Federal Office
for Information Security

Technical Guideline BSI TR-03151
Secure Element API (SE API)

Version 1.0.0
6. June 2018

Federal Office for Information Security
Post Box 20 03 63
D-53133 Bonn

E-Mail: registrierkassen@bsi.bund.de
Internet: https://www.bsi.bund.de

Table of Contents

Figures

Tables

1 Introduction

1.1 Motivation

In the course of digitalisation, many applications nowadays rely on digital signatures in order to protect the authenticity and integrity of information. Due to legal or organisational requirements, such applications need a certain level of protection of their signature keys, i.e. private signature keys must be protected from unauthorized usage and disclosure via a suitable secure component. Such a level of protection can be achieved by the usage of a Secure Element (SE).

This document specifies the *Secure Element API* (*SE API*). The SE API is a digital interface that wraps functionality of a Secure Element and allows access to security functionalities by an application in a unified way regardless of the specific type of Secure Element in use (see Figure 1).

The main purpose of this interface is to secure authenticity and integrity of information by creating digital signatures over them.

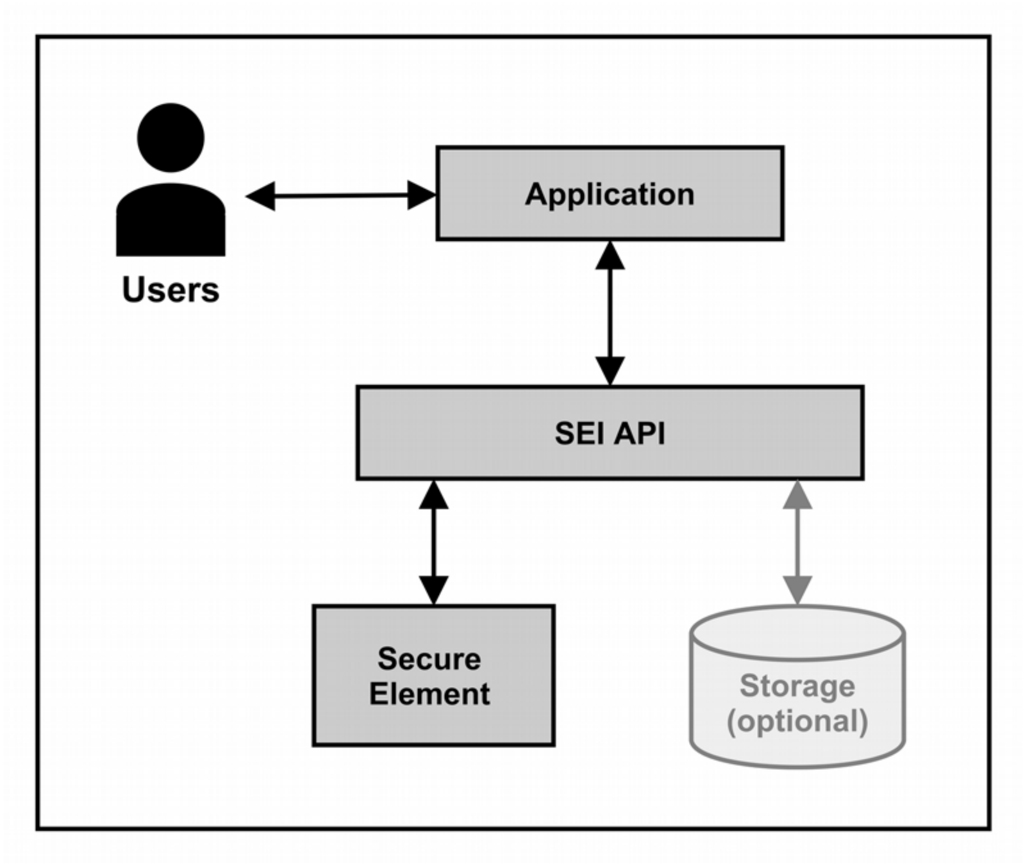

Figure 1: General system architecture in the context of the SE API

1.2 Purpose and general functionality of the SE API

A more specific purpose is the protection of externally provided process data with a unique transaction number and a time-stamp. Both additional parameters, time and counter, as well as the process data, are covered by the signature. The time-stamp provides information about point in time when the signature creation was performed. Furthermore, a signature counter that is increased with every signature allows to detect the absence of a signature can be easily detected.

The SE API works in the following way:

1. Process data flow into the SE API.
2. In the SE API, the process data are forwarded to the Secure Element.
3. If required, the Secure Element creates a new transaction number for the current transaction.
4. The Secure Element creates a signature over the process data and the additional self-provided data (a time-stamp, a signature counter and other information).
5. Then, the SE API composes and returns a log message; the Secure Element is invoked for parts of the log message. The log message contains the signature and all additional data, that was used in the signature creation process (and is needed to verify the signature).

The set of log messages can be used to verify the completeness of transactions. Furthermore, it is ensured that the contained process data was signed at a certain time (and has not been altered since) and that the signature creation happened in succession to a signature creation over data containing a smaller signature counter value.

1.3 Content and scope

This Technical Guideline focuses on the creation and structure of the log messages, their export, and the specification of the integration interfaces to the application.

The integration interface is defined in the OMG Interface Definition Language (IDL) - a generic interface description language. The OMG IDL [OMG2017a] is a text based, language independent definition language for interfaces.

The descriptions in this document are designed to be independent of any concrete implementation. Possible implementations of an API as described in this document include (but are not limited to) a SOAP-API or the direct exposition of a classical API from the programming language that the API is developed in.

Neither the physical interface, by which the SE API is exposed, nor any other layers in terms of the ISO/OSI reference model are defined by this Technical Guideline. This allows a maximum degree of flexibility for a Secure Element application developer.

The functions of the SE API are described regarding their function parameters, their behavior, and the exceptions. The interaction between the SE API and the Secure Element is only considered in an abstract, technologically independent way.

This document is accompanied by a ZIP archive [SPECZIP] that contains:

– the definitions of the SE API in form of its OMG IDL definition,
– the translation of the OMG IDL definitions to Java,
– the translation of the OMG IDL definitions to ANSI C.

1.4 Key words

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document SHALL be interpreted as described in [RFC2119].

The key word "CONDITIONAL" is to be interpreted as follows: The usage of an item is dependent on the usage of other items. It is therefore further qualified under which conditions the item is REQUIRED or RECOMMENDED.

When used in tables (profiles), the key words are abbreviated as shown in Table 1.

Key word		Abbrev.
MUST / SHALL	REQUIRED	m
MUST NOT / SHALL NOT	–	x
SHOULD	RECOMMENDED	r
MAY	OPTIONAL	o
–	CONDITIONAL	c

Table 1: Key words

2 Log messages and their creation

Log messages contain application or system data to be protected, protocol data that generated by a Secure Element during the logging process, and a signature protecting these two data structures.

2.1 Log Message Structure

The log message MUST be structured in the following way:

Data field	Tag	Data type	Mandatory?
Log Message	0x30	SEQUENCE	m
version	0x02	INTEGER	m
certifiedDataType	0x06	OBJECT IDENTIFIER	m
certifiedData		ANY DEFINED BY certifiedDataType	
Protocol Data	0x30	SEQUENCE	m
transactionNumber	0x80	INTEGER	m
signatureCounter	0x81	INTEGER	m
logTime	0x82	UTCTime	m
operationType	0x83	IA5String	m
serialNumber	0x84	OCTET STRING	m
optionalProtocolData	0x85	OCTET STRING	o
signature	0x30	SEQUENCE	m
signatureAlgorithm	0x30	SEQUENCE	m
algorithm	0x06	OBJECT IDENTIFIER	m
parameters		ANY DEFINED BY algorithm	
signatureValue	0x04	OCTET STRING	m

Table 2: Structure of a log message

2.2 Contents of Log Message

All elements of the log message structure MUST contain the following information:

Data	Description	Origin of data
version	Represents the version of the log message format and SHALL be set to ‘1’	Predefined value SHALL be “1”. Is provided by the Secure Element.
certifiedDataType	The element certifiedDataType MUST identify the type of the certified data.	Dependent of the certifiedDataType

Data	Description	Origin of data
certifiedData	MUST contain the application of system data that is to be protected by the Secure Element. The structure of certifiedData depends on the certifiedDataType and contains the actual protected data cf section 2.3.	
protocolData	MUST contain the protocol data generated by the Secure Element during the protection process.	Created by the Secure Element.
transactionNumber	MUST contain the transaction number generated by the Secure Element at the start of the transaction.	Created by the Secure Element.
signatureCounter	Shall contain the current count of signatures as created by the Secure Element.	Created by the Secure Element.
logTime	MUST contain the point in time of the Secure Element when the log message was created.	Created by the Secure Element.
operationType	MUST contain the name of the API or system function whose execution is recorded by the log message.	Created by the Secure Element.
serialNumber	MUST contain the hash value of the public key encoded as octet string of the certificate that corresponds to the private key of the key pair used for creating signatures.	Provided by the Secure Element.
optionalProtocolData	MAY be present. If present MUST contain additional BER-TLV encoded protocol data encoded as octet string	Created by the Secure Element.
signature	MUST provide information about the signature creation.	Provided by the Secure Element.
signatureAlgorithm	MUST provide information about the signature algorithm used by the Secure Element. The field algorithm MUST contain the object identifier of the used signature algorithm. The structure parameters depends on the algorithm.	Provided by the Secure Element. The signature algorithm MUST be configured in the Secure Element according to [BSI TR-03116].
signatureValue	MUST contain the signature that is built over the application or system data and protocol data.	Created by the Secure Element.

