */

4.2.2. mqtt5_connection_property_config_t

/* mqtt5 connection configuration */

4.2.3. mqtt5_publish_property_config_t

/* mqtt5 publish message configuration */

4.2.4. mqtt5_subscribe_property_config_t

/* mqtt5 subscribe message configuration */

4.2.5. mqtt5_unsubscribe_property_config_t

/* mqtt5 unsubscribe message configuration */

4.2.6. mqtt5_disconnect_property_config_t

/* mqtt5 disconnect message configuration */

4.2.7. mqtt5_publish_resp_property_t

/* mqtt5 publish message reply configuration */

4.2.8. mqtt5_connection_property_storage_t

/* mqtt5 connection storage configuration */

4.2.9. mqtt5_connection_will_property_storage_t

/* mqtt5 connection last will storage configuration */

4.3. Basic feature API

4.3.1. mqtt5_param_cfg

Prototype: int mqtt5_param_cfg(mqtt_client_t *mqtt_client)

Purpose: Configure parameters related to MQTT5 features.

Input parameter: mqtt_client, a pointer to the mqtt_client_t client information structure.

Output parameter: None.

Return value: Configuration success (0)/failure (non-0).

4.3.2. mqtt_client_connect

Prototype: err_t mqtt_client_connect(mqtt_client_t *client,
const ip_addr_t *ip_addr, u16_t port,
mqtt_connection_cb_t cb, void *arg,
const struct mqtt_connect_client_info_t *client_info,
u8_t mutual_auth)

Purpose: It is used by mqtt3.1.1 to initiate a request to connect the client to the proxy server. The function binds the IP address and port number of the remote proxy service, registers the relevant TCP callback function, and fills the message content of the MQTT connection request into the specified buffer. After the underlying TCP handshake is successful and the TCP connection is established, execute the callback function in "TCP connect" to send the request message.

Input parameter: client, a pointer to the mqtt_client_t client information structure.
ip_addr, which points to the IP address of the proxy server.
port, the proxy server port.
cb, the connection state transition callback function, which is used by the transport layer to notify the upper layer applications when the MQTT connection state changes. This function can be customized.
arg, which points to parameters of the connection state transition

callback function.

client_info, which points to the mqtt_connect_client_info_t client connection

configuration information structure.

mutual_auth, the mutual authentication identifier.

Output parameter: None.

Return value: Execution success (ERR_OK)/failure (non-ERR_OK).

4.3.3. mqtt5_client_connect

Prototype: err_t mqtt5_client_connect(mqtt_client_t *client,

const ip_addr_t *ip_addr, u16_t port,

mqtt_connection_cb_t cb, void *arg,

const struct mqtt_connect_client_info_t *client_info,

const mqtt5_connection_property_storage_t *property,

const mqtt5_connection_will_property_storage_t *will_property,

u8_tmutual_auth)

Purpose: It is used by mqtt5.0 to initiate a request to connect the client to the proxy server. The function binds the IP address and port number of the remote proxy server, registers the relevant TCP callback function, and fills the message content of the MQTT connection request into the specified buffer. After the underlying TCP handshake is successful and the TCP connection is established, execute the callback function in "TCP connect" to send the request message.

Input parameter: client, a pointer to the mqtt_client_t client information structure.

ip_addr, which points to the IP address of the proxy server.

port, the proxy server port.

cb, the connection state transition callback function, which is used by the transport layer to notify the upper layer applications when the MQTT connection state changes. This function can be customized.

arg, which points to parameters of the connection state transition callback function.

client_info, which points to the mqtt_connect_client_info_t client connection configuration information structure.

property, which points to the mqtt5_connection_property_storage_t structure of mqtt5 connection feature configuration.

will_property, which points to the mqtt5_connection_will_property_storage_t structure and is used for mqtt5 connection last will feature configuration.

mutual_auth, the mutual authentication identifier.

Output parameter: None.

Return value: Execution success (ERR_OK)/failure (non-ERR_OK).

