TLS and DTLS User Guide

Version 1.10

For use with TLS/DTLS module versions 3.15 and above

Date: 22-Mar-2016 14:00

Table of Contents

System Overview 5
Introduction 5
Feature Check 6
System Integration 7
Packages and Documents 8
Packages 8
Documents 8
Change History 9
Source File List 10
API Header File 10
Configuration Files 10
Public Certificates 10
Version File 11
System Files 11
Configuration Options 13
Specifying TLS or DTLS and the Native or Sockets Interface 13
config_tls.h 13
TLS Options 13
DTLS Options 17
config_tls.c 18
About TLS and DTLS 19
Client and Server Roles 19
TLS 19
Clients 19
Servers 19
DTLS 20
Clients 20
Servers 20
The TLS Session 21
TLS Connection Establishment (Handshaking) 21
TLS Data Exchange 23
The DTLS Session 25
DTLS Connection Establishment (Handshaking) 25
DTLS Data Exchange 27
The Embedded Encryption Manager (EEM) 29
Algorithms 29
Hash Algorithms 29
Signature Algorithms 29
Bulk Encryption Algorithms 29
Certificates 30
Cipher Suites 30
Application Programming Interface 31

Module Management 31
tls_init 31
tls_register_bulk 32
tls_register_hash 33
tls_register_sign 34
tls_start 35
tls_stop 36
tls_delete 37
TLS Native TCP Interface 38
tls_start_tcp 38
tls_close_tcp 39
tls_client_handshake_tcp 40
tls_server_handshake_tcp 41
tls_tcp_accept 42
tls_tcp_connect 43
tls_send_tcp 44
tls_receive_tcp 45
tls_get_buffer_tcp 46
tls_get_ticket_tcp 47
tls_get_state_tcp 48
DTLS Native UDP Interface 49
dtls_start_udp 50
dtls_close_udp 51
dtls_udp_srv_open 52
dtls_udp_srv_close 53
dtls_get_srv_conn_udp 54
dtls_accept_udp 55
dtls_connect_udp 56
dtls_send_udp 57
dtls_receive_udp 58
dtls_get_buffer_udp 59
dtls_get_ticket_udp 60
dtls_get_state_udp 61
TLS Sockets Interface 62
tls_client_handshake_socket 63
tls_server_handshake_socket 64
tls_send_socket 65
tls_receive_socket 66
tls_get_ticket_socket 67
tls_get_state_socket 68
tls_close_socket 69
DTLS Sockets Interface 70
dtls_client_handshake_socket 71
dtls_server_handshake_socket 73
dtls_send_socket 74
dtls_receive_socket 75

dtls_get_ticket_socket 76
dtls_get_state_socket 77
dtls_close_socket 78
Error Codes 79
Types and Definitions 80
t_tls_connection 80
t_tls_session 83
t_tls_security_params 84
t_tls_hash_alg_attr 84
t_tls_certificate 85
Notification Codes 85
t_tls_conn_inf 86
t_dtls_conn_data 87
Integration 89
OS Abstraction Layer (OAL) 89
PSP Porting 90
Sample Code 92
Server Application 92
TLS Server Interface using Native TCP 92
DTLS Server Interface using UDP 94
TLS Server Interface using Sockets 97
DTLS Server Interface using Sockets 99
Client Application 101
TLS Client Interface using Native TCP 101
DTLS Client Interface using UDP 103
TLS Client Interface using Sockets 106
DTLS Client Interface using Sockets 107

1 System Overview

1.1 Introduction

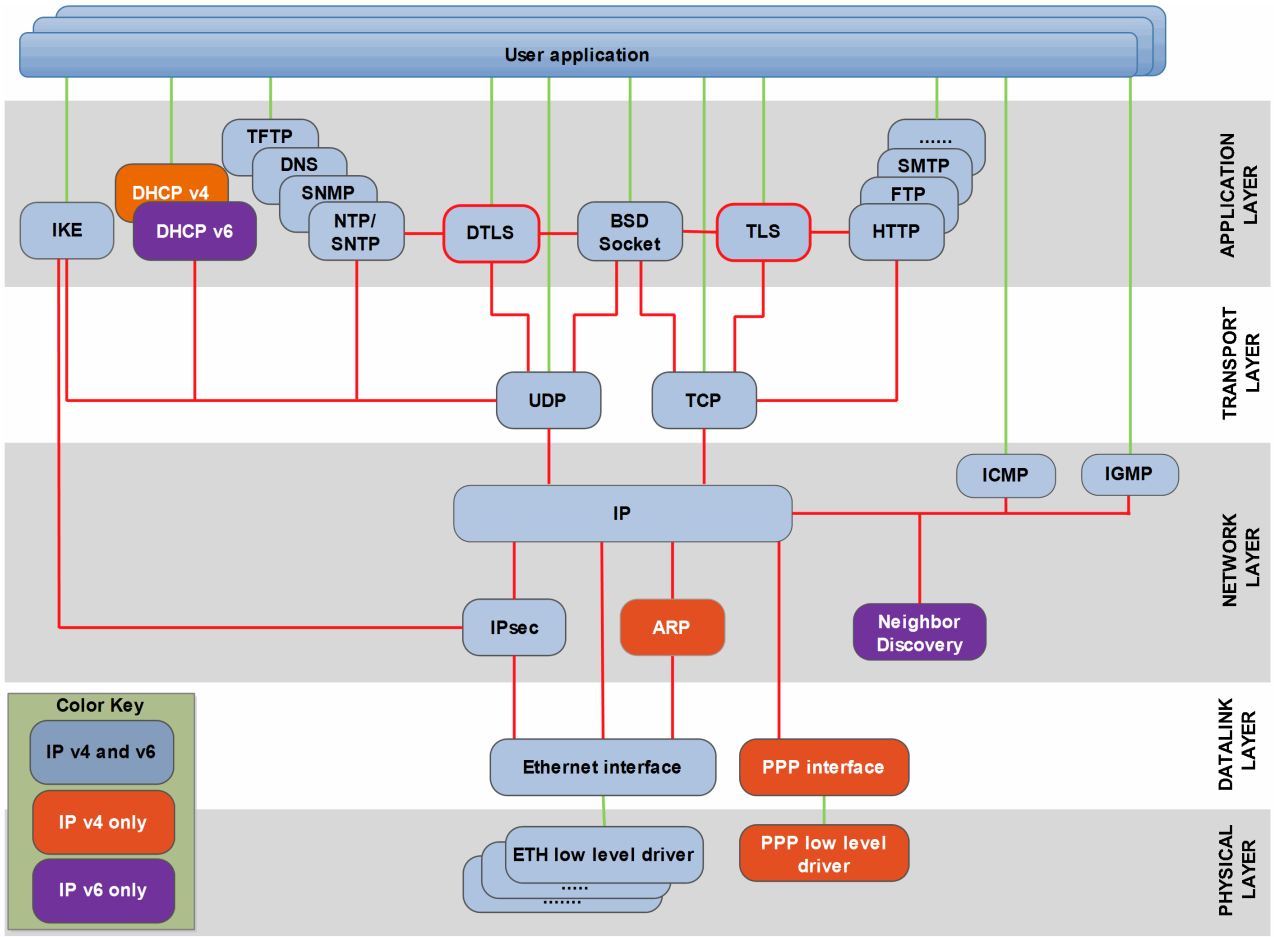
This guide is for those who want to implement HCC Embedded's verifiable Transport Layer Security (TLS) or Datagram Transport Layer Security (DTLS) as a framework for secure communication in computer networks based on the TCP/IP or UDP protocols. The module supports Secure Sockets Layer (SSL) 3.0 but this is deprecated as TLS 1.2 is the currently recommended standard.

Applications that use this module use its API to communicate with remote systems reliably.

This module provides two options:

- TLS interfacing to either HCC's MISRA-compliant TCP or to a TCP Sockets interface.
- DTLS interfacing to either HCC's MISRA-compliant UDP or to a UDP Sockets interface.

The TLS and DTLS module forms part of the HCC MISRA-compliant TCP/IP stack, as shown below, and is designed specifically for use with it. (In this diagram green lines show interfaces available to users of the stack, red lines show interfaces internal to the TCP/IP system.)



The TLS/DTLS implementation can be used as client or server (host). The module provides the following guaranteed capabilities, regardless of the components that lie beneath it:

- Privacy – it ensures that nobody else can read the message.
- Authenticity – it ensures that each party really is talking to the peer they think they are talking to.
- Integrity – it ensures that the data payload has not been modified/tampered with.

Note: You may not require all three of the above capabilities for all use cases; HCC can advise on this.

The module uses HCC's Embedded Encryption Manager (EEM) to provide encryption and certificate management.

Note: Although every attempt has been made to simplify the system's use, in order to obtain the maximum practical benefits you must understand the requirements of the systems you design. HCC Embedded offers hardware and firmware development consultancy to help you implement your system.

1.2 Feature Check

The main features of the system are the following:

- It conforms to the HCC Advanced Embedded Framework.
- It can be used with or without an RTOS.
- It is MISRA-compliant. A full MISRA compliance report is provided and, for specialized applications, a full UML description is available that can be licensed as a separate component.
- It is designed for microcontrollers, ensuring a low memory footprint. This is typically around 20KB of ROM or 8KB of RAM.
- It typically uses a standard Sockets interface, allowing easy integration with many embedded applications.
- It supports TLS 1.0, 1.1 and 1.2 (RFC 5246) and SSL 3.0 and is verifiable.
- It supports DTLS version 1.2 (RFC 6347) and version 1.0 (RFC 4347).
- It supports heartbeat extensions (RFC 6520).
- It supports HTTP over TLS (RFC 2818).
- It provides HTTP or FTP Server support for HTTPS and FTPS implementations, or for connection to any other secure client or server application.
- It uses HCC's Embedded Encryption Manager (EEM) to provide full certificate management.
- It supports all the algorithms supported by the EEM, including AES, 3DES, DSS, EDH, MD5, RSA, SHA-1, SHA-256, SHA-384, and SHA-512. These acronyms are expanded below.
- It supports all the mandatory cipher suites required by different versions of TLS.

The supported algorithms are the following:

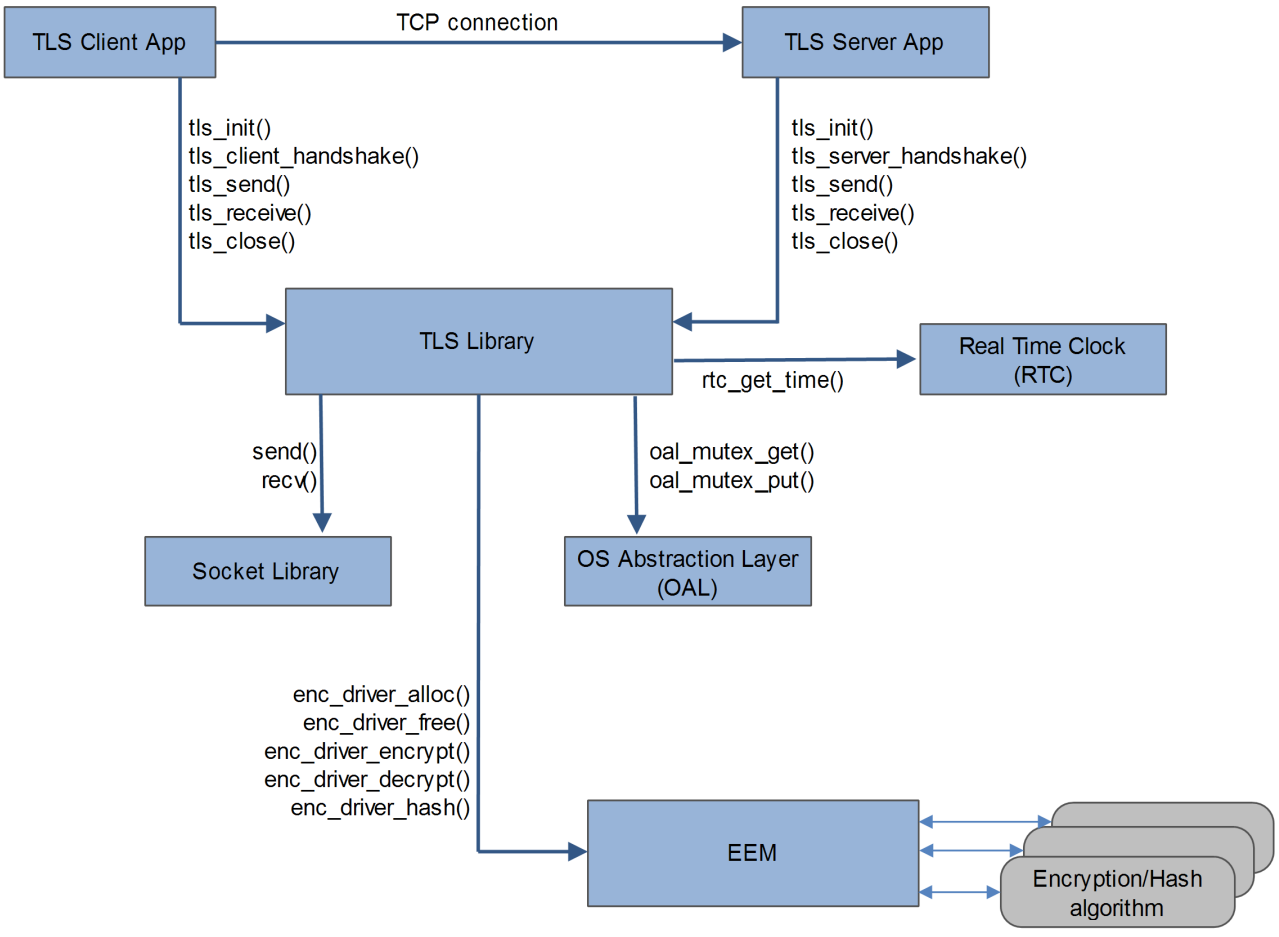
- Advanced Encryption Standard (AES).
- Digital Signature Standard (DSS).
- Ephemeral Diffie-Hellman (EDH) algorithm.
- Message Digest Algorithm 5 (MD5).
- RSA Signature Algorithm (RSA).
- Secure Hash Algorithm (SHA-1, SHA-256, SHA-384, and SHA-512).
- Triple Data Encryption Standard (3DES).

1.3 System Integration

The collaboration diagram below shows:

- All the components and their interactions. (This example shows TLS not DTLS.)
- The client/server relationship, in this case initiated by the client. The TCP connection between these has already been established when the client application calls **tls_init()**, the first call in the sequence.

Note: To improve clarity, the module management functions are omitted.



1.4 Packages and Documents

Packages

The table below lists the packages that you need in order to use this module.

Package	Description
hcc_base_docs	This contains the two guides that will help you get started.
mip_base	The IP base package.
ip_base_v4, ip_base_v6	The base packages for IPv4 and IPv6, respectively.
mip_tcp	The TCP package.
mip_udp	The UDP package.
ip_socket	The BSD Sockets interface.
ip_nos_misra	The TLS/DTLS package described in this document.
enc_base	The Embedded Encryption Manager (EEM) package.
oal_base	The OS Abstraction Layer (OAL) base package.

Documents

For an overview of the HCC TLS/DTLS software, see Product Information on the main HCC website.

Readers should note the points in the HCC Documentation Guidelines on the HCC documentation website.

HCC Firmware Quick Start Guide

This document describes how to install packages provided by HCC in the target development environment. Also follow the *Quick Start Guide* when HCC provides package updates.

HCC Source Tree Guide

This document describes the HCC source tree. It gives an overview of the system to make clear the logic behind its organization.

HCC TCP/IP IPv4 Stack System User Guide, HCC TCP/IP Dual Stack System User Guide

These are the core documents that describe the complete TCP/IP system. The second manual covers both IPv4 and IPv6 systems.

