Document Title	Specification of Crypto Interface for Adaptive Platform
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	883

Document Status	Final
Part of AUTOSAR Standard	Adaptive Platform
Part of Standard Release	17-10

Document Change History

Date	Release	Changed by	Description
2017-10-27	17-10	AUTOSAR Release Management	• Initial release

Disclaimer

Table of Contents

1 Introduction and functional overview 4

2 Acronyms and Abbreviations 5

3 Related documentation 6

3.1 Input documents 6
3.2 Related standards and norms 6
3.3 Related specification 6

4 Constraints and assumptions 7

4.1 Limitations 7
4.2 Applicability to car domains 7

5 Dependencies to other functional clusters 8

6 Requirements Tracing 9

7 Functional specification 13

7.1 Architectural concepts 13
7.2 Integration of Adaptive Application and Crypto Stack 14
7.2.1 Process isolation 14
7.2.2 Hardware isolation 14
7.3 Supported algorithms 15

8 API specification 17

8.1 C++ language binding 17
8.1.1 API Header files 17
8.1.1.1 CryptoNeed header files 18
8.1.1.2 Common header files 18
8.1.2 API Reference 20
8.1.2.1 Common API 20
8.1.2.2 CryptoNeed API 31
8.2 Client Server Interfaces 32

A Mentioned Class Tables 45

B Span 47

1 Introduction and functional overview

This specification describes the functionality, API and the configuration for the AUTOSAR Adaptive Crypto Stack as part of the functional cluster Security Management of the AUTOSAR Adaptive Platform.
The Crypto Stack offers applications a standardized interface to cryptographic operations. The Crypto Stack realizes the APIs and manages actual implementations of operations, as well as management functionality handling configuration and brokering.

2 Acronyms and Abbreviations

The glossary below includes acronyms and abbreviations relevant to the Crypto Stack module that are not included in the AUTOSAR glossary [1].

Abbreviation / Acronym:	Description:
HSM	Hardware Security Module
TPM	Trusted Platform Module
IPC	Inter-Process Communication

3 Related documentation

3.1 Input documents

[1] Glossary
AUTOSAR_TR_Glossary

[2] Requirements on Security Management for Adaptive Platform
AUTOSAR_RS_SecurityManagement

[3] Specification of Manifest
AUTOSAR_TPS_ManifestSpecification

3.2 Related standards and norms

See chapter 3.1.

3.3 Related specification

See chapter 3.1.

4 Constraints and assumptions

4.1 Limitations

The current version of this document is missing some functionality that was available in the AUTOSAR Classic Platform:

- **Secure Counter**
  There is currently no API available to access secure counter primitives that an implementation may provide.

The following functionality is required but not worked out currently:

- **Asynchronous interface**
  Currently there is only a synchronous API specification and asynchronous behavior must be implemented by the client.
- **Memory management**
  An asynchronous interface requires a specification for managing memory and access to memory (e.g. shared state for `std::shared_ptr` or `std::future`). Currently this has to be addressed by the client.

4.2 Applicability to car domains

No restrictions to applicability.

5 Dependencies to other functional clusters

There are currently no dependencies to other functional clusters.

6 Requirements Tracing

The following tables reference the requirements specified in [2] and links to the fulfillment of these. Please note that if column "Satisfied by" is empty for a specific requirement this means that this requirement is not fulfilled by this document.

Requirement	Description	Satisfied by
[RS_CRYPTO_02001]	No description	[SWS_CRYPTO_01233] [SWS_CRYPTO_01234] [SWS_CRYPTO_01235] [SWS_CRYPTO_01236] [SWS_CRYPTO_01237] [SWS_CRYPTO_01238] [SWS_CRYPTO_01239] [SWS_CRYPTO_01240] [SWS_CRYPTO_01241] [SWS_CRYPTO_01242] [SWS_CRYPTO_01243]
[RS_CRYPTO_02002]	No description	[SWS_CRYPTO_01233] [SWS_CRYPTO_01234] [SWS_CRYPTO_01235] [SWS_CRYPTO_01236] [SWS_CRYPTO_01237] [SWS_CRYPTO_01238] [SWS_CRYPTO_01239] [SWS_CRYPTO_01240] [SWS_CRYPTO_01241] [SWS_CRYPTO_01242] [SWS_CRYPTO_01243]
[RS_CRYPTO_02101]	No description	[SWS_CRYPTO_01109] [SWS_CRYPTO_01242] [SWS_CRYPTO_01244]
[RS_CRYPTO_02102]	No description	[SWS_CRYPTO_01235] [SWS_CRYPTO_01238] [SWS_CRYPTO_01239] [SWS_CRYPTO_01241] [SWS_CRYPTO_01242] [SWS_CRYPTO_01245] [SWS_CRYPTO_01246]
[RS_CRYPTO_02103]	No description	[SWS_CRYPTO_01110] [SWS_CRYPTO_01248] [SWS_CRYPTO_01249]
[RS_CRYPTO_02104]	No description	[SWS_CRYPTO_01111] [SWS_CRYPTO_01250] [SWS_CRYPTO_01251] [SWS_CRYPTO_01252]
[RS_CRYPTO_02105]	No description	[SWS_CRYPTO_01109] [SWS_CRYPTO_01242] [SWS_CRYPTO_01246] [SWS_CRYPTO_01247]

Requirement	Description	Satisfied by
[RS_CRYPTO_02201]	No description	[SWS_CRYPTO_01103] [SWS_CRYPTO_01104] [SWS_CRYPTO_01203] [SWS_CRYPTO_01204] [SWS_CRYPTO_01205] [SWS_CRYPTO_01206] [SWS_CRYPTO_01207]
[RS_CRYPTO_02202]	No description	[SWS_CRYPTO_01103] [SWS_CRYPTO_01104] [SWS_CRYPTO_01203] [SWS_CRYPTO_01204] [SWS_CRYPTO_01205] [SWS_CRYPTO_01206] [SWS_CRYPTO_01207]
[RS_CRYPTO_02203]	No description	[SWS_CRYPTO_01104] [SWS_CRYPTO_01105] [SWS_CRYPTO_01106] [SWS_CRYPTO_01217] [SWS_CRYPTO_01218] [SWS_CRYPTO_01219] [SWS_CRYPTO_01220] [SWS_CRYPTO_01221] [SWS_CRYPTO_01222] [SWS_CRYPTO_01223] [SWS_CRYPTO_01225] [SWS_CRYPTO_01226] [SWS_CRYPTO_01227] [SWS_CRYPTO_01228]
[RS_CRYPTO_02204]	No description	[SWS_CRYPTO_01104] [SWS_CRYPTO_01105] [SWS_CRYPTO_01106] [SWS_CRYPTO_01217] [SWS_CRYPTO_01218] [SWS_CRYPTO_01219] [SWS_CRYPTO_01220] [SWS_CRYPTO_01221] [SWS_CRYPTO_01222] [SWS_CRYPTO_01223] [SWS_CRYPTO_01225] [SWS_CRYPTO_01226] [SWS_CRYPTO_01227] [SWS_CRYPTO_01228]
[RS_CRYPTO_02205]	No description	[SWS_CRYPTO_01107] [SWS_CRYPTO_01211] [SWS_CRYPTO_01212] [SWS_CRYPTO_01213] [SWS_CRYPTO_01214] [SWS_CRYPTO_01215] [SWS_CRYPTO_01216]
[RS_CRYPTO_02206]	No description	[SWS_CRYPTO_01108] [SWS_CRYPTO_01229] [SWS_CRYPTO_01230] [SWS_CRYPTO_01231]

Requirement	Description	Satisfied by
[RS_CRYPTO_02207]	No description	[SWS_CRYPTO_01103] [SWS_CRYPTO_01104] [SWS_CRYPTO_01203] [SWS_CRYPTO_01204] [SWS_CRYPTO_01205] [SWS_CRYPTO_01206] [SWS_CRYPTO_01207]
[RS_CRYPTO_02301]	No description	[SWS_CRYPTO_00001] [SWS_CRYPTO_00002] [SWS_CRYPTO_01001] [SWS_CRYPTO_01101] [SWS_CRYPTO_01102] [SWS_CRYPTO_01103] [SWS_CRYPTO_01104] [SWS_CRYPTO_01105] [SWS_CRYPTO_01106] [SWS_CRYPTO_01107] [SWS_CRYPTO_01108] [SWS_CRYPTO_01109] [SWS_CRYPTO_01110] [SWS_CRYPTO_01111] [SWS_CRYPTO_01112] [SWS_CRYPTO_01114] [SWS_CRYPTO_01201] [SWS_CRYPTO_01202] [SWS_CRYPTO_01301] [SWS_CRYPTO_01302]
[RS_CRYPTO_02302]	No description	[SWS_CRYPTO_01204] [SWS_CRYPTO_01205] [SWS_CRYPTO_01206] [SWS_CRYPTO_01213] [SWS_CRYPTO_01214] [SWS_CRYPTO_01215] [SWS_CRYPTO_01218] [SWS_CRYPTO_01219] [SWS_CRYPTO_01220] [SWS_CRYPTO_01225] [SWS_CRYPTO_01226] [SWS_CRYPTO_01227]
[RS_CRYPTO_02401]	No description	[SWS_CRYPTO_01208] [SWS_CRYPTO_01209] [SWS_CRYPTO_01210] [SWS_CRYPTO_01303] [SWS_CRYPTO_01304] [SWS_CRYPTO_01305] [SWS_CRYPTO_01306] [SWS_CRYPTO_01307] [SWS_CRYPTO_01308] [SWS_CRYPTO_01309] [SWS_CRYPTO_01310] [SWS_CRYPTO_01311]
[RS_CRYPTO_02402]	No description	[SWS_CRYPTO_01232]
[RS_CRYPTO_02403]	No description	[SWS_CRYPTO_01243]

Requirement	Description	Satisfied by
[RS_CRYPTO_02404]	No description	[SWS_CRYPTO_01236] [SWS_CRYPTO_01240] [SWS_CRYPTO_01245] [SWS_CRYPTO_01246]

7 Functional specification

The AUTOSAR Adaptive architecture organizes the software of the AUTOSAR Adaptive foundation as functional clusters. These clusters offer common functionality as services to the applications. The Security Management (SEC) for AUTOSAR Adaptive is such a functional cluster and is part of “AUTOSAR Runtime for Adaptive Applications” - ARA. The functional cluster consists of multiple modules. The Crypto Stack is a module of this functional cluster that offers interfaces to Adaptive applications. It is responsible for the construction and supervision of cryptographic primitives.

