cv act *library/es*

Cryptographic Library for Embedded Systems

- Technical Information -

Authors: M. Ethen, B. Drisch, T. Zeggel, M. Neuhaus, M. Gnida, S. Stappert and A. Lohoff
cv cryptovision GmbH
Munscheidstr. 14
45886 Gelsenkirchen
Doc-Ref: cv_act_libraryES.doc
Version: 1.2.1 VectorInformatik 1.14
Date: 10. 06. 2011

Contents

1 Introduction

1.1 The cv act *library/es*

This document describes cryptovision's **cv act *library/es***, a cryptographic library specifically developed for the use within embedded systems. The **cv act *library/es*** contains a set of fast and compact implementations of standard cryptographic algorithms.

It should be underlined that for the correct use of the cryptographic functions of this library a deeper understanding of cryptography and the design principles of secure applications is essential. Although the mechanisms provided by the **cv act *library/es*** provide all means usually needed to secure an application, it is strongly suggested to consult experts in information security and cryptography when setting up and implementing a new security concept for an application.

The **cv act *library/es*** can easily be ported to new hardware environments; this has already been done for example for these platforms including both *Harvard* and *von Neumann* architectures:

- (Embedded) PCs (Visual Studio VS 6.0, VS 2003, VS 2005, …)
- NEC V850 (IAR)
- Renesas PowerPC (GreenHills)
- TriCore (Tasking)
- Motorola HC12, Star12 ()
- ARM7 family (GNU)
- Texas Instruments MSP430 (IAR)
- 80C51 family (Keil)
- ATMEL AVR (AVR Studio + AVR-GCC)

For more detailed information please contact cryptovision.

1.2 Scope of this document

This document contains general design information as well as a complete interface description of the **cv act *library/es***. The reader should be familiar with ANSI-C and basic concepts of cryptography.

There is a separate document [4] available that provides a short introduction to modern cryptography with background information about how to employ cryptographic mechanisms; it is especially suited as a first starting point as well as for later reference.

An overview of the calculation times and the resource consumption on different platforms is also available [5]. You can easily use our Performance Estimation Tool (PET) to collect important data about your environment, which can be evaluated by cryptovision to give a good estimation of the performance of your system.

Please consult the changelog located in the library's root folder for detailed information about the history of the library.

1.3 Document organization

This document is structured as follows:

- Chapter 2 explains the general structure and the design principles behind the **cv act *library/es*** and gives a short overview of the cryptographic functions contained in this library.
- Chapter 3 describes the library package and the user interface.
- Chapter 4 gives a reference for the used callback functions.

The document concludes with a table of return codes, abbreviations and glossary, the bibliography and index.

2 Design of the library

2.1 Design principles and goals

The library was designed to be easily portable to most platforms used for embedded systems. It is based on ANSI-C, checked for MISRA conformity, and will usually be delivered as object code. As mentioned in 1.1 the library has already been ported to a number of platforms. Versions for additional processors are available on request.

For applications with special needs regarding speed, special security issues or resource consumption, critical modules of the **cv act *library/es*** can be exchanged on demand by cryptovision with adapted assembly code for a specific target platform, which allows performance improvement or further reduction in memory needs. For detailed information please contact cryptovision.

2.2 Structure of the library

The following figure depicts the conceptual approach behind the **cv act *library/es***:

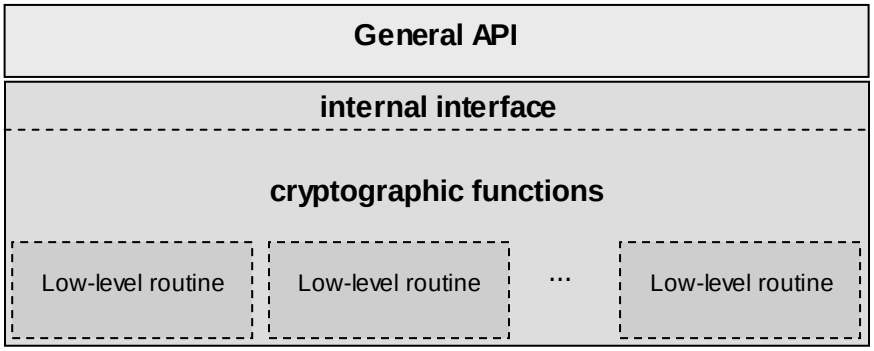

Figure 1: Basic structure of the cv act *library/es*

The API-layer provides an easy-to-use interface for the cryptographic functions listed in chapter 2.4 including all necessary data conversion, sequence handling and padding; a detailed description of this API-layer can be found in chapter 3.

The internal interface provides access to the underlying basic cryptographic primitives. This interface will only be needed for special (non-standard) applications; a description of this interface is not included in this documentation but will be supplied on demand.

The low-level routines pictured above fulfill very basic tasks like finite field arithmetic or arithmetic on elliptic curves. These low-level routines can be replaced by cryptovision with alternative versions optimized for speed or memory needs on demand.

To help you start your **cv act *library/es*** based application the library comes with a testBench to verify the library's calculations on your system and a number of demo applications, which demonstrate the use of most library functions.

The functions for asymmetric cryptographic mechanisms based on elliptic curves (*Elliptic Curve Cryptography, ECC*) over prime fields (*GF(p)*) need ECC domain parameters. Some standard data is included in our testBench and Performance Estimation Tool. There are tools available from cryp-

tovision that compute these domain parameters in advance and produce .c-files including the necessary data-fields. For further details see section 3.12.1.

If you are using RSA based algorithms the appropriate keys need to be generated or converted to be included in your project. A tool to generate keys ready to include in your application is available from cryptovision.

2.3 Design details

The **cv act *library/es*** was designed to be thread-safe, that means the calculations provide correct results even if parallel processes are active on the processor. This is realized by a wide abandonment of global variables. Only for the implemented random number module (cf. 3.18), the user has to take care when using it in a multithreaded application, because random numbers are used by different cryptographic primitives. Memory needed by the library (in following called "workSpace") must be provided by the user by giving a pointer to this memory area of the appropriate size. The workSpace contains the temporary used data and information about the current state of the algorithm as well as checksums and length information. The necessary size of the workSpaces for each cryptographic function can be found in the corresponding header file. The memory demands can be optimized by using special assembly code versions of these functions; please contact cryptovision for further information. The input and output values of the cryptographic functions can be integrated into most standard environments.

2.4 Functional overview

There are different classes of cryptographic mechanisms that can be used according to the needs of a specific application. Typical mechanisms are:

- symmetric encryption algorithms using the same key for encryption and decryption, usually based on very efficient operations and thus leading to high data throughput,
- hash algorithms converting a data input of arbitrary length to a cryptographically secure checksum of a fixed length,
- message authentication codes converting a data input of arbitrary length (message) together with a key into a cryptographically secure checksum of fixed length, and
- asymmetric algorithms basically being encryption algorithms with two different keys for encryption and decryption, referred to as *public* and *private* key.

Asymmetric algorithms can be used for different purposes; the most important ones are

- digital signature, in which a data input of arbitrary length and a private key are used to calculate the signature, which can be verified with the public key of the key pair, and
- key exchange, which allows the secure exchange of a common secret (usually a secret key for further communication) between two communication partners.

For further information about these mechanisms and the significance and comparability of parameters like key size, please take a look at our introduction to modern cryptography [4].

The **cv act *library/es*** in its present version contains the following algorithms:

2.4.1 Symmetric algorithms and additional mechanisms

- encryption/decryption using the Advanced Encryption Standard (AES) in various modes of operation and a key length of 128, 192 and 256 bit according to FIPS 197 [9].

- encryption/decryption using the Data Encryption Standard (DES) in various modes of operation and a key length of 64 bit according to FIPS 46-3 [22].
- encryption/decryption using the Triple Data Encryption Standard (TDES) in various modes of operation and a key length of 192 bit according to FIPS 46-3 [22].
- encryption/decryption using the Rivest Cipher 2 (RC2) in various modes of operation according to RFC 2268 [9].
- Hash calculation with the Secure Hash Algorithm SHA-1 according to FIPS 180-2 [7].
- Hash calculation with the Secure Hash Algorithm SHA-256 according to FIPS 180-2 [7].
- Hash calculation with the RIPEMD 160 according to ISO/IEC 10118-3 [12].
- HMAC based message authentication using SHA-1 according to FIPS 198 [10].
- HMAC based message authentication using RIPEMD 160 according to FIPS 198 [10].
- The key derivation function KDF2 based on HMAC-SHA1 according to [2].
- The key derivation function KDF X9.63 based on HMAC-SHA1 according to [2].
- Pseudo random number generation (based on AES) according to FIPS 186-2 [8].

The implemented pseudo random number generator (PRNG) can – if appropriately seeded by a true random source on the target platform – be used for the key generation procedures and cryptographic protocols. See [4] for further information on random number generation and its importance for cryptography.

2.4.2 Asymmetric algorithms – ECC and RSA

- Digital signatures based on the Elliptic Curve Digital Signature Algorithm EC-DSA with key lengths of 128-521 bit according to ANSI X9.62 [1].
- Key exchange using the Elliptic Curve Diffie-Hellman protocol EC-DH with key lengths of 128-521 bit according to ANSI X9.63 [2].
- Digital signatures based on RSA with key length of 512-2048 bit according to PKCS #1 [19].
- Asymmetric encryption and decryption based on RSA with key length of 512-2048 bit according to PKCS #1 [19].

3 API of the library

3.1 General concept

In general each cryptographic operation is executed performing the three following steps:

- initialization,
- calculation and
- finalization (if necessary).

Not all of the cryptographic operations use all these three steps, some functions are omitted if possible (e.g. the AES single block encryption/decryption has no finalization step).

During the initialization step the necessary (internal) data structures are set up. The memory required for these structures must be provided by the user. These data must not be changed (or de-allocated) until successful finalization. If the initialization functions contain parameters like key components or domain parameters, these must not be deleted until the corresponding finalization step has been executed. These parameters are linked into the workspace and assumed to be available until the finalization. Using malloc and free for such variables requires very careful handling.

The calculation step performs the actual cryptographic operation and typically allows setting various parameters (like the mode of operation for AES).

In the finalization step the results of the operation are returned. In some operations (e.g. SHA-1) padding is done in this stage as well.

3.2 Representation of cryptographic data

All numbers are handled as strings of bytes in big endian byte order, as this is usually preferred by ASN.1 coding (cf. [13]). The output of the functions is formatted according to ASN.1-DER coding (cf. [14]), that means, that the output can be placed in ASN.1-encoded arrays without inconvenient reformatting. The input of the functions is expected to be formatted as described in this document. The typical ASN.1-DER coded format is a subset of the input requirements.

Please note that some values (RSA: n, p, q; ECC: p, n) are defined to be signed integers and thus the MSB has to be zero for positive values.

If pointers to single values (e.g. a public key) are passed into functions they are always expected to point to the value itself, not to the leading tag or length field. The length is known implicitly or passed separately.

3.3 Basic data types and constants

This section explains the basic data types and constants used in the **cv act *library/es***:

3.3.1 Ordinal types

Data type	Remark
`eslt_Byte`	unsigned 8 bit value
`eslt_Length`	length of a eslt_Byte string
`eslt_Mode`	If a cryptographic primitive supports different modes, the mode to use is defined by a parameter of this type. Further information is provided in the respective function's description.
`eslt_ErrorCode`	Return value indicating the correct execution of a function or an error. A return value of 0 (zero) means, that the operation was executed successfully. Common return values are listed on page 101, individual error codes are specified in the respective function's description.

3.3.2 Compound data types

Data type	Remark
`eslt_WorkSpaceXYZ`	This type represents memory required for internal data structures. The user must provide this memory to the functions. In a "real world" application this memory will be allocated on the stack (local variable) or on the heap (via `malloc()`). For each initialization function a constant called ESL_MAXSIZEOF_WS_XYZ exists. (They are named with "…MAX…" because of a possible optimization of the value under special circumstances, e.g. if a length of a byte array to be encrypted is a multiple of a block length the algorithm uses less bytes. For further details contact cryptovision.)
`eslt_WorkSpaceHeader`	This type is the first part of each `eslt_workSpaceXYZ`-Type in this library, it contains status information, size information and a checksum over important data that have to be constant.
`eslt_EccDomain`	This type contains the domain parameters used for the Elliptic Curve primitives as defined in X9.62-ASN.1-Type Module ECParameters (cf. [1]).
`eslt_EcPDomainExt`	This type is used to store additional information about the domain parameters. This information depends on the used arithmetic.
`eslt_EcPSpeedUpExt`	This type is used to store further pre-computed values, depending on the used arithmetic that is used to speed up algorithms for ECC signature and key generation.

3.3.3 Constants

The following constants are used in the **cv act *library/es***:

Name	Value	Remark
ESL_SIZEOF_AES128_KEY	16	The byte length of an AES128, AES192,

ESL_SIZEOF_AES192_KEY ESL_SIZEOF_AES256_KEY	24 32	AES256, DES, TDES and RC2 key.
ESL_SIZEOF_DES_KEY ESL_SIZEOF_2TDES_KEY ESL_SIZEOF_3TDES_KEY	8 16 24	
ESL_SIZEOF_RC2_KEY ESL_SIZEOF_RC2_EFF_KEY	16 16	The key length values for RC2 are just recommended. Key lengths are variable according to RFC 2268.
ESL_SIZEOF_AES128_BLOCK ESL_SIZEOF_AES192_BLOCK ESL_SIZEOF_AES256_BLOCK	16	The byte length of a data block of an AES128, AES192, AES256, DES, RC2 operation. The AES block size is identical for all key lengths.
ESL_SIZEOF_DES_BLOCK ESL_SIZEOF_RC2_BLOCK	8 8	
ESL_SIZEOF_SHA1_BLOCK ESL_SIZEOF_SHA256_BLOCK ESL_SIZEOF_RIPEMD160_BLOCK	64	The byte length of a data block of a SHA-1/SHA-256/RIPEMD160 operation.
ESL_SIZEOF_SHA1_DIGEST ESL_SIZEOF_SHA256_DIGEST ESL_SIZEOF_RIPEMD160_DIGEST	20 32 20	The byte length of a SHA-1/SHA-256/RIPEMD-160 hash value.
ESL_SIZEOF_RNGFIPS186_STATE	20	The byte length of the state of the random number generator according to FIPS186 (SHA-1 based).
ESL_KDF2_DEFAULT_ITERATION_COUNT	1	The default iteration count for the key derivation function.

3.3.4 Defines for constant data in ROM

On Harvard architectures ECC domain parameters and RSA keys are assumed to be located in program memory. The qualifiers `ROM` an `ROM_POST` are defined according to the compilers specifications and should be used to place these constants into ROM. Special care needs to be taken with some compilers as pointers into RAM and ROM are indistinguishable and need to be used correctly by the user.

Example

```
ROM_CONST unsigned char esl_EccDomain [] ROM_POST = {
     0x30,0x81,0xb0, … } ;
```

3.4 General initialization

The library in total as well as the memory used by each of the algorithms needs to be initialized before any calculations can be done.

3.4.1 Library initialization

The subsequent initialization routine has to be called once before using any of the library functions.

Prototype

```
eslt_ErrorCode esl_initESCryptoLib(void);
```

Return codes

Return code	Remark
ESL_ERC_NO_ERROR	The function executed successfully.
ESL_ERC_ERROR	A general error occurred.

3.4.2 WorkSpace initialization

Each function needs some temporary memory (the so called workSpace) of an appropriate size. This memory area must be first initialized with a common initialization routine. It expects a pointer to a workSpaceHeader of a subsequent workSpace (e.g. workSpaceAES.workSpaceHeader for AES). It also needs the appropriate size (e.g. ESL_MAXSIZEOF_WS_AES128) of the workSpace in order to initialize the header correctly.

Please note that ESL_MAXSIZEOF_WS_AES128 holds only for the data area inside the work-Space and hence cannot be used with malloc.

The library can be configured to include a watchdog-functionality at compile-time. In this case a watchdog-function can be passed to the initialization function, which will then be called from inside the library functions. A NULL-pointer disables this.

Prototype

```
eslt_ErrorCode esl_initWorkSpaceHeader(
                  eslt_workSpaceHeader *workSpaceHeader,
          const   eslt_Length           workSpaceSize,
                  void                 (*watchdog) (void) );
```

Return codes

```
ESL_ERC_NO_ERROR, ESL_ERC_PARAMETER_INVALID
```

Example

```
eslt_WorkSpaceAES256Block *workSpace_AES256;

workSpace_AES256 = malloc ( sizeof (eslt_WorkSpaceAES256Block) );

result = esl_initWorkSpaceHeader (&workSpace_AES256->header,
                                  ESL_MAXSIZEOF_WS_AES256BLOCK,
                                  &watchdog);

if ( result!=ESL_ERC_NO_ERROR ) return result;
```

Note: All Input- and Output-Parameter must lie outside the current used workSpace.

3.5 AES encryption/decryption

The following functions provide AES encryption and decryption according to FIPS PUB 197 [9].

Two variants of AES are provided, single block mode encryption for proprietary use without additional features like padding, and block mode encryption with some commonly used modes of operation like ECB, CBC, CFB and OFB. For CBC, CFB and OFB modes a random initialization vector must be provided (cf. [6]).

The AES key length can be chosen to be 128, 192 or 256 bit. While the following description of the AES functions only explains the 128 bit variant, the other variants are used exactly the same way. The respective function prototypes are named `~AES192` and `~AES256`, respectively.

The sizes of the workSpaces for these functions are defined by

```
ESL_MAXSIZEOF_WS_AES128
ESL_MAXSIZEOF_WS_AES128BLOCK

ESL_MAXSIZEOF_WS_AES192
ESL_MAXSIZEOF_WS_AES192BLOCK

ESL_MAXSIZEOF_WS_AES256
ESL_MAXSIZEOF_WS_AES256BLOCK
```

3.5.1 AES encryption initialization

This function carries out the necessary initialization for a subsequent encryption operation.

Note: The initialization functions for the 192-bit-variant and the 256-bit-variant are called `esl_initEncryptAES192` and `esl_initEncryptAES256`, respectively.