HCC TLS and DTLS User Guide

This is this document.

HCC Embedded Encryption Manager User Guide

This document describes the EEM which handles all aspects of encryption for TLS/DTLS.

1.5 Change History

This section includes recent changes to this product. For a list of all the changes, refer to the file **src/history/ip/ip_tls.txt** in the distribution package.

Version	Changes
3.15	Updated to work with IP protocol version 6. Removed restriction on using Cipher-suite RSA_AESCBC_SHA256. Fixed problem that caused system to run out of IP buffers in the case of a DTLS native connection. Corrected resetting of retransmision timeout in the case of the last DTLS flight. When a DTLS client helo message (that contains a cookie) is sent, a random number is not regenerated.
3.14	Updated module to work with IP stack version 6.02.
3.13	Updated module to work with IP base major version 6.
3.12	Added support for multiple handshake messages in RecordLayer. Fixed handling of multiple RecordLayer blocks in a single PDU. Cleaned up configuration template by setting values to default. Added the following authorisation certificates: • Geotrust Global CA (expiry date: 2018-08-21 04:00:00 (UTC)) • GlobalSign Organization Validation CA - G2 (expiry date: 2022-04-13 10:00:00 (UTC)) • VeriSign Class 3 Public Primary Certification (expiry date: 2021-11-07 23:59:59 (UTC))

2 Source File List

This section describes all the source code files included in the system. These files follow the HCC Embedded standard source tree system, described in the *HCC Source Tree Guide*. All references to file pathnames refer to locations within this standard source tree, not within the package you initially receive.

> **Note:** Do not modify any files except the configuration files.

2.1 API Header File

The file **src/api/api_tls.h** should be included by any application using the system. This is the only file that should be included by an application using this module. For details of the functions, see Application Programming Interface.

2.2 Configuration Files

The following files in the directory **src/config** contain the system configuration options. Configure these as required.

File	Description
config_tls.h	Contains the TLS/DTLS parameters. For details, see config_tls.h.
config_tls.c	Contains algorithm options and array definitions. For details, see config_tls.c.

2.3 Public Certificates

The following files in the directory **src/config/authcertificates** are the public certificates of trusted Certification Authorities (CAs). These are available for you to include in your project when you think that your application will communicate with devices that have signed certificates from these CAs.

File	Description
ca_geotrustglobal.h and **.c**	Contains the Geotrust Global CA certificate.
ca_globalsigng2.h and **.c**	Contains the GlobalSign Organization Validation CA - G2 certificate.
ca_verisignclass3.h and **.c**	Contains the VeriSign Class 3 Public Primary Certification certificate.

2.4 Version File

The file **src/version/ver_tls.h** contains the version number of this module. This version number is checked by all modules that use this module to ensure system consistency over upgrades.

2.5 System Files

These files are in the directory **src/ip/stack/tls**. **These files should only be modified by HCC**.

The files common to DTLS and TLS are the following:

File	Description
tls_certificate.c and **.h**	Source code and header file for certificate handling.
dtls_common.c and **.h**	Source code and header file for DTLS common elements.
tls_common.c and **.h**	Source code and header file for TLS common elements.

The DTLS files are the following:

File	Description
dtls_client.c and **.h**	Source code and header file for the DTLS client.
dtls_client_socket.c	Source code for DTLS client socket components.
dtls_client_tcp.c	Source code for client TCP components.
dtls_client_udp.c	Source code for client UDP components.
tls_ext.c and **.h**	Source code and header file for the heartbeat extension.
dtls_server.c and **.h**	Source code and header file for the DTLS server.
dtls_server_socket.c	Source code and header file for DTLS server socket components.
dtls_server_tcp.c	Source code and header file for server TCP components.
dtls_server_udp.c	Source code and header file for server UDP components.
dtls_socket.c	Source code and header file for socket components.
dtls_tcp.c and **.h**	Source code and header file for DTLS TCP components.
dtls_udp.c and **.h**	Source code and header file for DTLS UDP components.

The TLS files are the following:

File	Description
tls_client.c and **.h**	Source code and header file for the TLS client.
tls_client_socket.c	Source code for TLS client socket components.
tls_client_tcp.c	Source code for client TCP components.
tls_ext.c and **.h**	Source code and header file for the heartbeat extension.
tls_server.c and **.h**	Source code and header file for the TLS server.
tls_server_socket.c	Source code and header file for TLS server socket components.
tls_server_tcp.c	Source code and header file for server TCP components.
tls_socket.c and **.h**	Source code and header file for socket components.
tls_tcp.c and **.h**	Source code and header file for TLS TCP components.

3 Configuration Options

Set the configuration options in the files listed below. These are in the directory **src/config**.

3.1 Specifying TLS or DTLS and the Native or Sockets Interface

Use the various ENABLE options in the file **config_tls.h** as follows to specify the combination of TLS or DTLS and native TCP/UDP or Sockets to run.

Configuration wanted	Set these options to 1
TLS and Native TCP	TLS_TCPIP_IFC_ENABLE
TLS and Sockets	TLS_SOCKET_IFC_ENABLE
DTLS and Native UDP	DTLS_IFC_ENABLE, DTLS_UDP_IFC_ENABLE
DTLS and Sockets	DTLS_IFC_ENABLE, DTLS_SOCKET_IFC_ENABLE

After enabling the option(s) you want, set the other ENABLE options to zero.

3.2 config_tls.h

Set the following system configuration options in the file **src/config/config_tls.h**.

TLS Options

TLS_MIN_KEY_EX_SIZE

The minimum size of key exchange message sent by server to client. The default value is 128.

TLS_MAX_MKEY_LEN

The maximum length of a Message Authentication Code (MAC) key. This must be equal to/greater than the largest MAC key configured in the cipher suite. The default value is 20.

TLS_MAX_BKEY_LEN

The maximum length of a bulk key. This must be equal to/greater than the largest bulk key configured in the cipher suite. The default value is 24.

TLS_MAX_HMAC_LEN

The maximum length of a Hashed MAC (HMAC) output. The default value is 64.

TLS_MIN_PUB_KEY_SIZE

The minimum size of public key used to encrypt a message with the master key. The default value is 32.

TLS_MAX_PUB_KEY_SIZE

The maximum size of public key used to encrypt a message with the master key. The default value is 260.

TLS_PRE_MASTER_SECRET_SIZE

The size of the buffer holding the pre-master secret. This must not be shorter than TLS_MASTER_SEC_LEN (48). The default value is 64.

TLS_MAX_HOST_CERT

The maximum number of host certificates. The default value is 1.

TLS_MAX_CA_CERT

The maximum number of Certificate Authority (CA) certificates. The default value is 1.

TLS_MAX_RV_CERT

The maximum number of revoked certificates. The default value is 1.

TLS_MAX_CIPHER_SUITES

The maximum number of supported cipher suites. The default value is 1.

TLS_MAX_HASH_ALG_SIZE

The maximum number of hash algorithms. The default value is 3.

TLS_MAX_BULK_ALG_SIZE

The maximum number of bulk encryption algorithms. The default value is 2.

TLS_MAX_SIGN_ALG_SIZE

The maximum number of signature (public key) algorithms. The default value is 3.

TLS_MAX_CONN_SIZE

The maximum number of supported connections. The default value is 1.

TLS_MAX_SES_SIZE

The maximum number of supported sessions. The default value is 1.

TLS_PEER_NAME_SIZE

The maximum size of the peer name field. The default value is 16.

TLS_CBC_PADDING_LEN

The maximum length of the padding added by bulk encryption algorithm. The default value is 9.

TLS_MAX_DATA_LEN

The maximum size of the TLS user data. The default value is 2700.

TLS_RECV_BUF_SIZE

The size of the buffer for received data. The default value is 2700.

The size depends on the bulk encryption algorithm for AES. The recommended value is:

- TLS_RP_HDR_LEN + cbc_size * (2 + (TLS_MAX_DATA_LEN + TLS_MAX_HMAC_LEN) / cbc_size)

where *cbc_size* is the length of the cipher block (for example, 128 or 256 for AES).

TLS_DATA_BUF_SIZE

The size of the buffer used for encryption. This should not be less than 2 * TLS_RECV_BUF_SIZE. The default value is 3400.

TLS_HS_MSG_STORE_SIZE

The size of the buffer used for all handshake messages. The default value is (8*1024).

TLS_DIGEST_PREFIX_LEN

The length of the digest prefix used for signature generation. The default value is 32.

TLS_OID_SIZE

The size of the Object Identifier (OID). The default value is 9.

Note: TLS uses TCP functions to establish the connection with the peer and to send/receive the data. Depending of the setting of the following two parameters, it uses functions directly from the TCP module or the socket interface.

TLS_SOCKET_IFC_ENABLE

Set this to 1 for a Socket interface. The default value is 1.

TLS_TCPIP_IFC_ENABLE

Set this to 1 for a native TCP/IP interface. The default value is 1.

TLS_CHECK_CERT_EXP

Set this to 1 if you want certificate validity to be verified. The default value is 0.

TLS_TCP_CONN_STACK_SIZE

The stack size for TCPIP connections. The default value is 1256.

TLS_TIMER_PERIOD

The timer period for TLS TCP connections in ms. The default value is 100.

TLS_TCP_HS_TIMEOUT

The timeout for handshake operation in ms. The default value is 100.

TLS_HB_TIME

The heartbeat resend time in seconds. The default value is 3.

TLS_HB_ALLOW_PEER_REQ

Set this to 1 to allow the peer to send heartbeat requests. The default value is 0.

TLS_EXT_HB_PAYLOAD_SIZE

The size of the payload section in a heartbeat extension. The default value is 8.

TLS_TREAT_RSA512_CERT_INV

Keep this at the default of 1 to reject certificates with an RSA 512 signature. Otherwise set it to 0.

TLS_TRUST_ALL_CERT

Set this to 1 if all certificates are to be trusted. The default value is 0.

DTLS Options

DTLS_IFC_ENABLE

Set this to 1 to enable the DTLS interface. The default value is 0.

DTLS_UDP_IFC_ENABLE

Set this to 1 to enable the native UDP interface for DTLS. The default value is 1.

DTLS_SOCKET_IFC_ENABLE

Set this to 1 to enable the Socket interface for DTLS. The default value is 1.

DTLS_CLIENT_CONN_PORT

The DTLS client connection port start number. The default value is 1000.

DTLS_PMTU_SIZE

The MTU size used for packet fragmenting in DTLS mode. The default value is 1400.

DTLS_SCKSRV_RCV_SIZE

The size of buffer used by DTLS server (Socket version) for receiving data. The default value is 500.

DTLS_MSG_RETRY_NR

The number of message retransmissions after which a DTLS_NTF_TIMEOUT notify is sent. The default value is 5.

DTLS_UDP_MAX_PORT_SRV_CNT

The maximum number of server ports for the DTLS UDP native interface. The default value is 1.

DTLS_RETRANSMIT_INIT_TIME

The DTLS handshake retransmit time initial value in units of 100 ms. The default value is 10.

DTLS_RETRANSMIT_MAX_TIME

The DTLS handshake retransmit time maximum value in units of 100 ms. The default value is 600.

3.3 config_tls.c

The file **src/config/config_tls.c** contains the algorithm options listed below.

RSA_KEY_MODULO_LEN

The length of an RSA modulo key. The default value is 128.

RSA_KEY_EXP_LEN

The length of an RSA modulo key. The default value is 3.

4 About TLS and DTLS

This section covers:

- Client and server roles.
- The stages in a session. These are described separately for TLS and DTLS.
- Encryption – the algorithms, certificates, and cipher suites used.

4.1 Client and Server Roles

TLS

TLS uses TCP functions to establish the connection with the peer and to send/receive the data. Depending on the ENABLE options selected, it uses functions either from the TCP module or the Sockets interface.

Clients

The first step for the TLS client implementation is to establish the TCP connection with the server. Next the client side TLS handshake must be completed.

- If the device running the TLS and TCP modules is configured to use a preemptive environment, the call to the handshake procedure is blocking and returns either success (TLS_OK) or an error code.
- If the device is not running a preemptive OS, the handshake procedure is polled and returns TLS_WAIT status until handshaking is completed (indicated by the return value TLS_OK or an error code).

Servers

In order to implement TLS server, a port must be opened on which the application accepts TCP connections from clients. Once the TCP connection with the client is established, the server side TLS handshake must be completed.

- If the device running the TLS and TCP modules is configured to use a preemptive environment, the call to the handshake procedure is blocking and returns either success (TLS_OK) or an error code.
- If the device is not running a preemptive environment, the handshake procedure is polled and returns TLS_WAIT status until the handshake is completed (indicated by the return code TLS_OK or an error code).

Requests coming in from clients are read in a loop using **tls_receive()** and handled appropriately.

DTLS

DTLS uses UDP functions to establish the connection with the peer and to send/receive the data. Depending on the ENABLE options selected, it uses functions either from the UDP module or the Sockets interface.

Clients

The first step for the DTLS client implementation is to establish the UDP connection with the server. Next the client side DTLS handshake must be completed.

- If the device running the DTLS and UDP modules is configured to use a preemptive environment, the call to the handshake procedure is blocking and returns either success (TLS_OK) or an error code.
- If the device is not running a preemptive OS, the handshake procedure is polled and returns TLS_WAIT status until handshaking is completed (indicated by the return value TLS_OK or an error code).

Servers

In order to implement DTLS server, a port must be opened on which the application accepts UDP connections from clients. Once the UDP connection with the client is established, the server side DTLS handshake must be completed.

- If the device running the DTLS and UDP modules is configured to use a preemptive environment, the call to the handshake procedure is blocking and returns either success (TLS_OK) or an error code.
- If the device is not running a preemptive environment, the handshake procedure is polled and returns TLS_WAIT status until the handshake is completed (indicated by the return code TLS_OK or an error code).

Requests coming in from clients are read in a loop using **dtls_receive()** and handled appropriately.

4.2 The TLS Session

Secure communication as defined by TLS comprises two stages:

- Connection establishment (handshaking).
- Encrypted data exchange.

These are described in the following sections.