4.3.4. mqtt_msg_publish

Prototype: err_t mqtt_msg_publish (mqtt_client_t *client,
const char *topic,
const void *payload, u16_t payload_length,
u8_t qos, u8_t retain,
mqtt_request_cb_t cb, void *arg)

Purpose: It is used to publish the topic of mqtt3.1.1. The function fills the publish message in the mqtt3.1.1 version format through the configuration of publish message and input message content, stores it in the specified buffer, and sends it out at the appropriate time.

Input parameter: client, a pointer to the mqtt_client_t client information structure.

topic, which points to the name of the message topic to be published.

payload, which points to the content of the message topic to be published.

payload_length, the content length of the message topic to be published.

qos, the qos level of the message topic to be published.

retain, the message retention flag bit.

cb, which points to the published callback function and is used as a processing function after the message is sent successfully or the confirmation is not received after a timeout. This function can be customized.

arg, callback function parameters.

Output parameter: None.

Return value: Execution success (ERR_OK)/failure (non-ERR_OK).

4.3.5. mqtt5_msg_publish

Prototype: err_t mqtt5_msg_publish (mqtt_client_t *client,
const char *topic,
const void *payload, u16_t payload_length,
u8_t qos, u8_t retain,
mqtt_request_cb_t cb, void *arg,
const mqtt5_publish_property_config_t *property,
const char *resp_info)

Purpose: It is used to publish the topic of mqtt5.0. The function fills the publish message in the mqtt5.0 version format through the configuration of publish message and input message content, stores it in the specified buffer, and sends it out at the appropriate time.

Input parameter: client, a pointer to the mqtt_client_t client information structure.

topic, which points to the name of the message topic to be published.

payload, which points to the content of the message topic to be published.

payload_length, the content length of the message topic.

qos, the qos level of the topic to be published.

retain, the message retention flag bit.

cb, which points to the published callback function and is used as a processing function after the message is sent successfully or the confirmation is not received after a timeout. This function can be customized.

arg, callback function parameters.

property, which points to the mqtt5_publish_property_config_t structure of mqtt5 publish feature configuration.

resp_info, which points to the name of the message topic to be replied.

Output parameter: None.

Return value: Execution success (ERR_OK)/failure (non-ERR_OK).

4.3.6. mqtt_sub_unsub

Prototype: err_t mqtt_sub_unsub(mqtt_client_t *client,

const char *topic,

u8_t qos, u8_t retain，

mqtt_request_cb_t cb, void *arg,

u8_t sub)

Purpose: It is used to subscribe to/unsubscribe from the topic of mqtt3.1.1.

The function fills the subscribe/unsubscribe message in the mqtt3.1.1 version format through the configuration of publish message and input message content. The two are distinguished mainly based on the last parameter sub, that is, if the flag bit is 1, it is a subscribe message, and if the flag bit is 0, it is an unsubscribe message. Then the function will store it in the specified buffer, and send it out at the appropriate time.

Input parameter: client, a pointer to the mqtt_client_t client information structure.

topic, which points to the name of the message topic to be subscribed to/unsubscribed from.

qos, the qos level of the message topic to be subscribed to/unsubscribed from.

retain, the message retention flag bit.

cb, which points to the publish callback function and is used as a processing function after the message is sent successfully or the confirmation is not received after a timeout.

arg, callback function parameters.

sub, the subscribe/unsubscribe flag bit. When the flag bit is 1, it is a subscribe message; when the flag bit is 0, it is an unsubscribe message.

Output parameter: None.

Return value: Execution success (ERR_OK)/failure (non-ERR_OK).

4.3.7. mqtt5_msg_subscribe

Prototype: err_t mqtt5_msg_subscribe(mqtt_client_t *client,

mqtt_request_cb_t cb, void *arg,

const mqtt5_topic_t *topic_list, size_t size,

u8_t retain,

const mqtt5_subscribe_property_config_t *property)

Purpose: It is used to subscribe to the topic of mqtt5.0. The function fills

the subscribe message in the mqtt5.0 version format through t

he configuration of subscribe message and input subscribe message topic,

stores it in the specified buffer, and sends it out at the appropriate time.