The Crypto Stack provides the infrastructure to access multiple implementations of cryptographic algorithms through a standardized interface.

This specification includes the syntax of the API, the relationship of the API to the model and describes semantics. The specification does not pose constraints on the internal architecture and implementation of the Crypto Stack.

7.1 Architectural concepts

The Crypto Stack of AUTOSAR Adaptive can be logically divided into the following parts:

- Language binding
- Drivers
- Crypto Stack management software

There are several types of interfaces available in the context of the Crypto Stack:

- **Public Interface**
  Part of the AUTOSAR Adaptive API and specified in this document. This is the standardized ara::sec::crypto API.
- **Protected Interface**
  Used for interaction between functional clusters. This may be a custom API but it can also re-use the Public Interface.
- **Private Interface**
  Used for interaction within the module. These interfaces are not described in the specification and are implementation-specific.

For the design of the ARA API the following constraints apply:

- Support the independence of application software components from a specific platform implementation
- Make the API as lean as possible, no specific use cases are supported which could also be layered on top of the API

- Offer a “comfort layer” to enable the use of C++11/14 features
- Support the integration into safety relevant systems

Therefore the API of the Crypto Stack follows a specific set of design decisions:

- It uses a pure virtual API to access different algorithms through a unified interface
- Its API has zero-copy capabilities delegating memory management to the caller
- A “comfort layer” provides functionality like asynchronous operation and memory management

7.2 Integration of Adaptive Application and Crypto Stack

The Adaptive Application should not have direct access to keys wihtin its own process. Therefore the Crypto Stack has to support features for isolating Adaptive Applications from the Crypto Stack implementation. The following separation mechanisms are envisioned:

1. Process isolation
2. Hardware isolation

The two mechanisms will be outlined briefly below.

7.2.1 Process isolation

The integrator of this specification may choose to isolate the cryptographic algorithm implementation from the `Adaptive Application`s by means of separating them into two different processes. The Crypto Stack implementation shall provide `ClientServerInterface`s according to the modeled `CryptoJob`s. The connection between the two pieces of software is implicitly made by the mapping of `CryptoNeed` onto `CryptoJob`. The communication interface is described by the `ClientServerInterface`. The interface visible to the `Adaptive Application` developer is specified in section 8.1.2.1. The actual IPC protocol is specific per platform implementation.

7.2.2 Hardware isolation

The integrator of this specification may choose to isolate the cryptographic algorithm implementation from the `Adaptive Application`s by means of separating them using a hardware mechanism (e.g. HSM, TPM). The Crypto Stack implementation shall provide `CryptoJob`s that are implemented in the selected hardware. The connection between the software and hardware is made by the mapping of `CryptoNeed` onto `CryptoJob`s that can be identifier in the hardware or its driver. The interface visible to the `Adaptive Application` developer is specified in section 8.1.2.1. The actual implementation of the driver is specific per platform implementation.

7.3 Supported algorithms

At least the following cryptographic algorithms or primitives should be supported by the Crypto Stack:

- Random Number Generation
  - Deterministic Random Number Generator (DRNG)
  - True Random Number Generator (TRNG)
- Symmetric Encryption
  - AES
    - Key Length: 128 and 256 bits
    - Modes: CBC, GCM, CCM
- Asymmetric Encryption/Decryption and Signature Handling
  - RSA
    - Key Length: 2048, 3072 and 4096 bits
    - Padding: PKCS#1 v2.2
  - Curve25519/Ed25519
  - NIST curves P256, P384 and P521 / ECDSA
- Hash
  - SHA-2
    - Length: 256, 384 and 512 bits
  - SHA-3
    - Length: 256, 384 and 512 bits
- MAC
  - CMAC
  - GMAC
  - HMAC
- Key Exchange
  - Diffie-Hellman

  - ECDH

The Crypto Stack may support handling the following cryptographic objects:

- Certificate Management
  - Handling of X.509 Certificates
  - Im/Export in DER format
  - Creation of CSRs

8 API specification

The API supports a streaming interface and a single call interface. Selected interfaces therefore provide the following methods:

- `Start`
- `Update`
- `Finish`
- `Process`

The `Start` method resets the internal states of the algorithm to begin processing chunks of data. The `Update` method updates the internal state of the algorithm by processing the given chunk of data and, if feasible, returning a transformed data chunk. The `Finish` method concludes the operation cycle of the algorithm by returning the final transformation.
The `Process` method can be used if all data is available at once and shall be processed in a single call. The `Process` method may internally call `Start`, `Update` and `Finish`.

8.1 C++ language binding

8.1.1 API Header files

This chapter describes the header files of the ara::sec::crypto API. The input for the header files are AUTOSAR Adaptive meta model classes within the `CryptoNeed` description, as defined in the AUTOSAR Manifest Specification [3].

The following requirements are applicable for all header files.

[SWS_CRYPTO_00001] No memory allocation in header files ⌈ The header files shall not contain code that creates objects on the heap. ⌋*(RS_CRYPTO_02301)*

[SWS_CRYPTO_00002] Folder structure ⌈ The *CryptoNeed header files* defined by [SWS_CRYPTO_01001] and the *Common header file* defined by [SWS_CRYPTO_01101], [SWS_CRYPTO_01103], [SWS_CRYPTO_01104], [SWS_CRYPTO_01105], [SWS_CRYPTO_01106], [SWS_CRYPTO_01107], [SWS_CRYPTO_01108], [SWS_CRYPTO_01109], [SWS_CRYPTO_01110], [SWS_CRYPTO_01111], [SWS_CRYPTO_01112] and [SWS_CRYPTO_01113] shall be located within the folder:

`<folder>/`

where:
`<folder>` is the start folder for the ara::sec::crypto header files specific for a project or platform vendor. ⌋*(RS_CRYPTO_02301)*

8.1.1.1 CryptoNeed header files

The *CryptoNeed header files* are the central definitions of the ara::sec::crypto API that are required to perform cryptographic operations.

[SWS_CRYPTO_01001] CryptoNeed header files existence ⌈ The Crypto Stack shall provide one *CryptoNeed header file* for each `CryptoNeed` defined in the modeling and mapped onto the Adaptive Application's `PortPrototype`'s. The file name for the *CryptoNeed header file* shall be `<name>_cryptoneed.h`, where `<name>` is the `CryptoNeed.shortName` converted to lower-case letters. ⌋*(RS_CRYPTO_02301)*

8.1.1.2 Common header files

The *Common header files* are central definitions of the ara::sec API that are required to describe the API of the Crypto Stack.

[SWS_CRYPTO_01101] Existence of Span header file ⌈ The Crypto Stack shall provide the *Common header file* `span.h`. The file shall be located in the start folder for the ara::sec header files specific for a project or platform vendor. ⌋*(RS_CRYPTO_02301)*

The `Span` defines a container class that has non-owning properties while still offering the benefits of iterators and size information. The API is shown in the appendix' section B.

[SWS_CRYPTO_01102] An ara::sec::Span header file shall be provided by the implementation ⌈ The Crypto Stack shall provide a `ara::sec::Span` implementation in the *Common header file* defined in [SWS_CRYPTO_01101]. ⌋*(RS_CRYPTO_02301)*

The Crypto Stack supports standardized access to selected primitives. The standardized access is ensured by the definition of pure virtual C++ interfaces as part of the *Common header files*. The API for these interfaces is shown in section 8.1.2.