Prototype

```
eslt_ErrorCode esl_initEncryptAES128 (
                eslt_WorkSpaceAES128 *workSpace,
     const      eslt_Byte            *key,
     const      eslt_BlockMode        blockMode,
     const      eslt_PaddingMode      paddingMode,
     const      eslt_Byte            *initializationVector );
```

Description

This function implements the initialization routine for the AES encryption.

Input

Parameter	Remark
`workSpace`	A pointer to a memory area (context) that has to be at least ESL_MAXSIZESIZEOF_WS_AES128 bytes long.
`*key`	The key for the AES operation with the length of ESL_SIZEOF_AES128_KEY (= 16) bytes.

`blockMode`	One of the following constants has to be passed to choose the block mode (ECB, CBC, CFB or OFB):
	`ESL_BM_ECB`
	`ESL_BM_CBC.`

	The modes `ESL_BM_CFB` and `ESL_BM_OFB` are not yet implemented but can be added if needed. For further details please contact cryptovision.
`paddingMode`	The padding mode has to be `ESL_AES_PM_PKCS5`. The modes `ESL_AES_PM_OFF` and `ESL_AES_PM_ONEWITHZEROES` are not yet implemented but can be added if needed. For further details please contact cryptovision.
`initializationVector`	A pointer to an initialization vector for the AES operation.

Output

None.

Return codes

Return code	Remark
`ESL_ERC_MODE_INVALID`	The chosen mode is not supported.
`ESL_ERC_MODE_INCOMPATIBLE`	The combination of padding and block mode is not valid.
`ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL`	

Prerequisites

The general initialization function esl_initWorkSpaceHeader (cf. 3.4) must have been executed successfully before.

3.5.2 AES encryption

This function provides a high level interface to the AES encryption including different modes of operation. It concatenates succeeding inputs and computes the encryption of this input stream. The output of these functions can be concatenated to give the encrypted stream.

The optional padding is handled by the finalize function (cf. 3.5.3).

Note: The encryption functions for the 192-bit-variant and the 256-bit-variant are called `esl_encryptAES192` and `esl_encryptAES256`, respectively.

Prototype

```
eslt_ErrorCode esl_encryptAES128(
                     eslt_WorkSpaceAES128 *workSpace,
               const eslt_Length           inputSize,
               const eslt_Byte            *input,
                     eslt_Length          *outputSize,
                     eslt_Byte            *output      );
```

Description

This function implements the core part of the AES encryption: It reads input length bytes from input, appends them to the inputs from the preceding call to this function and encrypts as many blocks as possible. The remaining bytes are stored in an internal buffer.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initEncryptAES128.
inputSize	The number of bytes in input that are to be encrypted.
input	A pointer to the input that has to be encrypted. In case that preceding inputs had lengths different from multiples of the block length, the input bytes are appended to the preceding bytes and then encrypted. Remaining bytes are stored internally.
*outputSize	The number of bytes that can be used at output for storing the encrypted input. This size must be longer than inputSize by ESL_SIZEOF_AES_BLOCKLENGTH to guarantee, that the function can store all the encrypted output necessary. The reason for this precaution are remaining bytes from preceding inputs that can result in an additional output block. If all inputs have lengths that are multiples of ESL_SIZEOF_AES_BLOCKLENGTH this additional area is not necessary. Input and output can use the same memory, but if the pointers differ the memory must not overlap.
output	A pointer to outputSize bytes of free memory where the output can be stored.

Output

Parameter	Remark
*outputSize	The number of bytes that are really used at output for storing the encrypted input.
*output	The encrypted input.

Return codes

Return code	Remark
ESL_ERC_OUTPUT_SIZE_TOO_SHORT	The given output length is too short. Another call can be done with less input or more outputSize.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID, ESL_ERC_MEMORY_CONFLICT	

Prerequisites

The initialization function must have been executed successfully before (cf. 3.5.1).

3.5.3 AES finalize encryption

This function carries out the necessary finalization for a previous AES encryption.

Note: The finalize functions for the 192-bit-variant and the 256-bit-variant are called esl_finalyzeEncryptAES192 and esl_finalizeEncryptAES256, respectively.

Prototype

```
eslt_ErrorCode esl_finalizeEncryptAES128(
                  eslt_WorkSpaceAES128 *workSpace,
                  eslt_Length          *outputSize,
                  eslt_Byte            *output        );
```

Description

This function implements the finalization routine for an AES encryption. It especially does the necessary padding, which is needed for some modes of operation.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initEncryptAES128.
*outputSize	The number of bytes that can be used at output for storing the encryption of the input and the padding. For execution of this function 2*ESL_SIZEOF_AES_BLOCK is sufficient in any case.
output	A pointer to outputSize bytes of free memory where the output will be stored.

Output

Parameter	Remark
*outputSize	The byte length of the last output block; always exactly one or two blocks.
*output	The encrypted last one or two blocks.

Return codes

Return code	Remark
ESL_ERC_OUTPUT_SIZE_TOO_SHORT	The given output length is too short. Another call can be done with less input or more outputSize.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID, ESL_ERC_MEMORY_CONFLICT	

Prerequisites

The initialization function and AES encryption operation must have been executed successfully before (cf. 3.5.1 and 3.5.2).

3.5.4 AES decryption initialization

This function carries out the necessary initialization for a subsequent decryption operation.

Note: The initialization functions for the 192-bit-variant and the 256-bit-variant are called esl_initDecryptAES192 and esl_initDecryptAES256, respectively.

Prototype

```
eslt_ErrorCode esl_initDecryptAES128 (
                  eslt_WorkSpaceAES128 *workSpace,
```

```
const      eslt_Byte            *key,
const      eslt_BlockMode        blockMode,
const      eslt_PaddingMode      paddingMode,
const      eslt_Byte            *initializationVector );
```

Description

This function implements the initialization routine for the AES decryption.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that has to be at least ESL_MAXSIZEOF_WS_AES128 bytes long.
*key	The key for the AES operation with the length of ESL_SIZEOF_AES128_KEY (= 16) bytes.
blockMode	One of the following constants has to be passed to choose the block mode (ECB, CBC, CFB or OFB): ESL_BM_ECB ESL_BM_CBC. The modes ESL_BM_CFB and ESL_BM_OFB are not yet implemented but can be added if needed. For further details please contact cryptovision.
paddingMode	The padding mode has to be ESL_AES_PM_PKCS5. The modes ESL_AES_PM_OFF and ESL_AES_PM_ONEWITHZEROES are not yet implemented but can be added if needed. For further details please contact cryptovision.
initializationVector	A pointer to an initialization vector for the AES operation.

Output

None.

Return codes

Return code	Remark
ESL_ERC_MODE_INVALID	The chosen mode is not supported.
ESL_ERC_MODE_INCOMPATIBLE	The combination of padding and block mode is not valid.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The general initialization function esl_initWorkSpaceHeader (cf. 3.4) must have been executed successfully before.

3.5.5 AES decryption

This function provides a high level interface to the AES decryption including different modes of operation. It concatenates many succeeding inputs and computes the decryption of this input stream. The output of these functions can be concatenated to give the decrypted stream.

The padding of the output will be checked and not passed out by the finalize function (cf. 3.5.6).

Note: The decryption functions for the 192-bit-variant and the 256-bit-variant are called `esl_decryptAES192` and `esl_decryptAES256`, respectively.

Prototype

```
eslt_ErrorCode esl_decryptAES128(
                    eslt_WorkSpaceAES128 *workSpace,
              const eslt_Length           inputSize,
              const eslt_Byte            *input,
                    eslt_Length          *outputSize,
                    eslt_Byte            *output       );
```

Description

This function implements the core part of the AES decryption: It reads input length bytes from input, appends them to the inputs from the preceding call to this function and decrypts as many blocks as possible. The remaining bytes are stored in an internal buffer.

Input

Parameter	Remark
`workSpace`	A pointer to a memory area (context) that was initialized by esl_initDecryptAES128.
`inputSize`	The number of bytes of the input that are to be decrypted.
`input`	A pointer to a part of input, that has to be decrypted. In case that preceding inputs had lengths different from multiples of the block length, the input bytes are appended to the preceding bytes and then decrypted. Remaining bytes are stored internally.
`*outputSize`	The number of bytes that can be used at output for storing the decryption of the input. This size must be longer than inputSize by ESL_SIZEOF_AES128_BLOCK to guarantee, that the function can store all the decrypted output necessary. The reason for this precaution are remaining bytes from preceding inputs that can result in an additional output block. If all inputs have lengths that are multiples of ESL_SIZEOF_AES128_BLOCK this additional space is not necessary.
`output`	A pointer to the memory where the decrypted input will be stored. Here esl_decryptAES128 stores the decrypted message blocks that can be computed from already provided input bytes.

Output

Parameter	Remark
`*outputSize`	The number of bytes that are really used at output for storing the decryption of the input.
`*output`	The decrypted input.

Return codes

Return code	Remark
`ESL_ERC_OUTPUTLENGTH_TOO_SHORT`	The given output length is too short. Another call can be done with less input or more outputSize.
`ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID, ESL_ERC_MEMORY_CONFLICT`	

Prerequisites

The initialization function must have been executed successfully before (cf. 3.5.4).

3.5.6 AES finalize decryption

This function carries out the necessary finalization for a previous AES decryption.

Note: The finalization functions for the 192-bit-variant and the 256-bit-variant are called `esl_finalizeDecryptAES192` and `esl_finalizeDecryptAES256`, respectively.

Prototype

```
eslt_ErrorCode esl_finalizeDecryptAES128(
                  eslt_WorkSpaceAES128 *workSpace,
                  eslt_Length          *outputSize,
                  eslt_Byte            *output    );
```

Description

This function implements the finalization routine for an AES encryption. It especially checks the padding at the end of the decrypted data.

Input

Parameter	Remark
`workSpace`	A pointer to a memory area (context) that was initialized by esl_initDecryptAES128.
`*outputSize`	The number of bytes that can be used at output for storing the decryption of the input. To guarantee correct execution a value of ESL_SIZEOF_AES128_BLOCK is sufficient in any case.
`output`	A pointer to outputSize bytes of free memory where the output will be stored (without the padding).

Output

Parameter	Remark
`*outputSize`	The byte length of the output.
`*output`	The decrypted last blocks.

Return codes

Return code	Remark
ESL_ERC_OUTPUT_SIZE_TOO_SHORT	The given output length is too short.
ESL_ERC_INPUT_INVALID	The complete length of the input is not a multiple of the block length.
ESL_ERC_AES_PADDING_INVALID	The padding of the input is invalid.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID, ESL_ERC_MEMORY_CONFLICT	

Prerequisites

The initialization function and AES decryption operation with non-zero total length must have been executed successfully before (cf. 3.5.1 and 3.5.2).

3.5.7 AES single block encryption initialization

This function carries out the necessary initialization for a subsequent encryption operation in single block mode. It requires less memory than its more general pendant.

Note: The initialization functions for the 192-bit-variant and the 256-bit-variant are called `esl_initEncryptAES192Block` and `esl_initEncryptAES256Block`, respectively.

Prototype

```
eslt_ErrorCode esl_initEncryptAES128Block (
                  eslt_WorkSpaceAES128Block *workSpace,
      const       eslt_Byte                 *key       );
```

Description

This function implements the initialization routine for the AES encryption in single block mode.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that has to be at least ESL_MAXSIZEOF_WS_AES128BLOCK bytes long.
key	A pointer to a key for AES with the length of ESL_SIZEOF_AES128_KEY (=16) bytes.

Output

None.

Return codes

Return code	Remark
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The general initialization function esl_initWorkSpaceHeader (cf. 3.4) must have been executed successfully before.

3.5.8 AES single block encryption

This function provides the AES encryption of a single block; it performs no padding or modes of operation.

Note: The encryption functions for the 192-bit-variant and the 256-bit-variant are called `esl_EncryptAES192Block` and `esl_EncryptAES256Block`, respectively.

Prototype

```
eslt_ErrorCode esl_encryptAES128Block(
                  eslt_WorkSpaceAES128Block   *workSpace,
      const       eslt_Byte                   *inputBlock,
                  eslt_Byte                   *outputBlock   );
```

Description

This function implements the AES encryption of a single block (16 Bytes). As it is strictly single block oriented, and does not pad the data automatically, this function has no finalize pendant.

Input

Parameter	Remark
`workSpace`	A pointer to a memory area (context) that was initialized by esl_initEncryptAES128Block. Neither input nor output must point to memory inside this area.
`inputBlock`	This pointer must reference exactly one AES Block (ESL_SIZEOF_AES128_BLOCK bytes).
`outputBlock`	A pointer to a memory area with enough space for the encrypted output block (ESL_SIZEOF_AES128_BLOCK bytes). Input and output can use the same memory, but if the pointers differ the memory must not overlap.

Output

Parameter	Remark
`*outputBlock`	The encrypted input block.

Return codes

Return code	Remark
`ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID, ESL_ERC_MEMORY_CONFLICT`	

Prerequisites

The initialization function must have been executed successfully before (cf. 3.5.7).

3.5.9 AES single block decryption initialization

This function carries out the necessary initialization for a subsequent decryption operation in single block mode. It requires less memory than its more general pendant.

Note: The initialization functions for the 192-bit-variant and the 256-bit-variant are called `esl_initDecryptAES192Block` and `esl_initDecryptAES256Block`, respectively.

Prototype

```
eslt_ErrorCode esl_initDecryptAES128Block (
                eslt_WorkSpaceAES128Block *workSpace,
      const     eslt_Byte                 *key          );
```

Description

This function implements the initialization routine for the AES decryption in single block mode.

Input

Parameter	Remark
`workSpace`	A pointer to a memory area (context) that has to be at least ESL_MAXSIZEOF_WS_AES128_BLOCK bytes long.
`key`	A pointer to a key for AES with the length of ESL_SIZEOF_AES128_KEY (= 16) bytes.

Output

None.

Return codes

Return code	Remark
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The general initialization function esl_initWorkSpaceHeader (cf. 3.4) must have been executed successfully before.

3.5.10 AES single block decryption

This function provides direct access to the AES decryption of a single block; it therefore performs no padding or modes of operation handling.

Note: The decryption functions for the 192-bit-variant and the 256-bit-variant are called `esl_decryptAES192Block` and `esl_decryptAES256Block`, respectively.

Prototype

```
eslt_ErrorCode esl_decryptAES128Block(
                eslt_WorkSpaceAES128Block   *workSpace,
      const     eslt_Byte                   *inputBlock,
                eslt_Byte                   *outputBlock  );
```

Description

This function implements the AES decryption of a single block (16 Bytes). As it is strictly single block oriented, and does not pad the data automatically, this function has no finalize pendant.

Input

Parameter	Remark
`workSpace`	A pointer to a memory area (context) that was initialized by esl_initDecryptAES128Block.
`inputBlock`	This pointer must reference exactly one AES Block (ESL_SIZEOF_AES128_BLOCK bytes).
`outputBlock`	A pointer to a memory area with enough space for the decrypted output block (ESL_SIZEOF_AES128_BLOCK bytes). Input and output can use the same memory, but if the pointers differ the memory must not overlap.

Output

Parameter	Remark
`*outputBlock`	The decrypted input block.

Return codes

Return code	Remark
`ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID, ESL_ERC_MEMORY_CONFLICT`	

Prerequisites

The initialization function must have been executed successfully before (cf. 3.5.7).

3.5.11 AES key generation

This function provides the AES key generation operation.

Note: The key generation functions for the 192-bit-variant and the 256-bit-variant are called `esl_generateKeyAES192` and `esl_generateKeyAES256`, respectively.

Prototype

```
eslt_ErrorCode esl_generateKeyAES128(
            eslt_Byte              *key      );
```

Description

This function generates a key for AES128 and stores the key at the memory area referenced by key. It can be called without an initialization function. The random number generator must have been initialized before calling this function (cf. 3.18 and 4.1).

Input

Parameter	Remark
`key`	A pointer to a memory area for the key to be generated

	(ESL_SIZEOF_AES128_KEY Bytes).

Output

Parameter	Remark
*key	The generated key for AES encryption or decryption.

Return codes

Return code	Remark
ESL_ERC_NO_ERROR	
Additionally all possible return codes from the function esl_getBytesRNG can be returned from this function (cf. 4).	

Prerequisites

The random number module must have been initialized successfully before (cf. 3.18.1). Especially the callback function must be provided by the user (cf. 4).

3.6 DES encryption/decryption

The following functions provide DES encryption and decryption according to FIPS PUB 46-3 [9].

Two variants of DES are provided, single block mode encryption for proprietary use without additional features like padding, and block mode encryption with some commonly used modes of operation like ECB and CBC. For CBC mode a random initialization vector must be provided (cf. [6]).

The sizes of the workSpace for this function is defined by

```
ESL_MAXSIZEOF_WS_DES
ESL_MAXSIZEOF_WS_DESBLOCK
```

3.6.1 DES encryption initialization

This function carries out the necessary initialization for a subsequent encryption operation.

Prototype

```
eslt_ErrorCode esl_initEncryptDES (
                 eslt_WorkSpaceDES    *workSpace,
      const      eslt_Byte            *key,
      const      eslt_BlockMode        blockMode,
      const      eslt_PaddingMode      paddingMode,
      const      eslt_Byte            *initializationVector );
```

Description

This function implements the initialization routine for the DES encryption.

Input

Parameter	Remark

workSpace	A pointer to a memory area (context) that has to be at least ESL_MAXSIZESIZEOF_WS_DES bytes long.
*key	The key for the DES operation with the length of ESL_SIZEOF_DES_KEY (= 8) bytes.
blockMode	One of the following constants has to be passed to choose the block mode (ECB, CBC, CFB or OFB): ESL_BM_ECB ESL_BM_CBC. The modes ESL_BM_CFB and ESL_BM_OFB are not yet implemented but can be added if needed. For further details please contact cryptovision.
paddingMode	The padding mode has to be ESL_PM_PKCS5. The modes ESL_PM_OFF and ESL_PM_ONEWITHZEROES are not yet implemented but can be added if needed. For further details please contact cryptovision.
initializationVector	A pointer to an initialization vector for the DES operation.

