

GigaDevice Semiconductor Inc.

GD32VW553 网络应用开发指南

应用笔记

AN185

1.0 版本

（2024 年 1 月）

目录

目录.....	2
图索引	5
表索引	6
1. 前言.....	7
2. LwIP Sockets 应用开发	8
2.1. 结构体.....	8
2.1.1. sockaddr_in.....	8
2.1.2. sockaddr.....	8
2.1.3. timeval	8
2.1.4. fd_set.....	9
2.2. API 接口	9
2.2.1. socket	9
2.2.2. bind	9
2.2.3. connect	10
2.2.4. listen.....	10
2.2.5. accept.....	11
2.2.6. sendto.....	11
2.2.7. send.....	11
2.2.8. recvfrom	12
2.2.9. recv	12
2.2.10. shutdown.....	13
2.2.11. close.....	13
2.2.12. setsockopt	13
2.2.13. getsockname.....	14
2.2.14. fcntl.....	14
2.2.15. select.....	15
2.3. Errno 错误码	16
2.4. LwIP Sockets 编程示例.....	17
3. MbedTLS 应用开发.....	18
3.1. 相关结构体.....	18
3.1.1. mbedtls_net_context.....	18
3.1.2. mbedtls_ssl_context.....	18
3.1.3. mbedtls_ssl_config.....	18
3.1.4. mbedtls_x509_crt.....	18
3.1.5. mbedtls_pk_context.....	18

3.2. 初始化 API	19
3.2.1. mbedtls_debug_set_threshold	19
3.2.2. mbedtls_net_init	19
3.2.3. mbedtls_ssl_init	19
3.2.4. mbedtls_ssl_config_init	19
3.2.5. mbedtls_x509_cert_init	20
3.2.6. mbedtls_pk_init	20
3.3. 配置 API	20
3.3.1. mbedtls_x509_cert_parse	20
3.3.2. mbedtls_pk_parse_key	21
3.3.3. mbedtls_ssl_conf_own_cert	21
3.3.4. mbedtls_ssl_config_defaults	21
3.3.5. mbedtls_ssl_conf_rng	22
3.3.6. mbedtls_ssl_conf_dbg	22
3.3.7. mbedtls_ssl_conf_authmode	23
3.3.8. mbedtls_ssl_conf_ca_chain	23
3.3.9. mbedtls_ssl_conf_verify	23
3.3.10. mbedtls_ssl_setup	24
3.3.11. mbedtls_ssl_set_hostname	24
3.4. 连接与握手 API	24
3.4.1. mbedtls_net_connect	24
3.4.2. mbedtls_ssl_set_bio	25
3.4.3. mbedtls_ssl_handshake	26
3.4.4. mbedtls_ssl_get_verify_result	26
3.4.5. mbedtls_x509_cert_verify_info	26
3.5. 读写 API	26
3.5.1. mbedtls_ssl_write	26
3.5.2. mbedtls_ssl_read	27
3.6. 断连及资源释放 API	27
3.6.1. mbedtls_ssl_close_notify	27
3.6.2. mbedtls_net_free	27
3.6.3. mbedtls_x509_cert_free	28
3.6.4. mbedtls_pk_free	28
3.6.5. mbedtls_ssl_free	28
3.6.6. mbedtls_ssl_config_free	28
3.7. 代码示例	29
3.8. 证书获取	29
3.8.1. 服务器证书	29
3.8.2. 客户端证书	32
4. MQTT	33

4.1.	配置参数初始化.....	33
4.2.	相关结构体.....	34
4.2.1.	mqtt_connect_client_info_t.....	34
4.2.2.	mqtt5_connection_property_config_t.....	34
4.2.3.	mqtt5_publish_property_config_t.....	34
4.2.4.	mqtt5_subscribe_property_config_t.....	34
4.2.5.	mqtt5_unsubscribe_property_config_t.....	34
4.2.6.	mqtt5_disconnect_property_config_t.....	34
4.2.7.	mqtt5_publish_resp_property_t.....	34
4.2.8.	mqtt5_connection_property_storage_t.....	34
4.2.9.	mqtt5_connection_will_property_storage_t.....	34
4.3.	基本功能 API	35
4.3.1.	mqtt5_param_cfg.....	35
4.3.2.	mqtt_client_connect.....	35
4.3.3.	mqtt5_client_connect.....	36
4.3.4.	mqtt_msg_publish.....	36
4.3.5.	mqtt5_msg_publish.....	37
4.3.6.	mqtt_sub_unsub.....	38
4.3.7.	mqtt5_msg_subscribe.....	39
4.3.8.	mqtt5_msg_unsub.....	39
4.3.9.	mqtt_disconnect.....	40
4.3.10.	mqtt5_disconnect.....	40
4.4.	APP 代码示例	40
4.4.1.	客户端初始化.....	40
4.4.2.	连线服务器.....	41
4.4.3.	发布消息.....	42
4.4.4.	订阅/取消订阅消息.....	42
4.4.5.	自动重连.....	42
4.4.6.	断开连接.....	43
5.	版本历史	44

图索引

图 3-1. 服务商网站导出证书 1	29
图 3-2. 服务商网站导出证书 2	30
图 3-3. 服务商网站导出证书 3	30
图 3-4. 服务商网站导出证书 4	31
图 3-5. 服务商网站导出证书 5	31

表索引

表 2-1 sockaddr_in 结构体.....	8
表 2-2 sockaddr 结构体.....	8
表 2-3 timeval 结构体.....	8
表 2-4 fd_set 结构体.....	9
表 2-5 bind()示例.....	10
表 2-6 fcntl 示例.....	15
表 2-7 select 示例.....	16
表 2-8 errno 示例.....	16
表 4-1 版本不支持切换版本示例.....	41
表 4-2 自动重连示例.....	42
表 5-1. 版本历史.....	44

1. 前言

本文档基于 GD32VW553 系列芯片，介绍了如何使用 SDK 集成的各种组件实现网络应用开发。

GD32VW553 系列芯片是以 RISC-V 为内核的 32 位微控制器（MCU），包含了 Wi-Fi4/ Wi-Fi6 及 BLE5.3 连接技术。GD32VW553 Wi-Fi+BLE SDK 集成 Wi-Fi 驱动、BLE 驱动、LwIP TCP/IP 协议栈和 MbedTLS 等组件，可使开发者基于 GD32VW553 快速开发物联网应用程序。

关于 GD32VW553 系列芯片的快速使用，请参考文档《GD32VW553 快速开发指南》，关于 Wi-Fi 及 BLE 应用的开发，请参考文档《GD32VW553 Wi-Fi 开发指南》和《GD32VW553 BLE 开发指南》。

2. LwIP Sockets 应用开发

本章节介绍了如何使用 LwIP Sockets API 实现 UDP Server/UDP Client/TCP Server/TCP Client。

相关 LwIP Sockets 函数声明位于以下头文件中，LwIP 为 2.1.2 版本。

MSDK\lwip\lwip-2.1.2\src\include\lwip\sockets.h

MSDK\lwip\lwip-2.1.2\src\include\lwip\priv\sockets_priv.h

2.1. 结构体

2.1.1. sockaddr_in

表 2-1 sockaddr_in 结构体

```
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

表 2-2 sockaddr 结构体

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

2.1.3. timeval

表 2-3 timeval 结构体

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


2.1.4. fd_set

表 2-4 fd_set 结构体

