TO-Protect TLS User Manual

Release 6.1.2

March 16, 2022

Authored by *Trusted Objects*

Contents

1. Legal

1.1 Confidentiality

The information contained in these documents is only for the information of the intended recipient and may not be modified even partially or published without the prior written consent of Trusted Objects. It is provided without any guarantees or warranty on the product.

1.2 Property

These softwares, source codes, header files, and documentations are the entire property of Trusted-Objects and cant be copied or modified, even partially, without written Trusted-Objects agreement.

2. Licensed product

Two types of license are offered for:

- **Evaluation**: For TO-Protect TLS evaluation, a **free evaluation release** with the associated documentation can be downloaded on Trusted Objects web site. There is no maintenance included.
- **Production**: Trusted Objects will grant a non-exclusive, non-transferable yearly license. A Technology Access License (TAL) agreement will be put in place. The yearly license fee will include technical support and access to updates. All related materials (sources, binary library, etc.) are provided after the agreement signing.

3. TO-Protect TLS overview

3.1 Highlights

TO-Protect is a family of lightweight secure software libraries developed by Trusted Objects for constrained applications such as industrial IoT or embedded application where strong security and limited code footprint are required.

TO-Protect delivers higher security value on the application, by enabling security functions (cryptography, secure boot, anti-cloning,), secure storage and secure connectivity (LoRaWAN, TLS,).

TO-Protect can run on most generic MCU (ARM Cortex, RISC V) of the market. TO-Protect has been developed to address different use cases, including LoRaWAN, TLS, IP Protection.

In brief, TO Protect is a 100% software secure library for generic MCU providing the same functionalities as a Secure Element.

The characteristics of TO-Protect are:

- Based on state-of-the-art software security countermeasures
- Connectivity security already implemented for easier integration
- Reduced code size and fast execution time

With the following benefits:

- Prevent physical and logical security attacks on threatened embedded devices
- Easy to integrate, including for devices already in the field (assuming host MCU or RF SoC have FOTA capability)
- Compatible with existing hardware (no redesign)

This document describes **TO-Protect TLS**.

3.2 Key features

libTO and TO-Protect altogether implement a full TLS stack, including the following TLS security mechanisms:

- key generation
- keys secure storage
- mutual authentication
- integrity protection
- confidentiality

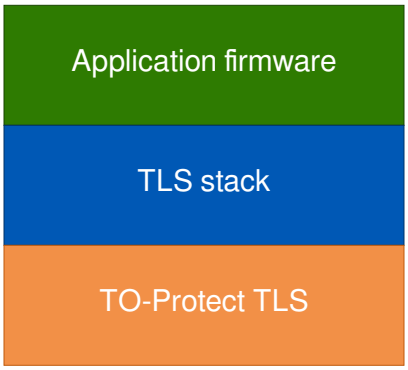
4. TO-Protect TLS integration

4.1 How to use TO-Protect TLS

If you are starting a new project, you can use *Trusted-Objects TLS stack* which is as optimized as possible in terms of footprint, and which contains a full TLS Stack. This is the easiest way to implement a firmware with a full TLS stack.

On projects already using Mbed TLS, you also have the choice to continue to use TLS stack directly. TLS stack will be adapted to use TO-Protect automatically (see Mbed TLS Application Note for details).

It has the advantage to avoid modification in your firmware, but the footprint will be higher.



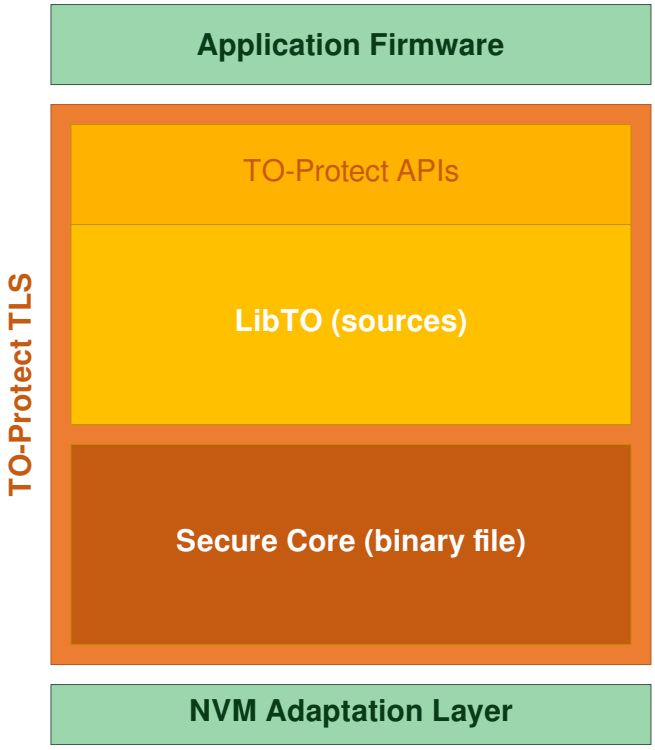
TO-Protect is designed to be able to run on standard MCUs compatible with standard C API. Dynamic allocation is not used by the library.

You can find in this documentation details about the library, installation and settings instructions, and API references.

4.2 Overall architecture

TO-Protect TLS consists of 2 parts:

- libTO library: provided as source code, it provides all TO-Protect TLS APIs (*TO-Protect TLS API*).
- TO-Protect TLS Secure Core: provided as binary file to be flashed at a memory location of your choice, it provides all TO-Protect TLS features.



Green parts above represent the customer software, while orange parts represent Trusted Objects deliveries.

The NVM adaptation layer must provide the functions to read/write/erase NVM on your platform. See *Secure storage*.

4.3 Library files tree

The libTO library files tree structure is the following:

- **/include:** headers providing library APIs, see *Provided API*
- **/src:** library sources
- **/examples:** some examples to use the library from your project

4.4 TO-Protect files tree

TO-Protect TLS Secure Core is delivered separately as a binary file **TO-Protect.bin**.

Along with TO-Protect.bin comes the header file **TOP_info.h**.

The libTO library includes TOP_info.h, which contains information about TO-Protect TLS memory footprint information (such as Flash size and NVM size). So you must ensure that its path is in the compiler include paths.

4.5 Footprint

Table 1: TO-Protect footprint examples

	Flash	Secure Storage	RAM	Stack
TO-Protect v2.1x.xx	75K	< 7K	< 8K (1)	< 4K

Note (1): TO-Protect does NOT perform any RAM allocation (static or dynamic). So this RAM static allocation is deferred to the libTO driver part, thanks to the values exposed in TOP_info.h.

Table 2: libTO footprint examples

	Flash	Secure Storage	RAM	Stack
libTO (driver excluded)	< 16K	N/A	< 1K+Helper(2)	< 0.5 K

Note (2): For TLS Helper buffer, see description of

- TOSE_HELPER_TLS_IO_BUFFER_SIZE (default: 2048)
- TOSE_HELPER_TLS_FLIGHT_BUFFER_SIZE (default: 2048)

So, to calculate the total RAM requested you must add libTO requirement above, plus the size reserved for these buffers (default will be + 4K).

These values have been measured on mbed-os-example-tls/tls-client (tag mbed-os-5.11.5) compiled with GNU Arm Embedded Toolchain Version 6-2017-q2-update.

Supported MCUs:

- ARM Cortex M0, M0+, M1, M3, M4 and M7,
- ARM Cortex-M23, M33, M35P and M55 with TrustZone®,
- other architectures available on demand, please contact us for more details.

4.6 Limitations

Your application must manage concurrent accesses to TO-Protect functions calls, and cumulative APIs sequences (init, update, final) must not be interrupted by another call.

5. TO-Protect TLS setup and configuration

5.1 Use TO-Protect TLS in an MCU project

In order to use TO-Protect TLS in an MCU project, please follow the integration instructions below.

Note: The following prerequisites are expected in this article:

- the ability to build C code for the target hardware

5.1.1 Reserve space to flash TO-Protect TLS

TO-Protect Secure Core is delivered separately as a binary file, **TO-Protect.bin**. You just need to reserve a free area in your Flash space, where **TO-Protect.bin** will be programmed.

Then you will need you enter this address value, either:

- in TODRV_SSE_cfg.h (see TODRV_SSE_TOP_ADDRESS), or
- on the compiler command line.

We recommend that no RW data are on the same sectors as **TO-Protect.bin**

5.1.2 Implement the HAL NVM

TO-Protect TLS relies on an NVM adaptation layer to read, write, and erase NVM area dedicated to TO-Protect TLSs secure storage.

See *NVM HAL implementation* for an implementation example.

5.1.3 Install TO-Protect TLS in your sources tree

TO-Protect TLS delivery contains the file **TOP_info.h**. This file must be placed in your project, at a place known in your project include path.

5.1.4 Install libTO in your sources tree

libTO source files must be added to your project, as detailed below.

The following directories are to be considered:

- **include**: header files, providing definitions and APIs
- **src**: implementation

There are two ways to install the library in your sources tree:

- you can simply put TO-Protect TLS folder in your sources tree, it will be easier to update it on future deliveries, by replacing the folder
- or you can integrate different TO-Protect TLS parts among your project files, for example copy the *include* directory content with your own headers, or *src* directory contents with your HAL APIs sources

From the *src* folder, the following files are to be included into your build process:

- **src/*.c** library files

5.1.5 Configure your project

Your build process needs some configurations to be able to build and use TO-Protect TLS.

5.1.5.1 Headers include paths

No matter the way you installed the library into your source tree, be sure its headers (the files provided in the *include* directory of the library) are accessible from one of your include path.

5.1.5.2 Preprocessor definitions

The **TO_LOG_LEVEL_MAX** preprocessor definition is available to set maximal logs level. Debug level prints out additional logs to help you debugging your application.

Read *Library configuration for an MCU project* for details on all available preprocessor definitions. This document also details endianness settings.

5.2 Library configuration for an MCU project

The library allows various settings with different granularity in order to customize global settings and select features to enable. These settings may be important, especially to minimize library memory usage.

Note: It is assumed you have read the library setup guide, *Use TO-Protect TLS in an MCU project*.

The settings below can be defined through preprocessor definitions from your build environment, or by editing the following files provided with library header files:

- *TO_cfg.h*: provides a way to configure libTO build
- *TODRV_SSE_cfg.h*: provides a way to configure the driver part of libTO
- *TOSE_helper_cfg.h*: provides a way to configure libTO helpers

5.2.1 User configuration file

It might be convenient to define your settings in your configuration file, outside the library tree, in order to isolate your configuration in a single file, and not having your configuration spread into various IDE menus.

For this, you can define the symbol *TO_USER_CONFIG*, in your IDE or on the command line.

When *TO_USER_CONFIG* is defined, the file *TO_user_config.h* will be included by the library.

5.2.2 Global settings

The following preprocessor definitions are available:

Table 1: Global MCU settings

Flag	Description
TO_LOG_LEVEL_MAX	Select maximal log level to compile (log level is also configureable at runtime with `TO_set_log_level()`): -1 (disabled), 0 (error), 1 (warning, default), 2 (info), 3 (debug)
TO_BIG_ENDIAN	Force big endian
TO_LITTLE_ENDIAN	Force little endian
HAVE_ENDIAN_H	Toolchain provides endian.h
HAVE_BYTESWAP_H	Toolchain provides byteswap.h
HAVE_NO_STDINT_H	Toolchain does not provide stdint.h
TO_USER_CONFIG	User provides file TO_user_config.h
TO_TLS_SESSIONS_NB	TLS sessions number (default: 2)
TOSE_HELPER_TLS_IO_BUFFER_SIZE	(expert) Customize internal TLS I/O buffer size, must be at least as big as biggest handshake message (defragmented, with handshake header, without record header) except messages containing certificates
TOSE_HELPER_TLS_RX_BUFFER_SIZE	(expert) Customize internal TLS I/O buffer size reserved for reception (default value: half of TOSE_HELPER_TLS_IO_BUFFER_SIZE)
TOSE_HELPER_TLS_FLIGHT_BUFFER_SIZE	(expert) Customize internal TLS flight buffer size, must be at least as big as biggest client flight (defragmented, with handshake header, without record header, adding 4 bytes per handshake message). Unused without DTLS retransmission feature.
TOSE_HELPER_TLS_RECEIVE_TIMEOUT	(expert) Customize internal TLS receive timeout

For the enable/disable flags, just define to enable the expected setting.

5.2.2.1 Endianness

If your target system build environment provides *endian.h* header file (defining functions such as *be32toh()* or *htobe32()*), you can just define the *HAVE_ENDIAN_H* preprocessor macro to 1. If your target system build environment provides *byteswap.h* header file (defining functions such as *___bswap16()* or *___bswap32()*), you can just define the *HAVE_BYTESWAP_H* preprocessor macro to 1. Else, endianness settings may be computed by the library from preprocessor pre-defined macros if available.

If previous solutions are not available, endianness is going to be detected at run time, when `TOSE_init()` function is called by client application.

In all cases, if you know your target endianness, you can force it by defining *TO_BIG_ENDIAN* or *TO_LITTLE_ENDIAN* preprocessor macros to 1 according to your architecture characteristics.

5.2.2.2 Integers (stdint)

If your target system does not provide *stdint.h* header file, you must define *HAVE_NO_STDINT_H* preprocessor macro to 1. The library will declare its needed integer declarations from *TO_stdint.h*.

5.2.3 Features settings

It may be interesting to only enable features required in order to minimize library memory usage.

5.2.3.1 Macroscopic settings

These settings are used to enable or disable large sets of features (macroscopic settings). The following preprocessor definitions are available:

Table 2: Macroscopic settings

Flag	Description
TO_DISABLE_TLS_STACK	TLS stack (default: enabled)
TO_DISABLE_TLS_HELPER	TLS handshake helper (default: enabled)
TO_ENABLE_DTLS	DTLS APIs (default: disabled)
TO_DISABLE_DTLS_RETRANSMISSION	DTLS retransmission (default: enabled)

Some features are disabled by default and enabled if the relevant flag is defined, the other ones are enabled by default and disabled by defining a flag.

The value of these flags does not matter, only the definition is taken into account.

5.2.3.2 Microscopic settings

These settings are used to enable or disable features with a per-API granularity (microscopic settings).

Every API has its own disable flag to tell compiler to not build the related function.

Disable flags have the following form: *TO_DISABLE_API_<API_NAME>*. For example, *get_serial_number()* API can be disabled by defining the *TO_DISABLE_API_GET_SERIAL_NUMBER* flag.

Some APIs can be disabled by groups:

- ***_init/update/final()** form APIs, as *sha256_init()*, *sha256_update()* and *sha256_final()*, which can be disabled by group using **TO_DISABLE_API_<API_NAME>_INIT_UPDATE_FINAL** definition
- **TLS** APIs
- **TLS Optimized** APIs

5.2.4 Memory settings

All the configuration needed to configure TO-Protect location in Flash memory, NVM address and NVM memory settings is shown in the table below.