Output

None.

Return codes

Return code	Remark
ESL_ERC_MODE_INVALID	The chosen mode is not supported.
ESL_ERC_MODE_INCOMPATIBLE	The combination of padding and block mode is not valid.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL	

Prerequisites

The general initialization function esl_initWorkSpaceHeader (cf. 3.4) must have been executed successfully before.

3.6.2 DES encryption

This function provides a high level interface to the DES encryption including different modes of operation. It concatenates succeeding inputs and computes the encryption of this input stream. The output of these functions can be concatenated to give the encrypted stream.

The optional padding is handled by the finalize function (cf. 3.6.3).

Prototype

```
eslt_ErrorCode esl_encryptDES(
                    eslt_WorkSpaceDES    *workSpace,
              const eslt_Length           inputSize,
              const eslt_Byte            *input,
                    eslt_Length          *outputSize,
                    eslt_Byte            *output      );
```

Description

This function implements the core part of the DES encryption: It reads input length bytes from input, appends them to the inputs from the preceding call to this function and encrypts as many blocks as possible. The remaining bytes are stored in an internal buffer.

Input

Parameter	Remark
`workSpace`	A pointer to a memory area (context) that was initialized by esl_initEncryptDES.
`inputSize`	The number of bytes in input that are to be encrypted.
`input`	A pointer to the input that has to be encrypted. In case that preceding inputs had lengths different from multiples of the block length, the input bytes are appended to the preceding bytes and then encrypted. Remaining bytes are stored internally.
`*outputSize`	The number of bytes that can be used at output for storing the encrypted input. This size must be longer than inputSize by `ESL_SIZEOF_DES_BLOCK` to guarantee, that the function can store all the encrypted output necessary. The reason for this precaution are remaining bytes from preceding inputs that can result in an additional output block. If all inputs have lengths that are multiples of `ESL_SIZEOF_DES_BLOCK` this additional area is not necessary. Input and output can use the same memory, but if the pointers differ the memory must not overlap.
`output`	A pointer to outputSize bytes of free memory where the output can be stored.

Output

Parameter	Remark
`*outputSize`	The number of bytes that are really used at output for storing the encrypted input.
`*output`	The encrypted input.

Return codes

Return code	Remark
`ESL_ERC_OUTPUT_SIZE_TOO_SHORT`	The given output length is too short. Another call can be done with less input or more outputSize.
`ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID, ESL_ERC_MEMORY_CONFLICT`	

Prerequisites

The initialization function must have been executed successfully before (cf. 3.5.1).

3.6.3 DES finalize encryption

This function carries out the necessary finalization for a previous DES encryption.

Prototype

```
eslt_ErrorCode esl_finalizeEncryptDES(
                   eslt_WorkSpaceDES    *workSpace,
                   eslt_Length          *outputSize,
                   eslt_Byte            *output        );
```

Description

This function implements the finalization routine for an DES encryption. It especially does the necessary padding, which is needed for some modes of operation.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initEncryptDES.
*outputSize	The number of bytes that can be used at output for storing the encryption of the input and the padding. For execution of this function 2*ESL_SIZEOF_DES_BLOCK is sufficient in any case.
output	A pointer to outputSize bytes of free memory where the output will be stored.

Output

Parameter	Remark
*outputSize	The byte length of the last output block; always exactly one or two blocks.
*output	The encrypted last one or two blocks.

Return codes

Return code	Remark
ESL_ERC_OUTPUT_SIZE_TOO_SHORT	The given output length is too short. Another call can be done with less input or more outputSize.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID, ESL_ERC_MEMORY_CONFLICT	

Prerequisites

The initialization function and DES encryption operation must have been executed successfully before (cf. 3.6.1and 3.6.2).

3.6.4 DES decryption initialization

This function carries out the necessary initialization for a subsequent decryption operation.

Prototype

```
eslt_ErrorCode esl_initDecryptDES (
                 eslt_WorkSpaceDES    *workSpace,
      const      eslt_Byte            *key,
      const      eslt_BlockMode        blockMode,
      const      eslt_PaddingMode      paddingMode,
      const      eslt_Byte            *initializationVector );
```

Description

This function implements the initialization routine for the DES decryption.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that has to be at least ESL_MAXSIZEOF_WS_DES bytes long.
*key	The key for the DES operation with the length of ESL_SIZEOF_DES_KEY (= 8) bytes.
blockMode	One of the following constants has to be passed to choose the block mode (ECB, CBC, CFB or OFB): ESL_BM_ECB ESL_BM_CBC. The modes ESL_BM_CFB and ESL_BM_OFB are not yet implemented but can be added if needed. For further details please contact cryptovision.
paddingMode	The padding mode has to be ESL_PM_PKCS5. The modes ESL_PM_OFF and ESL_PM_ONEWITHZEROES are not yet implemented but can be added if needed. For further details please contact cryptovision.
initializationVector	A pointer to an initialization vector for the DES operation.

Output

None.

Return codes

Return code	Remark
ESL_ERC_MODE_INVALID	The chosen mode is not supported.
ESL_ERC_MODE_INCOMPATIBLE	The combination of padding and block mode is not valid.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The general initialization function esl_initWorkSpaceHeader (cf. 3.4) must have been executed successfully before.

3.6.5 DES decryption

This function provides a high level interface to the DES decryption including different modes of operation. It concatenates many succeeding inputs and computes the decryption of this input stream. The output of these functions can be concatenated to give the decrypted stream.

The padding of the output will be checked and not passed out by the finalize function (cf. 3.6.6).

Prototype

```
eslt_ErrorCode esl_decryptDES(
                    eslt_WorkSpaceDES    *workSpace,
              const eslt_Length           inputSize,
              const eslt_Byte            *input,
                    eslt_Length          *outputSize,
                    eslt_Byte            *output       );
```

Description

This function implements the core part of the DES decryption: It reads input length bytes from input, appends them to the inputs from the preceding call to this function and decrypts as many blocks as possible. The remaining bytes are stored in an internal buffer.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initDecryptDES.
inputSize	The number of bytes of the input that are to be decrypted.
input	A pointer to a part of input, that has to be decrypted. In case that preceding inputs had lengths different from multiples of the block length, the input bytes are appended to the preceding bytes and then decrypted. Remaining bytes are stored internally.
*outputSize	The number of bytes that can be used at output for storing the decryption of the input. This size must be longer than inputSize by ESL_SIZEOF_DES_BLOCK to guarantee, that the function can store all the decrypted output necessary. The reason for this precaution are remaining bytes from preceding inputs that can result in an additional output block. If all inputs have lengths that are multiples of ESL_SIZEOF_DES_BLOCK this additional space is not necessary.
output	A pointer to the memory where the decrypted input will be stored. Here esl_decryptDES stores the decrypted message blocks that can be computed from already provided input bytes.

Output

Parameter	Remark
*outputSize	The number of bytes that are really used at output for storing the decryption of the input.
*output	The decrypted input.

Return codes

Return code	Remark
ESL_ERC_OUTPUTLENGTH_TOO_SHORT	The given output length is too short. Another call can be done with less input or more outputSize.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID, ESL_ERC_MEMORY_CONFLICT	

Prerequisites

The initialization function must have been executed successfully before (cf. 3.6.4).

3.6.6 DES finalize decryption

This function carries out the necessary finalization for a previous DES decryption.

Prototype

```
eslt_ErrorCode esl_finalizeDecryptDES(
                   eslt_WorkSpaceDES    *workSpace,
                   eslt_Length          *outputSize,
                   eslt_Byte            *output   );
```

Description

This function implements the finalization routine for an DES encryption. It especially checks the padding at the end of the decrypted data.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initDecryptDES.
*outputSize	The number of bytes that can be used at output for storing the decryption of the input. To guarantee correct execution a value of ESL_SIZEOF_DES_BLOCK is sufficient in any case.
output	A pointer to outputSize bytes of free memory where the output will be stored (without the padding).

Output

Parameter	Remark
*outputSize	The byte length of the output.
*output	The decrypted last blocks.

Return codes

Return code	Remark
ESL_ERC_OUTPUT_SIZE_TOO_SHORT	The given output length is too short.
ESL_ERC_INPUT_INVALID	The complete length of the input is not a multiple of the block length.
ESL_ERC_DES_PADDING_INVALID	The padding of the input is invalid.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID, ESL_ERC_MEMORY_CONFLICT	

Prerequisites

The initialization function and DES decryption operation with non-zero total length must have been executed successfully before (cf. 3.6.4 and 3.6.5).

3.6.7 DES single block encryption initialization

This function carries out the necessary initialization for a subsequent encryption operation in single block mode. It requires less memory than its more general pendant.

Prototype

```
eslt_ErrorCode esl_initEncryptDESBlock (
                  eslt_WorkSpaceDESBlock    *workSpace,
      const       eslt_Byte                 *key       );
```

Description

This function implements the initialization routine for the DES encryption in single block mode.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that has to be at least ESL_MAXSIZEOF_WS_DESBLOCK bytes long.
key	A pointer to a key for DES with the length of ESL_SIZEOF_DES_KEY (= 8) bytes.

Output

None.

Return codes

Return code	Remark
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The general initialization function esl_initWorkSpaceHeader (cf. 3.4) must have been executed successfully before.

3.6.8 DES single block encryption

This function provides the DES encryption of a single block; it performs no padding or modes of operation.

Prototype

```
eslt_ErrorCode esl_encryptDESBlock(
                eslt_WorkSpaceDESBlock      *workSpace,
     const      eslt_Byte                   *inputBlock,
                eslt_Byte                   *outputBlock   );
```

Description

This function implements the DES encryption of a single block (8 Bytes). As it is strictly single block oriented, and does not pad the data automatically, this function has no finalize pendant.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initEncryptDESBlock. Neither input nor output must point to memory inside this area.
inputBlock	This pointer must reference exactly one DES Block (ESL_SIZEOF_DES_BLOCK bytes).
outputBlock	A pointer to a memory area with enough space for the encrypted output block (ESL_SIZEOF_DES_BLOCK bytes). Input and output can use the same memory, but if the pointers differ the memory must not overlap.

Output

Parameter	Remark
*outputBlock	The encrypted input block.

Return codes

Return code	Remark
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID, ESL_ERC_MEMORY_CONFLICT	

Prerequisites

The initialization function must have been executed successfully before (cf. 3.6.7).

3.6.9 DES single block decryption initialization

This function carries out the necessary initialization for a subsequent decryption operation in single block mode. It requires less memory than its more general pendant.

Prototype

```
eslt_ErrorCode esl_initDecryptDESBlock (
```

```
                eslt_WorkSpaceDESBlock    *workSpace,
        const   eslt_Byte                 *key          );
```

Description

This function implements the initialization routine for the DES decryption in single block mode.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that has to be at least ESL_MAXSIZEOF_WS_DES_BLOCK bytes long.
key	A pointer to a key for DES with the length of ESL_SIZEOF_DES_KEY (= 8) bytes.

Output

None.

Return codes

Return code	Remark
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The general initialization function esl_initWorkSpaceHeader (cf. 3.4) must have been executed successfully before.

3.6.10 DES single block decryption

This function provides direct access to the DES decryption of a single block; it therefore performs no padding or modes of operation handling.

Prototype

```
eslt_ErrorCode esl_decryptDESBlock(
                eslt_WorkSpaceDESBlock    *workSpace,
        const   eslt_Byte                 *inputBlock,
                eslt_Byte                 *outputBlock  );
```

Description

This function implements the DES decryption of a single block (8 Bytes). As it is strictly single block oriented, and does not pad the data automatically, this function has no finalize pendant.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initDecryptDESBlock.
inputBlock	This pointer must reference exactly one DES Block (ESL_SIZEOF_DES_BLOCK bytes).

outputBlock	A pointer to a memory area with enough space for the decrypted output block (ESL_SIZEOF_DES_BLOCK bytes). Input and output can use the same memory, but if the pointers differ the memory must not overlap.

Output

Parameter	Remark
*outputBlock	The decrypted input block.

Return codes

Return code	Remark
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID, ESL_ERC_MEMORY_CONFLICT	

Prerequisites

The initialization function must have been executed successfully before (cf. 3.6.9).

3.6.11 DES key generation

This function provides the DES key generation operation.

Prototype

```
eslt_ErrorCode esl_generateKeyDES(
            eslt_Byte              *key      );
```

Description

This function generates a key for DES and stores the key at the memory area referenced by key. It can be called without an initialization function. The random number generator must have been initialized before calling this function (cf. 3.18 and 4.1).

Input

Parameter	Remark
key	A pointer to a memory area for the key to be generated (ESL_SIZEOF_DES_KEY Bytes).

Output

Parameter	Remark
*key	The generated key for DES encryption or decryption.

Return codes

Return code	Remark
ESL_ERC_NO_ERROR	
Additionally all possible return codes from the function esl_getBytesRNG can be returned from this function (cf. 4).	

Prerequisites

The random number module must have been initialized successfully before (cf. 3.18.1). Especially the callback function must be provided by the user (cf. 4).

3.7 TripleDES encryption/decryption

The following functions provide TripleDES encryption and decryption according to FIPS PUB 46-3 [9].

Two variants of TripleDES are provided, single block mode encryption for proprietary use without additional features like padding, and block mode encryption with some commonly used modes of operation like ECB and CBC. For CBC mode a random initialization vector must be provided (cf. [6]).

The TripleDES key length can be chosen to be 128 or 192 bit according to FIPS PUB 46-3 [22].

The sizes of the workSpace for this function is defined by

```
ESL_MAXSIZEOF_WS_TDES
ESL_MAXSIZEOF_WS_TDESBLOCK
```

3.7.1 TripleDES encryption initialization

This function carries out the necessary initialization for a subsequent encryption operation.

Prototype

```
eslt_ErrorCode esl_initEncryptTDES (
                  eslt_WorkSpaceTDES   *workSpace,
      const       eslt_Byte            *key,
      const       eslt_BlockMode        blockMode,
      const       eslt_PaddingMode      paddingMode,
      const       eslt_Byte            *initializationVector );
```

Description

This function implements the initialization routine for the TripleDES encryption.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that has to be at least ESL_MAXSIZESIZEOF_WS_TDES bytes long.
*key	The key for the TripleDES operation with the length of ESL_SIZEOF_2TDES_KEY (= 16) bytes or ESL_SIZEOF_3TDES_KEY (= 24) bytes
blockMode	One of the following constants has to be passed to choose the block mode (ECB, CBC, CFB or OFB): ESL_BM_ECB ESL_BM_CBC.

	The modes `ESL_BM_CFB` and `ESL_BM_OFB` are not yet implemented but can be added if needed. For further details please contact cryptovision.
`paddingMode`	The padding mode has to be `ESL_PM_PKCS5`. The modes `ESL_PM_OFF` and `ESL_PM_ONEWITHZEROES` are not yet implemented but can be added if needed. For further details please contact cryptovision.
`initializationVector`	A pointer to an initialization vector for the TripleDES operation.

Output

None.

Return codes

Return code	Remark
`ESL_ERC_MODE_INVALID`	The chosen mode is not supported.
`ESL_ERC_MODE_INCOMPATIBLE`	The combination of padding and block mode is not valid.
`ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL`	

Prerequisites

The general initialization function esl_initWorkSpaceHeader (cf. 3.4) must have been executed successfully before.

3.7.2 TripleDES encryption

This function provides a high level interface to the TripleDES encryption including different modes of operation. It concatenates succeeding inputs and computes the encryption of this input stream. The output of these functions can be concatenated to give the encrypted stream.

The optional padding is handled by the finalize function (cf. 3.7.3).

Prototype

```
eslt_ErrorCode esl_encryptTDES(
                    eslt_WorkSpaceTDES   *workSpace,
              const eslt_Length           inputSize,
              const eslt_Byte            *input,
                    eslt_Length          *outputSize,
                    eslt_Byte            *output      );
```

Description

This function implements the core part of the TripleDES encryption: It reads input length bytes from input, appends them to the inputs from the preceding call to this function and encrypts as many blocks as possible. The remaining bytes are stored in an internal buffer.

Input

Parameter	Remark
`workSpace`	A pointer to a memory area (context) that was initialized by

	esl_initEncryptTDES.
inputSize	The number of bytes in input that are to be encrypted.
input	A pointer to the input that has to be encrypted. In case that preceding inputs had lengths different from multiples of the block length, the input bytes are appended to the preceding bytes and then encrypted. Remaining bytes are stored internally.
*outputSize	The number of bytes that can be used at output for storing the encrypted input. This size must be longer than inputSize by ESL_SIZEOF_DES_BLOCK to guarantee, that the function can store all the encrypted output necessary. The reason for this precaution are remaining bytes from preceding inputs that can result in an additional output block. If all inputs have lengths that are multiples of ESL_SIZEOF_DES_BLOCK this additional area is not necessary. Input and output can use the same memory, but if the pointers differ the memory must not overlap.
output	A pointer to outputSize bytes of free memory where the output can be stored.

Output

Parameter	Remark
*outputSize	The number of bytes that are really used at output for storing the encrypted input.
*output	The encrypted input.

Return codes

Return code	Remark
ESL_ERC_OUTPUT_SIZE_TOO_SHORT	The given output length is too short. Another call can be done with less input or more outputSize.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID, ESL_ERC_MEMORY_CONFLICT	

Prerequisites

The initialization function must have been executed successfully before (cf. 3.7.1).

3.7.3 TripleDES finalize encryption

This function carries out the necessary finalization for a previous TripleDES encryption.

Prototype

```
eslt_ErrorCode esl_finalizeEncryptTDES(
                    eslt_WorkSpaceTDES   *workSpace,
                    eslt_Length          *outputSize,
                    eslt_Byte            *output        );
```

Description

This function implements the finalization routine for an TripleDES encryption. It especially does the necessary padding, which is needed for some modes of operation.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initEncryptTDES.
*outputSize	The number of bytes that can be used at output for storing the encryption of the input and the padding. For execution of this function 2*ESL_SIZEOF_DES_BLOCK is sufficient in any case.
output	A pointer to outputSize bytes of free memory where the output will be stored.