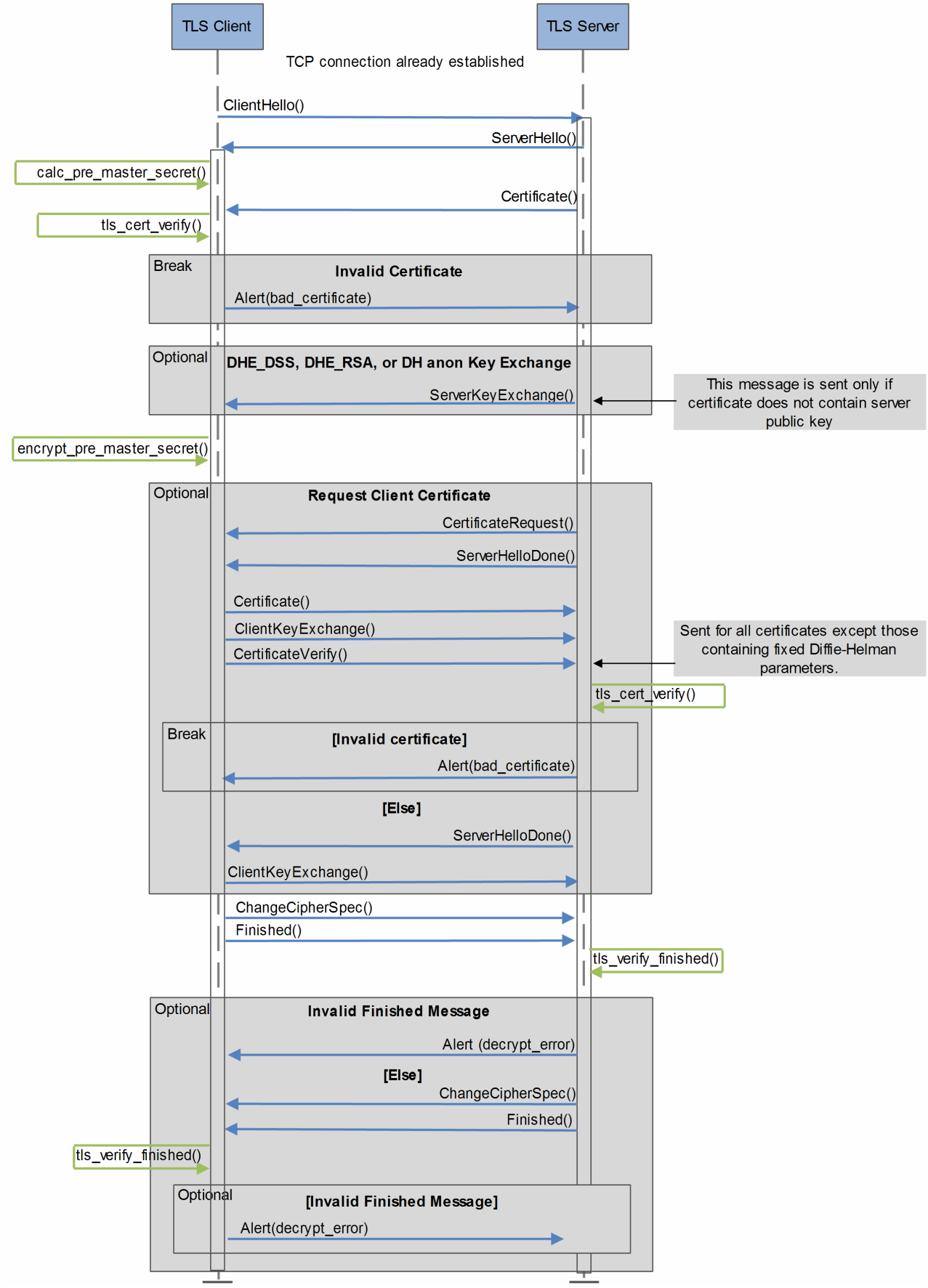
TLS Connection Establishment (Handshaking)

Before data exchange can begin, handshaking must complete successfully. A TCP connection between client and server must already be established when handshaking starts.

The diagram below shows the handshaking process. The main steps in this are:

1. The TLS client and server exchange hellos. The version of the protocol which is to be used is decided at this stage.
2. The TLS client calculates the pre-master secret.
3. The server sends its certificate to the client, which verifies it. If the certificate is invalid, the client sends an alert to the server.
4. Optionally, and only if the certificate does not contain a server public key, keys are exchanged.
5. The TLS client encrypts the pre-master secret.
6. The server requests the client's certificate from the client, which sends it. The server verifies it and, if the certificate is invalid, sends an alert to the client. Otherwise, it sends a **ServerHelloDone()** and the client responds with **ClientKeyExchange()**.
7. The client calls **ChangeCipherSpec()** and sends a Finished message.
8. The server verifies that the Finished message is from the client. If it is invalid, it sends an alert to the client, otherwise it calls **ChangeCipherSpec()** and sends a Finished message. If the client verifies this successfully, the handshake has completed without error.

Application messages sent between client and server from this point onwards are authenticated (and also encrypted, if this applies).


TLS Client
TLS Server
TCP connection already established
ClientHello()
ServerHello()
calc_pre_master_secret()
Certificate()
tls_cert_verify()
Break
Invalid Certificate
Alert(bad_certificate)
Optional
DHE_DSS, DHE_RSA, or DH anon Key Exchange
ServerKeyExchange()
This message is sent only if certificate does not contain server public key
encrypt_pre_master_secret()
Optional
Request Client Certificate
CertificateRequest()
ServerHelloDone()
Certificate()
ClientKeyExchange()
CertificateVerify()
Sent for all certificates except those containing fixed Diffie-Helman parameters.
tls_cert_verify()
Break
[Invalid certificate]
Alert(bad_certificate)
[Else]
ServerHelloDone()
ClientKeyExchange()
ChangeCipherSpec()
Finished()
tls_verify_finished()
Optional
Invalid Finished Message
Alert (decrypt_error)
[Else]
ChangeCipherSpec()
Finished()
tls_verify_finished()
Optional
[Invalid Finished Message]
Alert(decrypt_error)

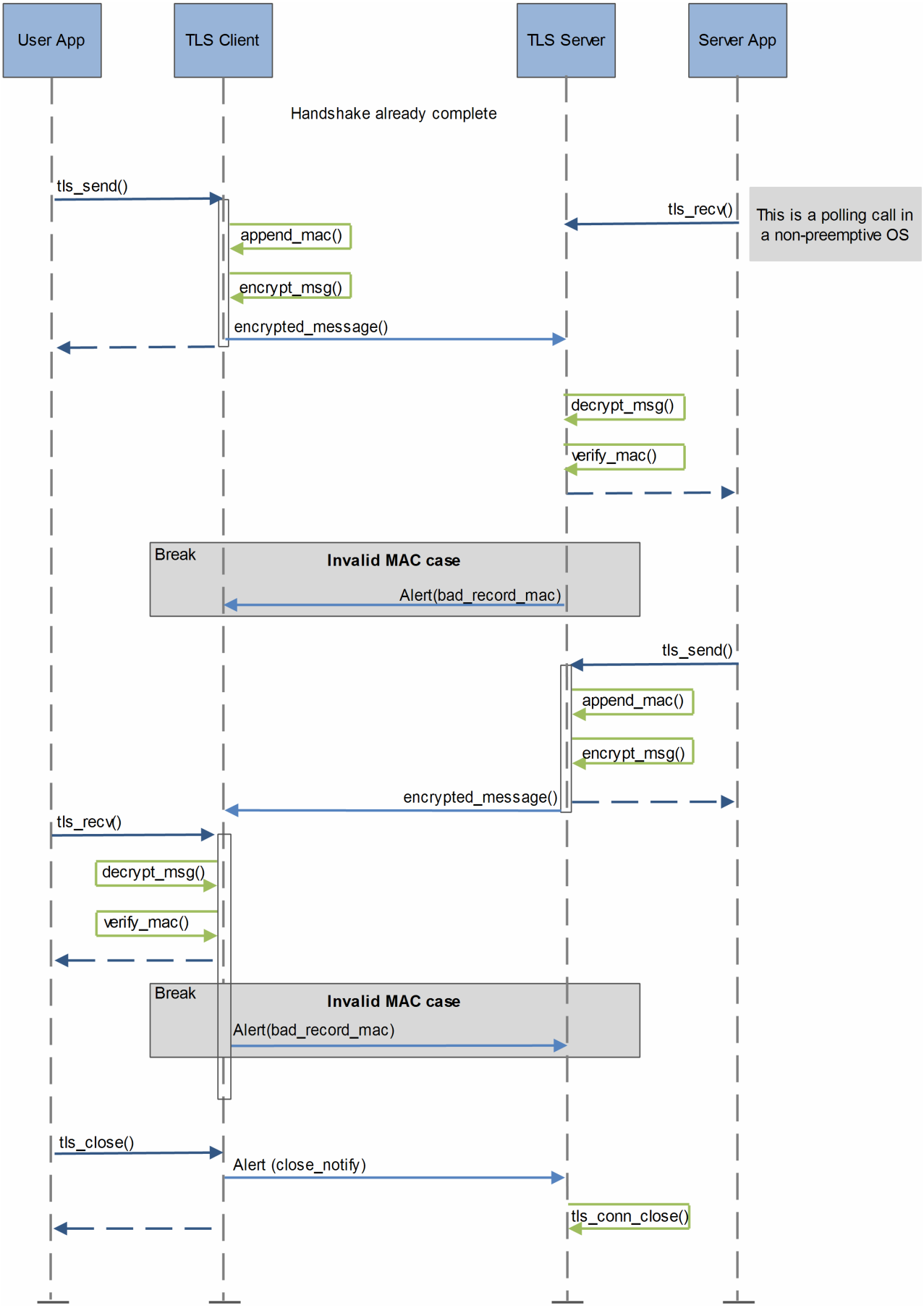
TLS Data Exchange

Once handshaking is complete, data exchange can begin.

> **Note:** The functions shown in the steps below as **tls_send()**, **tls_receive()** and **tls_close()** would actually be either **tls_xxx_tcp()** or **tls_xxx_socket()**.

The diagram below shows the process of data exchange. The steps are:

1. The user application uses **tls_send()** to send a message to the TLS client.
2. The TLS client appends the Message Authentication Code (MAC), encrypts the message, and sends it to the server.
3. The server decrypts the message and verifies the MAC. If the MAC is invalid, it sends an alert to the client. Otherwise, it passes the message to the server application.
4. The server application uses **tls_send()** to send a message to the server.
5. The TLS server appends the MAC, encrypts the message, and sends it to the client.
6. The client decrypts the message and verifies the MAC. If the MAC is invalid, the client sends an alert to the server. Otherwise, it passes the message to the client application, which has sent a **tls_receive()**.
7. After receiving the response, the user application wants to close the connection. It sends the client a **tls_close()**. The client sends a **close_notify** alert to the server, which in turn closes the connection at its end.


User App
TLS Client
TLS Server
Server App
Handshake already complete
tls_send()
tls_recv()
This is a polling call in a non-preemptive OS
append_mac()
encrypt_msg()
encrypted_message()
decrypt_msg()
verify_mac()
Break
Invalid MAC case
Alert(bad_record_mac)
tls_send()
append_mac()
encrypt_msg()
encrypted_message()
tls_recv()
decrypt_msg()
verify_mac()
Break
Invalid MAC case
Alert(bad_record_mac)
tls_close()
Alert (close_notify)
tls_conn_close()

4.3 The DTLS Session

Secure communication as defined by DTLS comprises two stages:

- Connection establishment (handshaking).
- Encrypted data exchange.

These are described in the following sections.

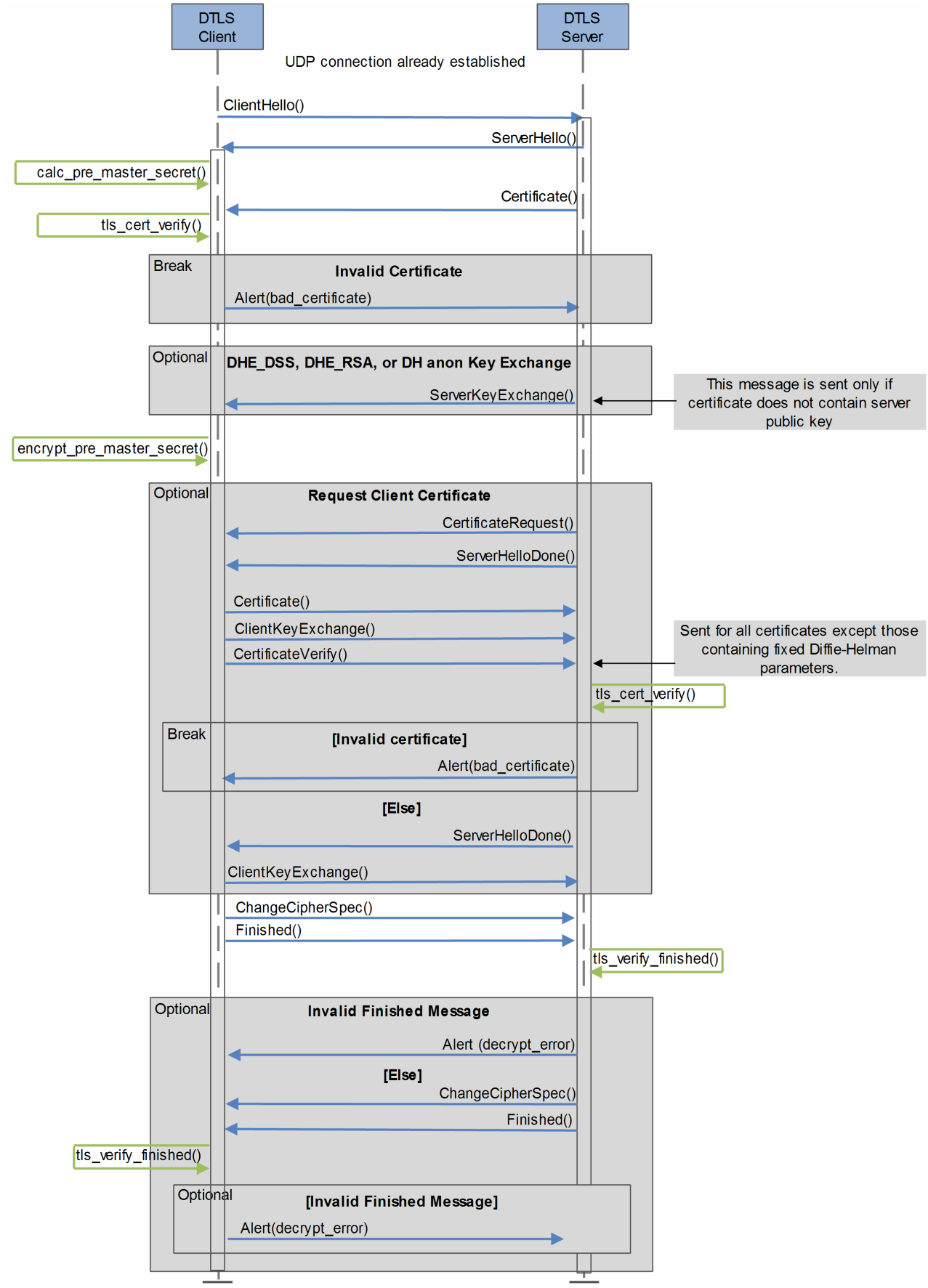
DTLS Connection Establishment (Handshaking)

Before data exchange can begin, handshaking must complete successfully. A UDP connection between client and server must already be established when handshaking starts.

The diagram below shows the handshaking process. The main steps in this are:

1. The DTLS client and server exchange hellos. The version of the protocol which is to be used is decided at this stage.
2. The DTLS client calculates the pre-master secret.
3. The server sends its certificate to the client, which verifies it. If the certificate is invalid, the client sends an alert to the server.
4. Optionally, and only if the certificate does not contain a server public key, keys are exchanged.
5. The DTLS client encrypts the pre-master secret.
6. The server requests the client's certificate from the client, which sends it. The server verifies it and, if the certificate is invalid, sends an alert to the client. Otherwise, it sends a **ServerHelloDone()** and the client responds with **ClientKeyExchange()**.
7. The client calls **ChangeCipherSpec()** and sends a Finished message.
8. The server verifies that the Finished message is from the client. If it is invalid, it sends an alert to the client, otherwise it calls **ChangeCipherSpec()** and sends a Finished message. If the client verifies this successfully, the handshake has completed without error.

Application messages sent between client and server from this point onwards are authenticated (and also encrypted, if this applies).