Table 3: Description of the elements of the log message

2.3 certifiedDataType

This Technical Guideline distinguishes between the two types of log messages, i.e. *transaction logs* and *system logs*. Transaction logs are log messages created to protect the actual transactions of the external application. System logs are generated to protect management or configuration operations of the Secure Element.

2.3.1 Certified data of transaction logs

A transaction log MUST be identified by the following object identifier (id-SE-API-transaction-log):

- bsi-de (0.4.0.127.0.7.0) applications (3) sE-API (7) sE-API-dataformats(1) 1

The certifiedData MUST be structured as follows:

Data field	Tag	Data type	Mandatory?
clientID	0x81	INTEGER	c
processData	0x82	OCTET STRING	m
processType	0x83	PrintableString	o
additionalData	0x84	OCTET STRING	o

Table 4: Structure of the certifiedData of a transaction log

The elements MUST contain the following information:

Data	Description
clientID	MUST be present if the Secure Element is used by more than one client in order to identify the client application
processData	MUST contain the process data
processType	If present, MUST contain information about the type of transaction
additionalData	MAY be present. If present, MUST contain additional (TLV encoded) data

Table 5: Description of the elements of the log message

2.3.2 Certified data of system logs

A system log MUST be identified by the following object identifier (id-SE-API-system-log):

- bsi-de (0.4.0.127.0.7.0) applications (3) sE-API (7) sE-API-dataformats(1) 2

The certifiedData MUST be structured as follows:

Data field	Tag	Data type	Mandatory?
systemFunctionData	0x81	OCTET STRING	m

Table 6: Structure of the certifiedData of a system log

The element systemFunctionData MUST contain information about the system operation. A list of system operations and the corresponding certifiedData is given in Appendix A.

2.4 Transaction Log Message Signature Creation

This signature included in a log message is calculated over the application data and protocol data parts of the log message. The signature SHALL be created by the signature algorithm.

2.4.1 Simplified Log Message Signature Creation

Text 1 provides an overview on how the log message signature is created.

First the message M is created. M is the input data of the signature function. The message is built by the concatenation of the plain values of clientID, processData, processType, additionalData and protocolData encoded as octet string.

Essential for the creation of a signature over the various data elements is a private key. This private key, here called $key_{private}$, belongs to the key pair that is managed by the Secure Element. This key SHALL be used for the creation of signature sig over M.

Note: The symbol "||" depicts a concatenation of data.

Input:	
From application:	clientID processData processType additionalData
From Secure Element:	protocolData
Actions: message M :=	version || certifiedDataType || clientID || processData || processType || additionalData || protocolData
signature sig :=	**SignatureFunction** ($key_{private}$, M)
Output:	sig

Text 1: General process of creating signatures

Text 2 shows the definition for protocolData.

protocolData :=	transactionNumber || signatureCounter || logTime || operationType || serialNumber || optionalProtocolData

Text 2: Creation of protocolData

2.4.2 Log Message Signature Creation

Every transaction SHALL be started and finished by calling the corresponding functions. Additionally, every transaction MAY be updated between start and finish.

2.4.2.1 Start a transaction

The result of a started transaction SHALL be a log message. The structure of this log message SHALL follow the definitions from table 2 and 4.

Text 3 shows the data that is used during the creation of the log message signature.

Input:	
From application:	clientID processData processType additionalData
From Secure Element:	protocolData
Actions: message M :=	version || certifiedDataType || clientID || processData || processType || additionalData || protocolData
signature sig :=	**SignatureFunction** ($key_{private}$, M)
Create log-message log	
Output:	log

Text 3: Creation of signatures in the context of a transaction start

To facilitate the signature creation, the actual input to the signature function are the plain data and TLV structure of the content of protocol data of the log message. Both components contain all previously mentioned elements. Both elements SHALL be concatenated. The protocol data SHALL contain the TLV structure (tags and length bytes) according to table 2.

message M :=	version || certifiedDataType || clientID || processData || processType || additionalData || protocolData
signature sig :=	**SignatureFunction** ($key_{private}$, M)

Text 4: Signature creation after a transaction start

2.4.2.2 Update a transaction

During the first execution of the transaction update the process data SHALL be managed by the Secure Element. On every subsequent call the passed process data SHALL be added to the value of concatenated process data from previous calls of the function to update the transaction.

Note: The process data already signed by the transaction start log message are not included in the update process data.

Text 5 shows the concatenation of process data during a transaction update:

Input:	
From application:	clientID $processData_{input}$
From Secure Element:	initialization of $processData_{update\ i-1}$ with an empty value during the first of call of updateTransaction, otherwise previous $processData_{update\ i-1}$
Actions: $processData_{update\ i}$ =	$processData_{update\ i-1}$ || $processData_{input}$
Output	nothing / status message

Text 5: Concatenation of process data in the context of a transaction update

Due to the limited memory capacity of some Secure Elements, the process data passed with every update MAY already be fed into the hash function of the signature algorithm. After this, the Secure Element can discard the passed process data from its internal memory. Only the hash function of the signature algorithm SHALL be used for this. The signature hash function SHALL remain open to add additional data in subsequent updates. The hash function SHALL be closed during the finish of the transaction.

If necessary, the Secure Element MAY sign not yet secured process data received during the update process. In this case, the signature SHALL be calculated over all process data that has been received so far and that has not yet been secured in accordance with the description in chapter 2.4.1.

2.4.2.3 Finish a transaction

To finish a transaction, a signature over all process data sent to the Secure Element after the start of the transaction and additional protocol data SHALL be created. This SHALL cause the creation of a log message. The log message SHALL follow the scheme defined in Table 2 and Table 4.

Text 6 pictures the creation of the final log message:

Input:	
From application:	clientID processDatafinish processType additionalData
From Secure Element:	$\text{processData}_{\text{update I}}$ protocolData
Actions: message M :=	version || certifiedDataType || clientID || $\text{processData}_{\text{update i}}$ || $\text{processData}_{\text{finish}}$ || processType || additionalData || protocolData
signature sig :=	**SignatureFunction** ($\text{key}_{\text{private}}$, M)
Create log-message log	
Output:	log

Text 6: Creation of signatures in the context of a transaction finish

The following cases SHALL be considered:

1. If the process data has been concatenated during updates *without* being already fed into the hash function of the signature algorithm, the signature SHALL be calculated over the whole array of stored process data and the additional protocol data.
2. If the process data of updates has already been fed into the hash function of signature algorithm, the hash SHALL be updated with the function call of the transaction finish, and the protocol data. Subsequently, the hash function SHALL be closed and the signature SHALL be created.

   To facilitate the signature creation, the actual input to the signature function are the plain process data and TLV structure of the protocol data of the log message. Both components contain all previously mentioned elements. Both elements SHALL be concatenated. The protocol data SHALL contain the TLV structure (tags and length bytes) according to Table 2 and Table 4.

message M :=	version || certifiedDataType || clientID || signatureCounter || $\text{processData}_{\text{update i}}$ || $\text{processData}_{\text{finish}}$ || processType || additionalData || protocolData
signature sig :=	SignatureFunction ($\text{key}_{\text{private}}$, M)

Text 7: Signature creation after a transaction finish

2.4.3 Log Message Signature Creation with Signed Updates

Due to limited capacities of a Secure Element, it might be needed to not only create log messages for the start and finish of a transaction, but also for process data updates. By implementing this technique, no intermediate data needs to be stored on the Secure Element, as every input is directly fed into a signature.

Text 8 displays the creation of a log message:

Input:	
From application:	clientID processData processType additionalData
From Secure Element:	protocolData
Actions: message M :=	certifiedDataType || clientID || processData || processType || additionalData || protocolData
signature sig :=	SignatureFunction (key$_{private}$, M)
Create log-message log	
Output:	log

Text 8: Creation of log messages for start, update, and finish of a transaction

To facilitate the signature creation, the actual input to the signature function are the plain process data and TLV structure of the protocol data of the log message. Both components contain all previously mentioned elements. Both elements SHALL be concatenated. The protocol data SHALL contain the TLV structure (tags and length bytes) according to Table 2 and Table 4.

message M :=	certifiedDataType || clientID || processData || processType || additionalData || protocolData
signature sig :=	**SignatureFunction** (key$_{private}$, M)

Text 9: Signature creation if every function call results in a log message

If an implementation with signed updates is chosen, every function call (start, update, finish) SHALL result in exactly one log message. Every signature SHALL be created immediately after calling each function.