These parameters can be set in the file named **TODRV_SSE_cfg.h**. Or you can define these precprocessor symbols in your IDE.

Table 3: Memory settings

Flag	Description
TODRV_SSE_TOP_ADDRESS	Base address for TO-Protect Secure Core in flash memory. This address MUST be word aligned

We recommend that no RW data are on the same sectors as **TO-Protect.bin**

5.3 Flashing TO-Protect TLS

5.3.1 Commands to convert toolchain generated binary

Depending on your toolchain, your tools can generate and handle output files with format ELF, bin, hex or S-REC.

TO-Protect is delivered as a binary file. Below are some commands that you can use to perform the conversions required to generate the file to flash.

- convert ELF to SREC:

```
arm-none-eabi-objcopy -v -O srec app.elf app.s19
```

- convert bin to SREC:

```
arm-none-eabi-objcopy -v -O srec -I binary TO-Protect.bin TO-Protect.s19
```

- SREC concatenation:

```
srec_cat app.s19 TO-Protect.s19 -offset <TO-Protect address, example: 0x8020000> --
↪line-length=46 -o full_fw.s19
```

- convert SREC to bin:

```
arm-none-eabi-objcopy -v -O binary -I srec full.s19 full.bin
```

- convert SREC to ELF:

```
arm-none-eabi-objcopy -v -O elf32-littlearm -I srec full.s19 full.elf
```

- insert TO-Protect into an existing ELF:

```
arm-none-eabi-objcopy --add-section .to-protect=TO-Protect.bin --set-section-flags .
↪to-protect=code,alloc,load,readonly --change-section-address .to-protect=<TO-
↪Protect address, example: 0x8020000> app.elf full_fw.elf
```

- insert TO-Protect into an existing AXF:

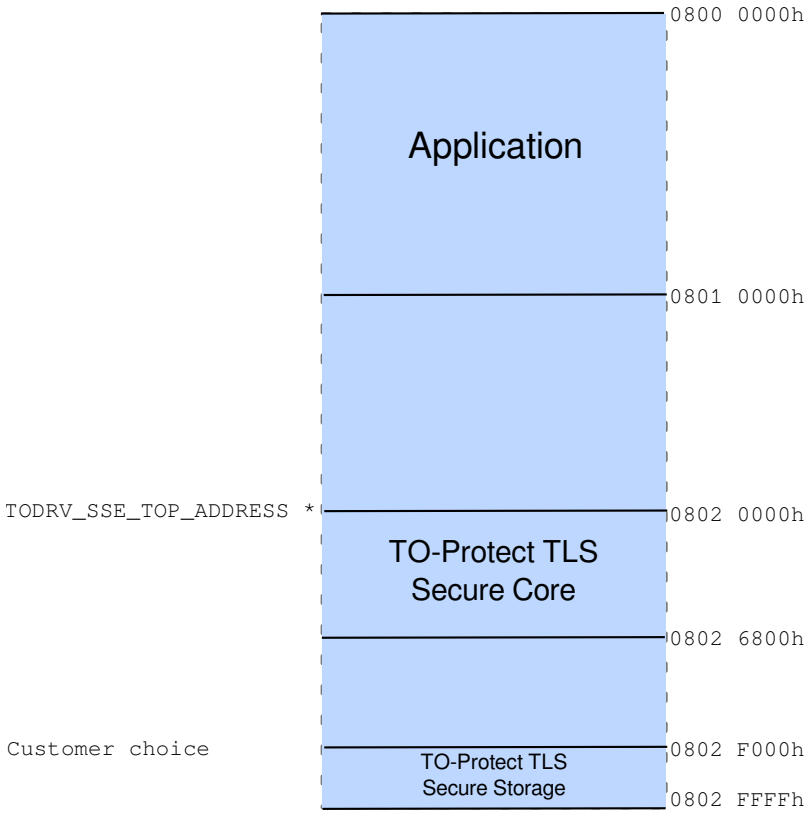
```
see "single file" flashing below
```

5.3.2 Typical Flash memory mapping

Use the commands above to combine your firmware with TO-Protect Secure Core (delivered as a binary file).

Below an example for a Cortex-M0+ with 192K of Flash. In this example, we suppose that:

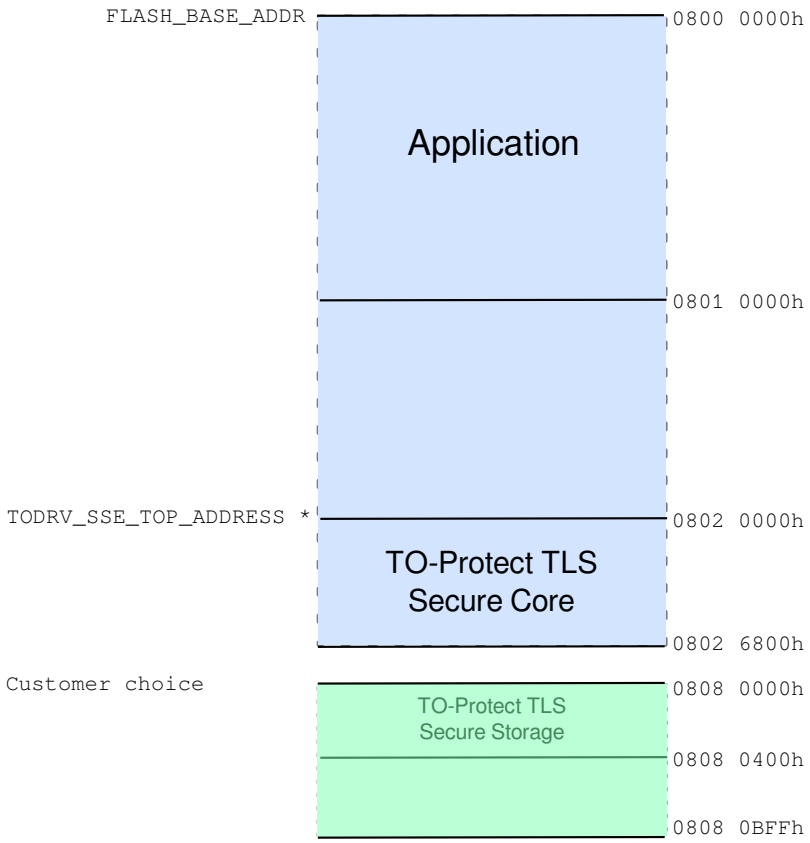
- you place TO-Protect Secure Storage in Flash, at the end of the Flash space,
- you place TO-Protect Secure Core just before the Secure Storage,
- you place the Application at the beginning of the Flash space.



(*) This address must be defined in TODRV_SSE_cfg.h, or in your IDE

Below an example for a Cortex-M0+ with 192K of Flash and 3K of E^{2}PROM: In this example, we suppose that:

- you place TO-Protect Secure Storage in E^{2}PROM, at the end of the E^{2}PROM space,
- you place TO-Protect Secure Core at the end of the Flash space,
- you place the Application at the beginning of the Flash space.



(*) This address must be defined in TODRV_SSE_cfg.h, or in your IDE

5.3.3 How to Flash

There are two ways to Flash a device:

- using a programming tool, for example STM32 ST-Link utility.
- copying the firmware binary file to the devices virtual disk, with a shell command or using drag-and-drop. This is commonly used with MBED boards.

When using drag-and-drop for flashing, you need to produce a single file, as required by this flashing process. So you have to combine two binaries: application firmware and TO-Protect.

While when using a programming tool, you can flash the application and TO-Protect separately. Or if you prefer, you can combine the application and TO-Protect into a single file. Both options are valid.

These two options are described below.

5.3.3.1 Flashing Firmware and TO-Protect separately

This method supposes that you have a tool capable to Flash a file at a given address in memory. An example is having an ST-Link probe (already soldered on Nucleo development boards), and the STM32 ST-Link Utility software.

With the above mapping examples, you would perform:

1. Flashing of TO-Protect.bin at address 0x0802.0000
2. Flashing of Application at address 0x0800.0000

Depending on your flashing tool, you can use the commands above to convert your firmware into the file format required by your flashingtool.

5.3.3.2 Flashing a single file combining Firmware and TO-Protect

This method is required if you have to Flash:

- using file copy, or drag-and-drop to an MBed board,
- using Flashing Tool, but preferring to Flash a single file, in one shot.

Typical sequence for generating the single result file is:

```
srec_cat TO-Protect.bin -Binary -offset 0x08020000 -o TO-Protect.hex -Intel
srec_cat KEIL-Application.hex -Intel TO-Protect.hex -Intel -o full_firmware.hex -Intel
arm-none-eabi-objcopy -I srec -O binary --gap-fill 0xFF full_firmware.hex full_firmware.
↪bin
```

The file **full_firmware.bin** is then ready to be flashed.

NOTE: for a KEIL application firmware, arm-none-eabi-objcopy cannot be used to manipulate .axf files produced by KEIL. You must configure KEIL to generate a .hex file.

6. TO-Protect TLS usage examples

6.1 Configuration

To configure TO-Protect TLS, the following sequence must be used.

6.1.1 Static configuration

First, you need to define the address where TO-Protect TLS has been flashed. This is detailed in *Memory settings*.

This can be achieved either:

- in your IDE, by adding compiler symbols definitions,
- in the file TODRV_SSE_cfg.h, like below.

For example:

```
/* TO-Protect address */
#define TODRV_SSE_TOP_ADDRESS 0x08020000
```

6.1.2 NVM HAL implementation

The file **examples/secure_storage_ram.c** contains an NVM HAL example, for a fake NVM implementation in RAM. This example needs to be adapted to your platform.

In this file you will implement the three functions read/write/erase:

- *TOP_secure_storage_read_func_t* TODRV_SSE_secure_storage_read
- *TOP_secure_storage_write_func_t* TODRV_SSE_secure_storage_write
- *TOP_secure_storage_erase_func_t* TODRV_SSE_secure_storage_erase

In order to verify your implementation, we recommend to call *TODRV_SSE_nvm_self_test()* (only in development mode, not in production as it wears down the Flash memory).

In order to verify that TO-Protect is correctly flashed and not corrupted, we recommend to call *TODRV_SSE_top_self_test()* (only in development mode).

6.2 Initialization

Here is how you initialize TO-Protect TLS:

```
static TOSE_ctx_t* se_ctx;

int user_init (void)
{
```

(continues on next page)

(continued from previous page)

```
        int ret;

        // Retrieve driver instance
        se_ctx = TODRV_SSE_get_ctx();

        // Initialize
        ret = TOSE_init(se_ctx);

        if (ret != TO_OK)
        {
                // Handle the situation
                // ...
                return -1;
        }

        return 0;
}
```

Now, you can access TO-Protect TLS functions.

6.2.1 TLS usage

In *Helper API* are described the functions needed to perform a TLS handshake.

First you need to implement the transport callbacks for sending/receiving data

```
int user_transport_send(void *ctx, const uint8_t *data, const uint32_t len)
{
        // User code to send data to the server
        // ...
}
int user_transport_receive(void *ctx, uint8_t *data, const uint32_t len, uint32_t *read_
↪len, int32_t timeout)
{
        // User code to receive data to the server
        // ...
}
```

Then, performing a TLS handshake can be done with one function call.

```
int user_tls_do_handshake(void)
{
        int ret;
        TO_helper_tls_ctx_t *tls_ctx;
        user_ctx_t *user_ctx; // Opaque user context

        if ((ret = TOSE_helper_tls_init_session(se_ctx, &tls_ctx, 0, (void*)&user_ctx,␣
↪user_transport_send, user_transport_receive)) != TO_OK)
        {
```

(continues on next page)

(continued from previous page)

```
                // Handle the situation
                // ...
                return -1;
        }

        ret = TOSE_helper_tls_do_handshake(tls_ctx);

        if (ret != TO_OK)
        {
                // Handle the situation
                // ...
                return -1;
        }

        return 0;
}
```

After handshake succeeded, use TOSE_helper_tls_send() and TO_helper_tls_receive() to send and receive data on the TLS link.

```
ret = TOSE_helper_tls_send(tls_ctx, (uint8_t *)rest_command, http_len);
if (ret != TO_OK)
{
        // Handle the situation
        // ...
        return -1;
}
```

```
ret = TOSE_helper_tls_receive(tls_ctx, (uint8_t *)response, buffer_len, response_len,␣
↪5000);
if (ret != TO_OK)
{
        // Handle the situation
        // ...
        return -1;
}
```

6.3 Uninitialization

When you are done, uninitialize TO-Protect TLS

```
int user_terminate(void)
{
        int ret;

        ret = TOSE_fini(se_ctx);
```

(continues on next page)

(continued from previous page)

```
        if (ret != TO_OK)
        {
                // Handle the situation
                // ...
                return -1;
        }

        return 0;
}
```

7. Provided API

7.1 Helper API

Helper APIs are high level APIs designed to make integration easier.

```
#include "TO_helper.h"
```

7.1.1 TLS handshake

TLS Helper is a set of functions making TLS handshake easy to integrate.

It only requires to provide 2 callbacks to physically send and receive data.

7.1.1.1 TLS callback functions to define

These callbacks need to be implemented and passed to TLS helper APIs to be able to send / receive data to / from the server.

typedef *TO_lib_ret_t* (***TOSE_helper_tls_send_func***)(void *priv_ctx, const uint8_t *data, const uint32_t len)

Handshake helper network send function.

This function is used by TOSE_helper_tls_handshake to send data on the network.

Param priv_ctx [**in**] Opaque context given to TOSE_helper_tls_handshake

Param data [**in**] Data to send

Param len [**in**] Length of data

Return TO_OK if data has been sent successfully, else TO_ERROR

typedef *TO_lib_ret_t* (***TOSE_helper_tls_receive_func***)(void *priv_ctx, uint8_t *data, const uint32_t len, uint32_t *read_len, int32_t timeout)

Handshake helper network receive function.

This function is used by TOSE_helper_tls_handshake to receive data from the network.

Note: #TO_AGAIN may be returned for example when this callback is implemented with POSIX recv(), and recv() returns #EINTR

Param priv_ctx [**in**] Opaque context given to TOSE_helper_tls_handshake

Param data [**in**] Data output

Param len [**in**] Length of data to read

Param read_len [**out**] Length of data read

Param timeout [**in**] Receive timeout in milliseconds (-1 for no timeout)

Retval TO_OK if some data has been received successfully, read_len is updated and >0

Retval TO_TIMEOUT timed out elapsed before any data was available

Retval TO_AGAIN the function has been interrupted before receiving any data

Retval TO_ERROR Other error

typedef *TO_ret_t* (***TOSE_helper_tls_unsecure_record***)(void *ctx, uint16_t header_length, uint8_t *in, uint16_t in_length, uint8_t **out, uint16_t *out_length)
callback to unsecure a received protected record (HANDSHAKE_ONLY_MODE)

Note: The in parameter isnt const because the callback can reuse it to unsecure in place provided it doesnt write above in_length. For example if it uses hardware decryption with constraints on memory regions used by the DMA.