Output

Parameter	Remark
*outputSize	The byte length of the last output block; always exactly one or two blocks.
*output	The encrypted last one or two blocks.

Return codes

Return code	Remark
ESL_ERC_OUTPUT_SIZE_TOO_SHORT	The given output length is too short. Another call can be done with less input or more outputSize.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID, ESL_ERC_MEMORY_CONFLICT	

Prerequisites

The initialization function and TripleDES encryption operation must have been executed successfully before (cf. 3.7.1 and 3.7.2).

3.7.4 TripleDES decryption initialization

This function carries out the necessary initialization for a subsequent decryption operation.

Prototype

```
eslt_ErrorCode esl_initDecryptTDES (
                eslt_WorkSpaceTDES   *workSpace,
     const      eslt_Byte            *key,
     const      eslt_BlockMode        blockMode,
     const      eslt_PaddingMode      paddingMode,
     const      eslt_Byte            *initializationVector );
```

Description

This function implements the initialization routine for the TripleDES decryption.

Input

Parameter	Remark
`workSpace`	A pointer to a memory area (context) that has to be at least ESL_MAXSIZEOF_WS_TDES bytes long.
`*key`	The key for the TripleDES operation with the length of ESL_SIZEOF_2TDES_KEY (= 16) bytes or ESL_SIZEOF_3TDES_KEY (= 24) bytes
`blockMode`	One of the following constants has to be passed to choose the block mode (ECB, CBC, CFB or OFB): `ESL_BM_ECB` `ESL_BM_CBC`. The modes `ESL_BM_CFB` and `ESL_BM_OFB` are not yet implemented but can be added if needed. For further details please contact cryptovision.
`paddingMode`	The padding mode has to be `ESL_PM_PKCS5`. The modes `ESL_PM_OFF` and `ESL_PM_ONEWITHZEROES` are not yet implemented but can be added if needed. For further details please contact cryptovision.
`initializationVector`	A pointer to an initialization vector for the TripleDES operation.

Output

None.

Return codes

Return code	Remark
`ESL_ERC_MODE_INVALID`	The chosen mode is not supported.
`ESL_ERC_MODE_INCOMPATIBLE`	The combination of padding and block mode is not valid.
`ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID`	

Prerequisites

The general initialization function esl_initWorkSpaceHeader (cf. 3.4) must have been executed successfully before.

3.7.5 TripleDES decryption

This function provides a high level interface to the TripleDES decryption including different modes of operation. It concatenates many succeeding inputs and computes the decryption of this input stream. The output of these functions can be concatenated to give the decrypted stream.

The padding of the output will be checked and not passed out by the finalize function (cf. 3.7.6).

Prototype

```
eslt_ErrorCode esl_decryptTDES(
                    eslt_WorkSpaceTDES   *workSpace,
              const eslt_Length           inputSize,
              const eslt_Byte            *input,
                    eslt_Length          *outputSize,
                    eslt_Byte            *output       );
```

Description

This function implements the core part of the TripleDES decryption: It reads input length bytes from input, appends them to the inputs from the preceding call to this function and decrypts as many blocks as possible. The remaining bytes are stored in an internal buffer.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initDecryptTDES.
inputSize	The number of bytes of the input that are to be decrypted.
input	A pointer to a part of input, that has to be decrypted. In case that preceding inputs had lengths different from multiples of the block length, the input bytes are appended to the preceding bytes and then decrypted. Remaining bytes are stored internally.
*outputSize	The number of bytes that can be used at output for storing the decryption of the input. This size must be longer than inputSize by ESL_SIZEOF_DES_BLOCK to guarantee, that the function can store all the decrypted output necessary. The reason for this precaution are remaining bytes from preceding inputs that can result in an additional output block. If all inputs have lengths that are multiples of ESL_SIZEOF_DES_BLOCK this additional space is not necessary.
output	A pointer to the memory where the decrypted input will be stored. Here esl_decryptTDES stores the decrypted message blocks that can be computed from already provided input bytes.

Output

Parameter	Remark
*outputSize	The number of bytes that are really used at output for storing the decryption of the input.
*output	The decrypted input.

Return codes

Return code	Remark
ESL_ERC_OUTPUTLENGTH_TOO_SHORT	The given output length is too short. Another call can be done with less input or more outputSize.

ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID, ESL_ERC_MEMORY_CONFLICT

Prerequisites

The initialization function must have been executed successfully before (cf. 3.7.4).

3.7.6 TripleDES finalize decryption

This function carries out the necessary finalization for a previous TripleDES decryption.

Prototype

```
eslt_ErrorCode esl_finalizeDecryptTDES(
                  eslt_WorkSpaceTDES   *workSpace,
                  eslt_Length          *outputSize,
                  eslt_Byte            *output   );
```

Description

This function implements the finalization routine for an TripleDES encryption. It especially checks the padding at the end of the decrypted data.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initDecryptTDES.
*outputSize	The number of bytes that can be used at output for storing the decryption of the input. To guarantee correct execution a value of ESL_SIZEOF_DES_BLOCK is sufficient in any case.
output	A pointer to outputSize bytes of free memory where the output will be stored (without the padding).

Output

Parameter	Remark
*outputSize	The byte length of the output.
*output	The decrypted last blocks.

Return codes

Return code	Remark
ESL_ERC_OUTPUT_SIZE_TOO_SHORT	The given output length is too short.
ESL_ERC_INPUT_INVALID	The complete length of the input is not a multiple of the block length.
ESL_ERC_PADDING_INVALID	The padding of the input is invalid.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID, ESL_ERC_MEMORY_CONFLICT	

Prerequisites

The initialization function and TripleDES decryption operation with non-zero total length must have been executed successfully before (cf. 3.7.4 and 3.7.5).

3.7.7 TripleDES single block encryption initialization

This function carries out the necessary initialization for a subsequent encryption operation in single block mode. It requires less memory than its more general pendant.

Prototype

```
eslt_ErrorCode esl_initEncryptTDESBlock (
                eslt_WorkSpaceTDESBlock   *workSpace,
      const     eslt_Byte                 *key       );
```

Description

This function implements the initialization routine for the TripleDES encryption in single block mode.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that has to be at least ESL_MAXSIZEOF_WS_TDESBLOCK bytes long.
key	The key for the TripleDES operation with the length of ESL_SIZEOF_2TDES_KEY (= 16) bytes or ESL_SIZEOF_3TDES_KEY (= 24) bytes.

Output

None.

Return codes

Return code	Remark
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The general initialization function esl_initWorkSpaceHeader (cf. 3.4) must have been executed successfully before.

3.7.8 TripleDES single block encryption

This function provides the TripleDES encryption of a single block; it performs no padding or modes of operation.

Prototype

```
eslt_ErrorCode esl_encryptTDESBlock(
                eslt_WorkSpaceTDESBlock     *workSpace,
      const     eslt_Byte                   *inputBlock,
                eslt_Byte                   *outputBlock   );
```

Description

This function implements the TripleDES encryption of a single block (8 Bytes). As it is strictly single block oriented, and does not pad the data automatically, this function has no finalize pendant.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initEncryptTDESBlock. Neither input nor output must point to memory inside this area.
inputBlock	This pointer must reference exactly one TripleDES Block (ESL_SIZEOF_DES_BLOCK bytes).
outputBlock	A pointer to a memory area with enough space for the encrypted output block (ESL_SIZEOF_DES_BLOCK bytes). Input and output can use the same memory, but if the pointers differ the memory must not overlap.

Output

Parameter	Remark
*outputBlock	The encrypted input block.

Return codes

Return code	Remark
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID, ESL_ERC_MEMORY_CONFLICT	

Prerequisites

The initialization function must have been executed successfully before (cf. 3.7.7).

3.7.9 TripleDES single block decryption initialization

This function carries out the necessary initialization for a subsequent decryption operation in single block mode. It requires less memory than its more general pendant.

Prototype

```
eslt_ErrorCode esl_initDecryptTDESBlock (
                  eslt_WorkSpaceTDESBlock   *workSpace,
      const       eslt_Byte                 *key          );
```

Description

This function implements the initialization routine for the TripleDES decryption in single block mode.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that has to be at least ESL_MAXSIZEOF_WS_TDES_BLOCK bytes long.
key	The key for the TripleDES operation with the length of ESL_SIZEOF_2TDES_KEY (= 16) bytes or ESL_SIZEOF_3TDES_KEY (= 24) bytes.

Output

None.

Return codes

Return code	Remark
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The general initialization function esl_initWorkSpaceHeader (cf. 3.4) must have been executed successfully before.

3.7.10 TripleDES single block decryption

This function provides direct access to the TripleDES decryption of a single block; it therefore performs no padding or modes of operation handling.

Prototype

```
eslt_ErrorCode esl_decryptTDESBlock(
                eslt_WorkSpaceTDESBlock     *workSpace,
        const   eslt_Byte                   *inputBlock,
                eslt_Byte                   *outputBlock  );
```

Description

This function implements the TripleDES decryption of a single block (8 Bytes). As it is strictly single block oriented, and does not pad the data automatically, this function has no finalize pendant.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initDecryptTDESBlock.
inputBlock	This pointer must reference exactly one TripleDES Block (ESL_SIZEOF_DES_BLOCK bytes).
outputBlock	A pointer to a memory area with enough space for the decrypted output block (ESL_SIZEOF_DES_BLOCK bytes). Input and output can use the same memory, but if the pointers differ the memory must not overlap.

Output

Parameter	Remark
*outputBlock	The decrypted input block.

Return codes

Return code	Remark
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID, ESL_ERC_MEMORY_CONFLICT	

Prerequisites

The initialization function must have been executed successfully before (cf. 3.7.9).

3.7.11 TripleDES key generation

This function provides the TripleDES key generation operations for 192 bit keys.

Note: The key generation function for the 128-bit-variant is called `esl_generateKey2TDES`, respectively.

Prototype

```
eslt_ErrorCode esl_generateKey3TDES(
            eslt_Byte              *key      );
```

Description

This function generates a key for TripleDES and stores the key at the memory area referenced by key. The parity bits are calculated as even parity bits by default. It can be called without an initialization function. The random number generator must have been initialized before calling this function (cf. 3.18 and 4.1).

Input

Parameter	Remark
key	A pointer to a memory area for the key to be generated (ESL_SIZEOF_3TDES_KEY Bytes).

Output

Parameter	Remark
*key	The generated key for TripleDES encryption or decryption.

Return codes

Return code	Remark
ESL_ERC_NO_ERROR	
Additionally all possible return codes from the function esl_getBytesRNG can be returned from this function (cf. 4).	

Prerequisites

The random number module must have been initialized successfully before (cf. 3.18.1). Especially the callback function must be provided by the user (cf. 4).

3.8 RC2 encryption/decryption

The following functions provide RC2 encryption and decryption according to RFC 2268 [9].

Two variants of RC2 are provided, single block mode encryption for proprietary use without additional features like padding, and block mode encryption with some commonly used modes of

operation like ECB and CBC. For CBC mode a random initialization vector must be provided (cf. [6]).

The sizes of the workSpace for this function is defined by

```
ESL_MAXSIZEOF_WS_RC2
ESL_MAXSIZEOF_WS_RC2BLOCK
```

3.8.1 RC2 encryption initialization

This function carries out the necessary initialization for a subsequent encryption operation.

Prototype

```
eslt_ErrorCode esl_initEncryptRC2 (
                eslt_WorkSpaceRC2    *workSpace,
     const      eslt_Byte            *key,
     const      eslt_Byte             key_len,
     const      eslt_Byte             eff_key_len,
     const      eslt_BlockMode        blockMode,
     const      eslt_PaddingMode      paddingMode,
     const      eslt_Byte            *initializationVector );
```

Description

This function implements the initialization routine for the RC2 encryption.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that has to be at least ESL_MAXSIZESIZEOF_WS_RC2 bytes long.
*key	The key for the RC2 operation.
key_len	The key length in byte for the RC2 operation.
eff_key_len	The effective key length in byte for the RC2 operation. **Note**: It's recommended to use the same length as key_len.
blockMode	One of the following constants has to be passed to choose the block mode (ECB, CBC, CFB or OFB): ESL_BM_ECB ESL_BM_CBC. The modes ESL_BM_CFB and ESL_BM_OFB are not yet implemented but can be added if needed. For further details please contact cryptovision.
paddingMode	The padding mode has to be ESL_PM_PKCS5. The modes ESL_PM_OFF and ESL_PM_ONEWITHZEROES are not yet implemented but can be added if needed. For further details please contact cryptovision.

initializationVector	A pointer to an initialization vector for the RC2 operation.

Output

None.

Return codes

Return code	Remark
ESL_ERC_MODE_INVALID	The chosen mode is not supported.
ESL_ERC_MODE_INCOMPATIBLE	The combination of padding and block mode is not valid.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL	

Prerequisites

The general initialization function esl_initWorkSpaceHeader (cf. 3.4) must have been executed successfully before.

3.8.2 RC2 encryption

This function provides a high level interface to the RC2 encryption including different modes of operation. It concatenates succeeding inputs and computes the encryption of this input stream. The output of these functions can be concatenated to give the encrypted stream.

The optional padding is handled by the finalize function (cf. 3.8.3).

Prototype

```
eslt_ErrorCode esl_encryptRC2(
                    eslt_WorkSpaceRC2    *workSpace,
              const eslt_Length           inputSize,
              const eslt_Byte            *input,
                    eslt_Length          *outputSize,
                    eslt_Byte            *output      );
```

Description

This function implements the core part of the RC2 encryption: It reads input length bytes from input, appends them to the inputs from the preceding call to this function and encrypts as many blocks as possible. The remaining bytes are stored in an internal buffer.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initEncryptRC2.
inputSize	The number of bytes in input that are to be encrypted.
input	A pointer to the input that has to be encrypted. In case that preceding inputs had lengths different from multiples of the block length, the input bytes are appended to the preceding bytes and then encrypted. Remaining bytes are stored internally.
*outputSize	The number of bytes that can be used at output for storing the encrypted input. This size must be longer than inputSize by ESL_SIZEOF_RC2_BLOCK to guarantee, that the function can

	store all the encrypted output necessary. The reason for this precaution are remaining bytes from preceding inputs that can result in an additional output block. If all inputs have lengths that are multiples of `ESL_SIZEOF_RC2_BLOCK` this additional area is not necessary. Input and output can use the same memory, but if the pointers differ the memory must not overlap.
`output`	A pointer to outputSize bytes of free memory where the output can be stored.

Output

Parameter	Remark
`*outputSize`	The number of bytes that are really used at output for storing the encrypted input.
`*output`	The encrypted input.

Return codes

Return code	Remark
`ESL_ERC_OUTPUT_SIZE_TOO_SHORT`	The given output length is too short. Another call can be done with less input or more outputSize.
`ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID, ESL_ERC_MEMORY_CONFLICT`	

Prerequisites

The initialization function must have been executed successfully before (cf. 3.8.1).

3.8.3 RC2 finalize encryption

This function carries out the necessary finalization for a previous RC2 encryption.

Prototype

```
eslt_ErrorCode esl_finalizeEncryptRC2(
                    eslt_WorkSpaceRC2    *workSpace,
                    eslt_Length          *outputSize,
                    eslt_Byte            *output        );
```

Description

This function implements the finalization routine for an RC2 encryption. It especially does the necessary padding, which is needed for some modes of operation.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initEncryptRC2.
*outputSize	The number of bytes that can be used at output for storing the encryption of the input and the padding. For execution of this function 2*ESL_SIZEOF_RC2_BLOCK is sufficient in any case.
output	A pointer to outputSize bytes of free memory where the output will be stored.

Output

Parameter	Remark
*outputSize	The byte length of the last output block; always exactly one or two blocks.
*output	The encrypted last one or two blocks.

Return codes

Return code	Remark
ESL_ERC_OUTPUT_SIZE_TOO_SHORT	The given output length is too short. Another call can be done with less input or more outputSize.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID, ESL_ERC_MEMORY_CONFLICT	

Prerequisites

The initialization function and RC2 encryption operation must have been executed successfully before (cf. 3.8.1 and 3.8.2).

3.8.4 RC2 decryption initialization

This function carries out the necessary initialization for a subsequent decryption operation.

Prototype

```
eslt_ErrorCode esl_initDecryptRC2 (
                eslt_WorkSpaceRC2    *workSpace,
     const      eslt_Byte            *key,
     const      eslt_Byte             key_len,
     const      eslt_Byte             eff_key_len,
     const      eslt_BlockMode        blockMode,
     const      eslt_PaddingMode      paddingMode,
     const      eslt_Byte            *initializationVector );
```

Description

This function implements the initialization routine for the RC2 decryption.

Input

Parameter	Remark

workSpace	A pointer to a memory area (context) that has to be at least ESL_MAXSIZEOF_WS_RC2 bytes long.
*key	The key for the RC2 operation.
key_len	The key length in byte for the RC2 operation.
eff_key_len	The effective key length in byte for the RC2 operation. **Note**: It's recommended to use the same length as key_len.
blockMode	One of the following constants has to be passed to choose the block mode (ECB, CBC, CFB or OFB): ESL_BM_ECB ESL_BM_CBC. The modes ESL_BM_CFB and ESL_BM_OFB are not yet implemented but can be added if needed. For further details please contact cryptovision.
paddingMode	The padding mode has to be ESL_PM_PKCS5. The modes ESL_PM_OFF and ESL_PM_ONEWITHZEROES are not yet implemented but can be added if needed. For further details please contact cryptovision.
initializationVector	A pointer to an initialization vector for the RC2 operation.

Output

None.

Return codes

Return code	Remark
ESL_ERC_MODE_INVALID	The chosen mode is not supported.
ESL_ERC_MODE_INCOMPATIBLE	The combination of padding and block mode is not valid.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The general initialization function esl_initWorkSpaceHeader (cf. 3.4) must have been executed successfully before.

3.8.5 RC2 decryption

This function provides a high level interface to the RC2 decryption including different modes of operation. It concatenates many succeeding inputs and computes the decryption of this input stream. The output of these functions can be concatenated to give the decrypted stream.

