# Table of Contents

1 System Overview ........................................................................................................ 7

1.1 Introduction .............................................................................................................. 8

   EAP-TLS ....................................................................................................................... 9

   Random Number Generation ...................................................................................... 9

1.2 System Integration ..................................................................................................... 10

   TLS Native TCP Interface ......................................................................................... 10

   DTLS Native UDP Interface ....................................................................................... 11

   TLS Sockets Interface ................................................................................................. 12

   DTLS Sockets Interface ............................................................................................... 13

   TLS Raw Interface ....................................................................................................... 14

1.3 Feature Check .......................................................................................................... 15

1.4 Packages and Documents ......................................................................................... 16

   Packages .................................................................................................................... 16

   Documents ................................................................................................................ 16

1.5 Change History ......................................................................................................... 18

2 Source File List ........................................................................................................... 20

2.1 API Header File ......................................................................................................... 20

2.2 Configuration Files ................................................................................................. 20

2.3 Demo Files ................................................................................................................ 20

2.4 Public Certificates .................................................................................................... 21

2.5 config_suite_b .......................................................................................................... 21

2.6 Version File ............................................................................................................. 21

2.7 System Files ............................................................................................................ 22

3 Configuration Options ................................................................................................. 24

3.1 Specifying TLS or DTLS and the Interface Used .................................................... 24

3.2 config_tls.h ............................................................................................................. 25

   Protocol Support Options .......................................................................................... 25

   TLS Options ............................................................................................................... 25

   DTLS Options ............................................................................................................ 30

   Further TLS Options .................................................................................................. 31

   Elliptic Curve cryptography configuration ................................................................ 33

3.3 config_tls.c ............................................................................................................. 35
# TLS and DTLS User Guide

## Running the HCC Demo

Running the HCC Demo

## About TLS and DTLS

### 5.1 Client and Server Roles

- **TLS**
  - Clients
  - Servers
- **DTLS**
  - Clients
  - Servers

### 5.2 The TLS Session

- **TLS Connection Establishment: Handshaking**
- **TLS Data Exchange**

### 5.3 The DTLS Session

- **DTLS Connection Establishment: Handshaking**
- **DTLS Data Exchange**

### 5.4 The Embedded Encryption Manager

- **Algorithms**
- **Hash Algorithms**
- **Signature Algorithms**
- **Bulk Encryption Algorithms**
- **Elliptic Curve Cryptography**
- **Certificates**
- **Cipher Suites**

## Application Programming Interface

### 6.1 Module Management

- **tls_init**
- **tls_start**
- **tls_stop**
- **tls_delete**

### 6.2 Algorithm Management

- **tls_register_aead**
- **tls_register_bulk**
- **tls_register_hash**
- **tls_register_sign**
- **tls_register_sign_alg**
- **tls_add_certificate**
6.3 Certificate Management Functions .......................................................... 66
  tls_set_host_ca_chain ................................................................. 67
  tls_add_cipher_suite ............................................................ 68
  tls_register_cert_store_cb ................................................... 69
  tls_register_cert_validate_cb ............................................. 70
  tls_register_get_ca_chain_cb ............................................. 71

6.4 Certificate Management Callbacks ...................................................... 72
  t_tls_cert_get_cb ............................................................. 73
  t_tls_cert_get_ca_chain_cb .............................................. 74
  t_tls_cert_valid_cb .......................................................... 75

6.5 TLS Native TCP Interface .................................................................... 76
  tls_start_tcp ................................................................. 77
  tls_close_tcp ................................................................. 78
  tls_renegotiate_tcp ......................................................... 79
  tls_tcp_accept ............................................................... 80
  tls_tcp_connect .............................................................. 81
  tls_send_tcp ................................................................. 82
  tls_receive_tcp .............................................................. 83
  tls_rx_ready_tcp .............................................................. 84
  tls_get_buffer_tcp ........................................................... 85
  tls_get_ticket_tcp ............................................................ 86
  tls_get_state_tcp ............................................................. 87

6.6 DTLS Native UDP Interface ............................................................... 88
  dtls_start_udp ............................................................... 89
  dtls_close_udp ............................................................... 90
  dtls_udp_srv_open .......................................................... 91
  dtls_udp_srv_close .......................................................... 92
  dtls_get_srv_conn_udp .................................................... 93
  dtls_connect_udp ............................................................. 94
  dtls_send_udp ............................................................... 95
  dtls_receive_udp ............................................................ 96
  dtls_get_buffer_udp ........................................................ 97
  dtls_get_ticket_udp ........................................................ 98
  dtls_get_state_udp .......................................................... 99
  dtls_get_ip_udp ..............................................................100

6.7 TLS Sockets Interface ......................................................................... 101
7 Integration .................................................................................................................. 157
  7.1 OS Abstraction Layer ......................................................................................... 157
  7.2 PSP Porting ........................................................................................................ 158
      Collaboration Diagrams ......................................................................................... 158
          TLS Native TCP Interface .............................................................................. 160
          DTLS Native UDP Interface ............................................................................ 161
          TLS Sockets Interface ....................................................................................... 162
          DTLS Sockets Interface ..................................................................................... 163
          TLS Raw Interface ............................................................................................. 164
  8 Sample Code .......................................................................................................... 166
    8.1 Server Application ............................................................................................ 166
        TLS Server Interface using Native TCP ............................................................. 167
        DTLS Server Interface using Native TCP ......................................................... 168
        TLS Server Interface using Sockets ................................................................... 171
        DTLS Server Interface using Sockets ................................................................ 172
    8.2 Client Application .............................................................................................. 188
        TLS Client Interface using Native TCP ............................................................. 188
        DTLS Client Interface using Native TCP ......................................................... 189
        TLS Client Interface using Sockets ................................................................... 191
        DTLS Client Interface using Sockets ................................................................ 192

    Flags ....................................................................................................................... 156
    Notification Codes ................................................................................................. 149
    t_tls_certificate ..................................................................................................... 149
    t_tls_connection ...................................................................................................... 144
    t_tls_ecc_curve_dsc ............................................................................................... 152
    t_tls_cipher_suite ................................................................................................. 153
    t_tls_conn_inf ......................................................................................................... 150
    t_tls_conn_data ...................................................................................................... 151
    t_tls_conn_status ................................................................................................. 155
    t_tls_alert_type ....................................................................................................... 154
    t_tls_hash_alg_attr .............................................................................................. 148
    t_tls_session ........................................................................................................... 147
1 System Overview

This chapter contains the fundamental information for this module.

The component sections are as follows:

- **Introduction** – describes the main elements of the module. This section includes a diagram showing the position of this module within HCC's TCP/IP stack.
- **System Integration** - presents collaboration diagrams that show the components and their interactions, and also the client/server relationship.
- **Feature Check** – summarizes the main features of the module as bullet points.
- **Packages and Documents** – the *Packages* section lists the packages that you need in order to use this module. The *Documents* section lists the relevant user guides.
- **Change History** – lists the earlier versions of this manual, giving the software version that each manual describes.

**Note:** To download this manual as a PDF, see [TCP/IP Security PDFs](#).
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 three 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.
- TLS interfacing to HCC’s EAP-TLS module (EAP is the Extensible Authentication Protocol). The EAP-TLS module interfaces to the TLS RAW 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.

**EAP-TLS**

The TLS RAW interface that is used to interface TLS to HCC’s EAP-TLS module is described in this manual, but for all other information on EAP and EAP-TLS, refer to the relevant manuals.

**Random Number Generation**

Random Number Generation (RNG) is critical in security. Random numbers are used as secret values when negotiating keys. If an attacker can predict the secret numbers, it is easy for them to generate the keys.

Use of a True Random Number Generator (TRNG) is recommended. Software implementations can only implement a Pseudo Random Number Generator (PRNG), which may be predicted by an attacker.

To implement a TRNG, a special hardware module is needed. Most modern chips implement a special module that can be used as a TRNG. If a device does not have a dedicated RND module, it can add an external random generator chip or implement the NIST recommended random generator which uses RealTimeClock: see *ANSI X9.31-1998 Appendix A.2.4*.

HCC provides a simple pseudo-random generator module `psp_getrand();` port this to use your platform-specific RNG module.
1.2 System Integration

The five collaboration diagrams in this section show the different ways you can use the system. Each interface can be used individually, or any combination of the five interfaces can be used at the same time. All five interfaces can be used simultaneously if desired.

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

**TLS Native TCP Interface**
DTLS Native UDP Interface

TLS Client App
- dtls_start_udp()
- dtls_close_udp()
- dtls_connect_udp()
- dtls_send_udp()
- dtls_receive_udp()
- dtls_get_buffer_udp()
- dtls_get_ticket_udp()
- dtls_get_state_udp()

TCP connection

TLS Server App
- dtls_start_udp()
- dtls_close_udp()
- dtls_udp_srv_open()
- dtls_udp_srv_close()
- dtls_get_srv_conn_udp()
- dtls_send_udp()
- dtls_receive_udp()
- dtls_get_buffer_udp()
- dtls_get_ticket_udp()
- dtls_get_state_udp()

TLS Library
- udp_get_buf()
- udp_open()
- udp_close()
- udp_rx_ready()
- udp_send()
- udp_receive()

TCP Native Interface
TLS Sockets Interface

TLS Client App
- tls_client_handshake_socket()
- tls_send_socket()
- tls_receive_socket()
- tls_select_socket()
- tls_poll_socket()
- tls_get_ticket_socket()
- tls_get_state_socket()
- tls_close_socket()

TLS Server App
- tls_server_handshake_socket()
- tls_send_socket()
- tls_receive_socket()
- tls_select_socket()
- tls_poll_socket()
- tls_get_ticket_socket()
- tls_get_state_socket()
- tls_close_socket()

TLS Library
- socket_select()
- socket_poll()
- socket_send()
- socket_recv()
- socket_ioctl()
- socket_getsockopt()

Socket Library
DTLS Sockets Interface

TLS Client App

TCP connection

tls_client_handshake_socket()
dtls_send_socket()
dtls_receive_socket()
dtls_get_ticket_socket()
dtls_get_state_socket()
dtls_close_socket()

TLS Library

socket_sendto()
socket_recvfrom()
socket_ioctl()
socket_getsockopt()
socket_htons()
socket_ntohs()
socket_ntols()

Socket Library

TLS Server App

dtls_server_handshake_socket()
dtls_send_socket()
dtls_receive_socket()
dtls_get_ticket_socket()
dtls_get_state_socket()
dtls_close_socket()
TLS Raw Interface

RAW Interface

Client

tls_init_raw()
tls_close_raw()
tls_read_data_raw()
tls_hdl_data_raw()
tls_start_raw()
tls_get_ticket_raw()
tls_get_state_raw()
tls_get_read_data_raw()
tls_read_data_ack_raw()
tls_get_secrets_raw()
tls_get_randoms_raw()

RAW Interface

Server

tls_init_raw()
tls_close_raw()
tls_read_data_raw()
tls_hdl_data_raw()
tls_start_raw()
tls_get_ticket_raw()
tls_get_state_raw()
tls_get_read_data_raw()
tls_read_data_ack_raw()
tls_get_secrets_raw()
tls_get_randoms_raw()

TLS Library

t_tls_cert_get_cb()
t_tls_cert_valid_cb()

External Certificate Store
1.3 Feature Check

The main features of the system are the following:

- Conforms to the HCC Advanced Embedded Framework.
- Designed for integration with both RTOS and non-RTOS based systems.
- 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.
- Designed for microcontrollers, ensuring a low memory footprint. This is typically around 20KB of ROM or 8KB of RAM.
- Typically uses a standard Sockets interface, allowing easy integration with many embedded applications.
- Supports TLS 1.0, 1.1 and 1.2 (RFC 5246) and SSL 3.0 and is verifiable.
- Supports DTLS version 1.2 (RFC 6347) and version 1.0 (RFC 4347).
- Supports HCC’s EAP-TLS module (through its RAW interface).
- Supports heartbeat extensions (RFC 6520).
- Supports HTTP over TLS (RFC 2818).
- Provides HTTP or FTP Server support for HTTPS and FTPS implementations, or for connection to any other secure client or server application.
- Uses HCC’s Embedded Encryption Manager (EEM) to provide full certificate management.
- 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.
- Supports all the mandatory cipher suites required by different versions of TLS.
- Supports Elliptic Curve Cryptography (ECC) (RFC 4492).
- Supports Authenticated Encryption with Associated Data (AEAD).

The supported algorithms are the following:

- Advanced Encryption Standard (AES).
- Digital Signature Standard (DSS).
- Elliptic Curve Digital Signature Algorithm (ECDSA).
- Ephemeral Diffie-Hellman (EDH) algorithm.
- Message Digest Algorithm 5 (MD5).
- RSA Signature Algorithm (RSA).
- Secure Hash Algorithm SHA-1, SHA-1 HMAC, SHA1-HMAC-96, SHA-256, SHA-384 and SHA-512. (HMAC stands for Hash Message Authentication Code.)
- Tiger/128, Tiger/160, Tiger/192 and Tiger/192 HMAC.
- Triple Data Encryption Standard (3DES).
1.4 Packages and Documents

Packages

The table below lists the packages that you need in order to use this module, and also optional modules that may interact with this module, depending on your system’s design:

<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hcc_base_docs</td>
<td>This contains the two guides that will help you get started.</td>
</tr>
<tr>
<td>ip_tls</td>
<td>The TLS/DTLS package described in this document.</td>
</tr>
<tr>
<td>mip_base</td>
<td>The IP base package.</td>
</tr>
<tr>
<td>mip_tcp</td>
<td>The TCP package.</td>
</tr>
<tr>
<td>psp_template_base</td>
<td>The base Platform Support Package (PSP).</td>
</tr>
<tr>
<td>psp_template_socket</td>
<td>The PSP Sockets interface.</td>
</tr>
<tr>
<td>enc_base</td>
<td>The Embedded Encryption Manager (EEM) package.</td>
</tr>
<tr>
<td>oal_base</td>
<td>The OS Abstraction Layer (OAL) base package.</td>
</tr>
<tr>
<td>mutil_timer</td>
<td>The MISRA-compliant timer utility.</td>
</tr>
<tr>
<td>eap_tls</td>
<td>The EAP-TLS module that may be used with TLS.</td>
</tr>
</tbody>
</table>

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 Dual Stack System User Guide

This is the core document that describes the complete TCP/IP stack. It 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 that handles all aspects of encryption for TLS/DTLS.

HCC EAP-TLS User Guide

This document describes the EAP-TLS package.
1.5 Change History

This section describes past changes to this manual.

- To download this manual or a PDF describing an earlier software version, see TCP/IP Security PDFs.
- For the history of changes made to the package code itself, see History: ip_tls.

The current version of this manual is 2.10. The full list of versions is as follows:

<table>
<thead>
<tr>
<th>Manual version</th>
<th>Date</th>
<th>Software version</th>
<th>Reason for change</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.10</td>
<td>2018-09-18</td>
<td>3.44</td>
<td>Added functions dtls_get_ip_udp(), dtls_get_ip_socket(), dtls_srv_max_client_socket(), and dtls_srv_clear_configuration(). Added DTLS_SOCKET_MAX_SERVER_CONF configuration option. Changed return value from t_ip_ret to t_tls_ret in several API calls. Removed non-existent functions tls_client_handshake_tcp() and tls_server_handshake_tcp(). Improved some Return Values sections in API calls. Made use of “The DTLS connection handle.” consistent in DTLS Sockets calls. Moved original System Integration diagrams to PSP Porting &gt; Collaboration Diagrams. Simplified the diagrams in System Integration.</td>
</tr>
<tr>
<td>2.00</td>
<td>2018-05-03</td>
<td>3.41</td>
<td>Added function dtls_get_srv_conn_socket(). Added TLS_DTLS_NEW_CONN return code to functions dtls_server_handshake_socket() and dtls_receive_socket().</td>
</tr>
<tr>
<td>Manual version</td>
<td>Date</td>
<td>Software version</td>
<td>Reason for change</td>
</tr>
<tr>
<td>----------------</td>
<td>------------</td>
<td>------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>1.90</td>
<td>2018-03-26</td>
<td>3.40</td>
<td>Added Random Number Generation. Added configuration options TLS_RENEGOTIATION_ENABLE, TLS_REN_UNSEC_ENABLE and TLS_RENCLIENT_DETECT_ENABLE. Added API functions tls_renegotiate_tcp() and tls_renegotiate_socket(). Added tls_server_handshake_socket_ext() and tls_client_handshake_socket_ext().</td>
</tr>
<tr>
<td>1.80</td>
<td>2018-01-30</td>
<td>3.38</td>
<td>Added references to EAP-TLS. Added the RAW interface API used by the EAP-TLS module.</td>
</tr>
<tr>
<td>1.70</td>
<td>2018-01-17</td>
<td>3.37</td>
<td>Added TLS_TRUST_NO_SUBJECT_CERT configuration option.</td>
</tr>
<tr>
<td>1.60</td>
<td>2017-09-18</td>
<td>3.29 - 3.36</td>
<td>Added AEAD files to Source files. Added psp_strnlen() to PSP Porting.</td>
</tr>
<tr>
<td>1.50</td>
<td>2017-08-30</td>
<td>3.28 R4</td>
<td>Updated Packages list.</td>
</tr>
<tr>
<td>1.40</td>
<td>2017-06-14</td>
<td>3.28 R3</td>
<td>New Change History format.</td>
</tr>
<tr>
<td>1.20</td>
<td>2016-04-11</td>
<td>3.17 - 3.25</td>
<td>Added Demo Files and Certificates to Source Files.</td>
</tr>
<tr>
<td>1.10</td>
<td>2016-03-22</td>
<td>3.15 - 3.16</td>
<td>Added Change History.</td>
</tr>
<tr>
<td>1.00</td>
<td>2015-04-17</td>
<td>3.11 - 3.14</td>
<td>First release.</td>
</tr>
</tbody>
</table>
2 Source File List

This section lists and describes all the source code files included in the system. These files follow HCC Embedded's 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.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>config_tls.h</td>
<td>Contains the TLS/DTLS parameters. For details, see config_tls.h.</td>
</tr>
<tr>
<td>config_tls.c</td>
<td>Contains algorithm options and array definitions. For details, see config_tls.c.</td>
</tr>
</tbody>
</table>

There is also a directory named `config_suite_b`, which contains copies of the above two files and also certificate and key files as described below in `config_suite_b`.

### 2.3 Demo Files

The following files in the directory `src/config` contain the example TLS configuration with a demo CA certificate and device certificate. For details, see Running the HCC Demo.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>demo_srv.crt</td>
<td>The HCC Demo security certificate.</td>
</tr>
<tr>
<td>demo_srv.key</td>
<td>The HCC Demo private key file.</td>
</tr>
<tr>
<td>demo2_srv.crt</td>
<td>The second HCC Demo security certificate.</td>
</tr>
<tr>
<td>demo2_srv.key</td>
<td>The second HCC Demo private key file.</td>
</tr>
</tbody>
</table>

There may be more certificates, numbered `demo3_srv.crt`, `demo3_srv.key` and so on.
The following files in the directory `src/config/authcertificates` are the demo certificates. 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.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ca_hccdemo.h</td>
<td>Define the <code>t_tls_certificate</code> variables for the certificate used by the HCC demo project.</td>
</tr>
<tr>
<td>hccdemo_ca.crt</td>
<td>The HCC demo security certificate.</td>
</tr>
<tr>
<td>hccdemo_ca.key</td>
<td>The HCC demo key file.</td>
</tr>
</tbody>
</table>

### 2.4 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.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ca_geotrustglobal.h  and .c</td>
<td>Contain the Geotrust Global CA certificate.</td>
</tr>
<tr>
<td>ca Globalsign2.h  and .c</td>
<td>Contain the GlobalSign Organization Validation CA - G2 certificate.</td>
</tr>
<tr>
<td>ca Verisignclass3.h  and .c</td>
<td>Contain the VeriSign Class 3 Public Primary Certification certificate.</td>
</tr>
</tbody>
</table>

### 2.5 config_suite_b

This directory is in the main `src/config`. It contains copies of the `config_tls.h` and `config_tls.c` files and also certificate and key files as described below.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ecc_sha_ca.crt and ecc_sha_ca.key</td>
<td>The ECC demo security certificate and demo key.</td>
</tr>
<tr>
<td>ecc_sha256_ca.crt and ecc_sha256_ca.key</td>
<td>The ECC demo security certificate and demo key for SHA256.</td>
</tr>
<tr>
<td>ecc_sha384_ca.crt and ecc_sha384_ca.key</td>
<td>The ECC demo security certificate and demo key for SHA384.</td>
</tr>
</tbody>
</table>

### 2.6 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.
2.7 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:

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tls_certificate.c and .h</td>
<td>Source code and header file for certificate handling.</td>
</tr>
<tr>
<td>dtls_common.c and .h</td>
<td>Source code and header file for DTLS common elements.</td>
</tr>
<tr>
<td>tls_common.c and .h</td>
<td>Source code and header file for TLS common elements.</td>
</tr>
</tbody>
</table>

The DTLS files are the following:

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dtls_client.c and .h</td>
<td>Source code and header file for the DTLS client.</td>
</tr>
<tr>
<td>dtls_client_socket.c</td>
<td>Source code for DTLS client socket components.</td>
</tr>
<tr>
<td>dtls_client_udp.c</td>
<td>Source code for client UDP components.</td>
</tr>
<tr>
<td>dtls_server.c and .h</td>
<td>Source code and header file for the DTLS server.</td>
</tr>
<tr>
<td>dtls_server_socket.c</td>
<td>Source code and header file for DTLS server socket components.</td>
</tr>
<tr>
<td>dtls_server_tcp.c</td>
<td>Source code and header file for server TCP components.</td>
</tr>
<tr>
<td>dtls_server_udp.c</td>
<td>Source code and header file for server UDP components.</td>
</tr>
<tr>
<td>dtls_socket.c and .h</td>
<td>Source code and header file for socket components.</td>
</tr>
<tr>
<td>dtls_udp.c and .h</td>
<td>Source code and header file for DTLS UDP components.</td>
</tr>
</tbody>
</table>
The TLS files are the following:

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tls_aead.c and .h</td>
<td>Source code and header file for Authenticated Encryption with Associated Data (AEAD).</td>
</tr>
<tr>
<td>tls_client.c and .h</td>
<td>Source code and header file for the TLS client.</td>
</tr>
<tr>
<td>tls_client_socket.c</td>
<td>Source code for TLS client socket components.</td>
</tr>
<tr>
<td>tls_client_tcp.c</td>
<td>Source code for client TCP components.</td>
</tr>
<tr>
<td>tls_ext.c and .h</td>
<td>Source code and header file for the heartbeat extension.</td>
</tr>
<tr>
<td>tls_raw.c and .h</td>
<td>Source code and header file for RAW interface components.</td>
</tr>
<tr>
<td>tls_server.c and .h</td>
<td>Source code and header file for the TLS server.</td>
</tr>
<tr>
<td>tls_server_socket.c</td>
<td>Source code and header file for TLS server socket components.</td>
</tr>
<tr>
<td>tls_server_tcp.c</td>
<td>Source code and header file for server TCP components.</td>
</tr>
<tr>
<td>tls_socket.c and .h</td>
<td>Source code and header file for socket components.</td>
</tr>
<tr>
<td>tls_tcp.c and .h</td>
<td>Source code and header file for TLS TCP components.</td>
</tr>
</tbody>
</table>
3 Configuration Options

Set the configuration options in the files `config_tls.h` and `config_tls.c`. These are located in the directory `src/config`.

3.1 Specifying TLS or DTLS and the Interface Used

Use the ENABLE options shown in the following table to specify the combination of TLS or DTLS and native TCP/UDP or Sockets to run. These options are from the file `config_tls.h`.

For each interface you want to use, set the relevant option(s) to 1. To use all the interfaces, set all the options shown to 1.