Param ctx [**inout**] cipher context

Param header_length [**in**] length of the records header

Param in [**in**] input buffer containing the entire protected record (e.g. with the header)

Param in_length [**in**] length of the protected record in the input buffer

Param out [**out**] buffer with the plain text content of the record (e.g. without the header)

Param out_length [**out**] length of the plain text content

Retval TO_OK if the record is authenticated and decrypted

Retval TO_ERROR Otherwise

typedef *TO_ret_t* (***TOSE_helper_tls_secure_record***)(void *ctx, uint8_t *hdr, uint16_t hdr_length, const uint8_t *in, uint16_t in_length, uint8_t **out, uint16_t *out_length)
callback to secure a plain text record before sending (HANDSHAKE_ONLY_MODE)

Note: input and output buffers provided by the caller may overlap with a gap of at least 1 AES block (*out + hdr_length + AES_BLOCK_LEN <= in).

Param ctx [**inout**] cipher context

Param hdr [**in**] plain text record header buffer

Param hdr_length [**in**] plain text record header length

Param in [**in**] input buffer with the plain text records content data (e.g. withot header)

Param in_length [**in**] length of the plain text records content data

Param out [**inout**] output buffer with the ciphered content of the protected record

Retval TO_OK if the record cannot be encrypted

Retval TO_ERROR Otherwise

typedef *TO_ret_t* (***TOSE_helper_tls_setup_cipher_ctx***)(void *ctx, uint16_t cipher_suite, uint8_t **key_block, uint8_t *key_block_length, uint16_t *cipher_overhead_length, *TOSE_helper_tls_unsecure_record* *unsecure_record, *TOSE_helper_tls_secure_record* *secure_record)
callback to setup the cipher context (HANDSHAKE_ONLY_MODE)

Note: This callback is called during the handshake after the cipher suite is negotiated with the server and before extracting the derived key from the Secure Element.

Param ctx [**inout**] cipher context

Param cipher_suite [**in**] the negociated cipher_suite identifier (as specified in TLS RFCs)

Param key_block [**out**] pointer on the key block where key derivation from master secret is stored

Param key_block_length [**out**] length of the key block, depends upon the negociated cipher suite

Param cipher_overhead_length [**inout**] the maximum difference of length between the plain text content and the ciphered text content. The caller provides its own value if possible, the callee can lower it to 0 if it provides its own buffer to store protected records.

Param unsecure_record [**out**] callback used to authenticate and decrypt incoming records

Param secure_record [**out**] callback used to encrypt data to the outcoming records

Retval TO_OK if setup completed correctly

Retval TO_ERROR Otherwise

typedef struct TOSE_helper_tls_ctx_s **TOSE_helper_tls_ctx_t**
Opaque TLS helper context

7.1.1.2 Handshake API

Calling one function will do all the steps of the TLS handshake.

void ***default_cipher_ctx***
default cipher context when NULL is passed to *TOSE_helper_tls_set_mode_handshake_only()*

TOSE_helper_tls_setup_cipher_ctx **default_setup_cipher_ctx**
default setup cipher context when NULL is passed to *TOSE_helper_tls_set_mode_handshake_only()*

TO_lib_ret_t **TOSE_helper_tls_init_session**(*TOSE_ctx_t* *ctx, *TOSE_helper_tls_ctx_t* **tls_ctx,
const uint8_t session, void *priv_ctx,
TOSE_helper_tls_send_func send_func,
TOSE_helper_tls_receive_func receive_func)
Initialize TLS handshake.

This function initialize TLS handshake. It configures the Secure Element and initialize static envrionment.

Each initialized session must be cleaned with *TOSE_helper_tls_cleanup()*.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **tls_ctx** – [**in**] TLS context assigned
- **session** – [**in**] TLS session to use
- **priv_ctx** – [**in**] Opaque context to forward to given functions
- **send_func** – [**in**] Function to send on network
- **receive_func** – [**in**] Function to receive from network

Returns TO_OK if initialization succeed, else TO_ERROR

TO_lib_ret_t **TOSE_helper_tls_close**(*TOSE_helper_tls_ctx_t* *tls_ctx)
Close TLS handshake.

This function closes TLS handshake by sending a close notify alert to the TLS server. Given context must not be used anymore. In TCP, the socket used by this session might not be usable anymore due to close notify alert.

Parameters

- **tls_ctx** – [**in**] TLS context

Returns TO_OK if close succeed, else TO_ERROR

TO_lib_ret_t **TOSE_helper_tls_fini**(*TOSE_helper_tls_ctx_t* *tls_ctx)
Finalize TLS context.

It is needed to call this function if TCP socket closed for any reason.

Parameters

- **tls_ctx** – [**in**] TLS context

Returns TO_OK if finalize succeed, else TO_ERROR

TO_lib_ret_t **TOSE_helper_tls_cleanup**(*TOSE_helper_tls_ctx_t* *tls_ctx)
Cleanup TLS handshake.

This function closes and finalizes TLS handshake and session using TOSE_helper_tls_close and TOSE_helper_tls_fini.

Parameters

- **tls_ctx** – [**in**] TLS context

Returns TO_OK if cleanup succeed, else TO_ERROR

TO_lib_ret_t **TOSE_helper_tls_set_retransmission_timeout**(*TOSE_helper_tls_ctx_t* *tls_ctx, const uint32_t min_timeout, const uint32_t max_timeout)
Set DTLS retransmission timeout min/max values.

Parameters

- **tls_ctx** – [**in**] TLS context
- **min_timeout** – [**in**] Minimal (initial) retransmission timeout, in milliseconds
- **max_timeout** – [**in**] Maximal retransmission timeout, in milliseconds

Returns TO_OK if cleanup succeed, else TO_ERROR

TO_lib_ret_t **TOSE_helper_tls_set_retransmission_max**(*TOSE_helper_tls_ctx_t* *tls_ctx, const uint32_t max_retransmissions)

Set DTLS retransmission max value.

Retransmission counter is reset in case of successful receive.

Parameters

- **tls_ctx** – **[in]** TLS context
- **max_retransmissions** – **[in]** Maximal retransmissions count

Returns TO_OK if cleanup succeed, else TO_ERROR

TO_lib_ret_t **TOSE_helper_tls_set_fragment_max_size**(*TOSE_helper_tls_ctx_t* *tls_ctx, const uint16_t max_size)

Set DTLS fragment maximum size.

Parameters

- **tls_ctx** – **[in]** TLS context
- **max_size** – **[in]** Maximum fragment size in bytes (record & handshake headers excluded)

Returns TO_OK if cleanup succeed, else TO_ERROR

TO_lib_ret_t **TOSE_helper_tls_set_cipher_suites**(*TOSE_helper_tls_ctx_t* *tls_ctx, const uint16_t *cipher_suites, const uint16_t cipher_suites_cnt)

Set cipher suites list.

cipher_suites values must be values defined in helper header (TO_TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256, etc)

Parameters

- **tls_ctx** – **[in]** TLS context
- **cipher_suites** – **[in]** Array of cipher suites (array of 16-bits integer values. See TO_tls_cipher_suite_e.)
- **cipher_suites_cnt** – **[in]** Cipher suites count

Returns TO_OK in case of success, else TO_ERROR

TO_lib_ret_t **TOSE_helper_tls_set_config_mode**(*TOSE_helper_tls_ctx_t* *tls_ctx, *TO_tls_mode_t* mode)

Set configuration mode of the TLS session.

Note: updating the mode is persistent across reboot.

Parameters

- **tls_ctx** – **[in]** TLS context
- **mode** – **[in]** configuration mode (see TO_tls_mode_e)

Returns TO_OK in case of success, else TO_ERROR

TO_lib_ret_t **TOSE_helper_tls_set_config_certificate_slot**(*TOSE_helper_tls_ctx_t* *tls_ctx, uint8_t certificate_slot)

Configure client certificate slot of the TLS session.

Note: updating the certificate slot is persistent across reboot.

Parameters

- **tls_ctx** – **[in]** TLS context
- **mode** – **[in]** client certificate mode

Returns TO_OK in case of success, else TO_ERROR

TO_lib_ret_t **TOSE_helper_tls_set_server_name**(*TOSE_helper_tls_ctx_t* *tls_ctx, const char *server_name)

Configure the servers domain name.

When the server name is configured, it is used during handshake within the SNI extension (section 3 - RFC 6066)

Note: server_name may be NULL or empty, in that case the TLS context is configured to not use the SNI extension.

Parameters

- **tls_ctx** – **[inout]** context of the TLS session
- **server_name** – **[in]** a string with the servers domain name

Return values

- **TO_OK** – the server name is configured inside the TLS context
- **TO_ERROR** – the server name configuration failed

TO_lib_ret_t **TOSE_helper_tls_set_mode_handshake_only**(*TOSE_helper_tls_ctx_t* *tls_ctx, void *cipher_ctx, *TOSE_helper_tls_setup_cipher_ctx* setup_cipher_ctx)

configure the TLS session in HANDSHAKE_ONLY_MODE

In this mode the encryption and decryption of TLS records is delegated to the upper layer. This layer shall provide a set of callbacks to be called by the libTO to transmit the key block and to secure/unsecure records.

Note: The callback **setup_cipher_ctx** can been NULL if the libTO has been built with a default callback enabled. In that case the parameter **cipher_ctx** is ignored.

Note: This function shall be called with a initialized tls_ctx, so after calling *TOSE_helper_tls_init_session()*, and it shall be called before starting a handshake, so before *TOSE_helper_tls_do_handshake()* The following sequence show the calls needed to use the mode Handshake Only with the default cipher in AES128-GCM:

```
// function returns are ignored for compactness but should be handled in
// production code.
#define CIPHER_SUITE_CNT 2
uint16_t cipher_suites[CIPHER_SUITE_CNT] =
            {TO_TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256,
            TO_TLS_PSK_WITH_AES_128_GCM_SHA256};
TOSE_helper_tls_ctx_t *tls_ctx;
TOSE_helper_tls_init_session(DEFAULT_CTX, &tls_ctx, session_slot,
            &your_recv_send_ctx, your_send_func, your_receive_func);
TOSE_helper_tls_set_cipher_suites(tls_ctx, // setting the cipher suite is optional
            cipher_suites,     // if the server is known to always
            CIPHER_SUITE_CNT); // choose an AES-GCM cipher suite
TOSE_helper_tls_set_mode_handshake_only(tls_ctx, NULL, NULL);
TOSE_helper_tls_do_handshake(tls_ctx);
```

Note: Once the handshake is completed. The Secure Element can be shutdown with *TOSE_fini()* as the encryption/decryption/authentication of payloads are done at the library layer.

Note: Setting the mode Handshake Only has for effect to change the persistent configuration of the Secure Element. In order to go back to the mode Full TLS, the session shall be re-configured using the following sequence:

```
// function returns are ignored for compactness but should be handled in
// production code.
TOSE_helper_tls_ctx_t *tls_ctx;
TOSE_helper_tls_init_session(DEFAULT_CTX, &tls_ctx, session_slot,
            &your_recv_send_ctx, your_send_func, your_receive_func);
TOSE_helper_tls_set_mode(tls_ctx, TO_TLS_MODE_TLS_1_2_FULL);
TOSE_helper_tls_do_handshake(tls_ctx);
```

Parameters

- **tls_ctx** – [**inout**] context of the TLS session
- **cipher_ctx** – [**in**] private cipher context given to the callbacks
- **setup_cipher_ctx** – [**in**] callback used to setup the cipher context, call during the TLS handshake after cipher suite have been negotiated.

Return values

- **TO_OK** – the TLS session switched to HANDSHAKE_ONLY_MODE
- **TO_ERROR** – the TLS session didnt switch to HANDSHAKE_ONLY_MODE

TO_lib_ret_t **TOSE_helper_tls_do_handshake_step**(*TOSE_helper_tls_ctx_t* *tls_ctx)

Do TLS handshake step.

This function does one step of a TLS handshake. It encapsulates Secure Element payloads from optimized API in a TLS record, and sends it on the network through given function. It decapsulates TLS records received from the network and sends it to the Secure Element.

Parameters

- **tls_ctx** – [**in**] TLS context

Returns TO_AGAIN if intermediate step suceed, TO_OK if last step succeed, else TO_ERROR

TO_lib_ret_t **TOSE_helper_tls_do_handshake**(*TOSE_helper_tls_ctx_t* *tls_ctx)

Do TLS handshake.

This function does all the steps of a TLS handshake except initialization and cleanup. It encapsulates the Secure Element payloads from optimized API in a TLS record, and sends it on the network through given function. It decapsulates TLS records received from the network and sends it to the Secure Element. This function uses TOSE_helper_tls_handshake_init() and TOSE_helper_tls_handshake_step().

Parameters

- **tls_ctx** – [**in**] TLS context

Returns TO_OK if data has been sent successfully, else TO_ERROR

TO_lib_ret_t **TOSE_helper_tls_get_certificate_slot**(*TOSE_helper_tls_ctx_t* *tls_ctx, uint8_t *slot)

Get certificate slot used during TLS handshake.

This function must be called after handshake.

Parameters

- **tls_ctx** – [**in**] TLS context
- **slot** – [**out**] Certificate slot

Returns TO_OK if slot has been retrieved successfully, else TO_ERROR

7.1.1.3 Messaging API

Once handshake is done, these 2 functions will allow to send and receive with TLS encryption using just negociated session.

TO_lib_ret_t **TOSE_helper_tls_send**(*TOSE_helper_tls_ctx_t* *tls_ctx, const uint8_t *msg, const uint32_t msg_len)

Send TLS encrypted data.

This function uses TLS handshake keys to encrypt and send a message on the network through given function.

Parameters

- **tls_ctx** – [**in**] TLS context
- **msg** – [**in**] Message
- **msg_len** – [**in**] Message length

Returns TO_OK if message has been sent successfully, else TO_ERROR

TO_lib_ret_t **TOSE_helper_tls_receive**(*TOSE_helper_tls_ctx_t* *tls_ctx, uint8_t *msg, uint32_t max_msg_len, uint32_t *msg_len, int32_t timeout)

Receive TLS encrypted data.

This function uses given function to receive a message from the network and decrypts it with TLS handshake keys.

Parameters

- **tls_ctx** – [**in**] TLS context
- **msg** – [**out**] Message output buffer
- **max_msg_len** – [**in**] Message output buffer length
- **msg_len** – [**out**] Receive message length
- **timeout** – [**in**] Receive timeout in milliseconds (-1 for no timeout)

Returns TO_OK if message has been received successfully, TO_TIMEOUT if given timeout has been exceeded, else TO_ERROR

TO_lib_ret_t **TOSE_helper_tls_recv**(*TOSE_helper_tls_ctx_t* *tls_ctx, uint8_t *msg, uint32_t max_msg_len, uint32_t *msg_len, int32_t timeout_ms)

receive plain text application data

More precisely, receives at most a plain text record of type application data, less if the receiving buffer is too short or if a record has been partially received previously.