The padding of the output will be checked and not passed out by the finalize function (cf. 3.8.6).

Prototype

```
eslt_ErrorCode esl_decryptRC2(
```

```
      eslt_WorkSpaceRC2    *workSpace,
const eslt_Length           inputSize,
const eslt_Byte            *input,
      eslt_Length          *outputSize,
      eslt_Byte            *output       );
```

Description

This function implements the core part of the RC2 decryption: It reads input length bytes from input, appends them to the inputs from the preceding call to this function and decrypts as many blocks as possible. The remaining bytes are stored in an internal buffer.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initDecryptRC2.
inputSize	The number of bytes of the input that are to be decrypted.
input	A pointer to a part of input, that has to be decrypted. In case that preceding inputs had lengths different from multiples of the block length, the input bytes are appended to the preceding bytes and then decrypted. Remaining bytes are stored internally.
*outputSize	The number of bytes that can be used at output for storing the decryption of the input. This size must be longer than inputSize by ESL_SIZEOF_RC2_BLOCK to guarantee, that the function can store all the decrypted output necessary. The reason for this precaution are remaining bytes from preceding inputs that can result in an additional output block. If all inputs have lengths that are multiples of ESL_SIZEOF_RC2_BLOCK this additional space is not necessary.
output	A pointer to the memory where the decrypted input will be stored. Here esl_decryptRC2 stores the decrypted message blocks that can be computed from already provided input bytes.

Output

Parameter	Remark
*outputSize	The number of bytes that are really used at output for storing the decryption of the input.
*output	The decrypted input.

Return codes

Return code	Remark
ESL_ERC_OUTPUTLENGTH_TOO_SHORT	The given output length is too short. Another call can be done with less input or more outputSize.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID, ESL_ERC_MEMORY_CONFLICT	

Prerequisites

The initialization function must have been executed successfully before (cf. 3.8.4).

3.8.6 RC2 finalize decryption

This function carries out the necessary finalization for a previous RC2 decryption.

Prototype

```
eslt_ErrorCode esl_finalizeDecryptRC2(
                  eslt_WorkSpaceRC2    *workSpace,
                  eslt_Length          *outputSize,
                  eslt_Byte            *output   );
```

Description

This function implements the finalization routine for an RC2 encryption. It especially checks the padding at the end of the decrypted data.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initDecryptRC2.
*outputSize	The number of bytes that can be used at output for storing the decryption of the input. To guarantee correct execution a value of ESL_SIZEOF_RC2_BLOCK is sufficient in any case.
output	A pointer to outputSize bytes of free memory where the output will be stored (without the padding).

Output

Parameter	Remark
*outputSize	The byte length of the output.
*output	The decrypted last blocks.

Return codes

Return code	Remark
ESL_ERC_OUTPUT_SIZE_TOO_SHORT	The given output length is too short.
ESL_ERC_INPUT_INVALID	The complete length of the input is not a multiple of the block length.
ESL_ERC_PADDING_INVALID	The padding of the input is invalid.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID, ESL_ERC_MEMORY_CONFLICT	

Prerequisites

The initialization function and RC2 decryption operation with non-zero total length must have been executed successfully before (cf. 3.8.4 and 3.8.5).

3.8.7 RC2 single block encryption initialization

This function carries out the necessary initialization for a subsequent encryption operation in single block mode. It requires less memory than its more general pendant.

Prototype

```
eslt_ErrorCode esl_initEncryptRC2Block (
                eslt_WorkSpaceRC2Block    *workSpace,
      const     eslt_Byte                 *key,
    const       eslt_Byte                  key_len,
    const       eslt_Byte                  eff_key_len   );
```

Description

This function implements the initialization routine for the RC2 encryption in single block mode.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that has to be at least ESL_MAXSIZEOF_WS_RC2BLOCK bytes long.
key	A pointer to a key for RC2.
key_len	The key length in byte for the RC2 operation.
eff_key_len	The effective key length in byte for the RC2 operation. **Note**: It's recommended to use the same length as key_len.

Output

None.

Return codes

Return code	Remark
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The general initialization function esl_initWorkSpaceHeader (cf. 3.4) must have been executed successfully before.

3.8.8 RC2 single block encryption

This function provides the RC2 encryption of a single block; it performs no padding or modes of operation.

Prototype

```
eslt_ErrorCode esl_encryptRC2Block(
                eslt_WorkSpaceRC2Block      *workSpace,
     const      eslt_Byte                   *inputBlock,
                eslt_Byte                   *outputBlock   );
```

Description

This function implements the RC2 encryption of a single block (8 Bytes). As it is strictly single block oriented, and does not pad the data automatically, this function has no finalize pendant.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initEncryptRC2Block. Neither input nor output must point to memory inside this area.
inputBlock	This pointer must reference exactly one RC2 Block (ESL_SIZEOF_RC2_BLOCK bytes).
outputBlock	A pointer to a memory area with enough space for the encrypted output block (ESL_SIZEOF_RC2_BLOCK bytes). Input and output can use the same memory, but if the pointers differ the memory must not overlap.

Output

Parameter	Remark
*outputBlock	The encrypted input block.

Return codes

Return code	Remark
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID, ESL_ERC_MEMORY_CONFLICT	

Prerequisites

The initialization function must have been executed successfully before (cf. 3.8.7).

3.8.9 RC2 single block decryption initialization

This function carries out the necessary initialization for a subsequent decryption operation in single block mode. It requires less memory than its more general pendant.

Prototype

```
eslt_ErrorCode esl_initDecryptRC2Block (
                eslt_WorkSpaceRC2Block    *workSpace,
      const     eslt_Byte                 *key
      const     eslt_Byte                  key_len,
      const     eslt_Byte                  eff_key_len   );
```

Description

This function implements the initialization routine for the RC2 decryption in single block mode.

Input

Parameter	Remark

workSpace	A pointer to a memory area (context) that has to be at least ESL_MAXSIZEOF_WS_RC2_BLOCK bytes long.
key	A pointer to a key for RC2.
key_len	The key length in byte for the RC2 operation.
eff_key_len	The effective key length in byte for the RC2 operation. **Note**: It's recommended to use the same length as key_len.

Output

None.

Return codes

Return code	Remark
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The general initialization function esl_initWorkSpaceHeader (cf. 3.4) must have been executed successfully before.

3.8.10 RC2 single block decryption

This function provides direct access to the RC2 decryption of a single block; it therefore performs no padding or modes of operation handling.

Prototype

```
eslt_ErrorCode esl_decryptRC2Block(
                eslt_WorkSpaceRC2Block      *workSpace,
       const    eslt_Byte                   *inputBlock,
                eslt_Byte                   *outputBlock  );
```

Description

This function implements the RC2 decryption of a single block (8 Bytes). As it is strictly single block oriented, and does not pad the data automatically, this function has no finalize pendant.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initDecryptRC2Block.
inputBlock	This pointer must reference exactly one RC2 Block (ESL_SIZEOF_RC2_BLOCK bytes).
outputBlock	A pointer to a memory area with enough space for the decrypted output block (ESL_SIZEOF_RC2_BLOCK bytes). Input and output can use the same memory, but if the pointers differ the memory must not overlap.

Output

Parameter	Remark
`*outputBlock`	The decrypted input block.

Return codes

Return code	Remark
`ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID, ESL_ERC_MEMORY_CONFLICT`	

Prerequisites

The initialization function must have been executed successfully before (cf. 3.8.9).

3.8.11 RC2 key generation

This function provides the RC2 key generation operation.

Prototype

```
eslt_ErrorCode esl_generateKeyRC2(
            eslt_Byte             *key      );
```

Description

This function generates a key for RC2 and stores the key at the memory area referenced by key. It can be called without an initialization function. The random number generator must have been initialized before calling this function (cf. 3.18 and 4.1).

Input

Parameter	Remark
`key`	A pointer to a memory area for the key to be generated (ESL_SIZEOF_RC2_KEY Bytes).

Output

Parameter	Remark
`*key`	The generated key for RC2 encryption or decryption.

Return codes

Return code	Remark
`ESL_ERC_NO_ERROR`	
Additionally all possible return codes from the function esl_getBytesRNG can be returned from this function (cf. 4).	

Prerequisites

The random number module must have been initialized successfully before (cf. 3.18.1). Especially the callback function must be provided by the user (cf. 4).

3.9 Hashing

The library provides the three different hash-algorithms SHA-1, SHA-256, RipeMD-160 and MD5 according to [7], [12] and [21]. While the following description only explains the SHA-1, the other variants are used exactly the same way. The respective function prototypes are named `~SHA256`, `~RIPEMD160` and `~MD5`, respectively.

The following functions provide the Secure Hash Standard (SHA-1) operations. The size of the workSpaces needed for hashing is given by these constants:

```
ESL_MAXSIZEOF_WS_SHA1
ESL_MAXSIZEOF_WS_SHA256
ESL_MAXSIZEOF_WS_RIPEMD160
ESL_MAXSIZEOF_WS_MD5
```

3.9.1 Hash initialization

This function carries out the necessary initialization for a subsequent hash operation.

Prototype

```
eslt_ErrorCode esl_initSHA1 (
          eslt_WorkSpaceSHA1     *workSpace    );
```

Description

This function implements the initialization routine for the SHA-1 hash operation.

Input

Parameter	Remark
`workSpace`	A pointer to a memory area (context) that has to be at least ESL_MAXSIZEOF_WS_SHA1 bytes long.

Output

None.

Return codes

Return code	Remark
`ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID`	

Prerequisites

The general initialization function esl_initWorkSpaceHeader (cf. 3.4) must have been executed successfully before.

3.9.2 Hash update

This function provides a high level interface to the SHA-1 hash algorithm. It concatenates succeeding inputs and computes the hash value over the complete input string (except the padding that will be done by the corresponding finalize function, cf. 3.9.3).

Prototype

```
eslt_ErrorCode esl_updateSHA1 (
           eslt_WorkSpaceSHA1   *workSpace,
     const eslt_Length           inputSize,
     const eslt_Byte            *input          );
```

Description

This function updates the current chaining values of the SHA-1 computation.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initSHA1.
input	A pointer to parts of the message (or the complete message) that has to be hashed. The user can pass more than one input area by consecutive calls of this function: This function computes a hash over the concatenation of all inputs.
inputSize	The number of bytes that shall be used for hashing.

Output

None.

Return codes

Return code	Remark
ESL_ERC_HASH_TOTAL_LENGTH_OVERFLOW	The total size of a hash input exceeds 2^{56} bytes.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The initialization function must have been executed successfully before (cf. 3.9.1).

3.9.3 Hash finalize

This function finishes the hashing of a message by padding the input as defined for SHA-1 (cf. [7]).

Prototype

```
eslt_ErrorCode esl_finalizeSHA1 (
           eslt_WorkSpaceSHA1    *workSpace,
           eslt_Byte             *messageDigest    );
```

Description

This function pads the message as defined for SHA-1.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initSHA1.

Output

Parameter	Remark
`messageDigest`	This pointer references the memory, where the message digest will be stored. The output size for SHA-1 is ESL_SIZEOF_SHA1_DIGEST (= 20) bytes.

Return codes

Return code	Remark
`ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID`	

Prerequisites

The esl_updateSHA1 function must have been executed at least once successfully before (cf. 3.9.2).

3.10 Message authentication code (Hash-MAC)

The following functions provide a message authentication code (MAC) according to [10] that internally uses a hash function. There are two versions available in this library, based on SHA-1 and RIPEMD160 [20]. While the following description only explains the HashMACSHA1, the RIPEMD160 variant is used exactly the same way. The respective function prototypes are named `~RIPEMD160`.

Depending on the operation to be performed, the following two different calling sequences are to be used which differ in the last step:

MAC generation:

- MAC initialization (`esl_initHashMACSHA1`)
- MAC update (`esl_updateHashMACSHA1`)
- MAC finalization (`esl_finalizeHashMACSHA1`)

MAC verification:

- MAC initialization (`esl_initHashMACSHA1`)
- MAC update (`esl_updateHashMACSHA1`)
- MAC verification (`esl_verifyHashMACSHA1`)

For Hash-MAC key generation, the function `esl_generateKeyHashMACSHA1` (3.10.5) can be used.

The size of the required workSpaces is given by these constants:

```
ESL_MAXSIZEOF_WS_HMACSHA1
ESL_MAXSIZEOF_WS_HMACRIPEMD160
```

3.10.1 Hash-MAC initialization

This function carries out the necessary initialization for a subsequent HMACSHA-1 operation.

Prototype

```
eslt_ErrorCode esl_initHashMACSHA1 (
            eslt_WorkSpaceHMACSHA1     *workSpace,
      const eslt_Length                 keyLength,
            eslt_Byte                  *key         );
```

Description

This function implements the initialization routine for the Hash-MAC operation.

Input

Parameter	Remark
`workSpace`	A pointer to a memory area (context) that has to be at least ESL_MAXSIZEOF_WS_HASHMACSHA1 bytes long.
`keyLength`	The key length of the Hash-Mac.
`key`	A pointer to a key that will be used for the MAC.

Output

None.

Return codes

Return code	Remark
ESL_ERC_HMAC_KEY_LENGTH_OUT_OF_RANGE	The key length is zero.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The general initialization function esl_initWorkSpaceHeader (cf. 3.4) must have been executed successfully before.

3.10.2 Hash-MAC update

This function provides a high level interface to the HashMACSHA-1 algorithm. It concatenates many succeeding inputs and computes the MAC over the complete input string.

Prototype

```
eslt_ErrorCode esl_updateHashMACSHA1 (
                eslt_WorkSpaceHMACSHA1  *workSpace,
        const   eslt_Length              inputLength,
        const   eslt_Byte               *input          );
```

Description

This function updates the current chaining values of the Hash-MAC computation. The message is passed block wise. These blocks can be of arbitrary length.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initHashMACSHA1.
inputLength	The length of the input in bytes.
input	A pointer to the input, that has to be authenticated by a Hash-MAC. This pointer must not point into workSpace. In case that preceding inputs had lengths different from multiples of the block length, the input bytes are appended to the preceding bytes and then processed.

Output

None.

Return codes

Return code	Remark
ESL_ERC_HASH_TOTAL_LENGTH_OVERFLOW	If the internal hash returns an overflow, the error is passed to the user.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The initialization function must have been executed successfully before (cf. 3.10.1).

3.10.3 Hash-MAC finalize

This function finishes the HMACSHA-1 generation of a message and returns the computed MAC (cf. [16]).

Prototype

```
eslt_ErrorCode esl_finalizeHashMACSHA1 (
                eslt_WorkSpaceHMACSHA1  *workSpace,
                eslt_Byte               *messageHashMAC   );
```

Description

This function computes the HashMAC and stores it at messageHashMAC.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initHashMAC.
messageHashMAC	A pointer to the memory, where the HashMAC shall be stored.

Output

Parameter	Remark
*messageHashMAC	The calculated HashMAC.

Return codes

Return code	Remark
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The esl_updateHashMACSHA1 function must have been executed successfully before with total length greater than zero (cf. 3.10.2).

3.10.4 Hash-MAC verification

This function finishes the Hash-MAC calculation of a message or an input and compares the computed MAC to a given one (cf. [10]). Successful verification is indicated by ESL_ERC_NO_ERROR.

Prototype

```
eslt_ErrorCode esl_verifyHashMACSHA1 (
              eslt_WorkSpaceHMACSHA1  *workSpace,
       const  eslt_Byte               *messageHashMAC   );
```

Description

This function verifies the internally computed HashMAC and compares it with the expected value.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initHashMAC.
messageHashMAC	A pointer to the given Hash-MAC that is to be verified.

Output

None.

Return codes

Return code	Remark
ESL_ERC_HMAC_INCORRECT_MAC	The Hash-MAC is different from the value referenced by messageHashMAC.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID, ESL_ERC_MEMORY_CONFLICT	

Prerequisites

The esl_updateHashMACSHA1 function must have been executed successfully before with total length greater than zero (cf. 3.10.3).

3.10.5 Hash-MAC key generation

This function provides the key generation for a HashMAC.

Prototype

```
eslt_ErrorCode esl_generateKeyHashMACSHA1 (
      const     eslt_Length        keyLength,
                eslt_Byte         *key     );
```

Description

This function generates a key with the given key length. The key length for the MAC can be chosen arbitrary. The recommended length is the output length of the underlying hash function (20 bytes for SHA1).

Input

Parameter	Remark
keyLength	The length of the key to be generated.
key	A pointer to a memory area for the key to be generated (ESL_SIZEOF_SHA1_DIGEST Bytes).

Output

Parameter	Remark
*key	The generated key with the length given in keyLength.

Return codes

Return code	Remark
ESL_ERC_NO_ERROR	
Additionally all possible return codes from the function esl_getBytesRNG can be returned from this function (cf. 4).	

Prerequisites

The random number module must have been initialized successfully before (cf. 3.18.1). Especially the callback function must be provided by the user (cf. 4).

3.11 Key derivation functions

The following functions provide key derivation functions according to PKCS #5 V2.0 [17] and ANSI X9.63 [2]. While the following description only explains the KDF2HMACSHA1, the KDFX963SHA1 variant is used exactly the same way. The respective function prototypes are named `~KDFX963SHA1`.

The size of the required workSpaces is given by these constants:

```
ESL_MAXSIZEOF_WS_KDF2HMACSHA1
ESL_MAXSIZEOF_WS_KDFX963SHA1
```

3.11.1 KDF initialization

This function carries out the necessary initialization for a subsequent key derivation using KDF2 with HMAC SHA-1 as the underlying pseudo random number function.

Prototype

```
eslt_ErrorCode esl_initKDF2HMACSHA1 (
            eslt_WorkSpaceKDF2HMACSHA1   *workSpace,
      const eslt_Length                   iterationCount    );
```

Description

This function implements the initialization routine for the KDF2 key derivation.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that has to be at least ESL_MAXSIZEOF_WS_KDF2HMACSHA1 bytes long.
iterationCount	The number of rounds that has to be performed to get the derived key. The iteration count linearly influences the execution time of the key derivation (cf. [17]). The constant ESL_KDF2_DEFAULT_ITERATION_COUNT can be used here.