[SWS_CRYPTO_01103] A cipher interface header file shall be provided by the implementation ⌈ The Crypto Stack shall provide the *Common header file* `cipher.h`. The file shall be located in the start folder for the ara::sec::crypto header files specific for a project or platform vendor. ⌋*(RS_CRYPTO_02301, RS_CRYPTO_02201, RS_CRYPTO_02202, RS_CRYPTO_02207)*

[SWS_CRYPTO_01104] A cipher parameters interface header file shall be provided by the implementation ⌈ The Crypto Stack shall provide the *Common header file* `cipher_parameters.h`. The file shall be located in the start folder for the ara::sec::crypto header files specific for a project or platform vendor. ⌋*(RS_CRYPTO_02301, RS_CRYPTO_02201, RS_CRYPTO_02202, RS_CRYPTO_02207, RS_CRYPTO_02203, RS_CRYPTO_02204)*

[SWS_CRYPTO_01105] A signer interface header file shall be provided by the implementation ⌈ The Crypto Stack shall provide the *Common header file* `signer.h`. The file shall be located in the start folder for the ara::sec::crypto header files specific for a project or platform vendor. ⌋*(RS_CRYPTO_02301, RS_CRYPTO_02203, RS_CRYPTO_02204)*

[SWS_CRYPTO_01106] A verifier interface header file shall be provided by the implementation ⌈ The Crypto Stack shall provide the *Common header file* `verifier.h`. The file shall be located in the start folder for the ara::sec::crypto header files specific for a project or platform vendor. ⌋*(RS_CRYPTO_02301, RS_CRYPTO_02203, RS_CRYPTO_02204)*

[SWS_CRYPTO_01107] A hash interface header file shall be provided by the implementation ⌈ The Crypto Stack shall provide the *Common header file* `hash.h`. The file shall be located in the start folder for the ara::sec::crypto header files specific for a project or platform vendor. ⌋*(RS_CRYPTO_02301, RS_CRYPTO_02205)*

[SWS_CRYPTO_01108] A random number generation interface header file shall be provided by the implementation ⌈ The Crypto Stack shall provide the *Common header file* `random.h`. The file shall be located in the start folder for the ara::sec::crypto header files specific for a project or platform vendor. ⌋*(RS_CRYPTO_02301, RS_CRYPTO_02206)*

[SWS_CRYPTO_01109] A key management interface header file shall be provided by the implementation ⌈ The Crypto Stack shall provide the *Common header file* `key_management.h`. The file shall be located in the start folder for the ara::sec::crypto header files specific for a project or platform vendor. ⌋*(RS_CRYPTO_02301, RS_CRYPTO_02105, RS_CRYPTO_02101)*

[SWS_CRYPTO_01110] A key derivation interface header file shall be provided by the implementation ⌈ The Crypto Stack shall provide the *Common header file* `key_derivation.h`. The file shall be located in the start folder for the ara::sec::crypto header files specific for a project or platform vendor. ⌋*(RS_CRYPTO_02301, RS_CRYPTO_02103)*

[SWS_CRYPTO_01111] A key exchange interface header file shall be provided by the implementation ⌈ The Crypto Stack shall provide the *Common header file* `key_exchange.h`. The file shall be located in the start folder for the ara::sec::crypto header files specific for a project or platform vendor. ⌋*(RS_CRYPTO_02301, RS_CRYPTO_02104)*

[SWS_CRYPTO_01112] A key interface header file shall be provided by the implementation ⌈ The Crypto Stack shall provide the *Common header file* `key.h`. The file shall be located in the start folder for the ara::sec::crypto header files specific for a project or platform vendor. ⌋*(RS_CRYPTO_02301)*

[SWS_CRYPTO_01113] A keyed primitive interface header file shall be provided by the implementation ⌈ The Crypto Stack shall provide the *Common header file* `keyed_primitive.h`. The file shall be located in the start folder for the ara::sec::crypto header files specific for a project or platform vendor. ⌋*()*

[SWS_CRYPTO_01114] Every header file shall include the ara::sec::Span header file ⌈ The *Common header files* shall include the *Common header file* defined in [SWS_CRYPTO_01101]:

```
#include "ara/sec/span.h"
```

⌋*(RS_CRYPTO_02301)*

8.1.2 API Reference

8.1.2.1 Common API

The *Common header files* have a static interface that is described in the *Common API*.

8.1.2.1.1 KeyedPrimitive interface

[SWS_CRYPTO_01201] Keyed primitive interface ⌈ The Crypto Stack shall provide a pure virtual interface for `KeyedPrimitive` located in the header file defined in [SWS_CRYPTO_01113] in the namespace `ara::sec::crypto`.

```
class KeyedPrimitive
```

⌋*(RS_CRYPTO_02301)*

[SWS_CRYPTO_01202] GetKey method ⌈ The Crypto Stack shall provide a method to retrieve a representation of a key associated with a primitive for the `KeyedPrimitive` interface.

```
virtual Key const& GetKey() const = 0;
```

⌋*(RS_CRYPTO_02301)*

The returned `Key` should not contain raw key data but should be a proxy to query information about the key or to use it in another primitive.

8.1.2.1.2 Cipher interface

[SWS_CRYPTO_01203] Cipher interface ⌈ The Crypto Stack shall provide an interface for `Cipher` derived from `KeyedPrimitive` located in the header file defined in [SWS_CRYPTO_01103] in the namespace `ara::sec::crypto`.

```
class Cipher : public KeyedPrimitive
```

⌋*(RS_CRYPTO_02201, RS_CRYPTO_02202, RS_CRYPTO_02207)*

The `Cipher` interface is employed for encrypting and decrypting data. The transformation may be done by a symmetric or asymmetric algorithm. The `Cipher` interface supports a single call interface via the `Process` method or a streaming interface via the `Start`, `Update` and `Finish` methods.

[SWS_CRYPTO_01204] Start method ⌈ The Crypto Stack shall provide a method to start an operation cycle of the cipher. The start methods accepts an optional argument of the type `CipherParameters` to initialize the cipher if required.

```
virtual void Start(CipherParameters* parameters) = 0;
```

⌋*(RS_CRYPTO_02201, RS_CRYPTO_02202, RS_CRYPTO_02207, RS_CRYPTO_02302)* **[SWS_CRYPTO_01205] Update method** ⌈ The Crypto Stack shall provide a method to update an operation cycle of the cipher. The method allows transforming an arbitrary chunk of data by supplying the untransformed data in the `input` variable and the transformed data will be supplied in the `output` variable.

```
virtual void Update(ara::sec::Span<const uint8_t> input, ara::sec::Span
    <uint8_t> output) = 0;
```

⌋*(RS_CRYPTO_02201, RS_CRYPTO_02202, RS_CRYPTO_02207, RS_CRYPTO_02302)*

[SWS_CRYPTO_01206] Finish method ⌈ The Crypto Stack shall provide a method to finish an operation cycle of the cipher. The method closes the operation of the cipher and may return final bytes from the cipher operation in the variable `output`.

```
virtual void Finish(ara::sec::Span<uint8_t> output) = 0;
```

⌋*(RS_CRYPTO_02201, RS_CRYPTO_02202, RS_CRYPTO_02207, RS_CRYPTO_02302)*

[SWS_CRYPTO_01207] Process method ⌈ The Crypto Stack shall provide a method to perform an operation cycle of the cipher in a single call. The `Process` method uses the same parameters with identical semantics as described for the streaming interface in [SWS_CRYPTO_01204], [SWS_CRYPTO_01205] and [SWS_CRYPTO_01206].

```
virtual void Process(CipherParameters* parameters, ara::sec::Span<const
     uint8_t> input, ara::sec::Span<uint8_t> output) = 0;
```

⌋*(RS_CRYPTO_02201, RS_CRYPTO_02202, RS_CRYPTO_02207)*

8.1.2.1.3 CipherParameters interface

[SWS_CRYPTO_01208] CipherParameters interface ⌈ The Crypto Stack shall provide an interface for `CipherParameters` located in the header file defined in [SWS_CRYPTO_01104] in the namespace `ara::sec::crypto`.

```
class CipherParameters
```

⌋*(RS_CRYPTO_02401)*

[SWS_CRYPTO_01209] GetFlags method ⌈ The Crypto Stack shall provide a method to query the flags for optimizing the algorithms operation. The returned value may be used by the implementation to select implementation strategies.

```
virtual AlgorithmFlags GetFlags() const = 0;
```

⌋*(RS_CRYPTO_02401)*

[SWS_CRYPTO_01210] GetNonceIV method ⌈ The Crypto Stack shall provide a method to retrieve a nonce or initialization vector for an algorithm to use.

```
virtual ara::sec::Span<uint8_t> const& GetNonceIV() const = 0;
```

⌋*(RS_CRYPTO_02401)*

[SWS_CRYPTO_01232] AlgorithmFlags enumeration ⌈ The Crypto Stack shall provide an enumeration of algorithm flags. The following flags shall be supported:

```
enum class AlgorithmFlags
{
  None,
  Latency,
  Background
};
```

The `AlgorithmFlags` values' semantics are described in table 8.1. ⌋*(RS_CRYPTO_02402)*

AlgorithmFlag	Explanation
`None`	No optimization preferences.
`Latency`	Optimize for latency.
`Background`	Optimize for processing in the background.