Note: the parameter timeout_ms is given to the receive_func() callback provided to *TOSE_helper_tls_init_session()*. To ensure to not block more than timeout_ms, the recv() callback is called just once, thus the #TO_AGAIN retval if partial data has been received.

Parameters

- **tls_ctx** – [**inout**] the TLS context
- **msg** – [**out**] received data
- **max_msg_len** – [**in**] maximum length of data writable in msg
- **msg_len** – [**out**] number of bytes read
- **timeout_ms** – [**in**] the maximum time to wait data in milliseconds

Return values

- **TO_OK** – application data received with success, msg is updated and *msg_len is greater than 0
- **TO_AGAIN** – some data has been received but not enough to receive a complete record, or it was not application data (see note above)
- **TO_TIMEOUT** – timeout elapsed before any bytes were received
- **TO_ERROR** – data cannot be received, the connection shall be (re-)initialized

7.2 TO-Protect TLS API

These APIs are used to communicate with TO-Protect.

```
#include "TO.h"
```

7.2.1 Secure storage

Secure storage is a persistent storage for TO-Protect. It is used by the TO-Protect binary to store data. Data in the secure storage could be counter or keys, and are protected by a smart mechanism against read and modifications.

As TO-Protect is an external binary, developers need to:

- Provide a non-volatile area where TO-Protect will store data
- Provide read, write and erase functions to manage this area

7.2.1.1 Provide a non-volatile area

First, you have to choose the good non-volatile device. Empty flash area is usually easy to find, but you have to reserve full sectors as each write to a sector is equivalent to a full sector erase. That is to say that if you choose flash as NVM for TO-Protect, you need to align this area with a sector, and reserve a entire number of sectors. Moreover, flash endurance is expressed as the maximum number of erase cycles that you can do before sector wear-out. It is typically 10K, or 50K erase cycles.

EEPROM are not always present in a MCU, and is generally smaller. But it doesnt require to erase a full sector to modify it content, that is to say that you generally have to align NVM area to a word or a byte. Moreover, it has a higher endurance value, typically about 100K writes.

So prefer EEPROM if it is available.

The required NVM size can be calculated with the help of `TOP_STORAGE_NVM_SIZE` macro.

7.2.1.2 Provide read, write and erase functions to NVM area

You have to declare 3 functions in your program to allow TO-Protect to access its NVM storage area.

TO-Protect uses these functions, passing a relative address, starting from 0 (which is the secure storage relative base address). When you implement these functions, you have to convert this address to absolute address, depending on where you want to put the secure storage, and then call the platform functions to respectively read, write and erase the NVM (flash or EEPROM).

Address in Flash = <SECURE_STORAGE_ADDRESS> + address provided by TO-Protect

In case of error, you have to return *TO_ERROR*.

7.2.1.2.1 Read function

- *TOP_secure_storage_read_func_t* TODRV_SSE_secure_storage_read

with the following prototype:

typedef *TO_lib_ret_t* **TOP_secure_storage_read_func_t**(uint8_t *data, const void *address, uint32_t size)

Secure storage read function.

This function is used by TO-Protect to read data from NVM. You have to implement this function with read NVM function of your platform.

Param address [**in**] Address to read, related to Secure Storage base address

Param data [**out**] Data destination

Param size [**in**] Data length

Return TO_OK if data has been read successfully, else TO_ERROR

7.2.1.2.2 Write function

- *TOP_secure_storage_write_func_t* TODRV_SSE_secure_storage_write

with the following prototype:

typedef *TO_lib_ret_t* **TOP_secure_storage_write_func_t**(void *address, const uint8_t *data, uint32_t size)

Secure storage write function.

This function is used by TO-Protect to write a block to NVM. You have to implement this function with write NVM function of your platform. This function must NOT perform any erase, as it is handled by secure storage implementation directly.

Param address [**in**] Address to write, related to Secure Storage base address

Param data [**in**] Data source

Param size [**in**] Data length (always equal to block size)

Return TO_OK if data has been written successfully, else TO_ERROR

7.2.1.2.3 Erase function

- *TOP_secure_storage_erase_func_t* TODRV_SSE_secure_storage_erase

with the following prototype:

typedef *TO_lib_ret_t* **TOP_secure_storage_erase_func_t**(void *address, uint32_t size)

Secure storage erase function.

This function is used by TO-Protect to erase NVM blocks. You have to implement this function with erase NVM function of your platform.

Param address [**in**] Address of block to erase, related to Secure Storage base address

Return TO_OK if data has been erased successfully, else TO_ERROR

7.2.2 TO-Protect functions

You will find below the list of TLS functions.

Nevertheless, please refer to *Helper API* for high-level functions to simplify integration.

7.2.2.1 Initialization

The following functions are used to initialize the library with given driver configuration.

TOSE_ctx_t ***TODRV_HSE_get_ctx**(void)
Get HSE context.

Returns HSE context pointer

TOSE_ctx_t ***TODRV_SSE_get_ctx**(void)
Get SSE context.

Returns SSE context pointer

TO_ret_t **TOSE_init**(*TOSE_ctx_t* *ctx)
Initialize TO-Protect.

Parameters

- **ctx** – [**in**] Pointer to the SE context

TO_ret_t **TOSE_fini**(*TOSE_ctx_t* *ctx)
Uninitialize TO-Protect.

Parameters

- **ctx** – [**in**] Pointer to the SE context

7.2.2.2 Self-test

The following functions are used to self-test driver configuration.

TO_lib_ret_t **TODRV_SSE_nvm_self_test**(*TO_log_ctx_t* *log_ctx)
Self-test NVM read/write/erase functions with driver configuration.

In order to verify your implementation, we recommend to call this function (only in development, not in production, as it wears down the Flash memory).

Returns TO_OK in case of success, error otherwise

TO_lib_ret_t **TODRV_SSE_top_self_test**(*TO_log_ctx_t* *log_ctx)
Self-test TO-Protect.

In order to verify that TO-Protect is correctly flashed and not corrupted, we recommend to call this function while in development mode.

Returns TO_OK in case of success, error otherwise

7.2.2.3 System

Misc. system functions.

TO_ret_t **TOSE_get_serial_number**(*TOSE_ctx_t* *ctx, uint8_t serial_number[*TO_SN_SIZE*])
Returns the unique Secure Element serial number.

The Serial Number is encoded on 8 bytes :

- The first 3 bytes identify the application ID.
- The last 5 bytes are the chip ID. Each Secure Element has an unique serial number.

Parameters

- **ctx** – **[in]** Pointer to the SE context
- **serial_number** – **[out]** Secure Element serial number

TO_ret_t **TOSE_get_hardware_serial_number**(*TOSE_ctx_t* *ctx, uint8_t hardware_serial_number[*TO_HW_SN_SIZE*])
Returns the hardware serial number.

Parameters

- **ctx** – **[in]** Pointer to the SE context
- **hardware_serial_number** – **[out]** Hardware serial number

TO_ret_t **TOSE_get_product_number**(*TOSE_ctx_t* *ctx, uint8_t product_number[*TO_PN_SIZE*])
Returns the Secure Element product number.

Product Number is a text string encoded on 12 bytes, e.g: TOSF-IS1-001

Parameters

- **ctx** – **[in]** Pointer to the SE context
- **product_number** – **[out]** Secure Element product number

TO_ret_t **TOSE_get_hardware_version**(*TOSE_ctx_t* *ctx, uint8_t hardware_version[*TO_HW_VERSION_SIZE*])
Returns the Secure Element hardware version.

Hardware version is encoded on 2 bytes. Available values are:

- 00 00: Software
- 00 01: SCO136i

Parameters

- **ctx** – **[in]** Pointer to the SE context
- **hardware_version** – **[out]** Secure Element hardware version

TO_ret_t **TOSE_get_software_version**(*TOSE_ctx_t* *ctx, uint8_t *major, uint8_t *minor, uint8_t *revision)
Returns the Secure Element software version.

Parameters

- **ctx** – **[in]** Pointer to the SE context

- **major** – [**out**] Major number. When this byte changes, API changes have occurred, incompatibility issues may be met, depending on your application.
- **minor** – [**out**] Minor number. This byte is incremented when changes happen without breaking the API.
- **revision** – [**out**] Revision number. This byte is incremented on each new build (when released).

TO_ret_t **TOSE_get_product_id**(*TOSE_ctx_t* *ctx, uint8_t product_id[*TO_PRODUCT_ID_SIZE*])
Returns the Secure Element product identifier.

The product identifier is a text string, encoded on maximum 15 ASCII bytes. It identifies the personalization profile.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **product_id** – [**out**] Secure Element product identifier

TO_ret_t **TOSE_get_random**(*TOSE_ctx_t* *ctx, const uint16_t random_length, uint8_t *random)
Returns a random number of the given length.

Request a random number to Secure Element random number generator.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **random_length** – [**in**] Requested random length
- **random** – [**out**] Returned random number

7.2.2.4 Hashes

Hashing functions.

TO_ret_t **TOSE_sha256**(*TOSE_ctx_t* *ctx, const uint8_t *data, const uint16_t data_length, uint8_t *sha256)
SHA256 computation.

Compute SHA256 hash on the given data.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **data** – [**in**] Data to compute SHA256 on
- **data_length** – [**in**] Data length, max. 512 bytes
- **sha256** – [**out**] returned computed SHA256

Returns

- TORSP_SUCCESS on success
- TO_DEVICE_WRITE_ERROR: error writing data to Secure Element
- TO_DEVICE_READ_ERROR: error reading data from Secure Element
- TO_INVALID_RESPONSE_LENGTH: unexpected response length from device

- TO_MEMORY_ERROR: internal I/O buffer overflow
- TO_ERROR: generic error

TO_ret_t **TOSE_sha256_init**(*TOSE_ctx_t* *ctx)
Compute SHA256 on more than 512 bytes of data.

This function must be followed by calls to *TOSE_sha256_update()* and *TOSE_sha256_final()*.

Parameters

- **ctx** – [**in**] Pointer to the SE context

Returns

- TORSP_SUCCESS on success
- TO_DEVICE_WRITE_ERROR: error writing data to Secure Element
- TO_DEVICE_READ_ERROR: error reading data from Secure Element
- TO_INVALID_RESPONSE_LENGTH: unexpected response length from device
- TO_MEMORY_ERROR: internal I/O buffer overflow
- TO_ERROR: generic error

TO_ret_t **TOSE_sha256_update**(*TOSE_ctx_t* *ctx, const uint8_t *data, const uint16_t length)
Update SHA256 computation with new data.

This function can be called several times to provide data to compute SHA256 on, and must be called after *TOSE_sha256_init()*.

This command is used to transmit data. It can be called several times, typically splitting the data into several blocks of 512 bytes.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **data** – [**in**] Data to compute SHA256 on
- **length** – [**in**] Data length, max. 512 bytes

Returns

- TORSP_SUCCESS on success
- TORSP_COND_OF_USE_NOT_SATISFIED if not called after *TOSE_sha256_init()* or *TOSE_sha256_update()*
- TO_DEVICE_WRITE_ERROR: error writing data to Secure Element
- TO_DEVICE_READ_ERROR: error reading data from Secure Element
- TO_INVALID_RESPONSE_LENGTH: unexpected response length from device
- TO_MEMORY_ERROR: internal I/O buffer overflow
- TO_ERROR: generic error

TO_ret_t **TOSE_sha256_final**(*TOSE_ctx_t* *ctx, uint8_t *sha256)
Returns the SHA256 hash of the data previously given.

This function must be called after *TOSE_sha256_init()* and *TOSE_sha256_update()*.

This command finalizes the process and returns the SHA256 hash of the given data. This command handles the padding computation.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **sha256** – [**out**] returned computed SHA256

Returns

- TORSP_SUCCESS on success
- TORSP_COND_OF_USE_NOT_SATISFIED: if not called after *TOSE_sha256_update()*
- TO_DEVICE_WRITE_ERROR: error writing data to Secure Element
- TO_DEVICE_READ_ERROR: error reading data from Secure Element
- TO_INVALID_RESPONSE_LENGTH: unexpected response length from device
- TO_MEMORY_ERROR: internal I/O buffer overflow
- TO_ERROR: generic error

7.2.2.5 Authentication

Certificates management and signature functions.

Secure Element certificate index is starting from 0 (if the AVNET TO136 version supports several certificates).

For details on Secure Element certificate formats see Secure Element Datasheet.

TO_ret_t **TOSE_sign**(*TOSE_ctx_t* *ctx, const uint8_t key_index, const uint8_t *challenge, const uint16_t challenge_length, uint8_t *signature)

Returns the Elliptic Curve Digital Signature of the given data.

Note that calling this function is equivalent to calling *TOSE_sha256()* followed by *TOSE_sign_hash()*.

Signature Size is twice the size of the ECC key in bytes. With a 256 bits key, signature is 64 bytes.

Parameters

- **ctx** – [**in**] Pointer to the SE contex
- **key_index** – [**in**] Key index to use for signature
- **challenge** – [**in**] Challenge to be signed
- **challenge_length** – [**in**] Challenge length (maximum 512)
- **signature** – [**out**] Returned challenge signature (64 bytes)

Returns

- TORSP_SUCCESS on success
- TORSP_ARG_OUT_OF_RANGE: invalid key index
- TO_DEVICE_WRITE_ERROR: error writing data to Secure Element
- TO_DEVICE_READ_ERROR: error reading data from Secure Element

- TO_INVALID_RESPONSE_LENGTH: unexpected response length from device
- TO_MEMORY_ERROR: internal I/O buffer overflow
- TO_ERROR: generic error

TO_ret_t **TOSE_verify**(*TOSE_ctx_t* *ctx, const uint8_t key_index, const uint8_t *data, const uint16_t data_length, const uint8_t *signature)

Verifies the given Elliptic Curve Digital Signature of the given data.

The public key used for the signature verification must be previously provided using the TOSE_set_remote_public_key() call.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **key_index** – [**in**] Remote Public Key index to use for verification
- **data** – [**in**] Data to verify signature on
- **data_length** – [**in**] Data length (maximum 512)
- **signature** – [**in**] Expected data signature (64 bytes)

Returns

- TORSP_SUCCESS on success
- TORSP_ARG_OUT_OF_RANGE: invalid key index
- TORSP_BAD_SIGNATURE: invalid signature
- TO_DEVICE_WRITE_ERROR: error writing data to Secure Element
- TO_DEVICE_READ_ERROR: error reading data from Secure Element
- TO_INVALID_RESPONSE_LENGTH: unexpected response length from device
- TO_MEMORY_ERROR: internal I/O buffer overflow
- TO_ERROR: generic error

TO_ret_t **TOSE_sign_hash**(*TOSE_ctx_t* *ctx, const uint8_t key_index, const uint8_t hash[*TO_HASH_SIZE*], uint8_t *signature)

Returns the Elliptic Curve Digital Signature of the given hash.