<table>
<thead>
<tr>
<th>Interface</th>
<th>To use the interface, set these options to 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS and Native TCP</td>
<td>TLS_TCPIP_IFC_ENABLE</td>
</tr>
<tr>
<td>TLS and Sockets</td>
<td>TLS_SOCKET_IFC_ENABLE</td>
</tr>
<tr>
<td>DTLS and Native UDP</td>
<td>DTLS_IFC_ENABLE, DTLS_UDP_IFC_ENABLE</td>
</tr>
<tr>
<td>DTLS and Sockets</td>
<td>DTLS_IFC_ENABLE, DTLS_SOCKET_IFC_ENABLE</td>
</tr>
<tr>
<td>TLS RAW</td>
<td>TLS_RAW_IFC_ENABLE</td>
</tr>
</tbody>
</table>

Use the `TLS_USE_STD_SOCKET` option to select either HCC's Sockets implementation or the standard Sockets implementation.

**Note:** The file `config_tls.h` also has ENABLE options that specify the versions of SSL, TLS, and DTLS that the TLS stack accepts. See `config_tls.h`. 
3.2 config_tls.h

Set the following system configuration options in either src/config/config_tls.h or config_suite_b/config_tls.h. The parameters are the same in each file but some defaults differ and these are specified below.

Protocol Support Options

**TLS_PROT_SSL_3_0_ENABLE**
Set this to 1 to enable the TLS stack to accept the SSL 3.0 protocol. The default value is 0.

**TLS_PROT_TLS_1_0_ENABLE**
Set this to 1 to enable the TLS stack to accept the TLS 1.0 protocol. The default value is 0.

**TLS_PROT_TLS_1_1_ENABLE**
Set this to 1 to enable the TLS stack to accept the TLS 1.1 protocol. The default value is 0.

**TLS_PROT_TLS_1_2_ENABLE**
Keep the default value of 1 to enable the TLS stack to accept the TLS 1.2 protocol.

**TLS_PROT_DTLS_1_0_ENABLE**
Set this to 1 to enable the TLS stack to accept the DTLS 1.0 protocol. The default value is 0.

**TLS_PROT_DTLS_1_2_ENABLE**
Keep the default value of 1 to enable the TLS stack to accept the DTLS 1.2 protocol.

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.

In the standard file the default value is 32. In config_suite_b/config_tls.h it is 48.

**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.

In the standard file the default value is 24. In config_suite_b/config_tls.h it is 32.
**TLS_MAX_HMAC_LEN**

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

**TLS_MAX_HBLOCK_LEN**

The maximum length of a hash block. The default value is 64, which is also the minimum.

**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 128.

**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. In the standard file the default value is 3. In `config_suite_b/config_tls.h` it is 1.

**TLS_MAX_IM_CA_CERT**

The maximum number of intermediate host certificates. The default is 0.

**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. In the standard file the default value is 1. In `config_suite_b/config_tls.h` it is 2.

**TLS_CERT_STORE_LOC**

Set this to 1 to copy set certificates into a local buffer. The default is 0.

**TLS_CERT_BUF_DYNALLOC**

Set this to 1 to enable dynamic allocation for certificate local store.
Note: The following four parameters only apply when TLS_CERT_STORE_LOC is set to 1 and TLS_CERT_BUF_DYNALLOC is set to 0.

**TLS_CERT_STORE_SIZE**
The size of the store for single certificate. The default is 512.

**TLS_CERT_STORE_PRIV_SIZE**
The size of the store for a certificate private key. The default is 256.

**TLS_CERT_STORE_CERT_MAX**
The number of certificates that can be stored locally. The default is 2.

**TLS_CERT_STORE_PRIV_MAX**
The number of private keys that can be stored locally. The default is 1.

**TLS_MAX_HASH_ALG_SIZE**
The maximum number of hash algorithms. In the standard file the default value is 3. In `config_suite_b/config_tls.h` it is 4.

**TLS_MAX_BULK_ALG_SIZE**
The maximum number of bulk encryption algorithms. In the standard file the default value is 2. In `config_suite_b/config_tls.h` it is 3.

**TLS_MAX_SIGN_ALG_SIZE**
The maximum number of signature (public key) algorithms. The default value is 5.

**TLS_AEAD_ENABLE**
Keep the default value of 1 to enable AEAD support.

**TLS_MAX_EAED_CNDF_SIZE**
The number of AEAD bulk algorithm configuration entries. The default value is 1.

**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:

\[ TLS_{RP\_HDR\_LEN} + \text{cbc\_size} \times (2 + (\text{TLS\_MAX\_DATA\_LEN} + \text{TLS\_MAX\_HMAC\_LEN}) / \text{cbc\_size}) \]

where \( \text{cbc\_size} \) 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 TLS_RECV_BUF_SIZE + 200. The default value is 3400.

**TLS_REM_DATA_BUF_SIZE**

The size of the buffer used for storing the data that remains to be sent. The default value is 200.

**TLS_CONN_BUF_DYNALLOC**

Set this to 1 to enable connection data buffers to be dynamically allocated. The default value is 0.

**TLS_HS_MSG_STORE_SIZE**

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

**TLS_HS_SALLOC_NR**

The number of static handshake buffers. The default value is 1. (This is only used if TLS_CONN_BUF_DYNALLOC is set.)

**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 16.
**Note:** TLS uses TCP functions to establish the connection with the peer and to send/receive the data. Depending on the setting of the following two parameters, it uses functions directly from the TCP module or the socket interface.

### TLS_SOCKET_IFC_ENABLE

Keep the default value of 1 to enable the Socket interface. Otherwise, set this to 0.

### TLS_TCPIP_IFC_ENABLE

Keep the default value of 1 to enable the native TCP/IP interface. Otherwise, set this to 0.

### TLS_RAW_IFC_ENABLE

Keep the default value of 1 to enable the TLS RAW interface for use with the EAP-TLS module. Otherwise, set this to 0.

### TLS_CHECK_CERT_EXP

Keep the default value of 1 if you want certificate validity to be verified. Otherwise, set this to 0.

### TLS_SOCKET_SELECT_EN

Keep the default value of 1 to enable `tls_select_socket()`.

### TLS_SOCKET_POLL_EN

Keep the default value of 1 to enable `tls_poll_socket()`.
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. If DTLS_IFC_ENABLE is set the default value is 1.

**DTLS_SOCKET_IFC_ENABLE**
Set this to 1 to enable the Socket interface for DTLS. If DTLS_IFC_ENABLE is set the default value is 1.

**DTLS_SOCKET_MAX_SERVER_CONF**
The maximum number of DTLS server configurations with connection limits. 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 1400.

**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 re-transmit time initial value in units of 100 ms. The default value is 10.

**DTLS_RETRANSMIT_MAX_TIME**
The maximum DTLS handshake re-transmit time in units of 100 ms. The default value is 600.
Further TLS Options

**TLS_TCP_CONN_STACK_SIZE**
The TCP connection stack size. 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_ENABLE**
Keep the default value of 1 if you want to enable the heartbeat extension. To disable this, set it to 0.

**TLS_HB_TIME**
The heartbeat resend time in seconds. The default value is 10.

**TLS_HB_ALLOW_PEER_REQ**
Keep the default of 1 to allow the peer to send heartbeat requests. To disable this, set it to 0.

**TLS_RENEGOTIATION_ENABLE**
Keep the default of 1 to enable TLS handshake renegotiation. To disable this, set it to 0.

**TLS_REN_UNSEC_ENABLE**
Set this to 1 to enable TLS insecure renegotiation. The default value is 0.

**TLS_REN_CLIENT_DETECT_ENABLE**
Keep the default of 1 to enable detection of client renegotiation support. To disable this, set it to 0.

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

**TLS_EXT_SIGN_ALG_ENABLE**
Keep this at the default of 1 to enable the signature Algorithms extension.

**TLS_SIGN_HASH_ALG_TBL_SIZE**
The number of entries in the signature/hash algorithms table. The default value is 2.

**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.

**TLS_TRUST_NO_SUBJECT_CERT**
Set this to 1 to trust certificates that have no subject when the peer name is set to null. The default value is 0.

**TLS_USE_STD_SOCKET**
Keep the default value of 0 to use HCC’s Socket implementation. Set it to 1 to use the standard Socket implementation.

**TLS_SOCKET_IP_V6_ENABLE**
Keep the default value of 1 to enable use of IPv6 for Socket connections. Otherwise set it to 0.

**TLS_RSA_ENABLE**
Keep the default value of 1 to enable RSA signature support. To disable this, set it to 0.

**TLS_EDH_ENABLE**
Keep the default value of 1 to enable the EDH keying mechanism. To disable this, set it to 0.

**TLS_DSA_ENABLE**
Keep the default value of 1 to enable DSA signature support. To disable this, set it to 0.

**Note:** Defines that enable hash algorithms are used also in the code. Other algorithms can also be added if the cipher suite uses them for MAC authorization.

**TLS_MD5_ENABLE**
Keep the default value of 1 to enable MD5 hash (needed by DSS, SSL3.0, TLS1.0, TLS 1.1, and DTLS).

**TLS_SHA1_ENABLE**
Keep the default value of 1 to enable SHA1 hash (needed by SSL3.0, TLS1.0, TLS 1.1, and DTLS).

**TLS_SHA256_ENABLE**
Keep the default value of 1 to enable SHA256 hash (needed by TLS1.2 if SHA384/SHA512 are disabled).

**TLS_SHA384_ENABLE**
Set this to 1 to enable SHA384 hash. The default value is 0. In `config_suite_b/config_tls.h` it is 1.

**TLS_SHA512_ENABLE**
Set this to 1 to enable SHA512 hash. The default value is 0.
Note: The following five algorithm enabling defines are used only in this file. Other algorithms can also be added to the configuration.

**TLS_TDES_ENABLE**
Keep the default value of 1 to enable TDES support. To disable this, set it to 0. In `config_suite_b/config_tls.h` it is 1.

**TLS_AES_ENABLE**
Keep the default value of 1 to enable AES CBC support. To disable this, set it to 0. In `config_suite_b/config_tls.h` it is 1.

**TLS_AES_GCM_ENABLE**
Keep the default value of 1 to enable AES GCM support.

**TLS_AES_CCM_ENABLE**
Set this to 1 to enable AES CCM support. The default is 0.

**TLS_AES_CCM_8_ENABLE**
Set this to 1 to enable AES CCM_8 support. The default is 0.

**Elliptic Curve cryptography configuration**

**TLS_ECC_ENABLE**
Keep the default value of 1 to enable Elliptic Curve Cryptography (ECC) support. To disable this, set it to 0.

**TLS_ECDH_ENABLE**
Keep the default value of 1 to enable the ECC keying mechanism. To disable this, set it to 0.

**TLS_ECDSA_ENABLE**
Keep the default value of 1 to enable ECC signature support. To disable this, set it to 0.

**TLS_ECC_CURVE_COUNT**
The number of ECC curves that are supported by TLS. In the standard file the default value is 1. In `config_suite_b/config_tls.h` it is 2.

**TLS_ECC_CURVE_OID_MAX_SIZE**
The maximum size of the Named OID. The default value is 8.
The following three options define the conversion table of `t_tls_hash_algorithm_index` to server key exchange signature id (RFC 5246 Appendix-A.4.2):

**TLS_HASH_IDX_TO_ALG_TBL**

The default value is:

\[
\{ (\text{uint8}_t)\text{TLS\_HASH\_ALG\_SHA1}, (\text{uint8}_t)\text{TLS\_HASH\_ALG\_SHA256}, (\text{uint8}_t)\text{TLS\_HASH\_ALG\_MD5} \}
\]

**TLS_ALG_TBL_TO_HASH_IDX**

The default value is:

\[
\{ (\text{uint8}_t)\text{TLS\_LST\_HASH\_IDX}, (\text{uint8}_t)\text{TLS\_MD5\_IDX}, (\text{uint8}_t)\text{TLS\_SHA1\_IDX}, (\text{uint8}_t)\text{TLS\_LST\_HASH\_IDX}, (\text{uint8}_t)\text{TLS\_SHA256\_IDX} \}
\]

**TLS_ALG_TBL_TO_HASH_IDX_SIZE**

The default value is 5.

The following two options define the conversion table of `t_tls_sign_algorithm_index` to server key exchange signature id (RFC 5246, Appendix-A.4.2):

**TLS_SIGN_IDX_TO_ALG_TBL**

The default value is:

\[
\{ \text{TLS\_RSA\_SID\_TO\_ALG} \text{ TLS\_EDH\_SID\_TO\_ALG} \text{ TLS\_DSA\_SID\_TO\_ALG} \\
\text{ TLS\_ECDH\_SID\_TO\_ALG} \text{ TLS\_ECDSA\_SID\_TO\_ALG} \}
\]

**TLS_SIGN_IDX_TO_ALG_TBL_SIZE**

The default value is:

\[
( \text{TLS\_RSA\_SID\_TO\_ALG\_NR} + \text{TLS\_EDH\_SID\_TO\_ALG\_NR} + \text{TLS\_DSA\_SID\_TO\_ALG\_NR} + \\
\text{TLS\_ECDH\_SID\_TO\_ALG\_NR} + \text{TLS\_ECDSA\_SID\_TO\_ALG\_NR} )
\]
The following two options define the conversion table of `t_tls_sign_algorithm_index` to certificate type:

**TLS_SIGN_IDX_TO_CTYPE_TBL**

The default value is:

```
{ TLS_RSA_SID_TO_CTYPE TLS_EDH_SID_TO_CTYPE TLS_DSA_SID_TO_CTYPE
  TLS_ECDH_SID_TO_CTYPE TLS_ECDSA_SID_TO_CTYPE }
```

**TLS_SIGN_IDX_TO_CTYPE_TBL_SIZE**

The default value is:

```
( TLS_RSA_SID_TO_ALG_NR + TLS_EDH_SID_TO_ALG_NR + TLS_DSA_SID_TO_ALG_NR +
  TLS_ECDH_SID_TO_ALG_NR + TLS_ECDSA_SID_TO_ALG_NR )
```

**TLS_ALG_TBL_TO_SIGN_IDX**

The default value is:

```
{ (uint8_t)TLS_INV_IDX, (uint8_t)TLS_RSA_IDX, (uint8_t)TLS_DSA_IDX,
  (uint8_t)TLS_ECDSA_IDX}
```

**TLS_ALG_TBL_TO_SIGN_IDX_SIZE**

The default value is 4.

**TLS_RSA_PADDING_FF**

Keep the default value to set the initialization vector to make RSA sign data with the padding "FF". The default value is RSA_PADDING_SET.

### 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 Running the HCC Demo

An example TLS configuration is provided in src/config/authcertificates. This comprises:

- a demo CA certificate - the file hccdemo_ca.crt.
- a demo key - the file hccdemo_ca.key.

The certificate's peer name is 'hcc demo'.

To install the certificate:

1. Go to the directory config/authcertificates and double-click the file hccdemo_ca.crt. The Certificate dialog shown below opens:
2. Click **Install Certificate...** to open the **Certificate Import Wizard** shown below.

3. Under **Store Location** select **Current User** and click **Next**.

4. On the **Certificate Store** page, keep Automatic selection as shown below and click **Next**:
5. The **Completion** page appears as shown below:

6. Check the settings and if they are correct click **Finish**.

7. If the import succeeds, the following dialog appears. The certificate is now installed successfully and you can proceed with the demo. Click **OK** to close the dialog.
5 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.

5.1 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.
5.2 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 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.
Handshake already complete

Invalid MAC case
Alert(bad_record_mac)

Break

Invalid MAC case
Alert(bad_record_mac)

Break

Alert (close_notify)
5.3 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 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.
5.4 The Embedded Encryption Manager

The module uses HCC’s Embedded Encryption Manager (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's operation and its Application Programming Interface (API), see the HCC Embedded Encryption Manager User 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 the following:

- Message Digest Algorithm 5 (MD5).
- Tiger/128, Tiger/160, Tiger/192 and Tiger/192 HMAC.

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).
Elliptic Curve Cryptography

To use Elliptic Curve Cryptography (ECC), you must configure the supported curves. To do this, add entries to the `g_curve_name_id_array` table. Each entry in the table is a structure of type `t_tls_ecc_curve_dsc`.

The maximum number of curves is configured by `TLS_ECC_CURVE_COUNT`. To configure a curve, you must specify its TLS identifier and OID number.

**Note:** The ECC implementation supports only primary field curves. To use binary field elliptic curves, you must provide a library that supports these.

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, two signature algorithms, and four bulk encryption algorithms, theoretically 24 (3*2*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.

<table>
<thead>
<tr>
<th>Hash</th>
<th>Signature</th>
<th>Bulk encryption</th>
<th>Defined in the RFCs (and api_tls.h) as:</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA</td>
<td>RSA</td>
<td>3DESEDECBC</td>
<td>TLS_RSA_WITH_3DES_EDE_CBC_SHA</td>
<td>0x000A</td>
</tr>
<tr>
<td>SHA</td>
<td>RSA</td>
<td>AES128CBC</td>
<td>TLS_RSA_WITH_AES_128_CBC_SHA</td>
<td>0x002F</td>
</tr>
<tr>
<td>SHA</td>
<td>RSA</td>
<td>3DESEDECBC</td>
<td>TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA</td>
<td>0x0013</td>
</tr>
</tbody>
</table>
6 Application Programming Interface

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

6.1 Module Management

The functions are the following:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tls_init()</td>
<td>Initializes the module and allocates the required resources.</td>
</tr>
<tr>
<td>tls_start()</td>
<td>Starts the module.</td>
</tr>
<tr>
<td>tls_stop()</td>
<td>Stops the module.</td>
</tr>
<tr>
<td>tls_delete()</td>
<td>Deletes the module and releases the resources it used.</td>
</tr>
</tbody>
</table>

**Note:** You must start the EEM and call any Algorithm Management 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:** Do not call other TLS/DTLS functions before **tls_init()**.

**Format**

```
t_tls_ret tls_init (void)
```

**Arguments**

**Argument**

None.

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_INIT_ERR</td>
<td>Initialization failed.</td>
</tr>
<tr>
<td>Else</td>
<td>See Error Codes.</td>
</tr>
</tbody>
</table>
tls_start

Use this function to start the module.

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

Note:

- You must call `tls_init()` before this to initialize the module.
- You must start the EEM before calling this function.
- 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

<table>
<thead>
<tr>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>None.</td>
</tr>
</tbody>
</table>

Return values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_INIT_ERR</td>
<td>Operation failed.</td>
</tr>
</tbody>
</table>
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**

```c
void tls_stop();
```

**Arguments**

None.

**Return values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_INIT_ERR</td>
<td>The TLS connection did not stop.</td>
</tr>
</tbody>
</table>
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**

```c
void t_tls_ret tls_delete ( void )
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>None.</td>
</tr>
</tbody>
</table>

**Return values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>Else</td>
<td>See Error Codes.</td>
</tr>
</tbody>
</table>
6.2 Algorithm Management

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

The functions are the following:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tls_register_aead()</code></td>
<td>Registers an AEAD algorithm with the EEM.</td>
</tr>
<tr>
<td><code>tls_register_bulk()</code></td>
<td>Registers a bulk encryption algorithm with the EEM.</td>
</tr>
<tr>
<td><code>tls_register_hash()</code></td>
<td>Registers a hash encryption algorithm with the EEM.</td>
</tr>
<tr>
<td><code>tls_register_sign()</code></td>
<td>Registers a signature encryption algorithm with the EEM.</td>
</tr>
<tr>
<td><code>tls_register_sign_alg()</code></td>
<td>Configures a signature/hash table that is used to generate a signature/algorithms extension in hello messages and the signature list in a certificate request.</td>
</tr>
<tr>
<td><code>tls_add_certificate()</code></td>
<td>Adds a new certificate to an appropriate array of certificates.</td>
</tr>
<tr>
<td><code>tls_del_certificate()</code></td>
<td>Deletes a certificate from an array of certificates.</td>
</tr>
<tr>
<td><code>tls_add_cipher_suite()</code></td>
<td>Adds a new cipher suite.</td>
</tr>
</tbody>
</table>
tls_register_aead

Use this function to register an Authenticated Encryption with Associated Data (AEAD) algorithm with the EEM.

For a description of `record_iv_len` and `fixed_iv_len`, see RFC 5246. Proper values for AES GCM can be found in RFC 5288.

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

**Format**

```c
#define t_tls_ret tls_register_aead (  
    t_tls_bulk_algorithm_index   idx,  
    uint8_t                      fixed_iv_len )  
```

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>idx</td>
<td>The index of the signature algorithm.</td>
<td>t_tls_bulk_algorithm_index</td>
</tr>
<tr>
<td>fixed_iv_len</td>
<td>The fixed Init vector length of the AEAD algorithm.</td>
<td>uint8_t</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>Parameter <code>idx</code> is invalid.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>No free entry was found.</td>
</tr>
</tbody>
</table>
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**

```c
#define t_tls_ret t_tls_register_bulk ( 
    t_tls_bulk_algorithm_index idx, 
    t_enc_ifc_hdl ifc_hdl, 
    uint16_t alg_id ) 
```

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>idx</td>
<td>The index of the bulk algorithm.</td>
<td>t_tls_bulk_algorithm_index</td>
</tr>
<tr>
<td>ifc_hdl</td>
<td>The encryption handle of the bulk algorithm.</td>
<td>t_enc_ifc_hdl</td>
</tr>
<tr>
<td>alg_id</td>
<td>The bulk algorithm identifier.</td>
<td>uint16_t</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>Parameter <code>idx</code> is invalid.</td>
</tr>
</tbody>
</table>
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**

```c
void tls_register_hash(
    t_tls_hash_algorithm_index idx,
    t_enc_ifc_hdl ifc_hdl,
    uint16_t alg_id
)
```

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>idx</td>
<td>The index of the hash algorithm.</td>
<td>t_tls_hash_algorithm_index</td>
</tr>
<tr>
<td>ifc_hdl</td>
<td>The encryption handle of the hash algorithm.</td>
<td>t_enc_ifc_hdl</td>
</tr>
<tr>
<td>alg_id</td>
<td>The hash algorithm identifier.</td>
<td>uint16_t</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>Parameter <code>idx</code> is invalid.</td>
</tr>
</tbody>
</table>
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**

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

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>idx</td>
<td>The index of the signature algorithm.</td>
<td>t_tls_sign_algorithm_index</td>
</tr>
<tr>
<td>ifc_hdl</td>
<td>The encryption handle of the signature algorithm.</td>
<td>t_enc_ifc_hdl</td>
</tr>
<tr>
<td>alg_id</td>
<td>The signature algorithm identifier.</td>
<td>uint16_t</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>Parameter <code>idx</code> is invalid.</td>
</tr>
</tbody>
</table>
tls_register_sign_alg

Use this function to configure a signature/hash table that is used to generate a signature_algorithms extension in hello messages and the signature list in a certificate request.

The first entry in the table contains hash algorithm index, the second the signature algorithm index.

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

**Format**

```c
#include <tls.h>

uint8_t t_tls_ret tls_register_sign_alg ( uint8_t p_sign[][2U], uint8_t cnt );
```

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_sign[]</td>
<td>A pointer to the table containing signature/hash algorithms.</td>
<td>uint8_t</td>
</tr>
<tr>
<td>cnt</td>
<td>The number of entries in the table.</td>
<td>uint8_t</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>Parameter idx is invalid.</td>
</tr>
</tbody>
</table>
tls_add_certificate

Use this function to add a new certificate to an appropriate array of certificates.

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

**Format**

```
t_tls_ret tls_add_certificate ( 
    const t_tls_certificate * const p_cert, 
    uint8_t type, 
    uint8_t * p_idx 
)
```

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_cipher_suite</td>
<td>A pointer to the certificate.</td>
<td>t_tls_cipher_suite *</td>
</tr>
<tr>
<td>type</td>
<td>The certificate type.</td>
<td>uint8_t</td>
</tr>
<tr>
<td></td>
<td>If this is neither TLS_CERT_HOST nor NUM_CA_CERT, a TLS_TYPE_ERR is returned.</td>
<td></td>
</tr>
<tr>
<td>p_idx</td>
<td>On return, a pointer to the index of the added certificate in the array.</td>
<td>uint8_t *</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>The parameter is invalid.</td>
</tr>
<tr>
<td>TLS_TYPE_ERR</td>
<td>Incorrect type value.</td>
</tr>
<tr>
<td>TLS_FULL_ERR</td>
<td>No free space is available for the certificate in the array.</td>
</tr>
</tbody>
</table>
**tls_del_certificate**

Use this function to delete a certificate from an array of certificates.

**Format**