Input parameter: client, a pointer to the mqtt_client_t client information structure.

cb, which points to the publish callback function and is used as a

processing function after the message is sent successfully or

the confirmation is not received after a timeout.

arg, callback function parameters.

topic_list, which points to the list of message topic names

to be subscribed to.

size, the length of the topic_list.

retain, the message retention flag bit.

property, which points to the mqtt5 _subscribe_property_config_t

structure of mqtt5 subscribe feature configuration.

Output parameter: None.

Return value: Execution success (ERR_OK)/failure (non-ERR_OK).

4.3.8. mqtt5_msg_unsub

Prototype: err_t mqtt5_msg_unsub (mqtt_client_t *client, const char *topic,

u8_t qos,

mqtt_request_cb_t cb, void *arg,

const mqtt5_unsubscribe_property_config_t *property)

Purpose: It is used to unsubscribe from the topic of mqtt5.0. The function fills the unsubscribe message in the mqtt5.0 version format through the configuration of unsubscribe message and input unsubscribe message topic, stores it in the specified buffer, and sends it out at the appropriate time.

Input parameter: client, a pointer to the mqtt_client_t client information structure.

topic, which points to the content of the message topic to be unsubscribed from.

qos, the qos level of the message topic to be unsubscribed from.

cb, which points to the unsubscribe message topic callback function and is used as a processing function after the message is sent successfully or the confirmation is not received after a timeout.

arg, callback function parameters.

property, which points to the mqtt5_unsubscribe_property_config_t structure of mqtt5 subscribe feature configuration.

Output parameter: None.

Return value: Execution success (ERR_OK)/failure (non-ERR_OK).

4.3.9. mqtt_disconnect

Prototype: void mqtt_disconnect (mqtt_client_t *client)

Purpose: It is used for mqtt3.1.1 to disconnect from the server.

The function fills the "disconnect" message, stores it in the specified buffer, and sends it out at the appropriate time.

Input parameter: client, a pointer to the mqtt_client_t client information structure.

Output parameter: None.

Return value: None.

4.3.10. mqtt5_disconnect

Prototype: void mqtt_disconnect (mqtt_client_t *client)

Purpose: It is used for mqtt5.0 to disconnect from the server. The function fills the "disconnect" message, stores it in the specified buffer, and sends it out at the appropriate time.

Input parameter: client, a pointer to the mqtt_client_t client information structure.

Output parameter: None.

Return value: None.

4.4. APP code example

The MQTT protocol layer is divided into mqtt (3.1.1) and mqtt5 (5.0), which are called by mqtt_cmd.c in the APP. In mqtt_cmd.c, when the application layer starts the connection, the client configures the MQTT mode to MODE_TYPE_MQTT5: mqtt_mode_type_set(MODE_TYPE_MQTT5) by default.

4.4.1. Client initialization

1. Initialization configuration of mqtt basic parameters

Configure the client connection information in the \app\mqtt_app\mqtt_client_config.c file:

struct mqtt_connect_client_info_t base_client_user_info;

Configure CA and client certificate information in the MSDK\app\mqtt_app\mqtt_ssl_config.c file.

root_CA

2. Initialization of basic parameters

When creating a task in mqtt_cmd.c, call mqtt_base_param_cfg to initialize the basic parameters.

3. Configure corresponding processing functions of publishing/subscription/subscription message receipt according to the needs of the application layer.

mqtt_pub_cb

mqtt_sub_cb

mqtt_unsub_cb

mqtt_receive_msg_print

mqtt_receive_pub_msg_print

mqtt_connect_callback

Note: Currently, only some simple print tasks are done in these cbs, and users can process the data as needed.

4.4.2. Server connection

When the MQTT client is started, the APP first creates a task named mqtt task. After preparing the parameters, mqtt5_client_connect() is used. That is, use the MQTT5.0 version to attempt a connection request to the proxy server. If the connection fails and the proxy server returns the error code showing that the version is not supported, the client will switch to MQTT3.1.1 and then try to connect to the proxy server again. The code is as follows:

Table 4-1 Example of version switching when an error of unsupported version occurs

```
if((mqtt_mode_type_get()==MODE_TYPE_MQTT5)&&(connect_fail_reason==MQTT_CONNECTIO
    N_REFUSE_PROTOCOL)) {
    mqtt5_disconnect(mqtt_client);
    mqtt_mode_type_set(MODE_TYPE_MQTT);
    app_print("MQTT: The server does not support version 5.0, now switch to version 3.1.1\r\n");
    return mqtt_connect_to_server();
};
```

If there is no IP address of the remote server currently awaiting connection, the APP also supports inputting the URL, and the program will parse it by calling mqtt_ip_prase(ip_addr_t *addr_ip, char *domain). The port of the remote server defaults to 1883, which is defined in the mqtt_cmd.h file: #define MQTT_DEFAULT_PORT 1883.