Signature Size is twice the size of the ECC key in bytes. With a 256 bits key, signature is 64 bytes.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **key_index** – [**in**] Key index to use for signature
- **hash** – [**in**] Hash to be signed
- **signature** – [**out**] Returned hash signature

Returns

- TORSP_SUCCESS on success
- TORSP_ARG_OUT_OF_RANGE: invalid key index
- TO_DEVICE_WRITE_ERROR: error writing data to Secure Element

- TO_DEVICE_READ_ERROR: error reading data from Secure Element
- TO_INVALID_RESPONSE_LENGTH: unexpected response length from device
- TO_MEMORY_ERROR: internal I/O buffer overflow
- TO_ERROR: generic error

TO_ret_t **TOSE_verify_hash_signature**(*TOSE_ctx_t* *ctx, const uint8_t key_index, const uint8_t hash[*TO_HASH_SIZE*], const uint8_t *signature)

Verifies the given Elliptic Curve Digital Signature of the data that generates the given hash.

The public key used for the signature verification must be previously provided using the TOSE_set_remote_public_key() call.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **key_index** – [**in**] Remote Public Key index to use for verification
- **hash** – [**in**] Hash to verify signature on (32 bytes)
- **signature** – [**in**] Expected hash signature (64 bytes)

Returns

- TORSP_SUCCESS on success
- TORSP_ARG_OUT_OF_RANGE: invalid key index
- TORSP_BAD_SIGNATURE: invalid signature
- TO_DEVICE_WRITE_ERROR: error writing data to Secure Element
- TO_DEVICE_READ_ERROR: error reading data from Secure Element
- TO_INVALID_RESPONSE_LENGTH: unexpected response length from device
- TO_MEMORY_ERROR: internal I/O buffer overflow
- TO_ERROR: generic error

TO_ret_t **TOSE_get_certificate_subject_cn**(*TOSE_ctx_t* *ctx, const uint8_t certificate_index, char subject_cn[*TO_CERT_SUBJECT_CN_MAXSIZE* + 1])

Returns subject common name of one of the Secure Element certificates.

Request a certificate subject common name to Secure Element according to the given index.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **certificate_index** – [**in**] Requested certificate index
- **subject_cn** – [**out**] Returned certificate subject common name null terminated string

Returns

- TORSP_SUCCESS on success
- TORSP_NOT_AVAILABLE: certificate Format not supported
- TORSP_ARG_OUT_OF_RANGE: invalid Certificate Number

- TO_DEVICE_WRITE_ERROR: error writing data to Secure Element
- TO_DEVICE_READ_ERROR: error reading data from Secure Element
- TO_INVALID_RESPONSE_LENGTH: unexpected response length from device
- TO_MEMORY_ERROR: internal I/O buffer overflow
- TO_ERROR: generic error

TO_ret_t **TOSE_set_certificate_signing_request_dn**(*TOSE_ctx_t* *ctx, const uint8_t certificate_index, const uint8_t csr_dn[*TO_CERT_DN_MAXSIZE*], const uint16_t csr_dn_len)

Set CSR distinguished name.

Set certificate distinguished name which will be used in next CSR.

openssl can be used to generate a fake CSR and extract the Distinguished Name sequence in DER format, like:

- openssl ecparam -out acme.key -name prime256v1 -genkey
- openssl req -new -key acme.key -out acme.csr -subj /CN=*.ACME.com/O=ACME/OU=Security Services
- openssl asn1parse -in acme.csr Note the number of the first SEQUENCE with depth=2; in example above, this is item number 9
- openssl asn1parse -in acme.csr -strparse 9 -out extract_acme_DN.der and the file extract_acme_DN.der contains the Distinguished Name in DER format, that can be used as parameter to *TOSE_set_certificate_signing_request_dn()* Double-check that Distinguished Name size (check *extract_acme_DN.der* file size on the disk) does not exceed TO_CERT_DN_MAXSIZE; else this will be rejected by libTO.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **certificate_index** – [**in**] Certificate index
- **csr_dn** – [**in**] CSR distinguished name (without main sequence tag & length)
- **csr_dn_len** – [**in**] CSR distinguished name length

Returns

- TORSP_SUCCESS on success
- TORSP_ARG_OUT_OF_RANGE: invalid Certificate Number
- TORSP_INVALID_LEN: invalid Distinguished Name length (> TO_CERT_DN_MAXSIZE)
- TO_DEVICE_WRITE_ERROR: error writing data to Secure Element
- TO_DEVICE_READ_ERROR: error reading data from Secure Element
- TO_MEMORY_ERROR: internal I/O buffer overflow
- TO_ERROR: generic error

TO_ret_t **TOSE_get_certificate_signing_request**(*TOSE_ctx_t* *ctx, const uint8_t certificate_index, uint8_t *csr, uint16_t *size)

Get new certificate signing request.

Request a x509 DER formated certificate signing request according to the given index. CSR distinguished name can be set with *TOSE_set_certificate_signing_request_dn()*, otherwise existing certificate DN will be used (if any). Secure Element CSR size will not exceed TO_CERT_X509_MAXSIZE.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **certificate_index** – [**in**] Certificate index to renew
- **csr** – [**out**] Returned CSR data (can be NULL to determine needed buffer size)
- **size** – [**out**] Returned CSR real size

Returns

- TORSP_SUCCESS on success
- TORSP_ARG_OUT_OF_RANGE: invalid Certificate Number
- TO_DEVICE_WRITE_ERROR: error writing data to Secure Element
- TO_DEVICE_READ_ERROR: error reading data from Secure Element
- TO_INVALID_RESPONSE_LENGTH: unexpected response length from device
- TO_MEMORY_ERROR: internal I/O buffer overflow
- TO_ERROR: generic error

TO_ret_t **TOSE_set_certificate_x509**(*TOSE_ctx_t* *ctx, const uint8_t certificate_index, const uint8_t *certificate, const uint16_t size)

Set new certificate from previously generated CSR.

Set a x509 DER formated certificate according to the given index. The new certificate must be signed by a CA trusted by the Secure Element. Secure Element certificate size cannot exceed TO_CERT_X509_MAXSIZE.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **certificate_index** – [**in**] Requested certificate index
- **certificate** – [**in**] New certificate data (x509 DER formated)
- **size** – [**in**] New certificate size

Returns

- TORSP_SUCCESS on success
- TORSP_NOT_AVAILABLE: certificate Format not supported
- TORSP_ARG_OUT_OF_RANGE: invalid Certificate Number
- TO_DEVICE_WRITE_ERROR: error writing data to Secure Element
- TO_DEVICE_READ_ERROR: error reading data from Secure Element
- TO_INVALID_RESPONSE_LENGTH: unexpected response length from device

- TO_MEMORY_ERROR: internal I/O buffer overflow
- TO_ERROR: generic error

TO_ret_t **TOSE_set_certificate_x509_init**(*TOSE_ctx_t* *ctx, const uint8_t certificate_index)

Initialize to set new certificate from previously generated CSR.

See TOSE_set_certificate_x509

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **certificate_index** – [**in**] Requested certificate index

Returns

- TORSP_SUCCESS on success
- TORSP_NOT_AVAILABLE: certificate Format not supported
- TORSP_ARG_OUT_OF_RANGE: invalid Certificate Number
- TO_DEVICE_WRITE_ERROR: error writing data to Secure Element
- TO_DEVICE_READ_ERROR: error reading data from Secure Element
- TO_INVALID_RESPONSE_LENGTH: unexpected response length from device
- TO_MEMORY_ERROR: internal I/O buffer overflow
- TO_ERROR: generic error

TO_ret_t **TOSE_set_certificate_x509_update**(*TOSE_ctx_t* *ctx, const uint8_t *certificate, const uint16_t size)

Update to set new certificate from previously generated CSR.

See TOSE_set_certificate_x509

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **certificate** – [**in**] New certificate partial data (from x509 DER formated)
- **size** – [**in**] New certificate partial data size

Returns

- TORSP_SUCCESS on success
- TORSP_NOT_AVAILABLE: certificate Format not supported
- TORSP_ARG_OUT_OF_RANGE: invalid Certificate Number
- TO_DEVICE_WRITE_ERROR: error writing data to Secure Element
- TO_DEVICE_READ_ERROR: error reading data from Secure Element
- TO_INVALID_RESPONSE_LENGTH: unexpected response length from device
- TO_MEMORY_ERROR: internal I/O buffer overflow
- TO_ERROR: generic error

TO_ret_t **TOSE_set_certificate_x509_final**(*TOSE_ctx_t* *ctx)

Finalize to set new certificate from previously generated CSR.

See TOSE_set_certificate_x509

Parameters

- **ctx** – [**in**] Pointer to the SE context

Returns

- TORSP_SUCCESS on success
- TORSP_NOT_AVAILABLE: certificate Format not supported
- TORSP_ARG_OUT_OF_RANGE: invalid Certificate Number
- TO_DEVICE_WRITE_ERROR: error writing data to Secure Element
- TO_DEVICE_READ_ERROR: error reading data from Secure Element
- TO_INVALID_RESPONSE_LENGTH: unexpected response length from device
- TO_MEMORY_ERROR: internal I/O buffer overflow
- TO_ERROR: generic error

TO_ret_t **TOSE_get_certificate**(*TOSE_ctx_t* *ctx, const uint8_t certificate_index, const TO_certificate_format_t format, uint8_t *certificate)

Returns one of the Secure Element certificates.

Request a certificate to Secure Element according to the given index and format.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **certificate_index** – [**in**] Requested certificate index
- **format** – [**in**] Requested certificate format
- **certificate** – [**out**] Certificate, size depends on the certificate type (see TO_cert_*_t)

Returns

- TORSP_SUCCESS on success
- TORSP_NOT_AVAILABLE: certificate Format not supported
- TORSP_ARG_OUT_OF_RANGE: invalid Certificate Number
- TO_DEVICE_WRITE_ERROR: error writing data to Secure Element
- TO_DEVICE_READ_ERROR: error reading data from Secure Element
- TO_INVALID_RESPONSE_LENGTH: unexpected response length from device
- TO_MEMORY_ERROR: internal I/O buffer overflow
- TO_ERROR: generic error

TO_ret_t **TOSE_get_certificate_x509**(*TOSE_ctx_t* *ctx, const uint8_t certificate_index, uint8_t *certificate, uint16_t *size)

Returns one of the certificates, x509 DER formated.

Request a x509 DER formated certificate according to the given index. Secure Element certificate size will not exceed TO_CERT_X509_MAXSIZE.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **certificate_index** – [**in**] Requested certificate index
- **certificate** – [**out**] Returned certificate data (can be NULL to determine needed buffer size)
- **size** – [**out**] Returned certificate real size

Returns

- TORSP_SUCCESS on success
- TORSP_NOT_AVAILABLE: certificate Format not supported
- TORSP_ARG_OUT_OF_RANGE: invalid Certificate Number
- TO_DEVICE_WRITE_ERROR: error writing data to Secure Element
- TO_DEVICE_READ_ERROR: error reading data from Secure Element
- TO_INVALID_RESPONSE_LENGTH: unexpected response length from device
- TO_MEMORY_ERROR: internal I/O buffer overflow
- TO_ERROR: generic error

TO_ret_t **TOSE_verify_ca_certificate_and_store**(*TOSE_ctx_t* *ctx, const uint8_t ca_key_index, const uint8_t subca_key_index, const uint8_t *certificate, const uint16_t certificate_len)

Requests to verify signature of the given subCA certificate; if verification succeeds, this certificate is stored into Secure Element CA slot.

Note: the only supported certificate format for this command is DER X509.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **ca_key_index** – [**in**] index of the CA slot used to verify subCA
- **subca_key_index** – [**in**] subCA index to store certificate
- **certificate** – [**in**] Certificate to be verified and stored
- **certificate_len** – [**in**] Certificate length

Returns

- TORSP_SUCCESS on success
- TORSP_ARG_OUT_OF_RANGE: invalid CA Key index
- TO_DEVICE_WRITE_ERROR: error writing data to Secure Element
- TO_DEVICE_READ_ERROR: error reading data from Secure Element
- TO_INVALID_RESPONSE_LENGTH: unexpected response length from device
- TO_MEMORY_ERROR: internal I/O buffer overflow

- TO_ERROR: generic error

TO_ret_t **TOSE_get_challenge_and_store**(*TOSE_ctx_t* *ctx, uint8_t challenge[*TO_CHALLENGE_SIZE*])

Returns a challenge (random number of fixed length) and store it into Secure Element memory.

This command must be called before TOSE_verify_challenge_signature().

Parameters

- **ctx** – **[in]** Pointer to the SE context
- **challenge** – **[out]** Returned challenge

Returns

- TORSP_SUCCESS on success
- TO_DEVICE_WRITE_ERROR: error writing data to Secure Element
- TO_DEVICE_READ_ERROR: error reading data from Secure Element
- TO_INVALID_RESPONSE_LENGTH: unexpected response length from device
- TO_MEMORY_ERROR: internal I/O buffer overflow
- TO_ERROR: generic error

7.2.2.6 TLS

TO_ret_t **TOSE_tls_reset**(*TOSE_ctx_t* *ctx)

Resets the current TLS/DTLS session.

Note:

After resetting the session, a full handshake will have to be re-negociated, as the session keys and master secrets are reset for this session. It does not have any influence on the other sessions that may be opened.

It can be used also to fix a malfunctioning TLS slot.

Parameters

- **ctx** – **[in]** Pointer to the SE context

TO_ret_t TOSE_tls_set_mode (TOSE_ctx_t *ctx, const TO_tls_mode_t mode) TO_DEPRECATED

Selects between TLS and DTLS mode and resets the session for the current selected slot.

Deprecated:

Parameters

- **ctx** – **[in]** Pointer to the SE context
- **mode** – **[in]** TLS mode. Currently only *TO_TLS_MODE_TLS_1_2* and *TO_TLS_MODE_DTLS_1_2* are supported.

TO_ret_t **TOSE_tls_set_config**(*TOSE_ctx_t* *ctx, const *TO_tls_config_id_t* config_id, const uint8_t *config, const uint16_t config_len)

Set TLS config (either mode or cipher suite selection).

Permits to switch to TLS or DTLS, to select a cipher suite for the handshake and resets the current session (if the configuration has changed).

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **config_id** – [**in**] TLS configuration ID (either *TO_TLS_CONFIG_ID_MODE* or *TO_TLS_CONFIG_ID_CIPHER_SUITES*)
- **config** – [**in**] Pointer to the desired new TLS configuration
- **config_len** – [**in**] TLS configuration length (1 for the mode, 2 for the cipher suite)

TO_ret_t **TOSE_tls_set_session**(*TOSE_ctx_t* *ctx, const uint8_t session)

Selects the current TLS session slot to be used.