```c
void tls_del_certificate ( uint8_t type, uint8_t idx );
```

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>The certificate type.</td>
<td>uint8_t</td>
</tr>
<tr>
<td>idx</td>
<td>The index of the certificate in the array.</td>
<td>uint8_t *</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>The parameter is invalid.</td>
</tr>
</tbody>
</table>
tls_add_cipher_suite

Use this function to add a new cipher suite.

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

**Format**

```c
t_tls_ret tls_add_cipher_suite ( const t_tls_cipher_suite * const p_cipher_suite)
```

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_cipher_suite</td>
<td>A pointer to the cipher suite structure.</td>
<td>t_tls_cipher_suite *</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>The parameter is invalid.</td>
</tr>
<tr>
<td>TLS_NULL_PTR_ERR</td>
<td>The encryption/hash function was not specified.</td>
</tr>
<tr>
<td>TLS_FULL_ERR</td>
<td>No free slot is available for the cipher suite.</td>
</tr>
</tbody>
</table>
6.3 Certificate Management Functions

The certificate management functions implement an easy way to take certificates from the external store and add an external certificate validation mechanism:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tls_add_cipher_suite_ex()</td>
<td>Adds a new cipher suite and assigns it a host certificate.</td>
</tr>
<tr>
<td>tls_set_host_ca_chain()</td>
<td>Sets a host certification chain.</td>
</tr>
<tr>
<td>tls_register_cert_store_cb()</td>
<td>Registers a callback that can be used to get device certificates from external store.</td>
</tr>
<tr>
<td>tls_register_cert_validate_cb()</td>
<td>Registers a callback that can be used to validate a received certificate chain.</td>
</tr>
<tr>
<td>tls_register_get_ca_chain_cb()</td>
<td>Registers a callback that can be used to get a host authorization chain.</td>
</tr>
</tbody>
</table>
tls_set_host_ca_chain

Use this function to set a host certification chain.

This host certificate chain is sent with the host certificate to authorize it. A chain sent by TLS has this format:

host_cert, p_cert_list[0] .. p_cert_list[n]

Format

```c
#include "t_tls.h"

int8_t tls_set_host_ca_chain (  
    t_tls_certificate ** pp_cert_list,  
    uint8_t cert_cnt )
```

Arguments

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>pp_cert_list</td>
<td>An array of pointers to certificates.</td>
<td>t_tls_certificate **</td>
</tr>
<tr>
<td>cert_cnt</td>
<td>The number of certificates in the chain.</td>
<td>uint8_t</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>A parameter is invalid.</td>
</tr>
</tbody>
</table>
tls_add_cipher_suite_ex

Use this function to add a new cipher suite and assign it a host certificate.

Format

```c
int tls_add_cipher_suite_ex(
    const t_tls_cipher_suite * p_cipher_suite,
    uint8_t cert_idx)
```

Arguments

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_cipher_suite</td>
<td>A pointer to the cipher suite structure.</td>
<td>t_tls_cipher_suite *</td>
</tr>
<tr>
<td>cert_idx</td>
<td>The index of the host certificate that is assigned to this cipher suite.</td>
<td>uint8_t</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>A parameter is invalid.</td>
</tr>
<tr>
<td>TLS_NULL_PTR_ERR</td>
<td>No encryption/hash function was specified.</td>
</tr>
<tr>
<td>TLS_FULL_ERR</td>
<td>There is no free slot for the cipher suite.</td>
</tr>
</tbody>
</table>
tls_register_cert_store_cb

Use this function to register a t_tls_cert_get_cb() callback that can be called to get device certificates from an external store.

If such a callback is not registered, TLS gets certificates from its own certificate store.

Format

```
t_tls_ret tls_register_cert_store_cb( t_tls_cert_get_cb p_cb )
```

Arguments

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_cb</td>
<td>A pointer to the callback function.</td>
<td>t_tls_cert_get_cb</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>A parameter is invalid.</td>
</tr>
</tbody>
</table>
**tls_register_cert_validate_cb**

Use this function to register a `t_tls_cert_valid_cb()` callback function that can be called to validate a received certificate chain.

If no such callback is defined, TLS validates certificates using its own mechanism.

**Format**

```
t_tls_ret tls_register_cert_validate_cb ( t_tls_cert_valid_cb p_cb )
```

**Arguments**

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>p_cb</code></td>
<td>A pointer to the callback function.</td>
<td><code>t_tls_cert_valid_cb</code></td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>A parameter is invalid.</td>
</tr>
</tbody>
</table>
tls_register_get_ca_chain_cb

Use this function to register a `t_tls_cert_get_ca_chain_cb()` callback that can be called to get a host authorization chain.

This type of chain includes intermediate certificates and CA certificates that were used to sign the host certificate.

**Format**

```
t_tls_ret tls_register_get_ca_chain_cb ( t_tls_cert_get_ca_chain_cb p_cb )
```

**Arguments**

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_cb</td>
<td>A pointer to the callback function.</td>
<td><code>t_tls_cert_get_ca_chain_cb</code></td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>A parameter is invalid.</td>
</tr>
</tbody>
</table>
6.4 Certificate Management Callbacks

These callback functions implement an easy way to take certificates from the external store and add an external certificate validation mechanism:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_tls_cert_get_cb()</td>
<td>Gets a device certificate.</td>
</tr>
<tr>
<td>t_tls_cert_get_ca_chain_cb()</td>
<td>Gets a device certificate chain.</td>
</tr>
<tr>
<td>t_tls_cert_valid_cb()</td>
<td>Validates a certificate chain.</td>
</tr>
</tbody>
</table>
t_tls_cert_get_cb

Use this function to get a device certificate.

The certificate should match the given signature type and hash type.

Format

```c
typedef t_tls_ret (*t_tls_cert_get_cb)(
    t_tls_hash_algorithm hash,
    t_tls_alg_type sign,
    const t_tls_certificate ** pp_cert )
```

Arguments

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash</td>
<td>The hash algorithm.</td>
<td>t_tls_hash_algorithm</td>
</tr>
<tr>
<td>sign</td>
<td>The signature algorithm.</td>
<td>t_tls_alg_type</td>
</tr>
<tr>
<td>pp_cert</td>
<td>A pointer to the variable that receives the device certificate from the certificate store.</td>
<td>t_tls_certificate **</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_CERTIFICATE_ERR</td>
<td>Operation failed; no matching certificate was found.</td>
</tr>
</tbody>
</table>
t_tls_cert_get_ca_chain_cb

Use this function to get a device certificate chain.

Pass the certificate chain in a single buffer. Every certificate has a 3 byte header containing the certificate length (big-endian) as shown in the following example:

```
| length 1 | cert. 1 data | ... | length 2 | cert. 2 data | ... | length n | cert. n data |
| XX XX XX | XX .. XX     | ... | XX XX XX | XX .. XX     | ... | XX XX XX | XX .. XX ... |
```

**Note:** The certificate length does not include the three byte header.

**Format**

```c
typedef t_tls_ret (*t_tls_cert_get_ca_chain_cb)(
    const uint8_t **   pp_cert,
    uint16_t *         p_cert_length )
```

**Arguments**

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>pp_cert</td>
<td>On return, a pointer to the variable holding the address of the certificate list.</td>
<td>uint8_t **</td>
</tr>
<tr>
<td>p_cert_length</td>
<td>On return, the number of certificates in the chain.</td>
<td>uint16_t *</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_CERTIFICATE_ERR</td>
<td>Operation failed; no matching certificate was found.</td>
</tr>
</tbody>
</table>
t_tls_cert_valid_cb

Use this function to validate a certificate chain received from a peer.

Certificates are stored in the buffer. Each certificate is preceded by a 3 byte header containing the certificate length (big-endian) as shown in the following example:

```
| length 1 | cert. 1 data | ... | length 2 | cert. 2 data | ...| length n | cert. n data |
| XX XX XX | XX .. XX     | ... | XX XX XX | XX .. XX     |...| XX XX XX | XX .. XX ... |
```

**Note:** The certificate length does not include the three byte header.

**Format**

```c
typedef t_tls_ret t_tls_cert_valid_cb (  
    const uint8_t    p_cert[],  
    uint16_t         cert_length )
```

**Arguments**

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_cert[]</td>
<td>A pointer to a buffer that contains a certificate chain. Each certificate consists of a three byte (big endian) length and certificate data.</td>
<td>uint8_t</td>
</tr>
<tr>
<td>cert_length</td>
<td>The length (in bytes) of the whole buffer.</td>
<td>uint16_t</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>The certificate chain is valid.</td>
</tr>
<tr>
<td>TLS_ALERT_CERTIFICATE_REVOKED</td>
<td>One of the certificates has been revoked.</td>
</tr>
<tr>
<td>TLS_ALERT_CERTIFICATE_EXPIRED</td>
<td>One of the certificates has expired.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>A parameter is invalid.</td>
</tr>
</tbody>
</table>
6.5 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.

The functions are the following:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tls_start_tcp()</code></td>
<td>Called from the client or server, starts the TLS handshake mechanism.</td>
</tr>
<tr>
<td><code>tls_close_tcp()</code></td>
<td>Called from the client or server, closes the TLS connection.</td>
</tr>
<tr>
<td><code>tls_renegotiate_tcp()</code></td>
<td>Starts TLS handshake renegotiation.</td>
</tr>
<tr>
<td><code>tls_tcp_accept()</code></td>
<td>Called from the server, accepts a connection from a remote node on a port previously opened with <code>tcp_open()</code>.</td>
</tr>
<tr>
<td><code>tls_tcp_connect()</code></td>
<td>Called from the client, initiates a connection to a remote port using TLS.</td>
</tr>
<tr>
<td><code>tls_send_tcp()</code></td>
<td>Sends data to the peer over a connection.</td>
</tr>
<tr>
<td><code>tls_receive_tcp()</code></td>
<td>Receives data (a PDU) and decodes it.</td>
</tr>
<tr>
<td><code>tls_rx_ready_tcp()</code></td>
<td>Checks whether there is any unread data on a connection.</td>
</tr>
<tr>
<td><code>tls_get_buffer_tcp()</code></td>
<td>Allocates a buffer big enough for the protocol PDU.</td>
</tr>
<tr>
<td><code>tls_get_ticket_tcp()</code></td>
<td>Retrieves a ticket that can be used to resume a previously established TCP connection.</td>
</tr>
<tr>
<td><code>tls_get_state_tcp()</code></td>
<td>Gets the status of a TLS connection.</td>
</tr>
</tbody>
</table>
tls_start_tcp

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

Format

```c
#define t_tls_ret ts_start_tcp ( t_tcp_conn_hdl conn_hdl )
```

Arguments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn_hdl</td>
<td>The handle of the previously established TCP connection.</td>
<td>t_tcp_conn_hdl</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>The connection was not found.</td>
</tr>
</tbody>
</table>
tls_close_tcp

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

**Format**

```c
*t_tls_ret tls_close_tcp ( t_tcp_conn_hdl conn_hdl )
```

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn_hdl</td>
<td>The TCP connection handle.</td>
<td>t_tcp_conn_hdl</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>The connection was not found.</td>
</tr>
</tbody>
</table>
tls_renegotiate_tcp

Call this function to start TLS handshake renegotiation.

**Note:** This is only available if configuration option `TLS_RENEGOTIATION_ENABLE` is set.

**Format**

```c
void t_tls_ret tls_renegotiate_tcp (  
    const t_tcp_conn_hdl tcp_conn_hdl,  
    uint8_t b_verify )
```

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcp_conn_hdl</td>
<td>The handle of the previously established TCP connection.</td>
<td>t_tcp_conn_hdl</td>
</tr>
<tr>
<td>b_verify</td>
<td>A flag indicating whether the client must be verified.</td>
<td>uint8_t</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>A parameter is invalid.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>The connection was not found.</td>
</tr>
<tr>
<td>TLS_WAIT</td>
<td>Connection not open yet; wait for handshake to end.</td>
</tr>
<tr>
<td>TLS_HS_ERR</td>
<td>Not allowed to renegotiate.</td>
</tr>
<tr>
<td>TLS_MEM_ERR</td>
<td>Not enough memory to allocate handshake buffer.</td>
</tr>
</tbody>
</table>
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

```c
#define t_tls_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

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

Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_WAIT</td>
<td>The connection was not accepted.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>No free TLS connection was found.</td>
</tr>
</tbody>
</table>
tlsTcpConnect

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**

```c
void tlsTcpConnect (
const t_ip_port * p_ip_port,
const t_tls_conn_inf * p_inf,
t_tls_ticket                   conn_ticket,
t_tcp_conn_hdl * p_tcp_conn_hdl )
```

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_ip_port</td>
<td>A pointer to the address and port number of the server to connect to.</td>
<td>t_ip_port *</td>
</tr>
<tr>
<td>p_inf</td>
<td>A pointer to the structure containing connection parameters.</td>
<td>t_tls_conn_inf *</td>
</tr>
<tr>
<td>conn_ticket</td>
<td>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.</td>
<td>t_tls_ticket</td>
</tr>
<tr>
<td>p_tcp_conn_hdl</td>
<td>Where to write the connection handle.</td>
<td>t_tcp_conn_hdl *</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_WAIT</td>
<td>The connection was not opened.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>The p_tci_peer_name in p_inf was not specified.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>No free TLS connection was found.</td>
</tr>
</tbody>
</table>
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

```c
#include <tls.h>

uint32_t tls_send_tcp ( const t_tcp_conn_hdl conn_hdl, uint8_t * p_data, uint16_t data_len );
```

Arguments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn_hdl</td>
<td>The TCP connection handle.</td>
<td>t_tcp_conn_hdl</td>
</tr>
<tr>
<td>p_data</td>
<td>A pointer to the IP buffer which contains the data to send. Allocate this by using <code>tls_get_buffer_tcp()</code>.</td>
<td>uint8_t *</td>
</tr>
<tr>
<td>data_len</td>
<td>The length of the data in bytes.</td>
<td>uint16_t</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>Else</td>
<td>See Error Codes.</td>
</tr>
</tbody>
</table>
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**

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

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn_hdl</td>
<td>The TCP connection handle.</td>
<td>t_tcp_conn_hndl</td>
</tr>
<tr>
<td>pp_buf</td>
<td>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 <code>tcp_release_buf()</code>.</td>
<td>uint8_t **</td>
</tr>
<tr>
<td>p_data_len</td>
<td>On return, a pointer to the length of the decoded PDU.</td>
<td>uint16_t *</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_WAIT</td>
<td>There is no data to receive.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>A parameter was invalid.</td>
</tr>
<tr>
<td>TLS_IO_ERR</td>
<td>I/O operation failed.</td>
</tr>
</tbody>
</table>
**tls_rx_ready_tcp**

Use this function to check whether there is any unread data on a connection.

The function first checks the TLS stack for unread data, then the TCP stack.

**Format**

```c
void tls_rx_ready_tcp ( const t_tcp_conn_hdl tcp_conn_hdl, uint32_t * p_len );
```

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcp_conn_hdl</td>
<td>The TCP connection handle.</td>
<td>t_tcp_conn_hdl</td>
</tr>
<tr>
<td>p_len</td>
<td>On return, the length of data available to be read.</td>
<td>uint32_t *</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP_SUCCESS</td>
<td>There is unread data.</td>
</tr>
<tr>
<td>Else</td>
<td>See Error Codes.</td>
</tr>
</tbody>
</table>
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**

```c
void 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**

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

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_WAIT</td>
<td>The function was called before the handshake was finished.</td>
</tr>
<tr>
<td>TLS_MEM_ERR</td>
<td>No buffer of requested size available.</td>
</tr>
</tbody>
</table>
tls_get_ticket_tcp

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

Format

```c
#include <t_secure.h>

#include <t_tls_ticket.h>

#include <t_tcp_conn.h>

int tls_get_ticket_tcp (const t_tcp_conn_hdl conn_hdl, t_tls_ticket *p_conn_ticket);
```

Arguments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn_hdl</td>
<td>The handle of the TCP connection.</td>
<td>t_tcp_conn_hdl</td>
</tr>
<tr>
<td>p_conn_ticket</td>
<td>On return, a pointer to the variable which received the ticket handle.</td>
<td>t_tls_ticket *</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>A parameter is incorrect.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>The specified connection was not found.</td>
</tr>
</tbody>
</table>

tls_get_state_tcp

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

Format

```c
typedef t_tls_conn_status tls_get_state_tcp ( const t_tcp_conn_hdl conn_hdl );
```

Arguments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn_hdl</td>
<td>The TCP connection handle.</td>
<td>t_tcp_conn_hdl</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_CONNST_CLOSED</td>
<td>The connection is closed or does not exist.</td>
</tr>
<tr>
<td>TLS_CONNST_HANDSHAKE</td>
<td>The connection is in handshake state.</td>
</tr>
<tr>
<td>TLS_CONNST_OPERATING</td>
<td>The connection is established and the user can send and receive data.</td>
</tr>
</tbody>
</table>
6.6 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.

The functions are the following:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dtls_start_udp()</td>
<td>Called from the client or server, starts the DTLS handshake mechanism.</td>
</tr>
<tr>
<td>dtls_close_udp()</td>
<td>Called from the client or server, closes the DTLS connection.</td>
</tr>
<tr>
<td>dtls_udp_srv_open()</td>
<td>Starts the DTLS server.</td>
</tr>
<tr>
<td>dtls_udp_srv_close()</td>
<td>Called from the client or server, closes the DTLS server.</td>
</tr>
<tr>
<td>dtls_get_srv_conn_udp()</td>
<td>Gets the last established DTLS connection.</td>
</tr>
<tr>
<td>dtls_connect_udp()</td>
<td>Called from the client, initiates a connection to the server.</td>
</tr>
<tr>
<td>dtls_send_udp()</td>
<td>Sends data to the peer over a DTLS connection.</td>
</tr>
<tr>
<td>dtls_receive_udp()</td>
<td>Called from the client or server, receives the decoded data.</td>
</tr>
<tr>
<td>dtls_get_buffer_udp()</td>
<td>Allocates a buffer big enough for the protocol PDU.</td>
</tr>
<tr>
<td>dtls_get_ticket_udp()</td>
<td>Retrieves a ticket that can be used to resume a previously established DTLS connection.</td>
</tr>
<tr>
<td>dtls_get_state_udp()</td>
<td>Gets the status of a DTLS connection.</td>
</tr>
<tr>
<td>dtls_get_ip_udp()</td>
<td>Gets the IP address and port information of a DTLS connection's peer.</td>
</tr>
</tbody>
</table>
dtls_start_udp

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

**Format**

```c
unsigned int dtls_start_udp ( t_dtls_hdl conn_hdl )
```

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn_hdl</td>
<td>The handle of the previously established UDP connection.</td>
<td>t_dtls_hdl</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>The connection was not found.</td>
</tr>
</tbody>
</table>
dtls_close_udp

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

Format

```
t_tls_ret dtls_close_udp ( t_dtls_hdl conn_hdl )
```

Arguments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn_hdl</td>
<td>The UDP connection handle.</td>
<td>t_dtls_hdl</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>The connection was not found.</td>
</tr>
</tbody>
</table>
dtls_udp_srv_open

Call this function to start the DTLS server.

Format

```c
uint16_t port, const t_tls_conn_inf * p_inf, t_udp_hdl * p_conn_hdl );
```

Arguments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>port</td>
<td>The port number of the server used for listening.</td>
<td>uint16_t</td>
</tr>
<tr>
<td>p_inf</td>
<td>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.</td>
<td>t_tls_conn_inf *</td>
</tr>
<tr>
<td>p_conn_hdl</td>
<td>Where to write the handle of the established UDP server listener.</td>
<td>t_udp_hdl *</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>No free TLS server entry was found.</td>
</tr>
<tr>
<td>TLS_IO_ERR</td>
<td>Failed to open UDP port listener.</td>
</tr>
</tbody>
</table>
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**

```c
#include "t_tls.h"

#include "t_udp.h"

t_tls_ret dtls_udp_srv_close ( t_udp_hdl conn_hdl )
```

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn_hdl</td>
<td>The handle of the previously established server listener.</td>
<td>t_udp_hdl</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>The connection was not found.</td>
</tr>
<tr>
<td>TLS_DTLS_CONNECT_ERR</td>
<td>The server is still connected; close all its connections.</td>
</tr>
</tbody>
</table>
dtls_get_srv_conn_udp

Call this function to get the last established DTLS connection.

Format

```c
int dtls_get_srv_conn_udp(
    t_udp_hdl srv_hdl,
    t_dtls_hdl * const p_conn_hdl
);
```

Arguments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>srv_hdl</td>
<td>The handle of the previously established server listener.</td>
<td>t_udp_hdl</td>
</tr>
<tr>
<td>conn_hdl</td>
<td>A pointer to the DTLS connection handle.</td>
<td>t_dtls_hdl *</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>The connection was not found.</td>
</tr>
</tbody>
</table>
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

```c
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_ip_port</td>
<td>A pointer to the address and port number of the server to connect to.</td>
<td>t_ip_port *</td>
</tr>
<tr>
<td>p_inf</td>
<td>A pointer to the structure containing connection parameters.</td>
<td>t_tls_conn_inf *</td>
</tr>
<tr>
<td>conn_ticket</td>
<td>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.</td>
<td>t_tls_ticket</td>
</tr>
<tr>
<td>p_conn_hdl</td>
<td>Where to write the connection handle.</td>
<td>t_dtls_hdl *</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_WAIT</td>
<td>The connection was not opened.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>The p_tci_peer_name in p_inf was not specified.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>No free DTLS connection was found.</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn_hdl</td>
<td>The UDP connection handle.</td>
<td>t_dtls_hdl</td>
</tr>
<tr>
<td>p_data</td>
<td>A pointer to the IP buffer which contains the data to send.</td>
<td>uint8_t *</td>
</tr>
<tr>
<td></td>
<td>Allocate this by using <code>dtls_get_buffer_udp()</code>.</td>
<td></td>
</tr>
<tr>
<td>data_len</td>
<td>The length of the data in bytes.</td>
<td>uint16_t</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>Else</td>
<td>See Error Codes.</td>
</tr>
</tbody>
</table>
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

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

Arguments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn_hdl</td>
<td>The UDP connection handle.</td>
<td>t_dtls_hndl</td>
</tr>
<tr>
<td>pp_buf</td>
<td>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().</td>
<td>uint8_t **</td>
</tr>
<tr>
<td>p_data_len</td>
<td>On return, a pointer to the length of the decoded PDU.</td>
<td>uint16_t *</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_WAIT</td>
<td>There is no data to receive.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>A parameter was invalid.</td>
</tr>
<tr>
<td>TLS_IO_ERR</td>
<td>I/O operation failed.</td>
</tr>
</tbody>
</table>
dtls_get_buffer_udp

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

If a non-preemptive RTOS is used, \textit{timeout} is ignored and function execution is non-blocking.

**Format**

```c
const t_dtls_hdl conn_hdl,
data_len,
uint8_t * * pp_buf,
uint32_t timeout,
t_ip_ntf * const p_ntf,
uint16_t * p_buf_len
```

**Arguments**

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

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_WAIT</td>
<td>The function was called before the handshake was finished.</td>
</tr>
<tr>
<td>TLS_MEM_ERR</td>
<td>No buffer of requested size available.</td>
</tr>
</tbody>
</table>
dtls_get_ticket_udp

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

Format

```c
void dtls_get_ticket_udp (const t_dtls_hdl conn_hdl, t_tls_ticket * p_conn_ticket);
```

Arguments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn_hdl</td>
<td>The handle of the DTLS connection.</td>
<td>t_dtls_hdl</td>
</tr>
<tr>
<td>p_conn_ticket</td>
<td>On return, a pointer to the variable that received the ticket handle.</td>
<td>t_tls_ticket *</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>A parameter is incorrect.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>The specified handle was not found.</td>
</tr>
</tbody>
</table>
dtls_get_state_udp

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

Format

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

Arguments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn_hdl</td>
<td>The UDP connection handle.</td>
<td>t_dtls_hdl</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_CONNST_CLOSED</td>
<td>The connection is closed or does not exist.</td>
</tr>
<tr>
<td>TLS_CONNST_HANDSHAKE</td>
<td>The connection is in handshake state.</td>
</tr>
<tr>
<td>TLS_CONNST_OPERATING</td>
<td>The connection is established and the user can send and receive data.</td>
</tr>
</tbody>
</table>
dtls_get_ip_udp

Call this function to get the IP address and port information of a DTLS connection's peer.