```
/* 套接字文件描述符集合 */
typedef struct fd_set
{
    unsigned char fd_bits [(FD_SETSIZE+7)/8];
} fd_set;
/* 以下是 fd_set 结构体操作符 */
FD_ZERO(fd_set *fdset) //将套接字文件描述符集合清空
FD_SET(int fd, fd_set *fdset) //在套接字文件描述符集合中增加一个描述符
FD_CLR(int fd, fd_set *fdset) //在套接字文件描述符集合中删除一个描述符
FD_ISSET(int fd, fd_set *fdset) //判断描述符是否在套接字文件描述符集合中
```

2.2. API 接口

2.2.1. socket

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

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

功能: 该函数用于申请一个套接字。

输入参数: domain, 套接字使用的协议族, 其中, AF_INET 对应 IPv4 协议, AF_INET6 对应 IPv6 协议。

type, 套接字使用的服务类型, SOCK_STREAM 对应 TCP, SOCK_DGRAM 对应 UDP, SOCK_RAW 对应原始套接字。

protocol, 套接字使用的特定协议, 一般使用默认值 0。

输出参数: 无。

返回值: 成功返回套接字描述符, 失败返回-1。

2.2.2. bind

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

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

功能: 该函数用于服务器端绑定套接字和网卡信息。

输入参数: s, 待绑定的服务器端套接字描述符。

name, 指向 sockaddr 结构体的指针, 其中包含了网卡的 IP 地址、端口号等信息。

这些信息保存在结构体的 `sa_data[14]` 连续 14 字节里面，使用并不友好，因此通常使用更加清晰的结构体 `sockaddr_in`。这两个结构体是等价的，只不过 `sockaddr_in` 对 `sa_data[14]` 重新进行了划分，分出了 `sin_port`，`sin_addr` 等字段。

`namelen`，`name` 结构体的长度。

输出参数：无。

返回值：成功返回 0，失败返回 -1。

表 2-5 bind() 示例

```
struct sockaddr_in server_addr;
/* 填充 sockaddr_in 结构体 */
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);
/* sockaddr_in 结构体强制类型转换 */
bind(server_fd, (struct sockaddr *)&server_addr, sizeof(server_addr));
```

2.2.3. connect

宏：`#define connect(s,name,namelen) lwip_connect(s,name,namelen)`

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

功能：该函数用于客户端，将套接字与远端的网卡信息绑定。对于 UDP，仅记录信息；对于 TCP，除了记录信息，还将发起握手过程并建立 TCP 连接。

输入参数：`s`，客户端套接字描述符。

`name`，指向 `sockaddr` 结构体的指针，保存了远端的网卡信息。

`namelen`，`name` 结构体的长度。

输出参数：无。

返回值：成功返回 0，失败返回 -1。

2.2.4. listen

宏：`#define listen(s,backlog) lwip_listen(s,backlog)`

原型：`int lwip_listen (int s, int backlog)`

功能：该函数只用于 TCP 服务器端，让服务器进入监听状态，等待远端的连接请求。

输入参数：`s`，服务器端套接字描述符。

`backlog`，请求队列的大小，只有宏 `TCP_LISTEN_BACKLOG==1` 时有效。

输出参数：无。

返回值：成功返回 0，失败返回-1。

2.2.5. **accept**

宏：`#define accept(s,addr,addrlen) lwip_accept(s,addr,addrlen)`

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

功能：该函数只用于 TCP 服务器端，等待远端的连接请求，并且建立一个新的 TCP 连接。调用该函数前需要先调用 `listen()` 函数进入监听状态。

输入参数：**s**，服务器端套接字描述符。

addrlen，**addr** 结构体的长度。

输出参数：**addr**，指向 `sockaddr` 结构体的指针，保存了远端的网卡信息。

返回值：成功返回一个代表远端的套接字描述符，失败返回-1。

2.2.6. **sendto**

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

原型：`ssize_t lwip_sendto (int s, const void *dataptr, size_t size, int flags, const struct sockaddr *to, socklen_t tolen)`

功能：该函数用于 UDP 传输数据，向远端发送 UDP 报文。

输入参数：**s**，套接字描述符。

dataptr，待发送数据的起始地址。

size，数据的长度。

flags，指定发送数据时的一些处理，如 `MSG_DONTWAIT(0x08)` 表示此次传输为非阻塞。通常设置为 0。

to，指向 `sockaddr` 结构体的指针，保存了远端的网卡信息。

tolen，**to** 结构体的长度。

输出参数：无。

返回值：成功返回发送数据的长度，失败返回-1。

2.2.7. **send**

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

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

功能: 该函数用于 UDP 和 TCP 传输数据, 向远端发送数据。由于 `send` 未指定远端信息, 所以需要在套接字处于连接状态时使用。

输入参数: `s`, 套接字描述符。

`dataptr`, 待发送数据的起始地址。

`size`, 数据的长度。

`flags`, 指定发送数据时的一些处理, 参考 `sendto`。

输出参数: 无。

返回值: 成功返回发送数据的长度, 失败返回-1。

2.2.8. `recvfrom`

宏:

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

原型: `ssize_t lwip_recvfrom (int s, void *mem, size_t len, int flags,
struct sockaddr *from, socklen_t *fromlen)`

功能: 该函数用于 UDP 和 TCP 接收数据。

输入参数: `s`, 套接字描述符。

`len`, 接收数据的最大长度。

`flags`, 指定发送数据时的一些处理, 参考 `sendto`。

`fromlen`, `from` 结构体的长度。

输出参数: `mem`, 接收数据的缓存起始地址。

`from`, 指向 `sockaddr` 结构体的指针, 保存了远端的网卡信息。

返回值: 成功返回接收数据的长度, 失败返回-1。

2.2.9. `recv`

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

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

功能: 该函数用于 UDP 和 TCP 接收数据, 实际等同于 `from` 与 `fromlen` 皆为 NULL 时的 `recvfrom` 函数。

输入参数: `s`, 套接字描述符。

len, 接收数据的最大长度。

flags, 指定发送数据时的一些处理, 参考 sendto。

输出参数: mem, 接收数据的缓存起始地址。

返回值: 成功返回接收数据的长度, 失败返回-1。

2.2.10. shutdown

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

原型: int lwip_shutdown(int s, int how)

功能: 该函数用于关闭连接, 只用于 TCP, 对于 UDP 不起作用。

输入参数: s, 套接字描述符。

how, 断开方式。SHUT_RD, 断开输入流; SHUT_WR, 断开输出流;
SHUT_RDWR, 断开输入输出流。

输出参数: 无。

返回值: 成功返回 0, 失败返回-1。

2.2.11. close

宏: #define close(s) lwip_close(s)

原型: int lwip_close(int s)

功能: 该函数用于关闭套接字。

输入参数: s, 套接字描述符。

输出参数: 无。

返回值: 成功返回 0, 失败返回-1。

2.2.12. setsockopt

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

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

功能: 该函数用于设置套接字的选项信息。

输入参数: s, 套接字描述符。

level, 选项级别。如 SOL_SOCKET 表示在套接字层; IPPROTO_IP 表示在 IP 层;

IPPROTO_TCP 表示在 TCP 层。

optname, level 层的具体选项名称。如 TCP 层, 有 TCP_NODELAY(不使用 Nagle 算法), TCP_KEEPALIVE(设置 TCP 保活时间); IP 层, 有 IP_TOS(设置服务类型), IP_TTL(设置生存时间); 套接字层, 有 SO_REUSEADDR(允许重用本地地址), SO_RCVTIMEO(设置接收数据超时时间)等。

opval, optname 选项设置的值。

optlen, opval 的长度。

输出参数: 无。

返回值: 成功返回 0, 失败返回-1。

2.2.13. getsockname

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

optname,opval,optlen)

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

功能: 该函数用于获取套接字的选项信息。

输入参数: s, 套接字描述符。

level, 选项级别。参考 setsockopt 函数。

optname, level 层的具体选项名称。参考 setsockopt 函数。

optlen, opval 的长度。

输出参数: opval, 获取到的 optname 选项设置的值。

返回值: 成功返回 0, 失败返回-1。

2.2.14. fcntl

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

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

功能: 该函数用于执行一些套接字操作。

输入参数: s, 套接字描述符。

cmd, 套接字操作。F_GETFL 用于获取套接字的属性; F_SETFL 用于设置套接字的属性。

val, cmd 为 F_GETFL 时, 此参数无效, 可设为 0; cmd 为 F_SETFL 时, val 是待设置的套接字属性。如 O_NONBLOCK, 表示套接字是非阻塞的。

输出参数：无。

返回值：cmd 为 F_SETFL 时，成功返回 0，失败返回-1。

cmd 为 F_GETFL 时，成功返回套接字的属性，失败返回-1。

表 2-6 fcntl 示例

```
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

宏：#define select (maxfdp1, readset, write set, except set, timeout) lwip_select(maxfdp1, \ readset, write set, except set, timeout)