DTLS Client
DTLS Server
UDP connection already established
ClientHello()
ServerHello()
calc_pre_master_secret()
Certificate()
tls_cert_verify()
Break
Invalid Certificate
Alert(bad_certificate)
Optional
DHE_DSS, DHE_RSA, or DH anon Key Exchange
ServerKeyExchange()
This message is sent only if certificate does not contain server public key
encrypt_pre_master_secret()
Optional
Request Client Certificate
CertificateRequest()
ServerHelloDone()
Certificate()
ClientKeyExchange()
CertificateVerify()
Sent for all certificates except those containing fixed Diffie-Helman parameters.
tls_cert_verify()
Break
[Invalid certificate]
Alert(bad_certificate)
[Else]
ServerHelloDone()
ClientKeyExchange()
ChangeCipherSpec()
Finished()
tls_verify_finished()
Optional
Invalid Finished Message
Alert (decrypt_error)
[Else]
ChangeCipherSpec()
Finished()
tls_verify_finished()
Optional
[Invalid Finished Message]
Alert(decrypt_error)

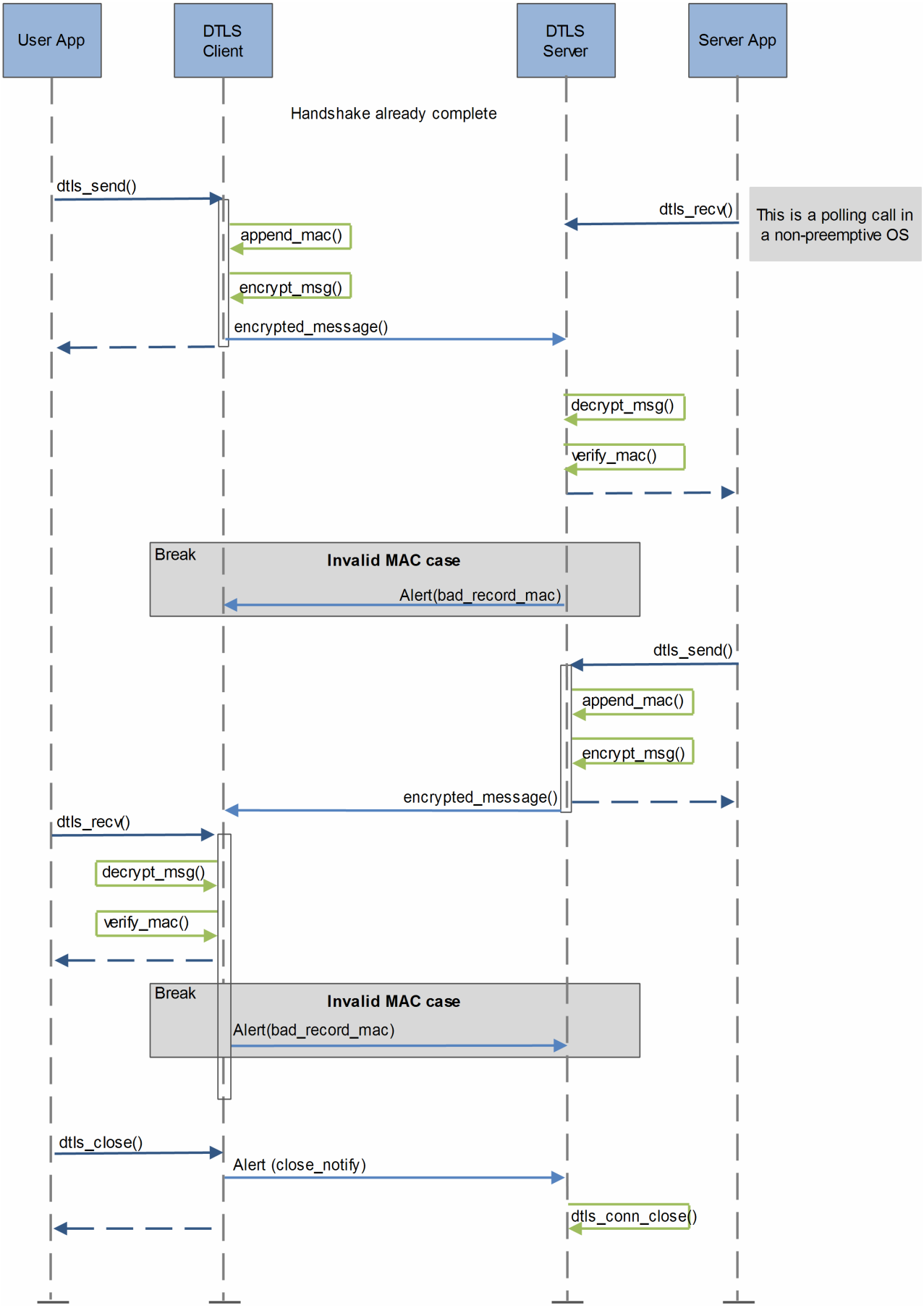
DTLS Data Exchange

Once handshaking is complete, data exchange can begin.

> **Note:** The functions shown in the steps below as **dtls_send()**, **dtls_receive()** and **dtls_close()** would actually be either **dtls_xxx_udp()** or **dtls_xxx_socket()**.

The diagram below shows the process of data exchange. The steps are:

1. The user application uses **dtls_send()** to send a message to the TLS client.
2. The TLS client appends the Message Authentication Code (MAC), encrypts the message, and sends it to the server.
3. The server decrypts the message and verifies the MAC. If the MAC is invalid, it sends an alert to the client. Otherwise, it passes the message to the server application.
4. The server application uses **dtls_send()** to send a message to the server.
5. The TLS server appends the MAC, encrypts the message, and sends it to the client.
6. The client decrypts the message and verifies the MAC. If the MAC is invalid, the client sends an alert to the server. Otherwise, it passes the message to the client application, which has sent a **dtls_receive()**.
7. After receiving the response, the user application wants to close the connection. It sends the client a **dtls_close()**. The client sends a **close_notify** alert to the server, which in turn closes the connection at its end.


User App
DTLS Client
DTLS Server
Server App
Handshake already complete
dtls_send()
dtls_recv()
This is a polling call in a non-preemptive OS
append_mac()
encrypt_msg()
encrypted_message()
decrypt_msg()
verify_mac()
Break
Invalid MAC case
Alert(bad_record_mac)
dtls_send()
append_mac()
encrypt_msg()
encrypted_message()
dtls_recv()
decrypt_msg()
verify_mac()
Break
Invalid MAC case
Alert(bad_record_mac)
dtls_close()
Alert (close_notify)
dtls_conn_close()

4.4 The Embedded Encryption Manager (EEM)

The module uses the HCC EEM which implements a standard interface to all the encryption algorithms. The process is:

1. Encryption algorithms are registered with the EEM, which returns the handle of the specific algorithm.
2. The encryption handle is passed to TLS/DTLS by the function **xxx_register_sign()**, **xxx_register_hash()** or **xxx_register_bulk()**.

The EEM also contains also a big number library which provides mathematical operations (for example, addition, subtraction, multiplication and modulo) on the big numbers (over 64 bits) that are used by some encryption algorithms.

For full details of the EEM operation and its API, refer to the *HCC Embedded Encryption Manager User's Guide*.

Algorithms

This section describes the different types of algorithm supported. These are provided by the EEM.

Hash Algorithms

Hash algorithms are used to calculate the keys of a secure session, or to generate the hash of all messages exchanged between client and server during connection establishment.

The hash algorithms supported by the EEM are MD5, SHA-1, SHA-256, SHA-384 and SHA-512.

Signature Algorithms

Signature algorithms are used during connection establishment to exchange session keys between client and server securely, and to verify peer certificates. Usually these algorithms are asymmetrical, based on public/private keys.

The signature algorithms supported by the EEM are RSA, EDH, and DSA.

Bulk Encryption Algorithms

Bulk encryption algorithms are used to encrypt data exchanged between client and server after a connection has been established. These algorithms use encryption keys which are negotiated between client and server during connection establishment.

The bulk encryption algorithms supported by the EEM are AES and 3DES (Triple DES).

Certificates

Three types of certificate are used: host certificates, Certification Authority (CA) certificates, and revoked certificates.

Each certificate is defined in the **config_tls.c** file by three objects:

- A byte array containing the DER (Distinguished Encoding Rules) encoded certificate (host, CA, or revoked).
- A byte array containing the certificate's private key.
- The structure *t_tls_certificate*, describing the certificate.

Arrays with references to the host, CA, and revoked certificates are initialized in the file **config_tls.c**. These arrays, *g_tls_arr_host_cert*, *g_tls_arr_ca_cert* and *g_tls_arr_rv_cert* respectively, are referenced by TLS /DTLS when it verifies the peer certificate or authenticates itself with the peer.

Use the provided public CA certificates in your project when you communicate with devices that have signed certificates from these CAs.

Cipher Suites

A cipher suite defines the combination of hash, signature, and bulk encryption algorithms that can be used by a single TLS/DTLS session.

As the module supports three hash algorithms, three signature algorithms, and four bulk encryption algorithms, theoretically 36 (3*3*4) cipher suites can be created. The file **src/api/api_tls.h** has the full list. The RFC for each version of TLS/DTLS specifies which cipher suites are mandatory and which are optional.

Examples of cipher suites supported by the implementation are shown below. The hex value is the ID of the cipher suite.

Hash	Signature	Bulk encryption	Defined in the RFCs (and api_tls.h) as:	ID
SHA	RSA	3DESEDECBC	TLS_RSA_WITH_3DES_EDE_CBC_SHA	0x000AU
SHA	RSA	AES128CBC	TLS_RSA_WITH_AES_128_CBC_SHA	0x002FU
SHA	RSA	3DESEDECBC	TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA	0x0013U

5 Application Programming Interface

This section describes the Application Programming Interface (API). It includes all the functions that are available to an application program.

5.1 Module Management

Note: You must call any **tls_register_xxx()** functions after **tls_init()** and before **tls_start()**. Otherwise the module returns an error.

tls_init

Use this function to initialize the module. This initializes arrays of connections and sessions, and creates *tls_mutex*.

Note:

- You must start the EEM before calling this function.
- Do not call other TLS/DTLS functions before **tls_init()**.

Format

```
t_tls_ret tls_init (void)
```

Arguments

Argument
None.

Return Values

Return value	Description
TLS_OK	Successful execution.
TLS_INIT_ERR	Initialization failed.
Else	See Error Codes.

tls_register_bulk

Use this function to register a bulk encryption algorithm with the EEM.

This passes an encryption handle and the id of the bulk algorithm to TLS.

> **Note:** Call this function after **tls_init()** and before **tls_start()**. Otherwise it returns an error.

Format

```
t_tls_ret tls_register_bulk (
    t_tls_bulk_algorithm_index   idx,
    t_enc_ifc_hdl                ifc_hdl,
    uint16_t                     alg_id )
```

Arguments

Parameter	Description	Type
idx	The index of the bulk algorithm.	t_tls_bulk_algorithm_index
ifc_hdl	The encryption handle of the bulk algorithm.	t_enc_ifc_hdl
alg_id	The bulk algorithm identifier.	uint16_t

Return Values

Return value	Description
TLS_OK	Successful execution.
TLS_PARAM_ERR	Parameter *idx* is invalid.

tls_register_hash

Use this function to register a hash algorithm with the EEM.

This passes an encryption handle and the id of the hash algorithm to TLS.

> **Note:** Call this function after **tls_init()** and before **tls_start()**. Otherwise it returns an error.

Format

```
t_tls_ret tls_register_hash (
    t_tls_hash_algorithm_index   idx,
    t_enc_ifc_hdl                ifc_hdl,
    uint16_t                     alg_id )
```

Arguments

Parameter	Description	Type
idx	The index of the hash algorithm.	t_tls_hash_algorithm_index
ifc_hdl	The encryption handle of the hash algorithm.	t_enc_ifc_hdl
alg_id	The hash algorithm identifier.	uint16_t

Return Values

Return value	Description
TLS_OK	Successful execution.
TLS_PARAM_ERR	Parameter *idx* is invalid.

tls_register_sign

Use this function to register a signature algorithm with the EEM.

This passes an encryption handle and the id of the signature algorithm to TLS.

> **Note:** Call this function after **tls_init()** and before **tls_start()**. Otherwise it returns an error.

If a signature algorithm with the current *idx* is already registered, the function overwrites the previous algorithm with this one.

Format

```
t_tls_ret tls_register_sign (
    t_tls_sign_algorithm_index   idx,
    t_enc_ifc_hdl                ifc_hdl,
    uint16_t                     alg_id )
```

Arguments

Parameter	Description	Type
idx	The index of the signature algorithm.	t_tls_sign_algorithm_index
ifc_hdl	The encryption handle of the signature algorithm.	t_enc_ifc_hdl
alg_id	The signature algorithm identifier.	uint16_t

Return Values

Return value	Description
TLS_OK	Successful execution.
TLS_PARAM_ERR	Parameter *idx* is invalid.

tls_start

Use this function to start the module.

This function starts and allocates instances of the EEM drivers used.

Note:

- Call **tls_init()** before this to initialize the module.
- You must register all required algorithms before you call this function.
- This function must complete successfully before TLS/DTLS can be used.

Format

```
t_tls_ret tls_start ( void )
```

Arguments

Argument
None.

Return values

Return value	Description
TLS_OK	Successful execution.
TLS_INIT_ERR	Operation failed.

tls_stop

Use this function to stop instances of EEM algorithms used by the module.

After this, the module cannot be used until a new call of **tls_start()** has been successfully completed.

Format

```
t_tls_ret tls_stop ( void )
```

Arguments

Argument
None.

Return values

Return value	Description
TLS_OK	Successful execution.
TLS_INIT_ERR	The TLS connection did not stop.

tls_delete

Use this function to delete the mutex and release the resources used by the module.

> **Note:** Only call this after **tls_stop()** has executed successfully.

Format

```
t_tls_ret tls_delete ( void )
```

Arguments

Argument
None.

Return values

Return value	Description
TLS_OK	Successful execution.
Else	See Error Codes.

5.2 TLS Native TCP Interface

TLS can use the native TCP functions described in this section to establish the connection with the peer and to send/receive data.

> **Note:** These functions are only available if the option TLS_TCPIP_IFC_ENABLE is enabled.

tls_start_tcp

Call this function from the client or server application to start the TLS handshake mechanism.

Format

```
t_tls_ret tls_start_tcp ( t_tcp_conn_hdl conn_hdl )
```

Arguments

Parameter	Description	Type
conn_hdl	The handle of the previously established TCP connection.	t_tcp_conn_hdl

Return Values

Return value	Description
TLS_OK	Successful execution.
TLS_NOT_FOUND_ERR	The connection was not found.

tls_close_tcp

Call this function from the client or server application to close the TLS connection.

Format

```
t_tls_ret tls_close_tcp ( t_tcp_conn_hdl conn_hdl )
```

Arguments

Parameter	Description	Type
conn_hdl	The TCP connection handle.	t_tcp_conn_hdl

Return Values

Return value	Description
TLS_OK	Successful execution.
TLS_NOT_FOUND_ERR	The connection was not found.

tls_client_handshake_tcp

Call this function from the client application to establish the TLS connection with the server.

This call is non-blocking. The handshake procedure returns TLS_WAIT status until the handshake is completed. Completion is indicated by the return value TLS_OK or an error code. Poll the handshake function every time a user module gets a notification from the TCP stack on connection *conn_hdl*.

> **Note:** TLS does not implement any timeout mechanism for handshake operation. If this is required, your application must provide it.

Format

```
t_tls_ret tls_client_handshake_tcp (
    const t_tcp_conn_hdl   conn_hdl,
    const char_t * const   p_peer_name,
    const t_tcp_conn_hdl   resume_hdl )
```

Arguments

Parameter	Description	Type
conn_hdl	The TCP connection handle.	t_tcp_conn_hdl
p_peer_name	The name of the server. This is used for certificate verification.	char_t *
resume_hdl	The handle of the connection whose handshake parameters are to be used for resumed sessions. If you do not want to use resumed sessions, set this to TCP_INVALID_CONN_HDL.	t_tcp_conn_hdl

Return Values

Return value	Description
TLS_OK	Successful execution; connection is established.
TLS_WAIT	No data was received from the server.
Else	See Error Codes.

tls_server_handshake_tcp

Call this function from the server application to establish the TLS connection with the client.

This call is non-blocking. The handshake procedure returns TLS_WAIT status until the handshake is completed. Completion is indicated by the return value TLS_OK or an error code. Poll the handshake function every time a user module gets a notification from the TCP stack on connection *conn_hdl*.

> **Note:** TLS does not implement any timeout mechanism for handshake operation. If this is required, your application must provide it.