Note: There are several session slots available which can be connected to different servers. Depending on your application you may have to switch between those session slots.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **session** – [**in**] TLS session ID

TO_ret_t **TOSE_tls_set_cid_ext_id**(*TOSE_ctx_t* *ctx, const *TO_tls_extension_t* cid_ext_id)

Set sets the type of the extension ID corresponding to the connection ID.

Currently, the ID corresponding to the connection ID is still part of a draft standard (dec. 2021). Until the moment the RFC standard is published, this entry-point is used to provide this information.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **cid_ext_id** – [**in**] Connection ID extension ID

TO_ret_t **TOSE_tls_get_client_hello**(*TOSE_ctx_t* *ctx, const uint8_t timestamp[*TO_TIMESTAMP_SIZE*], uint8_t *client_hello, uint16_t *client_hello_len)

Generates the TLS Client_Hello (client) message.

When a client first connects to a server, it is required to send the ClientHello as its first message. The client can also send a ClientHello in response to a HelloRequest or on its own initiative in order to renegotiate the security parameters in an existing connection.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **timestamp** – [**in**] Timestamp (seconds since epoch)
- **client_hello** – [**out**] Pointer to a buffer receiving the ClientHello payload (up to 79 bytes in TLS, 120 bytes in DTLS)
- **client_hello_len** – [**out**] Pointer to receive the ClientHello payload length

TO_ret_t **TOSE_tls_get_client_hello_ext**(*TOSE_ctx_t* *ctx, const uint8_t timestamp[*TO_TIMESTAMP_SIZE*], const uint8_t *ext_data, uint16_t ext_length, uint8_t *client_hello, uint16_t *client_hello_len)

Get TLS ClientHello with extension.

Return the TLS handshake payload of the standard TLS ClientHello message. This payload must be encapsulated in a TLS record. The length of the response can be different depending on the use case.

Parameters

- **ctx** – [**inout**] SE context
- **timestamp** – [**in**] Timestamp (seconds since epoch)
- **ext_data** – [**in**] extension data
- **ext_length** – [**in**] extension length
- **client_hello** – [**out**] ClientHello payload
- **client_hello_len** – [**out**] ClientHello payload length

Return values

- **TORSP_SUCCESS** – on success
- **TO_DEVICE_WRITE_ERROR** – error writing data to Secure Element
- **TO_DEVICE_READ_ERROR** – error reading data from Secure Element
- **TO_INVALID_RESPONSE_LENGTH** – unexpected response length from device
- **TO_MEMORY_ERROR** – internal I/O buffer overflow
- **TO_ERROR** – generic error

TO_ret_t **TOSE_tls_get_client_hello_init**(*TOSE_ctx_t* *ctx, const uint8_t timestamp[*TO_TIMESTAMP_SIZE*], const uint8_t *ext_data, uint16_t ext_length, uint16_t *client_hello_len, uint8_t *final_flag)

Get TLS ClientHello - CAPI version - Init.

Initialize retrieval of the TLS handshake payload of the standard TLS ClientHello message. This payload must be encapsulated in a TLS record.

Parameters

- **ctx** – [**inout**] SE context
- **timestamp** – [**in**] Timestamp (seconds since epoch)
- **ext_data** – [**in**] extension data
- **ext_length** – [**in**] extension length
- **client_hello_len** – [**out**] ClientHello payload length
- **final_flag** – [**out**] signal the final chunk of ClientHello to be received with *TOSE_tls_get_client_hello_final()*

Return values

- **TORSP_SUCCESS** – on success

- **TO_DEVICE_WRITE_ERROR** – error writing data to Secure Element
- **TO_DEVICE_READ_ERROR** – error reading data from Secure Element
- **TO_INVALID_RESPONSE_LENGTH** – unexpected response length from device
- **TO_MEMORY_ERROR** – internal I/O buffer overflow
- **TO_ERROR** – generic error

TO_ret_t **TOSE_tls_get_client_hello_update**(*TOSE_ctx_t* *ctx, uint8_t *data, uint16_t *part_len, uint8_t *final_flag)

Get TLS ClientHello - CAPI version - Update.

Return a part of the TLS handshake payload of the standard TLS ClientHello message. This payload must be encapsulated in a TLS record.

Parameters

- **ctx** – [**inout**] SE context
- **data** – [**out**] ClientHello payload part
- **part_len** – [**out**] ClientHello payload part length
- **final_flag** – [**out**] signal the final chunk of ClientHello to be received with *TOSE_tls_get_client_hello_final()*

Return values

- **TORSP_SUCCESS** – on success
- **TO_DEVICE_WRITE_ERROR** – error writing data to Secure Element
- **TO_DEVICE_READ_ERROR** – error reading data from Secure Element
- **TO_INVALID_RESPONSE_LENGTH** – unexpected response length from device
- **TO_MEMORY_ERROR** – internal I/O buffer overflow
- **TO_ERROR** – generic error

TO_ret_t **TOSE_tls_get_client_hello_final**(*TOSE_ctx_t* *ctx, uint8_t *data)

Get TLS ClientHello - CAPI version - Final.

Return the last part of the TLS handshake payload of the standard TLS ClientHello message. This payload must be encapsulated in a TLS record.

Parameters

- **ctx** – [**inout**] SE context
- **data** – [**out**] last ClientHello payload part

Return values

- **TORSP_SUCCESS** – on success
- **TO_DEVICE_WRITE_ERROR** – error writing data to Secure Element
- **TO_DEVICE_READ_ERROR** – error reading data from Secure Element
- **TO_INVALID_RESPONSE_LENGTH** – unexpected response length from device
- **TO_MEMORY_ERROR** – internal I/O buffer overflow

- **TO_ERROR** – generic error

TO_ret_t **TOSE_tls_handle_hello_verify_request**(*TOSE_ctx_t* *ctx, const uint8_t *hello_verify_request, const uint32_t hello_verify_request_len)

Handles the DTLS HelloVerifyRequest (server) message.

When the client sends its ClientHello message to the server, the server MAY respond with a HelloVerifyRequest message. This message contains a stateless cookie.

Note: This message processing is only needed in the case of DTLS

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **hello_verify_request** – [**in**] HelloVerifyRequest message
- **hello_verify_request_len** – [**in**] HelloVerifyRequest message length

TO_ret_t **TOSE_tls_handle_server_hello**(*TOSE_ctx_t* *ctx, const uint8_t *server_hello, const uint32_t server_hello_len)

Handles the ServerHello (server) message.

The server will send this message in response to a ClientHello message when it was able to find an acceptable set of algorithms. If it cannot find such a match, it will respond with a handshake failure alert.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **server_hello** – [**in**] ServertHello payload
- **server_hello_len** – [**in**] ServertHello payload length

TO_ret_t **TOSE_tls_handle_server_hello_init**(*TOSE_ctx_t* *ctx, const uint32_t server_hello_len)

Handle TLS ServerHello - CAPI version - Init.

Initialize handling of the TLS handshake payload of the standard TLS ServerHello message received during TLS handshake.

Parameters

- **ctx** – [**inout**] SE context
- **server_hello_len** – [**in**] ServerHello payload length

Return values

- **TORSP_SUCCESS** – on success
- **TO_DEVICE_WRITE_ERROR** – error writing data to Secure Element
- **TO_DEVICE_READ_ERROR** – error reading data from Secure Element
- **TORSP_ARG_OUT_OF_RANGE** – bad content
- **TO_MEMORY_ERROR** – internal I/O buffer overflow
- **TO_ERROR** – generic error

TO_ret_t **TOSE_tls_handle_server_hello_update**(*TOSE_ctx_t* *ctx, const uint8_t *data, const uint32_t part_len)

Handle TLS ServerHello - CAPI version - Update.

Handle a part of the TLS handshake payload of the standard TLS ServerHello message received during TLS handshake.

Parameters

- **ctx** – [**inout**] SE context
- **data** – [**in**] part of ServerHello payload
- **part_len** – [**in**] part length

Return values

- **TORSP_SUCCESS** – on success
- **TO_DEVICE_WRITE_ERROR** – error writing data to Secure Element
- **TO_DEVICE_READ_ERROR** – error reading data from Secure Element
- **TORSP_ARG_OUT_OF_RANGE** – bad content
- **TO_MEMORY_ERROR** – internal I/O buffer overflow
- **TO_ERROR** – generic error

TO_ret_t **TOSE_tls_handle_server_hello_final**(*TOSE_ctx_t* *ctx, const uint8_t *data, const uint32_t last_len)

Handle TLS ServerHello - CAPI version - Final.

Handle the last part of the TLS handshake payload of the standard TLS ServerHello message received during TLS handshake.

Parameters

- **ctx** – [**inout**] SE context
- **data** – [**in**] last part of ServerHello payload
- **last_len** – [**in**] last part len

Return values

- **TORSP_SUCCESS** – on success
- **TO_DEVICE_WRITE_ERROR** – error writing data to Secure Element
- **TO_DEVICE_READ_ERROR** – error reading data from Secure Element
- **TORSP_ARG_OUT_OF_RANGE** – bad content
- **TO_MEMORY_ERROR** – internal I/O buffer overflow
- **TO_ERROR** – generic error

TO_ret_t **TOSE_tls_handle_server_certificate**(*TOSE_ctx_t* *ctx, const uint8_t *server_certificate, const uint32_t server_certificate_len)

Handles the TLS Certificate (server) message.

The server MUST send a Certificate message whenever the agreed- upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except DH_anon). This message will always immediately follow the ServerHello message.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **server_certificate** – [**in**] Certificate payload
- **server_certificate_len** – [**in**] Certificate payload length

TO_ret_t **TOSE_tls_handle_server_certificate_init**(*TOSE_ctx_t* *ctx, const uint8_t *server_certificate_init, const uint32_t server_certificate_init_len)

Handles the TLS Server Certificate header (server)

Handle TLS Server Certificate header from TLS handshake payload of the standard TLS ServerCertificate message. The goal of TOSE_tls_handle_server_certificate_ init(), udate() and final(), is to validate a certificate chain, and to store the public key of the first certificate. You must decapsulate it from TLS record prior to use this command.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **server_certificate_init** – [**in**] Certificate payload header (handshake header – certificates list length)
- **server_certificate_init_len** – [**in**] Certificate payload header length

TO_ret_t **TOSE_tls_handle_server_certificate_update**(*TOSE_ctx_t* *ctx, const uint8_t *server_certificate_update, const uint32_t server_certificate_update_len)

Handles the TLS Server Certificate partial payload (server)

Handle TLS Server Certificate partial payload from TLS handshake payload of the standard TLS ServerCertificate message, and if possible, verify the signature and memories the key of the current certificate of the certificates chain. You must decapsulate it from TLS record prior to use this command. This command can be called several times. *TOSE_tls_handle_server_certificate_init()* must be called prior to this call.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **server_certificate_update** – [**in**] Certificate partial payload
- **server_certificate_update_len** – [**in**] Certificate partial payload length

TO_ret_t **TOSE_tls_handle_server_certificate_final**(*TOSE_ctx_t* *ctx)

Finishes the TLS Server Certificate handling (server)

Parameters

- **ctx** – [**in**] Pointer to the SE context * Finish Server Certificate TLS handshake payload handling by verifying signature of last certificate and store the public key of the first certificate of the chain. You must decapsulate it from TLS record prior to use this command. Functions *TOSE_tls_handle_server_certificate_init()*, and *TOSE_tls_handle_server_certificate_update()* must be called prior to this call.

TO_ret_t **TOSE_tls_handle_server_key_exchange**(*TOSE_ctx_t* *ctx, const uint8_t *server_key_exchange, const uint32_t server_key_exchange_len)

Handle the TLS ServerKeyExchange (server) message.

Handle TLS handshake payload of the standard TLS ServerKeyExchange message.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **server_key_exchange** – [**in**] ServerKeyExchange payload
- **server_key_exchange_len** – [**in**] ServerKeyExchange payload length

TO_ret_t **TOSE_tls_handle_server_key_exchange_init**(*TOSE_ctx_t* *ctx, const uint8_t *server_key_exchange_init, const uint32_t server_key_exchange_init_len)

Handles the TLS Server ServerKeyExchange (server) header.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **server_key_exchange_init** – [**in**] ServerKeyExchange payload header (handshake header

  – key_exchanges list length)
- **server_key_exchange_init_len** – [**in**] ServerKeyExchange payload header length

TO_ret_t **TOSE_tls_handle_server_key_exchange_update**(*TOSE_ctx_t* *ctx, const uint8_t *server_key_exchange_update, const uint32_t server_key_exchange_update_len)

Handles the TLS Server ServerKeyExchange partial payload (server)

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **server_key_exchange_update** – [**in**] ServerKeyExchange partial payload
- **server_key_exchange_update_len** – [**in**] ServerKeyExchange partial payload length

TO_ret_t **TOSE_tls_handle_server_key_exchange_final**(*TOSE_ctx_t* *ctx)

Finishes TLS Server ServerKeyExchange handling (server)

Parameters

- **ctx** – [**in**] Pointer to the SE context

TO_ret_t **TOSE_tls_handle_certificate_request**(*TOSE_ctx_t* *ctx, const uint8_t *certificate_request, const uint32_t certificate_request_len)

Handles the TLS CertificateRequest (server) message.

The server MUST send a Certificate message whenever the agreed- upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except DH_anon). This message will always immediately follow the ServerHello message.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **certificate_request** – [**in**] CertificateRequest payload
- **certificate_request_len** – [**in**] CertificateRequest payload length

TO_ret_t **TOSE_tls_handle_server_hello_done**(*TOSE_ctx_t* *ctx, const uint8_t *server_hello_done, const uint32_t server_hello_done_len)

Handles the DTLS ServerHelloDone (server) message.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **server_hello_done** – [**in**] ServerHelloDone payload
- **server_hello_done_len** – [**in**] ServerHelloDone payload length

TO_ret_t **TOSE_tls_get_certificate**(*TOSE_ctx_t* *ctx, uint8_t *certificate, uint16_t *certificate_len)

Generates the TLS Certificate (client) message.

This is the first message the client can send after receiving a ServerHelloDone message. This message is only sent if the server requests a certificate. If no suitable certificate is available, the client MUST send a certificate message containing no certificates.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **certificate** – [**out**] Certificate payload
- **certificate_len** – [**out**] Certificate payload length

TO_ret_t **TOSE_tls_get_certificate_init**(*TOSE_ctx_t* *ctx, uint8_t *certificate, uint16_t *certificate_len)

Get the TLS Certificate initialization (client)

This function is used with *TOSE_tls_get_certificate_update()* and *TOSE_tls_get_certificate_final()* to get TLS Certificate of more than 512 bytes without limitation. This first command initiates the process.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **certificate** – [**out**] Certificate payload
- **certificate_len** – [**out**] Certificate payload length

TO_ret_t **TOSE_tls_get_certificate_update**(*TOSE_ctx_t* *ctx, uint8_t *certificate, uint16_t *certificate_len)

Gets the TLS Certificate update (client)

This command can be called several times. Function *TOSE_tls_get_certificate_init()* must be called prior to this command.

