End-to-End Protection Wrapper Generator

User Manual

Version:	2.0.1
Date:	13.03.2015
Document number:	D-MSP-G-70-001

TTTech Automotive GmbH
Schoenbrunner Str. 7, A-1040 Vienna, Austria, Tel. + 43 1 585 34 34-0, Fax +43 1 585 34 34-90, support@tttech-automotive.com

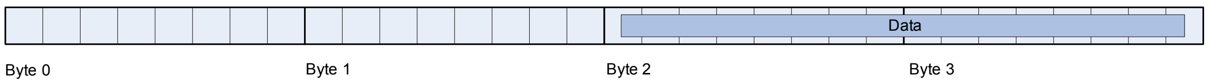
Table of Contents

1 Introduction

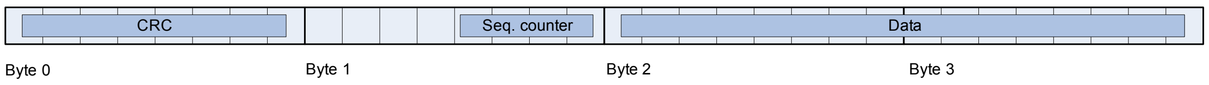
Many automotive applications are distributed among several **electronic control units (ECU)** and include communication via embedded networks. The exchanged data is often critical (for example, car speed or steering angle), and incorrect data could endanger both, the driver and the car. Therefore, special mechanisms have been introduced to prevent the processing of incorrect data. One of them is the **End-to-End Communication Protection Library (E2Elib)**, which has been standardized in AUTOSAR [AS_E2E_SWS] 68.

Most automotive networks protect data with checksums. These mechanisms are not adequate for protecting the application data, because errors in gateways or software layers within the ECU could destroy the data before and after it is transferred over the network. End-to-end protection also covers this path. The E2Elib uses an additional checksum and a sequence counter in order to detect false and missing data directly in the application.

The figure below shows an **I-PDU** with a length of four bytes and a **signal** with two bytes:



The **end-to-end communication protection** requires additional signals in the protected signal area for the checksum and the sequence counter:



For details about this mechanism, see the *AUTOSAR E2Elib Specification* [AS_E2E_SWS] 68 and the communication protection specification of the original equipment manufacturer.

1.1 E2E Protection Wrapper Generator

Applications using the **E2Elib** or similar communication protection mechanisms have one major problem: the E2E library protection routines need the I-PDU representation to apply protection mechanisms.

Therefore, the application would have to provide the DE in that I-PDU representation, which means that the signals of the DE must be marshaled using information the application normally does not have: byte order, bit length and start bit position of each signal, and the length of the I-PDU.

To write CPU-independent code and reuse application functions for more than one customer (OEM), it is necessary to have a software layer to deal with this. Usually, a so-called COM layer is used. The application of the E2E-Lib must create a similar layer beside or above COM. That is, the application must implement parts of the COM layer again. The application must know details of the communication system that can normally be accessed only in lower layers.

The **E2E Protection Wrapper (E2EPW)** provides a layer that circumvents this problem. The protection wrapper is generated code. That code consists of copy routines which know about the I-PDU layout and contain calls to the E2Elib routines. For transmission, the application passes the DE to the wrapper instead of the RTE. The wrapper then builds the I-PDU representation, invokes the E2E library function, and then the RTE function. For reception, the application calls the wrapper instead of the RTE. The wrapper receives the DE from the RTE and invokes the E2E library function before returning the DE.

According to the *AUTOSAR E2Elib Specification* [AS_E2E_SWS] 68, the **E2E Protection Wrapper Generator (E2EPWG)** generates the code for the protection wrapper from a specific **E2EConfig file**.