Table 8.1: AlgorithmFlags

8.1.2.1.4 Hash interface

[SWS_CRYPTO_01211] Hash interface ⌈ The Crypto Stack shall provide an interface for `Hash` located in the header file defined in [SWS_CRYPTO_01107] in the namespace `ara::sec::crypto`.

```
class Hash
```

⌋*(RS_CRYPTO_02205)*

[SWS_CRYPTO_01212] GetDigestSize method ⌈ The Crypto Stack shall provide a method to query the size of the produced digest. The size shall be returned in bytes.

```
virtual AlgorithmFlags GetFlags() const = 0;
```

⌋*(RS_CRYPTO_02205)*

[SWS_CRYPTO_01213] Start method ⌈ The Crypto Stack shall provide a method to start an operation cycle of the hash.

```
virtual void Start() = 0;
```

⌋*(RS_CRYPTO_02205, RS_CRYPTO_02302)*

[SWS_CRYPTO_01214] Update method ⌈ The Crypto Stack shall provide a method to update an operation cycle of the hash. The method allows transforming an arbitrary chunk of data by supplying the untransformed data in the `input` variable.

```
virtual void Update(ara::sec::Span<const uint8_t> input) = 0;
```

⌋*(RS_CRYPTO_02205, RS_CRYPTO_02302)*

[SWS_CRYPTO_01215] Finish method ⌈ The Crypto Stack shall provide a method to finish an operation cycle of the hash. The method closes the operation of the hash and return the digest in the variable `digest`.

```
virtual void Finish(ara::sec::Span<uint8_t> digest) = 0;
```

⌋*(RS_CRYPTO_02205, RS_CRYPTO_02302)*

[SWS_CRYPTO_01216] Process method ⌈ The Crypto Stack shall provide a method to perform an operation cycle of the hash in a single call. The `Process` method uses the same parameters with identical semantics as described for the streaming interface in [SWS_CRYPTO_01214] and [SWS_CRYPTO_01215].

```
virtual void Process(ara::sec::Span<const uint8_t> input, ara::sec::
    Span<uint8_t> digest) = 0;
```

⌋*(RS_CRYPTO_02205)*

8.1.2.1.5 Signer interface

[SWS_CRYPTO_01217] Signer interface ⌈ The Crypto Stack shall provide an interface for `Signer` derived from `KeyedPrimitive` located in the header file defined in [SWS_CRYPTO_01105] in the namespace `ara::sec::crypto`.

```
class Signer : public KeyedPrimitive
```

⌋*(RS_CRYPTO_02203, RS_CRYPTO_02204)*

The `Signer` interface is employed for signing data. The signature may be created using symmetric or asymmetric algorithms. The `Signer` interface supports a single call interface via the `Process` method or a streaming interface via the `Start`, `Update` and `Finish` methods.

[SWS_CRYPTO_01222] GetTagSize method ⌈ The Crypto Stack shall provide a method to query the size of the produced signature. The size shall be reported in bytes.

```
virtual std::size_t GetTagSize() const = 0;
```

⌋(RS_CRYPTO_02203, RS_CRYPTO_02204)

[SWS_CRYPTO_01218] Start method ⌈ The Crypto Stack shall provide a method to start an operation cycle of the signer. The start methods accepts an optional argument of the type `CipherParameters` to initialize the signer if required.

```
virtual void Start(CipherParameters* parameters) = 0;
```

⌋(RS_CRYPTO_02203, RS_CRYPTO_02204, RS_CRYPTO_02302)
[SWS_CRYPTO_01219] Update method ⌈ The Crypto Stack shall provide a method to update an operation cycle of the signer. The method allows transforming an arbitrary chunk of data by supplying the untransformed data in the `input` variable.

```
virtual void Update(ara::sec::Span<const uint8_t> input) = 0;
```

⌋(RS_CRYPTO_02203, RS_CRYPTO_02204, RS_CRYPTO_02302)

[SWS_CRYPTO_01220] Finish method ⌈ The Crypto Stack shall provide a method to finish an operation cycle of the signer. The method closes the operation of the signer and returns the signature bytes in the variable `output`.

```
virtual void Finish(ara::sec::Span<uint8_t> output) = 0;
```

⌋(RS_CRYPTO_02203, RS_CRYPTO_02204, RS_CRYPTO_02302)

[SWS_CRYPTO_01221] Process method ⌈ The Crypto Stack shall provide a method to perform an operation cycle of the cipher in a single call. The `Process` method uses the same parameters with identical semantics as described for the streaming interface in [SWS_CRYPTO_01218], [SWS_CRYPTO_01219] and [SWS_CRYPTO_01220].

```
virtual void Process(CipherParameters* parameters, ara::sec::Span<const
    uint8_t> input, ara::sec::Span<uint8_t> output) = 0;
```

⌋(RS_CRYPTO_02203, RS_CRYPTO_02204)

8.1.2.1.6 Verifier interface

[SWS_CRYPTO_01223] Verifier interface ⌈ The Crypto Stack shall provide an interface for `Verifier` derived from `KeyedPrimitive` located in the header file defined in [SWS_CRYPTO_01106] in the namespace `ara::sec::crypto`.

```
class Verifier : public KeyedPrimitive
```

⌋(RS_CRYPTO_02203, RS_CRYPTO_02204)

The `Verifier` interface is employed for verifying the authenticity data. The signature may be created using symmetric or asymmetric algorithms. The `Verifier` interface supports a single call interface via the `Process` method or a streaming interface via the `Start`, `Update` and `Finish` methods.

[SWS_CRYPTO_01225] Start method ⌈ The Crypto Stack shall provide a method to start an operation cycle of the verifier. The start methods accepts an optional argument of the type `CipherParameters` to initialize the verifier if required.

```
virtual void Start(CipherParameters* parameters) = 0;
```

⌋*(RS_CRYPTO_02203, RS_CRYPTO_02204, RS_CRYPTO_02302)*

[SWS_CRYPTO_01226] Update method ⌈ The Crypto Stack shall provide a method to update an operation cycle of the verifier. The method allows transforming an arbitrary chunk of data by supplying the untransformed data in the `input` variable.

```
virtual void Update(ara::sec::Span<const uint8_t> input) = 0;
```

⌋*(RS_CRYPTO_02203, RS_CRYPTO_02204, RS_CRYPTO_02302)*

[SWS_CRYPTO_01227] Finish method ⌈ The Crypto Stack shall provide a method to finish an operation cycle of the verifier. The method closes the operation of the verifier and performs the verification. The original authenticator is given in the variable `authenticator` and will be compared against the computed one. The optional argument `length` describes the amount of leftmost bits that shall be considered in the comparison, it is therefore given in bits. If its not present the entire authenticator is relevant.
If the relevant bits of the given `authenticator` match the computed authenticator `true` is returned, `false` otherwise.

```
virtual bool Finish(ara::sec::Span<const uint8_t> authenticator, std::
    size_t length = 0) = 0;
```

⌋*(RS_CRYPTO_02203, RS_CRYPTO_02204, RS_CRYPTO_02302)*

[SWS_CRYPTO_01228] Process method ⌈ The Crypto Stack shall provide a method to perform an operation cycle of the cipher in a single call. The `Process` method uses the same parameters with identical semantics and the same return values semantics as described for the streaming interface in [SWS_CRYPTO_01225], [SWS_CRYPTO_01226] and [SWS_CRYPTO_01227].

```
virtual bool Process(CipherParameters* parameters, ara::sec::Span<const
     uint8_t> input, ara::sec::Span<const uint8_t> authenticator, std::
    size_t length = 0) = 0;
```

⌋*(RS_CRYPTO_02203, RS_CRYPTO_02204)*

8.1.2.1.7 Random number generation interface

[SWS_CRYPTO_01229] Random interface ⌈ The Crypto Stack shall provide an interface for `Random` located in the header file defined in [SWS_CRYPTO_01108] in the namespace `ara::sec::crypto`.

```
class Random
```

⌋*(RS_CRYPTO_02206)*

[SWS_CRYPTO_01230] Seed method ⌈ The Crypto Stack shall provide a method to add additional random data to increase entropy of the random number generator. The additional entropy can be provided in the variable `input`.

```
virtual void Seed(ara::sec::Span const& input);
```

⌋(*RS_CRYPTO_02206*)

[SWS_CRYPTO_01231] Generate method ⌈ The Crypto Stack shall provide a method to generate random data. The random data will be place in the variable `output`.

```
virtual void Generate(ara::sec::Span<uint8_t> output) = 0;
```

⌋(*RS_CRYPTO_02206*)

8.1.2.1.8 Key interface

[SWS_CRYPTO_01233] Key interface ⌈ The Crypto Stack shall provide an interface for `Key` located in the header file defined in [SWS_CRYPTO_01112] in the namespace `ara::sec::crypto`.

```
class Key
```

⌋(*RS_CRYPTO_02001, RS_CRYPTO_02002*)

[SWS_CRYPTO_01234] GetId method ⌈ The Crypto Stack shall provide a method to query the unique identification of the key.

```
virtual uint32_t GetId() const = 0;
```

⌋(*RS_CRYPTO_02001, RS_CRYPTO_02002*)

[SWS_CRYPTO_01235] GetUsage method ⌈ The Crypto Stack shall provide a method to query the allowed usages of the key. See table 8.2 for more details.

```
virtual KeyUsage GetUsage() const = 0;
```

⌋(*RS_CRYPTO_02001, RS_CRYPTO_02002, RS_CRYPTO_02102*)

[SWS_CRYPTO_01236] GetProtection method ⌈ The Crypto Stack shall provide a method to query the protection flags of the key. See table 8.3 for more details.

```
virtual KeyProtection GetProtection() const = 0;
```

⌋(*RS_CRYPTO_02001, RS_CRYPTO_02002, RS_CRYPTO_02404*)

[SWS_CRYPTO_01237] GetSize method ⌈ The Crypto Stack shall provide a method to query the size of the key. The size shall be returned in bits.

```
virtual std::size_t GetSize() const = 0;
```