Format

```
t_tls_ret tls_server_handshake_tcp (
    const t_tcp_conn_hdl   conn_hdl,
    uint8_t                client_verify,
    const char * const     p_peer_name )
```

Arguments

Parameter	Description	Type
conn_hdl	The TCP connection handle.	t_tcp_conn_hdl
client_verify	Set this TRUE if you want to verify the authenticity of the client.	uint8_t
p_peer_name	The name of the client. This is used in certificate verification.	char *

Return Values

Return value	Description
TLS_OK	Successful execution.
TLS_WAIT	No data was received from the client.
Else	See Error Codes.

tls_tcp_accept

Call this function from the TLS server application to accept a connection from a remote node on a port previously opened with **tcp_open()**.

If TLS_TCP_CONN_INF_FLAG_START is set in the *t_tls_conn_inf* structure that *p_inf* points to, the TLS handshake starts immediately after a connection is established. The client is only verified if TLS_TCP_CONN_INF_FLAG_VERIFY is set in this structure.

This call is non-blocking. It returns a new connection handle when it succeeds, but this does not mean that the TLS connection is established. You are notified of the result by a callback, reporting a notification code.

Format

```
t_ip_ret tls_tcp_accept (
    const t_tcp_port_hdl           tcp_port_hdl,
    const t_tls_conn_inf * const   p_inf,
    t_ip_port * const              p_ip_port,
    t_tcp_conn_hdl * const         p_tcp_conn_hdl )
```

Arguments

Parameter	Description	Type
tcp_port_hdl	The port handle on which new connections are accepted.	t_tcp_port_hdl
p_inf	A pointer to the structure containing connection parameters.	t_tls_conn_inf *
p_ip_port	Where to write the remote node IP address and port.	t_ip_port *
p_tcp_conn_hdl	Where to write the connection handle.	t_tcp_conn_hdl *

Return Values

Return value	Description
TLS_OK	Successful execution.
TLS_WAIT	The connection was not accepted.
TLS_NOT_FOUND_ERR	No free TLS connection was found.

tls_tcp_connect

Call this function from the TLS client application to initiate a connection to a remote port using TLS.

If TLS_TCP_CONN_INF_FLAG_START is set in the t_tls_conn_inf structure that *p_inf* points to, the TLS handshake starts immediately after a connection is established. In this structure the field *p_tci_peer_name* must be specified; it is needed to verify the server's certificate.

You are notified of the result by a callback, reporting a notification code.

Format

```
t_ip_ret tls_tcp_connect (
    const t_ip_port *              p_ip_port,
    const t_tls_conn_inf * const   p_inf,
    t_tls_ticket                   conn_ticket,
    t_tcp_conn_hdl * const         p_tcp_conn_hdl )
```

Arguments

Parameter	Description	Type
p_ip_port	A pointer to the address and port number of the server to connect to.	t_ip_port *
p_inf	A pointer to the structure containing connection parameters.	t_tls_conn_inf *
conn_ticket	The handle of the session ticket that the handshake can use in the resume mechanism. If you do not want to resume a connection, set this to TLS_INVALID_CONN_TICKET.	t_ip_port *
p_tcp_conn_hdl	Where to write the connection handle.	t_tcp_conn_hdl *

Return Values

Return value	Description
TLS_OK	Successful execution.
TLS_WAIT	The connection was not opened.
TLS_PARAM_ERR	The *p_tci_peer_name* in *p_inf* was not specified.
TLS_NOT_FOUND_ERR	No free TLS connection was found.

tls_send_tcp

Use this function to send data to the peer over a connection.

This function is non-blocking. The TCP task performs encryption and sends the data.

Format

```
t_tls_ret tls_send_tcp (
    const t_tcp_conn_hdl   conn_hdl,
    uint8_t *              p_data,
    uint16_t               data_len )
```

Arguments

Parameter	Description	Type
conn_hdl	The TCP connection handle.	t_tls_conn_hdl
p_data	A pointer to the IP buffer which contains the data to send. Allocate this by using **tls_get_buffer_tcp()**.	uint8_t *
data_len	The length of the data in bytes.	uint16_t

Return Values

Return value	Description
TLS_OK	Successful execution.
Else	See Error Codes.

tls_receive_tcp

Use this function in either the server or client application to receive data (a PDU) and decode it.

This function is non-blocking. The TCP task performs decryption and notifies the user application that data is ready to read (the code is IP_NTF_TX_RDY). The user application should then call **tls_receive_tcp()**.

Format

```
t_tls_ret tls_receive_tcp (
    t_tcp_conn_hndl   conn_hdl,
    uint8_t * *       pp_buf,
    uint16_t *        p_data_len )
```

Arguments

Parameter	Description	Type
conn_hdl	The TCP connection handle.	t_tcp_conn_hndl
pp_buf	On return, a pointer to the variable holding the pointer to the received data IP buffer. When it is not used, release this buffer by using **tcp_release_buf()**.	uint8_t * *
p_data_len	On return, a pointer to the length of the decoded PDU.	uint16_t *

Return Values

Return value	Description
TLS_OK	Successful execution.
TLS_WAIT	There is no data to receive.
TLS_PARAM_ERR	A parameter was invalid.
TLS_IO_ERR	IO operation failed.

tls_get_buffer_tcp

Use this function to allocate a buffer big enough for the protocol PDU.

If a non-preemptive OS is used, *timeout* is ignored and function execution is non-blocking.

Format

```
t_tls_ret tls_get_buffer_tcp (
    const t_tcp_conn_hdl   conn_hdl,
    uint16_t               data_len,
    uint8_t * *            pp_buf,
    uint32_t               timeout,
    t_ip_ntf * const       p_ntf,
    uint16_t *             p_buf_len )
```

Arguments

Parameter	Description	Type
conn_hdl	The TCP connection handle.	t_tcp_conn_hdl
data_len	The length of the encrypted user data. If the requested size is not available, the call allocates the maximum available buffer.	uint16_t
pp_buf	On return, a pointer to the receive buffer.	uint8_t * *
timeout	The maximum time to wait for a buffer.	uint32_t
p_ntf	A pointer to the notification function. This is called if no buffer could be allocated initially but a buffer then becomes free.	t_ip_ntf *
p_buf_len	On return, a pointer to the variable that receives the obtained buffer size.	uint16_t *

Return Values

Return value	Description
TLS_OK	Successful execution.
TLS_WAIT	The function was called before the handshake was finished.
TLS_MEM_ERR	No buffer of requested size available.

tls_get_ticket_tcp

Use this function to retrieve a ticket that can be used to resume a previously established TCP connection.

Format

```
t_tls_ret tls_get_ticket_tcp (
    const t_tcp_conn_hdl   conn_hdl,
    t_tls_ticket *         p_conn_ticket )
```

Arguments

Parameter	Description	Type
conn_hdl	The handle of the TCP connection.	t_tcp_conn_hdl
p_conn_ticket	On return, a pointer to the variable which received the ticket handle.	t_tls_ticket *

Return Values

Return value	Description
TLS_OK	Successful execution.
TLS_PARAM_ERR	A parameter is incorrect.
TLS_NOT_FOUND_ERR	The specified handle was not found.

tls_get_state_tcp

Call this function to get the status of a TLS connection.

Format

```
t_tls_conn_status tls_get_state_tcp( const t_tcp_conn_hdl conn_hdl )
```

Arguments

Parameter	Description	Type
conn_hdl	The TCP connection handle.	t_tcp_conn_hdl

Return Values

Return value	Description
TLS_CONNST_CLOSED	The connection is closed or does not exist.
TLS_CONNST_HANDSHAKE	The connection is in handshake state.
TLS_CONNST_OPERATING	The connection is established and the user can send and receive data.

5.3 DTLS Native UDP Interface

DTLS can use the native UDP functions described in this section to establish the connection with the peer and to send/receive the data.

> **Note:** These functions are only available when the option DTLS_IFC_ENABLE is enabled.

dtls_start_udp

Call this function from the client or server application to start the DTLS handshake mechanism.

Format

```
t_tls_ret dtls_start_udp ( t_dtls_hdl conn_hdl )
```

Arguments

Parameter	Description	Type
conn_hdl	The handle of the previously established UDP connection.	t_dtls_hdl

Return Values

Return value	Description
TLS_OK	Successful execution.
TLS_NOT_FOUND_ERR	The connection was not found.

dtls_close_udp

Call this function from the client or server application to close the DTLS connection.

Format

```
t_tls_ret dtls_close_tcp ( t_dtls_hdl conn_hdl )
```

Arguments

Parameter	Description	Type
conn_hdl	The DTLS connection handle.	t_dtls_hdl

Return Values

Return value	Description
TLS_OK	Successful execution.
TLS_NOT_FOUND_ERR	The connection was not found.

dtls_udp_srv_open

Call this function to start the DTLS server.

Format

```
t_ip_ret dtls_udp_srv_open (
    uint16_t                        port,
    const t_tls_conn_inf * const    p_inf,
    t_udp_hdl * const               p_conn_hdl )
```

Arguments

Parameter	Description	Type
port	The port number of the server used for listening.	uint16_t
p_inf	A pointer to the structure containing DTLS connection parameters. The client is verified only if TLS_CONN_INF_FLAG_VERIFY is set in this structure.	t_tls_conn_inf *
p_conn_hdl	Where to write the handle of the established UDP server listener.	t_udp_hdl *

Return Values

Return value	Description
TLS_OK	Successful execution.
Else	See Error Codes.

dtls_udp_srv_close

Call this function from the client or server application to close the DTLS server.

This call fails if there are still pending connections on the server.

Format

```
t_tls_ret dtls_udp_srv_close ( t_udp_hdl conn_hdl )
```

Arguments

Parameter	Description	Type
conn_hdl	The handle of the previously established server listener.	t_udp_hdl

Return Values

Return value	Description
TLS_OK	Successful execution.
TLS_NOT_FOUND_ERR	The connection was not found.
TLS_DTLS_CONNECT_ERR	The server is still connected; close all its connections.

dtls_get_srv_conn_udp

Call this function to get the last established DTLS connection.

Format

```
t_tls_ret dtls_get_srv_conn_udp(
    t_udp_hdl            srv_hdl,
    t_dtls_hdl * const   p_conn_hdl )
```

Arguments

Parameter	Description	Type
srv_hdl	The handle of the previously established server listener.	t_udp_hdl
conn_hdl	A pointer to the DTLS connection handle.	t_dtls_hdl *

Return Values

Return value	Description
TLS_OK	Successful execution.
TLS_NOT_FOUND_ERR	The connection was not found.

dtls_accept_udp

Call this function from the DTLS server application to establish the connection with the client.

If TLS_TCP_CONN_INF_FLAG_START is set in the *t_tls_conn_inf* structure that *p_inf* points to, the DTLS handshake starts immediately after a connection is established. The client is only verified if TLS_CONN_INF_FLAG_VERIFY is set in this structure.

This call is non-blocking. It returns a new connection handle when it succeeds, but this does not mean that the DTLS connection is established. You are notified of the result by a callback that reports a notification code.

Format

```
t_ip_ret dtls_accept_udp (
    const t_udp_port_hdl           udp_port_hdl,
    const t_tls_conn_inf * const   p_inf,
    t_ip_port * const              p_ip_port,
    t_dtls_hdl * const             p_dtls_conn_hdl )
```

Arguments

Parameter	Description	Type
udp_port_hdl	The port handle on which new connections are accepted.	t_udp_port_hdl
p_inf	A pointer to the structure containing DTLS connection parameters.	t_tls_conn_inf *
p_ip_port	Where to write the remote node IP address and port.	t_ip_port *
p_dtls_conn_hdl	Where to write the connection handle.	t_dtls_hdl *

Return Values

Return value	Description
TLS_OK	Successful execution.
TLS_WAIT	The connection was not accepted.
TLS_NOT_FOUND_ERR	No free DTLS connection was found.

dtls_connect_udp

Call this function from the DTLS client application to initiate a connection with the server.

If TLS_CONN_INF_FLAG_START is set in the *t_tls_conn_inf* structure that *p_inf* points to, the DTLS handshake starts immediately after a connection is established. In this structure the field *p_tci_peer_name* must be specified; it is needed to verify the server's certificate.

You are notified of the result by a callback, reporting a notification code.

The DTLS connection task also decrypts incoming messages and notifies the user application that data is ready to read (IP_NTF_TX_RDY).

Format

```
t_ip_ret dtls_connect_udp (
    const t_ip_port *               p_ip_port,
    const t_tls_conn_inf * const    p_inf,
    t_tls_ticket                    conn_ticket,
    t_dtls_hdl * const              p_conn_hdl )
```

Arguments

Parameter	Description	Type
p_ip_port	A pointer to the address and port number of the server to connect to.	t_ip_port *
p_inf	A pointer to the structure containing connection parameters.	t_tls_conn_inf *
conn_ticket	The handle of the session ticket that the handshake can use in the resume mechanism. If you do not want to resume a connection, set this to TLS_INVALID_CONN_TICKET.	t_ip_port *
p_tcp_conn_hdl	Where to write the connection handle.	t_dtls_hdl *

Return Values

Return value	Description
TLS_OK	Successful execution.
TLS_WAIT	The connection was not opened.
TLS_PARAM_ERR	The *p_tci_peer_name* in *p_inf* was not specified.
TLS_NOT_FOUND_ERR	No free DTLS connection was found.

dtls_send_udp

Use this function to send data to the peer over a DTLS connection.

This function is non-blocking. The DTLS task performs encryption and sends the data.

If data is sent before the handshake is started, it is not encrypted. You cannot send data when the handshake is in progress.

Format

```
t_tls_ret dtls_send_udp (
    const t_dtls_hdl   conn_hdl,
    uint8_t *          p_data,
    uint16_t           data_len )
```

Arguments

Parameter	Description	Type
conn_hdl	The DTLS connection handle.	t_dtls_hdl
p_data	A pointer to the IP buffer which contains the data to send. Allocate this by using **dtls_get_buffer_udp()**.	uint8_t *
data_len	The length of the data in bytes.	uint16_t

Return Values

Return value	Description
TLS_OK	Successful execution.
Else	See Error Codes.

dtls_receive_udp

Use this function in either the server or client application to receive the decoded data.

This function is non-blocking. The DTLS task performs decryption and notifies the user application that data is ready to read (the notification code is IP_NTF_TX_RDY). The user application should then call **dtls_receive_udp()**.

Format

```
t_tls_ret dtls_receive_udp (
    t_dtls_hndl   conn_hdl,
    uint8_t * *       pp_buf,
    uint16_t *        p_data_len )
```

Arguments

Parameter	Description	Type
conn_hdl	The DTLS connection handle.	t_dtls_hndl
pp_buf	On return, a pointer to the variable holding the pointer to the received data IP buffer. When it is not used, release this buffer by using **tcp_release_buf()**.	uint8_t * *
p_data_len	On return, a pointer to the length of the decoded PDU.	uint16_t *

Return Values

Return value	Description
TLS_OK	Successful execution.
TLS_WAIT	There is no data to receive.
TLS_PARAM_ERR	A parameter was invalid.
TLS_IO_ERR	IO operation failed.

dtls_get_buffer_udp

Use this function to allocate a buffer big enough for the protocol PDU with encrypted user data of size *data_len*.

If a non-preemptive RTOS is used, *timeout* is ignored and function execution is non-blocking.

Format

```
t_tls_ret dtls_get_buffer_udp (
    const t_dtls_hdl   conn_hdl,
    uint16_t           data_len,
    uint8_t * *        pp_buf,
    uint32_t           timeout,
    t_ip_ntf * const   p_ntf,
    uint16_t *         p_buf_len )
```

Arguments

Parameter	Description	Type
conn_hdl	The DTLS connection handle.	t_dtls_hdl
data_len	The length of the encrypted user data. If the requested size is not available, the call allocates the maximum available buffer.	uint16_t
pp_buf	On return, a pointer to the receive buffer.	uint8_t * *
timeout	The maximum time to wait for a buffer.	uint32_t
p_ntf	A pointer to the notification function. This is called if no buffer could be allocated initially but a buffer then becomes free.	t_ip_ntf *
p_buf_len	On return, a pointer to the size of the buffer obtained.	uint16_t *

Return Values

Return value	Description
TLS_OK	Successful execution.
TLS_WAIT	The function was called before the handshake was finished.
TLS_MEM_ERR	No buffer of requested size available.

dtls_get_ticket_udp

Use this function to retrieve a ticket that can be used to resume a previously established DTLS connection.