原型：int lwip_select(int maxfdp1, fd_set *readset, fd_set *write set, fd_set *except set, struct timeval *timeout)

功能：该函数用于监视套接字的状态变化，包括读写和异常。

输入参数：maxfdp1，需要监视的最大的套接字描述符值+1，此处最大的套接字描述符值指三个套接字描述符集合 readset/ write set/ except set 中的最大值。

readset, fd_set 结构体指针，指向需要监视可读状态的套接字描述符集合，可为 NULL，表示不监视可读状态。

writeset, fd_set 结构体指针，指向需要监视可写状态的套接字描述符集合，可为 NULL，表示不监视可写状态。

exceptset, fd_set 结构体指针，指向需要监视异常状态的套接字描述符集合，可为 NULL，表示不监视异常状态。

timeout, timeval 结构体指针，指向超时时间值。当为 NULL 时，select 处于阻塞状态，直到监视的某个套接字描述符集合中有套接字发生变化才返回；当 timeout 值设为 0 时，select 处于非阻塞状态，无论监视的套接字描述符集合中是否有套接字发生变化都直接返回；当 timeout 值大于 0 时，select 在 timeout 时间内阻塞，超时时间内有套接字发生变化或是到达超时时间返回。

输出参数：无。

返回值：负值，select 出错；正值，监视到有套接字描述符发生状态变化；0，超时或没有监视到变化。

表 2-7 select 示例

```
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) {
    /* select 执行后套接字描述符集合会发生变化，所以需要重新初始化并设置监视的描述符 */
    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 错误码

上述 API 使用过程中，如果执行出错返回值通常是-1，无法提供具体错误信息。LwIP 使用了一个全局变量 `errno`，当 API 出错时，`errno` 将通过 `sock_set_errno()` 进行错误码赋值，因此在 API 出错位置读取 `errno` 即可判断具体错误原因。

错误码位于 `MSDK\lwip\lwip-2.1.2\src\include\lwip\errno.h` 文件中，使用示例如下：

表 2-8 errno 示例

```
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 编程示例

LwIP Sockets 编程示例可以参考 MSDK\lwip\lwip-2.1.2\demo\lwip_sockets_demo.c 文件。

示例使用本章介绍的 LwIP Sockets API 实现了 UDP Server/UDP Client/TCP Server/TCP Client。它们均可以与远端进行通信。

3. MbedTLS 应用开发

本章节介绍了如何使用 MbedTLS 组件实现一个 HTTPS Client，该 Client 可以访问 HTTPS Server 并与其进行交互。

相关 MbedTLS 接口函数声明位于以下头文件中，MbedTLS 为 2.17.0 版本。

ROM-EXPORT\mbedtls-2.17.0-rom\include\mbedtls\ssl.h

ROM-EXPORT\mbedtls-2.17.0-rom\include\mbedtls\net_sockets.h

ROM-EXPORT\mbedtls-2.17.0-rom\include\mbedtls\x509_crt.h

ROM-EXPORT\mbedtls-2.17.0-rom\include\mbedtls\pk.h

MSDK\mbedtls\mbedtls-2.17.0-ssl\include\mbedtls\debug.h

3.1. 相关结构体

3.1.1. mbedtls_net_context

/* TLS 网络上下文 */

```
typedef struct mbedtls_net_context mbedtls_net_context;
```

3.1.2. mbedtls_ssl_context

/* SSL 上下文 */

```
typedef struct mbedtls_ssl_context mbedtls_ssl_context;
```

3.1.3. mbedtls_ssl_config

/* SSL 配置 */

```
typedef struct mbedtls_ssl_config mbedtls_ssl_config;
```

3.1.4. mbedtls_x509_crt

/* x509 证书结构体 */

```
typedef struct mbedtls_x509_crt mbedtls_x509_crt;
```

3.1.5. mbedtls_pk_context

/* 公钥上下文 */

```
typedef struct mbedtls_pk_context mbedtls_pk_context;
```

3.2. 初始化 API

3.2.1. mbedtls_debug_set_threshold

原型: `void mbedtls_debug_set_threshold (int threshold)`

功能: 该函数用于设置调试日志的输出等级。

输入参数: `threshold`, 待设置的 `debug level`。

输出参数: 无。

返回值: 无。

该函数用于调试, 使用时需要打开宏 `MBEDTLS_DEBUG_C`, 该宏位于 `ROM-EXPORT\mbedtls-2.17.0-rom\include\mbedtls\config.h` 中。

3.2.2. mbedtls_net_init

原型: `void mbedtls_net_init (mbedtls_net_context *ctx)`

功能: 该函数用于初始化 TLS 网络上下文。

输入参数: `ctx`, `mbedtls_net_context` 结构体指针, 指向网络上下文对象。

输出参数: 无。

返回值: 无。

3.2.3. mbedtls_ssl_init

原型: `void mbedtls_ssl_init (mbedtls_ssl_context *ssl)`

功能: 该函数用于初始化 SSL 上下文。

输入参数: `ssl`, `mbedtls_ssl_context` 结构体指针, 指向 SSL 上下文对象。

输出参数: 无。

返回值: 无。

3.2.4. mbedtls_ssl_config_init

原型: `void mbedtls_ssl_config_init (mbedtls_ssl_config *conf)`

功能: 该函数用于初始化 SSL 配置。

输入参数: `conf`, `mbdttls_ssl_config` 结构体指针, 指向 SSL 配置。

输出参数: 无。

返回值: 无。

3.2.5. `mbdttls_x509_crt_init`

原型: `void mbdttls_x509_crt_init (mbdttls_x509_crt *crt)`

功能: 该函数用于初始化根证书链表。

输入参数: `crt`, `mbdttls_x509_crt` 结构体指针, 指向 x509 证书对象。

输出参数: 无。

返回值: 无。

3.2.6. `mbdttls_pk_init`

原型: `void mbdttls_pk_init (mbdttls_pk_context *ctx)`

功能: 该函数用于初始化公钥上下文。

输入参数: `ctx`, `mbdttls_pk_context` 结构体指针, 指向公钥上下文对象。

输出参数: 无。

返回值: 无。

3.3. 配置 API

3.3.1. `mbdttls_x509_crt_parse`

原型: `int mbdttls_x509_crt_parse (mbdttls_x509_crt *chain, const unsigned char *buf, size_t buflen)`

功能: 该函数用于解析 `buf` 中一个或多个证书并把它们添加到根证书链表。

输入参数: `chain`, `mbdttls_x509_crt` 结构体指针, 指向 x509 证书对象。

`buf`, 指向存储根证书的 `buffer`。

`buflen`, 存储根证书的 `buffer` 大小。

输出参数: 无。

返回值: 0 表示解析成功, 非 0 失败。

3.3.2. mbedtls_pk_parse_key

原型: `int mbedtls_pk_parse_key (mbedtls_pk_context *ctx,
 const unsigned char *key, size_t keylen,
 const unsigned char *pwd, size_t pwrlen)`

功能: 该函数用于解析公钥并将其添加到公钥上下文。

输入参数: `ctx`, `mbedtls_pk_context` 结构体指针, 指向公钥上下文对象。

`key`, 指向存储公钥的 `buffer`。

`keylen`, 存储公钥的 `buffer` 大小。

`pwd`, 指向存储解密密码的 `buffer`, 可为 `NULL`, 表示公钥未加密。

`pwrlen`, 存储解密密码的 `buffer` 大小, 当 `pwd` 为 `NULL` 时此参数被忽略。

输出参数: 无。

返回值: 0 表示解析成功, 非 0 失败。

3.3.3. mbedtls_ssl_conf_own_cert

原型: `int mbedtls_ssl_conf_own_cert (mbedtls_ssl_config *conf,
 mbedtls_x509_crt *own_cert,
 mbedtls_pk_context *pk_key)`

功能: 该函数用于关联自身的证书链及公钥。

输入参数: `conf`, `mbedtls_ssl_config` 结构体指针, 指向 SSL 配置。

`own_cert`, `mbedtls_x509_crt` 结构体指针, 指向 x509 证书对象。

`pk_key`, `mbedtls_pk_context` 结构体指针, 指向公钥上下文对象。

输出参数: 无。

返回值: 0 表示关联成功, 非 0 失败。

3.3.4. mbedtls_ssl_config_defaults

原型: `int mbedtls_ssl_config_defaults (mbedtls_ssl_config *conf,
 int endpoint, int transport, int preset)`