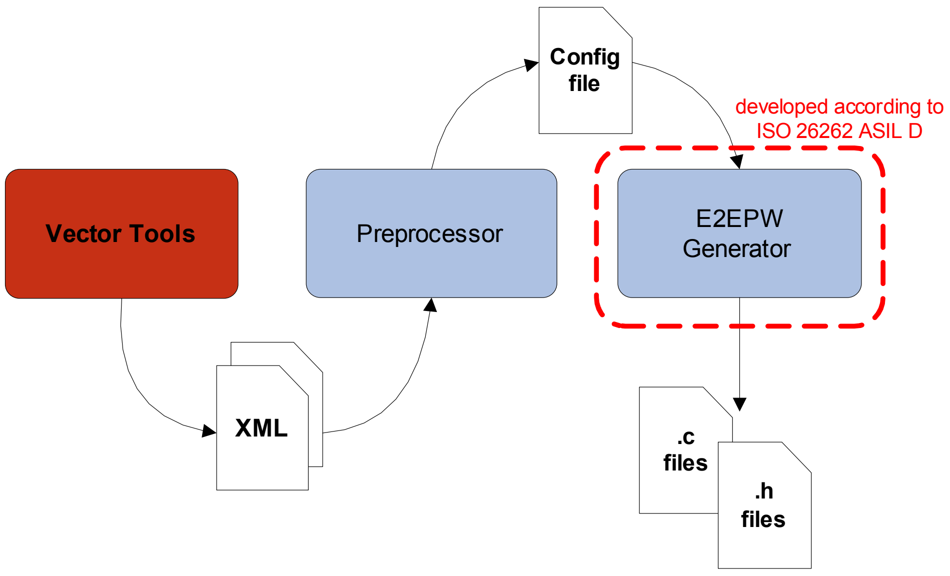
Note: This user manual does not cover safety-related topics. For safety-critical projects that need to fulfill ISO 26262 requirements, refer to the *End-to-End Protection Wrapper Safety Manual* [TT_E2EPW_SM] 68.

1.2 Tools Integration

This document describes how to use the **E2EPWG** developed by TTTech Automotive.

Example (integration into the Vector tools environment):

In this environment, a special preprocessor is available. This preprocessor converts the XML files produced by the Vector tools to a **E2EConfig file** that can be used as an input for the **E2EPWG**.



Overview of the E2EPW Generator and Preprocessor

1.3 Use Cases

End-to-end protection is a technique that has been used in safety-relevant distributed applications, in the automotive and other industrial sectors for quite some time. The **E2EPW** is intended for use in all areas where **checksum (CRC)** and **sequence counter (SC)** are used in the data area for **end-to-end communication protection**. However, the way data is exchanged differs slightly in different scenarios. The defined use cases are:

- AUTOSAR E2E Protection Wrapper with RTE for data exchange (use case **AUTOSAR RTE**)
- AUTOSAR E2E Protection Wrapper without RTE (use case **AUTOSAR NO**

RTE)

- J1939 CAN stack with an E2E Protection Wrapper using the transport layer (use case `J1939 CAN`)

Currently, only the use cases **AUTOSAR RTE** and **AUTOSAR NO RTE** are supported.

This user manual gives a detailed description of use case **AUTOSAR RTE** only.