Parameters

- **ctx** – [**in**] Pointer to the SE context

- **certificate** – [**out**] Certificate payload
- **certificate_len** – [**out**] Certificate payload length

TO_ret_t **TOSE_tls_get_certificate_final**(*TOSE_ctx_t* *ctx)
Gets the TLS Certificate finalize (client)

Parameters

- **ctx** – [**in**] Pointer to the SE context

TO_ret_t **TOSE_tls_get_client_key_exchange**(*TOSE_ctx_t* *ctx, uint8_t *client_key_exchange, uint16_t *client_key_exchange_len)
Gets the TLS ClientKeyExchange (client) message.

Get TLS handshake payload of the standard TLS message ClientKeyExchange, containing internal Secure Elements ephemeral public key if using ECDHE cipher suite.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **client_key_exchange** – [**out**] ClientKeyExchange payload
- **client_key_exchange_len** – [**out**] ClientKeyExchange payload length

TO_ret_t **TOSE_tls_get_certificate_verify**(*TOSE_ctx_t* *ctx, uint8_t *certificate_verify, uint16_t *certificate_verify_len)
Generates the TLS Certificate_Verify (client) message.

This message is used to provide explicit verification of a client certificate. This message is only sent following a client certificate that has signing capability (i.e., all certificates except those containing fixed Diffie-Hellman parameters). When sent, it MUST immediately follow the client key exchange message.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **certificate_verify** – [**out**] CertificateVerify payload
- **certificate_verify_len** – [**out**] CertificateVerify payload length

TO_ret_t **TOSE_tls_get_change_cipher_spec**(*TOSE_ctx_t* *ctx, uint8_t *change_cipher_spec, uint16_t *change_cipher_spec_len)
Generates the TLS Change_Cipher_Spec (client) message.

The ChangeCipherSpec message is sent by both the client and the server to notify the receiving party that subsequent records will be protected under the newly negotiated CipherSpec and keys. This message is technically not part of the handshake.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **change_cipher_spec** – [**out**] ChangeCipherSpec payload
- **change_cipher_spec_len** – [**out**] ChangeCipherSpec payload length

TO_ret_t **TOSE_tls_get_finished**(*TOSE_ctx_t* *ctx, uint8_t *finished, uint16_t *finished_len)
Generates the TLS Finished (client) message.

The Finished message is the first one protected with the just negotiated algorithms, keys, and secrets. Recipients of Finished messages MUST verify that the contents are correct. Once a side has sent its Finished message and received and validated the Finished message from its peer, it may begin to send and receive application data over the connection.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **finished** – [**out**] Finish payload
- **finished_len** – [**out**] Finish payload length

TO_ret_t **TOSE_tls_handle_change_cipher_spec**(*TOSE_ctx_t* *ctx, const uint8_t *change_cipher_spec, const uint32_t change_cipher_spec_len)

Handles the TLS ChangeCipherSpec (server) message.

The change cipher spec protocol exists to signal transitions in ciphering strategies. The protocol consists of a single message, which is encrypted and compressed under the current (not the pending) connection state.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **change_cipher_spec** – [**in**] ChangeCipherSpec payload
- **change_cipher_spec_len** – [**in**] ChangeCipherSpec payload length

TO_ret_t **TOSE_tls_handle_finished**(*TOSE_ctx_t* *ctx, const uint8_t *finished, const uint32_t finished_len)

Handles the TLS Finished (server) message.

The Finished message is the first one protected with the just negotiated algorithms, keys, and secrets. Recipients of Finished messages MUST verify that the contents are correct. Once a side has sent its Finished message and received and validated the Finished message from its peer, it may begin to send and receive application data over the connection.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **finished** – [**in**] Finished payload
- **finished_len** – [**in**] Finish payload length

TO_ret_t **TOSE_tls_get_certificate_slot**(*TOSE_ctx_t* *ctx, uint8_t *slot)

Generates the TLS certificate slot used during handshake (client) message.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **slot** – [**out**] Certificate slot

Post Handshake must have been proceeded before calling this function.

TO_ret_t **TOSE_tls_secure_payload**(*TOSE_ctx_t* *ctx, const uint8_t *header, const uint16_t header_len, const uint8_t *data, const uint16_t data_len, uint8_t *payload, uint16_t *payload_len)

Secures a (client) message with TLS.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **header** – [**in**] TLS header
- **header_len** – [**in**] TLS header length
- **data** – [**in**] TLS data
- **data_len** – [**in**] TLS data length
- **payload** – [**out**] Secured message (without header)
- **payload_len** – [**out**] Secured message (without header) length

Post Handshake must have been proceeded before calling this function.

TO_ret_t **TOSE_tls_unsecure_payload**(*TOSE_ctx_t* *ctx, const uint8_t *header, const uint16_t header_len, const uint8_t *payload, const uint16_t payload_len, uint8_t *data, uint16_t *data_len)

Unsecure message with TLS.

Decrypt data received from server through TLS. Take a TLS record as input with encrypted content and return a TLS record with clear content.

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **header** – [**in**] TLS header
- **header_len** – [**in**] TLS header length
- **payload** – [**in**] Secured message (without header)
- **payload_len** – [**in**] Secured message (without header) length
- **data** – [**out**] TLS data
- **data_len** – [**out**] TLS data length

Post Handshake must have been proceeded before calling this function.

TO_ret_t **TOSE_tls_handle_mediator_certificate**(*TOSE_ctx_t* *ctx, const uint8_t *mediator_certificate, const uint32_t mediator_certificate_len)

Handles the TLS proprietary MediatorCertificate (server) message.

This is a TO-specific message, used to handle the mediator certificate. This message is not part of any standard (TLS or DTLS).

Parameters

- **ctx** – [**in**] Pointer to the SE context
- **mediator_certificate** – [**in**] MediatorCertificate payload
- **mediator_certificate_len** – [**in**] MediatorCertificate payload length

7.3 Library core APIs

7.3.1 Logs

The following function is used to set library log level.

void **TO_set_log_level**(*TO_log_ctx_t* *log_ctx, const TO_log_level_t level, TO_log_func_t *log_function)

Sets the Log function and log level.

This function permits to change the log level and the log function.

Parameters

- **log_ctx** – Current log context
- **level** – Desired log level
- **log_function** – Log function (eg. TO_log)

7.4 Types and definitions

LibTO types and definitions.

```
#include "TO_defs.h"
```

7.4.1 Library error codes

group **lib_codes**

Error codes

Typedefs

typedef enum *TO_lib_ret_e* **TO_lib_ret_t**

Enums

enum **TO_lib_ret_e**

Values:

enumerator **TO_OK**

enumerator **TO_MEMORY_ERROR**

enumerator **TO_DEVICE_WRITE_ERROR**

enumerator **TO_DEVICE_READ_ERROR**

enumerator **TO_INVALID_CA_ID**

enumerator **TO_INVALID_CERTIFICATE_FORMAT**

enumerator **TO_INVALID_CERTIFICATE_NUMBER**

enumerator **TO_INVALID_RESPONSE_LENGTH**

enumerator **TO_SECLINK_ERROR**

enumerator **TO_TIMEOUT**

enumerator **TO_AGAIN**

enumerator **TO_INVALID_PARAM**

enumerator **TO_NOT_IMPLEMENTED**

enumerator **TO_ERROR**

Note: Less significant byte is left empty because it is reserved for Secure Element error codes, then it is possible to return Secure Element and library error codes in one single variable. See *Secure Element error codes*.

7.4.2 Secure Element error codes

group **se_codes**

Typedefs

typedef enum *TO_se_ret_e* **TO_se_ret_t**

Secure Element response codes.

These return codes are common to all TO Secure elements, including the TO-136 and TO-Protect. Therefore, some of these return values may have a different meaning depending on the SE you are using, and the context you are receiving it. Refer yourself to the called function to have a more precise information.

Enums

enum **TO_se_ret_e**

Secure Element response codes.

These return codes are common to all TO Secure elements, including the TO-136 and TO-Protect. Therefore, some of these return values may have a different meaning depending on the SE you are using, and the context you are receiving it. Refer yourself to the called function to have a more precise information.

Values:

enumerator **TORSP_UNKNOWN_CMD**

Indicates that the SE does not know how to handle this command

enumerator **TORSP_BAD_SIGNATURE**
The digital signature is wrong

enumerator **TORSP_INVALID_LEN**
The provided length is wrong

enumerator **TORSP_NOT_AVAILABLE**
The requested data cannot be retrieved

enumerator **TORSP_INVALID_PADDING**
The expected padding is not respected

enumerator **TORSP_COM_ERROR**
A communication error has occurred

enumerator **TO136RSP_COM_ERROR**
Deprecated, use TORSP_COM_ERROR instead

enumerator **TORSP_NEED_AUTHENTICATION**
An authentication process has to be conduced to pursue

enumerator **TORSP_COND_OF_USE_NOT_SATISFIED**
This command cannot be used in this context

enumerator **TORSP_ARG_OUT_OF_RANGE**
An argument is not in the expected range

enumerator **TORSP_SUCCESS**
The Commands execution has been conduced correctly

enumerator **TORSP_SECLINK_RENEW_KEY**
The SecLink key has to be renewed

enumerator **TORSP_INTERNAL_ERROR**
An internal error has occurred. It may be the proof that something unexpected has happened (for instance, a fault has been detected).

7.4.3 Combined error codes

group **error_codes**

Defines

TO_LIB_ERRCODE(errcode) ((errcode) & 0xFF00)
Mask error code to extract library error

TO_SE_ERRCODE(errcode) ((errcode) & 0x00FF)
Mask error code to extract SE error

Typedefs

typedef uint16_t **TO_ret_t**

7.4.4 Generic context

group **tose_defs**

Typedefs

typedef struct *TOSE_drv_ctx_s* **TOSE_drv_ctx_t**
Context structure for Secure Elements (Hardware / Software / other)

typedef struct *TOSE_ctx_s* **TOSE_ctx_t**
Context structure for Secure Elements (Hardware / Software / other)

struct **TOSE_drv_ctx_s**
#include <TO_defs.h> Context structure for Secure Elements (Hardware / Software / other)

Public Members

const struct TODRV_api_s ***api**
Driver API

uint32_t **func_offset**
Offset to add to driver API functions

TO_log_ctx_t ***log_ctx**
Running-platform specific log function

void ***priv_ctx**
Driver private context

struct **TOSE_ctx_s**
#include <TO_defs.h> Context structure for Secure Elements (Hardware / Software / other)

Public Members

TOSE_drv_ctx_t ***drv**
Driver context

uint8_t **initialized**
Context initialization state

7.4.5 Log levels

group `log_level`

Defines

`TO_LOG_LEVEL_NONE` -1
Log levels

`TO_LOG_LEVEL_ERR` 0

`TO_LOG_LEVEL_WRN` 1

`TO_LOG_LEVEL_INF` 2

`TO_LOG_LEVEL_DBG` 3

`TO_LOG_LEVEL_MASK` 0x0f

`TO_LOG_STRING` 0x00

`TO_LOG_BUFFER` 0x10

`TO_LOG_HEX_DISP` 0x20

Typedefs

typedef struct *TO_log_ctx_s* `TO_log_ctx_t`
The Log context, propagated to all layers.

`void() TO_log_func_t (TO_log_ctx_t *log_ctx, const TO_log_level_t level, void *ptr, ...)`
Pointer to a log function.

This function will be responsible for displaying/processing logs in a way suitable for your application.

Enums

enum `TO_log_level_e`
Different log levels that are available to the application.

Values:

Functions

enum TO_log_level_e __attribute__ ((packed)) TO_log_level_t
Different log levels that are available to the application.

void **print_log_function**(const TO_log_level_t level, const char *log)
Default print log function, potentialy to be customized per-target.

Depending on your target and the way to send string messages out, you may have to rewrite it. In this case, just declare a function having the same names/parameters, printing-out the messages.

Parameters

- **level** – Importance level of the message
- **log** – String to be displayed

void **TO_log**(*TO_log_ctx_t* *log_ctx, const TO_log_level_t level, void *ptr, ...)
Default LOG display function.

Parameters

- **log_ctx** – The LOG context
- **level** – The desired log display level
- **ptr** – Pointer to the string (mandatory parameter)

void **TO_set_log_level**(*TO_log_ctx_t* *log_ctx, const TO_log_level_t level, TO_log_func_t *log_function)
Sets the Log function and log level.

This function permits to change the log level and the log function.

Parameters

- **log_ctx** – Current log context
- **level** – Desired log level
- **log_function** – Log function (eg. TO_log)

TO_log_ctx_t ***TO_log_get_ctx**(void)
Get the LOG context.

This function is weak, and can be replaced by your own implementation.

Returns TO_log_ctx_t*

struct **TO_log_ctx_s**
#include <TO_log.h> The Log context, propagated to all layers.