功能: 该函数用于加载默认的 SSL 配置。

输入参数: `conf`, `mbedtls_ssl_config` 结构体指针, 指向 SSL 配置。

endpoint, 设置 SSL 为客户端还是服务器, MBEDTLS_SSL_IS_CLIENT 或者 MBEDTLS_SSL_IS_SERVER。

transport, 使用 TLS(MBEDTLS_SSL_TRANSPORT_STREAM) 或者使用 DTLS(MBEDTLS_SSL_TRANSPORT_DATAGRAM)。

preset, 默认使用 MBEDTLS_SSL_PRESET_DEFAULT。

输出参数: 无。

返回值: 0 表示配置成功, 非 0 失败。

3.3.5. mbedtls_ssl_conf_rng

原型: void mbedtls_ssl_conf_rng (mbedtls_ssl_config *conf,
int (*f_rng) (void *, unsigned char *, size_t),
void *p_rng)

功能: 该函数用于设置随机数生成器回调。

输入参数: conf, mbedtls_ssl_config 结构体指针, 指向 SSL 配置。

f_rng, 随机数生成器函数。

p_rng, 随机数生成器函数参数。

输出参数: 无。

返回值: 无。

f_rng 参考 MSDK\mbedtls\demo\ssl_client.c 文件内的 my_random() 函数。

3.3.6. mbedtls_ssl_conf_dbg

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

功能: 该函数用于设置 debug 回调。

输入参数: conf, mbedtls_ssl_config 结构体指针, 指向 SSL 配置。

f_dbg, debug 函数。

p_dbg, debug 函数参数。

输出参数: 无。

返回值: 无。

f_dbg 参考 MSDK\mbedtls\demo\ssl_client.c 文件内的 my_debug () 函数。

3.3.7. mbedtls_ssl_conf_authmode

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

功能: 该函数用于设置证书验证模式。

输入参数: conf, mbedtls_ssl_config 结构体指针, 指向 SSL 配置。

authmode, 证书验证模式。如下,

MBEDTLS_SSL_VERIFY_NONE, 不检查证书。服务器默认值, 如果是客户端表明连接是不安全的。

MBEDTLS_SSL_VERIFY_OPTIONAL, 检查对等证书, 但即使验证失败, 握手也会继续。

MBEDTLS_SSL_VERIFY_REQUIRED, 检查对等证书, 验证失败握手将终止, 客户端默认值。

输出参数: 无。

返回值: 无。

3.3.8. mbedtls_ssl_conf_ca_chain

原型: void mbedtls_ssl_conf_ca_chain (mbedtls_ssl_config *conf,

mbedtls_x509_crt *ca_chain,

mbedtls_x509_crl *ca_crl)

功能: 该函数用于设置验证证书所需要的数据。

输入参数: conf, mbedtls_ssl_config 结构体指针, 指向 SSL 配置。

ca_chain, 受信的 CA 证书链, 保存在 mbedtls_x509_crt 结构体中。

ca_crl, 受信的 CA CRLs, 保存在 mbedtls_x509_crl 结构体中, 可为 NULL。

输出参数: 无。

返回值: 无。

3.3.9. mbedtls_ssl_conf_verify

原型: void mbedtls_ssl_conf_verify (mbedtls_ssl_config *conf,

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

void *p_vrfy)

功能：该函数用于设置证书验证回调。

输入参数：**conf**，**mbbedtls_ssl_config** 结构体指针，指向 SSL 配置。

f_vrfy，验证回调函数。

p_vrfy，验证回调函数参数。

输出参数：无。

返回值：无。

f_vrfy 建议使用 MSDK\mbbedtls\demo\ssl_client.c 文件内的 **my_verify()** 函数，用户也可进行修改。

3.3.10. **mbbedtls_ssl_setup**

原型：**int mbedtls_ssl_setup** (**mbbedtls_ssl_context** *ssl,
 const mbedtls_ssl_config *conf)

功能：该函数用于将 SSL 配置设置到 SSL 上下文中，并初始化 handshake。

输入参数：**ssl**，**mbbedtls_ssl_context** 结构体指针，指向 SSL 上下文对象。

conf，**mbbedtls_ssl_config** 结构体指针，指向 SSL 配置。

输出参数：无。

返回值：0 表示配置成功，非 0 失败。

3.3.11. **mbbedtls_ssl_set_hostname**

原型：**int mbedtls_ssl_set_hostname** (**mbbedtls_ssl_context** *ssl, **const char** *hostname)

功能：该函数用于设置主机名。

输入参数：**ssl**，**mbbedtls_ssl_context** 结构体指针，指向 SSL 上下文对象。

hostname，主机名。主机名必须与服务器证书对应。

输出参数：无。

返回值：0 表示设置成功，非 0 失败。

3.4. 连接与握手 API

3.4.1. **mbbedtls_net_connect**

原型：**int mbedtls_net_connect** (**mbbedtls_net_context** *ctx, **const char** *host,

```
const char *port, int proto)
```

功能：该函数用于根据指定的 `host:port` 及 `protocol` 建立网络连接。

输入参数：`ctx`, `MBEDTLS_NET_CONTEXT` 结构体指针，指向网络上下文对象。

`host`, 待连接主机名。

`port`, 待连接的主机端口号。

`proto`, 指定的协议类型，`UDP(MBEDTLS_NET_PROTO_UDP)` 或者 `TCP(MBEDTLS_NET_PROTO_TCP)`。

输出参数：无。

返回值：0 表示建立连接成功，非 0 失败。

3.4.2. `MBEDTLS_SSL_SET_BIO`

原型：`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)`

功能：该函数用于为网络层设置读写函数。

输入参数：`ssl`, `MBEDTLS_SSL_CONTEXT` 结构体指针，指向 SSL 上下文对象。

`p_bio`, 读写函数的参数。

`f_send`, 写回调函数。

`f_recv`, 读回调函数。

`f_recv_timeout`, 阻塞回调函数。

输出参数：无。

返回值：无。

对于 TLS, `f_recv` 与 `f_recv_timeout` 提供其一即可，如果都有，默认使用 `f_recv_timeout`。

对于 DTLS, 要么提供 `f_recv_timeout`, 要么提供非阻塞的 `f_recv`。

MSDK\mbedtls\mbedtls-2.17.0-ssl\library\net_sockets.c 中提供了三个对应函数，`mbedtls_net_send()`, `mbedtls_net_recv()`和 `mbedtls_net_recv_timeout()`。

3.4.3. **mbbedtls_ssl_handshake**

原型: `int mbedtls_ssl_handshake (mbedtls_ssl_context *ssl)`

功能: 该函数用于执行 SSL 握手。

输入参数: `ssl`, `mbedtls_ssl_context` 结构体指针, 指向 SSL 上下文对象。

输出参数: 无。

返回值: 0 表示握手成功, 非 0 失败。

3.4.4. **mbbedtls_ssl_get_verify_result**

原型: `uint32_t mbedtls_ssl_get_verify_result (const mbedtls_ssl_context *ssl)`

功能: 该函数用于获取证书验证结果。

输入参数: `ssl`, `mbedtls_ssl_context` 结构体指针, 指向 SSL 上下文对象。

输出参数: 无。

返回值: 0 表示证书验证成功, 非 0 失败。

3.4.5. **mbbedtls_x509_crt_verify_info**

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

功能: 该函数用于获取证书验证状态的信息, 通常在 `mbedtls_ssl_get_verify_result()` 函数返回非 0 时获取证书验证失败的信息。