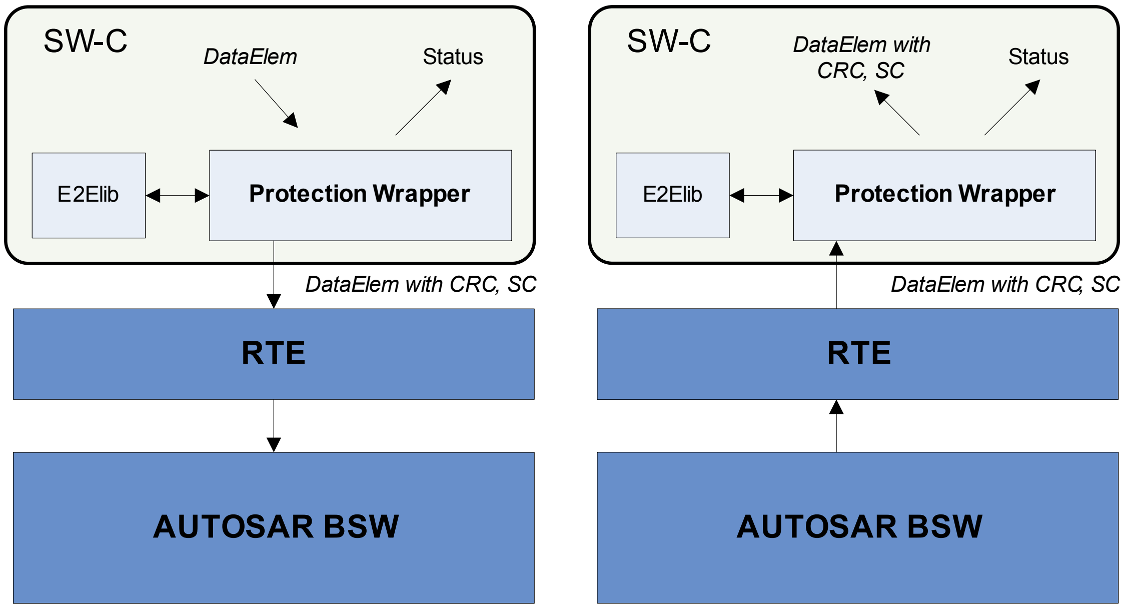
Overview of Use Case AUTOSAR RTE

In this use case, the **SW-C** sends/receives `DataElement` (**complex data element** of AUTOSAR) data structures, consisting of fields for signals and a special CRC and SC signal. The **Protection Wrapper** is called by the **SW-C**.

At **transmission** (see **left** figure below), the Protection Wrapper calculates the CRC and SC and passes the `DataElement` to the RTE.

At **reception** (see **right** figure below), the Protection Wrapper checks the CRC and SC in the received `DataElement` and passes it to the SW-C.

The status provides additional information, such as error codes and the number of lost DataElements.



Overview of Use Case AUTOSAR RTE

2 Versions

This User Manual describes

- the **Preprocessor Version** 2.0.2 and
- the **Protection Wrapper Generator Version** 2.0.1.

3 Installation

The **Preprocessor** and the **E2EPWG** are Windows console applications. There is no special installation procedure for them.

If the **E2EPW** is shipped together with Vector BSW, the files

- `E2EPW_MemMap.inc` and
- `E2EPW_Compiler_Cfg.inc`

are already incorporated into the Vector BSW.

Otherwise the files `E2EPW_Compiler_Cfg.inc` and `E2EPW_MemMap.inc` are also shipped with the **Preprocessor** and the **E2EPWG** and are example files that must be adapted to the actual build environment and requirements according to the *AUTOSAR Specification of Compiler Abstraction* [AS_COMABS_SWS] 68 and the *AUTOSAR Specification of Memory Mapping* [AS_MEM_SWS] 68 .

For tool integration, the Preprocessor can be invoked using the command line parameters. Please refer to the respective tool manual for how to do this.

Note: If the Preprocessor and the E2EPWG are part of a software package, they may be installed and integrated transparently. The content of `E2EPW_Compiler_Cfg.inc` and `E2EPW_MemMap.inc` may then be already included in the corresponding `Compiler_Cfg.h` and `MemMap.h` files of the software package.

The Preprocessor is built with `py2exe` and includes **Python 2.7.3** and **lxml 2.2.8**.

The corresponding software licenses for these open source packages are contained in the file `LICENSE`, which is bundled with the Preprocessor executable.

The following **DLLs** must be available in the system:

- `USER32.dll`
- `SHELL32.dll`
- `WSOCK32.dll`
- `ADVAPI32.dll`
- `WS2_32.dll`
- `KERNEL32.dll`
- `MSVCR90.dll`

  Note: If this DLL is not available, it must be installed manually. The installer for this DLL is availbale from http://www.microsoft.com/downloads/details.aspx?FamilyID=9b2da534-3e03-4391-8a4d-074b9f2bc1bf&displaylang=en

4 Preprocessor

The Preprocessor takes the **ECU information** of the system description and the **SW-C design** information as an input. The output of the Preprocessor is a **E2EConfig file** (in a specific format) that can be interpreted by the **E2EPWG** in order to generate the protection wrapper code.

