Bundesamt
für Sicherheit in der
Informationstechnik

Deutschland
Digital•Sicher•BSI•

BSI TR-03151-2 Appendix ANSI C

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

Part 2: Interface Mapping
Appendix ANSI C – Mapping of API definitions to ANSI C

Version 1.1.0
2023-02-13

Version history

Version	Date	Description
1.1.0	2023-02-13	Initial version

Table 1 Version history

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

1 Introduction

This document is an appendix to Technical Guideline [BSI TR-03151-2] which describes the programming language specific mapping of an API definition to ANSI C. The API definition is not part of this appendix.

Within [BSI TR-03151-2] certain concepts and constructs of the API are defined. Here, these constructs are mapped to ANSI C. Regarding the data types the approach is based on the OMG IDL mappings to ANSI C that are provided in [OMG2017a] 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 mapping used in [BSI TR-03151-2]).

2 Mapping of the API definition to ANSI C

The following sub-chapters contain the mapping of API constructs to ANSI C, including the mapping of data types, exceptions/error conditions and function parameters.

2.1 Mapping of data types

Data types are used to store values and are usually manipulated during the execution of functions. There are basic types, that usually only contain one value at a time and more complex ones which may contain multiple values. This sub-chapter introduces the mapping of OMG IDL data types to ANSI C.

2.1.1 Mapping of basic types

Table 2 contains the mapping of basic OMG IDL data types to ANSI C data types. 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 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	Comment
short $(-2^{15} \dots 2^{15}- 1)$	short int $(-(2^{15} - 1) \dots 2^{15} - 1)$	Common implementations provide a value range of $(-(2^{15}) \dots 2^{15} - 1)$
long $(-2^{31} \dots 2^{31}- 1)$	long int $(-(2^{31} - 1) \dots 2^{31} - 1)$	Common implementations provide a value range of $(-(2^{31}) \dots 2^{31} - 1)$
long long $(-2^{63} \dots 2^{63}- 1)$	long long int $(-(2^{63} - 1) \dots 2^{63} - 1)$	Common implementations provide a value range of $(-(2^{63}) \dots 2^{63}- 1)$
unsigned short $(0 \dots 2^{16}-1)$	unsigned short int $(0 \dots 2^{16}-1)$	Common implementations provide a value range of $0 \dots 2^{16}-1$
unsigned long $(0 \dots 2^{32}-1)$	unsigned long int $(0 \dots 2^{32}-1)$	Common implementations provide a value range of $0 \dots 2^{32}-1$
unsigned long long $(0 \dots 2^{64} -1)$	unsigned long long int $(0 \dots 2^{64} - 1)$	Common implementations provide a value range of $0 \dots 2^{64} – 1$

OMG IDL	ANSI C	Comment
octet (see [OMG2017a], Chap. 7.4.1.4.4.1.1.6, p. 27)	unsigned char	
boolean (see [OMG2017a], Chap. 7.4.1.4.4.1.1.5, p. 27)	_Bool (see [ANSI99], Chap. 6.2.5, p. 33).	ANSI C provides the header file <stdbool.h> (c.f. [ANSI99], Chap. 7.16, p. 252) that enables the use of the identifier bool for the type _Bool. In the current mapping context the specifier bool is used.
Native type DateTime	Presentation by the structure tm from the header file time.h. (cf. [ANSI99], Chap. 7.23.1, p. 337)	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 2 Mapping of data types

2.1.2 Mapping of complex types

2.1.2.1 Mapping of strings

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).

In the mapping context, it is possible to encounter empty strings. An empty string in ANSI C is a character array which contains only the null terminator (\0). For string literals, a null character typically is inserted by the C compiler to denote the end of a string, thus starting a char array with it means the array is empty.

Text 1 shows how OMG IDL strings are mapped to C.

OMG IDL

```
string<100> textWithBounds
string textWithoutBounds
string emptyText = ""
```

Corresponding ANSI C Code

```
unsigned char *textWithBounds
unsigned char *textWithoutBounds
unsigned char *emptyText = ""
```

Text 1: Example for mapping of OMG IDL strings to ANSI C

Note that within this particular context of mapping API functions the OMG IDL type “string” (text) and the type “octet[]” (binary data) are both represented by the C type “unsigned char[]” i.e. the pointer equivalent

“unsigned char*”. While the end of a string can be derived from the null terminator the end of binary data cannot be derived. Therefore an additional parameter representing the legth of the data is used.