If manual input is not supported, the system will use the default port number of MQTT 1883. The server may change the port number as needed. For example, the port number of the Baidu AI Cloud MQTT server without SSL encryption is 1883 by default, and it will change to 1884 with SSL encryption.

The current APP adopts a blocking processing method, and all publish and subscribe messages of the client are managed based on their own queues. When the APP calls the connection command, it first creates an "MQTT task", configures the relevant parameters, sets to the MODE_TYPE_MQTT5 mode, and then starts the connection process by calling mqtt5_client_connect.

4.4.3. Publish message

When the client prepares the publish message, the message will first be saved in the cmd_msg_pub_list queue and wait for the mqtt task to take it out for processing:

co_list_push_back(&(msg_pub_list.cmd_msg_pub_list), &(cmd_msg_pub->hdr));

pub_msg = (publish_msg_t *) co_list_pop_front(&(msg_pub_list.cmd_msg_pub_list));

After the message is taken out, use mqtt5_msg_publish()/mqtt_msg_publish() to publish. Users can configure the parameters required for publishing by entering a command or configuring them into fixed values in advance, or modify them by adding new interfaces. The current publish message is not retained by default, that is, retain is 0. If users need to retain the message, they can manually enter 1 in the flag bit.

4.4.4. Subscribe/unsubscribe message

When the client prepares the subscribe/unsubscribe message, the message will first be saved in the cmd_msg_sub_list queue and wait for the mqtt task to take it out for processing:

co_list_push_back(&(msg_sub_list.cmd_msg_sub_list), &(cmd_msg_sub->hdr));

sub_msg = (sub_msg_t *) co_list_pop_front(&(msg_sub_list.cmd_msg_sub_list));

After the message is taken out, use mqtt5_msg_subscribe()/mqtt5_msg_unsub() /mqtt_sub_unsub() to subscribe/unsubscribe. Users can configure the parameters required for subscribing/unsubscribing by entering a command or configuring them into fixed values in advance, or modify them by adding new interfaces. The current subscribe message is not retained by default, that is, retain is 0. If users need to retain the message, they can manually enter 1 in the flag bit.

4.4.5. Automatic reconnection

When the communication conditions are not ideal, the client will be disconnected from the proxy server. Users can use this switch to configure the start of reconnection after a non-active disconnection. This is achieved by configuring the auto_reconnect value in the APP. The mqtt task determines whether reconnection needs to be started based on this value in each loop. The code is as follows:

Table 4-2 Example of automatic reconnection

```
if (mqtt_client_is_connected(mqtt_client) == false) {
    if (auto_reconnect) {
            goto connect;
}else {
          break;
    }
}
```

4.4.6. Disconnection

When the client actively starts disconnection, demo will configure mqtt_client->run as false, and then process it in the MQTT task. That is, first end the loop, and then send a disconnect request to the proxy server through mqtt5_disconnect/mqtt_disconnect as needed. After that, start to release related resources, and delete the MQTT task.

5. CoAP

This chapter describes how to use the CoAP (Constrained Application Protocol) component to implement a CoAP client and a simple CoAP server.

CoAP version: 4.3.5.

5.1. Initialization of configuration parameters

To enable CoAP, enable the macro CONFIG_COAP in app_cfg.h.