Note: A Secure Element might use a single function to generate the protocol data and signature for a log message. If it does so, the function parameters of the Secure Element MAY pass the information whether the function call is a start, update, or finish, so that the operationType field in the protocol data indicates the correct log type. This information MAY also be passed by the usage of three different function calls. Nevertheless, every logType element SHALL indicate the correct transaction phase.

2.5 Transaction Log Message Signature Validation

After the affiliation and the correctness of the certificate belonging to the log messages has been verified, the log message signature verification SHALL be performed analogue to the signature creation.

The log message signatures SHALL be verified with the public key of the certificate whose serial number is included in the log message.

message M :=	certifiedDataType || clientID || processData || processType || additionalData || protocolData
verificationResult :=	**VerifySignatureFunction** (key$_{public}$, sig, M, signatureAlgorithm)

Text 10: General verification of protocol data

All data not covered by a valid signature SHALL be considered invalid.

2.6 System Log Message Creation

Text 11 provides an overview on how the system log messages are created.

The message is built by the concatenation of the plain values of systemFunctionData and protocolData encoded as octet string.

Input:	
From system:	systemFunctionData
From Secure Element:	protocolData
Actions: message M :=	systemFunctionData || protocolData
signature sig := Create log-message log	SignatureFunction (key$_{private}$, M)
Output:	log

Text 11: Creation of system log messages

3 Secure Element Functionality

The SE API serves as a wrapper around the functionality of a Secure Element. However, the specification of the SE API assumes that the Secure Element provides a certain set of functionality.

Table 7 provides a generic overview over this functionality. The Secure Element SHALL provide these functionalities. However, concrete aspects as the names of the functions should not be understood as normative requirements with respect to the Secure Element. Furthermore, the specification does not state any particular requirements regarding the practical implementation of this functionality.

Functionality	Description
Authenticate user	This function serves to authenticate the user and to establish a trusted channel.
Start a transaction	This function starts an external transaction with the Secure Element.
Update a transaction	This function updates an external transaction with the Secure Element.
Finish a transaction	This function finishes an external transaction with the Secure Element.
Sign a transaction	This function allows to secure process information by the use of a log message (which is signed).
Retrieve a log message	This function retrieves the last log message parts from the Secure Element.
Set time	This function sets the time that is maintained by the Secure Element.
Export certificate	This function retrieves the current certificate that is used by the Secure Element for signing log messages.

Table 7: Functionality of the Secure Element

4 Secure Element API Functionality

4.1 Error handling

If the application receives any kind of error from the SE API it falls into the responsibility of the application to handle the situation. Depending on the concrete function that threw the error and depending on the concrete error, the application may:

- decide to do nothing at all,
- simply repeat the last function call,
- repeat a larger amount of steps.

4.2 Maintenance and Time Synchronization

4.2.1 Initialize

This function Initialize MUST be used to start the initialization of the SE API.

4.2.1.1 Initialize – Input parameters

Name	Type (OMG IDL)	Required?	Meaning
description	string	REQUIRED	This parameter presents a short description of the Secure Element.

Table 8: Input parameters Initialize function

4.2.1.2 Initialize – Output parameters

None.

4.2.1.3 Initialize – Exceptions

Exception	Meaning
ErrorStoringInitDataFailed	Storing of the initialization data failed.

Table 9: Exceptions for the Initialize function

4.2.1.4 Detailed description

1. This function SHALL instruct the Secure Element to initialize the Secure Element and store the initialization data in form of the description of the Secure Element.
2. If storing of the initialization data fails, the Secure Element SHALL set back the initial data set to the state before storing the provided initialization data. Furthermore, the function SHALL raise the exception ErrorStoringInitDataFailed and exit the function.
3. If the storing has been successful, this function SHALL return EXECUTION_OK.

4.2.2 UpdateTime

The function updateTime can be used to update the current date/time that is maintained by the Secure Element.

4.2.2.1 UpdateTime – Input parameters

Name	Type (OMG IDL)	Required?	Meaning
newDateTime	DateTime	CONDITIONAL	The new time value for the date/time maintained by the Secure Element.
useTimeSync	boolean	REQUIRED	If the underlying Secure Element supports time synchronization, this parameter instructs the function to utilize this feature.

Table 10: Input parameters for the updateTime function

4.2.2.2 UpdateTime – Output parameters

None.

4.2.2.3 UpdateTime – Exceptions

Exception	Meaning
ErrorSetTimeFailed	The execution of the Secure Element functionality to set the time failed.
ErrorRetrieveLogMessageFailed	The execution of the Secure Element functionality to retrieve log message parts has failed.
ErrorStorageFailure	Storing of the log message has failed.

Table 11: Exceptions for updateTime function

4.2.2.4 UpdateTime – Detailed description

1. If the parameter useTimeSync has :
   1 the value true, the function SHALL instruct the Secure Element to use its time synchronization mechanism to update the local time. If the execution fails, the function SHALL raise the exception ErrorSetTimeFailed.
   2 the value false, the function SHALL invoke the functionality of the Secure Element to set the time with the provided newDateTime. If the execution of fails, the function SHALL raise the exception ErrorSetTimeFailed.
2. The function SHALL retrieve the log message of the Secure Element. If the retrieval of the log message fails, the function SHALL raise the exception ErrorRetrieveLogMessageFailed.
3. The function SHALL store the previously retrieved log message on the storage medium. If storing fails, the function SHALL raise the exception ErrorStorageFailure.
4. The function SHALL return the value EXECUTION_OK to indicate that the execution of the function updateTime has been successful.

4.3 Input Functions

4.3.1 StartTransaction

This function provides the functionality to start a new transaction.

4.3.1.1 StartTransaction – Input parameters

Name	Type (OMG IDL)	Required?	Meaning
clientID	unsigned long	CONDITIONAL	MUST be present if the Secure Element is used by more than one client
processData	octet []	REQUIRED	This parameter represents all the necessary information regarding the initial state of the process.
processsType	string<100>	OPTIONAL	This parameter is used to identify the type of the transaction as defined by the application
additionalData	octet []	OPTIONAL	Reserved for future use.

Table 12: Input parameters for startTransaction function

4.3.1.2 StartTransaction – Output parameters

Name	Type (OMG IDL)	Required?	Meaning
transactionNumber	unsigned long	REQUIRED	The value of this parameter represents a transaction number that has been assigned by the Secure Element to the process.
logTime	DateTime	REQUIRED	The value represents the point in time of the Secure Element when the log message was created
serialNumber	octet[]	REQUIRED	This field contains the hash value over the key that is part of the certificate in the secure element.
signatureCounter	unsigned long	REQUIRED	The current value of the signature counter.
signatureValue	octet[]	OPTIONAL	The value represents the signature value

Table 13: Output parameters for startTransaction function

4.3.1.3 StartTransaction – Exceptions

Exceptions	Meaning
ErrorStartTransactionFailed	The execution of the Secure Element functionality to start a transaction failed.
ErrorRetrieveLogMessageFailed	The execution of the Secure Element functionality to retrieve log message parts has failed.
ErrorStorageFailure	Storing of the log message failed.

Table 14: Exceptions of the startTransaction function

4.3.1.4 StartTransaction – Detailed description

The following description specifies the behavior of the *startTransaction* function in detail:

1. The function SHALL invoke the function to start a transaction of the Secure Element and pass on the clientId. In this step, the Secure Element will generate a transaction number for the transaction. If the execution of the Secure Element function fails, the function SHALL raise the exception ErrorStartTransactionFailed.
2. Next, the function SHALL retrieve the log message parts created by the Secure Element. If the execution of this function fails, the exception ErrorRetrieveLogMessageFailed SHALL be raised.
3. The input data and the log message parts created by the Secure Element SHALL be combined into a complete log message and the log message SHALL be stored. If storing fails, the function SHALL raise the exception ErrorStorageFailure.
4. After successfully storing the log message, the function SHALL return the current transaction number (as an output parameter) by transactionNumber, the current signature counter by signatureCounter, the time of the log message creation by logTime, the hash value over the used public key by hashValueKey and MAY return the signature Value by signatureValue. Additionally, the function SHALL return the return value EXECUTION_OK to indicate that of the function startTransaction has been successful.

4.3.2 UpdateTransaction

This function updates an existing transaction.