If an input parameter of a function is of the type “unsigned char*”, an additional input parameter representing the length of the data is passed (see chapter 2.3.1). If an output parameter of a function is of the type “unsigned char*”, an additional output parameter representing the length of the data is returned (see chapter 2.3.2).

For binary data the length parameter shall represent the number of “unsigned char” elements that make up the data. For text the length parameter shall represent the number of “unsigned char” elements that make up the text including the null terminator, which is the same value that the C standard function “strlen” would return increased by one.

2.1.2.2 Mapping of enumerations

OMG IDL enumerations are represented by ANSI C enumerations (see [ANSI99], Chap. 6.7.2.2). The names of the enumeration elements are prefixed with the name of the enumeration followed by an underscore to avoid naming conflicts.

Text 2 shows an example for the mapping of an OMG IDL enumeration to ANSI C.

OMG IDL

```
enum Color {red, green, blue, orange};
enum Fruit {apple, orange};
```

Corresponding ANSI C code

```
enum Color {Color_red, Color_green, Color_blue, Color_orange};
enum Fruit {Fruit_apple, Fruit_orange};
```

Text 2 Example for mapping of OMG IDL enumerations to ANSI C

2.1.2.3 Mapping of arrays

OMG IDL arrays are represented by arrays in C (see [ANSI99], Chap. 6.7.5.2, p. 116 f.). The data type that is used for the array definition is based on the mapping in chapter 2.1.1.

Text 3below shows how an octet array is mapped to ANSI C. Observe how ANSI C requires the length of the array to be passed as a separate parameter.

OMG IDL as used in the Technical Guideline

```
octet variableExample[]
```

Corresponding ANSI C code

```
unsigned char *variableExample, unsigned long int variableExampleLength
```

Text 3 Example of an array in the API definition and ANSI C

2.1.2.4 Definition context

All function signatures in the context of the API mapping are definied within a header file to be included and implemented in a corresponding source code file.

2.2 Mapping of exceptions

In ANSI C the concept of exceptions is not supported explicitly. Rather, it allows the implementation of an error handling by using error codes. In this mapping context, the functions return error codes as a return value of type short int to indicate that the execution of a function failed. In contrast the return value EXECUTION_OK is used to indicate a successful execution of a function.

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 exception. An UpperCamelCase notation of the API definition is transformed into a UPPER_CHARACTER_WITH_UNDERSCORES notation.

The following text shows an example of this translation into ANSI C.

Properties of a function to be implemented

Function name: saveTheDate

Input parameter 1: short day

Input parameter 2: short month

Possible exception 1: ErrorIllegalDayValue

Possible exception 2: ErrorIllegalMonthValue

Corresponding ANSI C code

```
#define EXECUTION_OK 0
#define ERROR_ILLEGAL_DAY_VALUE -20000
#define ERROR_ILLEGAL_MONTH_VALUE -20001
short int saveTheDate(short int day, short int month);
```

Text 4 Example for mapping of exceptions to ANSI C

2.2.1 Mapping exceptions with additional information

Since the concept of exceptions is realised using error codes in ANSI C, it is not possible to simply add information to an existing error code.

In the case additional information is necessary to be conveyed in an exception event, it is stated in the detailed description of a function. The additional information might be retrieved by using the function “getLastFunctionCallStatus”. This function shall return the same value as the most recent call to the API. That might be either EXECUTION_OK or an error code. It provides additional information as output parameters if available.

Text 5 below shows ErrorIncorrectPin as an example for an exception with an additional information. For futher information about the “length” and “limit” parameters compare chapter 2.1.2.1 and 2.3.4.

C code excerpt for using the function getLastFunctionCallStatus

```
// Call a function which throws an error. In this example passing a wrong PIN
// is assumed and thererfore the exception ERROR_INCORRECT_PIN is thrown.

authenticateUser("user123", 8, "1234", 5);



// Calling the function getLastFunctionCallStatus will return the same error code
// ERROR_INCORRECT_PIN again.

short int getLastFunctionCallStatus(unsigned long int errorDataLimit,
        unsigned char *errorData,
        unsigned long int *errorDataLength);



// In the next example the error code ERROR_INCORRECT_PIN is also returned.
// This time the PIN retry counter is returned via an output parameter

unsigned long int dataLength;
unsigned char *data = malloc(sizeof(unsigned char) * 5);
getLastFunctionCallStatus(5, data, &dataLength);

// data now contains the PIN retry counter as text
// dataLength now contains the length of the text
```

Text 5: Code excerpt for using the function getLastFunctionCallStats which outputs the number of remaining retries after a previously unsuccessful authentication attempt

The output parameter errorDataLength might be used by the caller to check if additional information is actually available. In case the additional information is a numeric value it shall be converted to a textual representation.