Format

```
t_tls_ret tls_get_ticket_tcp (
    const t_dtls_hdl   conn_hdl,
    t_tls_ticket *     p_conn_ticket )
```

Arguments

Parameter	Description	Type
conn_hdl	The handle of the DTLS connection.	t_dtls_hdl
p_conn_ticket	On return, a pointer to the variable that received the ticket handle.	t_tls_ticket *

Return Values

Return value	Description
TLS_OK	Successful execution.
TLS_PARAM_ERR	A parameter is incorrect.
TLS_NOT_FOUND_ERR	The specified handle was not found.

dtls_get_state_udp

Call this function to get the status of a TLS connection.

Format

```
t_tls_conn_status tls_get_state_tcp( const t_dtls_hdl conn_hdl )
```

Arguments

Parameter	Description	Type
conn_hdl	The DTLS connection handle.	t_dtls_hdl

Return Values

Return value	Description
TLS_CONNST_CLOSED	The connection is closed or does not exist.
TLS_CONNST_HANDSHAKE	The connection is in handshake state.
TLS_CONNST_OPERATING	The connection is established and the user can send and receive data.

5.4 TLS Sockets Interface

TLS can use the Sockets functions described in this section to establish the connection with the peer and to send/receive data.

> **Note:** These functions are only available when the option TLS_SOCKET_IFC_ENABLE is enabled.

tls_client_handshake_socket

Call this function from the client application to establish a connection with the server.

The call operates as follows:

- If the device running TLS is configured to use a preemptive environment, the call is blocking.
- If the device is not running a preemptive OS, the handshake procedure is polled and returns TLS_WAIT status until it is completed.

Note: TLS does not implement any timeout mechanism for handshake operation. If this is required, your application must provide it.

Format

```
t_tls_ret tls_client_handshake_socket (
    int                     sockfd,
    const char_t * const    p_peer_name,
    t_tls_ticket            conn_ticket )
```

Arguments

Parameter	Description	Type
sockfd	The socket descriptor of the previously established socket connection.	int
p_peer_name	A pointer to the name of the server. This is used for certificate verification.	char_t *
conn_ticket	The handle of the session ticket that can be used in the handshake resume mechanism. If you do not want to resume a connection, set this to TLS_INVALID_CONN_TICKET.	t_tls_ticket

Return Values

Return value	Description
TLS_OK	Successful execution; connection is established.
TLS_WAIT	No data was received from the server.
Else	See Error Codes.

tls_server_handshake_socket

Call this function from the server application to establish the TLS connection with the client.

The call operates as follows:

- If the device running TLS is configured to use a preemptive environment, the call is blocking.
- If the device is not running a preemptive OS, the handshake procedure is polled and returns TLS_WAIT status until it is completed.

> **Note:** TLS does not implement any timeout mechanism for handshake operation. If this is required, your application must provide it.

Format

```
t_tls_ret tls_server_handshake_socket (
    int                 sockfd,
    uint8_t             b_cli_verify,
    const char * const  p_peer_name )
```

Arguments

Parameter	Description	Type
sockfd	The socket descriptor of the previously established socket connection.	int
b_cli_verify	Set this TRUE if you want to verify the authenticity of the client.	uint8_t
p_peer_name	The name of the client. This is used for certificate verification.	char *

Return Values

Return value	Description
TLS_OK	Successful execution; the connection is established.
TLS_WAIT	No data was received from the client.
Else	See Error Codes.

tls_send_socket

Use this function to send application data over an established connection.

Before it is sent, the data is encrypted using connection/session parameters associated with *sockfd*.

Format

```
t_tls_ret tls_send_socket (
    int               sockfd,
    uint8_t * const   p_data,
    uint16_t          data_len )
```

Arguments

Parameter	Description	Type
sockfd	The socket descriptor.	int
p_data	A pointer to the data to be sent.	uint8_t *
data_len	The length of the data in bytes.	uint16_t

Return Values

Return value	Description
TLS_OK	Successful execution.
Else	See Error Codes.

tls_receive_socket

Use this function in either the server or client application to receive and decode the data (PDU).

The receive buffer must be provided by the calling application.

Format

```
t_tls_ret tls_receive_socket (
    int             sockfd,
    uint8_t * *     pp_buf,
    uint16_t        buf_size,
    uint16_t *      p_data_len )
```

Arguments

Parameter	Description	Type
sockfd	The socket descriptor.	int
pp_buf	On return, a pointer to the receive buffer.	uint8_t *
buf_size	The size of the receive buffer.	uint16_t
p_data_len	On return, the length of the decoded PDU.	uint16_t *

Return Values

Return value	Description
TLS_OK	Successful execution.
TLS_WAIT	No data was received.
Else	See Error Codes.

tls_get_ticket_socket

Use this function to retrieve a ticket that can be used to resume a previously established socket connection.

Format

```
t_tls_ret tls_get_ticket_socket (
    int              sockfd,
    t_tls_ticket *   p_conn_ticket )
```

Arguments

Parameter	Description	Type
sockfd	The handle of the socket connection.	int
p_conn_ticket	On return, a pointer to the variable which received the ticket handle.	t_tls_ticket *

Return Values

Return value	Description
TLS_OK	Successful execution.
TLS_PARAM_ERR	A parameter is incorrect.
TLS_NOT_FOUND_ERR	The specified socket was not found.

tls_get_state_socket

Call this function to get the status of a TLS connection.

Format

```
t_tls_conn_status tls_get_state_socket ( int sockfd )
```

Arguments

Parameter	Description	Type
sockfd	The socket descriptor.	int

Return Values

Return value	Description
TLS_CONNST_CLOSED	The connection is closed or does not exist.
TLS_CONNST_HANDSHAKE	The connection is in handshake state.
TLS_CONNST_OPERATING	The connection is established and the user can send and receive data.

tls_close_socket

Call this function from the client or server application to close a TLS connection.

Format

```
t_tls_ret tls_close_socket ( int sockfd )
```

Arguments

Parameter	Description	Type
sockfd	The socket descriptor.	int

Return values

Return value	Description
TLS_OK	Successful execution.
TLS_NOT_FOUND_ERR	Connection not found.

5.5 DTLS Sockets Interface

DTLS can use the Sockets functions described in this section to establish the connection with the peer and to send/receive data.

> **Note:** These functions are only available when the option DTLS_SOCKET_IFC_ENABLE is enabled.

dtls_client_handshake_socket

Call this function from the client application to establish a connection with the server.

The call operates as follows:

- If the device running DTLS is configured to use a preemptive environment, the call is blocking.
- If the device is not running a preemptive RTOS, the handshake procedure is polled and returns TLS_WAIT status until it is completed.

Note: DTLS does not implement any timeout mechanism for handshake operation. If this is required, your application must provide it.

Format

```
t_tls_ret dtls_client_handshake_socket (
    int                     sockfd,
    const t_ip_port *       p_ip_port,
    const char_t * const    p_peer_name,
    t_tls_ticket            conn_ticket,
    t_dtls_hdl * const      p_conn_hdl )
```

Arguments

Parameter	Description	Type
sockfd	The socket descriptor of the previously established socket connection.	int
p_ip_port	A pointer to the port number.	t_ip_port *
p_peer_name	A pointer to the name of the server. This is used for certificate verification.	char_t *
conn_ticket	The handle of the session ticket that can be used in the handshake resume mechanism. If you do not want to resume a connection, set this to TLS_INVALID_CONN_TICKET.	t_tls_ticket
p_conn_hdl	A pointer to the variable that will receive the DTLS connection handle.	t_dtls_hdl *

Return Values

Return value	Description
TLS_OK	Successful execution; connection is established.
TLS_WAIT	No data was received from the server.
TLS_TIMEOUT_ERR	Retransmission failed. You must decide whether to close the connection in this case. The message will still be retransmitted by the DTLS stack.
Else	See Error Codes.

dtls_server_handshake_socket

Call this function from the server application to establish the TLS connection with the client.

The call operates as follows:

- If the device running TLS is configured to use a preemptive environment, the call is blocking.
- If the device is not running a pre-emptive RTOS, the handshake procedure is polled and returns TLS_WAIT status until it is completed.

Note: TLS does not implement any timeout mechanism for handshake operation. If this is required, your application must provide it.

Format

```
t_tls_ret dtls_server_handshake_socket (
    int                     sockfd,
    uint8_t                 client_verify,
    const char * const      p_peer_name,
    t_dtls_socket_hdl *     const p_conn_hdl )
```

Arguments

Parameter	Description	Type
sockfd	The socket descriptor of the previously established socket connection.	int
client_verify	Set this TRUE if you want to verify the authenticity of the client.	uint8_t
p_peer_name	The name of the client. This is used for certificate verification.	char *
p_conn_hdl	A pointer to the variable that will receive the DTLS Sockets interface connection handle.	t_dtls_socket_hdl *

Return Values

Return value	Description
TLS_OK	Successful execution; the connection is established.
TLS_WAIT	No data was received from the client.
Else	See Error Codes.

dtls_send_socket

Use this function to send application data over an established connection.

Before it is sent, the data is encrypted using connection/session parameters associated with *conn_hdl*.

Format

```
t_tls_ret dtls_send_socket (
    t_dtls_hdl       conn_hdl,
    uint8_t * const  p_data,
    uint16_t         data_len )
```

Arguments

Parameter	Description	Type
conn_hdl	The socket handle.	t_dtls_hdl
p_data	A pointer to the data to be sent.	uint8_t *
data_len	The length of the data in bytes.	uint16_t

Return Values

Return value	Description
TLS_OK	Successful execution.
Else	See Error Codes.

dtls_receive_socket

Use this function in either the server or client application to receive the decoded data (PDU).

> **Note:** The receive buffer must be provided by the calling application.

Format

```
t_tls_ret dtls_receive_socket (
    t_dtls_hdl   conn_hdl,
    uint8_t *    p_buf,
    uint16_t     buf_size,
    uint16_t *   p_data_len )
```

Arguments

Parameter	Description	Type
conn_hdl	The socket handle.	t_dtls_hdl
p_buf	On return, a pointer to the receive data buffer.	uint8_t *
buf_size	The size of the receive buffer.	uint16_t
p_data_len	On return, the length of the decoded PDU.	uint16_t *

Return Values

Return value	Description
TLS_OK	Successful execution.
TLS_WAIT	No data was received.
TLS_TIMEOUT_ERR	Heartbeat retransmission failed. You must decide whether to close the connection in this case. The heartbeat will still be retransmitted by the DTLS stack.
Else	See Error Codes.

dtls_get_ticket_socket

Use this function to retrieve a ticket that can be used to resume a previously established DTLS connection.

Format

```
t_tls_ret dtls_get_ticket_socket (
    t_dtls_hdl       conn_hdl,
    t_tls_ticket *   p_conn_ticket )
```

Arguments

Parameter	Description	Type
conn_hdl	The handle of the DTLS connection.	t_dtls_hdl
p_conn_ticket	On return, a pointer to the variable that received the ticket handle.	t_tls_ticket *

Return Values

Return value	Description
TLS_OK	Successful execution.
TLS_PARAM_ERR	A parameter is incorrect.
TLS_NOT_FOUND_ERR	The specified socket was not found.

dtls_get_state_socket

Call this function to get the status of a DTLS connection.

Format

```
t_tls_conn_status dtls_get_state_socket( t_dtls_hdl conn_hdl )
```

Arguments

Parameter	Description	Type
conn_hdl	The socket handle.	t_dtls_hdl

Return Values

Return value	Description
TLS_CONNST_CLOSED	The connection is closed or does not exist.
TLS_CONNST_HANDSHAKE	The connection is in handshake state.
TLS_CONNST_OPERATING	The connection is established and the user can send and receive data.

dtls_close_socket

Call this function from the client or server application to close a DTLS connection.

Format

```
t_tls_ret dtls_close_socket ( t_dtls_socket_hdl p_conn_hdl )
```

Arguments

Parameter	Description	Type
p_conn_hdl	The DTLS connection handle.	t_dtls_socket_hdl

Return values

Return value	Description
TLS_OK	Successful execution.
TLS_NOT_FOUND_ERR	Connection not found.

5.6 Error Codes

The table below lists all the return codes that may be generated by the API calls.

Error code	Value	Meaning
TLS_OK	0	Successful execution.
TLS_SET	1	Flag is set.
TLS_NOT_SET	2	Flag is not set.
TLS_FOUND	3	Specified object was found.
TLS_SIGN_VERIFY	5	Signature should be verified.
TLS_WAIT	6	Function not executed, should be repeated.
TLS_INIT_ERR	7	Initialization error.
TLS_NOT_FOUND_ERR	8	The specified object was not found.
TLS_IO_ERR	9	IO operation error.
TLS_TYPE_ERR	10	The certificate type is incorrect.
TLS_FULL_ERR	11	Array is full, no free slot found.
TLS_NULL_PTR_ERR	12	One of the parameters was a NULL pointer.
TLS_HS_ERR	13	Handshake protocol error.
TLS_PARAM_ERR	14	Invalid parameter error.
TLS_MEM_ERR	15	Memory allocation error.
TLS_VERIFY_ERR	16	Signature verification error.
TLS_CERTIFICATE_ERR	17	A certificate is invalid.
TLS_ENCRYPTION_MODULE_ERR	18	Error returned by Embedded Encryption Manager.
TLS_HEARTBEAT_TIMEOUT_ERR	19	Heartbeat message timeout error.
TLS_DROP_MESSAGE	20	Drop current message silently.
TLS_TIMEOUT_ERR	21	Operation timed out.
TLS_DTLS_NEW_CONN	22	New connection found.
TLS_DTLS_DROP_FRMSG	23	Drop DTLS message fragment.
TLS_DTLS_CONNECT_ERR	24	Server is still connected. Cannot proceed with function.

5.7 Types and Definitions

t_tls_connection

The *t_tls_connection* structure takes this form:

Element	Type	Description
tc_encryp_buf [TLS_DATA_BUF_SIZE]	uint8_t	A temporary buffer used for encryption operations.
tc_cert_oid [TLS_OID_SIZE]	uint8_t	The certificate identifier.
tc_recv_buf [TLS_RECV_BUF_SIZE]	uint8_t	The buffer for received data.
tc_conn_hndl	t_tcp_conn_hdl	The connection handle for TLS with TCP.
tc_conn_hdl_dtls	t_udp_hndl	The connection handle for DTLS with UDP.
		(The following three elements are only used for DTLS UDP.)
tc_ip_addr	t_ip_port	Peer IP address data.
tc_host_port	uint16_t	Host port.
tc_conn_cnt	uint8_t	Connection execution counter.
		(The following four elements are only used for Sockets.)
tc_ntf_idx	uint8_t	The index of the user notify structure that is bound with this connection.
p_tc_rx_q	t_ip_buffer *	The received messages queue.
p_tc_tx_q	t_ip_buffer *	The transmitted messages queue.
tc_timeout	uint16_t	The handshake timeout counter.
tc_dtls_tim_cnt	uint16_t	DTLS timeout counter.
tc_dtls_timeout	uint16_t	DTLS timeout value.
tc_conn_hndl_sock	int	The connection handle for the Sockets API, used by both TLS and DTLS.
tc_mutex	oal_mutex_t	The mutex protecting the connection.
tc_seq_num	uint64_t	The connection sequence number.