4.3.2.1 UpdateTransaction – Input parameters

Name	Type (OMG IDL)	Required?	Meaning
clientID	unsigned long	CONDITIONAL	MUST be present if the Secure Element is used by more than one client
transactionNumber	unsigned long	REQUIRED	This parameter is used to unambiguously identify the current transaction .
processData	octet []	REQUIRED	This parameter represents all the necessary information regarding the initial state of the process.
processsType	string<100>	OPTIONAL	This parameter is used to identify the type of the transaction as defined by the application

Table 15: Input parameters for updateTransaction function

4.3.2.2 UpdateTransaction – Output parameters

Name	Type (OMG IDL)	Required?	Meaning
logTime	DateTime	CONDITIONAL	The value represents the point in time of the Secure Element when the log message was created
signatureValue	Octet[]	CONDITIONAL	The value represents the signature value
signatureCounter	unsigned long	CONDITIONAL	The current value of the signature counter.

Table 16: Output parameters for updateTransaction function

4.3.2.3 UpdateTransaction – Exceptions

Exception	Meaning
ErrorUpdateTransactionFailed	The execution of the Secure Element functionality to update a transaction failed.
ErrorStorageFailure	Storing of the log message has failed.
ErrorLogMessageRetrievalFailed	Retreival of the log message from the Secure Element failed.
ErrorNoTransaction	No transaction is known to be open under the provided transaction number.

Table 17: Exceptions of the updateTransaction function

4.3.2.4 UpdateTransaction – Detailed description

The following description specifies the behavior of the *updateTransaction* (without signed updates) function in detail:

1. The function SHALL invoke the functionality of the Secure Element to update a transaction and pass on the clientId and the transactionNumber. If the execution of the function fails, the exception ErrorUpdateExternalTransactionFailed SHALL be raised.
2. The Secure Element SHALL check whether the transactionNumber belongs to an open transaction. If this is not the case, the function SHALL return the error ErrorNoTransaction and exit.
3. In case of UpdateTransaction without signed updates:
    1. The function SHALL return the return value EXECUTION_OK to indicate that the execution of the function updateTransaction has been successful.
4. In case of UpdateTransaction with signed updates:
    1. Next, the function SHALL retrieve the log message parts created by the Secure Element. If the execution of this function fails, the exception ErrorLogMessageRetrievalFailed SHALL be raised.
    2. The input data and the log message parts created by the Secure Element SHALL be combined into a complete log message and the log message SHALL be stored. If storing fails, the function SHALL raise the exception ErrorStorageFailure.
    3. After successfully storing the log message, the function SHALL return the time of the log message creation by logTime and MAY return the signature Value by signatureValue and signature counter by signatureCounter. Additionally, the function SHALL return the return value EXECUTION_OK to indicate that of the function UpdateTransaction has been successful.

4.3.3 FinishTransaction

This function finalizes an existing transaction.

4.3.3.1 FinishTransaction – Input parameters

Name	Type (OMG IDL)	Required?	Meaning
clientID	unsigned long	CONDITIONAL	MUST be present if the Secure Element is used by more than one client
transactionNumber	unsigned long	REQUIRED	This parameter is used to unambiguously identify the current transaction.
processData	octet []	REQUIRED	This parameter represents all the necessary information regarding the initial state of the process.
processsType	string<100>	OPTIONAL	This parameter is used to identify the type of the transaction as defined by the application
additionalData	octet []	OPTIONAL	Reserved for future use.

Table 18: Input parameter of the finishTransaction function

4.3.3.2 FinishTransaction – Output Parameters

Name	Type (OMG IDL)	Required?	Meaning
logTime	DateTime	CONDITIONAL	The value represents the point in time of the Secure Element when the log message was created
signatureValue	Octet[]	OPTIONAL	The value represents the signature value
signatureCounter	unsigned long	REQUIRED	The current value of the signature counter.

Table 19: Output parameter for finishTransaction function

4.3.3.3 FinishTransaction – Exceptions

Exception	Meaning
ErrorFinishTransactionFailed	The execution of the Secure Element functionality to finish a transaction failed.
ErrorRetrieveLogMessageFailed	The execution of the Secure Element functionality to retrieve log message parts has failed.
ErrorStorageFailure	Storing of the log message failed.

Table 20: Exceptions for finishTransaction function

4.3.3.4 FinishTransaction – Detailed description

The following description specifies the behavior of the *finishTransaction* function in detail:

1. The function SHALL invoke the functionality of the Secure Element to finish a transaction and pass on the clientId and the transactionNumber of process to finish and the processData. If the execution of the function fails, the exception ErrorFinishTransactionFailed SHALL be raised.
2. Next, the function SHALL retrieve the log message parts created by the Secure Element. If the execution of this function fails, the exception ErrorRetrieveLogMessageFailed SHALL be raised.
3. The process data, created since the start of the transaction and the log message parts created by the Secure Element SHALL be combined into a complete log message and the log message SHALL be stored. If storing fails, the function SHALL raise the exception ErrorStorageFailure.
4. The function SHALL return the time of the log message creation by logTime, the signature counter by signatureCounter and MAY return the signature Value by signatureValue. Additionally, the SE API SHALL return the return value EXECUTION_OK to indicate that the execution of the function finishTransaction has been successful.

4.4 Export Functions

4.4.1 Export

This function is used to export the log messages, containing the process and protocol data.

4.4.1.1 Export – Input parameters

Name	Type (OMG IDL)	Required?	Meaning
clientID	unsigned long	OPTIONAL	MUST be present if the Secure Element is used by more than one client.
transactionNumber	unsigned long	OPTIONAL	If present, the function SHALL only return the log messages associated with the given transaction number and, if present, clientID.
startTransactionNumber	string <100>[1]	OPTIONAL	If present, the function SHALL only return the log messages associated within the interval of startTransactionNumber to endTransactionNumber (inclusive).
endTransactionNumber	string <100>[1]	OPTIONAL	If present, the function SHALL only return the log messages associated within the interval of startTransactionNumber to endTransactionNumber (inclusive).
startDate	DateTime	OPTIONAL	If present, the function SHALL only return the log messages between startDate and endDate (inclusive). Date/Time SHALL be encoded in UTC.
endDate	DateTime	OPTIONAL	If this parameter is provided, the function SHALL only return the log messages between startDate and endDate (inclusive). Date/Time SHALL be encoded in UTC.
maximumNumberRecords	long	OPTIONAL	If this parameter is provided, and its value is not 0, the function SHALL only return the log messages if the number of relevant records is less or equal to the number of maximum records. Else, an error SHALL be returned[1]. If this parameter is provided and its value is 0, the function SHALL return all stored log messages.

Table 21: Input parameters for export function

1 This optional value allows the client to ensure that only a specific number of log messages is returned by the export function. If the function returns an error, the calling application can restructure the call and e.g. only ask for a smaller number of data records next time.

4.4.1.2 Export – Output parameters

Name	Type (OMG IDL)	Required?	Meaning
exportData	The function will return the following data in a defined export format (see chapter 5): – application data and corresponding protocol data in form of TLV – encoded log messages (see chapter 2.1) – initialization information – certifcate(s) that are needed to verify the log messages	REQUIRED	Log messages and additional files needed to verify the signatures included in the log messages.

Table 22: Output parameters for export function

4.4.1.3 Export – Exceptions

Exception	Meaning
ErrorIdNotFound	No data found for the provided clientID.
ErrorTransactionIdNotFound	No data found for the provided transactionNumber.
ErrorNoDataAvailable	No data found for the provided selection.
ErrorTooManyRecords	The amount of requested records exceeds the parameter maximumNumberRecords
ErrorParameterMismatch	Mismatch in parameters of function.

Table 23: Exceptions for export function

4.4.1.4 Export – Detailed Description

The following description specifies the behavior of the export function in detail.

1. The function SHALL check the input parameters for validity. If any of the checks fails, the function SHALL raise the exception ErrorParameterMismatch. This specifically includes the following checks:
    1. If transactionNumber has been provided, neither startDate nor endDate SHALL be provided.
    2. If startTransactionNumber and endTransactionNummber have been provided, neither startDate, endDate nor clientID SHALL be provided.
    3. If startDate has been provided, endDate MUST be provided as well.
    4. If endDate has been provided, startDate MUST be provided as well.
    5. If startTransactionID has been provided, endTransactionNumber SHALL be provided as well.
    6. If endTransactionID has been provided, startTransactionNumber SHALL be provided as well.
    7. If provided, startDate and endDate MUST be valid date/time values.
    8. If provided, endDate MUST lay after startDate.