⌋(*RS_CRYPTO_02001, RS_CRYPTO_02002*)

[SWS_CRYPTO_01238] GetType method ⌈ The Crypto Stack shall provide a method to query the type of the key. See table 8.4 for more details.

```
virtual KeyType GetType() const = 0;
```

⌋*(RS_CRYPTO_02001, RS_CRYPTO_02002, RS_CRYPTO_02102)*

[SWS_CRYPTO_01239] KeyUsage enumeration ⌈The Crypto Stack shall provide an enumeration of key usage flags. The following flags shall be supported:

```
#define CKI_BIT(n) (1 << (n))

enum class KeyUsage : uint32_t
{
  Encrypt     = CKI_BIT(1),
  Decrypt     = CKI_BIT(2),
  Sign        = CKI_BIT(3),
  Verify      = CKI_BIT(4),
  Exchange    = CKI_BIT(5),
  Derive      = CKI_BIT(6),
  Provision   = CKI_BIT(7),
  Migration   = CKI_BIT(8)
};
```

The `KeyUsage` values' semantics are described in table 8.2. ⌋*(RS_CRYPTO_02001, RS_CRYPTO_02002, RS_CRYPTO_02102)*

KeyUsage	**Explanation**
`Encrypt`	The key may be used for encryption.
`Decrypt`	The key may be used for decryption.
`Sign`	The key may be used for signing.
`Verify`	The key may be used for verification.
`Exchange`	The key may be used for key exchange.
`Derive`	The key may be used as a base key for deriving other keys.
`Provision`	The key may be used for unwrapping keys during key provisioning.
`Migration`	The key may be used for exporting keys for migration.

Table 8.2: AlgorithmFlags

[SWS_CRYPTO_01240] KeyProtection enumeration ⌈The Crypto Stack shall provide an enumeration of key protection flags. The following flags shall be supported:

```
#define CKI_BIT(n) (1 << (n))

enum class KeyProtection : uint8_t
{
  External    = CKI_BIT(1),
  Exportable  = CKI_BIT(2),
  Importable  = CKI_BIT(3),
  Unprotected = CKI_BIT(4)
};
```

The `KeyProtection` values' semantics are described in table 8.3. ⌋*(RS_CRYPTO_02001, RS_CRYPTO_02002, RS_CRYPTO_02404)*

KeyProtection	**Explanation**

External	The key may be stored externally (e.g. outside of the HSM).
Exportable	The key may be exported.
Importable	The key may be imported.
Unprotected	The key may be handled without protection (i.e. plaintext).

Table 8.3: KeyProtection

[SWS_CRYPTO_01241] KeyType enumeration ⌈ The Crypto Stack shall provide an enumeration of key type flags. The following flags shall be supported:

```
enum class KeyType : uint8_t
{
  Symmetric,
  RSA,
  DH,
  Ecc_NISTp256,
  Ecc_NISTp384,
  Ecc_NISTp521,
  Ecc_Ed25519,
  Ecc_X25519,
  Ecc_Ed448,
  Ecc_X448
};
```

The `KeyType` values' semantics are described in table 8.4. ⌋*(RS_CRYPTO_02001, RS_CRYPTO_02002, RS_CRYPTO_02102)*

KeyType	Explanation
Symmetric	The key is usable for symmetric algorithms.
RSA	The key is usable for RSA operations.
DH	The key is usable for key exchange.
Ecc_NISTp256	The key is usable for elliptic curve cryptography.
Ecc_NISTp384	The key is usable for elliptic curve cryptography.
Ecc_NISTp521	The key is usable for elliptic curve cryptography.
Ecc_Ed25519	The key is usable for elliptic curve cryptography.
Ecc_X25519	The key is usable for elliptic curve cryptography.
Ecc_Ed448	The key is usable for elliptic curve cryptography.
Ecc_X448	The key is usable for elliptic curve cryptography.

Table 8.4: KeyType

8.1.2.1.9 KeyManagement interface

[SWS_CRYPTO_01242] KeyManagement interface ⌈ The Crypto Stack shall provide an interface for `KeyManagement` located in the header file defined in [SWS_CRYPTO_01109] in the namespace `ara::sec::crypto`.

```
class KeyManagement
```

⌋(RS_CRYPTO_02001, RS_CRYPTO_02002, RS_CRYPTO_02101, RS_CRYPTO_02102, RS_CRYPTO_02105)

[SWS_CRYPTO_01243] GetKey method ⌈ The Crypto Stack shall provide a method to query a `Key` interface by its identifier.

```
virtual Key const& GetKey(uint32_t id) = 0;
```

⌋(RS_CRYPTO_02403, RS_CRYPTO_02001, RS_CRYPTO_02002)

[SWS_CRYPTO_01244] Generate method ⌈ The Crypto Stack shall provide a method to generate data into a key. The key to generate data into is identified by the parameter `targetKey`. Optionally a random number generator to be used can be specified in the parameter `random`.

```
virtual void Generate(Key const& targetKey, Random* random = nullptr) =
    0;
```

⌋(RS_CRYPTO_02101)

[SWS_CRYPTO_01245] Verify method ⌈ The Crypto Stack shall provide a method to verify the validity of raw key data for a specific key. The key to verify the data for is identified by the parameter `key`. The data to be used for verification is supplied in the parameter `data`.

```
virtual bool Verify(Key const& key, ara::sec::Span const& data) const =
    0;
```

⌋(RS_CRYPTO_02102, RS_CRYPTO_02404)

[SWS_CRYPTO_01246] Import method ⌈ The Crypto Stack shall provide a method to import a key. The key to import the data for is identified by the parameter `targetKey`. The key to use for decrypting the protected key data is identified by the parameter `provisioningKey`. The protected data is supplied in the parameter `data`.

```
virtual void Import(Key const& targetKey, Key const& provisioningKey,
    ara::sec::Span const& data);
```

⌋(RS_CRYPTO_02102, RS_CRYPTO_02404, RS_CRYPTO_02105)

[SWS_CRYPTO_01247] Export method ⌈ The Crypto Stack shall provide a method to export a key. The key to export the data from is identified by the parameter `sourceKey`. The key to use for ecnrypting the protected key data is identified by the parameter `migrationKey`. The protected data is supplied in the output parameter `data`.

```
virtual void Export(Key const& sourceKey, Key const& migrationKey, ara
    ::sec::Span& data);
```

⌋(RS_CRYPTO_02105)

8.1.2.1.10 KeyDeriviation interface

[SWS_CRYPTO_01248] KeyDeriviation interface ⌈ The Crypto Stack shall provide an interface for `KeyDeriviation` located in the header file defined in [SWS_CRYPTO_01110] in the namespace `ara::sec::crypto`.

```
class KeyDeriviation
```

⌋*(RS_CRYPTO_02103)*

[SWS_CRYPTO_01249] Derive method ⌈ The Crypto Stack shall provide a method to derive key material from a base key. The base key is identified by the parameter `baseKey`. The target key is identified by the parameter `targetKey`. The label to be used for derivation is supplied in the parameter `label`. An optional context of arbitrary data may be supplied in the parameter `context`.

```
virtual void Derive(Key const& baseKey, Key const& targetKey, ara::sec
    ::Span& label, ara::sec::Span* context = nullptr) = 0;
```

⌋*(RS_CRYPTO_02103)*

8.1.2.1.11 KeyExchange interface

[SWS_CRYPTO_01250] KeyExchange interface ⌈ The Crypto Stack shall provide an interface for `KeyExchange` derived from `KeyedPrimitive` located in the header file defined in [SWS_CRYPTO_01111] in the namespace `ara::sec::crypto`.

```
class KeyExchange : KeyedPrimitive
```

⌋*(RS_CRYPTO_02104)*

[SWS_CRYPTO_01251] GetPublicValue method ⌈ The Crypto Stack shall provide a method to get a public value that must be sent to the other party for exchanging keys. The public value is provided in the parameter `pubValue`.

```
virtual void GetPublicValue(ara::sec::Span<uint8_t> pubValue) = 0;
```

⌋*(RS_CRYPTO_02104)*

[SWS_CRYPTO_01252] Exchange method ⌈ The Crypto Stack shall provide a method to execute the key exchange operation. The public value of the client is provided in the parameter `ourPubVal`. The public value of the other party is provided in the parameter `theirPubVal`. The computed shared key data will be supplied in the key identified by the parameter `sharedKey`.

```
virtual void Exchange(ara::sec::Span const& ourPubVal, ara::sec::Span
    const& theirPubVal, Key const& sharedKey) = 0;
```

⌋*(RS_CRYPTO_02104)*

[SWS_CRYPTO_01312] Primitive Factory ⌈ The Crypto Stack shall provide a factory class for each primitive class to create the actual crypto primitives.

```
class HashFactory {
public:
  //this could also be defined with vendor specific deleter
  using HashPtr = std::unique_ptr<Hash>;

  HashPtr CreateHash(const std::string& hashId);
}
```

⌋*()*

8.1.2.2 CryptoNeed API

The CryptoNeed description is the input for the generation of the *CryptoNeed header files* content. The *CryptoNeed header files* contain classes representing the CryptoNeed and ClientServerInterface referenced in the role requiredInterface. The interface of the class is defined by the ClientServerInterface which in turn is influenced by the associated CryptoNeed.