The behavior of the preprocessor is controlled with command line parameters[8].

4.1 Preprocessor Help

The behavior of the Preprocessor is controlled with command line parameters.

Calling the preprocessor with the command line parameter `-h` gives the following output:

```
E2EPWG PreProcessor v2.0.2
Usage: pwg_preprocessor.exe [options] ProjectName [SystemDescriptionComm]
SystemDescription OutputDir

Options:
  -h, --help            show this help message and exit
  -b BYTEORDER, --cpu-byte-order=BYTEORDER
                        Set the cpu-byte-order to BYTEORDER. Valid values are:
                        BIG_ENDIAN, LITTLE_ENDIAN, HIGH_BYTE_FIRST,
                        LOW_BYTE_FIRST.
  -e FILE, --e2epwg=FILE
                        Execute E2EPWG using the config file created by the
                        PreProcessor.
  -v LEVEL, --verbose=LEVEL
                        Output extended information (with LEVEL verbosity
                        level). Levels above 1 are useful for debugging.
  -f, --error_format    Enables the Vector tool error format.
  -E FILE, --ecuc=FILE  Additional ECU Description File containing byte-order
                        data.
```

4.1.1 Using the Preprocessor

To run the **Preprocessor**, enter the following **command** in a command prompt window:

```
pwg_preprocessor.exe [<options>] <ProjectName>
[<SystemDescriptionComm>] <SystemDescription> <OutputDir>
```

The following table describes the **mandatory** command line **parameters**:

Name	**Meaning**
<ProjectName>	This is an arbitrary name. This parameter is used as the file **name** of the **generated configuration** file.
[<SystemDescriptionComm>]	This parameter specifies the **location** and **name** of the (ECU extract of the) **system description file**. It contains the communication part of the system description, e.g., the signals and their corresponding mapping to I-PDUs. This file is **optional** for **AUTOSAR 3** formats, because it is only necessary if the information is not already contained in `SystemDescription`, which is always the case for AUTOSAR 4 formats.
<SystemDescription>	This parameter specifies the **location** and **name** of the **system description** file (application part). This file contains the design of the software components together with all the tasks, events, ports and (variable) data prototypes. It can also contain signals, I-PDUs and mappings between signals and I-PDUs. For AUTOSAR 4 formats, this information is mandatory. The ECU file is an XML file with a format that conforms to the meta model of the AUTOSAR specification versions 3.1.4, 3.2.1, 3.2.2, 4.0.3, 4.1.2, or 4.1.3. The file is generated by the Vector tools. The file extension is `.arxml`.
<OutputDir>	This parameter specifies the **path** where the **generated configuration** file will be stored.

The following table describes the **optional** command line **parameters**:

Name	**Meaning**
`-h`	Shows the help message 8.

`-b BYTEORDER,`
`--cpu-byte-order=BYTEORDER`
	This parameter specifies the **value** that must be assigned to the **attribute** `Byte_Order_CPU` of the generated **E2EConfig file**.
	The CPU byte order must be provided, either by using command line parameter -b or -E.
	Possible values:
	▪ `BIG_ENDIAN (=HIGH_BYTE_FIRST),`

Name	Meaning
	▪ `LITTLE_ENDIAN (=LOW_BYTE_FIRST)`.
`-e FILE, --e2epwg=FILE`	This parameter specifies the location of the **E2EPWG**. When this parameter is specified, the generator is invoked after the Preprocessor has successfully generated the **E2EConfig file**. The command line option `-e` accepts a path to an executable file (the E2EPWG) and calls this file after configuration generation as follows: `<FILE> <OutputDir>\<EcuProjectName>.cfg <OutputDir>` where ▪ `<FILE>` is the **executable** file *`FILE`*, ▪ `<OutputDir>` is the *`OutputDir`* parameter provided to the Preprocessor and ▪ `<EcuProjectName>` is the *`EcuProjectName`* parameter provided to the Preprocessor.