   9. If a startDate and endDate have been provided, neither a clientID nor a transactionID MUST be provided.

2. If transactionNumber has been provided, the function SHALL check whether any data has been stored regarding this transactionNumber and, if present, clientID.
   1. If no data is available for the transactionNumber, the function SHALL raise the exception ErrorTransactionIdNotFound. If no data is available for the clientID, the function SHALL raise the exception ErrorIdNotFound.
   2. Else, the function SHALL return the data that corresponds to the provided transactionNumber (as an output parameter by exportData). Additionally, the function SHALL return the return value EXECUTION_OK to indicate that the execution of the function has been successful.
3. If startTransactionINumber and endTransactionNumber have been provided, the function SHALL check whether any data for the relevant period between these two transaction numbers has been stored.
   1. If no data is available, the function SHALL raise the exception ErrorTransactionIdNotFound.
   2. If maximumNumberRecords has been provided and its value is not 0, the function SHALL check whether the amount of records that have been found is less than maximumNumberRecords. If this is not the case, the function SHALL raise ErrorTooManyRecords.
   3. Else, the function SHALL return the data (as an output parameter) by exportData. Additionally, the function SHALL return the return value EXECUTION_OK to indicate that the execution of the function has been successful.
4. If a startDate and endDate have been provided, the function SHALL check whether any information has been stored in the period between startDate and endDate (inclusive).
   1. If no data is available, the function SHALL raise the exception ErrorNoDataAvailable.
   2. If maximumNumberRecords has been provided and its value is not 0, the function SHALL check whether the amount of records that have been found is less than maximumNumberRecords. If this is not the case, the function SHALL raise ErrorTooManyRecords.
   3. Else, the function SHALL return the data that corresponds to the provided period between startDate and endDate (inclusive, as an output parameter) by exportData. Additionally, the function SHALL return the return value EXECUTION_OK to indicate that the execution of the function has been successful.
5. If NEITHER , a transactionNumber nor startDate and endDate have been provided, the function SHALL check if any data has been stored.
   1. If no data is available, the function SHALL raise the exception ErrorNoDataAvailable.
   2. If maximumNumberRecords has been provided and its value is not 0, the function SHALL check whether the amount of records that have been found is less than maximumNumberRecords. If this is not the case, the function SHALL raise ErrorTooManyRecords.
   3. Else, the function SHALL return all the stored data (as an output parameter) by exportData. Additionally, the function SHALL return the return value EXECUTION_OK to indicate that the execution of the function has been successful.

4.4.2 GetLogMessageCertificate

This function returns the certificate that belongs to the current key pair used for creating the signatures contained in log messages.

4.4.2.1 GetLogMessageCertificate – Input parameters

None

4.4.2.2 GetLogMessageCertificate – Output parameters

Name	Type (OMG IDL)	Required?	Meaning
certificate	octet[]	REQUIRED	This parameter MAY occur several times, each containing a card-verifiable certificate of the certificate chain belonging to the key pair that is currently used to create signatures contained in log messages.

Table 24: Output parameters for getLogMessageCertificate function

4.4.2.3 GetLogMessageCertificate – Exceptions

Exception	Meaning
ErrorExportCertFailed	The execution of the Secure Element function exportCert failed.

Table 25: Exceptions for getLogMessageCertificate function

4.4.2.4 Detailed description

1. This function SHALL invoke the certificate export function of the Secure Element to retrieve the certificates of the certificate chain that belongs to the key pair that is used to create the signatures contained in a log message. If the execution of this function fails, the ErrorExportCertFailed exception SHALL be raised.
2. The function SHALL return the certificate encoded according to 5.1.3. Additionally, the function SHALL return the return value EXECUTION_OK to indicate that the execution of the function GetLogMessageCertificate has been successful.

4.4.3 RestoreFromBackup

This function enables the restoring of a backup in the SE API and storage. The function utilizes data that has been exported by the use of the export function (see chapter 4.4.1).

4.4.3.1 RestoreFromBackup – Input parameters

Name	Type (OMG IDL)	Required?	Meaning
restoreData	octet []	REQUIRED	Contains the data that SHALL be restored in the SE API and storage.

Table 26: Input parameters restoreFromBackup function

4.4.3.2 RestoreFromBackup – Output parameters

None.

4.4.3.3 RestoreFromBackup – Exceptions

Exception	Meaning
ErrorRestoreFailed	The restore process has failed.
ErrorDeletingFailed	Deleting of the secured data from the storage from the SE API before the restoring has failed.

Table 27: Exceptions of restoreFromBackup function

4.4.3.4 RestoreFromBackup – Detailed description

The following description specifies the behavior of the restoreFromBackup function in detail:

1. The function SHALL perform the following tasks within one atomic transaction.
    1. The function SHALL delete the secured data in the storage. If this step fails, the function SHALL raise the exception ErrorDeletingFailed and exit the function. In this case, it has to be ensured that the SE API and the storage are left in a consistent state.
    2. The function SHALL restore the secured data provided in restoreData. If the restore procedure fails, the function SHALL raise the exception ErrorRestoreFailed and exit the function. In this case, it SHALL be ensured that the data stock of the storage are set back to the state that corresponds to the point in time before the restore procedure.
2. If the restoring has been successful, the function SHALL return EXECUTION_OK.

4.4.4 ReadLogMessage

This function enables the reading of a log message that bases on the last log message parts that have been produced and processed by the Secure Element.

4.4.4.1 ReadLogMessage – Input parameters

None

4.4.4.2 ReadLogMessage – Output parameters

Name	Type (OMG IDL)	Required?	Meaning
logMessage	octet []	REQUIRED	Contains the last log message that the Secure Element has produced

Table 28: Output parameters from ReadLogMessage function

4.4.4.3 ReadLogMessage – Exceptions

Exception	Meaning
ErrorNoLogMessage	No log message parts found
ErrorReadingLogMessage	Error while retrieving log message parts.

Table 29: Exceptions for ReadLogMessage function

4.4.4.4 ReadLogMessage – Detailed description

The following description specifies the behavior of the readLogMessage function in detail:

1. The function SHALL retrieve the log message parts created and processed by the Secure Element most recently. If no log message parts are found in the Secure Element, the exception ErrorNoLogMessage SHALL be raised and the function SHALL be exited. If the retrieving of the log message parts fails, the exception ErrorReadingLogMessage SHALL be raised and the function SHALL be exited.
2. The retrieved log message parts SHALL be combined into a complete log message. This log message SHALL be returned to the application over the output parameter logMessage. Additionally, the SE API SHALL return the return value EXECUTION_OK to indicate that the execution of the function readLogMessage has been successful

5 Export Formats

5.1 TAR and TLV Export

If the export information is requested, the requested information SHALL be exported into a [POSIX.1-1988] compliant TAR file that in turn contains the following files:

– the initialization information

– the log messages

– the certificate(s) that are needed to verify the log messages

5.1.1 Initialization Information File

The TAR file SHALL contain a file named ‘info.csv’. This coma separated value (CSV) text file SHALL follow the structure shown in Text 12.

```
„description:“, $1; “manufacturer:” $2,”version”, $3
```

Text 12: content of info.csv

The variables $1 to $3 shall be replaced with the following values:

– $1 to be replaced by the content of the parameter description from the initialization function,

– $2 to be replaced by the content of the parameter manufacturer from the initialization function,

– $3 to be replaced by the version information

Line endings in the text file SHALL be encoded in UNIX style (i.e. Line feed, '\n', 0x0A) and the delimiter SHALL be ‘,’ (a comma).

5.1.2 Log Messages Files

Log message files in a TAR container SHALL be named in the following way:

CLIENT-ID_PROCESS-ID_TRANSACTION_DATE_TYPE.der

The parts of the file name separated by an underscore should be replaced by values according to table 30.

Part of file name	Description
CLIENT-ID	The ID that identifies the client.
TRANSACTION	The ID of the transaction that has been assigned by the Secure Element.
DATE	The DATE/TIME that the log file has been created. The format SHALL be "YYMMDDhhmmssZ". This implies that the date/time SHALL only be stored in UTC and be agnostic of any time zone.
TYPE	This part of the file name shows whether the log message in the file is of type "START", "UPDATE", "END" or the file is a certificate "CERT". – START is used if the file contains process data for the start of a process (used in the context of the function startTransaction), – UPDATE is used if the file contains process data for an update (used in the context of the function updateTransaction), – END is used if the file contains process data for the finalization of a process (used in the context of the function finishTransaction), – CERT is used if the file contains a certificate.
SERIAL	This part of the file name refers to the serial number of the certificate.

Table 30: Parts of the file names in export tar archive

If a file of this name already exists, the filename SHALL be appended by a counter (before the .der suffix).

The contents of the file SHALL be structured as defined in table 2 and 4.

If a log message is not related to a process, the PROCESS-ID part of the filename and the following underscore SHALL be omitted.

5.1.3 Certificate files

The TAR file SHALL contain all certificates needed to verify the exported log messages, encoded as Card-Verifiable-Certificates (CVC) according to [ISO7816-8].