Format

```c
#define dtls_get_ip_udp

t_tls_ret dtls_get_ip_udp ( t_dtls_hdl conn_hdl, t_ip_port * p_ip_port )
```

Arguments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn_hdl</td>
<td>The UDP connection handle.</td>
<td>t_dtls_hdl</td>
</tr>
<tr>
<td>p_ip_port</td>
<td>Where to write the IP address and port.</td>
<td>t_ip_port *</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>A parameter is invalid.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>The connection was not found.</td>
</tr>
</tbody>
</table>
6.7 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.

The functions are the following:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tls_client_handshake_socket()</code></td>
<td>Called from the client application, establishes a connection with the server.</td>
</tr>
<tr>
<td><code>tls_client_handshake_socket_ext()</code></td>
<td>Called from the client application, executes the extended version of the client handshake function.</td>
</tr>
<tr>
<td><code>tls_server_handshake_socket()</code></td>
<td>Called from the server application, establishes the TLS connection with the client.</td>
</tr>
<tr>
<td><code>tls_server_handshake_socket_ext()</code></td>
<td>Called from the server application, executes the extended version of the server handshake function.</td>
</tr>
<tr>
<td><code>tls_renegotiate_socket()</code></td>
<td>Starts handshake renegotiation.</td>
</tr>
<tr>
<td><code>tls_send_socket()</code></td>
<td>Sends data to the peer over a connection.</td>
</tr>
<tr>
<td><code>tls_receive_socket()</code></td>
<td>Receives data (a PDU) and decodes it.</td>
</tr>
<tr>
<td><code>tls_select_socket()</code></td>
<td>Waits until data is received on any socket in a specified set or transmitted on a socket in another specified set.</td>
</tr>
<tr>
<td><code>tls_poll_socket()</code></td>
<td>Waits until either a specified event or timeout occurs.</td>
</tr>
<tr>
<td><code>tls_get_ticket_socket()</code></td>
<td>Retrieves a ticket that can be used to resume a previously established socket connection.</td>
</tr>
<tr>
<td><code>tls_get_state_socket()</code></td>
<td>Gets the status of a TLS connection.</td>
</tr>
<tr>
<td><code>tls_close_socket()</code></td>
<td>Called from the client or server application, closes a TLS connection.</td>
</tr>
</tbody>
</table>
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**

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

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sockfd</td>
<td>The socket descriptor of the previously established socket connection.</td>
<td>t_psp_socket</td>
</tr>
<tr>
<td>p_peer_name</td>
<td>A pointer to the name of the server. This is used for certificate verification.</td>
<td>char_t *</td>
</tr>
<tr>
<td>conn_ticket</td>
<td>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.</td>
<td>t_tls_ticket</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution; connection is established.</td>
</tr>
<tr>
<td>TLS_WAIT</td>
<td>No data was received from the server.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>p_peer_name not specified.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>The specified object was not found.</td>
</tr>
<tr>
<td>TLS_MEM_ERR</td>
<td>Memory allocation error.</td>
</tr>
<tr>
<td>Return value</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>TLS_HS_ERR</td>
<td>Handshake protocol error.</td>
</tr>
</tbody>
</table>
tls_client_handshake_socket_ext

Call this function from the client application to execute the extended version of the client handshake function.

Received messages are passed to tls_hdlr_pdu for further processing. If a communication error occurs, an alarm is sent and an error code is returned.

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

**Format**

```c
int t_tls_ret tls_client_handshake_socket_ext ( 
    t_psp_socket sockfd, 
    uint8_t    flags, 
    const char_t * const p_peer_name, 
    t_tls_ticket conn_ticket )
```

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sockfd</td>
<td>The socket descriptor of the previously established socket connection.</td>
<td>t_psp_socket</td>
</tr>
<tr>
<td>flags</td>
<td>The t_tls_conn_inf connection flags.</td>
<td>uint8_t</td>
</tr>
<tr>
<td>p_peer_name</td>
<td>A pointer to the name of the server. This is used for certificate verification.</td>
<td>char_t *</td>
</tr>
<tr>
<td>conn_ticket</td>
<td>The handle of the session ticket that can be used for handshake resumption. If you do not want to resume a connection, set this to TLS_INVALID_CONN_TICKET.</td>
<td>t_tls_ticket</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution; connection is established.</td>
</tr>
<tr>
<td>TLS_WAIT</td>
<td>No data was received from the server.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>The connection was not found.</td>
</tr>
<tr>
<td>TLS_HS_ERR</td>
<td>Not allowed to renegotiate.</td>
</tr>
<tr>
<td>Return value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>------------------------------------------------------</td>
</tr>
<tr>
<td>TLS_MEM_ERR</td>
<td>Not enough memory to allocate handshake buffer.</td>
</tr>
</tbody>
</table>
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**

```c
void tls_server_handshake_socket ( t_psp_socket sockfd, uint8_t b_cli_verify, const char *const p_peer_name );
```

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sockfd</td>
<td>The socket descriptor of the previously established socket connection.</td>
<td>t_psp_socket</td>
</tr>
<tr>
<td>b_cli_verify</td>
<td>Set this TRUE if you want to verify the authenticity of the client.</td>
<td>uint8_t</td>
</tr>
<tr>
<td>p_peer_name</td>
<td>The name of the client. This is used for certificate verification.</td>
<td>char *</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution; the connection is established.</td>
</tr>
<tr>
<td>TLS_WAIT</td>
<td>No data was received from the client.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>The specified object was not found.</td>
</tr>
<tr>
<td>TLS_MEM_ERR</td>
<td>Memory allocation error.</td>
</tr>
<tr>
<td>TLS_HS_ERR</td>
<td>Handshake protocol error.</td>
</tr>
</tbody>
</table>
tls_server_handshake_socket_ext

Call this function from the server to execute the extended version of the server handshake function.

Format

```c
#include <hcc/tls/tls_functions.h>

int t_tls_ret tls_server_handshake_socket ( t_psp_socket sockfd, uint8_t flags, const char_t * p_peer_name );
```

Arguments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sockfd</td>
<td>The socket descriptor of the previously established socket connection.</td>
<td>t_psp_socket</td>
</tr>
<tr>
<td>flags</td>
<td>The t_tls_conn_inf connection flags.</td>
<td>uint8_t</td>
</tr>
<tr>
<td>p_peer_name</td>
<td>A pointer to the name of the peer.</td>
<td>char_t *</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution; connection is established.</td>
</tr>
<tr>
<td>TLS_WAIT</td>
<td>No data was received from the server.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>The connection was not found.</td>
</tr>
<tr>
<td>TLS_HS_ERR</td>
<td>Not allowed to renegotiate.</td>
</tr>
<tr>
<td>TLS_MEM_ERR</td>
<td>Not enough memory to allocate handshake buffer.</td>
</tr>
</tbody>
</table>
tls_renegotiate_socket

Call this function to start TLS handshake renegotiation.

**Note:** This is only available if configuration option `TLS_RENEGOTIATION_ENABLE` is set.

**Format**

```c
void tls_renegotiate_socket ( t_psp_socket sockfd, uint8_t b_verify );
```

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sockfd</td>
<td>The socket descriptor of the previously established socket connection.</td>
<td>t_psp_socket</td>
</tr>
<tr>
<td>b_verify</td>
<td>A flag indicating whether the client should be verified.</td>
<td>uint8_t</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>A parameter is invalid.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>The connection was not found.</td>
</tr>
<tr>
<td>TLS_WAIT</td>
<td>Connection is not open yet; wait for handshake to end.</td>
</tr>
<tr>
<td>TLS_HS_ERR</td>
<td>Not allowed to renegotiate.</td>
</tr>
<tr>
<td>TLS_MEM_ERR</td>
<td>Not enough memory to allocate handshake buffer.</td>
</tr>
</tbody>
</table>
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

```c
void tls_send_socket ( t_psp_socket sockfd, uint8_t * const p_data, uint16_t data_len );
```

Arguments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sockfd</td>
<td>The socket descriptor.</td>
<td>t_psp_socket</td>
</tr>
<tr>
<td>p_data</td>
<td>A pointer to the data to be sent.</td>
<td>uint8_t *</td>
</tr>
<tr>
<td>data_len</td>
<td>The length of the data in bytes.</td>
<td>uint16_t</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>Else</td>
<td>See Error Codes.</td>
</tr>
</tbody>
</table>
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 (  
t_psp_socket sockfd,  
uint8_t ** pp_buf,  
uint16_t buf_size,  
uint16_t * p_data_len )
```

Arguments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sockfd</td>
<td>The socket descriptor.</td>
<td>t_psp_socket</td>
</tr>
<tr>
<td>pp_buf</td>
<td>On return, a pointer to the receive buffer.</td>
<td>uint8_t **</td>
</tr>
<tr>
<td>buf_size</td>
<td>The size of the receive buffer.</td>
<td>uint16_t</td>
</tr>
<tr>
<td>p_data_len</td>
<td>On return, the length of the decoded PDU.</td>
<td>uint16_t *</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_WAIT</td>
<td>No data was received.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>A parameter was invalid.</td>
</tr>
<tr>
<td>TLS_IO_ERR</td>
<td>I/O operation failed.</td>
</tr>
</tbody>
</table>
**tls_select_socket**

Use this function to wait until data is received on any socket in a specified set or transmitted on a socket in another specified set.

This function is a wrapper for the standard `socket_select()`. It also checks whether the TLS stack contains data to be read by the user.

**Format**

```c
int32_t tls_select_socket ( int32_t nfds, t_psp_sc_fdset *p_readfds, t_psp_sc_fdset *p_writefds, t_psp_sc_fdset *p_exceptfds, t_psp_sc_timeval *p_timeout );
```

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>nfds</td>
<td>The maximum number of the set.</td>
<td>int32_t</td>
</tr>
<tr>
<td>p_readfds</td>
<td>A pointer to the set of socket descriptors to monitor for incoming data.</td>
<td>t_psp_sc_fdset *</td>
</tr>
<tr>
<td>p_writefds</td>
<td>A pointer to the set of socket descriptors to monitor for completion of a send operation.</td>
<td>t_psp_sc_fdset *</td>
</tr>
<tr>
<td>p_exceptfds</td>
<td>A pointer to the set of socket descriptors to monitor for exceptional events (this only happens if a socket is disconnected).</td>
<td>t_psp_sc_fdset *</td>
</tr>
<tr>
<td>p_timeout</td>
<td>A pointer to a timeout structure.</td>
<td>t_psp_sc_timeval *</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A number.</td>
<td>The number of socket descriptors ready for input/output.</td>
</tr>
<tr>
<td>SOCKET_ERROR</td>
<td>Operation failed.</td>
</tr>
</tbody>
</table>
tls_poll_socket

Use this function to wait until either a specified event or timeout occurs.

This function is a wrapper for the standard `socket_poll()`. It also checks whether the TLS stack contains data to be read by the user.

**Format**

```c
void tls_poll_socket ( 
    t_psp_sc_pollfd *   p_fds, 
    uint32_t            nfds, 
    int32_t             timeout );
```

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_fds</td>
<td>A pointer to the array of <code>pollfd</code> structures.</td>
<td>t_psp_sc_pollfd *</td>
</tr>
<tr>
<td>nfds</td>
<td>The number of entries in the <code>p_fds</code> array.</td>
<td>uint32_t</td>
</tr>
<tr>
<td>timeout</td>
<td>The time in milliseconds that the caller will wait for the result. Use -1 for no timeout.</td>
<td>int32_t</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A number.</td>
<td>The number of sockets where an event occurred: successful execution.</td>
</tr>
<tr>
<td>SOCKET_ERROR</td>
<td>Operation failed.</td>
</tr>
</tbody>
</table>
tls_get_ticket_socket

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

Format

\[
\text{t_tls_ret tls_get_ticket_socket (}
\text{\hspace{10pt} t_psp_socket sockfd,}
\text{\hspace{10pt} t_tls_ticket * p_conn_ticket )}
\]

Arguments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sockfd</td>
<td>The socket descriptor.</td>
<td>t_psp_socket</td>
</tr>
<tr>
<td>p_conn_ticket</td>
<td>On return, a pointer to the variable which received the ticket handle.</td>
<td>t_tls_ticket *</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>A parameter is incorrect.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>The specified connection was not found.</td>
</tr>
</tbody>
</table>


tls_get_state_socket

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

**Format**

```c
t_tls_conn_status tls_get_state_socket ( t_psp_socket sockfd )
```

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sockfd</td>
<td>The socket descriptor.</td>
<td>t_psp_socket</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_CONNST_CLOSED</td>
<td>The connection is closed or does not exist.</td>
</tr>
<tr>
<td>TLS_CONNST_HANDSHAKE</td>
<td>The connection is in handshake state.</td>
</tr>
<tr>
<td>TLS_CONNST_OPERATING</td>
<td>The connection is established and the user can send and receive data.</td>
</tr>
</tbody>
</table>
tls_close_socket

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

Format

\[
\text{t_tls_ret tls_close_socket ( t_psp_socket sockfd )}
\]

Arguments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sockfd</td>
<td>The socket descriptor.</td>
<td>t_psp_socket</td>
</tr>
</tbody>
</table>

Return values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>Connection not found.</td>
</tr>
</tbody>
</table>
6.8 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.

The functions are the following:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dtls_client_handshake_socket()</code></td>
<td>Called from the client application, establishes a DTLS connection with the server.</td>
</tr>
<tr>
<td><code>dtls_server_handshake_socket()</code></td>
<td>Called from the server application, establishes the DTLS connection with the client.</td>
</tr>
<tr>
<td><code>dtls_send_socket()</code></td>
<td>Sends data to the peer over an established connection.</td>
</tr>
<tr>
<td><code>dtls_receive_socket()</code></td>
<td>Receives data (a PDU) and decodes it.</td>
</tr>
<tr>
<td><code>dtls_get_ticket_socket()</code></td>
<td>Retrieves a ticket that can be used to resume a previously established DTLS connection.</td>
</tr>
<tr>
<td><code>dtls_get_state_socket()</code></td>
<td>Gets the status of a DTLS connection.</td>
</tr>
<tr>
<td><code>dtls_get_srv_conn_socket()</code></td>
<td>Looks for an established DTLS connection that is related to a given port handle.</td>
</tr>
<tr>
<td><code>dtls_get_ip_socket()</code></td>
<td>Gets the IP address and port information of a DTLS connection's peer.</td>
</tr>
<tr>
<td><code>dtls_srv_max_client_socket()</code></td>
<td>Limits the number of connections allowed for a specific DTLS server.</td>
</tr>
<tr>
<td><code>dtls_srv_clear_configuration()</code></td>
<td>Resets the DTLS server configuration to the default values.</td>
</tr>
<tr>
<td><code>dtls_close_socket()</code></td>
<td>Called from the client or server application, closes a DTLS connection.</td>
</tr>
</tbody>
</table>
dtls_client_handshake_socket

Call this function from the client application to establish a DTLS 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**

```c
int dtls_client_handshake_socket (           
    t_psp_socket           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**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sockfd</td>
<td>The socket descriptor of the previously established socket connection.</td>
<td>t_psp_socket</td>
</tr>
<tr>
<td>p_ip_port</td>
<td>A pointer to the port number.</td>
<td>t_ip_port *</td>
</tr>
<tr>
<td>p_peer_name</td>
<td>A pointer to the name of the server. This is used for certificate verification.</td>
<td>char_t*</td>
</tr>
<tr>
<td>conn_ticket</td>
<td>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.</td>
<td>t_tls_ticket</td>
</tr>
<tr>
<td>p_conn_hdl</td>
<td>A pointer to the variable that will receive the DTLS connection handle.</td>
<td>t_dtls_hdl *</td>
</tr>
</tbody>
</table>
## Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution; connection is established.</td>
</tr>
<tr>
<td>TLS_WAIT</td>
<td>No data was received from the server.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>$p_{peer_name}$ not specified.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>The specified object was not found.</td>
</tr>
<tr>
<td>TLS_MEM_ERR</td>
<td>Memory allocation error.</td>
</tr>
<tr>
<td>TLS_HS_ERR</td>
<td>Handshake protocol error.</td>
</tr>
</tbody>
</table>
dtls_server_handshake_socket

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

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 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**

```c
T_TLS_RET dtls_server_handshake_socket ( 
    t_psp_socket sockfd, 
    uint8_t b_cli_verify, 
    const char * const p_peer_name, 
    t_dtls_hdl * const p_conn_hdl )
```

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sockfd</td>
<td>The socket descriptor of the previously established socket connection.</td>
<td>t_psp_socket</td>
</tr>
<tr>
<td>b_cli_verify</td>
<td>Set this TRUE if you want to verify the authenticity of the client.</td>
<td>uint8_t</td>
</tr>
<tr>
<td>p_peer_name</td>
<td>The name of the client. This is used for certificate verification.</td>
<td>char *</td>
</tr>
<tr>
<td>p_conn_hdl</td>
<td>A pointer to the variable that receives the DTLS Sockets interface connection handle.</td>
<td>t_dtls_hdl *</td>
</tr>
</tbody>
</table>
Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution; the connection is established.</td>
</tr>
<tr>
<td>TLS_DTLS_NEW_CONN</td>
<td>Successful execution; in this case a new connection was found.</td>
</tr>
<tr>
<td>TLS_WAIT</td>
<td>No data was received from the client.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>The specified object was not found.</td>
</tr>
<tr>
<td>TLS_MEM_ERR</td>
<td>Memory allocation error.</td>
</tr>
<tr>
<td>TLS_HS_ERR</td>
<td>Handshake protocol error.</td>
</tr>
</tbody>
</table>
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

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

Arguments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn_hdl</td>
<td>The DTLS connection handle.</td>
<td>t_dtls_hdl</td>
</tr>
<tr>
<td>p_data</td>
<td>A pointer to the data to be sent.</td>
<td>uint8_t *</td>
</tr>
<tr>
<td>data_len</td>
<td>The length of the data in bytes.</td>
<td>uint16_t</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>Else</td>
<td>See Error Codes.</td>
</tr>
</tbody>
</table>
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**

```c
void dtls_receive_socket ( t_dtls_hdl conn_hdl, uint8_t ** pp_buf, uint16_t buf_size, uint16_t * p_data_len )
```

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn_hdl</td>
<td>The DTLS connection handle.</td>
<td>t_dtls_hdl</td>
</tr>
<tr>
<td>pp_buf</td>
<td>On return, a pointer to the receive data buffer.</td>
<td>uint8_t **</td>
</tr>
<tr>
<td>buf_size</td>
<td>The size of the receive buffer.</td>
<td>uint16_t</td>
</tr>
<tr>
<td>p_data_len</td>
<td>On return, the length of the decoded PDU.</td>
<td>uint16_t *</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_WAIT</td>
<td>No data was received.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>A parameter is invalid.</td>
</tr>
<tr>
<td>TLS_IO_ERR</td>
<td>Operation failed; I/O operation error.</td>
</tr>
</tbody>
</table>
dtls_get_ticket_socket

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

Format

```c
int t_tls_ret dtls_get_ticket_socket ( t_dtls_hdl conn_hdl,
                                         t_tls_ticket * p_conn_ticket );
```

Arguments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn_hdl</td>
<td>The DTLS connection handle.</td>
<td>t_dtls_hdl</td>
</tr>
<tr>
<td>p_conn_ticket</td>
<td>On return, a pointer to the variable that received the ticket handle.</td>
<td>t_tls_ticket *</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>A parameter is incorrect.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>The specified socket was not found.</td>
</tr>
</tbody>
</table>
dtls_get_state_socket

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

**Format**

```c
#include <tls.h>
int dtls_get_state_socket(t_tls_conn_info *conn, t_dtls_hdl conn_hdl);
```

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn_hdl</td>
<td>The DTLS connection handle.</td>
<td>t_dtls_hdl</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_CONNST_CLOSED</td>
<td>The connection is closed or does not exist.</td>
</tr>
<tr>
<td>TLS_CONNST_HANDSHAKE</td>
<td>The connection is in handshake state.</td>
</tr>
<tr>
<td>TLS_CONNST_OPERATING</td>
<td>The connection is established and the user can send and receive data.</td>
</tr>
</tbody>
</table>
dtls_get_srv_conn_socket

Use this function to look for an established DTLS connection related to a given port handle.

Format

```c
int dtls_get_srv_conn_socket ( t_psp_socket sockfd, t_dtls_hdl * p_conn_hdl );
```

Arguments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sockfd</td>
<td>The socket descriptor.</td>
<td>t_psp_socket</td>
</tr>
<tr>
<td>p_conn_hdl</td>
<td>On return, a pointer to the handle of the DTLS connection.</td>
<td>t_dtls_hdl *</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>The connection is closed or does not exist.</td>
</tr>
</tbody>
</table>
dtls_get_ip_socket

Call this function to get the IP address and port information of a DTLS connection's peer.

Format

```c
int t_tls_ret dtls_get_ip_socket ( 
    t_dtls_hdl conn_hdl, 
    uint8_t * p_ip_addr, 
    uint8_t * p_len, 
    uint16_t * p_port )
```

Arguments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn_hdl</td>
<td>The DTLS connection handle.</td>
<td>t_dtls_hdl</td>
</tr>
<tr>
<td>p_ip_addr</td>
<td>On return, a pointer to the IP address.</td>
<td>uint8_t *</td>
</tr>
<tr>
<td>p_len</td>
<td>On return, a pointer to the IP address length.</td>
<td>uint8_t *</td>
</tr>
<tr>
<td>p_port</td>
<td>On return, a UDP port number.</td>
<td>uint16_t *</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution; connection is established.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>A parameter is invalid.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>The connection was not found.</td>
</tr>
</tbody>
</table>
dtls_srv_max_client_socket

Use this function to limit the number of connections allowed for a specific DTLS server.

Format

```
t_tls_ret dtls_srv_max_client_socket (  
    t_psp_socket   sockfd,  
    uint8_t        max_conn )
```

Arguments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sockfd</td>
<td>The socket descriptor.</td>
<td>t_psp_socket</td>
</tr>
<tr>
<td>max_conn</td>
<td>The maximum number of connections allowed for the DTLS server.</td>
<td>uint8_t</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>The socket descriptor was not found.</td>
</tr>
</tbody>
</table>
dtls_srv_clear_configuration

Call this function from the DTLS client application to reset the DTLS server configuration to the default values.

Call this when connection limits need to be removed or when the socket for the specific DTLS server has been closed.

**Format**

```c
 t_tls_ret dtls_srv_clear_configuration( t_psp_socket sockfd )
```

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sockfd</td>
<td>The DTLS server's socket descriptor.</td>
<td>t_psp_socket</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>No free DTLS connection was found.</td>
</tr>
</tbody>
</table>
dtls_close_socket

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

**Format**

```c
void dtls_close_socket(t_dtls_socket_hdl p_conn_hdl);
```

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_conn_hdl</td>
<td>The DTLS connection handle.</td>
<td>t_dtls_socket_hdl</td>
</tr>
</tbody>
</table>

**Return values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>Connection not found.</td>
</tr>
</tbody>
</table>
6.9 TLS RAW Interface

This is used by the EAP-TLS module to communicate with the TLS stack.