`-v LEVEL,`
`--verbose=LEVEL`
	This parameter is used for **detailed error reporting**. The higher the verbosity level, the more detailed is the report.
	Possible values:
	▪ Levels **above 1** are useful for **debugging**. The **default** value is **0** (a short error notice is written to `stderr`).
	▪ For **level 1**, the context of the action that failed is also printed to `stderr`. If the error occurred during the parsing of the XML input data, the location within the XML file (line number) is provided.
	▪ For **level 2**, the XPATH query that failed is also printed. If the XPATH query is not based on the root node, the location within the XML document, from where the XPATH query was started, is also provided.

`-f,`
`--error_format`
	This parameter enables the error format needed by the Vector tools to parse the output messages of the Preprocessor.

`-E,`
`--ecuc`
	This parameter is used to provide the ECUC file, which contains the definition of the CPU byte order. If this option is not used, the byte order must be provided by using the `-b` parameter.

Note: Given the call parameters above, the **E2EPWG** will put all the generated files into the *`<OutputDir>`* directory. If a file created by the **E2EPWG** exists already, e.g., from a previous run, the generator will output an error message and exit. It is therefore recommended to provide an empty, but existing *`<OutputDir>`* directory to the Preprocessor when the command line option `-e` is used.

4.1.2 Behavior and Log Output

The overall behavior of the Preprocessor was changed for version 1.6.1, 2.0.1 and onward. Previously, the Preprocessor would have aborted with an error message if the resulting E2EConfig file had not been complete/valid regarding the provided ARXML input.

The new behavior performs a best-effort strategy to provide as much content derived from the input ARXML as possible, even if the result is only partial and not a valid E2EConfig file. This has several implications:

- Almost all error messages are now reported as warning. This is mainly due to the default behavior of the default tool chain (Vector DaVinci), which discards all generated output if an error is reported.
- Warnings may build up and result in causally linked warning messages.
- The overall amount of log output messages increases drastically as the preprocessor continues to process the input ARXML, where it previously would have simply aborted with a single error message.
- The generated E2EConfig file may not be valid and result in an error message of the E2EPW Generator. Manual adaptions of the E2EConfig file or of the ARXML input may be necessary to get a valid E2EConfig file. Hints are provided in [Warning] and [Info] messages.
- EndToEndProtections may be skipped due to errors and a warning message is provided. There will not be a corresponding Protected_Area defined in the eventually generated E2EConfig file.

In general, log output is written to stderr.

4.1.3 Log Message Format

The format of log output messages is aligned to the format specification of the Vector DaVinci tools.

The format is as follows:

```
[<Severity>] E2E<ID> - <Summary>
- <Description>
```

Where *`<Severity>`* is one of the following:

- `[Info]` - For information only, guiding through the process or indicating what the Preprocessor currently does.
- `[Warning]` - Depending on the particular message, this can be a simple indication as "that there might be a problem" as well as a severe warning like "the output file is incomplete / cannot be used as is".

- `[Error]` - For hard errors where no output can be created. Typically, these are run-time errors, e.g. input file missing, cannot write to file, or unrecoverable parsing errors.

`<ID>` is a five-digit number indicating the message type.

`<Summary>` is optional and provides a short summary of the particular message.

`<Description>` provides detailed information about the cause of the message.

4.1.4 Warning and Info Log Messages

The Preprocessor provides helpful log output, which should always be checked for [Warning] and [Info] messages.

Some of the [Info] messages are only provided if the verbose level is greater 0 (command line option `-v` / `--verbose`).

This section lists common [Warning] and [Info] log messages and describes their meaning. Some of them are also related to each other.