The certificate files SHALL be named

SERIAL_TYPE.der

The parts of the filename SHALL be replaced by values according to table 30.

6 Appendix A: System log messages

This section contains management or configuration functions that MUST be logged by the Secure Element.

6.1 Initialization

The systemFunctionData octet string MUST contain the following information:

Data field	Tag	Data type	Mandatory?	Description
description	0x81	PrintableString	m	MUST contain the description of the initialization function call

6.2 Set time

The systemFunctionData octet string MUST contain the following information:

Data field	Tag	Data type	Mandatory?	Description
timeBeforeUpdate	0x81	UTCTime	m	MUST contain the current time of the Secure Element (before the update). If the time of the Secure Element has not been set, timeBeforeUpdate is set to 0.
timeafterUpdate	0x82	UTCTime	m	MUST contain the new reference time of the Secure Element

7 Appendix B: Mapping of OMG IDL constructs to ANSI C and Java

7.1 Introduction

This annex describes the OMG IDL constructs that have been used in the context of the SE API and how the OMG IDL definitions of the functions in Chapter 4 have been translated into their respective translations in ANSI C and Java.

The approach is based on the OMG IDL mappings to ANSI C and Java that are provided in [OMG2017a], [OMGx] and [OMG1999]. These descriptions are adopted in various places as they focus on a translation into CORBA constructs (which is not the objective for the interface defined in this document).

The chapters of [OMG2017a] that are listed in Table 31 can be used without any modification, except for the usage of OMG IDL arrays (see Chap. 7.4). These chapters describe the specification of the OMG IDL language constructs that have been used to define the SE API.

Building block	Chapter	Description
Building Block Core Data Types	7.4.1	Specification of language constructs for: • IDL specifications • modules • constants • data types
Building Block Interfaces – Basic	7.4.3	Specification of language constructs for: • exceptions • interfaces • operations • attributes

Table 31: Relevant building blocks in [OMG2017a]

Table 32 provides an overview over the following sub chapters and shows, which parts of the OMG IDL standard needed adoption.

Section	Description
7.2	This chapter shows the representation of OMG IDL basis data types in ANSI C and JAVA.
7.3	This chapter shows the representation of OMG IDL strings in ANSI C and JAVA.
7.4	This chapter considers the representation of OMG IDL arrays in ANSI C and JAVA.
7.5	General aspects of the representation of an OMG IDL specification in ANSI C and JAVA.
7.6	This chapter shows how OMG IDL exceptions are specified in ANSI C and JAVA.
7.7	This chapter shows how OMG IDL optional function parameters are specified in ANSI C and JAVA.
7.8	This chapter shows how OMG IDL input parameters are specified in ANSI C and JAVA.
7.9	This chapter shows how OMG IDL output parameters in ANSI C and JAVA.
7.10	This chapter shows how OMG IDL return values are defined in ANSI C and JAVA.

Table 32: Overview of the following chapters

7.2 Mapping of basic types

Table 33 contains information regarding the basic data types that have been used to define the SE API. This table has been specified under consideration of the following aspects:

– OMG IDL standard definition for the syntax of basic types in [OMG2017a], Chap. 7.4.1.4.4.1.1, p. 25 f.

– The value ranges for the integer types in OMG IDL as defined in [OMG2017a], Chap. 7.4.1.4.4.1.1.1, p. 26.

– The mapping of the OMG IDL integer types to the corresponding Java types as defined in [OMGx], Table 4.1, p. 6. The definition of the value ranges regarding the relevant Java types occurs in [ORACLE2017] (see Chap. 4.2.1, p. 43).

– The mapping of the OMG IDL integer types to the corresponding C types as defined in [OMG1999], Chap. 1.7, p. 1-10. As [OMG1999] does not consider ANSI C, the definitions of integer types in [ANSI99] have to be taken into account. Here, the limit values for the ranges of the different integer types are defined in [ANSI99], Chap. 5.2.4.2.1. Regarding signed integer values, the limit values define that the

  – minimal limit value in a concrete implementation has to be equal or smaller than the corresponding minimal limit value defined in [ANSI99].

  – maximal limit value in a concrete implementation has to be equal or greater than the corresponding maximal limit value defined in [ANSI99].

  For an unsigned integer type, the maximal limit value in a concrete implementation has to be equal or greater than the corresponding maximal limit value defined in [ANSI99].

OMG IDL	ANSI C	Java	Comment
short (-2^{15} ... 2^{15}- 1)	short int ($-(2^{15}$ - 1) ... 2^{15} - 1)	short (-2^{15} ... 2^{15}- 1)	ANSI C: Common implementations provide a value range of($-(2^{15}$) ... 2^{15} - 1)
long (-2^{31} ... 2^{31}- 1)	long int ($-(2^{31}$ - 1) ... 2^{31} - 1)	int (-2^{31} ... 2^{31}- 1)	ANSI C: Common implementations provide a value range of($-(2^{31}$) ... 2^{31} - 1)

OMG IDL	ANSI C	Java	Comment
long long (-2^{63} ... 2^{63}- 1)	long long int (-(2^{63} - 1) ... 2^{63} - 1)	long (-2^{63} ... 2^{63}- 1)	ANSI C: Common implementations provide a value range of(-(2^{63}) ... 2^{63}- 1)
unsigned short (0 ... 2^{16}-1)	unsigned short int (0 ... 2^{16}-1)	int (-2^{31} ... 2^{31}- 1)	ANSI C: Common implementations provide a value range of 0 ... 2^{16}-1 Java: The range of the corresponding Java type does not match because Java does not support unsigned types. The Java type int SHALL be used. The developers SHALL examine that relevant parameter values belong to the correct range.
unsigned long (0 ... 2^{32}-1)	unsigned long int (0 ... 2^{32}-1)	long (-2^{63} ... 2^{63}- 1)	ANSI C: Common implementations provide a value range of 0 ... 2^{32}-1 Java: The range of the corresponding Java type does not match because Java does not support unsigned types. Therefore, the Java type long SHALL be used. Here, the developers SHALL examine that relevant parameter values belong to the correct range.
unsigned long long (0 ... 2^{64} -1)	unsigned long long int (0 ... 2^{64} - 1)	long (-2^{63} ... 2^{63}- 1)l	ANSI C: Common implementations provide a value range of 0 ... 2^{64} – 1 Java: The range of the corresponding Java type does not match because Java does not support unsigned types. The Java type long SHALL be used for the mapping. In this context, the value range from 0 to 2^{63}- 1 SHALL be relevant for the mapping. Accordingly, the maximal value of the used Java type long is smaller than the maximal value of the corresponding OMG IDL type.

OMG IDL	ANSI C	Java	Comment
octet (see [OMG2017a], Chap. 7.4.1.4.4.1.1.6, p. 27)	unsigned char	byte (see [ORACLE2017], Chap. 4.2.1, p. 43)	OMG IDL: "The octet type is an opaque 8-bit quantity" ([OMG2017a], Chap.7.4.1.4.4.1.1.6, p. 27)
boolean (see [OMG2017a], Chap. 7.4.1.4.4.1.1.5, p. 27)	_Bool (see [ANSI99], Chap. 6.2.5, p. 33).	boolean (see [ORACLE2017], Chap. 4.2, p. 43)	OMG IDL: The boolean data type can only take the values TRUE and FALSE. ANSI C: ANSI C provides the header file <stdbool.h> (, Chap. 7.16, p. 252) that enables the use of the identifier bool for the type _Bool. In the context of the SE API the specifier bool is used.
Presentation of date/time by the native type DateType.	Presentation by the structure tm from the header file time.h. (cf. [ANSI99], Chap. 7.23.1, p. 337)	Presentation by using the Java class java.util.Gregorian Calendar	OMG IDL: An OMG IDL native type allows a mapping to a type of a specific programming language. The date/time SHALL be represented in UTC.

Table 33: Mapping of data types

7.3 Definition of strings

Regarding the definition of strings, the OMG IDL distinguishes between non-wide strings and wide strings. As no wide strings have been used to define the SE API, the following discussion refers only to non-wide strings.

In the context of the OMG IDL, non-wide strings are represented by the type string (see [OMG2017a], Chap. 7.4.1.4.4.1.2.2, p. 27). Optionally, it is possible to define the maximum size of a string. The size of a string is represented by a positive integer value that is surrounded by the signs < and >.

In Java OMG IDL strings are mapped to the data type java.lang.String (see [OMG2017a], Chap. 4.3.3, p. 56)

In ANSI C OMG IDL, strings are implemented by string literals (see [ANSI99], Chap. 6.4.5, p. 62 f.). A string literal is represented by an array of elements of the type char. String literals are terminated by a null character (see [ANSI99], Chap. 7.1.1, p. 164).