5.2. Related structures

5.2.1. coap_context_t

/* CoAP context information */

5.2.2. coap_address_t

/* CoAP address information */

5.2.3. coap_endpoint_t

/* Abstraction of virtual endpoint that can be attached to coap_context_t */

5.2.4. coap_session_t

/* Abstraction of virtual session that can be attached to coap_context_t */

5.2.5. coap_pdu_t

/* structure for CoAP PDUs */

5.2.6. coap_optlist_t

/* Representation of chained list of CoAP options to install */

5.3. Basic feature API

5.3.1. coap_new_context

Prototype: coap_context_t *coap_new_context(const coap_address_t *listen_addr)

Purpose: Creates a new coap_context_t object that will hold the CoAP stack status.

Input parameter: listen_addr, a pointer to the coap_address_t structure, used to specify the address and port to listen on.

Output parameter: None.

Return value: Return a pointer to the newly created coap_context_t structure on success, and NULL on failure.

5.3.2. coap_new_endpoint

Prototype: coap_endpoint_t *coap_new_endpoint(coap_context_t *context,
const coap_address_t *listen_addr,
coap_proto_t proto)

Purpose: Create a new endpoint for communicating with peers.

Input parameter: context, the coap context that will own the new endpoint.

listen_addr, address the endpoint will listen for incoming requests.

proto, protocol used on this endpoint.

Output parameter: None.

Return value: Return a pointer to the newly created coap_endpoint_t structure on success, and NULL on failure.

5.3.3. coap_new_client_session

Prototype: coap_session_t *coap_new_client_session(coap_context_t *ctx,
const coap_address_t *local_if,
const coap_address_t *server,
coap_proto_t proto)

Purpose: Creates a new client session to the designated server.

Input parameter: ctx, the coap context.

local_if, address of local interface.

server, the server's address.

proto, protocol used on this session.

Output parameter: None.

Return value: Return a pointer to the newly created coap_session_t structure on success, and NULL on failure.

5.3.4. coap_send

Prototype: coap_mid_t coap_send(coap_session_t *session, coap_pdu_t *pdu)

Purpose: Sends a CoAP message to given peer.

Input parameter: session, the CoAP session.

pdu, the CoAP PDU to send.

Output parameter: None.

Return value: Return the message id of the sent message on success, and COAP_INVALID_MID on failure.

5.3.5. coap_add_option

```
Prototype: size_t coap_add_option(coap_pdu_t *pdu,
                                  coap_option_num_t number,
                                  size_t len,
                                  const uint8_t *data)
```

Purpose: Adds option of given number to pdu.

Input parameter: pdu, the PDU where the option is to be added.

number, the number of the new option.

len, the length of the new option.

data, the data of the new option.

Output parameter: None.

Return value: Return the overall length of the option on success, and 0 on failure.

5.3.6. coap_free_context

Prototype: void coap_free_context(coap_context_t *context)

Purpose: CoAP stack context must be released with coap_free_context().

Input parameter: pdu, the current coap_context_t object to free off.

Output parameter: None.

Return value: None.

5.3.7. coap_io_process

Prototype: int coap_io_process(coap_context_t *ctx, uint32_t timeout_ms)

Purpose: The main I/O processing function. All pending network I/O is completed, and then optionally waits for the next input packet.

Input parameter: ctx, the CoAP context.

timeout_ms, timeout duration for waiting for events (in milliseconds). If COAP_IO_NO_WAIT, the function will return immediately.

Output parameter: None.

Return value: Return the number of milliseconds spent in function on success, and -1 on failure.

5.3.8. coap_session_release

Prototype: void coap_session_release(coap_session_t *session)

Purpose: Decrement reference counter on a session.

Input parameter: session, the CoAP session.

Output parameter: None.

Return value: None.

5.3.9. coap_delete_optlist

Prototype: void coap_delete_optlist(coap_optlist_t *optlist_chain)

Purpose: Removes all entries from the optlist_chain.

Input parameter: optlist_chain, the optlist chain to remove all the entries from.

Output parameter: None.

Return value: None.

5.3.10. coap_add_data

Prototype: int coap_add_data(coap_pdu_t *pdu,

size_t len,

const uint8_t *data)

Purpose: Adds given data to the pdu that is passed as first parameter.

Input parameter: pdu, the PDU where the data is to be added.

len, the length of the data.

data, the data to add.

Output parameter: None.

Return value: Return 1 on success, and 0 on failure.