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

The functions are the following:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tls_init_raw()</td>
<td>Initiates a connection to a remote port using TLS.</td>
</tr>
<tr>
<td>tls_close_raw()</td>
<td>Closes the TLS connection.</td>
</tr>
<tr>
<td>tls_start_raw()</td>
<td>Starts a TLS handshake connection.</td>
</tr>
<tr>
<td>tls_get_read_data_raw()</td>
<td>Gets the number of bytes of data to be read from the TLS stack. This is data that should be sent to the peer.</td>
</tr>
<tr>
<td>tls_hdl_data_raw()</td>
<td>Sends data to the TLS stack for handling.</td>
</tr>
<tr>
<td>tls_read_data_raw()</td>
<td>Gets data from the TLS stack for passing to the peer.</td>
</tr>
<tr>
<td>tls_read_data_ack_raw()</td>
<td>Tells the TLS stack the number of PDUs that the peer has acknowledged as being sent. This data can then be released by the TLS stack.</td>
</tr>
<tr>
<td>tls_get_state_raw()</td>
<td>Gets the state of a TLS connection.</td>
</tr>
<tr>
<td>tls_get_ticket_raw()</td>
<td>Gets a connection's session ticket handle.</td>
</tr>
<tr>
<td>tls_get_secrets_raw()</td>
<td>Generates a secret value using the TLS pseudorandom function.</td>
</tr>
<tr>
<td>tls_get_randoms_raw()</td>
<td>Gets the random numbers that were used by a connection during a handshake.</td>
</tr>
</tbody>
</table>
tls_init_raw

Use this function to initiate a connection to a remote port using TLS.

**Note:** To start a TLS handshake immediately, assign the value TLS_TCP_CONN_INF_FLAG_START to `t_tls_conn_inf->tci_flags`.

**Format**

```c
void tls_init_raw(
    const char_t * p_peer_name,
    uint8_t b_verify,
    t_tls_ticket conn_ticket,
    t_tls_raw_hdl * const p_raw_hdl
);
```

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_peer_name</td>
<td>A pointer to the string containing the peer name.</td>
<td>char_t *</td>
</tr>
<tr>
<td>b_verify</td>
<td>Set this to TRUE to enable client verification by the server.</td>
<td>uint8_t</td>
</tr>
<tr>
<td>conn_ticket</td>
<td>The handle of the session ticket that is used for resumption.</td>
<td>t_tls_ticket</td>
</tr>
<tr>
<td>p_raw_hdl</td>
<td>Where to write the connection handle.</td>
<td>t_tls_raw_hdl *</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_WAIT</td>
<td>The connection was not opened.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>Parameter <code>t_tls_conn_inf-&gt;tci_peer_name</code> was not specified.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>No free TLS connection was found.</td>
</tr>
</tbody>
</table>
tls_close_raw

Use this function to close a connection.

Format

```c
int tls_close_raw (  
    t_tls_raw_hdl conn_hdl,  
    uint8_t b_send_alert )
```

Arguments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn_hdl</td>
<td>The connection handle.</td>
<td>t_tls_raw_hdl</td>
</tr>
<tr>
<td>b_send_alert</td>
<td>Set this to TRUE to send a close notification to the peer.</td>
<td>uint8_t</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>The TLS connection was not found.</td>
</tr>
</tbody>
</table>
tls_start_raw

Use this function to start a TLS handshake connection.

**Format**

```c
uint8_t     b_server)
```

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn_hdl</td>
<td>The TCP connection handle.</td>
<td>t_tls_raw_hdl</td>
</tr>
<tr>
<td>b_server</td>
<td>A flag indicating whether to start a client or server connection. Set this TRUE for a server connection.</td>
<td>uint8_t</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>Operation failed.</td>
</tr>
</tbody>
</table>
**tls_get_read_data_raw**

Use this function to retrieve the number of bytes of data to be read from the TLS stack. This is data that should be sent to the peer.

**Format**

```c
void tls_get_read_data_raw (.const t_tls_raw_hdl conn_hdl, uint16_t * p_rd_cnt );
```

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn_hdl</td>
<td>The RAW connection handle.</td>
<td>t_tls_raw_hdl</td>
</tr>
<tr>
<td>p_rd_cnt</td>
<td>On return, a pointer to the number of bytes of data to be read from the TLS stack.</td>
<td>uint16_t *</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_ERROR</td>
<td>Operation failed.</td>
</tr>
</tbody>
</table>
**tls_hdl_data_raw**

Use this function to send data to the TLS stack for handling.

**Format**

```c
#include <tls.h>
t_tls_ret tls_hdl_data_raw ( 
    t_tls_raw_hdl     conn_hdl,
    const uint8_t *   p_data,
    uint16_t          data_len )
```

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn_hdl</td>
<td>The connection handle.</td>
<td>t_tls_raw_hdl</td>
</tr>
<tr>
<td>p_data</td>
<td>A pointer to the data to be sent to the TLS stack.</td>
<td>uint8_t*</td>
</tr>
<tr>
<td>data_len</td>
<td>The length of the data in bytes.</td>
<td>uint16_t</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>A parameter is incorrect.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>The TLS connection was not found.</td>
</tr>
</tbody>
</table>
tls_read_data_raw

Use this function to get data from the TLS stack for passing to the peer.

Format

\[
\text{t_tls_ret tls_read_data_raw (}
  \text{t_tls_raw_hdl conn_hdl,}
  \text{uint8_t * p_buf,}
  \text{uint16_t * p_data_len )}
\]

Arguments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn_hdl</td>
<td>The RAW connection handle.</td>
<td>t_tls_raw_hdl</td>
</tr>
<tr>
<td>p_buf</td>
<td>On return, a pointer to the received TLS data.</td>
<td>uint8_t *</td>
</tr>
<tr>
<td>p_data_len</td>
<td>On return, a pointer to the length of the decoded received PDU.</td>
<td>uint16_t *</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_WAIT</td>
<td>No data was received.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>A parameter was invalid.</td>
</tr>
<tr>
<td>TLS_IO_ERR</td>
<td>There was a problem during the I/O operation.</td>
</tr>
</tbody>
</table>
tls_read_data_ack_raw

Use this function to tell the TLS stack the number of PDUs that the peer has acknowledged as being sent. This data can then be released by the TLS stack.

**Format**

```c
void tls_read_data_ack_raw (t_tls_raw_hdl conn_hdl, uint16_t ack_cnt );
```

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn_hdl</td>
<td>The RAW connection handle.</td>
<td>t_tls_raw_hdl</td>
</tr>
<tr>
<td>ack_cnt</td>
<td>The number of PDUs that were acknowledged by the peer.</td>
<td>uint16_t</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_ERROR</td>
<td>Operation failed.</td>
</tr>
</tbody>
</table>
tls_get_state_raw

Use this function to get the state of a TLS connection.

Format

```c
    t_ttls_conn_status tls_get_state_raw ( const t_tls_raw_hdl conn_hdl )
```

Arguments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn_hdl</td>
<td>The RAW connection handle.</td>
<td>t_tls_raw_hdl</td>
</tr>
</tbody>
</table>

Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_CONNST_CLOSED</td>
<td>The connection is closed or does not exist.</td>
</tr>
<tr>
<td>TLS_CONNST_HANDSHAKE</td>
<td>The connection is in handshake state.</td>
</tr>
<tr>
<td>TLS_CONNST_OPERATING</td>
<td>The connection is established and the user can send and receive data.</td>
</tr>
</tbody>
</table>
tls_get_ticket_raw

Use this function to get a connection's session ticket handle.

**Format**

```c
int t_tls_ret tls_get_ticket_raw(
    const t_tls_raw_hdl conn_hdl,
    t_tls_ticket * p_conn_ticket)
```

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn_hdl</td>
<td>The connection handle.</td>
<td>t_tls_raw_hdl</td>
</tr>
<tr>
<td>p_conn_ticket</td>
<td>On return, a pointer to the session ticket handle.</td>
<td>t_tls_ticket *</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_PARAM_ERR</td>
<td>A parameter is incorrect.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>The TLS connection was not found.</td>
</tr>
</tbody>
</table>
tls_get_secrets_raw

Use this function to generate a secret value using the TLS pseudorandom function.

The pseudorandom function takes as its input:

- key - the master secret value (if `b_use_ms` is TRUE) otherwise a 0 length key.
- label - the `p_label` provided by the user.
- seed - session random values.

**Format**

```c
#include "tls.h"

int tls_get_secrets_raw(
    const t_tls_raw_hdl conn_hdl,
    uint8_t const * p_label,
    uint8_t b_use_ms,
    uint16_t out_length,
    uint8_t const ** pp_secret )
```

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>conn_hdl</code></td>
<td>The RAW connection handle.</td>
<td>t_tls_raw_hdl</td>
</tr>
<tr>
<td><code>p_label</code></td>
<td>A pointer to the label to use to generate the secret value.</td>
<td>uint8_t *</td>
</tr>
<tr>
<td><code>b_use_ms</code></td>
<td>A flag indicating whether the master secret should be used as a key (see above).</td>
<td>uint8_t</td>
</tr>
<tr>
<td><code>out_length</code></td>
<td>The length of the generated output value.</td>
<td>uint16_t</td>
</tr>
<tr>
<td><code>pp_secret</code></td>
<td>On return, a pointer to the pointer to the generated secret value (32 bytes).</td>
<td>uint8_t **</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_WAIT</td>
<td>The parameters are not available yet.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>The TLS connection was not found.</td>
</tr>
</tbody>
</table>
tls_get_randoms_raw

Use this function to get the random numbers that were used by a connection during a handshake.

**Format**

```c
int tls_get_randoms_raw (const t_tls_raw_hdl conn_hdl, uint8_t const ** pp_cli_rnd, uint8_t const ** pp_srv_rnd);
```

**Arguments**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn_hdl</td>
<td>The RAW connection handle.</td>
<td>t_tls_raw_hdl</td>
</tr>
<tr>
<td>pp_cli_rnd</td>
<td>On return, a pointer to the pointer to the client random value (32 bytes).</td>
<td>uint8_t **</td>
</tr>
<tr>
<td>pp_srv_rnd</td>
<td>On return, a pointer to the pointer to the server random value (32 bytes).</td>
<td>uint8_t **</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_OK</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>TLS_WAIT</td>
<td>The parameters are not available yet.</td>
</tr>
<tr>
<td>TLS_NOT_FOUND_ERR</td>
<td>The TLS connection was not found.</td>
</tr>
</tbody>
</table>
## 6.10 Error Codes

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

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

This section describes the main elements that are defined in the API Header file.

**t_tls_connection**

The `t_tls_connection` structure takes this form:

<table>
<thead>
<tr>
<th>Element</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tc_encryp_buf</td>
<td>uint8_t</td>
<td>A temporary buffer used for encryption operations.</td>
</tr>
<tr>
<td>[TLS_DATA_BUF_SIZE]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tc_cert_oid</td>
<td>uint8_t</td>
<td>The certificate identifier.</td>
</tr>
<tr>
<td>[TLS_OID_SIZE]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tc_recv_buf</td>
<td>uint8_t</td>
<td>The buffer for received data.</td>
</tr>
<tr>
<td>[TLS_RECV_BUF_SIZE]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tc_conn_hndl</td>
<td>t_tcp_conn_hdl</td>
<td>The connection handle for TLS with TCP (only when a native TCP/IP interface is enabled).</td>
</tr>
<tr>
<td>tc_host_port</td>
<td>uint16_t</td>
<td>Host port.</td>
</tr>
<tr>
<td>tc_ip_addr</td>
<td>t_ip_port</td>
<td>Peer IP address data.</td>
</tr>
<tr>
<td>tc_conn_cnt</td>
<td>uint8_t</td>
<td>Connection execution counter.</td>
</tr>
<tr>
<td>tc_dtls_data</td>
<td>t_dtls_conn_data</td>
<td>DTLS connection-specific data.</td>
</tr>
<tr>
<td>tc_ntf_idx</td>
<td>uint8_t</td>
<td>The index of the user notify structure that is bound with this connection.</td>
</tr>
<tr>
<td>p_tc_rx_q</td>
<td>t_ip_buffer *</td>
<td>The received messages queue.</td>
</tr>
<tr>
<td>p_tc_tx_q</td>
<td>t_ip_buffer *</td>
<td>The transmitted messages queue.</td>
</tr>
<tr>
<td>tc_timeout</td>
<td>uint16_t</td>
<td>The handshake timeout counter.</td>
</tr>
<tr>
<td>tc_dtls_tim_cnt</td>
<td>uint16_t</td>
<td>DTLS timeout counter.</td>
</tr>
<tr>
<td>tc_dtls_timeout</td>
<td>uint16_t</td>
<td>DTLS timeout value.</td>
</tr>
<tr>
<td>tc_conn_hndl_sock</td>
<td>int</td>
<td>The connection handle for the Sockets API, used by both TLS and DTLS with Sockets.</td>
</tr>
<tr>
<td>Element</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>tc_mutex</td>
<td>oal_mutex_t</td>
<td>The mutex protecting the connection.</td>
</tr>
<tr>
<td>tc_seq_num</td>
<td>uint64_t</td>
<td>The connection sequence number.</td>
</tr>
<tr>
<td>tc_peer_seq_num</td>
<td>uint64_t</td>
<td>The peer sequence number.</td>
</tr>
<tr>
<td>p_tc_hs_msg_store</td>
<td>uint8_t *</td>
<td>A pointer to the buffer with handshake messages.</td>
</tr>
<tr>
<td>p_tc_next_pdu</td>
<td>uint8_t *</td>
<td>A pointer to the next PDU to be processed.</td>
</tr>
<tr>
<td>p_tc_pub_key</td>
<td>uint8_t *</td>
<td>A pointer to the DER-encoded public key of the peer.</td>
</tr>
<tr>
<td>tc_peer_name [TLS_PEER_NAME_SIZE]</td>
<td>char_t</td>
<td>The name of the peer (only used if peer identity is to be verified).</td>
</tr>
<tr>
<td>p_tc_rd_session</td>
<td>t_tls_session *</td>
<td>A pointer to the session object used to read incoming data.</td>
</tr>
<tr>
<td>p_tc_wr_session</td>
<td>t_tls_session *</td>
<td>A pointer to the session object used to send data.</td>
</tr>
<tr>
<td>p_tc_pd_session</td>
<td>t_tls_session *</td>
<td>A pointer to the session object being negotiated.</td>
</tr>
<tr>
<td>p_tc_cert</td>
<td>t_tls_certificate *</td>
<td>The certificate the client uses to identify itself.</td>
</tr>
<tr>
<td>p_tc_key_ex_priv</td>
<td>uint8_t *</td>
<td>A pointer to the private value of the key exchange algorithm (used by the EDH algorithm).</td>
</tr>
<tr>
<td>tc_host_sec_params</td>
<td>t_tls_security_params</td>
<td>Host side security parameters.</td>
</tr>
<tr>
<td>tc_peer_sec_params</td>
<td>t_tls_security_params</td>
<td>Peer side security parameters.</td>
</tr>
<tr>
<td>tc_hs_count</td>
<td>uint16_t</td>
<td>Size of the data in the handshake buffer plus four bytes of the handshake message header.</td>
</tr>
<tr>
<td>tc_pub_key_len</td>
<td>uint16_t</td>
<td>The length of the DER encoded public key of the peer.</td>
</tr>
<tr>
<td>tc_key_ex_len</td>
<td>uint16_t</td>
<td>The length of the private value of the key exchange algorithm (used by the EDH algorithm).</td>
</tr>
<tr>
<td>tc_rx_len</td>
<td>uint16_t</td>
<td>The length of the data in the receive buffer.</td>
</tr>
<tr>
<td>tc_rx_len</td>
<td>uint16_t</td>
<td>The length of the data in the receive buffer.</td>
</tr>
<tr>
<td>Element</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>tc_rx_decr_len</td>
<td>uint16_t</td>
<td>The length of decrypted data waiting in the receive buffer.</td>
</tr>
<tr>
<td>tc_rx_decr_pos</td>
<td>uint16_t</td>
<td>The position of decrypted data waiting in the receive buffer.</td>
</tr>
<tr>
<td>tc_tx_rem_len</td>
<td>uint16_t</td>
<td>The length of remaining data to be sent.</td>
</tr>
<tr>
<td>tc_tx_rem_buf</td>
<td>uint8_t</td>
<td>The buffer containing remaining data to be sent.</td>
</tr>
<tr>
<td>tc_exp_count</td>
<td>uint16_t</td>
<td>The connection expiry count (this is not used).</td>
</tr>
<tr>
<td>tc_state</td>
<td>t_tls_conn_state</td>
<td>The connection state.</td>
</tr>
<tr>
<td>tc_flags</td>
<td>t_tls_conn_flags</td>
<td>The connection flags (role and type). The role is client or server.</td>
</tr>
<tr>
<td>tc_full_hs</td>
<td>uint8_t</td>
<td>TRUE if a full handshake must be executed, FALSE otherwise.</td>
</tr>
<tr>
<td>tc_client_verify</td>
<td>uint8_t</td>
<td>TRUE if the client identity must be verified, FALSE otherwise.</td>
</tr>
<tr>
<td>tc_ver_minor</td>
<td>uint8_t</td>
<td>Minor version of TLS protocol for the connection.</td>
</tr>
<tr>
<td>tc_hello_ver</td>
<td>uint8_t</td>
<td>The TLS protocol version in the hello message, used to detect rollback attack.</td>
</tr>
<tr>
<td>tc_ext</td>
<td>t_tls_ext</td>
<td>TLS extension-specific parameters.</td>
</tr>
</tbody>
</table>

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.

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

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.

<table>
<thead>
<tr>
<th>Element</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>random_val[TLS_RANDOM_LEN]</td>
<td>uint8_t</td>
<td>Random value used for pre-master key generation.</td>
</tr>
<tr>
<td>mac_key[TLS_MAX_MKEY_LEN]</td>
<td>uint8_t</td>
<td>Key used for MAC generation.</td>
</tr>
<tr>
<td>bulk_key[TLS_MAX_BKEY_LEN]</td>
<td>uint8_t</td>
<td>Key used for bulk encryption/decryption.</td>
</tr>
<tr>
<td>init_vector[TLS_MAX_BKEY_LEN]</td>
<td>uint8_t</td>
<td>Initialization vector for bulk encryption/decryption.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Element</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>digest_prefix[TLS_DIGEST_PREFIX_LEN]</td>
<td>uint8_t</td>
<td>The DER-encoded prefix used for certificate signature calculation.</td>
</tr>
<tr>
<td>digest_prefix_len</td>
<td>uint16_t</td>
<td>The length of the signature prefix.</td>
</tr>
<tr>
<td>block_len</td>
<td>uint16_t</td>
<td>The length of the data block used by the algorithm.</td>
</tr>
<tr>
<td>key_len</td>
<td>uint16_t</td>
<td>The length of the key used by the algorithm.</td>
</tr>
<tr>
<td>digest_len</td>
<td>uint16_t</td>
<td>The length of the digest generated by algorithm.</td>
</tr>
</tbody>
</table>
t_tls_certificate

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

<table>
<thead>
<tr>
<th>Element</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cert_pub_key_len</td>
<td>uint16_t</td>
<td>The length of the buffer holding the public key.</td>
</tr>
<tr>
<td>cert_priv_key_len</td>
<td>uint16_t</td>
<td>The length of the buffer holding the private key.</td>
</tr>
<tr>
<td>cert_block_len</td>
<td>uint32_t</td>
<td>The length of the buffer holding the X.509 certificate.</td>
</tr>
<tr>
<td>p_cert_pub_key</td>
<td>uint8_t *</td>
<td>A pointer to the buffer with the DER-encoded public key.</td>
</tr>
<tr>
<td>p_cert_priv_key</td>
<td>uint8_t *</td>
<td>A pointer to the buffer with the DER-encoded private key.</td>
</tr>
<tr>
<td>p_cert_block</td>
<td>uint8_t *</td>
<td>A pointer to the buffer with the DER-encoded X.509 certificate.</td>
</tr>
<tr>
<td>cert_type</td>
<td>uint8_t</td>
<td>The type of certificate.</td>
</tr>
</tbody>
</table>

Notification Codes

The callback notification definitions are as follows:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_NTF_HANDSHAKE_FAILED</td>
<td>0x10000</td>
<td>TLS handshake failed.</td>
</tr>
<tr>
<td>TLS_NTF_HANDSHAKE_DONE</td>
<td>0x20000</td>
<td>TLS handshake succeeded.</td>
</tr>
<tr>
<td>TLS_NTF_HANDSHAKE_TIMEOUT</td>
<td>0x40000</td>
<td>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.</td>
</tr>
<tr>
<td>DTLS_NTF_TIMEOUT</td>
<td>0x80000</td>
<td>DTLS retransmit timeout.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP_NTF_TX_RDY</td>
<td>0x00000002</td>
<td>Ready to transmit.</td>
</tr>
</tbody>
</table>
t_tls_conn_inf

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

<table>
<thead>
<tr>
<th>Element</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tci_timeout</td>
<td>uint32_t</td>
<td>The connect timeout.</td>
</tr>
<tr>
<td>p_tci_ntf</td>
<td>t_ip_ntf *</td>
<td>The notify structure.</td>
</tr>
<tr>
<td>tci_flags</td>
<td>uint8_t</td>
<td>The connection flags (described below).</td>
</tr>
<tr>
<td>p_tci_peer_name</td>
<td>char_t *</td>
<td>The peer name.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_TCP_CONN_INF_FLAG_START</td>
<td>0x00001</td>
<td>If this is set the TLS/DTLS handshake is executed immediately after a connection. If this is not set you must call <code>tls_start_tcp()</code> or <code>dtls_start_udp()</code> to start the handshake process.</td>
</tr>
<tr>
<td>TLS_TCP_CONN_INF_FLAG_VERIFY</td>
<td>0x00002</td>
<td>If this is set the TLS stack verifies the client peer. This is only used for the TLS server (in the function <code>tls_accept_tcp()</code>). Note: This value is not currently supported.</td>
</tr>
<tr>
<td>TLS_TCP_CONN_INF_FLAG_DIS_REN</td>
<td>0x00004</td>
<td>Set this to disable renegotiation for the given connection.</td>
</tr>
<tr>
<td>TLS_TCP_CONN_INF_FLAG_REN_FATAL</td>
<td>0x00008</td>
<td>Set this to to treat &quot;renegotiation not supported&quot; as failure. If this is set, renegotiation failure (peer sends ALERT_NO_RENEGOTIATION) leads to a connection FATAL error (closing of the connection). In other cases such an error stops renegotiation without an error.</td>
</tr>
</tbody>
</table>
t_dtls_conn_data

The \texttt{t_dtls_conn_data} structure is used to encapsulate parameters used by DTLS connections.

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

t_tls_ecc_curve_dsc

The structure `t_tls_ecc_curve_dsc` is used to configure an elliptic curve.

<table>
<thead>
<tr>
<th>Element</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
ecl_curve_id         | uint16_t           | The curve ID.                      |
ecl_curve_oid [TLS_ECC_CURVE_OID_MAX_SIZE] | uint8_t | An array of Object Identifiers (OIDs). |
ecl_curve_oid_len    | uint8_t            | The curve's OID length.            |
t_tls_cipher_suite

The *t_tls_cipher_suite* structure defines a cipher suite, as follows.