Element	Type	Description
tc_peer_seq_num	uint64_t	The peer sequence number.
p_tc_hs_msg_store	uint8_t *	A pointer to the buffer with handshake messages.
p_tc_next_pdu	uint8_t *	A pointer to the next PDU to be processed.
p_tc_pub_key	uint8_t *	A pointer to the DER-encoded public key of the peer.
tc_peer_name [TLS_PEER_NAME_SIZE]	char_t	The name of the peer (only used if peer identity is to be verified).
p_tc_rd_session	t_tls_session *	A pointer to the session object used to read incoming data.
p_tc_wr_session	t_tls_session *	A pointer to the session object used to send data.
p_tc_pd_session	t_tls_session *	A pointer to the session object being negotiated.
p_tc_cert	t_tls_certificate *	The certificate the client uses to identify itself.
p_tc_key_ex_priv	uint8_t *	A pointer to the private value of the key exchange algorithm (used by the EDH algorithm).
tc_host_sec_params	t_tls_security_params	Host side security parameters.
tc_peer_sec_params	t_tls_security_params	Peer side security parameters.
tc_hs_count	uint16_t	Size of the data in the handshake buffer plus four bytes of the handshake message header.
tc_pub_key_len	uint16_t	The length of the DER encoded public key of the peer.
tc_key_ex_len	uint16_t	The length of the private value of the key exchange algorithm (used by the EDH algorithm).
tc_rx_len	uint16_t	The length of the data in the receive buffer.
tc_exp_count	uint16_t	The connection expiry count (this is not used).
tc_state	t_tls_conn_state	The connection state.
tc_flags	t_tls_conn_flags	The connection flags (role and type). The role is client or server.
tc_full_hs	uint8_t	TRUE if a full handshake must be executed, FALSE otherwise.
tc_client_verify	uint8_t	TRUE if the client identity must be verified, FALSE otherwise.

Element	Type	Description
tc_ver_minor	uint8_t	Minor version of TLS protocol for the connection.
tc_ext	t_tls_ext	TLS extension-specific parameters.

The most important elements of *t_tls_connection* are the following:

- The handle of the previously established TCP/IP connection that is used by the application to identify the connection object. This is *tc_conn_hdl* for TLS with TCP, *tc_conn_hdl_dtls* for DTLS with UDP, or *tc_conn_hdl_sock* for either with Sockets.
- Pointers to the read and write TLS sessions. During connection establishment one pending session object is used (*p_tc_pd_session*). This is initialized with the cipher suite negotiated by the two peers. When the connection is established, the *p_tc_pd_session* object is assigned to variables pointing to the read and write session (*p_tc_wr_session* and *p_tc_rd_session*).
- The security parameters of the host and the connection peer. Read and write sessions are used to encrypt outgoing data and decrypt received data using the keys stored in the host side and peer side security parameter objects (*tc_host_sec_params* and *tc_peer_sec_params*).

t_tls_session

The *t_tls_session* structure defines the session, as follows.

Element	Type	Description
ses_id [TLS_MAX_SES_ID_LEN + 1U]	uint8_t	The session id, with one additional byte for its length.
ref_count	uint16_t	A counter of connections using the session object.
ref_neg	uint8_t	A counter of negotiations executed during this session.
master_secret [TLS_PRE_MASTER_SECRET_SIZE]	uint8_t	The pre-master secret.
pre_master_secret_len	uint16_t	The length of the pre-master secret.
p_cipher_suite	t_tls_cipher_suite *	A pointer to the cipher suite used by the session. In the full handshake version (described below) this is set dynamically during connection establishment.

There are two types of session:

- Full handshake – a completely new connection is established after handshaking.
- Resumed connection – a previously negotiated session is used. The *ses_id[]* is used by the client and server when establishing this connection, which means that not all parameters are negotiated. Thus the *t_tls_session* object can be used by more than one connection; *ref_count* contains the number of connections using a given session.

t_tls_security_params

The *t_tls_security_params* structure defines the four types of security parameter supported, as follows.

Element	Type	Description
random_val[TLS_RANDOM_LEN]	uint8_t	Random value used for pre-master key generation.
mac_key[TLS_MAX_MKEY_LEN]	uint8_t	Key used for MAC generation.
bulk_key[TLS_MAX_BKEY_LEN]	uint8_t	Key used for bulk encryption/decryption.
init_vector[TLS_MAX_BKEY_LEN]	uint8_t	Initialization vector for bulk encryption/decryption.

The *mac_key[]* is used by the HMAC algorithm to create the hash value appended to the data of the PDU. This is then encrypted by the bulk algorithm using *bulk_key[]* and *init_vector[]*.

t_tls_hash_alg_attr

The *t_tls_hash_alg_attr* structure defines the hash algorithms, as follows.

Element	Type	Description
digest_prefix [TLS_DIGEST_PREFIX_LEN]	uint8_t	The DER-encoded prefix used for certificate signature calculation.
digest_prefix_len	uint16_t	The length of the signature prefix.
block_len	uint16_t	The length of the data block used by the algorithm.
key_len	uint16_t	The length of the key used by the algorithm.
digest_len	uint16_t	The length of the digest generated by algorithm.

t_tls_certificate

The *t_tls_certificate* structure defines the X.509 certificates, as follows.

Element	Type	Description
cert_pub_key_len	uint16_t	The length of the buffer holding the public key.
cert_priv_key_len	uint16_t	The length of the buffer holding the private key.
cert_block_len	uint32_t	The length of the buffer holding the X.509 certificate.
p_cert_pub_key	uint8_t *	A pointer to the buffer with the DER-encoded public key.
p_cert_priv_key	uint8_t *	A pointer to the buffer with the DER-encoded private key.
p_cert_block	uint8_t *	A pointer to the buffer with the DER-encoded X.509 certificate.
cert_type	uint8_t	The type of certificate.

Notification Codes

The callback notification definitions are as follows:

Name	Value	Description
TLS_NTF_HANDSHAKE_FAILED	0x10000U	TLS handshake failed.
TLS_NTF_HANDSHAKE_DONE	0x20000U	TLS handshake succeeded.
TLS_NTF_HANDSHAKE_TIMEOUT	0x40000U	TLS handshake timed out. This does not close the connection. If a timeout occurs, it is your responsibility to close the connection or wait for the handshake to end.
DTLS_NTF_TIMEOUT	0x80000U	DTLS retransmit timeout.

The following TCP/IP notification may be reported by the TCP interface:

Name	Value	Description
IP_NTF_TX_RDY	0x00000002U	Ready to transmit.

t_tls_conn_inf

The structure *t_tls_conn_inf* is used to encapsulate parameters for TLS and DTLS TCP connections, as follows:

Element	Type	Description
tci_timeout	uint32_t	The connect timeout.
p_tci_ntf	t_ip_ntf *	The notify structure.
tci_flags	t_ip_ntf *	The connection flags (described below).
p_tci_peer_name	char_t *	The peer name.

The following *tci_flags* flags are used to set connection-specific options:

Name	Value	Description
TLS_TCP_CONN_INF_FLAG_START	0x00001U	If this is set the TLS/DTLS handshake is immediately executed after a connection. If this is not set you must call **tls_start_tcp()** or **dtls_start_udp()** to start the handshake process.
TLS_TCP_CONN_INF_FLAG_VERIFY	0x00002U	**This flag is not currently supported - do not use it.** If this is set the TLS stack verifies the client. This is only used for the TLS server (in **tls_accept_tcp()**).

t_dtls_conn_data

The *t_dtls_conn_data* structure is used to encapsulate parameters used by DTLS connections.

Element	Type	Description
dtd_cookie [DTLS_MAX_COOKIE_LEN]	uint8_t	A cookie.
dtd_cookie_length	uint8_t	The cookie length.
dtd_hepoch	uint16_t	The host epoch.
dtd_hseq_num	uint64_t	The host sequence number.
dtd_pepoch	uint16_t	The peer epoch.
dtd_pseq_num	uint64_t	The peer sequence number.
dtd_ar_win_right	uint64_t	anti reply window right value.
dtd_ar_win_left	uint64_t	anti reply window left value.
dtd_ar_window	uint32_t	anti reply window packet mask.
dtd_frg_data_cnt	uint16_t	number of send fragmented data.
dtd_hs_seq_num	uint16_t	handshake sequence number.
dtd_hs_peer_seqnum	uint16_t	handshake peer sequence numbe.
dtd_hs_frag_length	uint16_t	handshake message fragment length.
dtd_hs_send_offset	uint16_t	handshake send message fragment offset.
p_dtd_recv_msg	uint8_t *	pointer to last received message.
p_dtd_frg_msg	uint8_t *	pointer to begining of fragmented data.
dtd_recv_msg_len	uint16_t	length of last received message.
dtd_flight_timeout	uint16_t	handshake flight timeout.
dtd_rtr_cnt	uint16_t	number of messages in flight.
p_dtd_rtr_msg	uint8_t *	start of flight message buffer.
dtd_rtr_cnt_fl	uint8_t	number of messages in currently send flight.
p_dtd_rtr_msg_fl	uint8_t *	start of message buffer in currently send flight.
dtd_rtr_cs_idx	uint8_t	index of change cipher suite message in flight.
dtd_rtr_send_cnt	uint16_t	flight send retransmit count.
dtd_rtr_prv_seq_num	uint64_t	host previous sequence number.
dtd_rtr_cur_seq_num	uint64_t	host current sequence number.

6 Integration

The TLS/DTLS module is designed to be as open and as portable as possible. No assumptions are made about the functionality, the behavior, or even the existence, of the underlying Operating System (OS). For the system to work at its best, perform the porting outlined below. This is a straightforward task for an experienced engineer.

6.1 OS Abstraction Layer (OAL)

The module uses the OS Abstraction Layer (OAL) that allows it to run seamlessly with a wide variety of RTOSes, or without an RTOS.

The module uses the following OAL components:

OAL Resource	Number Required
Tasks	0
Mutexes	2
Events	0

6.2 PSP Porting

The Platform Support Package (PSP) is designed to hold all platform-specific functionality, either because it relies on specific features of a target system, or because this provides the most efficient or flexible solution for the developer. For full details of its functions and macros, see the *Platform Support Package (PSP) Base User Guide*.

The module makes use of the following standard PSP functions:

Function	Package	Element	Description
psp_getcurrenttimedate()	psp_base	psp_rtc	Returns the current time and date. This is used for date- and time-stamping files.
psp_timedatecmp()	psp_base	psp_rtc	Compares the current date/time with a specified date/time.
psp_get_tick_count()	psp_base	psp_tick	Returns the number of milliseconds that have elapsed since the system was started.
psp_malloc()	psp_base	psp_alloc	Allocates a block of memory, returning a pointer to the beginning of the block.
psp_free()	psp_base	psp_alloc	Deallocates a block of memory allocated by **psp_malloc()**, making it available for further allocation.
psp_memcmp()	psp_base	psp_string	Compares two blocks of memory.
psp_memcpy()	psp_base	psp_string	Copies a block of memory. The result is a binary copy of the data.
psp_memset()	psp_base	psp_string	Sets the specified area of memory to the defined value.
psp_strncpy()	psp_base	psp_string	Copies one string of defined length to another.
psp_strncmp()	psp_base	psp_string	Compares two strings of defined length.

The module makes use of the following standard PSP macros:

Macro	Package	Element	Description
PSP_RD_BE16	psp_base	psp_endianness	Reads a 16 bit value stored as big-endian from a memory location.
PSP_RD_BE24	psp_base	psp_endianness	Reads a 24 bit value stored as big-endian from a memory location.
PSP_WR_BE16	psp_base	psp_endianness	Writes a 16 bit value to be stored as big-endian to a memory location.
PSP_WR_BE24	psp_base	psp_endianness	Writes a 24 bit value to be stored as big-endian to a memory location.
PSP_WR_BE32	psp_base	psp_endianness	Writes a 32 bit value to be stored as big-endian to a memory location.
PSP_WR_BE64	psp_base	psp_endianness	Writes a 64 bit value to be stored as big-endian to a memory location.

7 Sample Code

This section gives example code for the module.

7.1 Server Application

The following examples show each of the server interfaces using TCP then Sockets.

TLS Server Interface using Native TCP

The main steps the TLS server implementation must take are:

1. Open a port on which the application accepts TCP connections from clients. TLS implements a special function for accepting a TLS connection.
2. When the TCP connection with a client is established, complete the server side TLS handshake. The handshake procedure is executed by the TLS TCP connection task. When the handshake ends, the user application is notified by a callback. TLS sends a receive notification to the user application.
3. Read incoming requests from clients in the loop using **tls_receive_tcp()** and handle these appropriately.

The following example shows pseudocode for the TLS server application:

```
#define TLS_TEST_EVENT    0x1
static t_ip_ntf        g_tls_test_ntf =
{
    tls_test_ntf_fn,
    0U,
    NULL
};
static oal_event_t     g_tls_test_event;

oal_event_create( &( g_tls_test_event ) );
tcp_open( TLS_PORT, 1U, NULL, &g_tls_port_hdl );
tls_tcp_accept( g_tls_port_hdl, &p_inf, &ip_rx_port, &g_tls_rx_conn_hdl );
static void tls_test_ntf_fn ( uint32_t param, uint32_t ntf )

{
    (void)param;
    if ( ntf != TLS_NTF_HANDSHAKE_DONE )
    {
        (void)oal_event_set( &g_tls_test_event, TLS_TEST_EVENT, g_tls_test_task_id );
    }
}

OAL_TASK_FN( tls_server_task )
{
    uint8_t   * p_rx_buf;
    uint16_t    rx_bytes;
    t_tls_ret   tls_res;
    oal_event_flags_t  event_flags;     /* Set event flags */
    oal_ret_t          oal_ret;         /* Event get return value */
#if OAL_TASK_POLL_MODE == 0
    ip_enter_task();
    for ( ; ; )
#endif
    {
        oal_ret = oal_event_get( &g_tls_test_event, TLS_TEST_EVENT, &event_flags,
OAL_WAIT_FOREVER );
        if ( ( oal_ret == OAL_SUCCESS ) && ( event_flags == TLS_TEST_EVENT ) )
        {   /* TLS connection is established */
            rx_bytes = 0;
            tls_res = tls_receive_tcp( conn_hdl, &p_rx_buf, &rx_bytes );
            if ( rx_bytes > 0 )
            {
                p_rx_buf[rx_bytes] = '\0';
                PRINTF( "%d bytes received: %s\n", rx_bytes, p_rx_buf );
                if ( tls_send_tcp( conn_hdl, p_rx_buf, rx_bytes ) != TLS_OK )
                {
                    tcp_release_buf( p_rx_buf );
                }
            }
        }
    }
}
```

DTLS Server Interface using UDP

The main steps the TLS server implementation must take are:

1. Open a port on which the application accepts UDP packets from clients. DTLS implements a special function for creating a server port.
2. When the first UDP packet from a client is received, complete the server side DTLS handshake. The handshake procedure is executed by the DTLS connection task. When the handshake ends, the user application is notified by a callback. DTLS sends a receive notification to the user application.
3. Read incoming requests from clients in the loop using **dtls_receive_udp()** and handle these appropriately.