7.4 Arrays

The OMG IDL provides the language construct array (see [OMG2017a], Chap. 7.4.1.4.4.3) that represents the data structure array. These arrays can be one-dimensional or multi-dimensional. In deviation from [OMG2017a]the use of arrays of variable length is allowed.

In the context of the SE API definition, only one-dimensional arrays are used as types of function parameters. At this, an array is represented by its type, its name and a following opening and closing squared bracket. If the array is of fixed length, the squared brackets contain the definition for the size of the array in form of a positive integer value. If an array has no fixed size, the squared brackets contain no integer number. This notation for the use of arrays with no fixed size represents an extension to [OMG2017a].

OMG IDL arrays are represented by the particular array constructs in Java (see [ORACLE2017], Chap. 10, p. 347) and ANSI C (see [ANSI99], Chap. 6.7.5.2, p. 116 f.) respectively.

7.5 Definition context

OMG IDL specifications are contained in IDL-files. An IDL specification can contain the definitions for interfaces, exceptions, types, and constants. These definitions can be grouped by modules (see [OMG2017a], 7.4.1.4.2, p. 20). Regarding the OMG IDL definition of the SE API, the above mentioned constructs are grouped by a module.

For the definition of interfaces, the OMG IDL provides the language construct interface (see [OMG2017a], 7.4.3.4.3, p. 37).

OMG IDL modules are implemented in Java by the construct package (cf. [ORACLE2017], Chap. 7.4, p. 181) of the same name. OMG IDL interfaces are mapped to the Java construct interface (see [ORACLE2017], Chap. 9, p. 293 ff.). Java interfaces contain the signature definitions of functions. OMG IDL interfaces are mapped to a public Java interface of the same name that is contained in a Java-file of the same name.

It is not possible to implement OMG IDL modules in ANSI C. For the implementation of OMG IDL interfaces ANSI C does not provide an explicit language construct. Accordingly, OMG IDL interfaces are implemented in ANSI C by defining the relevant function signatures in header files. Here, a header file has the same name as the corresponding OMG IDL interface.

7.6 Exceptions

Exceptions are a means of error handling. In the context of the OMG IDL, exceptions are supported by the language construct exception (see [OMG2017a], Chap. 7.4.3.4.2, p. 37). Defined exceptions are assigned to interface functions by using the keyword raises.

There are two basic ways to implement the exceptions that have been specified for the functions in this chapter:

– Some programming languages support exceptions as an explicit construct of the language. In erroneous situations, a function raises an exception at the point where an error is detected. Here, the program flow of the function is interrupted immediately and the exception is caught in the program code or by the code of the calling application. Therefore, the function does not return a return value.

– If no specialized language constructs are provided, exception handling is implemented by error codes. Here, the function exits its program flow when an error is detected by returning an appropriate error code as the return value.

For the translation of the SE API into ANSI C and Java this means:

In Java the concept of exceptions is implemented. OMG IDL exceptions are mapped to checked Java exceptions (see [ORACLE2017], Chap. 11.1, p 360). An OMG IDL exception is implemented by a Java class of the same name that extends the class java.lang.Exception. Java exceptions are assigned to an interface function by the key word throws. The following example shows the mapping of the exceptions ErrorIllegalDayValue and ErrorIllegalMonthValue defined in OMG IDL to Java. Regarding the Java code the corresponding exception classes have been defined (the definition itself is not shown in the code) and assigned to the relevant function.

OMG IDL

```
exception ErrorIllegalDayValue{};
exception ErrorIllegalMonthValue{};
 short saveTheDate(in short day, in short month, in short year)
 raises (ErrorIllegalDayValue, ErrorIllegalMonthValue);
```

Corresponding Java code

```
short saveTheDate(short day, short month, short year)
              throws ErrorIllegalDayValue,
ErrorIllegalMonthValue;
```

Text 13: Example for mapping of OMG IDL exception to Java

In ANSI C the concept of exceptions is not supported explicitly. Rather, it allows the implementation of an error handling. In this context, the functions return error codes as a return value to indicate that the execution of a function failed.
Error codes are implemented as constants in form of a pre-processor-directive. The name of a constant corresponds to the name of the relevant OMG IDL exception. Here, the UpperCamelCase notation of the OMG IDL exception name is transformed into a UPPER_CHARACTER_WITH_UNDERSCORES notation. The following text shows an example of this translation into ANSI C.

OMG IDL

```
exception ErrorIllegalDayValue{};
exception ErrorIllegalMonthValue{};
short saveTheDate(in short day, in short month, in short year)
 raises (ErrorIllegalDayValue, ErrorIllegalMonthValue);
```

Corresponding ANSI C code

```
#define EXECUTION_OK
#define ERROR_ILLEGAL_DAY_VALUE -20000
#define ERROR_ILLEGAL_MONTH_VALUE -20001
short saveTheDate(short int day, short int month,
                   short int year);
```

Text 14: Example for mapping of OMG IDL exception to ANSI C

7.7 Optional function parameters

```
     short export (in unsigned long transactionNumber,
                   in unsigned long clientID,
                   in unsigned long maximumNumberRecords,
                   out octet exportData []);

     short export (in unsigned long startTransactionNumber,
                   in unsigned long endTransactionNumber,
                   in unsigned long maximumNumberRecords,
                   out octet exportData []);

     short export (in DateTime startDate,
                   in DateTime endDate,
                   in unsigned long maximumNumberRecords,
                   out octet exportData []);

     short export (in unsigned long maximumNumberRecords,
                   out octet exportData []);
```

Text 15: Example for representing different expressions of a function

OMG IDL does not provide any construct for representing optional and conditional input and output parameters of a function.
Regarding this information a developer SHALL consider the appropriate definitions for the input and/or output parameters of the different functions of the SE API in chapter 4. If it is predictable that a function is called with certain combinations of input and/or output parameters, particular expressions of the function with appropriate parameter combinations are defined.

Text 15 shows the definition of the different expressions of the function export in OMG IDL (the exceptions are not represented).

7.8 Function input parameters

In OMG IDL function input parameters are defined by the keyword *in* (see [OMG2017a] , Chap. 7.4.3.4.3.3.1, p. 39 f.).

In ANSI C and Java input parameters of a primitive type are defined directly by replacing the appropriate OMG IDL types by the corresponding ANSI C and Java type respectively.

The following text shows an example of this translation.

OMG IDL specification

```
long calculateSum (in short summandOne, in short summandTwo,
 in boolean fastCalculation);
```

Corresponding Java code

```
int calculateSum (short summandOne, short summandTwo,
               boolean fastCalculation);
```

Corresponding ANSI C code

```
long int calculateSum (short int summandOne,
                        short int summandTwo,
                        bool fastCalculation);
```

Text 16: Example for representing OMG IDL input parameters in ANSI C and Java

OMG IDL input parameters in form of arrays are mapped to the particular array constructs in Java and ANSI C.

Regarding to Java, the definition of an appropriate input parameter follows the ordinary declaration of an array.

In the context of ANSI C, the length of the array is defined as an additional input parameter. The definition of this additional input parameter

– follows directly after the definition of the input parameter for the corresponding array,

– is of the ANSI C type unsigned long int and

– has the same name as the corresponding array plus the extension Length.

In ANSI C strings are represented in form of char arrays. Accordingly, the corresponding array length is also passed.

The Text 17 shows an example for the mapping of input parameters with the types of a byte array and a string respectively.

OMG IDL specification

```
short saveData (in octet inputData[], in string comment);
```

Corresponding Java code

```
short saveData (byte inputData[], String comment);
```

Corresponding ANSI C code

```
short int saveData(unsigned char *inputData,
unsigned long int inputDataLength,
unsigned char *comment,
              unsigned long int commentLength);
```

Text 17: Example for input parameters in form of arrays and strings

7.9 Function output parameters

In OMG IDL function output parameters are defined by the keyword *out* (see [OMG2017a] , Chap. 7.4.3.4.3.3.1, p. 39 f.).

To represent output parameters in Java, the data for the relevant function parameters must be passed in form of a call-by-reference. Accordingly, changes to a parameter value inside the function affect the original data. In Java only parameters of types that are specified by Java classes or interfaces can be passed as call-by-reference.

In Java parameter values of primitive number types (e. g. int) and the primitive type *boolean* as well as the type *String* can only be passed in form of call-by-value. In the context of call-by-value a copy of the parameter value is passed to a function. Accordingly, changes to a parameter value are only relevant in the context of the function and do not affect the original data.

To allow the definition of output parameters in Java for the above mentioned primitive types, appropriate holders in form of final[2] Java classes are specified. The name of such a holder class consists of the name of the type and the extension Holder. It has one private property named value that is of the particular type. The holder class provides a constructor with an input parameter for setting the value of the value property. To get the property value, the function getValue is defined. The function setValue is defined to set the property.

The example in Text 18 shows the definition regarding a holder class for the primitive Java type int.