[SWS_CRYPTO_01301] CryptoNeed class ⌈ The Crypto Stack shall provide the definition of a C++ class named <name>CryptoNeed in the *CryptoNeed header file* defined by [SWS_CRYPTO_01001], where name is the CryptoNeed.shortName.

```
class <CryptoNeed.shortName>CryptoNeed {
...
}
```

⌋*(RS_CRYPTO_02301)*

[SWS_CRYPTO_01302] CryptoNeed class base type ⌈ The CryptoNeed class defined in [SWS_CRYPTO_01201] shall have different base types with regard to the value in CryptoNeed.primitiveFamily.

```
#include "ara/sec/crypto/<name>.h"

class <CryptoNeed.shortName>CryptoNeed : public <baseType> {
...
}
```

The <baseType>s to be used are listed in Table 8.5. ⌋*(RS_CRYPTO_02301)*

primitiveFamily value	<baseType> value	<name> value
ASYMMETRIC_ENCRYPT	Cipher	cipher.h
ASYMMETRIC_DECRYPT	Cipher	cipher.h
SYMMETRIC_ENCRYPT	Cipher	cipher.h
SYMMETRIC_DECRYPT	Cipher	cipher.h
AEAD_ENCRYPT	Cipher	cipher.h
AEAD_DECRYPT	Cipher	cipher.h
SIGNATURE_GENERATE	Signer	signer.h
SIGNATURE_VERIFY	Verifier	verifier.h
MAC_GENERATE	Signer	signer.h
MAC_VERIFY	Verifier	verifier.h
HASH	Hash	hash.h

RANDOM	Random	random.h
KEY_DERIVE	KeyDerivation	key_derivation.h
KEY_EXCHANGE	KeyExchange	key_exchange.h
KEY_MANAGEMENT	KeyManagement	key_management.h

Table 8.5: CryptoNeed.primitiveFamily supported for interfaces

The concrete CryptoNeed class may be created by the Adaptive Platform's "ARA::CRYPTO" implementation (e.g. a factory or factory method). This implementation is responsible for creating the correct binding for a software or hardware isolation mechanism (see section 7.2).

8.2 Client Server Interfaces

This chapter lists the ClientServerInterfaces that are used to access the cryptographic implementation from an Adaptive Application.

[SWS_CRYPTO_01303] Port and ClientServerInterface for CryptoNeeds typed as a Cipher ⌈ If the CryptoNeed class defined in [SWS_CRYPTO_01302] has the base type Cipher its RPortPrototype shall reference a ClientServerInterface designed as follows:

Name	Cipher_{CryptoNeed.name}_{PrimitiveFamiliy}		
Kind	RequiredPort	***Interface***	ClientServerInterface
Description	Requires operations to encipher or decipher data.		
Variation	Defined by the name of the CryptoNeed and the primitive family.		

Table 8.6: Port - Cipher_{CryptoNeed.name}_{PrimitiveFamiliy}

Name	Start	
Description	Starts a streaming context of the cipher.	
Parameter	parameters	
	Description	The parameters for initializing the cipher's operation cycle.
	Type	CipherParameters: Contains information on preferred algorithm strategy and additional inizialization parameters.
	Variation	-
	Direction	IN

Table 8.7: ClientServerInterface Cipher - Method: Start

Name	Update	
Description	Updates the streaming context of the cipher and hence transforms a chunk of data.	
Parameter	input	
	Description	The input data to be transformed by the cipher.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	IN

Parameter	output	
	Description	The data that has been transformed by the cipher.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	OUT

Table 8.8: ClientServerInterface Cipher - Method: Update

Name	Finish	
Description	Finishes a streaming context of the cipher and retrieves the remaining data.	
Parameter	output	
	Description	The data that has been transformed by the cipher.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	OUT

Table 8.9: ClientServerInterface Cipher - Method: Finish

Name	Process	
Description	Single call interface to transform (encipher, decipher) data.	
Parameter	parameters	
	Description	The parameters for initializing the cipher's operation cycle.
	Type	CipherParameters: Contains information on preferred algorithm strategy and additional inizialization parameters.
	Variation	-
	Direction	IN
Parameter	input	
	Description	The input data to be transformed by the cipher.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	IN
Parameter	output	
	Description	The data that has been transformed by the cipher.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	OUT

Table 8.10: ClientServerInterface Cipher - Method: Process

⌋*(RS_CRYPTO_02401)*

[SWS_CRYPTO_01304] Port and ClientServerInterface for CryptoNeeds typed as a Signer ⌈ If the CryptoNeed class defined in [SWS_CRYPTO_01302] has the base type `Signer` its `RPortPrototype` shall reference a `ClientServerInterface` designed as follows:

Name	Signer_{CryptoNeed.name}_{PrimitiveFamiliy}		
Kind	RequiredPort	***Interface***	ClientServerInterface
Description	Requires operations to sign data.		

Variation	Defined by the name of the CryptoNeed and the primitive family.

Table 8.11: Port - Signer_{CryptoNeed.name}_{PrimitiveFamiliy}

Name	GetTagSize	
Description	Gets the size of the tag produced by the signer.	
Parameter	tagSize	
	Description	The size of the produced tag in bytes.
	Type	uint32
	Variation	-
	Direction	OUT

Table 8.12: ClientServerInterface Signer - Method: GetTagSize

Name	Start	
Description	Starts a streaming context of the signer.	
Parameter	parameters	
	Description	The parameters for initializing the signer's operation cycle.
	Type	CipherParameters: Contains information on preferred algorithm strategy and additional inizialization parameters.
	Variation	-
	Direction	IN

Table 8.13: ClientServerInterface Signer - Method: Start

Name	Update	
Description	Updates the streaming context of the signer and hence transforms a chunk of data.	
Parameter	input	
	Description	The input data to be processed by the signer.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	IN

Table 8.14: ClientServerInterface Signer - Method: Update

Name	Finish	
Description	Finishes a streaming context of the signer and retrieves the digital signature/authentication tag.	
Parameter	output	
	Description	The digital signature/authentication tag.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	OUT

Table 8.15: ClientServerInterface Signer - Method: Finish

Name	Process
Description	Single call interface to sign data.

Parameter	parameters	
	Description	The parameters for initializing the signers's operation cycle.
	Type	CipherParameters: Contains information on preferred algorithm strategy and additional inizialization parameters.
	Variation	-
	Direction	IN
Parameter	input	
	Description	The input data to be processed by the signer.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	IN
Parameter	output	
	Description	The digital signature/authentication tag.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	OUT

Table 8.16: ClientServerInterface Signer - Method: Process

⌋*(RS_CRYPTO_02401)*

[SWS_CRYPTO_01305] Port and ClientServerInterface for CryptoNeeds typed as a Verifier ⌈ If the CryptoNeed class defined in [SWS_CRYPTO_01302] has the base type `Verifier` its `RPortPrototype` shall reference a `ClientServerInterface` designed as follows:

Name	Verifier_{CryptoNeed.name}_{PrimitiveFamiliy}		
Kind	RequiredPort	**Interface**	ClientServerInterface
Description	Requires operations to verify data.		
Variation	Defined by the name of the CryptoNeed and the primitive family.		

Table 8.17: Port - Verifier_{CryptoNeed.name}_{PrimitiveFamiliy}

Name	Start	
Description	Starts a streaming context of the verifier.	
Parameter	parameters	
	Description	The parameters for initializing the verifier's operation cycle.
	Type	CipherParameters: Contains information on preferred algorithm strategy and additional inizialization parameters.
	Variation	-
	Direction	IN

Table 8.18: ClientServerInterface Verifier - Method: Start

Name	Update
Description	Updates the streaming context of the verifier and hence transforms a chunk of data.

Parameter	input	
	Description	The input data to be processed by the verifier.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	IN

Table 8.19: ClientServerInterface Verifier - Method: Update

Name	Finish	
Description	Finishes a streaming context of the verifier and performs the verification of the digital signature/authentication tag.	
Parameter	authenticator	
	Description	The digital signature/authentication tag to verify against.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	IN
Parameter	length	
	Description	The length in bits of the digital signature/authentication tag to verify the leftmost against.
	Type	uint32
	Variation	-
	Direction	IN
Parameter	result	
	Description	The result of the comparison.
	Type	boolean
	Variation	-
	Direction	OUT

Table 8.20: ClientServerInterface Verifier - Method: Finish

Name	Process	
Description	Single call interface to verify data.	
Parameter	parameters	
	Description	The parameters for initializing the verifier's operation cycle.
	Type	CipherParameters: Contains information on preferred algorithm strategy and additional inizialization parameters.
	Variation	-
	Direction	IN
Parameter	authenticator	
	Description	The digital signature/authentication tag to verify against.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	IN
Parameter	length	
	Description	The length in bits of the digital signature/authentication tag to verify the leftmost against.
	Type	uint32
	Variation	-
	Direction	IN

Parameter	result	
	Description	The result of the comparison.
	Type	boolean
	Variation	-
	Direction	OUT

Table 8.21: ClientServerInterface Verifier - Method: Process

⌋(*RS_CRYPTO_02401*)

[SWS_CRYPTO_01306] Port and ClientServerInterface for CryptoNeeds typed as a Random ⌈ If the CryptoNeed class defined in [SWS_CRYPTO_01302] has the base type `Random` its `RPortPrototype` shall reference a `ClientServerInterface` designed as follows:

Name	Random_{CryptoNeed.name}_{PrimitiveFamiliy}		
Kind	RequiredPort	***Interface***	ClientServerInterface
Description	Requires operations to generate random data.		
Variation	Defined by the name of the CryptoNeed and the primitive family.		