Public Members

TO_log_func_t *`log_function`
Pointer to a log function

TO_log_level_t `log_level`
Dynamic level management

7.4.6 Miscellaneous constants

group `constants`
Misc constants

Defines

`TO_INDEX_SIZE` 1UL

`TO_FORMAT_SIZE` 1UL

`TO_AES_BLOCK_SIZE` 16UL

`TO_INITIALVECTOR_SIZE` TO_AES_BLOCK_SIZE

`TO_AES_KEYSIZE` 16UL

`TO_AESGCM_INITIALVECTOR_SIZE` 12UL

`TO_AESGCM_TAG_SIZE` 16UL

`TO_AESGCM_AAD_LEN_SIZE` 2UL

`TO_AESCCM_NONCE_SIZE` 13UL

`TO_AESCCM_TAG_SIZE` 16UL

`TO_AESCCM_8_TAG_SIZE` 8UL

`TO_AESCCM_AAD_LEN_SIZE` 2UL

`TO_HMAC_KEYSIZE` 16UL

`TO_HMAC_SIZE` TO_SHA256_HASHSIZE

`TO_HMAC_MINSIZE` 10UL

`TO_CMAC_KEYSIZE` 16UL

`TO_CMAC_SIZE` TO_AES_BLOCK_SIZE

`TO_CMAC_MIN_SIZE` 4UL

`TO_SEQUENCE_SIZE` 4UL

`TO_SHA256_HASHSIZE` 32UL

`TO_HASH_SIZE` TO_SHA256_HASHSIZE

`TO_CHALLENGE_SIZE` 32UL

`TO_KEY_FINGERPRINT_SIZE` 3UL

`TO_TIMESTAMP_SIZE` 4UL

`TO_CRC_SIZE` 2UL

`TO_SN_SIZE` (TO_SN_CA_ID_SIZE+TO_SN_NB_SIZE)

`TO_HW_SN_SIZE` 23UL

`TO_SN_CA_ID_SIZE` 3UL

`TO_SN_NB_SIZE` 5UL

`TO_PN_SIZE` 12UL

`TO_HW_VERSION_SIZE` 2UL

`TO_SW_VERSION_SIZE` 3UL

`TO_PRODUCT_ID_SIZE` 15UL

`TO_SEED_SIZE` 32UL

7.4.7 Certificates constants

group `cert_constants`
Certificate constants

Defines

`TO_CERT_X509_MAXSIZE` 512UL

`TO_CERTIFICATE_SIZE` (TO_SN_SIZE+TO_ECC_PUB_KEYSIZE+TO_SIGNATURE_SIZE)

`TO_CERT_PRIVKEY_SIZE` 32UL

`TO_ECC_PRIV_KEYSIZE` TO_CERT_PRIVKEY_SIZE

`TO_ECC_PUB_KEYSIZE` (2*TO_ECC_PRIV_KEYSIZE)

`TO_SIGNATURE_SIZE` TO_ECC_PUB_KEYSIZE

`TO_CERT_GENERALIZED_TIME_SIZE` 15UL /* YYYYMMDDHHMMSSZ */

`TO_CERT_DATE_SIZE` ((TO_CERT_GENERALIZED_TIME_SIZE - 1) / 2)

`TO_CERT_SUBJECT_PREFIX_SIZE` 15UL

`TO_SHORTV2_CERT_SIZE` (TO_CERTIFICATE_SIZE + TO_CERT_DATE_SIZE)

`TO_REMOTE_CERTIFICATE_SIZE` (TO_SN_SIZE+TO_ECC_PUB_KEYSIZE)

`TO_REMOTE_CAID_SIZE` TO_SN_CA_ID_SIZE

`TO_CERT_SUBJECT_CN_MAXSIZE` 64UL

`TO_CERT_SUBJECT_CN_PREFIX_MAXSIZE` (TO_CERT_SUBJECT_CN_MAXSIZE - TO_SN_SIZE * 2)

TO_CERT_DN_MAXSIZE 127UL

TO_KEYTYPE_SIZE TO_SN_CA_ID_SIZE

TO_CA_PUBKEY_SIZE TO_ECC_PUB_KEYSIZE

TO_CA_PUBKEY_CAID_SIZE TO_SN_CA_ID_SIZE

TO_KEY_IDENTIFIER_SIZE 20UL

TO_KEY_IDENTIFIER_SHORT_SIZE 8UL

7.4.8 TLS constants

group **tls_constants**
TLS constants

Defines

TO_TLS_RECORD_CIPHER_OVERHEAD_MAX (352UL)
maximum data overhead between a protected record and its plain text version

The theorical maximum data overhead between a protected record and its plain text version is 1024 but in practice it should not exceed AES256 with SHA512 in CBC mode with max padding: (32 (IV) + 64 (hmac) + 256 (max padding)) => 352 This value is used when the real overhead is not known, to determine if a plain text record to send need to be fragmented before encryption

TO_TLS_RECORD_MAX_SIZE (1UL « 14)

TO_TLS_RANDOM_SIZE (TO_TIMESTAMP_SIZE + 28UL)

TO_TLS_MASTER_SECRET_SIZE 48UL

TO_TLS_SERVER_PARAMS_SIZE 69UL

TO_TLS_HMAC_KEYSIZE 32UL

TO_TLS_FINISHED_SIZE 12UL

TO_TLS_CHANGE_CIPHER_SPEC_SIZE 1UL

TO_TLS_CONNECTION_ID_MAXSIZE 8UL

TO_DTLS_HEADER_SIZE 13UL

TO_DTLS_HEADER_MAXSIZE (TO_DTLS_HEADER_SIZE + TO_TLS_CONNECTION_ID_MAXSIZE)

TO_DTLS_HANDSHAKE_HEADER_SIZE 12UL

TO_DTLS_HANDSHAKE_HEADER_MAXSIZE (TO_DTLS_HANDSHAKE_HEADER_SIZE + TO_TLS_CONNECTION_ID_MAXSIZE)

TO_TLS_HEADER_SIZE 5UL

TO_TLS_HANDSHAKE_HEADER_SIZE 4UL

`TO_TLS_AEAD_IMPLICIT_NONCE_SIZE` 4UL

`TO_TLS_AEAD_EXPLICIT_NONCE_SIZE` 8UL

`TO_DTLS_MAJOR` 254

`TO_DTLS_MINOR` 253

`TO_TLS_MAJOR` 3

`TO_TLS_MINOR` 3

`TO_TLS_SNI_LENGTH_MAX` 253U

Typedefs

typedef enum *TO_tls_mode_e* `TO_tls_mode_t`
Different modes available for TO-Protect TLS.

typedef enum *TO_tls_config_id_e* `TO_tls_config_id_t`

typedef enum *TO_tls_record_type_e* `TO_tls_record_type_t`

typedef enum *TO_tls_cipher_suite_e* `TO_tls_cipher_suite_t`

typedef enum *TO_tls_cipher_suite_type_e* `TO_tls_cipher_suite_type_t`

typedef enum *TO_tls_encryption_type_e* `TO_tls_encryption_type_t`

typedef enum *TO_tls_handshake_type_e* `TO_tls_handshake_type_t`

typedef enum *TO_tls_state_e* `TO_tls_state_t`

typedef enum *TO_tls_extensions_e* `TO_tls_extension_t`

Enums

enum `TO_tls_mode_e`
Different modes available for TO-Protect TLS.

Values:

enumerator `TO_TLS_MODE_UNKNOWN`
Unknown mode (uninitialized)

enumerator `TO_TLS_MODE_HANDSHAKE_ONLY`
Handshake Only mode

enumerator `TO_TLS_MODE_TLS_1_2`
TLS 1.2 only

enumerator `TO_TLS_MODE_TLS_1_2_HANDSHAKE_ONLY`
TLS 1.2, in Handshake only

enumerator **TO_TLS_MODE_DTLS_1_2**
DTLS 1.2 only

enumerator **TO_TLS_MODE_DTLS_1_2_HANDSHAKE_ONLY**
DTLS 1.2, in Handshake only

enum **TO_tls_config_id_e**
Values:

enumerator **TO_TLS_CONFIG_ID_UNKNOWN**

enumerator **TO_TLS_CONFIG_ID_MODE**
Configure mode on 1 byte. See TO_tls_mode_e.

enumerator **TO_TLS_CONFIG_ID_CIPHER_SUITES**
Configure cipher suites list (each cipher suite on 2 bytes, big-endian)

enumerator **TO_TLS_CONFIG_ID_CERTIFICATE_SLOT**
Configure certificate slot

enumerator **TO_TLS_CONFIG_ID_MAX**

enumerator **TO_TLS_CONFIG_ID_LAST**

enum **TO_tls_record_type_e**
Values:

enumerator **TO_TLS_RECORD_TYPE_CHANGE_CIPHER_SPEC**

enumerator **TO_TLS_RECORD_TYPE_ALERT**

enumerator **TO_TLS_RECORD_TYPE_HANDSHAKE**

enumerator **TO_TLS_RECORD_TYPE_APPLICATION_DATA**

enumerator **TO_TLS_RECORD_TYPE_TLS_12_CID**

enum **TO_tls_cipher_suite_e**
Values:

enumerator **TO_TLS_PSK_WITH_AES_128_CBC_SHA256**

enumerator **TO_TLS_PSK_WITH_AES_128_CCM**

enumerator **TO_TLS_PSK_WITH_AES_128_CCM_8**

enumerator **TO_TLS_PSK_WITH_AES_128_GCM_SHA256**

enumerator **TO_TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256**

enumerator **TO_TLS_ECDHE_ECDSA_WITH_AES_128_CCM**

enumerator **TO_TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8**

enumerator **TO_TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256**

enumerator **TO_TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256**

enumerator TO_TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256

enum TO_tls_cipher_suite_type_e
Values:

enumerator TO_TLS_CIPHER_SUITE_ECDHE

enumerator TO_TLS_CIPHER_SUITE_PSK

enum TO_tls_encryption_type_e
Values:

enumerator TO_TLS_ENCRYPTION_AES_CBC

enumerator TO_TLS_ENCRYPTION_AES_CCM

enumerator TO_TLS_ENCRYPTION_AES_CCM_8

enumerator TO_TLS_ENCRYPTION_AES_GCM

enum TO_tls_handshake_type_e
Values:

enumerator TO_TLS_HANDSHAKE_TYPE_CLIENT_HELLO

enumerator TO_TLS_HANDSHAKE_TYPE_SERVER_HELLO

enumerator TO_TLS_HANDSHAKE_TYPE_HELLO_VERIFY_REQUEST

enumerator TO_TLS_HANDSHAKE_TYPE_CERTIFICATE

enumerator TO_TLS_HANDSHAKE_TYPE_SERVER_KEY_EXCHANGE

enumerator TO_TLS_HANDSHAKE_TYPE_CERTIFICATE_REQUEST

enumerator TO_TLS_HANDSHAKE_TYPE_SERVER_HELLO_DONE

enumerator TO_TLS_HANDSHAKE_TYPE_CERTIFICATE_VERIFY

enumerator TO_TLS_HANDSHAKE_TYPE_CLIENT_KEY_EXCHANGE

enumerator TO_TLS_HANDSHAKE_TYPE_FINISHED

enumerator TO_TLS_HANDSHAKE_TYPE_MEDIATOR_CERTIFICATE

enum TO_tls_state_e
Values:

enumerator TO_TLS_STATE_HANDSHAKE_START

enumerator TO_TLS_STATE_FLIGHT_1

enumerator TO_TLS_STATE_CLIENT_HELLO

enumerator TO_TLS_STATE_FLIGHT_1_INIT

enumerator TO_TLS_STATE_FLIGHT_2

enumerator TO_TLS_STATE_SERVER_HELLO_VERIFY_REQUEST

enumerator TO_TLS_STATE_FLIGHT_2_INIT

enumerator TO_TLS_STATE_FLIGHT_3

enumerator TO_TLS_STATE_CLIENT_HELLO_WITH_COOKIE

enumerator TO_TLS_STATE_FLIGHT_3_INIT

enumerator TO_TLS_STATE_FLIGHT_4

enumerator TO_TLS_STATE_SERVER_HELLO

enumerator TO_TLS_STATE_FLIGHT_4_INIT

enumerator TO_TLS_STATE_SERVER_CERTIFICATE

enumerator TO_TLS_STATE_SERVER_KEY_EXCHANGE

enumerator TO_TLS_STATE_SERVER_CERTIFICATE_REQUEST

enumerator TO_TLS_STATE_SERVER_HELLO_DONE

enumerator TO_TLS_STATE_MEDIATOR_CERTIFICATE

enumerator TO_TLS_STATE_FLIGHT_5

enumerator TO_TLS_STATE_CLIENT_CERTIFICATE

enumerator TO_TLS_STATE_FLIGHT_5_INIT

enumerator TO_TLS_STATE_CLIENT_KEY_EXCHANGE

enumerator TO_TLS_STATE_FLIGHT_5_INIT_NO_CLIENT_AUTH

enumerator TO_TLS_STATE_CLIENT_CERTIFICATE_VERIFY

enumerator TO_TLS_STATE_CLIENT_CHANGE_CIPHER_SPEC

enumerator TO_TLS_STATE_CLIENT_FINISHED

enumerator TO_TLS_STATE_FLIGHT_6

enumerator TO_TLS_STATE_SERVER_CHANGE_CIPHER_SPEC

enumerator TO_TLS_STATE_FLIGHT_6_INIT

enumerator TO_TLS_STATE_SERVER_FINISHED

enumerator TO_TLS_STATE_HANDSHAKE_DONE

enumerator TO_TLS_STATE_HANDSHAKE_FAILED

enumerator TO_TLS_STATE_FATAL_RECEIVED

enumerator TO_TLS_STATE_CLOSE_RECEIVED

enum **TO_tls_extensions_e**

Values:

enumerator TO_TLS_EXTENSION_SERVER_NAME

enumerator TO_TLS_EXTENSION_SIG_ALG

enumerator TO_TLS_EXTENSION_ECC

enumerator TO_TLS_EXTENSION_ECC_POINT_FORMAT

enumerator TO_TLS_EXTENSION_TRUNCATED_HMAC

enumerator TO_TLS_EXTENSION_CONNECTION_ID

7.4.9 Admin constants

group **admin_constants**
Admin constants

Defines

TO_ADMIN_DIVERS_DATA_SIZE TO_SN_SIZE

TO_ADMIN_PROTO_INFO_SIZE 4UL

TO_ADMIN_OPTIONS_SIZE 2UL

TO_ADMIN_CHALLENGE_SIZE 8UL

TO_ADMIN_CRYPTOGRAM_SIZE 8UL

TO_ADMIN_MAC_SIZE 8UL

TO_ADMIN_DATAIDX_SIZE 4

8. Miscellany guides

8.1 Production optimizations

8.1.1 Features

In production environment, all unneeded features must be disabled using corresponding configuration flags.

8.1.2 Internal buffers

Internal buffers sizes must be reduced to fit the biggest used size in order to optimize library footprint.

8.1.3 Logs

Logs must be disabled as it can slow down performances, and increase the footprint. It can be done by setting *TO_LOG_LEVEL_MAX* at disabled state.

Note: Changing log level at runtime with `TO_set_log_level()` will fix performance issue but will not reduce the footprint.

See *Library configuration for an MCU project* to properly configure libTO.

8.1.4 Evaluation / Production differences

Evaluation version is delivered for integration purposes only.

Performances are similar in both versions.

Memory footprints are similar in both versions.

Evaluation version has exactly the same functional behavior than production version, but does not come with security protections as in production version:

- Evaluation version has no counter-measures against fault and side-channel attacks,
- Evaluation version has no resistance against reverse-engineering,
- Secure storage is replaced by a single masking with 0xFF,
- All crypto assets will remain unprotected in memory.
- some security breaches are willingly added.

8.2 Migration

8.2.1 TO library migration guide from 5.7.x to 5.8.x

The following changes are to be taken into account to update from 5.7.x to 5.8.x.

TLS helper *TOSE_helper_tls_cleanup()* function is splitted in 2 new functions:

- *TOSE_helper_tls_close()*
- *TOSE_helper_tls_fini()*

Note: *TOSE_helper_tls_cleanup()* can still be used.

8.2.2 TO library migration guide from 5.6.x to 5.7.x

The following changes are to be taken into account to update from 5.6.x to 5.7.x.

CSR function *TOSE_set_certificate_x509()* is now also available with helper `TOSE_helper_set_certificate_x509()` when it needs to be used with a small I2C buffer.

8.2.3 TO library migration guide from 5.5.x to 5.6.x

The following changes are to be taken into account to update from 5.5.x to 5.6.x.

New function *TOSE_get_product_id()* to get product ID.

8.2.4 TO library migration guide from 5.4.x to 5.5.x

The following changes are to be taken into account to update from 5.4.x to 5.5.x.

A user configuration file can now be included by libTO using TO_CONFIG_FILE define.