2.3 Mapping of function parameters

2.3.1 Mapping of function input parameters

When mapping input parameters of a primitive type to ANSI C, the appropriate OMG IDL types are replaced with the corresponding ANSI C types.

The following text shows an example of this translation.

Properties of a function to be implemented

Function name: addKeyValue

Input parameter 1: short key

Input parameter 2: long value

Corresponding ANSI C code

```
short int addKeyValue(short int key, long int value);
```

Text 6: Example for input parameters in ANSI C

Input parameters in form of OMG IDL arrays are mapped to the particular array constructs in ANSI C, with the length of the array 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[1] and
- has the same name as the corresponding array plus the extension “Length”.

In ANSI C strings and octet arrays are represented in form of char arrays. Accordingly, the corresponding array length including the null terminator is also passed as described in chapter 2.1.2.1. To denote the parameter as an input parameter the keyword “const” is added, as per definition it is never to be changed within the API function or used as an output paramter.

Text 7 shows an example for the mapping of input parameters with the types of a byte array and a string respectively.

Properties of a function to be implemented

Function name: saveData

Input parameter 1: octet inputData[]

Input parameter 2: string comment

Corresponding ANSI C code

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

Text 7 Example for input parameters in form of arrays and strings

2.3.2 Mapping of function output parameters

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

Text 8 shows an example for the mapping of output parameters of an OMG IDL basic type to ANSI C

[1] In cases where large input data is expected the type unsigned long long int is used to allow larger data inputs. The applicability of actually using larger inputs might depend on the implementation and the available memory.

Properties of a function to be implemented

Function name: calculateSum

Input parameter 1: short summandOne

Input parameter 2: short summandTwo

Output parameter: long sum

Corresponding ANSI C code

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

Text 8: Example for representing an output parameter of a primitive type in ANSI C

Output parameters require memory to be allocated to allow the function to fill the allocated memory with the output value. This applies to all output parameters but especially to array parameters. The memory allocation (malloc) and deallocation (free) are left to the caller functions. Within the specific function it is necessary to know the size of the available memory that is passed to the function to prevent buffer overflows. Therefore an additional input parameter is added containing the available size of the passed array. The definition of this additional input parameter:

- is right before the definition of the output parameter containing a pointer to the memory space,
- is of the ANSI C type unsigned long int[2],
- has the same name as the corresponding array plus the extension “Limit”.

The function fills the array with data to be passed back to the caller. As the array size might be larger than the data that is actually returned an additional output parameter denotes the length of the array that is filled with data.

Text 9 below shows how an output parameter is mapped to ANSI C.

Properties of a function to be implemented

Function name: getData

Output paramter: octet outputData[]

Corresponding ANSI C code

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

Text 9: Example for representing an OMG IDL output parameter in form of a byte array in ANSI C

The necessary amount of memory to be allocated might be unknown prior to a function call. Therefore the provided amount might not be enough to hold all output data (e.g. outputDataLength > outputDataLimit). Such an error is indicated by returning the error code MEMORY_ERROR_LIMIT_TOO_LOW which means the function call shall be repeated with a larger limit. In this case the function shall also set the “Length” output parameter to indicate the amount of data the caller has to expect and therefore allocate.

[2] In cases where large ouput data is expected the type unsigned long long int is used to allow larger exports. The applicability of actually using a larger output might depend on the implementation and the available memory.

Note that output parameters which feature an “offset” in addition to a “limit” parameter never cause the return of the error code MEMORY_ERROR_LIMIT_TOO_LOW (see chapter 2.3.4).

2.3.3 Mapping of optional and conditional function parameters

Usually, ANSI C does not use optional parameters and rarely supports them. Furthermore, the concept of overloading a function, meaning the declaration of functions with the same name and output type but different input parameters, does not exist in ANSI C. Therefore, functions with optional parameters are mapped to several functions of similar but slightly differing names in ANSI C.

In case of optional parameters functions with and without the optional parameter are offered. Text 10 below shows an example for a function with an optional input parameter.

Properties of a function to be implemented

```
Function name: exampleFunc
Input parameter 1: string requiredParam
Input parameter 2 (optional): boolean optionalParam
```

Corresponding ANSI C code

```
short int exampleFuncOptionalParam( unsigned char *requiredParam
                        unsigned long int requiredParamLength,
                        bool optionalParam);
short int exampleFunc( unsigned char *requiredParam
                        unsigned long int requiredParamLength);
```

Text 10: Example for the handling of functions with an optional input parameter

In case of conditional parameters a null pointer (macro NULL) is used when calling the function to indicate an optional parameter is not set. Text 11 below shows an example for a function with two conditional input parameters.