Table 8.22: Port - Random_{CryptoNeed.name}_{PrimitiveFamiliy}

Name	Seed	
Description	Provide additional seed.	
Parameter	seed	
	Description	The buffer to hold seed data.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	IN

Table 8.23: ClientServerInterface Random - Method: Seed

Name	Generate	
Description	Generate random data.	
Parameter	output	
	Description	The generate random data.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	out

Table 8.24: ClientServerInterface Random - Method: Generate

⌋(*RS_CRYPTO_02401*)

[SWS_CRYPTO_01307] Port and ClientServerInterface for CryptoNeeds typed as a Hash ⌈ If the CryptoNeed class defined in [SWS_CRYPTO_01302] has the base type `Hash` its `RPortPrototype` shall reference a `ClientServerInterface` designed as follows:

Name	Hash_{CryptoNeed.name}_{PrimitiveFamiliy}		
Kind	RequiredPort	***Interface***	ClientServerInterface
Description	Requires operations to generate hashes.		
Variation	Defined by the name of the CryptoNeed and the primitive family.		

Table 8.25: Port - Hash_{CryptoNeed.name}_{PrimitiveFamiliy}

Name	Start
Description	Starts a streaming context of the hash.

Table 8.26: ClientServerInterface Hash - Method: Start

Name	Update	
Description	Updates the streaming context of the hash and hence transforms a chunk of data.	
Parameter	input	
	Description	The input data to be transformed by the hash.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	IN

Table 8.27: ClientServerInterface Hash - Method: Update

Name	Finish	
Description	Finishes a streaming context of the hash and retrieves the digest.	
Parameter	digest	
	Description	The digest of the data.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	OUT

Table 8.28: ClientServerInterface Hash - Method: Finish

Name	Process	
Description	Single call operation for hashing.	
Parameter	input	
	Description	The input data to be transformed by the hash.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	IN
Parameter	digest	
	Description	The digest of the data.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	OUT

Table 8.29: ClientServerInterface Hash - Method: Process

⌋*(RS_CRYPTO_02401)*

[SWS_CRYPTO_01308] Port and ClientServerInterface for CryptoNeeds typed as a KeyDerivation ⌈ If the CryptoNeed class defined in [SWS_CRYPTO_01302] has the base type `KeyDerivation` its `RPortPrototype` shall reference a `ClientServerInterface` designed as follows:

Name	KeyDerivation_{CryptoNeed.name}_{PrimitiveFamiliy}		
Kind	RequiredPort	***Interface***	ClientServerInterface
Description	Requires operations to derive a key.		
Variation	Defined by the name of the CryptoNeed and the primitive family.		

Table 8.30: Port - KeyDerivation_{CryptoNeed.name}_{PrimitiveFamiliy}

Name	Derive	
Description	Perform a key derivation.	
Parameter	baseKey	
	Description	The key to derive from.
	Type	uint32
	Variation	-
	Direction	IN
Parameter	targetKey	
	Description	The key to be derived.
	Type	uint32
	Variation	-
	Direction	IN
Parameter	label	
	Description	The label to use for the derivation.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	IN
Parameter	context	
	Description	The context for the derivation.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	IN

Table 8.31: ClientServerInterface KeyDerivation - Method: Derive

⌋*(RS_CRYPTO_02401)*

[SWS_CRYPTO_01309] Port and ClientServerInterface for CryptoNeeds typed as a KeyExchange ⌈ If the CryptoNeed class defined in [SWS_CRYPTO_01302] has the base type `KeyExchange` its `RPortPrototype` shall reference a `ClientServerInterface` designed as follows:

Name	KeyExchange_{CryptoNeed.name}_{PrimitiveFamiliy}		
Kind	RequiredPort	***Interface***	ClientServerInterface
Description	Requires operations to exchange a shared key.		
Variation	Defined by the name of the CryptoNeed and the primitive family.		

Table 8.32: Port - KeyExchange_{CryptoNeed.name}_{PrimitiveFamiliy}

Name	GetPublicValue	
Description	Retrieve a public value to be sent to the other party.	
Parameter	pubValue	
	Description	Our public value to be sent to the other party.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	OUT

Table 8.33: ClientServerInterface KeyExchange - Method: GetPublicValue

Name	Exchange	
Description	Perform a key exchange.	
Parameter	ourPubVal	
	Description	Our public value sent to the other party.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	IN
Parameter	theirPubVal	
	Description	The public value received from by the other party.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	IN
Parameter	sharedKey	
	Description	The key handle into which to store the computed shared secret key.
	Type	uint32
	Variation	-
	Direction	IN

Table 8.34: ClientServerInterface KeyExchange - Method: Exchange

⌋*(RS_CRYPTO_02401)*

[SWS_CRYPTO_01310] Port and ClientServerInterface for CryptoNeeds typed as a KeyManagement ⌈ If the CryptoNeed class defined in [SWS_CRYPTO_01302] has the base type `KeyManagement` its `RPortPrototype` shall reference a `ClientServerInterface` designed as follows:

Name	KeyManagement		
Kind	RequiredPort	***Interface***	ClientServerInterface
Description	Requires operations to manage key.		

Table 8.35: Port - KeyManagement

Name	GetKey
Description	Retrieve a key handle for the given identification.

Parameter	keyId	
	Description	Key identification which is unique within the stack.
	Type	uint32
	Variation	-
	Direction	IN
Parameter	key	
	Description	The key handle.
	Type	uint32
	Variation	-
	Direction	OUT

Table 8.36: ClientServerInterface KeyManagement - Method: GetKey

Name	Generate	
Description	Generate key data.	
Parameter	targetKey	
	Description	The key handle to generate the key into.
	Type	uint32
	Variation	-
	Direction	IN
Parameter	random	
	Description	The random number generator to use for generating the key data.
	Type	uint32
	Variation	-
	Direction	IN

Table 8.37: ClientServerInterface KeyManagement - Method: Generate

Name	Verify	
Description	Verify the given key data.	
Parameter	key	
	Description	The key handle to check the data against.
	Type	uint32
	Variation	-
	Direction	IN
Parameter	data	
	Description	The data to check.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	IN
Parameter	result	
	Description	The result of the check.
	Type	boolean
	Variation	-
	Direction	OUT

Table 8.38: ClientServerInterface KeyManagement - Method: Verify

Name	Import	
Description	Import some protected key data into the key store.	
Parameter	targetKey	
	Description	The key handle to import the data into.
	Type	uint32
	Variation	-
	Direction	IN
Parameter	provisioningKey	
	Description	The key handle to use when decrypting the data to import.
	Type	uint32
	Variation	-
	Direction	IN
Parameter	data	
	Description	The data to import.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	IN

Table 8.39: ClientServerInterface KeyManagement - Method: Import

Name	Export	
Description	Export key data from the key handle.	
Parameter	sourceKey	
	Description	The key handle to export data from.
	Type	uint32
	Variation	-
	Direction	IN
Parameter	migrationKey	
	Description	The key handle to use when protecting the data to export.
	Type	uint32
	Variation	-
	Direction	IN
Parameter	data	
	Description	The exported key data.
	Type	Span: Contains arbitrary data of an arbitrary but fixed length.
	Variation	-
	Direction	OUT

Table 8.40: ClientServerInterface KeyManagement - Method: Export

⌋*(RS_CRYPTO_02401)*

[SWS_CRYPTO_01311] ClientServerInterface for Keys ⌈ The interfaces described in 8.1.2.1 which are derived from the `KeyedPrimitive` ([SWS_CRYPTO_01201]) and the interface `KeyManagement` ([SWS_CRYPTO_01242]) can return a `Key` handle. This handle cannot be modeled as a `RPortPrototype` to be used by the Adaptive Application. Still operations invoked on the returned `Key` shall behave as if they were modeled referencing a `ClientServerInterface` designed as follows:

Name	GetId	
Description	Get the identifier associated with this key.	
Parameter	keyId	
	Description	Key identification which is unique within the stack.
	Type	uint32
	Variation	-
	Direction	OUT

Table 8.41: ClientServerInterface Key - Method: GetId

Name	GetType	
Description	Return the type of the key.	
Parameter	type	
	Description	The key htype.
	Type	uint32
	Variation	-
	Direction	IN

Table 8.42: ClientServerInterface Key - Method: GetType

Name	GetSize	
Description	Return the key size in bits.	
Parameter	size	
	Description	The key size in bits.
	Type	uint32
	Variation	-
	Direction	OUT

Table 8.43: ClientServerInterface Key - Method: GetSize

Name	GetUsage	
Description	Return information on what this key may be used for.	
Parameter	usage	
	Description	The key's usage information.
	Type	uint32
	Variation	-
	Direction	out

Table 8.44: ClientServerInterface Key - Method: GetUsage

Name	GetProtection	
Description	Return information on what the restrictions for handling the key are.	
Parameter	protection	
	Description	The key protection flags.
	Type	uint32
	Variation	-
	Direction	OUT

Table 8.45: ClientServerInterface Key - Method: GetProtection

⌋*(RS_CRYPTO_02401)*

A Mentioned Class Tables

For the sake of completeness, this chapter contains a set of class tables representing meta-classes mentioned in the context of this document but which are not contained directly in the scope of describing specific meta model semantics.