Message	Description
[Info] E2E01004 - - Found maximum value for MaxDeltaCounterInit of 15 which should actually be 14 - value will be corrected in generated E2E-config file	This message occours if MaxDeltaCounterInit is set to the maximum value as described in SWS_E2Elibrary, which is apparently not a meaningful value and therefore rejected by the E2EPW Generator. Background: MaxDeltaCounterInit is used to initialize the status variable MaxDeltaCounter. The maximum possible value is 14 (Profile 1) respectively 15 (Profile 2). However, MaxDeltaCounter is always incremented by the E2Elib before it is used. Therefore, a MaxDeltaCounterInit value of 13 and 14 is semantically equivalent with Profile 1 (14 and 15 are equivalent with Profile 2). The E2EPW Generator, however, is pessimistic and assumes an erroneous configuration if a configuration value makes no sense. Therefore, the Preprocessor adapts the value and provides an [Info] message. Additionally, the Preprocessor places a comment after the corrected value in the E2EConfig file.
[Info] E2E01100 - Could not find adequate signals in signal group 'example_sgrp_with_offset' to be configured as protection signals - trying non-zero data offset of signal group to find matching signals.	Only appears if verbose level is greater `0`. It indicates that the protection signals could not be found at their expected position, but they simply might be offset from the start. The preprocessor then tries several offsets to find a matching value for all signals. If a matching offset is found, another `[Info] E2E01100` message is provided.
[Info] E2E01100 - Signal Group offset seems to fit with offset 16 (at begin of first signal)	This message only appears if the previous `[Info] E2E01100` message was provided. It states the result of the offset search. Please note that the E2EPW Generator has no direct support for non-zero offsets, so the Preprocessor will substract this offset value from the Bit_Position of each signal

Message	Description
	in the E2EConfig file. This is also stated as a comment before the signals block in the corresponding Protected_Area in the E2EConfig file.
[Warning] E2E00210 - - Could not determine position/name of (all) protection signals in signal group 'Eng_Rs_EngCtrl_Pr2_24huh6mozixac7vybqdhljanf' Resulting E2E Config file must be adapted manually regarding Bit_Position and Signal_Property	This [Warning] message indicates that there was no match for the determination of protection signals. This might be due to a mismatching `Bit_Position` of single signals, `Signal_Type` or `Bit_Length`, but most probably there is a misconfiguration in the system (missing mapping of signals to data elements, or signal layout configuration). The E2EPW Generator will not accept this E2EConfig file without modification.
[Info] E2E01101 - - There was an error/warning during processing an End-To-End-Protection ('End-To-End-Protection_SHORT-NAME'). No E2E config section will be generated for this End-To-End-Protection. However, continuing to process remaining End-To-End-Protections.	This info message refers to a previously stated warning for that End-To-End-Protection. The severity of that warning leads to this info message, which states that the preprocessor did not generate a Protected_Area for that End-To-End-Protection. However, it will go on with processing the next End-To-End-Protection.
[Warning] E2E01003 - - Profile 'PROFILE_04' is not supported/ignored for E2EPW configuration generation - Please use E2E Transformer (E2EXf) generator. Found End-To-End-Profile with Category 'PROFILE_04' Example.arxml.	If Profile 4, 5, or 6 are found, the Preprocessor provides a warning message, which states that these profiles are not handled by the Preprocessor. End-To-End-Protections are ignored, so that they can be processed using the E2E Transformer (E2EXf) tool chain.
[Warning] E2E00206 - - No supported End-to-End protection found.	A follow-up warning message to message `[Info] E2E01101` or `[Warning] E2E01003`. If all End-To-End-Protections are skipped due to severe warnings, no E2EConfig file will be generated. Also, no final "Configuration file <filename> successfully written." message will be provided.

5 E2E Protection Wrapper Generator

The generator is a Microsoft Windows console application. It uses an **E2EConfig file** to generate files that contain the **E2EPW** code.

The Windows version on which the generator was tested during development is noted in the corresponding Safety Manual [TT_E2EPW_SM] 68. An addendum also confirms compatibility with Microsoft Windows 7. However, as there are no special system requirements, the generator should run without problems on all Windows version since Microsoft Windows XP SP3.

5.1 Using the Generator

To use the generator, use the following command line:

```
pwg.exe <config-file> <output-path>
```

where <*config-file*> and <*output-path*> must exist.

The following table lists the command line **parameters**:

Name	Meaning
`<config-file>`	This parameter specifies the path to a valid E2EConfig file 15.
`<output-path>`	This parameter specifies the path to which the generator will write the created files.