<table>
<thead>
<tr>
<th>Element</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_cs_cert</td>
<td>t_tls_certificate *</td>
<td>A pointer to the certificate associated with the cipher suite (may be NULL for client side).</td>
</tr>
<tr>
<td>cs_oid[TLS_OID_SIZE]</td>
<td>uint8_t</td>
<td>The signature/hash algorithm OID.</td>
</tr>
<tr>
<td>cs_id</td>
<td>uint16_t</td>
<td>The cipher suite identifier.</td>
</tr>
<tr>
<td>cs_named_curve</td>
<td>uint16_t</td>
<td>The name of the curve if this cipher suite uses ECDH\ECDHE. If these are not used set it to 0.</td>
</tr>
<tr>
<td>cs_mac_key_len</td>
<td>uint16_t</td>
<td>The length of the key used for MAC generation.</td>
</tr>
<tr>
<td>cs_mac_len</td>
<td>uint16_t</td>
<td>The length of the MAC.</td>
</tr>
<tr>
<td>cs_iv_len</td>
<td>uint16_t</td>
<td>The length of the initialization vector used for bulk encryption/decryption.</td>
</tr>
<tr>
<td>cs_bulk_key_len</td>
<td>uint16_t</td>
<td>The length of the key used for bulk encryption/decryption.</td>
</tr>
<tr>
<td>cs_hash_idx</td>
<td>uint8_t</td>
<td>The index in the arr_hash_alg.</td>
</tr>
<tr>
<td>cs_sign_idx</td>
<td>uint8_t</td>
<td>The index in the arr_sign_alg.</td>
</tr>
<tr>
<td>cs_keyex_idx</td>
<td>uint8_t</td>
<td>The index in the arr_sign_alg of the key exchange algorithm.</td>
</tr>
<tr>
<td>cs_bulk_idx</td>
<td>uint8_t</td>
<td>The index in the arr_bulk_alg.</td>
</tr>
</tbody>
</table>
## t_tls_alert_type

The `t_tls_alert_type` typedef defines the TLS alert types:

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_ALERT_CLOSE_NOTIFY</td>
<td>0</td>
<td>The connection was closed.</td>
</tr>
<tr>
<td>TLS_ALERT_UNEXPECTED_MESSAGE</td>
<td>10</td>
<td>Unexpected message received.</td>
</tr>
<tr>
<td>TLS_ALERT_BAD_RECORD_MAC</td>
<td>20</td>
<td>Bad MAC in record.</td>
</tr>
<tr>
<td>TLS_ALERT_DECRYPTION_FAILED_RESERVED</td>
<td>21</td>
<td>Decryption failed.</td>
</tr>
<tr>
<td>TLS_ALERT_RECORD_OVERFLOW</td>
<td>22</td>
<td>Overflow in record.</td>
</tr>
<tr>
<td>TLS_ALERT_DECOMPRESSION_FAILURE</td>
<td>30</td>
<td>Packet decompression failed.</td>
</tr>
<tr>
<td>TLS_ALERT_HANDSHAKE_FAILURE</td>
<td>40</td>
<td>Handshake failed.</td>
</tr>
<tr>
<td>TLS_ALERT_NO_CERTIFICATE_RESERVED</td>
<td>41</td>
<td>No certificate has been reserved.</td>
</tr>
<tr>
<td>TLS_ALERT_BAD_CERTIFICATE</td>
<td>42</td>
<td>The certificate is invalid.</td>
</tr>
<tr>
<td>TLS_ALERT_UNSUPPORTED_CERTIFICATE</td>
<td>43</td>
<td>The certificate type is not supported.</td>
</tr>
<tr>
<td>TLS_ALERT_CERTIFICATE_REVOKED</td>
<td>44</td>
<td>The certificate has been revoked.</td>
</tr>
<tr>
<td>TLS_ALERT_CERTIFICATE_EXPIRED</td>
<td>45</td>
<td>The certificate has expired.</td>
</tr>
<tr>
<td>TLS_ALERT_CERTIFICATE_UNKNOWN</td>
<td>46</td>
<td>The certificate is not known.</td>
</tr>
<tr>
<td>TLS_ALERT_ILLEGAL_PARAMETER</td>
<td>47</td>
<td>A parameter is invalid.</td>
</tr>
<tr>
<td>TLS_ALERT_UNKNOWN_CA</td>
<td>48</td>
<td>The Certificate Authority is not known.</td>
</tr>
<tr>
<td>TLS_ALERT_ACCESS_DENIED</td>
<td>49</td>
<td>Access denied.</td>
</tr>
<tr>
<td>TLS_ALERT.DecodeErr</td>
<td>50</td>
<td>Error during decoding.</td>
</tr>
<tr>
<td>TLS ALERT_DECIPHER_ERR</td>
<td>51</td>
<td>Error during decryption.</td>
</tr>
<tr>
<td>TLS_ALERT_EXPORT_RESTRICTION_RESERVED</td>
<td>60</td>
<td>Export restriction.</td>
</tr>
<tr>
<td>TLS_ALERT_PROTOCOL_VERSION</td>
<td>70</td>
<td>The protocol version is invalid.</td>
</tr>
<tr>
<td>TLS_ALERT_INSUFFICIENT_SECURITY</td>
<td>71</td>
<td>Insufficient security available.</td>
</tr>
<tr>
<td>TLS_ALERT_INTERNAL_ERR</td>
<td>80</td>
<td>Internal error.</td>
</tr>
<tr>
<td>TLS_ALERT_USER_CANCELLED</td>
<td>90</td>
<td>The user cancelled the connection.</td>
</tr>
<tr>
<td>Element</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>TLS_ALERT_NO_RENEGOTIATION</td>
<td>100</td>
<td>No renegotiation was possible.</td>
</tr>
<tr>
<td>TLS_ALERT_UNSUPPORTED_EXTENSION</td>
<td>110</td>
<td>The extension is not supported.</td>
</tr>
</tbody>
</table>

**t_tls_conn_status**

The `t_tls_conn_status` typedef defines the TCP connection states:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_CONNST_CLOSED</td>
<td>The connection is closed.</td>
</tr>
<tr>
<td>TLS_CONNST_HANDSHAKE</td>
<td>Handshake is in progress.</td>
</tr>
<tr>
<td>TLS_CONNST_OPERATING</td>
<td>The connection is established.</td>
</tr>
</tbody>
</table>

**t_cert_type**

The `t_cert_type` typedef defines the certificate types:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_CERT_HOST</td>
<td>The connection is closed.</td>
</tr>
<tr>
<td>TLS_CERT_CA</td>
<td>Handshake is in progress.</td>
</tr>
<tr>
<td>TLS_CERT_RV</td>
<td>The connection is established.</td>
</tr>
</tbody>
</table>
## Flags

The flags are as follows:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_TCP_CONN_INF_FLAG_START</td>
<td>0x0001</td>
<td>Set this to start a connection immediately. (Only applies for TLS TCP and DTLS UDP interfaces.)</td>
</tr>
<tr>
<td>TLS_TCP_CONN_INF_FLAG_VERIFY</td>
<td>0x0002</td>
<td>Set this to verify the client (this is only used in accept commands).</td>
</tr>
<tr>
<td>TLS_TCP_CONN_INF_FLAG_DIS_REN</td>
<td>0x0004</td>
<td>Set this to disable renegotiation for a given connection.</td>
</tr>
<tr>
<td>TLS_TCP_CONN_INF_FLAG_REN_FATAL</td>
<td>0x0008</td>
<td>Set this to treat &quot;renegotiation not supported&quot; as a failure. If this is set, renegotiation failure (peer sends ALERT_NO_RENEGOTIATION) will lead to a connection FATAL error (closing of connection). In other cases such an error stops renegotiation without an error.</td>
</tr>
</tbody>
</table>
7 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.

7.1 OS Abstraction Layer

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:

<table>
<thead>
<tr>
<th>OAL Resource</th>
<th>Number Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tasks</td>
<td>1</td>
</tr>
<tr>
<td>Mutexes</td>
<td>2</td>
</tr>
<tr>
<td>Events</td>
<td>1</td>
</tr>
</tbody>
</table>
7.2 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 HCC Base Platform Support Package User Guide.

The module makes use of the following standard PSP functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Package</th>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>psp_getcurrenttimedate()</code></td>
<td>psp_base</td>
<td>psp_rtc</td>
<td>Returns the current date and time. This is used for date- and time-stamping files.</td>
</tr>
<tr>
<td><code>psp_getrand()</code></td>
<td>psp_base</td>
<td>psp_rand</td>
<td>Generates a random number.</td>
</tr>
<tr>
<td><code>psp_timedatecmp()</code></td>
<td>psp_base</td>
<td>psp_rtc</td>
<td>Compares the current date/time with a specified date/time.</td>
</tr>
<tr>
<td><code>psp_malloc()</code></td>
<td>psp_base</td>
<td>psp_alloc</td>
<td>Allocates a block of memory, returning a pointer to the beginning of the block.</td>
</tr>
<tr>
<td><code>psp_free()</code></td>
<td>psp_base</td>
<td>psp_alloc</td>
<td>De-allocates a block of memory allocated by <code>psp_malloc()</code>, making it available for further allocation.</td>
</tr>
<tr>
<td><code>psp_memcmp()</code></td>
<td>psp_base</td>
<td>psp_string</td>
<td>Compares two blocks of memory.</td>
</tr>
<tr>
<td><code>psp_memcpy()</code></td>
<td>psp_base</td>
<td>psp_string</td>
<td>Copies a block of memory. The result is a binary copy of the data.</td>
</tr>
<tr>
<td><code>psp_memmove()</code></td>
<td>psp_base</td>
<td>psp_string</td>
<td>Moves a block of memory from one location to another. The two areas of memory may overlap without this causing problems as a temporary intermediate array is used.</td>
</tr>
<tr>
<td><code>psp_memset()</code></td>
<td>psp_base</td>
<td>psp_string</td>
<td>Sets the specified area of memory to the defined value.</td>
</tr>
<tr>
<td><code>psp_strncpy()</code></td>
<td>psp_base</td>
<td>psp_string</td>
<td>Copies one string of defined length to another.</td>
</tr>
<tr>
<td><code>psp_strncmp()</code></td>
<td>psp_base</td>
<td>psp_string</td>
<td>Compares two strings of defined length.</td>
</tr>
<tr>
<td><code>psp_strnlen()</code></td>
<td>psp_base</td>
<td>psp_string</td>
<td>Gets the length of a string.</td>
</tr>
</tbody>
</table>
The module makes use of the following PSP sockets functions from the `psp_sockets` package:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>psp_socket_init()</code></td>
<td>Initializes the PSP sockets module.</td>
</tr>
<tr>
<td><code>psp_socket_start()</code></td>
<td>Starts the PSP sockets module.</td>
</tr>
<tr>
<td><code>psp_socket_stop()</code></td>
<td>Stops the PSP sockets module.</td>
</tr>
<tr>
<td><code>psp_socket_delete()</code></td>
<td>Deletes the PSP sockets module, freeing any associated resources.</td>
</tr>
<tr>
<td><code>psp_sc_ioctl()</code></td>
<td>Accesses the FIONBIO and FIONREAD commands.</td>
</tr>
<tr>
<td><code>psp_sc_recv()</code></td>
<td>Receives data over a TCP connection.</td>
</tr>
<tr>
<td><code>psp_sc_send()</code></td>
<td>Sends data over a TCP connection.</td>
</tr>
<tr>
<td><code>psp_sc_poll()</code></td>
<td>Waits until either timeout or any of the events specified by the call occurs.</td>
</tr>
<tr>
<td><code>psp_sc_select()</code></td>
<td>Waits until one of the following occurs:</td>
</tr>
<tr>
<td></td>
<td>• Data is received on any socket in the <code>p_readfds</code> set.</td>
</tr>
<tr>
<td></td>
<td>• Data is transmitted on any socket in the <code>p_writefds</code> set.</td>
</tr>
<tr>
<td></td>
<td>• An exception event occurs on any socket in the <code>p_exceptfds</code> set.</td>
</tr>
</tbody>
</table>
The module makes use of the following standard PSP macros:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Package</th>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSP_RD_BE16</td>
<td>psp_base</td>
<td>psp_endianness</td>
<td>Reads a 16 bit value stored as big-endian from a memory location.</td>
</tr>
<tr>
<td>PSP_RD_BE24</td>
<td>psp_base</td>
<td>psp_endianness</td>
<td>Reads a 24 bit value stored as big-endian from a memory location.</td>
</tr>
<tr>
<td>PSP_RD_BE32</td>
<td>psp_base</td>
<td>psp_endianness</td>
<td>Reads a 32 bit value stored as big-endian from a memory location.</td>
</tr>
<tr>
<td>PSP_RD_BE48</td>
<td>psp_base</td>
<td>psp_endianness</td>
<td>Reads a 48 bit value stored as big-endian from a memory location.</td>
</tr>
<tr>
<td>PSP_WR_BE16</td>
<td>psp_base</td>
<td>psp_endianness</td>
<td>Writes a 16 bit value to be stored as big-endian to a memory location.</td>
</tr>
<tr>
<td>PSP_WR_BE24</td>
<td>psp_base</td>
<td>psp_endianness</td>
<td>Writes a 24 bit value to be stored as big-endian to a memory location.</td>
</tr>
<tr>
<td>PSP_WR_BE32</td>
<td>psp_base</td>
<td>psp_endianness</td>
<td>Writes a 32 bit value to be stored as big-endian to a memory location.</td>
</tr>
<tr>
<td>PSP_WR_BE48</td>
<td>psp_base</td>
<td>psp_endianness</td>
<td>Writes a 48 bit value to be stored as big-endian to a memory location.</td>
</tr>
<tr>
<td>PSP_WR_BE64</td>
<td>psp_base</td>
<td>psp_endianness</td>
<td>Writes a 64 bit value to be stored as big-endian to a memory location.</td>
</tr>
</tbody>
</table>

Collaboration Diagrams

The five collaboration diagrams in this section show:

- All the components and their interactions.
- The client/server relationship.

Note: To improve clarity, the module management functions are omitted.
TLS Native TCP Interface

TLS Client App
- tls_start_tcp()
- tls_close_tcp()
- tls_client_handshake_tcp()
- tls_tcp_connect()
- tls_send_tcp()
- tls_receive_tcp()
- tls_rx_ready_tcp()
- tls_get_buffer_tcp()
- tls_get_ticket_tcp()
- tls_get_state_tcp()

TCP connection

TLS Server App
- tls_start_tcp()
- tls_close_tcp()
- tls_server_handshake_tcp()
- tls_tcp_accept()
- tls_send_tcp()
- tls_receive_tcp()
- tls_rx_ready_tcp()
- tls_get_buffer_tcp()
- tls_get_ticket_tcp()
- tls_get_state_tcp()

TLS Library
- tcp_connect()
- tcp_accept()
- tcp_send()
- tcp_recv()
- tcp_connection_state()
- tcp_get_buffer()
- tcp_rx_ready()
- psp_getcurrenttime_tdate()
- psp_getrand()

TCP Native Interface
- PSP_RD_BE16
- PSP_RD_BE24
- PSP_RD_BE32
- PSP_WR_BE16
- PSP_WR_BE24
- PSP_WR_BE32
- PSP_WR_BE64

Endianness
- enc_driver_alloc()
- enc_driver_free()
- enc_driver_encrypt()
- enc_driver_decrypt()
- enc_driver_hash()
- enc_driver_start()
- enc_driver_stop()
DTLS Native UDP Interface

TLS Client App
- dtls_start_udp()
- dtls_close_udp()
- dtls_connect_udp()
- dtls_send_udp()
- dtls_receive_udp()
- dtls_get_buffer_udp()
- dtls_get_ticket_udp()
- dtls_get_state_udp()

TCP Native Interface
- udp_get_buffer()
- udp_open()
- udp_close()
- udp_rx_ready()
- udp_send()
- udp_receive()

TCP connection

TLS Library
- psp_get_current_time()

TLS Server App
- dtls_start_udp()
- dtls_close_udp()
- dtls_connect_srv_open()
- dtls_udp_srv_close()
- dtls_get_srv_conn_udp()
- dtls_send_udp()
- dtls_receive_udp()
- dtls_get_buffer_udp()
- dtls_get_ticket_udp()
- dtls_get_state_udp()

Real Time Clock (RTC)
- psp_getrand()

Random Number Generator
- psp_gettimeofday()

PSP Types
- psp_malloc()
- psp_free()

Alloc
- psp_memcpy()
- psp_memcmp()
- psp_memmove()
- psp_strcmp()
- psp_memset()

String library

Endianness
- enc_driver_alloc()
- enc_driver_free()
- enc_driver_encrypt()
- enc_driver_decrypt()
- enc_driver_hash()
- enc_driver_start()
- enc_driver_stop()

OS Abstraction Layer (OAL)
- oal_mutex_get()
- oal_mutex_put()
- oal_event_create()
- oal_event_get()
- oal_task_create()
- oal_task_delete()
- oal_event_set()

Embedded Encryption Manager (EEM)

Encryption/Hash algorithms
**TLS Sockets Interface**

- **TLS Client App**
  - tls_client_handshake_socket()
  - tls_send_socket()
  - tls_receive_socket()
  - tls_select_socket()
  - tls_poll_socket()
  - tls_get_ticket_socket()
  - tls_get_state_socket()
  - tls_close_socket()

- **TLS Server App**
  - tls_server_handshake_socket()
  - tls_send_socket()
  - tls_receive_socket()
  - tls_select_socket()
  - tls_poll_socket()
  - tls_get_ticket_socket()
  - tls_get_state_socket()
  - tls_close_socket()

- **TLS Library**
  - psp_getcurrenttimedate()
  - psp_getrand()
  - psp_malloc()
  - psp_free()
  - psp_memcpy()
  - psp_mmemmove()
  - psp_memset()
  - psp_strncpy()

- **Socket Library**
  - socket_select()
  - socket_poll()
  - socket_send()
  - socket_recv()
  - socket_ioctl()
  - socket_geterno()

- **Endianness**
  - enc_driver_alloc()
  - enc_driver_free()
  - enc_driver_encrypt()
  - enc_driver_decrypt()
  - enc_driver_hash()
  - enc_driver_start()
  - enc_driver_stop()

- **Real Time Clock (RTC)**
  - psp_getcurrenttimedate()

- **Random Number Generator**
  - psp_getrand()

- **PSP Types**
  - psp_malloc()
  - psp_free()
  - psp_memcpy()
  - psp_mmemmove()
  - psp_memset()
  - psp_strncpy()

- **Alloc**
  - psp_malloc()
  - psp_free()
  - psp_memcpy()
  - psp_mmemmove()
  - psp_memset()
  - psp_strncpy()

- **String library**
  - psp_malloc()
  - psp_free()
  - psp_memcpy()
  - psp_mmemmove()
  - psp_memset()
  - psp_strncpy()

- **OS Abstraction Layer (OAL)**
  - oal_task_create()
  - oal_task_delete()
  - oal_event_create()
  - oal_event_get()
  - oal_event_set()

- **Embedded Encryption Manager (EEM)**

- **Encryption/Hash algorithms**
TLS Raw Interface

RAW Interface
Client
- tls_init_raw()
- tls_close_raw()
- tls_read_data_raw()
- tls_hdl_data_raw()
- tls_start_raw()
- tls_get_ticket_raw()
- tls_get_state_raw()
- tls_get_read_data_raw()
- tls_read_data_ack_raw()
- tls_get_secrets_raw()
- tls_get_randoms_raw()

RAW Interface
Server
- tls_init_raw()
- tls_close_raw()
- tls_read_data_raw()
- tls_hdl_data_raw()
- tls_start_raw()
- tls_get_ticket_raw()
- tls_get_state_raw()
- tls_get_read_data_raw()
- tls_read_data_ack_raw()
- tls_get_secrets_raw()
- tls_get_randoms_raw()

TLS Library
- psp_getcurrenttimedate()
- psp_getrand()

Real Time Clock (RTC)

Random Number Generator

PSP Types

Optional
- psp_malloc()
- psp_free()

Alloc

- psp_memcpy()
- psp_memmove()
- psp_memset()
- psp_strcmp()

String library

OS Abstraction Layer (OAL)

Embedded Encryption Manager (EEM)

Encryption/Hash algorithms

External Certificate Store
- t_tls_cert_get_cb()
- t_tls_cert_valid_cb()

Endianness
- enc_driver_alloc()
- enc_driver_free()
- enc_driver_encrypt()
- enc_driver_decrypt()
- enc_driver_hash()
- enc_driver_start()
- enc_driver_stop()
8 Sample Code

This section gives example code for the module.

8.1 Server Application

The following examples show each of the server interfaces using native TCP, UDP and 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 TLS connections.

2. When the TCP connection with a client is established, complete the server side TLS handshake. The handshake procedure is executed by the TLS connection task. The user application is notified by a callback when the handshake ends.

3. Send receive notifications to user applications. Requests incoming from clients are read in the loop using the `tls_receive_tcp()` function and handled appropriately.

The following example shows pseudocode implementing a TLS server as an echo port. The connection is closed when the server receives more than 100 bytes. The functions this uses are as follows:

- `tls_demo_tcp_socket_conn()` – handles the handshake and, when the handshake ends, echoes incoming data.
- `tls_demo_tcp_check_state_close()` - closes the connection if the peer disconnects or the user sets `b_force_close` to TRUE.
- `tls_demo_task()` – task function that handles the TLS connection.
- `tls_demo_init()` – initializes the task and opens the server port.
```c
#include "../../../config/config_tls.h"
#include "../../../psp/include/psp_types.h"
#include "../../../api/api_tls.h"
#include "../../../api/api_ip.h"
#include "../../../api/api_ip_tcp.h"
#include "../../../oal/oal_task.h"
#include "../../../oal/oal_event.h"

#define LISTEN_PORT 4433
#define RCV_CNT_CLOSE 100
#define TLS_DEMO_EVENT 0x1

static void tls_demo_ntf_fn ( uint32_t param, uint32_t ntf );
OAL_TASK_FN( tls_demo_task );
static t_tls_ret tls_demo_server_tcp_conn( void );
static uint8_t tls_demo_tcp_check_state_close( uint8_t b_force_close );

static t_tcp_port_hdl g_tls_demo_port_hdl = TCP_INVALID_PORT_HDL;
static t_tcp_conn_hdl g_tls_demo_conn_hdl = TCP_INVALID_CONN_HDL;
static uint16_t g_rcv_cnt;
static uint8_t g_state;
static oal_task_t g_tls_demo_task_hdl;
static oal_task_id_t g_tls_demo_task_id;
static oal_event_t g_tls_demo_event;
static t_ip_ntf g_tls_demo_ntf = { tls_demo_ntf_fn, 0U, NULL };
OAL_TASK_DSC( "TLS demo Task"
    , tls_demo_task
    , OAL_LOWEST_PRIORITY
    , 2048 );

static void tls_demo_ntf_fn ( uint32_t param, uint32_t ntf )
{
    (void)param;
    /* Notify if received new data or test in client mode */
    if ( (( ntf & IP_NTF_RX_RDY ) != 0U ) || (( ntf & TLS_NTF_HANDSHAKE_DONE ) != 0U ) )
    {
        (void)oal_event_set( &g_tls_demo_event
            , TLS_DEMO_EVENT
            , g_tls_demo_task_id );
    }
}

static t_tls_ret tls_demo_server_tcp_conn( void )
{
    uint8_t * p_rx_buf;  /* Pointer to received buffer */
    uint16_t rx_bytes;  /* Number of received data */
    t_ip_ret ip_ret;    /* IP stack return value */
    uint32_t unread_data; /* Unread data */
    t_tls_ret tls_res;   /* TLS return value */
    oal_event_flags_t event_flags; /* Set event flags */
    oal_ret_t oal_ret;  /* Event get return value */

    tls_res = TLS_WAIT;
```
```c
oal_ret = oal_event_get( &g_tls_demo_event,
                        TLS_DEMO_EVENT,
                        &event_flags,
                        0);

if ( ( oal_ret == OAL_SUCCESS ) && ( event_flags == TLS_DEMO_EVENT ) )
{
    /* TLS connection is established */
    rx_bytes = 0;
    p_rx_buf = NULL;

    ip_ret = tls_rx_ready_tcp( g_tls_demo_conn_hdl, &unread_data );
    while ( ( ip_ret == IP_SUCCESS ) && ( unread_data > 0 ) )
    {
        tls_res = tls_receive_tcp( g_tls_demo_conn_hdl, &p_rx_buf, &rx_bytes );
        if ( rx_bytes > 0 )
        {
            g_rcv_cnt += rx_bytes;
            if ( tls_send_tcp( g_tls_demo_conn_hdl, p_rx_buf, rx_bytes ) != TLS_OK )
            {
                tcp_release_buf( p_rx_buf );
            }
        }
        ip_ret = tls_rx_ready_tcp( g_tls_demo_conn_hdl, &unread_data );
    }
    return tls_res;
}

static uint8_t tls_demo_tcp_check_state_close( uint8_t b_force_close )
{
    uint8_t ret_val;
    t_tls_conn_status state;

    ret_val = FALSE;
    state = tls_get_state_tcp( g_tls_demo_conn_hdl );

    if ( ( state == TLS_CONNST_CLOSED ) || ( b_force_close == TRUE ) )
    {
        ret_val = TRUE;
        if ( state != TLS_CONNST_CLOSED )
        {
            (void) tls_close_tcp( g_tls_demo_conn_hdl );
        }
        if ( tcp_disconnect( g_tls_demo_conn_hdl ) >= 0 )
        {
            g_tls_demo_conn_hdl = TCP_INVALID_CONN_HDL;
        }
    }
    return ret_val;
}
```
OAL_TASK_FN( tls_demo_task )
{
    uint8_t    b_close;
    t_tls_conn_inf inf;
    t_tls_ret   tls_res;
    t_ip_port   ip_rx_port;

    OAL_TASK_PRE;

#if OAL_TASK_POLL_MODE == 0
    ip_enter_task();
    for ( ; ; )
#endif
{
    switch (g_state)
    {
    case 0:
        inf.p_tci_ntf = &g_tls_demo_ntf;
        inf.p_tci_peer_name = NULL;
        inf.tci_flags = TLS_TCP_CONN_INF_FLAG_START;
        inf.tci_timeout = IP_WAIT_FOREVER;

        tls_res = tls_tcp_accept( g_tls_demo_port_hdl,
                                &inf,
                                &ip_rx_port,
                                &g_tls_demo_conn_hdl );

        if ( tls_res == TLS_OK )
        {
            g_state++;
        }
        break;

    case 1:
        if ( tls_demo_server_tcp_conn() == TLS_WAIT )
        {
            if ( g_rcv_cnt > RCV_CNT_CLOSE )
            {
                b_close = TRUE;
            }
            else
            {
                b_close = FALSE;
            }

            if ( tls_demo_tcp_check_state_close( b_close ) == TRUE )
            {
                g_rcv_cnt = 0;
                g_state = 0;
            }
        }
        break;

    case 2:
        break;
    }
};
);
}
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 TLS handshake. The handshake procedure is executed by the TLS 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 `dtls_receive_udp()` and handle these appropriately.