Class	**ClientServerInterface**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::PortInterface			
Note	A client/server interface declares a number of operations that can be invoked on a server by a client. **Tags:** atp.recommendedPackage=PortInterfaces			
Base	ARElement, ARObject, AtpBlueprint, AtpBlueprintable, AtpClassifier, AtpType, CollectableElement, Identifiable, MultilanguageReferrable, PackageableElement, Port Interface, Referrable			
Attribute	***Type***	***Mul.***	***Kind***	***Note***
operation	ClientServerOp eration	1..*	aggr	ClientServerOperation(s) of this ClientServerInterface. **Stereotypes:** atpVariation **Tags:** vh.latestBindingTime=blueprintDerivation Time
possibleErr or	ApplicationError	*	aggr	Application errors that are defined as part of this interface.

Table A.1: ClientServerInterface

Class	**CryptoJob**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::Deployment::Crypto			
Note	This meta-class represents the ability to model a crypto job. The latter in turn represents a call to a specific routine that implements a crypto function and that uses a specific key and refers to a specific primitive as a formal representation of the crypto algorithm. **Tags:** atp.Status=draft			
Base	ARObject, Identifiable, MultilanguageReferrable, Referrable			
Attribute	***Type***	***Mul.***	***Kind***	***Note***
cryptoKey	CryptoKeySlot	0..1	ref	This represents the key slots to which the referencing crypto job applies. **Tags:** atp.Status=draft
primitive	CryptoPrimitive	1	aggr	This aggregation defines the crypto primitive applicable for the enclosing crypto job. **Tags:** atp.Status=draft

Table A.2: CryptoJob

Class	**CryptoNeed**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::ApplicationDesign::PortInterface			
Note	This meta-class represents a statement regarding the applicable crypto use case. **Tags:** atp.Status=draft; atp.recommendedPackage=CryptoNeeds			
Base	ARElement, ARObject, AtpBlueprint, AtpBlueprintable, CollectableElement, Identifiable, MultilanguageReferrable, PackageableElement, Referrable			
Attribute	***Type***	***Mul.***	***Kind***	***Note***
primitiveFamily	String	1	attr	This attribute represents the ability to specify the algorithm family of the crypto need. **Tags:** atp.Status=draft

Table A.3: CryptoNeed

Class	**PortPrototype (abstract)**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Components			
Note	Base class for the ports of an AUTOSAR software component. The aggregation of PortPrototypes is subject to variability with the purpose to support the conditional existence of ports.			
Base	ARObject, AtpBlueprintable, AtpFeature, AtpPrototype, Identifiable, Multilanguage Referrable, Referrable			
Attribute	***Type***	***Mul.***	***Kind***	***Note***
clientServerAnnotation	ClientServerAnnotation	*	aggr	Annotation of this PortPrototype with respect to client/server communication.
delegatedPortAnnotation	DelegatedPortAnnotation	0..1	aggr	Annotations on this delegated port.
ioHwAbstractionServerAnnotation	IoHwAbstractionServerAnnotation	*	aggr	Annotations on this IO Hardware Abstraction port.
modePortAnnotation	ModePortAnnotation	*	aggr	Annotations on this mode port.
nvDataPortAnnotation	NvDataPortAnnotation	*	aggr	Annotations on this non voilatile data port.
parameterPortAnnotation	ParameterPortAnnotation	*	aggr	Annotations on this parameter port.
portPrototypeProps	PortPrototypeProps	0..1	aggr	This attribute allows for the definition of further qualification of the semantics of a PortPrototype. **Tags:** atp.Status=draft
senderReceiverAnnotation	SenderReceiverAnnotation	*	aggr	Collection of annotations of this ports sender/receiver communication.
triggerPortAnnotation	TriggerPortAnnotation	*	aggr	Annotations on this trigger port.

Table A.4: PortPrototype

Class	**RPortPrototype**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Components			
Note	Component port requiring a certain port interface.			
Base	ARObject, AbstractRequiredPortPrototype, AtpBlueprintable, AtpFeature, Atp Prototype, Identifiable, MultilanguageReferrable, PortPrototype, Referrable			
Attribute	**Type**	**Mul.**	**Kind**	**Note**
requiredInt erface	PortInterface	1	tref	The interface that this port requires, i.e. the port depends on another port providing the specified interface. **Stereotypes:** isOfType

Table A.5: RPortPrototype

Class	**Referrable (abstract)**			
Package	M2::AUTOSARTemplates::GenericStructure::GeneralTemplateClasses::Identifiable			
Note	Instances of this class can be referred to by their identifier (while adhering to namespace borders).			
Base	ARObject			
Attribute	**Type**	**Mul.**	**Kind**	**Note**
shortName	Identifier	1	attr	This specifies an identifying shortName for the object. It needs to be unique within its context and is intended for humans but even more for technical reference. **Tags:** xml.enforceMinMultiplicity=true; xml.sequenceOffset=-100
shortName Fragment	ShortNameFrag ment	*	aggr	This specifies how the Referrable.shortName is composed of several shortNameFragments. **Tags:** xml.sequenceOffset=-90

Table A.6: Referrable

B Span

This section shall elaborate the concept of the `span` introduced in [SWS_CRYPTO_01102]. The listing below illustrates the interface of the span.

```
class span {
public:
  ~span() = default;

  // internal types (further types omitted for easy readability)
  using pointer = element_type*;
  using reference = element_type&;

  // range access
  /**
   * Retrieves the first \p count elements from the span.
```

```
     *
     * \param[in] count The number of elements to have in the subspan.
     * \return A subspan of \p count elements from the beginning.
    */
    constexpr span<element_type, dynamic_extent> first(index_type count)
        const;
    /**
     * Retrieves the last \p count elements from the span.
     *
     * \param[in] count The number of elements to have in the subspan.
     * \return A subspan of \p count elements from the end.
    */
    constexpr span<element_type, dynamic_extent> last(index_type count)
        const;

    /**
     * Retrieves a view on the span beginning from \p offset and
         containing \p count elemnents.
     *
     * \param[in] offset The index of the first element to be in the
         returned subspan.
     * \param[in] count The number of elements to have in the subspan.
     * \return A subspan of \p count elements from the element at \p
         offset.
    */
    constexpr span<element_type, dynamic_extent> subspan(index_type
        offset, index_type count = dynamic_extent) const;

    // size information
    /**
     * Return the number of elements in the span
     *
     * \return The number of elements.
    */
    constexpr index_type length() const;
    /**
     * Return the number of elements in the span
     *
     * \return The number of elements.
    */
    constexpr index_type size() const;
    /**
     * Return the number of bytes in the span.
     *
     * \return The number of bytes.
    */
    constexpr index_type length_bytes() const;
    /**
     * Return the number of bytes in the span.
     *
     * \return The number of bytes.
    */
    constexpr index_type size_bytes() const;

    /**
     * Query if there are elements in the span.
```

```
 *
 * \return False if there are elements in the span, true otherwise.
*/
constexpr bool empty() const;

// element access
/**
 * Access the element at \p idx.
 *
 * \param[in] idx The index where the element is located.
 * \return A reference to the element located an \p idx.
*/
constexpr reference at(index_type idx) const;
/**
 * Access the element at \p idx.
 *
 * \param[in] idx The index where the element is located.
 * \return A reference to the element located an \p idx.
*/
constexpr reference operator[](index_type idx) const;

// data access
/**
 * Access the data of the span directly.
 *
 * \return A pointer to the data managed by the span.
*/
constexpr pointer data() const;

// iterators
/**
 * Obtain an iterator pointing to the first element in the span.
 *
 * \return An iterator pointing to the first element in the span.
*/
iterator begin() const;
/**
 * Obtain an iterator pointing to the position after the last element
     in the span.
 *
 * \return An iterator pointing to the position after the last
    element in the span.
*/
iterator end() const;

/**
 * Obtain a constant iterator pointing to the first element in the
    span.
 *
 * \return A constant iterator pointing to the first element in the
    span.
*/
const_iterator cbegin() const;
/**
 * Obtain a constant iterator pointing to the position after the last
     element in the span.
     *
     * \return A constant iterator pointing to the position after the
        last element in the span.
    */
    const_iterator cend() const;

    /**
     * Obtain a reverse iterator pointing to the first element in the
        reversed span (i.e. the last element in the non-reversed span).
     *
     * \return A reverse iterator.
    */
    reverse_iterator rbegin() const;
    /**
     * Obtain a reverse iterator pointing to the position after the last
        element in the reversed span (i.e. before the fist element in
        the non-reversed span).
     *
     * \return A reverse iterator.
    */
    reverse_iterator rend() const;

    /**
     * Obtain a constant reverse iterator pointing to the first element
        in the reversed span (i.e. the last element in the non-reversed
        span).
     *
     * \return A constant reverse iterator.
    */
    const_reverse_iterator crbegin() const;
    /**
     * Obtain a constant reverse iterator pointing to the position after
        the last element in the reversed span (i.e. before the fist
        element in the non-reversed span).
     *
     * \return A constant reverse iterator.
    */
    const_reverse_iterator crend() const;
  }
```