Output

None.

Return codes

Return code	Remark
ESL_ERC_KDF_ITERATION_COUNT_OUT_OF_RANGE	The given iteration count 0 is not supported.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The general initialization function esl_initWorkSpaceHeader (cf. 3.4) must have been executed successfully before.

3.11.2 KDF derive key

This function derives a key according to the PKCS #5 standard (cf. [17]).

Prototype

```
eslt_ErrorCode esl_deriveKeyKDF2HMACSHA1 (
          eslt_WorkSpaceKDF2HMACSHA1      *workSpace,
          const eslt_Length                secretLength,
          const eslt_Byte                 *secret,
          const eslt_Length                infoLength,
          const eslt_Byte                 *info
          const eslt_Length                keyLength,
                eslt_Byte                 *outputKey       );
```

Description

This function implements the core part of the KDF2 key derivation: Starting from a common secret and some additional optional information this function derives an appropriate key with the key length given during initialization. The output is stored at the memory area referenced by outputKey that was passed to the initialization function.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initKDF2HMACSHA1.
secretLength	The number of bytes that represent the common secret. This length must not be zero.
secret	A pointer to the secret that has to be used for the key derivation.
infoLength	The number of bytes that represent some extra information. This length can be zero, as this parameter is optional.
info	A pointer to some optional extra information that is taken into account during derivation of a key (cf. [17]).
keyLength	The byte length of the key that has to be derived, all lengths are valid, except zero.
outputKey	A pointer to a memory area, where the output key is placed with the length given by keyLength.

Output

Parameter	Remark
`*outputKey`	The derived key.

Return codes

Return code	Remark
`ESL_ERC_KDF_ENTROPY_TOO_SMALL`	Given input secret length is shorter than 20 Bytes.
`ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID`	

Prerequisites

The initialization function must have been executed successfully before (cf. 3.11.1).

3.12 Elliptic Curve cryptography – general information

This paragraph describes a set of helper functions for asymmetric cryptographic mechanisms based on elliptic curves (*Elliptic Curve Cryptography, ECC*) over prime fields (*GF(p)*). They return the length of in- and outputs to ECC functions depending on the chosen ECC domain parameters.

The following paragraphs (3.13 - 3.15) then describe the set of elliptic curve functions, comprising:

- Digital signature generation and verification using the Elliptic Curve Digital Signature Algorithm (ECDSA) according to ANSI X9.62 (cf. [1]).
- ECDSA key generation according to ANSI X9.62 (cf. [1]).
- Key exchange using the Elliptic Curve Diffie-Hellman key exchange protocol according to ANSI X9.63 (cf. [2]).

These functions support arbitrary domain parameters with bit lengths between 128 and 521.

3.12.1 Domain parameters

Sample standard domain parameter sets (ECParameters, cf. [1]) for 128 to 521 bit are included in our Performance Estimation Tool and testBench that come with the library. These can be used in other projects and are sufficient for most purposes.

If special curves are needed there are tools available from cryptovision that compute these parameters and produce .c-files including the necessary data-fields, including:

- conversion of unformatted basic curve parameters into a defined structure (`domain`) that is expected by any of the library's ECC functions,
- generation of additional information about these domain parameters (`domainExt`). This information depends on the used arithmetic and is needed by any of the library's ECC functions to speed up calculations and
- generation of data containing further pre-computed values, depending on the used arithmetic that is used to speed up algorithms for ECC key generation and signature operations (`speedUpExt`).

The structure of the domain data is DER coded (cf. [13]) according to the ASN.1 module (cf. [14]) as defined in [1].

3.12.2 Keys

Private keys are interpreted as an array of bytes with length according to esl_getLengthOfEcPprivateKey(), which is the number of bytes of the order of the base point.

The esl_generateKeyEcP() function delivers an appropriate private key as output.

The coordinates of public keys are stored separately. The coordinates are interpreted as arrays of bytes with length according to esl_getLengthOfEcPpublicKey_comp() which is the number of bytes of the prime.

The function esl_generateKeyEcP() delivers an appropriate public key as output.

3.12.3 EC-DSA parameter length functions

These functions help users to extract the length of parameters out of the ECC domain parameters used by ECC functions.

Prototype

```
eslt_Length
esl_getMaxLengthOfEcPmessage      ( eslt_EccDomain ROM *domain );

eslt_Length
esl_getLengthOfEcPpublicKey_comp  ( eslt_EccDomain ROM *domain );

eslt_Length
esl_getLengthOfEcPprivateKey      ( eslt_EccDomain ROM *domain );

eslt_Length
esl_getLengthOfEcPsignature_comp  ( eslt_EccDomain ROM *domain );

eslt_Length
esl_getLengthOfEcPsecret_comp     ( eslt_EccDomain ROM *domain );
```

Description

These functions compute the length needed to handle in- and output values of the ECC primitives. They perform simple checks of the input structure to prevent wrong input by mistake. They also check that the output is inside the range supported by the library.

Input

Parameter	Remark
`domain`	This pointer references the domain parameters of an elliptic curve (cf. 3.12.1).

Output

None.

Return Value

The length of the current parameter, i.e.

- the maximum possible byte length of the message,
- the byte length of a public key component,
- the byte length of the private key,
- the byte length of a signature component and
- the byte length of a component of the common secret value

or 0 (zero) in case of an error.

Prerequisites

None.

3.13 EC-DSA digital signature generation and verification

The following functions implement the ECDSA signature computation and the ECDSA verification primitive according to ANSI X9.62 [1].

3.13.1 Signature generation initialization

This function initializes the ECDSA signature primitive.

Prototype

```
eslt_ErrorCode esl_initSignDSAEcP_prim(
                        eslt_WorkSpaceEcP          *workSpace,
                const   eslt_EccDomain         ROM *domain,
                const   eslt_EccDomainExt      ROM *domainExt,
                const   eslt_EccSpeedUpExt     ROM *speedUpExt );
```

Description

This function does the necessary initializations for the ECDSA signature generation.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that has to be at least ESL_MAXSIZEOF_WS_ECP bytes long. In this memory all computations are done during the SignDSAEcP operation.
domain	This pointer references the domain parameters of an elliptic curve (cf. 3.12.1).
domainExt	This pointer references the additional domain information.
speedUpExt	This pointer references the additional data needed to speed up the computations.

Output

None.

Return codes

Return code	Remark
ESL_ERC_ECC_DOMAIN_INVALID	The domain parameters contain invalid data.
ESL_ERC_ECC_DOMAINEXT_INVALID	The domainExt parameters contain invalid data.
ESL_ERC_ECC_SPEEDUPEXT_INVALID	The speedUpExt parameters contain invalid data.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The general initialization function esl_initWorkSpaceHeader (cf. 3.4) must have been executed successfully before.

3.13.2 Signature generation

This function computes a signature according to ANSI X9.62 [1]. All inputs and outputs are passed that they enable the user to implement applications that follow this standard without unnecessary work like reformatting of inputs and outputs.

Prototype

```
eslt_ErrorCode esl_signDSAEcP_prim(
                 eslt_WorkSpaceEcP     *workSpace,
           const eslt_Length            messageLength,
           const eslt_Byte             *message,
           const eslt_Byte             *privateKey,
                 eslt_Length           *signature_rLength,
                 eslt_Byte             *signature_r,
                 eslt_Length           *signature_sLength,
                 eslt_Byte             *signature_s );
```

Description

This function implements the ECDSA signature generation.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by init-signDSAEcP.
messageLength	The byte length of the input message. The maximum length is given by esl_getMaxLengthOfEcPmessage().
message	This pointer references the message that has to be signed. The message is interpreted as array of bytes.
privateKey	This pointer references the private key that will be used to sign the message (cf. 3.12.2).
signature_rLength	A pointer to a memory area holding the maximum length of r. This has to be at least the byte length of the public modulus.
signature_r	A pointer to a memory area, where the function stores the signature component r.
signature_sLength	A pointer to a memory area holding the maximum length of s. This has to be at least the byte length of the public modulus.
signature_s	A pointer to a memory area, where the function stores the signature component s.

Output

Parameter	Remark
*signature_rLength	The byte length of the generated signatures component r.
*signature_r	The first signature component r. The signature component is stored as arrays of bytes with lengths given by signature_rLength.
*signature_sLength	The byte length of the generated signatures component s.
*signature_s	The second signature component s. The signature component is stored as arrays of bytes with lengths given by signature_sLength.

Return codes

Return code	Remark
ESL_ERC_ECC_MESSAGE_TOO_LONG	The given message is invalid: the bit length of the message is greater than the bit length of the order of the given elliptic curve.

ESL_ERC_ECC_PRIVKEY_INVALID	The given private key is invalid (must be smaller than the order of the base point and not zero).
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	
Additionally all possible return codes from the function esl_getBytesRNG can be returned from this function (cf. 4).	

Prerequisites

The initialization function must have been executed successfully before (cf. 3.13.1).

3.13.3 Signature verification initialization

The following function initializes the ECDSA verification primitive.

Prototype

```
eslt_ErrorCode esl_initVerifyDSAEcP_prim(
                 eslt_WorkSpaceEcP          *workSpace,
       const     eslt_EccDomain         ROM *domain,
       const     eslt_EccDomainExt      ROM *domainExt);
```

Description

This function does the necessary initializations for the ECDSA verification.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that has to be at least ESL_MAXSIZEOF_WS_ECP bytes long. In this memory all computations are done during the VerifyDSAEcP operation.
domain	This pointer references the domain parameters of an elliptic curve (cf. 3.12.1).
domainExt	This pointer references the additional domain information.

Output

None.

Return codes

Return code	Remark
ESL_ERC_ECC_DOMAIN_INVALID	The domain parameters contain invalid data.
ESL_ERC_ECC_DOMAINEXT_INVALID	The domainExt parameters contain invalid data.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The initialization function must have been executed successfully before (cf. 3.13.1).

3.13.4 Signature verification

The following function implements the ECDSA signature revivification according to ANSI X9.62 [1]. Successful verification is indicated by `ESL_ERC_NO_ERROR`.

Prototype

```
eslt_ErrorCode esl_verifyDSAEcP_prim(
                    eslt_WorkSpaceEcP     *workSpace,
              const eslt_Length            messageLength,
              const eslt_Byte             *message,
              const eslt_Byte             *publicKey_x,
              const eslt_Byte             *publicKey_y,
              const eslt_Length            signature_rLength,
              const eslt_Byte             *signature_r,
              const eslt_Length            signature_sLength,
              const eslt_Byte             *signature_s);
```

Description

This function verifies an ECDSA signature.

Input

Parameter	Remark
`workSpace`	A pointer to a memory area (context) that was initialized by esl_initVerifyDSAEcP_prim.
`messageLength`	The byte length of the input message. The maximum length is given by esl_getMaxLengthOfEcPmessage().
`message`	This pointer references the message that has to be signed. The message is interpreted as array of bytes.
`*publicKey_x`, `*publicKey_y`	The coordinates of the public key that is used to verify the signature (cf. 3.12.2).
`signature_rLength`	The byte length of the signature component r; the length can be computed with the function esl_getLengthOfEcPsignature_comp().
`signature_r`	A pointer to the first signature component r
`signature_sLength`	The byte length of the signature component s; the length can be computed with the function esl_getLengthOfEcPsignature_comp().
`signature_s`	A pointer to the second signature component s.

Output

None.

Return codes

Return code	Remark
`ESL_ERC_ECC_SIGNATURE_INVALID`	The function executed successfully and the signature is invalid.
`ESL_ERC_ECC_PUBKEY_INVALID`	The given public key is invalid.
`ESL_ERC_ECC_MESSAGE_TOO_LONG`	The given message is invalid: the bit length of the message is greater than the bit length of the order of the given elliptic curve.

ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID

Prerequisites

The initialization function must have been executed successfully before (cf. 3.13.3).

3.14 EC key generation

The following functions implement the ECDSA key generation according to ANSI X9.62 [1].

3.14.1 Key generation initialization

The following function does the necessary initializations for the ECDSA key generation.

Prototype

```
eslt_ErrorCode esl_initGenerateKeyEcP_prim(
                eslt_WorkSpaceEcP          *workSpace,
        const   eslt_EccDomain         ROM *domain,
        const   eslt_EccDomainExt      ROM *domainExt,
        const   eslt_EccSpeedUpExt     ROM *speedUpExt);
```

Description

This function implements the initialization of the ECDSA key generation.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that has to be at least ESL_MAXSIZEOF_WS_ECP bytes long. In this memory all computations are done during the GenerateKeyEcP_prim operation.
domain	This pointer references the domain parameters of an elliptic curve (cf. 3.12.1).
domainExt	This pointer references the additional domain information.
speedUpExt	This pointer references the additional data needed to speed up the computations.

Output

None.

Return codes

Return code	Remark
ESL_ERC_ECC_DOMAIN_INVALID	The domain parameters contain invalid data.
ESL_ERC_ECC_DOMAINEXT_INVALID	The domainExt parameters contain invalid data.
ESL_ERC_ECC_SPEEDUPEXT_INVALID	The speedUpExt parameters contain invalid data.

ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID

Prerequisites

The initialization function must have been executed successfully before (cf. 3.15.1).

3.14.2 Key generation

The following function implements the ECDSA key generation according to ANSI X9.62 (cf. [1]).

Prototype

```
eslt_ErrorCode esl_generateKeyEcP_prim(
                    eslt_WorkSpaceEcP     *workSpace,
                    eslt_Byte             *privateKey,
                    eslt_Byte             *publicKey_x,
                    eslt_Byte             *publicKey_y);
```

Description

This function generates an ECDSA key pair.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by init-genrateKeyDSAEcP.
privateKey	A pointer to a memory area to store the private key; the needed size can be computed by the function esl_getLengthOfEcPprivateKey() (cf. 3.12.2).
publicKey_x	A pointer to a memory area to store the x-coordinate of the public key; the needed size can be computed by the function esl_getLengthOfEcPpublicKey_comp() (cf. 3.12.2).
publicKey_y	A pointer to a memory area to store the x-coordinate of the public key; the needed size can be computed by the function esl_getLengthOfEcPpublicKey_comp() (cf. 3.12.2).

Output

Parameter	Remark
*publicKey_x	The x-component of the generated public key.
*publicKey_y	The y-component of the generated public key
*privateKey	The generated private key.

Return codes

Return code	Remark
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	
All possible return codes from the function esl_getBytesRNG can be returned from this function (cf. 4).	

Prerequisites

The initialization function must have been executed successfully before (cf. 3.14.1).

3.15 EC-DH key exchange

The following function implements the EC-DH key exchange according to ANSI X9.63 [2].

3.15.1 Common secret initialization

The following function does the necessary initializations (for a part of the Diffie-Hellman key exchange) needed for the generation of a common secret.

Prototype

```
eslt_ErrorCode esl_initGenerateSharedSecretDHEcP_prim(
                eslt_WorkSpaceECP          *workSpace,
          const eslt_EccDomain        ROM *domain,
          const eslt_EccDomainExt     ROM *domainExt);
```

Description

This function does the necessary initializations for the generation of a common secret.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that has to be at least ESL_MAXSIZEOF_WS_ECP bytes long.
domain	This pointer references the domain parameters of an elliptic curve (cf. 3.12.1).
domainExt	This pointer references the additional domain information.

Output

None.

Return codes

Return code	Remark
ESL_ERC_ECC_DOMAIN_INVALID	The domain parameters contain invalid data.
ESL_ERC_ECC_DOMAINEXT_INVALID	The domainExt parameters contain invalid data.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The general initialization function esl_initWorkSpaceHeader (cf. 3.4) must have been executed successfully before.

3.15.2 Common secret generation

This function generates the common secret according to ANSI X9.63 [2].

Prototype

```
eslt_ErrorCode esl_generateSharedSecretDHEcP_prim(
```

```
                eslt_WorkSpaceEcP  *workSpace,
const           eslt_Byte          *privateKey,
const           eslt_Byte          *publicKey_x,
const           eslt_Byte          *publicKey_y,
                eslt_Byte          *secret_x,
                eslt_Byte          *secret_y);
```

Description

This function generates a common secret value.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initGenerateSharedSecretDHEcP_prim.
*privateKey	The private key (cf. 3.12.2).
*publicKey_x, *publicKey_y	The x- and y-coordinate of the public key of the communication partner (cf. 3.12.2).
secret_x, secret_y	These pointers reference memory to store the common secret's coordinates; the function esl_getLengthOfEcPsecret_comp() computes the needed length. Remark: In general (cf. [11]) only the value secret_x is used to derive an appropriate key.

Output

Parameter	Remark
*secret_x, *secret_y	the x- and y-coordinate of the common secret; stored as array of byte with length according to esl_getLengthOfEcPsecret_comp(). Remark: In general (cf. [11]) only the value secret_x is used to derive an appropriate key.

Return codes

Return code	Remark
ESL_ERC_ECC_PRIVKEY_INVALID	The given private key is invalid (must be smaller than the order of the base point and not equal zero).
ESL_ERC_ECC_PUBKEY_INVALID	The given public key is invalid.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The initialization function must have been executed successfully before (cf. 3.15.1).

3.15.3 Key exchange initialization

The following function does the necessary initializations for the Diffie-Hellman key exchange.

Prototype

```
eslt_ErrorCode esl_initExchangeKeyDHEcP_key(
                               eslt_WorkSpaceEcP     *workSpace,
```

```
                                const eslt_EccDomain    ROM *domain,
                                const eslt_EccDomainExt ROM *domainExt );
```

Description

This function does the necessary initializations for the EC-DH key exchange.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that has to be at least ESL_MAXSIZEOF_WS_ECP bytes long. In this memory all computations are done during the ExchangeKeyDHEcP_key operation.
domain	This pointer references the domain parameters of an elliptic curve (cf. 3.12.1).
domainExt	This pointer references the additional domain information.

Output

None.

Return codes

Return code	Remark
ESL_ERC_ECC_DOMAIN_INVALID	The domain parameters contain invalid data.
ESL_ERC_ECC_DOMAINEXT_INVALID	The domainExt parameters contain invalid data.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The general initialization function esl_initWorkSpaceHeader (cf. 3.4) must have been executed successfully before.