5.3.11. coap_insert_option

Prototype: size_t coap_insert_option(coap_pdu_t *pdu, coap_option_num_t number,

size_t len, const uint8_t *data)

Purpose: Inserts option of given number in the pdu with the appropriate data.

Input parameter: pdu, the PDU where the option is to be inserted.

number, the number of the new option.

len, the length of the new option.

data, the data of the new option.

Output parameter: None.

Return value: Return the overall length of the option on success, and 0 on failure.

5.3.12. coap_add_optlist_pdu

Prototype: int coap_add_optlist_pdu(coap_pdu_t *pdu, coap_optlist_t **optlist_chain)

Purpose: The current optlist of optlist_chain is first sorted and then added to the pdu.

Input parameter: pdu, the PDU to add the options to from the chain list.

optlist_chain, the chained list of optlist to add to the pdu.

Output parameter: None.

Return value: Return 1 on success, and 0 on failure.

5.3.13. coap_register_option

Prototype: void coap_register_option(coap_context_t *ctx, uint16_t type)

Purpose: Registers the option type type with the given context object ctx.

Input parameter: ctx, the context to use.

type, the option type to register.

Output parameter: None.

Return value: None.

5.3.14. coap_uri_into_options

Prototype: int coap_uri_into_options(const coap_uri_t *uri, const coap_address_t *dst,

coap_optlist_t **optlist_chain,

int create_port_host_opt,

uint8_t *buf, size_t buflen)

Purpose: Takes a coap_uri_t and then adds CoAP options into the optlist_chain.

Input parameter: uri, the coap_uri_t object.

dst, the destination, or NULL if URI_HOST not to be added.

create_port_host_opt, 1 if port/host option to be added else 0.

buf, parameter ignored. Can be NULL.

buflen, parameter ignored.

Output parameter: optlist_chain, where to store the chain of options.

Return value: Return 0 on success, and < 0 on failure.

5.3.15. coap_register_response_handler

Prototype: void coap_register_response_handler(coap_context_t *context,

coap_response_handler_t handler)

Purpose: Registers a new message handler that is called whenever a response is received.

Input parameter: context, the context to register the handler for.

handler, the response handler to register.

Output parameter: None.

Return value: None.

5.3.16. coap_register_nack_handler

Prototype: void coap_register_nack_handler(coap_context_t *context,

coap_nack_handler_t handler)

Purpose: Registers a new message handler that is called whenever a confirmable message (request or response) is dropped after all retries have been exhausted.

Input parameter: context, the context to register the handler for.

handler, the nack handler to register.

Output parameter: None.

Return value: None.

5.3.17. coap_register_event_handler

Prototype: void coap_register_event_handler(coap_context_t *context,

coap_event_handler_t hnd)

Purpose: Registers the function hnd as callback for events from the given CoAP context.

Input parameter: context, the context to register the event handler with.

hnd, the event handler to be registered. NULL if to be de-registered.

Output parameter: None.

Return value: None.

5.3.18. coap_context_set_block_mode

Prototype: void coap_context_set_block_mode(coap_context_t *context,

uint32_t block_mode)

Purpose: Set the context level CoAP block handling bits.

Input parameter: context, the coap_context_t object.

block_mode, Zero or more COAP_BLOCK_ or'd options.

Output parameter: None.

Return value: None.

5.3.19. coap_split_uri

Prototype: int coap_split_uri(const uint8_t *str_var, size_t len, coap_uri_t *uri)

Purpose: Parses a given string into URI components.

Input parameter: str_var, the string to split up.

len, the actual length of str_var.

Output parameter: uri, the coap_uri_t object to store the result.

Return value: Return 0 on success, and < 0 on failure.

5.3.20. coap_resolve_address_info

Prototype: coap_addr_info_t *coap_resolve_address_info(const coap_str_const_t *address,

uint16_t port,

uint16_t secure_port,

uint16_t ws_port,

uint16_t ws_secure_port,

int ai_hints_flags,

int scheme_hint_bits,

coap_resolve_type_t type)

Purpose: Resolve the specified address into a set of coap_address_t that can be used to bind() (local) or connect() (remote) to.