输入参数: `size`, 输出 `buffer` 的大小。

`prefix`, 行前缀。

`flags`, `mbedtls_ssl_get_verify_result()` 函数的返回值。

输出参数: `buf`, 存储验证状态信息字符串的 `buffer`。

返回值: 写到 `buffer` 的验证状态信息字符串的长度 (不包括结束符) 或者负的错误代码。

3.5. 读写 API

3.5.1. **mbbedtls_ssl_write**

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

功能: 该函数用于向 SSL 写入数据, 最多写入 'len' 字节长度的数据。

输入参数: `ssl`, `mbdttls_ssl_context` 结构体指针, 指向 SSL 上下文对象。

`buf`, 待写入数据的 `buffer`。

`len`, 待写入数据的长度。

输出参数: 无。

返回值: 非负数表示实际写入的数据长度, 负数表示其他 SSL 指定的错误码。

3.5.2. `mbdttls_ssl_read`

原型: `int mbdttls_ssl_read (mbdttls_ssl_context *ssl, unsigned char *buf, size_t len)`

功能: 该函数用于从 SSL 读取数据, 最多读取 ‘len’ 字节长度的数据。

输入参数: `ssl`, `mbdttls_ssl_context` 结构体指针, 指向 SSL 上下文对象。

`buf`, 接收读取数据的 `buffer`。

`len`, 要读取的数据长度。

输出参数: 无。

返回值: 正数表示读取到的数据长度, 0 表示读取到结束符, 负数表示其他 SSL 指定的错误码。

3.6. 断连及资源释放 API

3.6.1. `mbdttls_ssl_close_notify`

原型: `int mbdttls_ssl_close_notify (mbdttls_ssl_context *ssl)`

功能: 该函数用于通知对方连接正在关闭。

输入参数: `ssl`, `mbdttls_ssl_context` 结构体指针, 指向 SSL 上下文对象。

输出参数: 无。

返回值: 0 表示成功, 非 0 失败。

3.6.2. `mbdttls_net_free`

原型: `void mbdttls_net_free (mbdttls_net_context *ctx)`

功能: 该函数用于断开与对方的连接并且释放相关资源。

输入参数: `ctx`, `mbdttls_net_context` 结构体指针, 指向网络上下文对象。

输出参数: 无。

返回值：无。

3.6.3. **mbbedtls_x509_crt_free**

原型：void mbedtls_x509_crt_free (mbedtls_x509_crt *crt)

功能：该函数用于释放证书数据。

输入参数：crt，mbedtls_x509_crt 结构体指针，指向 x509 证书对象。

输出参数：无。

返回值：无。

3.6.4. **mbbedtls_pk_free**

原型：void mbedtls_pk_free (mbedtls_pk_context *ctx)

功能：该函数用于释放公钥数据。

输入参数：ctx，mbedtls_pk_context 结构体指针，指向公钥上下文对象。

输出参数：无。

返回值：无。

3.6.5. **mbbedtls_ssl_free**

原型：void mbedtls_ssl_free (mbedtls_ssl_context *ssl)

功能：该函数用于释放 SSL 上下文。

输入参数：ssl，mbedtls_ssl_context 结构体指针，指向 SSL 上下文对象。

输出参数：无。

返回值：无。

3.6.6. **mbbedtls_ssl_config_free**

原型：void mbedtls_ssl_config_free (mbedtls_ssl_config *conf)

功能：该函数用于释放 SSL 配置。

输入参数：conf，mbedtls_ssl_config 结构体指针，指向 SSL 配置。

输出参数：无。

返回值：无。

3.7. 代码示例

HTTPS Client 示例可以参考 MSDK\mbedtls\demo\ssl_client.c 及 MSDK\mbedtls\demo\ssl_certs.c。

示例中使用证书验证方式，支持两种证书校验等级，通过 `mbedtls_ssl_conf_authmode()` 函数配置。一种是 `MBEDTLS_SSL_VERIFY_NONE`，该等级下不校证书是否合法，是不安全的，不推荐使用；另一种是 `MBEDTLS_SSL_VERIFY_REQUIRED`，该等级下证书校验必须成功才能继续下一步。

同时，示例还提供了几个 HTTPS Request 方法，如 GET, HEAD, POST 等与服务器进行交互。

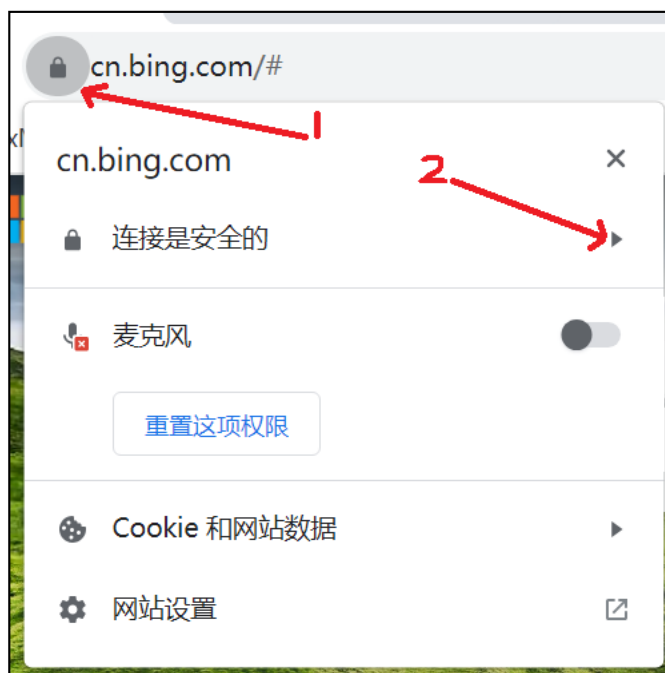
3.8. 证书获取

3.8.1. 服务器证书

如果是自搭服务器，可以使用 OpenSSL 创建生成证书。

如果是服务商的服务器，可以直接联系服务商获取 base64 编码 x.509 编码的 PEM 格式证书文件，也可以从服务商网站导出，下面以 Chrome 浏览器查看必应网站证书为例。

图 3-1. 服务商网站导出证书 1



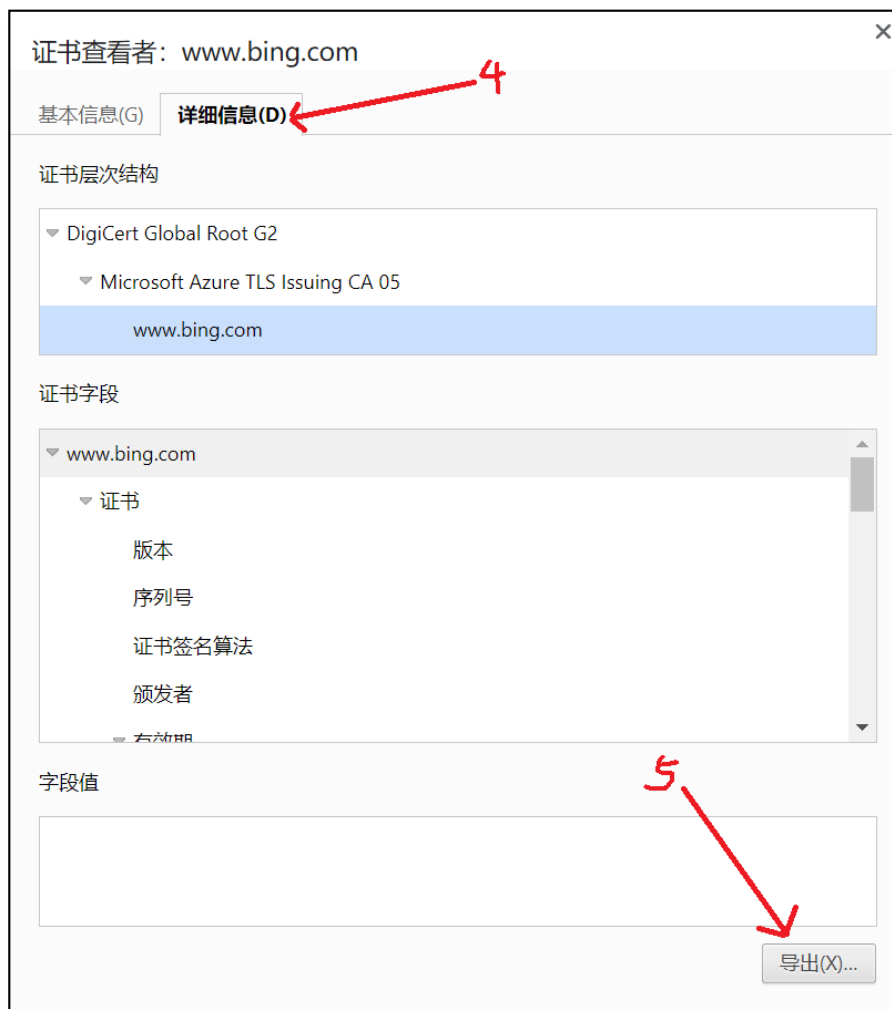
首先点击地址栏左侧“锁”状符号，然后点击“连接是安全的”一栏右侧“三角”符号，会进入[图 3-2. 服务商网站导出证书 2](#)界面。