The following example shows pseudocode implementing a DTLS server as an echo port. The connection is closed when the server receives more than 100 bytes. The functions this uses are as follows:

- `dtls_demo_server_udp_conn()` – handles the handshake and, when the handshake ends, echoes incoming data.
- `dtls_demo_udp_check_state_close()` - closes the connection if the peer disconnects or the user sets `b_force_close` to TRUE.
- `tls_demo_task()` – task function that handles the TLS connection.
- `tls_demo_init()` – initializes the task and opens the server port.
```c
#include "../../../config/config_tls.h"
#include "../../../psp/include/psp_types.h"
#include "../../../api/api_tls.h"
#include "../../../api/api_ip.h"
#include "../../../api/api_ip_tcp.h"
#include "../../../oal/oal_task.h"
#include "../../../oal/oal_event.h"

#define LISTEN_PORT 4433
#define RCV_CNT_CLOSE 100
#define TLS_DEMO_EVENT 0x1

static void tls_demo_ntf_fn ( uint32_t param, uint32_t ntf );
OAL_TASK_FN( tls_demo_task );
static t_tls_ret dtls_demo_server_udp_conn( void );
static uint8_t dtls_demo_udp_check_state_close( uint8_t b_force_close );

static t_udp_hdl g_tls_demo_hdl = UDP_INVALID_HDL;
static t_dtls_hdl g_tls_demo_conn_hdl = DTLS_INVALID_UDP_HDL;
static uint16_t g_rcv_cnt;
static uint8_t g_state;
static oal_task_t g_tls_demo_task_hdl;
static oal_task_id_t g_tls_demo_task_id;
static oal_event_t g_tls_demo_event;
static t_ip_ntf g_tls_demo_ntf =
{
    tls_demo_ntf_fn
 , 0U
 , NULL
};

OAL_TASK_DSC( "TLS demo Task"
 , tls_demo_task
 , OAL_LOWEST_PRIORITY
 , 2048 );

static void tls_demo_ntf_fn ( uint32_t param, uint32_t ntf )
{
    (void)param;
    /* Notify if received new data or test in client mode */
    if ( ( ( ntf & IP_NTF_RX_RDY ) != 0U )
        || ( ( ntf & TLS_NTF_HANDSHAKE_DONE ) != 0U ) )
    {
        (void)oal_event_set( &g_tls_demo_event
            , TLS_DEMO_EVENT
            , g_tls_demo_task_id );
    }
}

static t_tls_ret dtls_demo_server_udp_conn( void )
{

```
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 */

tls_res = TLS_WAIT;

oal_ret = oal_event_get( &g_tls_demo_event
 , TLS_DEMO_EVENT
 , &event_flags
 , 0 );

if ( ( oal_ret == OAL_SUCCESS ) && ( event_flags == TLS_DEMO_EVENT ) )
{
    if ( g_tls_demo_conn_hdl == DTLS_INVALID_UDP_HDL )
    {
        oal_ret = dtls_get_srv_conn_udp( g_tls_demo_hdl, &g_tls_demo_conn_hdl );
    }

    rx_bytes = 0;
    tls_res = dtls_receive_udp( g_tls_demo_conn_hdl, &p_rx_buf, &rx_bytes );    /* For native API buffer size is 0 */
    if ( rx_bytes > 0 )
    {
        g_rcv_cnt += rx_bytes;
        if ( dtls_send_udp( g_tls_demo_conn_hdl, p_rx_buf, rx_bytes ) != TLS_OK )
        {
            udp_release_buf( p_rx_buf );
        }
    }
}

return tls_res;

static uint8_t dtls_demo_udp_check_state_close( uint8_t b_force_close )
{
    uint8_t ret_val;
    t_tls_conn_status state;

    ret_val = FALSE;
    state = dtls_get_state_udp( g_tls_demo_conn_hdl );

    if ( ( ( state == TLS_CONNST_CLOSED )
        && ( g_tls_demo_conn_hdl != DTLS_INVALID_UDP_HDL ) )
        || ( b_force_close == TRUE ) )
    {
        ret_val = TRUE;
        if ( state != TLS_CONNST_CLOSED )
        {
            (void) dtls_close_udp( g_tls_demo_conn_hdl );
        }
    }

    g_tls_demo_conn_hdl = DTLS_INVALID_UDP_HDL;
OAL_TASK_FN( tls_demo_task )
{
    uint8_t        b_close;
    t_tls_conn_inf inf;
    t_tls_ret      tls_res;
OAL_TASK_PRE;

#if OAL_TASK_POLL_MODE == 0
#if ( TLS_TCPIP_IFC_ENABLE != 0 ) || ( DTLS_UDP_IFC_ENABLE != 0 ) || ( TLS_USE_STD_SOCKET == 0)
    ip_enter_task();
#endif
for ( ; ; )
#endif
{
    switch(g_state)
    {
        case 0:
            inf.p_tci_ntf = &g_tls_demo_ntf;
            inf.p_tci_peer_name = NULL;
            inf.tci_flags = TLS_TCP_CONN_INF_FLAG_START;

            if ( g_tls_demo_hdl == UDP_INVALID_HDL )
                /* Get server only once */
            {
#if OAL_PREEMPTIVE == 1
                oal_task_sleep( 6500 );
#endif
                tls_res = dtls_udp_srv_open( LISTEN_PORT
                                           , &inf
                                           , &g_tls_demo_hdl );
            }
            else
            {
                tls_res = TLS_OK;
            }
            g_tls_demo_conn_hdl = DTLS_INVALID_UDP_HDL;
            if ( tls_res == TLS_OK )
            {
                g_state++;
            }
            break;
        case 1:
            if ( dtls_demo_server_udp_conn() == TLS_WAIT )
            {
                if ( g_rcv_cnt > RCV_CNT_CLOSE )
                {
                    b_close = TRUE;
                }
            }
            else
            {

            }

}


```c
{
    b_close = FALSE;
}

if ( dtls_demo_udp_check_state_close( b_close ) == TRUE )
{
    g_rcv_cnt = 0;
    g_state = 0;
}
break;
case 2:
    break;
}

uint32_t tls_demo_init ( void )
{
    uint32_t ret_val; /* Return value (0 for success) */
    oal_ret_t ret_otc_evt; /* Event create return value */

    g_state = 0;
    g_rcv_cnt = 0;
    ret_val = 1; /* Non zero value is an error */

    ret_otc_evt = oal_event_create( &( g_tls_demo_event ) );
    if ( ret_otc_evt == OAL_SUCCESS )
    {
        if ( oal_task_create( &g_tls_demo_task_hdl,
            &tls_demo_task_dsc,
            &g_tls_demo_task_id ) == OAL_SUCCESS )
        {
            ret_val = 0;
        }
    }
    else
    {
        if ( ret_val == 1 )
        {
            if ( ret_otc_evt == OAL_SUCCESS )
            {
                oal_event_delete( &( g_tls_demo_event ) );
            }
        }
        return ret_val;
    }
}
```
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 that implements a TLS server as an echo port. The connection is closed when the server receives more than 100 bytes.

The functions this uses are the following:

- `tls_demo_server_socket()` - opens the TCP port for incoming connections.
- `tls_demo_server_socket_listen()` – listens for incoming connections.
- `tls_demo_server_socket_conn()` – handles the handshake and, when the handshake ends, echoes incoming data.
- `tls_demo_socket_check_state_close()` - closes the connection if the peer disconnects or the user sets `b_force_close` to TRUE.
- `tls_demo_task()` - task function handling the TLS connection.
- `tls_demo_init()` – initializes the task and opens the server port.
#include "../../../config/config_tls.h"
#include "../../../psp/include/psp_types.h"
#include "../../../api/api_tls.h"
#include "../../../api/api_ip_socket.h"
#include "../../../oal/oal_task.h"

#define BUFER_SIZE 1024
#define LISTEN_PORT 4433
#define RCV_CNT_CLOSE 100

OAL_TASK_FN(tls_demo_task);
static uint8_t tls_demo_server_socket_listen(void);
static uint8_t tls_demo_server_socket(void);
static t_tls_ret tls_demo_server_socket_conn(void);
static uint8_t tls_demo_socket_check_state_close(uint8_t b_force_close);

static int g_sd_svr = SOCKET_ERROR;
static int g_sd_conn = SOCKET_ERROR;
static uint8_t g_rcv_buf[BUFER_SIZE];
static uint16_t g_rcv_cnt;
static uint8_t g_state;
static oal_task_t g_tls_demo_task_hdl;
static oal_task_id_t g_tls_demo_task_id;
OAL_TASK_DSC("TLS demo Task",
    tls_demo_task,
    OAL_LOWEST_PRIORITY,
    2048);

static uint8_t tls_demo_server_socket(void)
{
    uint8_t  tls_res;
    struct sockaddr_in  sa_addr;

    tls_res = TRUE;
    if ( (g_sd_svr = socket_open(AF_INET, SOCK_STREAM, 0) ) == -1)
    {
        tls_res = FALSE;
    }
    else
    {
        psp_memset((char *)&sa_addr, 0, sizeof(sa_addr));
        (void)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 = socket_htons(LISTEN_PORT);
        if ( socket_bind(g_sd_svr, (struct sockaddr *)&sa_addr, sizeof(sa_addr)) != 0 )
        {
            tls_res = FALSE;
        }
        else
        {
            if (socket_listen( g_sd_svr, 4 ) < 0)
            {
                tls_res = FALSE;
            }
if ( tls_res != TRUE )
{
    socket_close(g_sd_svr);
}

return tls_res;

static uint8_t tls_demo_server_socket_listen( void )
{
    uint8_t  tls_res;
    struct sockaddr_in  sa_peer_addr;
    socklen_t addr_len;

    addr_len = sizeof( sa_peer_addr );

    g_sd_conn = socket_accept( g_sd_svr, (struct sockaddr *)&sa_peer_addr, &addr_len );
    if ( g_sd_conn == SOCKET_ERROR )
    {
        tls_res = FALSE;
    }
    else
    {
        tls_res = TRUE;
    }

    return tls_res;
}

static t_tls_ret tls_demo_server_socket_conn( void )
{
    uint16_t   recv_bytes;
    int         sock_ret;
    uint8_t * p_buf;
    t_tls_ret  tls_res;
    struct     timeval timeout;
    t_fd_set   rx_ready_set; /* RX ready socket set */
    /* First establish connection by handshake process */
#if OAL_TASK_POLL_MODE == 0
    do
    {
        #endif
        tls_res = tls_server_handshake_socket( g_sd_conn, FALSE, NULL );
        #if OAL_TASK_POLL_MODE == 0
        oal_task_sleep( 100 );
        #endif
        while ( tls_res == TLS_WAIT );
    #endif

    if ( tls_res == TLS_OK )
    {
        /* TLS connection is established. For native API buffer size is 0 */
timeout.tv_sec = 0;
timeout.tv_usec = 500000;
FD_ZERO( &rx_ready_set );
FD_SET( g_sd_conn, &rx_ready_set );
sock_ret = tls_select_socket(g_sd_conn+1, &rx_ready_set, NULL, NULL, &timeout);
if ( sock_ret > 0 )
{
    recv_bytes = 0;
p_buf = g_rcv_buf;
tls_res = tls_receive_socket( g_sd_conn, &p_buf, BUFFER_SIZE, &recv_bytes );
    if ( recv_bytes > 0 )
    {
        g_rcv_cnt += recv_bytes;
        (void)tls_send_socket( g_sd_conn, g_rcv_buf, recv_bytes );
    }
}
else
{
    tls_res = TLS_WAIT;
}
return tls_res;
}

static uint8_t tls_demo_socket_check_state_close( uint8_t b_force_close )
{
    uint8_t ret_val;
t_tls_conn_status state;

    ret_val = FALSE;
    state = tls_get_state_socket( g_sd_conn );

    if ( ( state == TLS_CONNST_CLOSED )
    || ( b_force_close == TRUE ) )
    {
        ret_val = TRUE;
        if ( state != TLS_CONNST_CLOSED )
        {
            (void)tls_close_socket( g_sd_conn );
        }

        if ( socket_close( g_sd_conn ) >= 0 )
        {
            g_sd_conn = SOCKET_ERROR;
        }
    }

    return ret_val;
}

OAL_TASK_FN( tls_demo_task )
{
    uint8_t b_close;
}
OAL_TASK_PRE;

#ifdef OAL_TASK_POLL_MODE == 0
    ip_enter_task();
    for ( ; ; )
#endif
{
    switch (g_state)
    {
        case 0:
            if ( tls_demo_server_socket_listen() == TRUE )
            {
                g_state++;
            }
            break;
        case 1:
            if ( tls_demo_server_socket_conn() == TLS_WAIT )
            {
                if ( g_rcv_cnt > RCV_CNT_CLOSE )
                {
                    b_close = TRUE;
                }
                else
                {
                    b_close = FALSE;
                }

                if ( tls_demo_socket_check_state_close( b_close ) == TRUE )
                {
                    g_rcv_cnt = 0;
                    g_state = 0;
                }
            }
            break;
        case 2:
            break;
    }
}

uint32_t tls_demo_init ( void )
{
    uint32_t ret_val;

    g_state = 0;
    g_rcv_cnt = 0;
    ret_val = 1; /* Non zero value is an error */

    if ( tls_demo_server_socket() == TRUE )
    {
        if ( oal_task_create( &g_tls_demo_task_hdl ,
                               &tls_demo_task_dsc ,
                               &g_tls_demo_task_id ) == OAL_SUCCESS )

            ret_val = 0; /* OK */
    }

    return ret_val;
}
{ ret_val = 0;
} else
{
    socket_close( g_sd_svr );
}

return ret_val;
} /* tls_test_init */
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 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 `dtls_receive_socket()` and handle these appropriately.

The following example shows pseudocode implementing a DTLS server as an echo port. The connection is closed when the server receives more than 100 bytes.

The functions this uses are the following:

- `dtls_demo_server_socket()` – opens a TCP port for incoming connections.
- `dtls_demo_server_socket_conn()` – handles the handshake and, when the handshake ends, echoes incoming data.
- `dtls_demo_socket_check_state_close()` - closes the connection if the peer disconnects or the user sets `b_force_close` to TRUE.
- `tls_demo_task()` – task function that handles the TLS connection.
- `tls_demo_init()` – initializes the task and opens the server port.
#include "../../../config/config_tls.h"
#include "../../../psp/include/psp_types.h"
#include "../../../api/api_tls.h"
#include "../../../api/api_ip_socket.h"
#include "../../../oal/oal_task.h"

#define BUFER_SIZE 1024
#define LISTEN_PORT 4433
#define RCV_CNT_CLOSE 100

OAL_TASK_FN( tls_demo_task );
static uint8_t dtls_demo_server_socket( void );
static t_tls_ret dtls_demo_server_socket_conn( void );
static uint8_t dtls_demo_socket_check_state_close( uint8_t b_force_close );

static int g_sd_svr = SOCKET_ERROR;
static t_dtls_hdl g_sdconn;
static uint8_t g_rcv_buf[BUFER_SIZE];
static uint16_t g_rcv_cnt;
static uint8_t g_state;
static oal_task_t g_tls_demo_task_hdl;
static oal_task_id_t g_tls_demo_task_id;
OAL_TASK_DSC( "TLS demo Task"
, tls_demo_task
, OAL_LOWEST_PRIORITY
, 2048 );

uint8_t dtls_demo_server_socket( void )
{
    uint8_t tls_res;
    struct sockaddr_in sa_addr;

    tls_res = TRUE;
    if ( (g_sd_svr = socket_open(AF_INET, SOCK_DGRAM, 0)) == -1 )
    {
        tls_res = FALSE;
    }
    else
    {
        psp_memset((char *)&sa_addr, 0, sizeof(sa_addr));
        (void) 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 = socket_htons(LISTEN_PORT);
        if ( (socket_bind(g_sd_svr, (struct sockaddr *)&sa_addr, sizeof(sa_addr)) != 0 )
        {
            tls_res = FALSE;
        }
    }

    if ( tls_res != TRUE )
    {
        socket_close(g_sd_svr);
    }
}
return tls_res;
}

static t_tls_ret dtls_demo_server_socket_conn( void )
{
    uint8_t   * p_buf;
    uint16_t   recv_bytes;
    unsigned long avail_data_len;
    int       sock_ret;
    t_tls_ret  tls_res;

    /* First establish connection by handshake process */
    #if OAL_TASK_POLL_MODE == 0
    do
    {
        #endif
        tls_res = dtls_server_handshake_socket( g_sd_svr, FALSE, NULL, &g_sd_conn );
        #if OAL_TASK_POLL_MODE == 0
        } while ( ( tls_res == TLS_WAIT ) ||  ( tls_res == TLS_DTLS_NEW_CONN ) );
        oal_task_sleep( 100 );
    #endif

    if ( tls_res == TLS_OK )
    {
        /* TLS connection is established */
        recv_bytes = 0;
        p_buf = g_rcv_buf;
        avail_data_len = 0;
        sock_ret = socket_ioctl( g_sd_svr, FIONREAD, &avail_data_len );
        if ( ( sock_ret == SOCKET_SUCCESS ) && ( avail_data_len > 0u ) )
        {
            tls_res = dtls_receive_socket( g_sd_conn, &p_buf, BUFER_SIZE, &recv_bytes );
            if ( recv_bytes > 0 )
            {
                g_rcv_cnt += recv_bytes;
                (void)dtls_send_socket( g_sd_conn, p_buf, recv_bytes );
            }
        }
    }
    else
    {
        tls_res = TLS_WAIT;
    }
}

return tls_res;
}

static uint8_t dtls_demo_socket_check_state_close( uint8_t b_force_close )
{
    uint8_t ret_val;
    t_tls_conn_status state;

    ret_val = FALSE;
    state = dtls_get_state_socket( g_sd_conn );
    if ( ( ( state == TLS_CONNST_CLOSED )
        && ( g_sd_conn != DTLS_INVALID_UDP_HDL ) )
        || ( b_force_close == 1 )
        && ( g_sd_conn == DTLS_INVALID_UDP_HDL ) )
|| ( b_force_close == TRUE ) )
{
  ret_val = TRUE;
  if ( state != TLS_CONNST_CLOSED )
  {
    (void) dtls_close_socket( g_sd_conn );
  }
  g_sd_conn = DTLS_INVALID_UDP_HDL;
}

return ret_val;
}

OAL_TASK_FN( tls_demo_task )
{
  uint8_t b_close;
  OAL_TASK_PRE;

#if OAL_TASK_POLL_MODE == 0
  ip_enter_task();
  for ( ; ; )
#endif
{
  if ( dtls_demo_server_socket_conn() == TLS_WAIT )
  {
    if ( g_rcv_cnt > RCV_CNT_CLOSE )
    {
      b_close = TRUE;
    }
    else
    {
      b_close = FALSE;
    }
    if ( dtls_demo_socket_check_state_close( b_close ) == TRUE )
    {
      g_rcv_cnt = 0;
      g_state = 0;
    }
  }
}

uint32_t tls_demo_init ( void )
{
  uint32_t ret_val;

  g_state = 0;
  g_rcv_cnt = 0;
  ret_val = 1; /* Non zero value is an error */
  g_sd_conn = DTLS_INVALID_UDP_HDL;

  if ( dtls_demo_server_socket() == TRUE )
  {

if ( oal_task_create( &g_tls_demo_task_hdl
        , &tls_demo_task_dsc
        , &g_tls_demo_task_id ) == OAL_SUCCESS )
{
    ret_val = 0;
}
else
{
    socket_close( g_sd_svr );
}

return ret_val;
} /* tls_test_init */
8.2 Client Application

The following examples show each of the client interfaces using TCP, UDP, and Sockets.

TLS Client Interface using Native TCP

The main steps the TLS client implementation must take are:

1. Establish the TCP 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 implementing a TLS client as an echo port. The connection is closed when the client receives more than 100 bytes of data.