Properties of a function to be implemented

Function name: doesUserExist

Input parameter 1 (conditional): string firstname

Input parameter 2 (conditional): string lastname

Output parameter: boolean exists

Condition: at least one of firstname or lastname is present

Corresponding ANSI C code

```
bool doesUserExist( unsigned char *firstname,
                    unsigned long int firstnameLength,
                    unsigned char *lastname,
                    unsigned long int lastnameLength);
```

Possible C calls of the method

```
doesUserExist("John", 5, "Doe", 4);
doesUserExist("John", 5, NULL, 0);
doesUserExist(NULL, 0, "Doe", 4);
```

Illegal C calls of the method

```
doesUserExist(NULL, 0, NULL, 0);
```

Text 11: Example for the handling of functions with two conditional input parameters

The specific implementation shall therefore consider that a passed value might be a null pointer and appropriate checks should be applied.

Some input parameters might have a specific value which also indicates optionality. For example the value "0" for an input parameter of type integer may have the same semantics as not having the parameter at all. Please consider the description of the parameter in the referring Technical Guideline for further information. In such cases not every possible combination of function signatures with and without that optional parameter might be represented. This is in contrast to the previous definition of optional parameters and aims to keep the number of copies for a single function to a reasonable level.

2.3.4 Mapping of large data function parameters

It might be necessary for functions to return a large amount of data in form of a byte array. This especially applies to functions which export data from a device. This might pose a challenge for devices with limited memory. Such functions therefore offer an additional input parameter to specify an offset which might be used in conjunction with a limit (cmp. chapter 2.3.2) to allow for chunked data export. The definition of this additional input parameter:

- Is right in front of the definition of the "Limit" parameter,
- is of the ANSI C type unsigned long int,
- has the same name as the corresponding array plus the extension "Offset".

By calling the same API function with an increasing offset, available data might be retrieved and processed (e.g. stored) in increments until all available data has been retrieved.

Text 12 below shows an example for a function which utilises a chunked export.

Properties of a function to be implemented

Function name: exportData

Output parameter: octet data[]

Corresponding ANSI C Code

```
short int exportData(unsigned long long int dataOffset,
                     unsigned long long int dataLimit,
                     unsigned char *data,
                     unsigned long long int dataLength);
```

Possible C call of the function

```
// in this example 75 bytes are to be exported (values: 1, 2, 3, ..., 75)
unsigned char *buffer = malloc(sizeof(unsigned char) * 50);
unsigned long long int dataLength;

// start at offset 0 to retrieve the first 50 values (1, 2, ..., 50)
// dataLength returned by the function is 50 which is the same as the limit so more data
// might be available
exportData(0, 50, buffer, &dataLength);
// ... do something with the data ...

// start at offset 50 (previous offset 0 + limit of 50) to retrieve the next 50 values (51, ..., 75)
// Unknown to the caller only 25 values are left so the dataLength is 25 indicating
// no futher calls to the function are necessary and all data has been retrieved[3]
exportData(50, 50, buffer, &dataLength);
// ... do something with the data ...
```

Text 12: Example for the handling of functions using a chunked export

The offset is applied to the data array as if it would be exported without any offset in place. All values before the offset position are discarded and not exported. As long as the returned length is not less than the passed limit the caller has to assume more data is available and futher calls to the function are necessary.

Note that output parameters which feature an “offset” in addition to a “limit” parameter never cause a return of the error code MEMORY_ERROR_LIMIT_TOO_LOW (see chapter 2.3.2). If the output data does not fit the provided “limit” only the amount of data which fits within the “limit” is returned and the “offset” shall be used in a consecutive call to retrieve the remaining data.

By using a large value for the limit all data might be exported without the need to call the function multiple times.

[3] In the rare case the complete length of the data to be exported is a multiple of the specified limit the last call to the function will return a length of zero.

References

ANSI99	ANSI, ISO: ISO/IEC 9899:1999, ANSI C, 1999
BSI TR-03151-2	BSI: Technical Guideline BSI TR-03151-2: Secure Element API (SE API) – Part 2: Interface Mapping
OMG1999	OMG: C Language Mapping Specification, 1999
OMG2017a	OMG: Interface Definition Language, Version 4.1, 2017