The following example shows pseudocode for the DTLS server application:

```
#define TLS_TEST_EVENT    0x1
static t_ip_ntf        g_tls_test_ntf =
{
    tls_test_ntf_fn,
    0U,
    NULL
};
static oal_event_t     g_tls_test_event;
static t_dtls_hdl      g_dtls_rx_conn_hdl;
static t_udp_hdl       g_tls_udp_hdl = UDP_INVALID_HDL;

t_tls_conn_inf inf;
oal_event_create( &( g_tls_test_event ) );
inf.p_tci_ntf = &g_tls_test_ntf;
inf.p_tci_peer_name = NULL;
inf.tci_flags = TLS_TCP_CONN_INF_FLAG_START;
tls_res = dtls_udp_srv_open( TLS_PORT, &inf, &g_tls_udp_hdl );

static void tls_test_ntf_fn ( uint32_t param, uint32_t ntf )
{
    (void)param;
    if ( ntf != TLS_NTF_HANDSHAKE_DONE )
    {
        (void)oal_event_set( &g_tls_test_event, TLS_TEST_EVENT, g_tls_test_task_id );
    }
}

OAL_TASK_FN( tls_server_task )
{
    uint8_t   * p_rx_buf;
    uint16_t    rx_bytes;
    t_tls_ret   tls_res;
    oal_event_flags_t  event_flags;     /* Set event flags */
    oal_ret_t          oal_ret;         /* Event get return value */
#if OAL_TASK_POLL_MODE == 0
    ip_enter_task();
    for ( ; ; )
#endif
    {
    oal_ret = oal_event_get( &g_tls_test_event, TLS_TEST_EVENT, &event_flags, OAL_WAIT_FOREVER
);
        if ( ( oal_ret == OAL_SUCCESS ) && ( event_flags == TLS_TEST_EVENT ) )
          {
            if ( *p_conn_hdl == DTLS_INVALID_UDP_HDL )
            {
                oal_ret = dtls_get_srv_conn_udp( g_tls_udp_hdl, &g_dtls_rx_conn_hdl );
            }
        }
        if ( ( oal_ret == OAL_SUCCESS ) && ( event_flags == TLS_TEST_EVENT ) )
        {   /* TLS connection is established */
            rx_bytes = 0;
            tls_res = dtls_receive_udp( g_dtls_rx_conn_hdl, &p_rx_buf, &rx_bytes );
            if ( rx_bytes > 0 )
            {
                p_rx_buf[rx_bytes] = '\0';
                PRINTF( "%d bytes received: %s\n", rx_bytes, p_rx_buf );
```

```
                if ( dtls_send_udp( g_dtls_rx_conn_hdl, p_rx_buf, rx_bytes ) != TLS_OK )
                {
                    udp_release_buf( p_rx_buf );
                }
            }
        }
    }
}
```

TLS Server Interface using Sockets

The main steps the TLS server implementation must take are:

1. Open a port on which the application accepts TCP connections from clients.
2. When the TCP connection with a client is established, complete the server side TLS handshake. The handshake procedure is polled and returns TLS_WAIT status until the handshake is completed; this is indicated by the return code TLS_OK or an error code.
3. Read incoming requests from clients in the loop using **tls_receive_socket()** and handle these appropriately.

The following example shows pseudocode for the TLS server application:

```
struct sockaddr_in  sa_addr;
struct sockaddr_in  sa_peer_addr;
socklen_t addr_len;
int sd_svr;
int sd_conn;
uint8_t  enc_buf[1024];
uint8_t  dec_buf[1024];

sd_svr = socket( AF_INET, SOCK_STREAM, 0 );
psp_memset( (char *)&sa_addr, 0, sizeof(sa_addr) );
sa_addr.sin_family = AF_INET;
sa_addr.sin_addr.s_addr = IN_ADDR_ANY ;
sa_addr.sin_port = TLS_PORT;
bind( sd_svr, (struct sockaddr *)&sa_addr, sizeof(sa_addr) );

if (listen( sd_svr, 4 ) > -1)
    {
        sd_conn = accept( sd_svr, (struct sockaddr *)&sa_peer_addr, &addr_len );
    }

OAL_TASK_FN( tls_server_task )
{
#if OAL_PREEMPTIVE
    tls_res = TLS_WAIT
    while ( tls_res == TLS_WAIT )
#endif  /* First establish connection by handshake process */
    {
        tls_res = tls_server_handshake_socket( sd_conn, FALSE, NULL );
    }

#if OAL_PREEMPTIVE
    while ( tls_res == TLS_OK )
#else
    if ( tls_res == TLS_OK )
#endif
    { /* TLS connection is established. For native API buffer size is 0 */
        recv_bytes = 0;
        p_buf = dec_buf;
        tls_res = tls_receive_socket( sd_conn, &p_buf, sizeof( dec_buf ), &recv_bytes );
        if ( recv_bytes > 0 )
        {
            p_rx_buf[rx_bytes] = '\0';
            PRINTF( "%d bytes received: %s\n", rx_bytes, p_rx_buf );
            tls_send_socket( conn_hdl, p_rx_buf, rx_bytes );
            ip_release_proto_buf( p_rx_buf );
        }
    }
}
```

DTLS Server Interface using Sockets

The main steps the DTLS server implementation must take are:

1. Open a port on which the application accepts UDP connections from clients.
2. When the UDP connection with a client is established, complete the server side DTLS handshake. The handshake procedure is polled and returns TLS_WAIT status until the handshake is completed; this is indicated by the return code TLS_OK or an error code.
3. Read incoming requests from clients in the loop using **dtls_receive_socket()** and handle these appropriately.

The following example shows pseudocode for the DTLS server application:

```
struct sockaddr_in  sa_addr;
struct sockaddr_in  sa_peer_addr;
socklen_t addr_len;
int sd_svr;
t_dtsl_hdl conn_hdl;
uint8_t  enc_buf[1024];
uint8_t  dec_buf[1024];

sd_svr = socket_open( AF_INET, SOCK_STREAM, 0 );
psp_memset( (char *)&sa_addr, 0, sizeof( sa_addr ) );
sa_addr.sin_family = AF_INET;
sa_addr.sin_addr.s_addr = IN_ADDR_ANY ;
sa_addr.sin_port = TLS_PORT;
socket_bind( sd_svr, (struct sockaddr *)&sa_addr, sizeof( sa_addr ) );

OAL_TASK_FN( tls_server_task )
{
#if OAL_PREEMPTIVE
    tls_res = TLS_WAIT
    while ( tls_res == TLS_WAIT )
#endif  /* First establish connection by handshake process */
    {
        tls_res = dtls_server_handshake_socket( sd_svr, FALSE, NULL, &conn_hdl );
    }

#if OAL_PREEMPTIVE
    while ( tls_res == TLS_WAIT ) ||  ( tls_res == TLS_DTLS_NEW_CONN ) )
#else
    if ( tls_res == TLS_OK )
#endif
    { /* TLS connection is established. For native API buffer size is 0. */
        recv_bytes = 0;
        p_buf = dec_buf;
        tls_res = dtls_receive_socket( conn_hdl, &p_buf, sizeof( dec_buf ), &recv_bytes );
        if ( recv_bytes > 0 )
        {
            p_rx_buf[rx_bytes] = '\0';
            PRINTF( "%d bytes received: %s\n", rx_bytes, p_rx_buf );
            dtls_send_socket( conn_hdl, p_rx_buf, rx_bytes );
        }
    }
}
```

7.2 Client Application

The following examples show each of the client interfaces using TCP then Sockets.

TLS Client Interface using Native TCP

The main steps the TLS client implementation must take are:

1. Establish the TCP socket connection with the server.
2. Complete the client side TLS handshake. This means calling **tls_tcp_connect()** and waiting for the TLS_NTF_HANDSHAKE_DONE notification.
3. Start to send and receive data.

The following example shows pseudocode for the TLS client application:

```
#define TLS_TEST_EVENT    0x1
static t_ip_ntf        g_tls_test_ntf =
{
    tls_test_ntf_fn
    , 0U
    , NULL
};
static oal_event_t     g_tls_test_event;
oal_event_create( &( g_tls_test_event ) );
ret_val = tls_tcp_connect( &ip_tx_port,
                           IP_WAIT_FOREVER,
                           &g_tls_test_ntf,
                           "ServerName",
                           TLS_INVALID_CONN_TICKET,
                           &g_tls_rx_conn_hdl );

static void tls_test_ntf_fn( uint32_t param, uint32_t ntf );
{
    (void)param;
    if ( ntf != TLS_NTF_HANDSHAKE_DONE )
    {
        (void)oal_event_set( &g_tls_test_event, TLS_TEST_EVENT, g_tls_test_task_id );
    }
}

OAL_TASK_FN( tls_server_task )
{
    uint8_t   * p_rx_buf;
    uint16_t    rx_bytes;
    t_tls_ret   tls_res;
    oal_event_flags_t  event_flags;     /* Set event flags */
    oal_ret_t          oal_ret;         /* Event get return value */
#if OAL_TASK_POLL_MODE == 0
    ip_enter_task();
    for ( ; ; )
#endif
    {
        oal_ret = oal_event_get( &g_tls_test_event, TLS_TEST_EVENT, &event_flags,
OAL_WAIT_FOREVER );
        if ( ( oal_ret == OAL_SUCCESS ) && ( event_flags == TLS_TEST_EVENT ) )
        { /* TLS connection is established */
            rx_bytes = 0;
            tls_res = tls_receive_tcp( conn_hdl, &p_rx_buf, &rx_bytes );
            if ( rx_bytes > 0 )
            {
                p_rx_buf[rx_bytes] = '\0';
                PRINTF( "%d bytes received: %s\n", rx_bytes, p_rx_buf );
                if ( tls_send_tcp( conn_hdl, p_rx_buf, rx_bytes ) != TLS_OK )
                {
                    tcp_release_buf( p_rx_buf );
                }
            }
        }
    }
}
```

DTLS Client Interface using UDP

The main steps the DTLS client implementation must take are:

1. Establish the connection with the server.
2. Complete the client side DTLS handshake. This means calling **dtls_connect()** and waiting for the TLS_NTF_HANDSHAKE_DONE notification.
3. Start to send and receive data.

The following example shows pseudocode for the DTLS client application:

```
#define TLS_TEST_EVENT    0x1
static t_ip_ntf        g_tls_test_ntf =
{
    tls_test_ntf_fn
    , 0U
    , NULL
};
static oal_event_t     g_tls_test_event;
static t_dtls_hdl      g_dtls_rx_conn_hdl;

oal_event_create( &( g_tls_test_event ) );
ip_tx_port.ipp_ip_addr = TLS_TEST_SVR_IP_ADDR;
ip_tx_port.ipp_port    = TLS_TEST_PORT;
inf.p_tci_ntf = &g_tls_cs_ntf;
inf.p_tci_peer_name = TLS_TEST_PEERNAME;
inf.tci_flags = TLS_TCP_CONN_INF_FLAG_START;
inf.tci_timeout = IP_WAIT_FOREVER;

ret_val = dtls_connect_udp( &ip_tx_port, IP_WAIT_FOREVER, &inf, TLS_INVALID_CONN_TICKET,
&g_dtls_rx_conn_hdl );

static void tls_test_ntf_fn( uint32_t param, uint32_t ntf );
{
    (void)param;
    if ( ntf != TLS_NTF_HANDSHAKE_DONE )
    {
        (void)oal_event_set( &g_tls_test_event, TLS_TEST_EVENT, g_tls_test_task_id );
    }
}

OAL_TASK_FN( tls_server_task )
{
    uint8_t   * p_rx_buf;
    uint16_t    rx_bytes;
    t_tls_ret   tls_res;
    oal_event_flags_t  event_flags;     /* Set event flags */
    oal_ret_t          oal_ret;         /* Event get return value */
#if OAL_TASK_POLL_MODE == 0
    ip_enter_task();
    for ( ; ; )
#endif
    {
        oal_ret = oal_event_get( &g_tls_test_event, TLS_TEST_EVENT, &event_flags,
OAL_WAIT_FOREVER );
        if ( ( oal_ret == OAL_SUCCESS ) && ( event_flags == TLS_TEST_EVENT ) )
        { /* TLS connection is established */
            rx_bytes = 0;
            tls_res = dtls_receive_udp( g_tls_rx_conn_hdl, &p_rx_buf, &rx_bytes );
            if ( rx_bytes > 0 )
            {
                p_rx_buf[rx_bytes] = '\0';
                PRINTF( "%d bytes received: %s\n", rx_bytes, p_rx_buf );
                if ( dtls_send_udp( g_tls_rx_conn_hdl, p_rx_buf, rx_bytes ) != TLS_OK )
                {
                    udp_release_buf( p_rx_buf );
                }
```

```
            }
        }
    }
}
```

TLS Client Interface using Sockets

The main steps the TLS client implementation must take are:

1. Establish the TCP Sockets connection with the server.
2. Complete the client side TLS handshake.

The following example shows pseudocode for the TLS client application:

```
int sd_conn;
uint8_t  enc_buf[1024];
uint8_t  dec_buf[1024];

sd_conn = socket( AF_INET, SOCK_STREAM, 0 )
memset( (char *)&sa_addr, 0, sizeof(sa_addr) );
sa_addr.sin_family = AF_INET;
sa_addr.sin_addr.s_addr = TLS_SVR_IP_ADDR;
sa_addr.sin_port = TLS_PORT;
connect( sd_conn, (struct sockaddr *)&sa_addr, sizeof(sa_addr) );

OAL_TASK_FN( tls_client_task )
{
#if OAL_PREEMPTIVE
    tls_res = TLS_WAIT
    while ( tls_res == TLS_WAIT )
#endif  /* First establish connection using handshake process */
    {
        tls_res = tls_client_handshake_socket( sd_conn, "ServerName", TLS_INVALID_CONN_TICKET );
    }
#if OAL_PREEMPTIVE
    while ( tls_res == TLS_OK )
#else
    if ( tls_res == TLS_OK )
#endif
    { /* TLS connection is established. For native API buffer size is 0 */
        recv_bytes = 0;
        p_buf = dec_buf;
        tls_res = tls_receive_socket( sd_conn, &p_buf, sizeof( dec_buf ), &recv_bytes );
        if ( recv_bytes > 0 )
        {
            p_rx_buf[rx_bytes] = '\0';
            PRINTF( "%d bytes received: %s\n", rx_bytes, p_rx_buf );
            tls_send_socket( conn_hdl, p_rx_buf, rx_bytes );
            ip_release_proto_buf( p_rx_buf );
        }
    }
}
```

DTLS Client Interface using Sockets

The main steps the DTLS client implementation must take are:

1. Establish the UDP Socket connection with the server.
2. Complete the client side DTLS handshake.

The following example shows pseudocode for the DTLS client application:

```
int sd_conn;
static t_dtls_hdl     g_dtls_rx_conn_hdl;
uint8_t  enc_buf[1024];
uint8_t  dec_buf[1024];

sd_conn = socket( AF_INET, SOCK_DGRAM, 0 )
memset( (char *)&sa_addr, 0, sizeof(sa_addr) );
sa_addr.sin_family = AF_INET;
sa_addr.sin_addr.s_addr = TLS_SVR_IP_ADDR;
sa_addr.sin_port = TLS_PORT;
socket_bind( sd_conn, (struct sockaddr *)&sa_addr, sizeof( sa_addr ) );

OAL_TASK_FN( tls_client_task )
{
#if OAL_PREEMPTIVE
    tls_res = TLS_WAIT
    while ( tls_res == TLS_WAIT )
#endif  /* First establish connection using handshake process */
    {
        tls_res = dtls_client_handshake_socket( sd_conn, "ServerName", TLS_INVALID_CONN_TICKET,
&g_dtls_rx_conn_hdl );
    }
#if OAL_PREEMPTIVE
    while ( tls_res == TLS_OK )
#else
    if ( tls_res == TLS_OK )
#endif
    { /* TLS connection is established. For native API buffer size is 0 */
        recv_bytes = 0;
        p_buf = dec_buf;
        tls_res = dtls_receive_socket( g_dtls_rx_conn_hdl, &p_buf, sizeof( dec_buf ),
&recv_bytes );
        if ( recv_bytes > 0 )
        {
            p_rx_buf[rx_bytes] = '\0';
            PRINTF( "%d bytes received: %s\n", rx_bytes, p_rx_buf );
            dtls_send_socket( g_dtls_rx_conn_hdl, p_rx_buf, rx_bytes );
        }
    }
}
```