Input parameter: address, the Address to resolve.

port, the unsecured protocol port to use.

secure_port, the secured protocol port to use.

ws_port, the unsecured WebSockets port to use.

ws_secure_port, the secured WebSockets port to use.

ai_hints_flags, AI_* Hint flags to use for internal getaddrinfo().

scheme_hint_bits, which schemes to return information for.

type, COAP_ADDRESS_TYPE_LOCAL or COAP_ADDRESS_TYPE _REMOTE.

Output parameter: None.

Return value: Return one or more linked sets of coap_addr_info_t on success, and NULL on failure.

5.3.21. coap_get_data_large

Prototype: int coap_get_data_large(const coap_pdu_t *pdu,

size_t *len,

const uint8_t **data,

size_t *offset,

size_t *total)

Purpose: Retrieves the data from a PDU.

Input parameter: pdu, the specified PDU.

Output parameter: len, the length of the current data.

data, the ptr to the current data.

offset, the offset of the current data from the start of the body comprising of many blocks.

total, the total size of the body.

Return value: Return 1 on success, and 0 on failure.

5.3.22. coap_pdu_get_code

Prototype: coap_pdu_code_t coap_pdu_get_code(const coap_pdu_t *pdu)

Purpose: Gets the PDU code associated with pdu.

Input parameter: pdu, the PDU object.

Output parameter: None.

Return value: Return the PDU code.

5.3.23. coap_resource_set_get_observable

Prototype: void coap_resource_set_get_observable(coap_resource_t *resource, int mode)

Purpose: Set whether a resource is observable.

Input parameter: resource, the CoAP resource to use.

mode, 1 if Observable is to be set, 0 otherwise.

Output parameter: None.

Return value: None.

5.3.24. coap_add_attr

Prototype: coap_attr_t *coap_add_attr(coap_resource_t *resource,

coap_str_const_t *name,

coap_str_const_t *value,

int flags)

Purpose: Registers a new attribute with the given resource.

Input parameter: resource, the resource to register the attribute with.

name, the attribute's name as a string.

value, the attribute's value as a string or NULL if none.

flags, flags for memory management

Output parameter: None.

Return value: Return a pointer to the new attribute on success, and NULL on failure.

5.3.25. coap_get_available_scheme_hint_bits

Prototype: uint32_t coap_get_available_scheme_hint_bits(int have_pki_psk, int ws_check,

coap_proto_t use_unix_proto)

Purpose: Determine and set up scheme_hint_bits for a server that can be used in a call to coap_resolve_address_info().

Input parameter: have_pki_psk, Set to 1 if PSK/PKI information is known else 0.

ws_check, Set to 1 is WebSockets is to be included in the list else 0.

use_unix_proto, Set to the appropriate protocol to use for Unix sockets.

Output parameter: None.

Return value: Return a bit mask of the available CoAP protocols (can be 0 if none).

5.3.26. coap_resource_notify_observers

Prototype: int coap_resource_notify_observers(coap_resource_t *resource,

const coap_string_t *query)

Purpose: Initiate the sending of an Observe packet for all observers of resource.

Input parameter: resource, the CoAP resource to use.

query, the Query to match against or NULL.

Output parameter: None.

Return value: Return 1 if the Observe has been triggered, 0 otherwise.

5.3.27. coap_check_notify

Prototype: void coap_check_notify(coap_context_t *context)

Purpose: Checks all known resources to see if they are dirty and then notifies subscribed observers.

Input parameter: context, the context to check for dirty resources.

Output parameter: None.

Return value: None.

5.4. APP code example

CoAP examples can refer to MSDK\lwip\libcoap\port\client-coap.c and server-coap.c.

The examples use the Libcoap APIs introduced in this chapter to implement a CoAP client and a simple CoAP server.

6. Revision history

Table 6-1. Revision history

Revision No.	Description	Date
1.0	Initial release	Jan, 26, 2024
1.1	The LwIP version has been updated from 2.1.2 to 2.2.0; The MbedTLS version has been updated from 2.17.0 to 3.6.2. Add CoAP development guide.	Mar, 27, 2025

Important Notice