3.15.4 Key exchange

This function generates the key according to ANSI X9.63 [2] using the KDF-2 derivation function (cf. 3.11.2).

Prototype

```
eslt_ErrorCode  esl_exchangeKeyDHEcP_key(
                eslt_WorkSpaceEcP    *workSpace,
     const      eslt_Byte            *privateKey,
     const      eslt_Byte            *publicKey_x,
     const      eslt_Byte            *publicKey_y,
                eslt_Length           infoLength,
     const      eslt_Byte            *info,
                eslt_Length           iterationCount,
                eslt_Length           keyLength,
                eslt_Byte            *key          );
```

Description

This function calculates the common secret and derives the output key from the x-coordinate of the secret.

Input

Parameter	Remark
`workSpace`	A pointer to a memory area (context) that was initialized by esl_initExchangeKeyDHEcP_key.
`privateKey`	The private key (cf. 3.12.2).
`*publicKey_x`	The x-coordinate of the public key of the communication partner (cf. 3.12.2).
`*publicKey_y`	The y-coordinate of the public key of the communication partner (cf. 3.12.2).
`infoLength`	The number of bytes that represent some extra information. This length can be zero, as this parameter is optional.
`info`	A pointer to some optional extra information that is taken into account during derivation of a key (cf. [17]).
`iterationCount`	The number of rounds that have to be performed to get the derived key. The constant `ESL_KDF2_DEFAULT_ITERATION_COUNT` can be used here.
`keyLength`	The byte length of the key to be generated.
`key`	A pointer to memory with enough space (keyLength bytes) to store the generated key.

Output

Parameter	Remark
`*key`	The generated key with length keyLength; the pointer and the value keyLength was already passed by the corresponding initialization function.

Return codes

Return code	Remark
`ESL_ERC_ECC_PRIVKEY_INVALID`	The given private key is invalid (must be smaller than the order of the base point).
`ESL_ERC_ECC_PUBKEY_INVALID`	The given public key is invalid.
`ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID`	

Prerequisites

The initialization function must have been executed successfully before (cf. 3.15.3).

3.16 RSA based digital signature generation/verification (RSASSA-PKCS1-V1_5)

The following functions implement the RSA based signature computation and verification according to RSASSA-PKCS1-V1_5 as specified in [19]. While the following description only explains the

SHA1 based version, the RIPEMD160 variant is used exactly the same way. The respective function prototypes are named `~RIPEMD160_V15`.

3.16.1 Signature generation initialization – CRT version

The following function initializes the RSA signature generation (CRT version) including the necessary processing of the input message according to RSASSA-PKCS1-V1_5 as specified in [19].

Prototype

```
eslt_ErrorCode esl_initSignRSACRTSHA1_V15 (
                eslt_WorkSpaceRSACRTsig   *workSpace,
                eslt_Length                keyPairPrimePSize,
          const eslt_Byte            ROM *keyPairPrimeP,
                eslt_Length                keyPairPrimeQSize,
          const eslt_Byte            ROM *keyPairPrimeQ,
                eslt_Length                privateKeyExponentDPSize,
          const eslt_Byte            ROM *privateKeyExponentDP,
                eslt_Length                privateKeyExponentDQSize,
          const eslt_Byte            ROM *privateKeyExponentDQ,
                eslt_Length                privateKeyInverseQISize,
          const eslt_Byte            ROM *privateKeyInverseQI     );
```

Description

This function performs the necessary initializations for the RSA signature generation (RSASSA-PKCS1-V1_5).

Input

Parameter	Remark
`workSpace`	A pointer to a memory area (context) that has to be at least ESL_MAXSIZEOF_WS_RSA_CRT_SIG bytes long.
`keyPairPrimePSize`	The byte length of the secret prime *p*.
`*keyPairPrimeP`	The secret prime *p*.
`keyPairPrimeQSize`	The byte length of the secret prime *q*.
`*keyPairPrimeQ`	The secret prime *q*.
`privateKeyExponentDPSize`	The byte length of the private key component d_p.
`*privateKeyExponentDP`	The private key component d_p.
`privateKeyExponentDQSize`	The byte length of the private key component d_q.
`*privateKeyExponentDQ`	The private key component d_q.
`privateKeyInverseQISize`	The byte length of the private key component $q^{-1} \bmod p$.
`*privateKeyInverseQI`	The private key component $q^{-1} \bmod p$.

Output

None.

Return codes

Return code	Remark
`ESL_ERC_RSA_MODULE_OUT_OF_RANGE`	The byte length of the public module *n* (= *p***q*) is smaller than 46 (minimum length according to PKCS #1 V1.5).
`ESL_ERC_RSA_PRIVKEY_INVALID`	The given private key is invalid.

ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID

Prerequisites

The general initialization function esl_initWorkSpaceHeader (cf. 3.4) must have been executed successfully before.

3.16.2 Signature generation update – CRT version

This function provides a high level interface to the RSA signature generation (CRT version). It concatenates many succeeding inputs and computes the RSA signature over the complete input string.

Prototype

```
eslt_ErrorCode esl_updateSignRSACRTSHA1_V15 (
                        eslt_WorkSpaceRSACRTsig    *workSpace,
                        eslt_Length                 inputSize,
                  const eslt_Byte                  *input);
```

Description

This function updates the internal state of the RSA signature computation by processing the given input block. This function can be called consecutively several times until the complete input message is handled.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initSignRSACRTSHA1_V15.
inputSize	The byte length of the input.
input	A pointer to parts of the input (or the complete input) that has to be signed. The user can pass more than one input area by consecutive calls of this function: This function computes the signature over the concatenation of all inputs.

Output

None.

Return codes

Return code	Remark
ESL_ERC_HASH_TOTAL_LENGTH_OVERFLOW	The total size of the input exceeds 2^{56} bytes.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The initialization function must have been executed successfully before (cf. 3.16.2).

3.16.3 Signature generation finalize – CRT version

This function finishes the RSA signature generation (CRT version, RSASSA-PKCS1-V1_5).

Prototype

```
eslt_ErrorCode esl_finalizeSignRSACRTSHA1_V15 (
                        eslt_WorkSpaceRSACRTsig    *workSpace,
                        eslt_Length                *signatureSize,
                        eslt_Byte                  *signature   );
```

Description

This function finalizes the RSA signature generation.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initSignRSACRTSHA1_V15.
signatureSize	A pointer to a memory area to store the length of the signature.
signature	A pointer to a memory area, where the function stores the signature.

Output

Parameter	Remark
*signatureSize	The byte length of the signature.
*signature	The computed signature.

Return codes

Return code	Remark
ESL_ERC_OUTPUT_SIZE_TOO_SHORT	The given length (*signatureSize) for the output buffer is too short.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The esl_updateSignRSACRTSHA1_V15 must have been executed successfully before (cf. 3.16.2).

3.16.4 Signature generation initialization – plain version

The following function initializes the RSA signature primitive (plain version) including the necessary processing of the input message according to RSASSA-PKCS1-V1_5 (cf. [19]).

Prototype

```
eslt_ErrorCode esl_initSignRSASHA1_V15 (
                        eslt_WorkSpaceRSAsig  *workSpace,
                        eslt_Length            keyPairModuleSize,
              const     eslt_Byte         ROM *keyPairModule,
                        eslt_Length            privateKeyExponentSize,
```

```
                const     eslt_Byte          ROM *privateKeyExponent  );
```

Description

This function performs the necessary initializations for the RSA signature generation (RSASSA-PKCS1-V1_5).

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that has to be at least ESL_MAXSIZEOF_WS_RSA_SIG bytes long.
keyPairModuleSize	The byte length of the public module *n*.
*keyPairModule	The public module *n*.
privateKeyExponentSize	The byte length of the private exponent *d*.
*privateKeyExponent	The private exponent *d*.

Output

None.

Return codes

Return code	Remark
ESL_ERC_RSA_MODULE_OUT_OF_RANGE	The byte length of the public modulus *n* is smaller than 46 (minimum length according to PKCS #1 V1.5).
ESL_ERC_RSA_PRIVKEY_INVALID	The private exponent *d* is invalid, i.e. equal to zero or greater than *n-1*.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The general initialization function esl_initWorkSpaceHeader (cf. 3.4) must have been executed successfully before.

3.16.5 Signature generation update – plain version

This function provides a high level interface to the RSA signature generation (plain version). It concatenates many succeeding inputs and computes the RSA signature over the complete input string.

Prototype

```
eslt_ErrorCode esl_updateSignRSASHA1_V15 (
                        eslt_WorkSpaceRSAsig  *workSpace,
                        eslt_Length            inputSize,
                const   eslt_Byte             *input);
```

Description

This function updates the internal state of the RSA signature computation by processing the given input block. This function can be called consecutively several times until the complete input message is processed.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initSignRSASHA1_V15.
inputSize	The byte length of the input.
input	A pointer to parts of the input (or the complete input) that has to be signed. The user can pass more than one input area by consecutive calls of this function: This function computes the signature over the concatenation of all inputs.

Output

None.

Return codes

Return code	Remark
ESL_ERC_HASH_TOTAL_LENGTH_OVERFLOW	The total size of the input exceeds 2^{56} bytes.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The initialization function must have been executed successfully before (cf. 3.16.4).

3.16.6 Signature generation finalize – plain version

This function finishes the RSA signature generation (plain version, RSASSA-PKCS1-V1_5).

Prototype

```
eslt_ErrorCode esl_finalizeSignRSASHA1_V15 (
                        eslt_WorkSpaceRSAsig  *workSpace,
                        eslt_Length           *signatureSize,
                        eslt_Byte             *signature   );
```

Description

This function finalizes the RSA signature generation.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initSignRSASHA1_V15.
*signatureSize	The byte length of the memory area referenced by signature.
signature	A pointer to a memory area, where the function stores the signature.

Output

Parameter	Remark
*signatureSize	The byte length of the signature.

*signature	The computed signature.

Return codes

Return code	Remark
ESL_ERC_OUTPUT_SIZE_TOO_SHORT	The given length (*signatureSize) for the output buffer is too short.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The esl_updateSignRSASHA1_V15 must have been executed successfully before (cf. 3.16.5).

3.16.7 Signature verification initialization

The following function initializes the RSA verification primitive including the necessary processing of the input message according to RSASSA-PKCS1-V1_5 as specified in [19].

Prototype

```
eslt_ErrorCode esl_initVerifyRSASHA1_V15 (
                      eslt_WorkSpaceRSAver  *workSpace,
                      eslt_Length            keyPairModuleSize,
            const     eslt_Byte         ROM *keyPairModule,
                      eslt_Length            publicKeyExponentSize,
            const     eslt_Byte         ROM *publicKeyExponent);
```

Description

The function performs the necessary initializations for the RSA verification (RSASSA-PKCS1-V1_5).

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that has to be at least ESL_MAXSIZEOF_WS_RSA_VER bytes long.
keyPairModuleSize	The byte length of the public modulus *n*.
*keyPairModule	The public modulus *n*.
publicKeyExponentSize	The byte length of the public exponent *e*.
*publicKeyExponent	The public exponent *e*.

Output

None.

Return codes

Return code	Remark
ESL_ERC_RSA_PUBKEY_INVALID	The given public exponent is invalid, i.e. smaller than 3 or even.

ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID

Prerequisites

The general initialization function esl_initWorkSpaceHeader (cf. 3.4) must have been executed successfully before.

3.16.8 Signature verification update

This function provides a high level interface to the RSA verification. It concatenates many succeeding inputs and verifies an RSA signature over the complete input string.

Prototype

```
eslt_ErrorCode esl_updateVerifyRSASHA1_V15 (
                        eslt_WorkSpaceRSAver  *workSpace,
                        eslt_Length           inputSize,
                  const eslt_Byte             *input);
```

Description

This function updates the internal state of the RSA signature verification by processing the given input block. This function can be called consecutively several times until the complete input message to verify is processed.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initVerifyRSASHA1_V15.
inputSize	The byte length of the input.
input	A pointer to parts of the input (or the complete input) that has to be verified. The user can pass more than one input area by consecutive calls of this function: This function verifies the message given by the concatenation of all inputs.

Output

None.

Return codes

Return code	Remark
ESL_ERC_HASH_TOTAL_LENGTH_OVERFLOW	The total size of the input exceeds 2^{56} bytes.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The initialization function must have been executed successfully before (cf. 3.16.7).

3.16.9 Signature verification finalize

This function finishes the RSA verification (RSASSA-PKCS1-V1_5). Successful verification is indicated by `ESL_ERC_NO_ERROR`.

Prototype

```
eslt_ErrorCode esl_finalizeVerifyRSASHA1_V15 (
                      eslt_WorkSpaceRSAver  *workSpace,
                      eslt_Length            signatureSize,
                const eslt_Byte             *signature   );
```

Description

This function finalizes the RSA signature generation.

Input

Parameter	Remark
`workSpace`	A pointer to a memory area (context) that was initialized by esl_initVerifyRSASHA1_V15.
`signatureSize`	The byte length of the given signature.
`*signature`	The given signature.

Output

None.

Return codes

Return code	Remark
`ESL_ERC_RSA_SIGNATURE_OUT_OF_RANGE`	The given signature is greater or equal to the public modulus *n*.
`ESL_ERC_RSA_ENCODING_INVALID`	The message encoding inside the given signature is invalid.
`ESL_ERC_RSA_NO_RSA_WITH_SHA1_SIGNATURE`	The given algorithm identifier within the signature is invalid (not RSA with SHA-1).
`ESL_ERC_RSA_SIGNATURE_INVALID`	The signature is invalid.
`ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID`	

Prerequisites

The esl_updateVerifyRSASHA1_V15 must have been executed successfully before (cf. 3.16.8).

3.17 RSA based encryption/decryption (RSAES-PKCS1-V1_5)

The following functions implement the RSA based encryption and decryption according to RSAES-PKCS1-V1_5 as specified in [19].

3.17.1 Encryption initialization

The following function initializes the RSA encryption primitive including the necessary padding of the input:

Prototype

```
eslt_ErrorCode esl_initEncryptRSA_V15 (
                 eslt_WorkSpaceRSAenc       *workSpace,
                 eslt_Length                 keyPairModuleSize,
          const  eslt_Byte             ROM *keyPairModule,
                 eslt_Length                 publicKeyExponentSize,
          const  eslt_Byte             ROM *publicKeyExponent);
```

Description

This function performs the necessary initializations for the RSA encryption (RSASSA-PKCS1-V1_5).

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that has to be at least ESL_MAXSIZEOF_WS_RSA_ENC bytes long.
keyPairModuleSize	The byte length of the public modulus *n*.
*keyPairModule	The public modulus *n*.
publicKeyExponentSize	The byte length of the public exponent *e*.
*publicKeyExponent	The public exponent *e*.

Output

None.

Return codes

Return code	Remark
ESL_ERC_RSA_PUBKEY_INVALID	The given public exponent is invalid, i.e. smaller than 3 or even.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The general initialization function esl_initWorkSpaceHeader (cf. 3.4) must have been executed successfully before.

3.17.2 Encryption

The following function performs the RSA encryption primitive including the necessary padding of the input:

Prototype

```
eslt_ErrorCode esl_encryptRSA_V15(
                      eslt_WorkSpaceRSAenc   *workSpace,
                      eslt_Length             messageSize,
```

```
                    const     eslt_Byte              *message,
                              eslt_Length            *cipherSize,
                              eslt_Byte              *cipher);
```

Description

This function performs the RSA encryption.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initEncryptRSA_V15.
messageSize	The byte length of the message to encrypt.
*message	The message to encrypt.
*cipherSize	The byte length of the memory area referenced by cipher.
cipher	A pointer to the memory where the encrypted input will be stored.

Output

Parameter	Remark
*cipherSize	The byte length of the encrypted message.
*cipher	The encrypted message.

Return codes

Return code	Remark
ESL_ERC_RSA_MESSAGE_OUT_OF_RANGE	The given messageSize is greater than keyPairModulSize – 11.
ESL_ERC_OUTPUT_SIZE_TOO_SHORT	The given length (*cipherSize) for the output buffer is too short.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	
Additionally all possible return codes from the function esl_getBytesRNG can be returned from this function (cf. 4).	

Prerequisites

The initialization function must have been executed successfully before (cf. 3.17.1).

3.17.3 Decryption initialization – CRT version

The following function initializes the RSA decryption primitive including the necessary padding of the input:

Prototype

```
eslt_ErrorCode esl_initDecryptRSACRT_V15 (
                        eslt_WorkSpaceRSACRTdec  *workSpace,
                        eslt_Length               keyPairPrimePSize,
                const   eslt_Byte            ROM *keyPairPrimeP,
                        eslt_Length               keyPairPrimeQSize,
                const   eslt_Byte            ROM *keyPairPrimeQ,
                        eslt_Length               privateKeyExponentDPSize,
                const   eslt_Byte            ROM *privateKeyExponentDP,
                        eslt_Length               privateKeyExponentDQSize,
```

```
const   eslt_Byte            ROM *privateKeyExponentDQ,
        eslt_Length               privateKeyInverseQISize,
const   eslt_Byte            ROM *privateKeyInverseQI    );
```

Description

This function performs the necessary initializations for the RSA decryption (RSAES-PKCS1-V1_5).

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that has to be at least ESL_MAXSIZEOF_WS_RSA_CRT_DEC bytes long.
keyPairPrimePSize	The byte length of the secret prime *p*.
*keyPairPrimeP	The secret prime *p*.
keyPairPrimeQSize	The byte length of the secret prime *q*.
*keyPairPrimeQ	The secret prime *q*.
privateKeyExponentDPSize	The byte length of the private key component d_p.
*privateKeyExponentDP	The private key component d_p.
privateKeyExponentDQSize	The byte length of the private key component d_q.
*privateKeyExponentDQ	The private key component d_q.
privateKeyInverseQISize	The byte length of the private key component $q^{-1} \bmod p$.
*privateKeyInverseQI	The private key component $q^{-1} \bmod p$.

Output

None.