图 3-2. 服务商网站导出证书 2



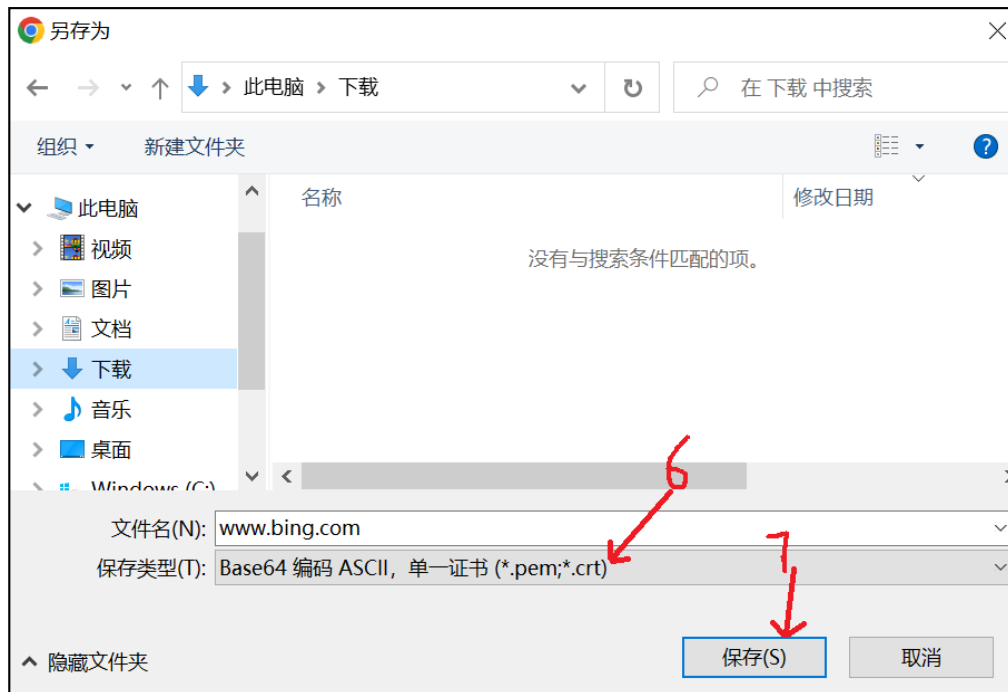
点击“证书有效”一栏右侧“箭头”符号，会进入[图 3-3. 服务商网站导出证书](#)界面。

图 3-3. 服务商网站导出证书 3



先点击界面上方“详细信息”，然后点击右下方“导出”，进入[图3-4. 服务商网站导出证书4](#)界面。

图 3-4. 服务商网站导出证书 4



保存类型选择“Base64 编码”格式，然后保存，将保存的文件使用文本编辑器打开，内容如[图3-5. 服务商网站导出证书5](#)所示，以“-----BEGIN CERTIFICATE-----”开头，以“-----END CERTIFICATE-----”结尾。注意，图中证书有作删减，不能直接使用。

图 3-5. 服务商网站导出证书 5

```
-----BEGIN CERTIFICATE-----
MIINgDCCC2igAwIBAgITMwDhr3NSOEMh032ZMgAAAMevczANBgkqhkiG9w0BAQwF
ADBZMQswCQYDVQQGEwJVUzEeMBwGA1UEChMVTWljb9zb2Z0IENvcnBvcnF0aW9u
MSowKAYDVQQDEyFNaWNYb3NvZnZnQXplcmUgVExTIElzc3VpbmcgQ0EgMDUwHhcN
MjMwNzI2MjM1NzIzWhcNMjMwNzI2MjM1NzIzWjBjMQswCQYDVQQGEwJVUzELMAkG
A1UECBMCV0ExEDAOBgNVBACTB1JlZGlvdjBmQXhAcBgNVBAoTFUlpY3Jvc29mdCBD
b3Jwb3JhdGlvbjEVMBMGA1UEAxMMd3d3LmJpbmcuY29tMIIBIjANBgkqhkiG9w0B
AQEFAAOCAQ8AMIIBCgKCAQEA0PlX464ApGyHePtN65ZCq/CKY5veIk2LmPaEH5Ec
bT4jsrRD+dtWxaamLUcH/WcODwv+t9Tssov4N3MR4C88jRHpZyrGFxFWFX3JWfj6
Fk3MxnJNbcJ2nnk0KFg+76MwM0u+Mr2Qzfd6l1orhBQc75h50N929Ge+7k4ZEJo7
jA0BCYbNLL5NHESiJmWnNfxY/vvQp1pQAhjTIWbZxqLuGR5ONaG5h0im1luTDI8
RcKMPaImsA1FkySwWFajrQG4XBihf5Q8yJGLMtNCUZgwKX6oJRGe4K7obYbfeJSL
zyJa2wuU5I0F6h2JhYqgDMjgg9PostGWMY2Obuwl3o5EDQIDAQAB04IJNTCCCTew
ggF/BgorBgEEAdZ5AgQCBIIBbWSCAwwBaQB2AHb/id8KtvuVUCJhZPWHujS0pM27
KdxoQgqf5mdMWjp0AAABiZSr/bgAAAQDAEcwRQIgdv0lml+RDVCQYyLaTy3tsijw
to9Zrw==
-----END CERTIFICATE-----
```

3.8.2. 客户端证书

自搭客户端，可以使用 OpenSSL 创建生成证书。

注意，使用 OpenSSL 生成的证书仅可测试使用。

4. MQTT

本章节介绍了 MQTT 的开发指导，它是一种基于发布/订阅模式的轻量级消息传输协议。

当前支持 mqtt3.1.1 和 mqtt5.0 两个版本，底层接口位于如下文件。

```
\\lwip\\lwip-2.1.2\\src\\apps\\mqtt.c
```

```
\\lwip\\lwip-2.1.2\\src\\apps\\mqtt5.c
```

4.1. 配置参数初始化

1、使能 MQTT，需要在 app_cfg.h 中打开宏 CONFIG_MQTT。

2、使能带 SSL 加密方式传输，需要打开宏 LWIP_SSL_MQTT（当前默认打开）。

SDK 加密方式是使用 LwIP 中自带的 SSL 加密套件，相关接口函数位于如下文件中。

```
lwip\\lwip-2.1.2\\src\\apps\\altcp_tls\\altcp_tls_mbedtls.c
```

```
lwip\\lwip-2.1.2\\src\\apps\\altcp_tls\\altcp_tls_mbedtls_mem.c
```

```
lwip\\lwip-2.1.2\\src\\apps\\altcp_tls\\altcp_tls_mbedtls_mem.h
```

```
lwip\\lwip-2.1.2\\src\\apps\\altcp_tls\\altcp_tls_mbedtls_structs.h
```

证书以及相关的客户端配置位于如下文件中。

```
\\lwip\\lwip-2.1.2\\src\\apps\\mqtt\\mqtt_ssl_config.c
```

```
lwip\\lwip-2.1.2\\src\\include\\lwip\\apps\\mqtt_ssl_config.h
```

3、客户端的配置接口函数位于以下文件中。

```
\\app\\mqtt_app\\mqtt_client_config.c
```

```
\\app\\mqtt_app\\mqtt5_client_config.c
```

其中，mqtt_client_config.c 主要是对 MQTT 客户端的基本配置，如客户端 ID、用户名密码、心跳等 MQTT 客户端通用配置。如果使用 SSL 安全加密传输，还需要配置代理服务的根证书。mqtt5_client_config.c 主要用来配置 MQTT5.0 版本相关的新增特性等。