In ANSI C output parameters can be specified by using appropriate pointers to the types of the corresponding parameters.

```
final public class IntHolder{

        public IntHolder(int newValue){
    value=newValue;
        }
        private int value;

        public int getValue(){
         return value;
        }

        public void setValue(int newValue){
         value=newValue;
        }
}
```

Text 18: Definition of a holder class for the primitive Java type int

Text 19 shows an example for the representation of OMG IDL output parameters of a primitive type in ANSI C and Java. Regarding the Java representation of the output parameter the holder class IntHolder from the previous example (see Text 18) is used.

2 A final Java class can not be extended by inheritance.

OMG IDL specification

```
short calculateSum (in short summandOne, in short summandTwo,
                    out long sum);
```

Corresponding Java code

```
short calculateSum (short summandOne, short summandTwo,
                   IntHolder sum);
```

Corresponding ANSI C code

```
short int calculateSum (short summandOne, short summandTwo,
                        long int *sum);
```

Text 19: Example for representing an output parameter of a primitive type in Java and ANSI C

In Java, arrays can *not* be passed by reference. Accordingly, an appropriate holder class is needed. The name of this class consists of the name of relevant type and the extension ArrayHolder. The property value of this class represents an array of the particular type.

In ANSI C, output parameters in form of arrays are defined by a double pointer of the relevant type. The additional output parameter regarding the length of the returned array is defined by a pointer of the type unsigned long int.

The Text 20 represents an example for the mapping of an OMG IDL output parameter in form of a byte array to Java and ANSI C.

OMG IDL specification

```
short getData(out octet outputData[]);
```

Corresponding Java code

```
short getData(ByteArrayHolder outputData);
```

Corresponding ANSI C code

```
short int getData(unsigned char **outputData,
                  unsigned long int *outputDataLength);
```

Text 20: Example for representing an OMG IDL output parameter in form of a byte array in Java and ANSI C

7.10 Return value

In OMG IDL the return value of a function is represented by the type of the function.

In ANSI C and Java return values are defined directly by replacing the appropriate OMG IDL types by the corresponding ANSI C and Java type respectively.

8 Appendix C: The TAR file format

The TAR file format allows it to combine multiple files into one. It also allows the analysis of the TAR file on all common Operating Systems (i.e. the TAR file can be easily unpacked and the content can be viewed).

This document references the format specified in [POSIX.1-1988].

A tar archive consists of a series of file objects. Figure 2 shows that the original information of each file that is contained in the tar archive stays unchanged.

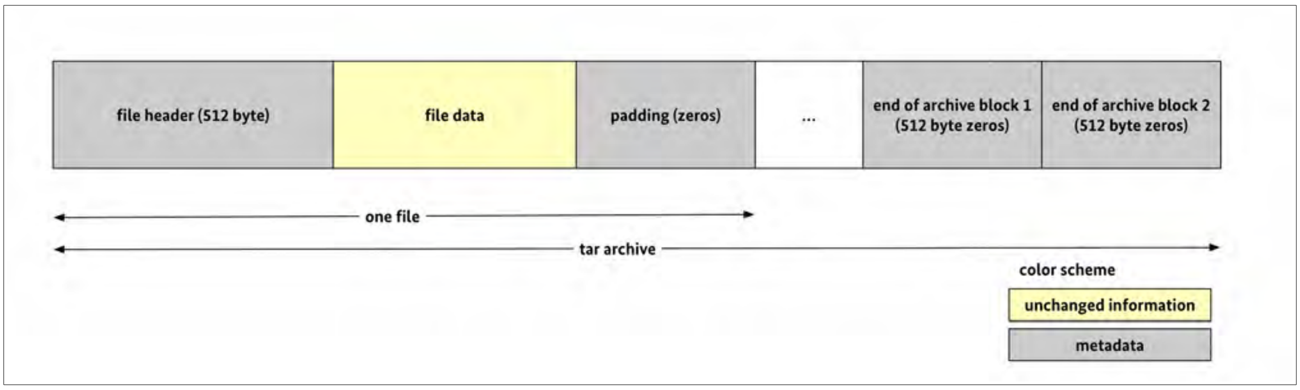

Figure 2: TAR file format

Each file object included in the tar archive is preceded by a 512-byte header record. The file data is written unaltered directly following the header, but is rounded up to a multiple of 512 bytes (denoted as padding in Figure 2). As most modern implementations use a padding of zeros, implementations following this document SHALL also use zeros for this padding.

The end of an archive is marked by at least two consecutive zero-filled records of 512 byte. The final block of an archive is padded out to full length with zeros.

The TAR standard as defined in [POSIX.1-1988] allows certain options for the headers that are explained and restricted in this chapter as follows:

1. In addition to entries describing archive members, an archive may contain entries which tar itself uses to store information. The implementation MAY use such entries but their use is not mandatory.
2. The file header of each file in the tar file SHALL be filled as explained in Table 34.

Header part	Data type	Description
name	char[100]	This field contains the name of the file, with directory names (if any) preceding the file name, separated by slashes. The implementation MUST not use any directory names, all files SHALL be stored in the root of the tar file.
mode	char[8]	The mode field provides nine bits specifying file permissions and three bits to specify the Set UID, Set GID, and Save Text (sticky) modes. This field MAY be used at the discretion of the developer.
uid	char[8]	The uid and gid fields are the numeric user and group ID of the file owners, respectively. If the operating system does not support numeric user or group IDs, these fields should be ignored. This field MAY be used at the discretion of the developer. If the developer does not use these fields for a certain purpose, the field SHOULD be set to ‘0’.
gid	char[8]	

Header part	Data type	Description
size	char[12]	The size field is the size of the following file in byte.
mtime	char[12]	The mtime field is the data modification time of the file at the time it was archived. It is the ASCII representation of the octal value of the last time the file's contents were modified, represented as an integer number of seconds since January 1, 1970, 00:00 Coordinated Universal Time. The API SHALL use the date and time of the tar archive creation for this entry.
chksum	char[8]	The chksum field is the ASCII representation of the octal value of the simple sum of all bytes in the header block. Each 8-bit byte in the header is added to an unsigned integer, initialized to zero, the precision of which shall be no less than seventeen bits. When calculating the checksum, the chksum field is treated as if it were all blanks.
typeflag	char	The typeflag field specifies the type of file archived. If a particular implementation does not recognize or permit the specified type, the file will be extracted as if it were a regular file. The implementation SHALL set this character to ‘0’ which represents a regular file. The implementation MUST not use any other file types.
linkname	char[100]	This entry is used if the typeflag is set to ‘1’ (link). As the implementation MUST not use links, this entry SHALL remain empty.
magic	char[6]	The magic field indicates whether this archive was output in the P1003 ([POSIX.1-1988]) archive format. To indicate that the generated tar file shall be compliant to [POSIX.1-1988], the implementation SHALL set this field to ‘ustar’.
version	char[2]	This field shall be set to ‘00’ (indicating a standard POSIX archive).
uname	char[32]	This field can be used to set the user name of the file. Please note that this value might be ignored if this functionality is not supported by the Operating System under which the tar file is unpacked.
gname	char[32]	This field can be used to set the group name of the file. Please note that this value might be ignored if this functionality is not supported by the Operating System under which the tar file is unpacked.
devmajor	char[8]	As the implementation shall only use regular files within the tar file, this field SHALL stay empty.
devminor	char[8]	As the implementation shall only use regular files within the tar file, this field SHALL stay empty.
prefix	char[155]	First part of pathname. If the pathname is too long to fit in the 100 bytes provided by the standard format, it can be split at any / character with the first portion going here. If the prefix field is not empty, the reader will prepend the prefix value and a / character to the regular name field to obtain the full path- name. As the implementation MUST NOT use any path names for the files in the tar archive, this field MUST remain empty.

Table 34: Tar file members header

References

OMG2017a	OMG: Interface Definition Language, Version 4.1, 2017
SPECZIP	BSI: SE API definition in OMG IDL, ANSI C and Java
RFC2119	S. Bradner: Key words for use in RFCs to Indicate Requirement Levels
BSI TR-03116	BSI: Technische Richtlinie TR-03116 Kryptographische Vorgaben für Projekte der Bundesregierung - Teil 5: Anwendungen der Secure Element API
POSIX.1-1988	The Open Group: POSIX.1-1988 -Portable Operating System Interface, 1988
ISO7816-8	ISO: ISO 7816-8 Identification cards — Integrated circuit cards —Part 8: Commands for security operations, Part 8, 2016
OMGx	OMG: IDL to Java Language Mapping, Version 1.3, 2008
OMG1999	OMG: C Language Mapping Specification, 1999
ORACLE2017	James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith: The Java® LanguageSpecificationJava SE, 9 Edition, 2017
ANSI99	ANSI, ISO: ISO/IEC 9899:1999, ANSI C, 1999