Return codes

Return code	Remark
ESL_ERC_RSA_PRIVKEY_INVALID	The given private key is invalid.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The general initialization function esl_initWorkSpaceHeader (cf. 3.4) must have been executed successfully before.

3.17.4 Decryption – CRT version

The following function performs the RSA decryption primitive (CRT version) including the necessary padding of the input:

Prototype

```
eslt_ErrorCode esl_decryptRSACRT_V15(
                        eslt_WorkSpaceRSACRTdec    *workSpace,
                        eslt_Length                 cipherSize,
                const   eslt_Byte                  *cipher
                        eslt_Length                *messageSize,
                        eslt_Byte                  *message );
```

Description

This function performs the RSA decryption (CRT version).

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initDecyptRSACRT_V15.
*cipherSize	The byte length of the cipher to decrypt.
cipher	The cipher to decrypt.
*messageSize	The byte length of the memory area referenced by message.
*message	A pointer to the memory where the decrypted input will be stored.

Output

Parameter	Remark
*messageSize	The byte length of the decrypted message.
*message	The decrypted message.

Return codes

Return code	Remark
ESL_ERC_RSA_CODE_OUT_OF_RANGE	The given cipher is greater or equal to the public modulus *n*.
ESL_ERC_RSA_ENCODING_INVALID	The given encoding of the input parameter is invalid.
ESL_ERC_OUTPUT_SIZE_TOO_SHORT	The given length (*messageSize) for the output buffer is too short.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The initialization function must have been executed successfully before (cf. 3.17.3).

3.17.5 Decryption initialization – plain version

The following function initializes the RSA decryption primitive including the necessary padding of the input:

Prototype

```
eslt_ErrorCode esl_initDecryptRSA_V15 (
                    eslt_WorkSpaceRSAdec  *workSpace,
                    eslt_Length            keyPairModuleSize,
          const     eslt_Byte         ROM *keyPairModule,
                    eslt_Length            privateKeyExponentSize,
          const     eslt_Byte         ROM *privateKeyExponent  );
```

Description

This function performs the necessary initializations for the RSA decryption (RSAES-PKCS1-V1_5).

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that has to be at least ESL_MAXSIZEOF_WS_RSA_DEC bytes long.
keyPairModuleSize	The byte length of the public modulus *n*.
*keyPairModule	The public modulus *n*.
privateKeyExponentSize	The byte length of the private exponent *d*.
*privateKeyExponent	The private exponent *d*.

Output

None.

Return codes

Return code	Remark
ESL_ERC_RSA_PRIVKEY_INVALID	The private exponent *d* is invalid, i.e. equal to zero or greater than *n-1*.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The general initialization function esl_initWorkSpaceHeader (cf. 3.4) must have been executed successfully before.

3.17.6 Decryption – plain version

The following function performs the RSA decryption primitive (plain version) including the necessary padding of the input:

Prototype

```
eslt_ErrorCode esl_decryptRSA_V15(
                        eslt_WorkSpaceRSAdec  *workSpace,
                        eslt_Length            cipherSize,
                  const eslt_Byte             *cipher
                        eslt_Length           *messageSize,
                        eslt_Byte             *message );
```

Description

This function performs the RSA decryption (plain version).

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initDecyptRSACRT_V15.
*cipherSize	The byte length of the cipher to decrypt.
cipher	The cipher to decrypt.
*messageSize	The byte length of the memory area referenced by message.
*message	A pointer to the memory where the decrypted input will be stored.

Output

Parameter	Remark
`*messageSize`	The byte length of the decrypted message.
`*message`	The decrypted message.

Return codes

Return code	Remark
`ESL_ERC_RSA_CODE_OUT_OF_RANGE`	The given cipher is greater or equal to the public modulus *n*.
`ESL_ERC_RSA_ENCODING_INVALID`	The given encoding of the input parameter is invalid.
`ESL_ERC_OUTPUT_SIZE_TOO_SHORT`	The given length (*messageSize) for the output buffer is too short.
`ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID`	

Prerequisites

The initialization function must have been executed successfully before (cf. 3.17.5).

3.18 Random number generation

The implemented random number generator (RNG) is realized as a pseudo random number generator (PRNG) according to FIPS PUB 186-2 [8]. The PRNG has to be initialized with a “true” random input value, the so-called *seed*. The FIPS186-2 standard requires a seed of at least 160 random bits. Typical input values for the seed are internal timing variables or – if possible – measured values of physical variables like input voltage or temperature, especially the least significant bits of this information. Since the seed provides the entropy for the whole random number generation process, it has to be carefully selected by the developer.

The following two sections (3.18.1 and 3.18.2) contain the interface description of the implemented PRNG.

The internal functions that need random numbers (e.g. the AES key generation, cf. 3.5.11) do not directly access the random number generator. The reason for this is the use of these functions in a multi-threaded system. For the random numbers needed by the cryptographic routines a callback function has to be provided by the user of the library as described in chapter 4.1. This function will typically use the FIPS-RNG.

3.18.1 RNG-FIPS-186 initialization

The following function initializes the random number generator provided by the library according to FIPS PUB 186-2 [8].

As each pseudo random number generator (PRNG) generates the same numbers each time it starts from the same initial state, this function has to be seeded with some nearly randomly (at least unpredictable) generated bytes (entropy) before any random numbers can be generated by the PRNG. As it may happen, due to an attack or by accident, that the source of entropy fails to deliver good enough input, the initialization function is designed in a way, that it combines the new entropy with some value (savedState) that was saved from the previous initialization. This protects the PRNG to deliver duplicates, even if the entropy source breaks down. The initialization has to be done only once during boot-up-phase. Saving this savedState is optional, but we strongly recom-

mend to use this feature as not using it results in identical PRNG-bytes if seeded with the same entropy. Using a NULL-pointer causes the function esl_initRNGFIPS186 to only use the given entropy for seeding.

The required workSpace must persist as long as the PRNG is to be used. So it will typically be static. The size of the workSpace is given by the constant:

```
ESL_MAXSIZEOF_WS_FIPS186
```

Prototype

```
eslt_ErrorCode esl_initFIPS186(
                  eslt_WorkSpaceFIPS186 *workSpace,
      const       eslt_Length            entropyLength,
      const       eslt_Byte             *entropy,
                  eslt_Byte             *savedState          );
```

Description

This function does the necessary initializations for the random number generator provided by the library.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initWorkSpaceHeader (cf. 3.4). The size of this area has to be at least ESL_MAXSIZEOF_WS_FIPS186 bytes long.
entropyLength	This length defines the number of bytes of entropy that are used to initialize the random number generator.
entropy	This pointer points to the input string (a random value) used as seed for the random number generator.
savedState	This pointer points to the state computed during the last initialization, in order to guarantee new random numbers, even if the entropy source has broken down. The length has to be ESL_SIZEOF_RNGFIPS186_STATE. This value is optional; NULL-pointer causes the function to ignore this input.

Output

Parameter	Remark
*savedState	The updated state to be persistently saved by the user for next startup.

Return codes

Return code	Remark
ESL_ERC_RNG_ENTROPY_TOO_SMALL	Not enough entropy.
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The general initialization function esl_initWorkSpaceHeader (cf. 3.4) must have been executed successfully before.

Prototype

```
eslt_ErrorCode esl_stirFIPS186(
                eslt_WorkSpaceFIPS186 *workSpace,
     const      eslt_Length            inputLength,
                eslt_Byte             *input          );
```

Description

This function stirs up the internal state of the random number generator.

Input

Parameter	Remark
workSpace	A pointer to a memory area (context) that was initialized by esl_initWorkSpaceHeader (cf. 3.4). The size of this area has to be at least ESL_MAXSIZEOF_WS_FIPS186 bytes long.
inputLength	This length defines the number of bytes of entropy that are used to stir up the random number generator.
input	This pointer points to the input string (a random value) that brings (additional) entropy into the random number generator.

Output

Parameter	Remark
*workSpace	The updated workSpace.

Return codes

Return code	Remark
ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID	

Prerequisites

The initialization function must have been executed successfully before.

3.18.2 RNG-FIPS-186 random number generation

This function implements the random number generator core routine. It generates as many random bytes as needed according to the algorithm described in [8] with an underlying hash-function. As random bytes are generated block by block, the remaining and unused bytes are stored internally (workSpace) and will be used later during the next call.

Prototype

```
eslt_ErrorCode esl_getBytesFIPS186(
           eslt_WorkSpace     *workSpace,
     const eslt_Length         targetLength,
           eslt_Byte          *target          );
```

Description

This function implements the core part of the pseudo random number generator.

Input

Parameter	Remark
`workSpace`	A pointer to a memory area (context) that was initialized by esl_initFIPS186.
`targetLength`	The number of random bytes that have to be generated.
`target`	A pointer to a memory area, where the generated random bytes are stored.

Output

Parameter	Remark
`*target`	The generated random bytes.

Return codes

Return code	Remark
`ESL_ERC_RNG_BAD_INTERNAL_STATE`	An internal error occurred.
`ESL_ERC_NO_ERROR, ESL_ERC_WS_STATE_INVALID, ESL_ERC_WS_TOO_SMALL, ESL_ERC_PARAMETER_INVALID`	

Prerequisites

The initialization function must have been executed successfully before (cf. 3.18.1).

4 Callback functions

4.1 RNG callback function

The user has to provide a callback function when using functions, which internally need random numbers, like the AES key generation. This callback function enhances the flexibility in multithreading systems. Section 4.1.1 describes a possible implementation of this callback function.

Prototype

```
eslt_ErrorCode  esl_getBytesRNG(
      const           eslt_Length  targetLength,
                      eslt_Byte   *target      );
```

Description

This function generates random bytes.

Input

Parameter	Remark
`targetLength`	The number of random bytes that shall be produced.
`target`	A pointer to a memory area where the generated random bytes will be stored.

Output

Parameter	Remark
`*target`	The generated random bytes. If an error has occurred these bytes are invalid and should not be used.

Return codes

The function must return `ESL_ERC_NO_ERROR` on success. Any other return code will be handed back to the caller.

Prerequisites

The random number generator must have been initialized by an adequate initialization function.

4.1.1 RNG sample implementation

The example implementation, which shows a possible implementation of the callback function for single threaded application, uses three functions that basically call the corresponding FIPS-RNG functions provided by the library (cf. 3.18.1 and 3.18.2). The main difference is that only the initialization function has the parameter workSpace. The other functions use a static variable to “remember” the workSpace location. These functions also contain comments on how a multithreaded application can be supported; the functions have to check if the RNG is used by any other application. If it is unused it locks the RNG and uses it. Afterwards it has to unlock the RNG. As the multithreading feature is realized differently on most machines or operating systems there is no de-

tailed example implementation. You can also find the implementation code in the `ESLib_RNG.c` file included in the demo applications.

For description of inputs and outputs of these functions please refer to the paragraph above.

Variables

```
static eslt_WorkSpaceRNG186 *esl_WS_RNG;
```

Prototypes

```
eslt_ErrorCode esl_initRNG(
                  eslt_WorkSpaceRNG186  *workSpace
     const        eslt_Length            entropyLength,
     const       eslt_Byte              *entropy,
                  eslt_Byte              *savedState   );

eslt_ErrorCode esl_getBytesRNG(
     const        eslt_Length            targetLength,
                  eslt_Byte              *target       );

eslt_ErrorCode esl_stirRNG(
     const        eslt_Length            inputLength,
                  eslt_Byte              *input        );
```

Implementation

```
eslt_ErrorCode
esl_initRNG(   eslt_WorkSpaceRNG     *workSpace,
    const      eslt_Length            entropyLength,
    const      eslt_Byte             *entropy,
               eslt_Byte             *savedState   )
{
   eslt_ErrorCode tmpERC;
   //lockRNG();

   // initialze static workspace pointer
   esl_WS_RNG = (eslt_WorkSpaceFIPS186*)workSpace;

   // call FIPS186 core generator
   tmpERC = esl_initFIPS186( esl_WS_RNG, entropyLength, entropy,
                             savedState);

   //unlockRNG();
   return tmpERC;
}
```

Description

The initialization simply calls the initialization of the FIPS186 RNG functions and stores the workSpace pointer into a static variable. This function has to be called before a cryptographic function is called which internally uses random numbers.

Implementation

```
eslt_ErrorCode
esl_getBytesRNG( const  eslt_Length  targetLength,
```

```
                        eslt_Byte    *target          )
{
   eslt_ErrorCode tmpERC;
   //lockRNG();

   // call FIPS186 core generator
   tmpERC = esl_getBytesFIPS186(esl_WS_RNG, targetLength, target);

   //unlockRNG();
   return tmpERC;
}
```

Description

The function `esl_getBytesRNG` reads the static variable and uses the saved pointer to the workSpace to complete the set of input parameters for the FIPS186 RNG and then calls that PRNG.

Implementation

```
eslt_ErrorCode
esl_stirRNG( const   eslt_Length         inputLength,
                     eslt_Byte          *input         )
{
   eslt_ErrorCode tmpERC;
   //lockRNG();

   // call FIPS186 core generator
   tmpERC = esl_stirFIPS186(esl_WS_RNG, inputLength, input);

   //unlockRNG();
   return tmpERC;
}
```

Description

The function `esl_stirRNG` reads the static variable and uses the saved pointer to the workSpace to complete the set of input parameters for the FIPS186 RNG.

4.2 Watchdog callback function

The library can be configured to include a watchdog-functionality at compile-time. In this case a watchdog-function can be passed to the initialization routine `esl_initWorkSpaceHeader (cf. 3.4)`, which will then be called from inside the library functions. A NULL-pointer disables this.

Prototype

```
void watchdog (void);
```

Table of return values

Return code	Remark
ESL_ERC_NO_ERROR	The function executed successfully. For verification functions this indicates successful verification.
ESL_ERC_WS_STATE_INVALID	The function cannot use the workSpace passed to it. This can be because it is initialized the wrong way or even not at all.
ESL_ERC_WS_TOO_SMALL	The function needs more memory to execute correctly.
ESL_ERC_PARAMETER_INVALID	A parameter passed to the function is not valid, e.g. a NULL-pointer is passed as the pointer to the workspace. See 3.2 for RSA and ECC parameters.
ESL_ERC_MEMORY_CONFLICT	Partially memory overlap between input and output, memory overlap between input and workSpace or output and workSpace.

Index of functions

Abbreviations

ECC	Elliptic Curve Cryptography
AES	Advanced Encryption Standard
DES	Data Encryption Standard
TDES	Triple Data Encryption Standard
RC2	Rivest Cipher 2
RSA	Rivest, Adleman, Shamir (inventors of the RSA system)
ECB	Mode of operation – electronic code book mode
CBC	Mode of operation – chaining block cipher
CFB	Mode of operation – cipher feedback block mode
OFB	Mode of operation – output feedback block mode
CTR	Mode of operation – Counter
PKCS	Public Key Cryptographic Standard
RNG	Random Number Generator
PRNG	Pseudo Random Number Generator
WS	WorkSpace

Glossary

workSpace Each algorithm needs some temporary memory (the so called workSpace) of an appropriate size.

This memory area must be first initialized with a common initialization routine.

Bibliography

[1] ANSI X9.62: *Public Key Cryptography for the Financial Service Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)*, American National Standards Institute (1999)

[2] ANSI X9.63: *Public Key Cryptography for the Financial Service Industry:* Key Agreement and Key Transport Using Elliptic Curve Cryptography, American National Standards Institute (1999)

[3] Blake, I.; Seroussi, G.; Smart, N.: "*Elliptic Curves in Cryptography*", Cambridge University Press (1999)

[4] cv cryptovision GmbH: *A short introduction to modern cryptography - explanatory supplement to the cv act library/es* (2005),
available online at http://www.cryptovision.com/en/technologies/whitepapers.html

[5] cv cryptovision GmbH: *Cryptographic Library for Embedded Systems - Footprint and Performance Data* (2008)

[6] NIST 800-38A: *Special Publication 800-38A 2001 ED,* National Institute of Standards and Technology (2001)

[7] FIPS 180-2: *Secure Hash Standard*, Federal Information Processing Standards Publication 180-2 (2002)

[8] FIPS PUB 186-2: *Digital Signature Standard (DSS),* Federal Information Processing Standards Publication 186-2 (2000)

[9] FIPS PUB 197: *Advanced Encryption Standard*, Federal Information Processing Standards Publication 197 (2001)

[10] FIPS PUB 198: *The Keyed-Hash Message Authentication Code (HMAC)*, Federal Information Processing Standards Publication 198 (2002)

[11] IEEE P1363: *Standard Specification for Public Key Cryptography*, Draft (1999), available online at http://grouper.ieee.org/groups/1363/index.html

[12] ISO/IEC 10118-3: *Information technology – Security techniques – Hash-functions – Part 3: Dedicated hash-functions,* International Organization for Standardization (2003)

[13] ITU-T X.680: *Information technology – Abstract Syntax Notation (ASN.1): Specification of basic notation*, International Telecommunication Union (2002)

[14] ITU-T X.690: *Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*, International Telecommunication Union (2002)

[15] Lenstra, A.; Verheul, E.: *Selecting cryptographic Key Sizes*, Journal of Cryptology, Springer Verlag (2002), available online at http://www.cryptosavvy.com

[16] Menezes, A.; Oorschot, P.; Vanstone, S.: *Handbook of applied Cryptography*, CRC Press (1996), available online at http://www.cacr.math.uwaterloo.ca/hac/

[17] PKCS #5 v2.0: *Password-Based Cryptography Standard*, RSA Laboratories (1999)

[18] PKCS #1 v1.5: *RSA Cryptography Standard*, RSA Laboratories (1993)

[19] PKCS #1 v2.1: *RSA Cryptography Standard*, RSA Laboratories (2002)

[20] RFC 2104: *HMAC: Keyed-Hashing for Message Authentication,* Internet RFC (1997)

[21] RFC 1321: The MD5 Message-Digest Algorithm (April 1992)

[22] FIPS PUB 46-3: *Data Encryption Standard*, Federal Information Processing Standards Publication 46-3 (1999)

[23] RFC 2268: *A Description of the RC2(r) Encryption Algorithm*, RFC 2268 (1998)