The functions this uses are the following:

- `tls_demo_client_tcp_conn()` - handles the handshake and, when the handshake ends, echoes incoming data.
- `tls_demo_tcp_check_state_close()` - closes the connection if the peer disconnects or the user sets `b_force_close` to TRUE.
- `tls_demo_task()` - task function that handles the TLS connection.
- `tls_demo_init()` - initializes the task and opens the server port.
#include "../../../config/config_tls.h"
#include "../../../config/config_psp_types.h"
#include "../../../psp/include/psp_types.h"
#include "../../../psp/include/psp_endianness.h"
#include "../../../api/api_tls.h"
#include "../../../api/api_ip.h"
#include "../../../api/api_ip_tcp.h"
#include "../../../oal/oal_task.h"
#include "../../../oal/oal_event.h"

#define SERVER_PORT 4433
#define RCV_CNT_CLOSE 100
#define TLS_DEMO_EVENT 0x1
#define SERVER_ADDR 0xC0A800CA /* 192.168.0.202 */
#define TLS_DEMO_PEERNAME "HCC demo" /* Peer name */

static void tls_demo_ntf_fn ( uint32_t param, uint32_t ntf );
OAL_TASK_FN( tls_demo_task );
static t_tls_ret tls_demo_client_tcp_conn( void );
static uint8_t tls_demo_tcp_check_state_close( uint8_t b_force_close );

static t_tcp_conn_hdl g_tls_demo_conn_hdl = TCP_INVALID_CONN_HDL;
static uint16_t g_rcv_cnt;
static uint8_t g_state;
static oal_task_t g_tls_demo_task_hdl;
static oal_task_id_t g_tls_demo_task_id;
static oal_event_t g_tls_demo_event;
static t_ip_ntf g_tls_demo_ntf =
{
    tls_demo_ntf_fn,
    0U,
    NULL
};

OAL_TASK_DSC( "TLS demo Task",
    tls_demo_task,
    OAL_LOWEST_PRIORITY,
    2048 );

static void tls_demo_ntf_fn ( uint32_t param, uint32_t ntf )
{
    (void)param;
    /* Notify if received new data or test in client mode */
    if ( !(( ntf & TLS_NTF_HANDSHAKE_DONE ) != 0U) )
    {
        (void)oal_event_set( &g_tls_demo_event,
                               TLS_DEMO_EVENT,
                               g_tls_demo_task_id );
    }
}

static t_tls_ret tls_demo_client_tcp_conn( void )
```c
{  
  uint8_t   * p_rx_buf;
  uint16_t    rx_bytes;
  t_ip_ret    ip_ret;    /* IP stack return value */
  uint32_t    unread_data; /* Unread data */
  t_tls_ret   tls_res;
  oal_event_flags_t  event_flags;       /* Set event flags */
  oal_ret_t    oal_ret;       /* Event get return value */

  tls_res = TLS_WAIT;

  oal_ret = oal_event_get( &g_tls_demo_event 
                             , TLS_DEMO_EVENT 
                             , &event_flags 
                             , 0);

  if ( ( oal_ret == OAL_SUCCESS ) && ( event_flags == TLS_DEMO_EVENT ) )
  {  /* TLS connection is established */
    rx_bytes = 0;
    p_rx_buf = NULL;

    ip_ret = tls_rx_ready_tcp( g_tls_demo_conn_hdl, &unread_data );
    while ( ( ip_ret == IP_SUCCESS ) && ( unread_data > 0 ) )
    {
      tls_res = tls_receive_tcp( g_tls_demo_conn_hdl, &p_rx_buf, &rx_bytes );
      if ( rx_bytes > 0 )
      {
        g_rcv_cnt += rx_bytes;
        if ( tls_send_tcp( g_tls_demo_conn_hdl, p_rx_buf, rx_bytes ) != TLS_OK )
        {
          tcp_release_buf( p_rx_buf );
        }
      }
      ip_ret = tls_rx_ready_tcp( g_tls_demo_conn_hdl, &unread_data );
    }
    return tls_res;
  }

static uint8_t tls_demo_tcp_check_state_close( uint8_t b_force_close )
{
  uint8_t ret_val;
  t_tls_conn_status state;

  ret_val = FALSE;
  state = tls_get_state_tcp( g_tls_demo_conn_hdl );

  if ( ( state == TLS_CONNST_CLOSED )
      || ( b_force_close == TRUE ) )
  {
    ret_val = TRUE;
    if ( state != TLS_CONNST_CLOSED )
    {
```
{  
  (void) tls_close_tcp(g_tls_demo_conn_hdl);
}

if ( tcp_disconnect(g_tls_demo_conn_hdl) >= 0 )
{
  g_tls_demo_conn_hdl = TCP_INVALID_CONN_HDL;
}

return ret_val;
}

OAL_TASK_FN(tls_demo_task)
{
  uint8_t    b_close;
  t_tls_conn_inf    inf;
  t_tls_ret      tls_res;
  t_ip_port      ip_tx_port;

  OAL_TASK_PRE;

#if OAL_TASK_POLL_MODE == 0
  ip_enter_task();
  for ( ; ; )
#endif
{
  switch(g_state)
  {
  case 0:
    ip_tx_port.ipp_port    = SERVER_PORT;
    ip_tx_port.ipp_ip_addr.ipa_version = IPV_IP_V4;
    PSP_WR_BE32(ip_tx_port.ipp_ip_addr.ipa_address, SERVER_ADDR);
    inf.p_tci_ntf = &g_tls_demo_ntf;
    inf.p_tci_peer_name = TLS_DEMO_PEERNAME;
    inf.tci_flags = TLS_TCP_CONN_INF_FLAG_START;
    inf.tci_timeout = IP_WAIT_FOREVER;
    tls_res = tls_tcp_connect(&ip_tx_port, &inf, TLS_INVALID_CONN_TICKET, &g_tls_demo_conn_hdl);
    if ( tls_res == TLS_OK )
    {
      g_state++;
    }
    break;
  case 1:
    if ( tls_demo_client_tcp_conn() == TLS_WAIT )
    {
      if ( g_rcv_cnt > RCV_CNT_CLOSE )
      {
        b_close = TRUE;
      }
    } else


```c
{  
    b_close = FALSE;
}

if ( tls_demo_tcp_check_state_close( b_close ) == TRUE )
{
    g_rcv_cnt = 0;
    g_state = 0;
}
break;
case 2:
    break;
}
}

uint32_t tls_demo_init ( void )
{
    uint32_t ret_val;  /* return value (0 for success) */
oal_ret_t ret_otcEvt; /* event create return value */

g_state = 0;
g_rcv_cnt = 0;
ret_val = 1;  /* Non zero value is an error */

ret_otcEvt = oal_event_create( &( g_tls_demo_event ) );
if ( ret_otcEvt == OAL_SUCCESS )
{
    if ( oal_task_create( &g_tls_demo_task_hdl,
        &tls_demo_task_dsc,
        &g_tls_demo_task_id ) == OAL_SUCCESS )
    {
        ret_val = 0;
    }
}
if ( ret_val == 1 )
{
    if ( ret_otcEvt == OAL_SUCCESS )
    {
        oal_event_delete( &( g_tls_demo_event ) );
    }
    return ret_val;
}
```
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 TLS handshake. This means calling `dtls_connect_udp()` and waiting for the `TLS_NTF_HANDSHAKE_DONE` notification.
3. Start to send and receive data.

The following example shows pseudocode implementing a DTLS client as an echo port. The connection is closed when the client receives more than 100 bytes of data.

The functions this uses are the following:

- `dtls_demo_client_udp_conn()` – handles the handshake and, when the handshake ends, echoes incoming data.
- `dtls_demo_udp_check_state_close()` - closes the connection if the peer disconnects or the user sets `b_force_close` to TRUE.
- `tls_demo_task()` – task function that handles the TLS connection.
- `tls_demo_init()` – initializes the task and opens the server port.
#include "../../../config/config_tls.h"
#include "../../../psp/include/psp_types.h"
#include "../../../psp/include/psp_endianness.h"
#include "../../../api/api_tls.h"
#include "../../../api/api_ip.h"
#include "../../../api/api_ip_tcp.h"
#include "../../../oal/oal_task.h"
#include "../../../oal/oal_event.h"

#define SERVER_PORT 4433
#define RCV_CNT_CLOSE 100
#define TLS_DEMO_EVENT 0x1
#define SERVER_ADDR 0xC0A800CA /* 192.168.0.202 */
#define TLS_DEMO_PEERNAME "HCC demo" /* Peer name */

static void tls_demo_ntf_fn ( uint32_t param, uint32_t ntf );
OAL_TASK_FN( tls_demo_task );
static t_tls_ret dtls_demo_client_udp_conn( void );
static uint8_t dtls_demo_udp_check_state_close( uint8_t b_force_close );

static t_dtls_hdl g_tls_demo_conn_hdl = DTLS_INVALID_UDP_HDL;
static uint16_t g_rcv_cnt;
static uint8_t g_state;
static oal_task_t g_tls_demo_task_hdl;
static oal_task_id_t g_tls_demo_task_id;
static oal_event_t g_tls_demo_event;
static t_ip_ntf g_tls_demo_ntf =
{
  tls_demo_ntf_fn, 0U, NULL
};
OAL_TASK_DSC( "TLS demo Task"
  , tls_demo_task
  , OAL_LOWEST_PRIORITY
  , 2048 );

static void tls_demo_ntf_fn ( uint32_t param, uint32_t ntf )
{
  (void)param;
  /* notify if received new data or test in client mode */
  if ( ( ( ntf & IP_NTF_RX_RDY ) != 0U )
      || ( ( ntf & TLS_NTF_HANDSHAKE_DONE ) != 0U ) )
  {
    (void)oal_event_set( &g_tls_demo_event
      , TLS_DEMO_EVENT
      , g_tls_demo_task_id );
  }
}

static t_tls_ret dtls_demo_client_udp_conn( void )
{
  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 */
tls_res = TLS_WAIT;

oal_ret = oal_event_get( &g_tls_demo_event
                        , TLS_DEMO_EVENT
                        , &event_flags
                        , 0 );

if ( ( oal_ret == OAL_SUCCESS ) && ( event_flags == TLS_DEMO_EVENT ) )
{/ * TLS connection is established */
  rx_bytes = 0;
p_rx_buf = NULL;
tls_res = dtls_receive_udp( g_tls_demo_conn_hdl, &p_rx_buf, &rx_bytes );
  if ( rx_bytes > 0 )
  {
    g_rcv_cnt += rx_bytes;
    if( dtls_send_udp( g_tls_demo_conn_hdl, p_rx_buf, rx_bytes ) != TLS_OK )
    {
      udp_release_buf( p_rx_buf );
    }
  }
  return tls_res;
}

static uint8_t dtls_demo_udp_check_state_close( uint8_t b_force_close )
{
  uint8_t ret_val;
t_tls_conn_status state;

  ret_val = FALSE;
  state = dtls_get_state_udp( g_tls_demo_conn_hdl );

  if ( ( ( state == TLS_CONNST_CLOSED )
        && ( g_tls_demo_conn_hdl != DTLS_INVALID_UDP_HDL ) )
       || ( b_force_close == TRUE ) )
  {
    ret_val = TRUE;
    if ( state != TLS_CONNST_CLOSED )
    {
      (void) dtls_close_udp( g_tls_demo_conn_hdl );
    }
    g_tls_demo_conn_hdl = DTLS_INVALID_UDP_HDL;
  }
  return ret_val;
}

OAL_TASK_FN( tls_demo_task )
{
  uint8_t b_close;
}
t_tls_conn_inf inf;
t_tls_ret tls_res;
t_ip_port ip_tx_port;

OAL_TASK_PRE;

#if OAL_TASK_POLL_MODE == 0
ip_enter_task();
for (; ; )
#endif{
switch(g_state){
    case 0:
        ip_tx_port.ipp_port = SERVER_PORT;
        ip_tx_port.ipp_ip_addr.ipa_version = IPV_IP_V4;
        PSP_WR_BE32(ip_tx_port.ipp_ip_addr.ipa_address, SERVER_ADDR);
        inf.p_tci_ntf = &g_tls_demo_ntf;
        inf.p_tci_peer_name = TLS_DEMO_PEERNAME;
        inf.tci_flags = TLS_TCP_CONN_INF_FLAG_START;
        inf.tci_timeout = IP_WAIT_FOREVER;
        tls_res = dtls_connect_udp(&ip_tx_port, &inf, TLS_INVALID_CONN_TICKET, &g_tls_demo_conn_hdl);
        if ( tls_res == TLS_OK )
            g_state++;
        break;
    case 1:
        if ( dtls_demo_client_udp_conn() == TLS_WAIT )
        {
            if ( g_rcv_cnt > RCV_CNT_CLOSE )
                b_close = TRUE;
            else
                b_close = FALSE;
        }
        else
        {
            b_close = FALSE;
        }
        if ( dtls_demo_udp_check_state_close( b_close ) == TRUE )
        {
            g_rcv_cnt = 0;
            g_state = 0;
        }
        break;
    case 2:
        break;
}
}
uint32_t tls_demo_init ( void )
{
    uint32_t ret_val; /* return value (0 for success) */
    oal_ret_t ret_otc_evt; /* event create return value */

    g_state = 0;
    g_rcv_cnt = 0;
    ret_val = 1; /* non zero value is an error */

    ret_otc_evt = oal_event_create( &( g_tls_demo_event ) );
    if ( ret_otc_evt == OAL_SUCCESS )
    {
        if ( oal_task_create( &g_tls_demo_task_hdl
                               , &tls_demo_task_dsc
                               , &g_tls_demo_task_id ) == OAL_SUCCESS )
            { ret_val = 0; }
    }
    if ( ret_val == 1 )
    {
        if ( ret_otc_evt == OAL_SUCCESS )
            { oal_event_delete( &( g_tls_demo_event ) ); }
    }
    return ret_val;
}
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 implementing a TLS client as an echo port. The connection is closed when the client receives more than 100 bytes of data.

The functions this uses are the following:

- `tls_demo_client_socket()` – opens the TCP connection.
- `tls_demo_client_socket_conn()` – handles the handshake and, when the handshake ends, echoes incoming data.
- `tls_demo_socket_check_state_close()` - closes the connection if the peer disconnects or the user sets `b_force_close` to TRUE.
- `tls_demo_task()` – task function that handles the TLS connection.
- `tls_demo_init()` – initializes the task and opens the server port.
#include "../../../config/config_tls.h"
#include "../../../psp/include/psp_types.h"
#include "../../../api/api_tls.h"
#include "../../../api/api_ip_socket.h"
#include "../../../oal/oal_task.h"

#define BUFER_SIZE 1024
#define SERVER_PORT 4433
#define RCV_CNT_CLOSE 100
#define SERVER_ADDR 0xC0A800CA /* 192.168.0.202 */
#define TLS_DEMO_PEERNAME "HCC demo" /* Peer name */

OAL_TASK_FN( tls_demo_task );
static uint8_t tls_demo_client_socket( void );
static t_tls_ret tls_demo_client_socket_conn( void );
static uint8_t tls_demo_socket_check_state_close( uint8_t b_force_close );

static int g_sd_svr = SOCKET_ERROR;
static int g_sd_conn = SOCKET_ERROR;
static uint8_t g_rcv_buf[BUFER_SIZE];
static uint16_t g_rcv_cnt;
static uint16_t g_state;
static oal_task_t g_tls_demo_task_hdl;
static oal_task_id_t g_tls_demo_task_id;
OAL_TASK_DSC( "TLS demo Task"
    , tls_demo_task
    , OAL_LOWEST_PRIORITY
    , 2048 );

static uint8_t tls_demo_client_socket( void )
{
    t_tls_ret tls_res;
    struct sockaddr_in sa_addr;
    struct sockaddr * p_sock_addr;
    socklen_t sock_len;

    tls_res = TLS_OK;
    #if OAL_PREEMPTIVE == 1
    oal_task_sleep( 6500 );
    #endif
    if ( (g_sd_conn = socket_open(AF_INET, SOCK_STREAM, 0)) == -1 )
    {
        tls_res = TLS_WAIT;
    }
    else
    {
        (void)psp_memset( (char *)&sa_addr, 0, sizeof(sa_addr) );
        sa_addr.sin_family = AF_INET;
        sa_addr.sin_addr.s_addr = socket_htonl(SERVER_ADDR);
        sa_addr.sin_port = socket_htons(SERVER_PORT);
        p_sock_addr = (struct sockaddr *)&sa_addr;
        sock_len = sizeof(sa_addr);
        if ( socket_connect( g_sd_conn

{  
  tls_res = TLS_WAIT;
  socket_close(g_sd_conn);
}

return tls_res;
}

static t_tls_ret tls_demo_client_socket_conn( void )
{
  uint8_t   * p_buf;
  uint16_t   recv_bytes;
  t_tls_ret  tls_res;
  int   sock_ret;
  struct timeval timeout;
  t_fd_set  rx_ready_set;       /* RX ready socket set */
  
  /* First establish connection by handshake process */
  #if OAL_TASK_POLL_MODE == 0
  do
  {
    #endif
    tls_res = tls_client_handshake_socket( g_sd_conn, TLS_DEMO_PEERNAME,
                  TLS_INVALID_CONN_TICKET );
  #if OAL_TASK_POLL_MODE == 0
  } while ( tls_res == TLS_WAIT );
  #endif

  if ( tls_res == TLS_OK )
  { /* TLS connection is established */
   recv_bytes = 0;
    p_buf = g_rcv_buf;

    /* Check if packet was received and get available bytes */
    timeout.tv_sec = 0;
    timeout.tv_usec = 500000;

    FD_ZERO( &rx_ready_set );
    FD_SET( g_sd_conn, &rx_ready_set );

    sock_ret = tls_select_socket(g_sd_conn+1, &rx_ready_set, NULL, NULL, &timeout);
    if ( sock_ret > 0 )
    {
      tls_res = tls_receive_socket( g_sd_conn, p_buf, BUFER_SIZE, &recv_bytes );
      if ( recv_bytes > 0 )
      {
        g_rcv_cnt += recv_bytes;
        (void)tls_send_socket( g_sd_conn, p_buf, recv_bytes );
      }
    }
else
{
    tls_res = TLS_WAIT;
}
return tls_res;

static uint8_t tls_demo_socket_check_state_close( uint8_t b_force_close )
{
    uint8_t ret_val;
    t_tls_conn_status state;
    ret_val = FALSE;
    state = tls_get_state_socket( g_sd_conn );
    if ( ( state == TLS_CONNST_CLOSED )
        || ( b_force_close == TRUE ) )
    {
        ret_val = TRUE;
        if ( state != TLS_CONNST_CLOSED )
        {
            (void) tls_close_socket( g_sd_conn );
        }
        if ( socket_close( g_sd_conn ) >= 0 )
        {
            g_sd_conn = SOCKET_ERROR;
        }
    }
    return ret_val;
}

OAL_TASK_FN( tls_demo_task )
{
    uint8_t b_close;
    OAL_TASK_PRE;
    #if OAL_TASK_POLL_MODE == 0
    ip_enter_task();
    for ( ; ; )
    #endif
    {
        switch( g_state )
        {
            case 0:
            if ( tls_demo_client_socket() == TLS_OK )
            {
                g_state++;
            }
            break;
            case 1:
            if ( tls_demo_client_socket_conn() == TLS_WAIT )
            {
                if ( g_rcv_cnt > RCV_CNT_CLOSE )
                {
{  
    b_close = TRUE;
}
else
{
    b_close = FALSE;
}

if ( tls_demo_socket_check_state_close( b_close ) == TRUE )
{
    g_rcv_cnt = 0;
    g_state = 0;
    
    break;
}
}

uint32_t tls_demo_init ( void )
{
    uint32_t   ret_val;
    
    g_rcv_cnt = 0;
    g_state = 0;
    ret_val = 1; /* Non zero value is an error */

    if ( oal_task_create( &g_tls_demo_task_hdl 
                         , &tls_demo_task_dsc 
                         , &g_tls_demo_task_id ) == OAL_SUCCESS )
    {
        ret_val = 0;
    }

    return ret_val;
} /* tls_test_init */
DTLS Client Interface using Sockets

The main steps the DTLS client implementation must take are:

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

The following example shows pseudocode implementing a DTLS client as an echo port. The connection is closed when the client receives more than 100 bytes of data.

The functions this uses are the following:

- `dtls_demo_client_socket_conn()` – handles the handshake and, when the handshake ends, echoes incoming data.
- `dtls_demo_socket_check_state_close()` - closes the connection if the peer disconnects or the user sets `b_force_close` to TRUE.
- `tls_demo_task()` – task function that handles the TLS connection.
- `tls_demo_init()` – initializes the task and opens the server port.
#include "../../../config/config_tls.h"
#include "../../../psp/include/psp_types.h"
#include "../../../psp/include/psp_endianess.h"
#include "../../../api/api_tls.h"
#include "../../../api/api_ip_socket.h"
#include "../../../oal/oal_task.h"

#define BUFER_SIZE 1024
#define SERVER_PORT 4433
#define RCV_CNT_CLOSE 100
#define SERVER_ADDR 0xC0A800CA /* 192.168.0.202 */
#define TLS_DEMO_PEERNAME "HCC demo" /* Peer name */

OAL_TASK_FN( tls_demo_task );
static uint8_t dtls_demo_client_socket( void );
static t_tls_ret dtls_demo_client_socket_conn( void );
static uint8_t dtls_demo_socket_check_state_close( uint8_t b_force_close );

static int g_sd_conn = SOCKET_ERROR;
static t_dtls_hdl g_tls_demo_conn_hdl;
static uint8_t g_rcv_buf[BUFER_SIZE];
static uint8_t g_rcv_cnt;
static uint8_t g_state;
static oal_task_t g_tls_demo_task_hdl;
static oal_task_id_t g_tls_demo_task_id;
OAL_TASK_DSC( "TLS demo Task"
, tls_demo_task
, OAL_LOWEST_PRIORITY
, 2048);

uint8_t dtls_demo_client_socket( void )
{
  uint8_t  tls_res;
  struct sockaddr_in sa_addr;
  struct sockaddr * p_sock_addr;
  socklen_t sock_len;

  tls_res = TRUE;
  if ( g_sd_conn = socket_open(AF_INET, SOCK_DGRAM, 0)) == -1 )
  {
    tls_res = FALSE;
  }
  else
  {
    (void)pssp_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 = socket_htons(SERVER_PORT);
    p_sock_addr = (struct sockaddr *)&( sa_addr );
    sock_len = sizeof(sa_addr);
  }
g_tls_demo_conn_hdl = DTLS_INVALID_UDP_HDL;
if ( socket_bind(g_sd_conn, p_sock_addr, sock_len) != 0 )
{
    tls_res = FALSE;
    socket_close(g_sd_conn);
}
}
return tls_res;
}

static t_tls_ret dtls_demo_client_socket_conn( void )
{
    uint8_t   * p_buf;
    uint16_t   recv_bytes;
    t_tls_ret  tls_res;
    unsigned long avail_data_len;
    int         sock_ret;
    t_ip_port   ip_tx_port;

    ip_tx_port.ipp_port = SERVER_PORT;
    ip_tx_port.ipp_ip_addr.ipa_version = IPV_IP_V4;
PSP_WR_BE32(ip_tx_port.ipp_ip_addr.ipa_address, SERVER_ADDR);

    /* First establish connection by handshake process */
    #if OAL_TASK_POLL_MODE == 0
    do
    {  
    #endif
        tls_res = dtls_client_handshake_socket( g_sd_conn , &ip_tx_port , TLS_DEMO_PEERNAME , TLS_INVALID_CONN_TICKET , &g_tls_demo_conn_hdl );
    #if OAL_TASK_POLL_MODE == 0
    } while ( tls_res == TLS_WAIT );
    #endif

    if ( tls_res == TLS_OK )
    {
        /* TLS connection is established */
        recv_bytes = 0;
        p_buf = g_rcv_buf;
        /* Check if packet was received and get available bytes */
        avail_data_len = 0;
        sock_ret = socket_ioctl( g_sd_conn , FIONREAD , &avail_data_len );
        if ( ( sock_ret == SOCKET_SUCCESS ) && ( avail_data_len > 0u ) )
        {
            tls_res = dtls_receive_socket( g_tls_demo_conn_hdl , &p_buf , BUFER_SIZE , &recv_bytes );
            if ( recv_bytes > 0 )
            {
                g_rcv_cnt += recv_bytes;
                (void) dtls_send_socket( g_tls_demo_conn_hdl , p_buf , recv_bytes );
            }
        }
    }
}
} } 

else { 
  tls_res = TLS_WAIT; 
} 
return tls_res; 

static uint8_t dtls_demo_socket_check_state_close( uint8_t b_force_close ) {
  uint8_t ret_val; 
  t_tls_conn_status state; 

  ret_val = FALSE; 
  state = dtls_get_state_socket( g_tls_demo_conn_hdl ); 

  if ( ( ( state == TLS_CONNST_CLOSED ) 
        && ( g_tls_demo_conn_hdl != DTLS_INVALID_UDP_HDL ) ) 
     || ( b_force_close == TRUE ) ) 
    { 
      ret_val = TRUE; 
      if ( state != TLS_CONNST_CLOSED ) 
      { 
        (void) dtls_close_socket( g_tls_demo_conn_hdl ); 
      } 
      g_tls_demo_conn_hdl = DTLS_INVALID_UDP_HDL; 
      socket_close(g_sd_conn); 
      g_sd_conn = SOCKET_ERROR; 
    } 
  return ret_val; 
}

OAL_TASK_FN( tls_demo_task ) {
  uint8_t b_close; 

  OAL_TASK_PRE; 

  #if OAL_TASK_POLL_MODE == 0 
  ip_enter_task(); 
  for ( ; ; ) 
  #endif 
  switch(g_state) 
  { 
    case 0: 
      if ( dtls_demo_client_socket() == TRUE ) 
      { 
        g_state++; 
      } 
      break;
case 1:
    if ( dtls_demo_client_socket_conn() == TLS_WAIT )
    {
        if ( g_rcv_cnt > RCV_CNT_CLOSE )
            b_close = TRUE;
        else
            b_close = FALSE;
        if ( dtls_demo_socket_check_state_close( b_close ) == TRUE )
            g_rcv_cnt = 0;
            g_state = 0;
    }
    break;
}
}

uint32_t tls_demo_init ( void )
{
    uint32_t ret_val;
    g_state = 0;
    g_rcv_cnt = 0;
    ret_val = 1; /* Non zero value is an error */
    g_sd_conn = DTLS_INVALID_UDP_HDL;
    if ( oal_task_create( &g_tls_demo_task_hdl
                        , &tls Demo_task_dsc
                        , &g_tls_demo_task_id ) == OAL_SUCCESS )
        {ret_val = 0;
        }
    return ret_val;
} /* tls_test_init */