注：当前软件只提供了 MQTT3.1.1 和 MQTT5.0 的基本功能使用，对于部分版本增强特性的配置，软件在两个配置文件中的相应位置写成了固定值，后期用户可以根据需要并结合协议自行添加接口灵活修改配置，以满足相应业务需求。

4.2. 相关结构体

4.2.1. mqtt_connect_client_info_t

/ 客户端信息和连线的基本配置 */*

4.2.2. mqtt5_connection_property_config_t

/ mqtt5 连线配置 */*

4.2.3. mqtt5_publish_property_config_t

/ mqtt5 发布消息配置 */*

4.2.4. mqtt5_subscribe_property_config_t

/ mqtt5 订阅消息配置 */*

4.2.5. mqtt5_unsubscribe_property_config_t

/ mqtt5 取消订阅消息配置 */*

4.2.6. mqtt5_disconnect_property_config_t

/ mqtt5 断开连线消息配置 */*

4.2.7. mqtt5_publish_resp_property_t

/ mqtt5 发布消息回复配置 */*

4.2.8. mqtt5_connection_property_storage_t

/ mqtt5 连线存储配置 */*

4.2.9. mqtt5_connection_will_property_storage_t

/ mqtt5 连线遗嘱存储配置 */*

4.3. 基本功能 API

4.3.1. mqtt5_param_cfg

原型: `int mqtt5_param_cfg (mqtt_client_t *mqtt_client)`

功能: 配置 MQTT5 特性功能相关参数。

输入参数: `mqtt_client`, 指向 `mqtt_client_t` 客户端信息结构体的指针。

输出参数: 无。

返回值: 配置成功 (0) / 失败 (非 0)。

4.3.2. mqtt_client_connect

原型: `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)`

功能: 用于 mqtt3.1.1 发起客户端连接代理服务器的请求。函数会绑定远端代理服务 IP 地址和端口号、注册相关的 TCP callback 函数, 并填充 MQTT 请求连接的消息内容到指定的 buffer 中。等底层 TCP 握手成功并建立 TCP 连接后, 在 TCP connect 中执行回调函数, 实现请求消息的发送。

输入参数: `client`, 指向 `mqtt_client_t` 客户端信息结构体的指针。

`ip_addr`, 指向代理服务器的 ip 地址。

`port`, 代理服务器端口。

`cb`, 连接状态转变回调函数, 当 MQTT 连接状态发生改变时, 用于传输层通知上层应用, 该函数可自定义。

`arg`, 指向连接状态转变回调函数的参数。

`client_info`, 指向 `mqtt_connect_client_info_t` 客户端连接配置信息结构体。

`mutual_auth`, 双向认证标识符。

输出参数: 无。

返回值: 执行成功 (ERR_OK) / 失败 (非 ERR_OK)。

4.3.3. mqtt5_client_connect

原型: `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_t mutual_auth)
```

功能: 用于 mqtt5.0 发起客户端连接代理服务器的请求。函数会绑定远端代理服务器 IP 地址和端口号、注册相关的 TCP callback 函数, 并填充 MQTT 请求连接的消息内容到指定的 buffer 中。等底层 TCP 握手成功并建立 TCP 连接后, 在 TCP connect 中执行回调函数, 实现消息请求的发送。

输入参数: client, 指向 mqtt_client_t 客户端信息结构体的指针。

ip_addr, 指向代理服务器的 ip 地址。

port, 代理服务器端口。

cb, 连接状态转变回调函数, 当 MQTT 连接状态发生改变时, 用于传输层通知上层应用, 该函数可自定义。

arg, 指向连接状态转变回调函数的参数。

client_info, 指向 mqtt_connect_client_info_t 客户端连接配置信息结构体。

property, 指向 mqtt5 连接特性配置的 mqtt5_connection_property_storage_t 结构体。

will_property, 指向 mqtt5_connection_will_property_storage_t 结构体, 用于 mqtt5 连接遗嘱特性配置。

mutual_auth, 双向认证标识符。

输出参数: 无。

返回值: 执行成功 (ERR_OK) / 失败 (非 ERR_OK)。

4.3.4. mqtt_msg_publish

原型: `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)

功能：用于发布 mqtt3.1.1 的主题，函数会通过发布消息配置和输入的消息内容，使用 mqtt3.1.1 版本格式填充 publish 消息，并存入指定的 buffer 中，并在合适的时间发送出去。

输入参数：client，指向 mqtt_client_t 客户端信息结构体的指针。

topic，指向待发布的消息主题名称。

payload，指向待发布的消息主题内容。

payload_length，待发布的消息主题内容长度。

qos，待发布消息主题的 qos 等级；

retain，消息保留标志位。

cb，指向发布的回调函数，用于发送成功或者超时未收到确认后的处理函数，该函数可以自定义。

arg，回调函数参数。

输出参数：无。

返回值：执行成功（ERR_OK）/失败（非 ERR_OK）。

4.3.5. mqtt5_msg_publish

原型：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)

功能：用于发布 mqtt5.0 主题，函数会通过发布消息的配置和输入的消息内容，使用 mqtt5.0 版本格式填充 publish 消息，然后存入指定的 buffer 中，并在合适的时间发送出去。

输入参数：client，指向 mqtt_client_t 客户端信息结构体的指针。

topic，指向待发布的消息主题名称。

payload，指向待发布的消息主题内容。

`payload_length`, 消息主题内容长度。

`qos`, 待发布主题的 `qos` 等级。

`retain`, 消息保留标志位。

`cb`, 指向发布后的回调函数, 用于发送成功或者超时未收到确认后的处理函数, 该函数可以自定义。

`arg`, 回调函数参数。

`property`, 指向 `mqtt5` 发布特性配置 `mqtt5_publish_property_config_t` 结构体。

`resp_info`, 指向待回复的消息主题名称。

输出参数: 无。

返回值: 执行成功 (`ERR_OK`) / 失败 (非 `ERR_OK`)。

4.3.6. `mqtt_sub_unsub`

原型: `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)`

功能: 用于订阅/取消订阅 `mqtt3.1.1` 主题, 函数会通过发布消息的配置和输入的消息内容, 使用 `mqtt3.1.1` 版本格式填充 `subscribe/unsubscribe` 消息主要是根据最后一个参数 `sub` 来区分, 如果标志位为 1 则为 `subscribe` 订阅, 如果为 0 则为 `unsubscribe` 取消订阅, 然后存入指定的 `buffer` 中, 并在合适的时间发送出去。

输入参数: `client`, 指向 `mqtt_client_t` 客户端信息结构体的指针。

`topic`, 指向待订阅/取消订阅的消息主题名称。

`qos`, 待订阅/取消订阅消息主题的 `qos` 等级。

`retain`, 消息保留标志位。

`cb`, 指向发布的回调函数, 用于发送成功或者超时未收到确认后的处理函数。

`arg`, 回调函数参数。

`sub`, 订阅/取消订阅标志位, 当标志位为 1 时, 表示为订阅消息, 为 0 时表示为取消订阅消息。

输出参数: 无。

返回值: 执行成功 (`ERR_OK`) / 失败 (非 `ERR_OK`)。

4.3.7. mqtt5_msg_subscribe

原型: `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)`

功能: 用于订阅 mqtt5.0 的主题。函数会通过订阅消息的配置和输入的订阅消息主题, 使用 mqtt5.0 版本格式填充 subscribe 消息, 然后存入指定的 buffer 中, 并在合适的时间发送出去。

输入参数: client, 指向 mqtt_client_t 客户端信息结构体的指针。

cb, 指向发布的回调函数, 用于发送成功或者超时未收到确认后的处理函数。

arg, 回调函数参数。

topic_list, 指向待订阅消息主题名称列表。

size, topic_list 列表长度。

retain, 消息保留标志位。

property, 指向 mqtt5 订阅特性配置 mqtt5_subscribe_property_config_t 结构体。

输出参数: 无。

返回值: 执行成功 (ERR_OK) / 失败 (非 ERR_OK)。

4.3.8. mqtt5_msg_unsub

原型: `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)`

功能: 用于取消订阅 mqtt5.0 的主题。函数会通过取消订阅消息的配置和输入的取消订阅消息主题, 使用 mqtt5.0 版本格式填充 unsubscribe 消息, 然后存入指定的 buffer 中, 并在合适的时间发送出去。

输入参数: client, 指向 mqtt_client_t 客户端信息结构体的指针

topic, 指向取消订阅消息主题内容。

qos, 取消订阅消息主题的等级 qos。

cb, 指向取消订阅消息主题回调函数, 用于发送成功或者超时未收到确认后的处理。

arg, 回调函数参数。

property, 指向 mqtt5 订阅特性配置 mqtt5_unsubscribe_property_config_t 结构体。

输出参数: 无。

返回值: 执行成功 (ERR_OK) / 失败 (非 ERR_OK)。

4.3.9. mqtt_disconnect

原型: void mqtt_disconnect (mqtt_client_t *client)

功能: 用于 mqtt3.1.1 断开与服务器的连接。函数会填充 disconnect 消息, 然后存入指定的 buffer 中, 并在合适的时间发送出去。

输入参数: client, 指向 mqtt_client_t 客户端信息结构体的指针。

输出参数: 无。

返回值: 无。

4.3.10. mqtt5_disconnect

原型: void mqtt_disconnect (mqtt_client_t *client)

功能: 用于 mqtt5.0 断开与服务器的连接。函数会填充 disconnect 消息, 然后存入指定的 buffer 中, 并在合适的时间发送出去。

输入参数: client, 指向 mqtt_client_t 客户端信息结构体的指针。

输出参数: 无。

返回值: 无。

4.4. APP 代码示例

MQTT 协议层分为 mqtt (3.1.1) 和 mqtt5 (5.0), APP 中统一由 mqtt_cmd.c 负责调用。mqtt_cmd.c 中, 当应用层启动连接时, 客户端默认将 MQTT 模式配置为 MODE_TYPE_MQTT5: mqtt_mode_type_set(MODE_TYPE_MQTT5);

4.4.1. 客户端初始化

1、mqtt 基本参数初始化配置

在文件\app\mqtt_app\mqtt_client_config.c 中配置客户端连接信息:


```
struct mqtt_connect_client_info_t base_client_user_info;
```

在文件\lwip\lwip-2.1.2\src\apps\mqtt\mqtt_ssl_config.c 中配置 CA 和客户端证书信息

```
root_CA
```

2、初始化基本参数

在 mqtt_cmd.c 创建 task 任务时调用 mqtt_base_param_cfg 进行基本参数的初始化。

3、根据应用层需要配置相应的消息发布/订阅/收到订阅消息的处理函数

```
mqtt_pub_cb
```

```
mqtt_sub_cb
```

```
mqtt_unsub_cb
```

```
mqtt_receive_msg_print
```

```
mqtt_receive_pub_msg_print
```

```
mqtt_connect_callback
```

注：当前这些 cb 里面仅仅做了一些简单打印，用户可根据需要对数据进行相关的处理。

4.4.2. 连线服务器

当启动 MQTT 客户端时，APP 首先创建名为 mqtt task 的任务，准备好参数后，使用 mqtt5_client_connect();即用 MQTT5.0 版本尝试对代理服务器的连接请求，如果连线失败，并且代理服务器返回错误码为版本不支持时，客户端会切换到 MQTT3.1.1，然后再次尝试连接代理服务器。代码如下：

表 4-1 版本不支持切换版本示例

```
if((mqtt_mode_type_get()==MODE_TYPE_MQTT5)&&(connect_fail_reason==MQTT_CONNECTION_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();
};
```

如果没有当前待连接的远端服务器的 IP 地址，APP 也支持输入网址，程序会通过调用 mqtt_ip_prase(ip_addr_t *addr_ip, char *domain) 进行解析，远端服务器的端口默认为 1883，即在 mqtt_cmd.h 文件中定义：#define MQTT_DEFAULT_PORT 1883

如果没有手动输入，系统会采用 MQTT 默认端口号 1883。服务器根据需要可能会变更端口号，比如百度智能云 MQTT 服务器不带 SSL 加密的端口号为默认 1883，但是如果带 SSL 加密的则会变更为 1884。

当前 APP 是采用可阻塞式的处理方法，客户端所有的发布和订阅消息都使用各自的队列来进

行管理。当 APP 调用连线命令时，APP 会先创建“MQTT task”，然后配置好相关参数，并将 APP 设置为 MODE_TYPE_MQTT5 模式，再通过调用 mqtt5_client_connect 启动连接流程。

4.4.3. 发布消息

当客户端准备发布消息时，该消息先会被保存到 cmd_msg_pub_list 队列中，等待 mqtt task 将其取出处理：

```
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));
```

取出后使用 mqtt5_msg_publish()/mqtt_msg_publish()进行发布。发布需要使用的参数可以通过命令输入或者提前配置成固定值，也可以通过新增加接口另行修改。当前发布的消息默认为不保留，即 retain 为 0，如果需要消息保留，可手动输入该标志位为 1。

4.4.4. 订阅/取消订阅消息

当客户端准备订阅/取消订阅消息时，该消息先会被保存到 cmd_msg_sub_list 队列中，等待 mqtt task 将其取出处理：

```
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));
```

取出后使用 mqtt5_msg_subscribe()/mqtt5_msg_unsub()/mqtt_sub_unsub()进行订阅/取消订阅。订阅/取消订阅需要使用的参数可以通过命令输入或者提前配置成固定值，也可以通过新增加接口另行修改。当前订阅的消息默认为不保留，即 retain 为 0，如果需要消息保留，可手动输入该标志位为 1。

4.4.5. 自动重连

在一些通信条件不理想的情况下，客户端和代理服务器会断开连接，可以通过该开关配置在非主动断开连接的情况下断线后启动重连。APP 中通过配置 auto_reconnect 的值来实现，mqtt task 任务每次循环时会判断该值来确定是否需要启动重新连接，代码如下：

表 4-2 自动重连示例

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

4.4.6. 断开连接

当客户端主动启动断开连接时，demo 会将 `mqtt_client->run` 配置为 `false`，然后在 MQTT task 中进行处理，即先结束循环，然后根据需要，通过 `mqtt5_disconnect/mqtt_disconnect` 先向代理服务器发送断开连接请求，然后开始释放相关资源，并删除 MQTT task。

5. 版本历史

[表 5-1. 版本历史](#)为本文档的版本更新历史。

表 5-1. 版本历史

版本号	说明	日期
1.0	初稿发布	2024 年 01 月 12 日

Important Notice

This document is the property of GigaDevice Semiconductor Inc. and its subsidiaries (the "Company"). This document, including any product of the Company described in this document (the "Product"), is owned by the Company under the intellectual property laws and treaties of the People's Republic of China and other jurisdictions worldwide. The Company reserves all rights under such laws and treaties and does not grant any license under its patents, copyrights, trademarks, or other intellectual property rights. The names and brands of third party referred thereto (if any) are the property of their respective owner and referred to for identification purposes only.

The Company makes no warranty of any kind, express or implied, with regard to this document or any Product, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The Company does not assume any liability arising out of the application or use of any Product described in this document. Any information provided in this document is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Except for customized products which has been expressly identified in the applicable agreement, the Products are designed, developed, and/or manufactured for ordinary business, industrial, personal, and/or household applications only. The Products are not designed, intended, or authorized for use as components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, atomic energy control instruments, combustion control instruments, airplane or spaceship instruments, transportation instruments, traffic signal instruments, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or Product could cause personal injury, death, property or environmental damage ("Unintended Uses"). Customers shall take any and all actions to ensure using and selling the Products in accordance with the applicable laws and regulations. The Company is not liable, in whole or in part, and customers shall and hereby do release the Company as well as its suppliers and/or distributors from any claim, damage, or other liability arising from or related to all Unintended Uses of the Products. Customers shall indemnify and hold the Company as well as its suppliers and/or distributors harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of the Products.

Information in this document is provided solely in connection with the Products. The Company reserves the right to make changes, corrections, modifications or improvements to this document and Products and services described herein at any time, without notice.