The E2EPWG will output an error message and quit if something goes wrong. This will be the case if:

- command line parameters are omitted.
- the E2EConfig file could not be found.
- the output path does not exist.
- the `<config-file>` or the `<output-path>` parameter exceeds the maximum length of a valid file path (260 characters).
- the major version of the E2EConfig file and the E2EPWG do not match.
- the minor version of the E2EConfig file is larger than the minor version of the E2EPWG.
- the **E2EConfig file** has an **invalid syntax**.
- the **E2EConfig file** has an **invalid content**.
- a file the E2EPWG wants to create already exists (for example, from a previous run).
- a **file I/O error** occurs, such as insufficient disk space or missing write permissions.
- there is not enough memory left.

Note: It is also possible to call the E2EPWG directly from the preprocessor.

5.2 E2EConfig file

This Section gives a description of the syntax [15] of the **E2EConfig file** and its elements [18].

5.2.1 Syntax

The **E2EConfig file** version 2.0.0 syntax has a **human-readable** text format. The encoding is **ASCII**. Comments can be added using C++-style syntax:

```
/* comment until asterisk-slash */
// comment until end of line
```

All the **keywords** (field names and keyword values) are **case-sensitive**.

Note: The order of the field names is fixed.

The **syntax** is given in **EBNF**:

Symbol	Meaning
`::=`	A name on the left side of the `::=` is expressed with the syntax on its right side.
`<>`	Angle brackets are used to mark objects that are specified later.
`|`	The definition separator symbol indicates **choice**. Exactly one of the choices must appear.
`[ ]`	The text between the square brackets is **optional**.
<u>{}</u>	The text between the curly brackets may be omitted or may appear once or may be repeated arbitrarily. The **brackets** are **underlined** to distinguish them from the **literals** "{" and "}".
`<string>`	Indicates any character string enclosed in double quotes (i.e., "string").

The syntax is as follows:

```
<E2EPW_configuration_file> ::=
    E2EPW_configuration
    {
        Version_Major = <integer> ;
        Version_Minor = <integer> ;
        Version_Patch = <integer> ;
        Protected_Areas <protected_area> {,<protected_area>} ;
    } ;

<protected_area> ::=
    {
        PDE_Name = <string> ;
        PDE_Type = <string> ;
        Node_Name = <string> ;
        Direction = <direction> ;
        Byte_Order_CPU = <byte_order> ;
        Bit_Order = <bit_order> ;
        Bit_Counting = <bit_counting> ;
        Unused_Bit_Value = <integer> ;
        Check_DeSerial = <no_yes> ;
        [ Includes_H = <string_list> ; ]
        [ Includes_C = <string_list> ; ]
        Profile <profile_p01>
              | <profile_p02>
        Signals <signal> {,<signal>} ;
        Protection_Wrapper <autosar_rte>
                         | <autosar_no_rte>
                         | <j1939_can>
    }

<profile_p01> ::=
    {
        Category = P01 ;
        Data_Length = <integer> ;
        Data_ID = <hex> ;
        Data_ID_Mode = <data_id_mode> ;
        Max_Delta_Counter_Init = <integer> ;
        CRC_Offset = <hex> ;
        Counter_Offset = <hex> ;
        Data_ID_Nibble_Offset = <hex> ;
        Max_No_New_Or_Repeated_Data = <integer> ;
        Sync_Counter_Init = <integer> ;
    } ;

<profile_p02> ::=
    {
        Category = P02 ;
        Data_Length = <integer> ;
        Data_ID_List = <hex_list_16> ;
        Max_Delta_Counter_Init = <integer> ;
        Max_No_New_Or_Repeated_Data = <integer> ;
        Sync_Counter_Init = <integer> ;
        Offset = <integer> ;
    } ;
<signal> ::=
    {
        Signal_Name = <string> ;
        Signal_Type = <sig_type> ;
        Signal_ID = <integer> ;
        Signal_Property = <sig_prop> ;
        Byte_Order = <byte_order> ;
        Bit_Length = <integer> ;
        Bit_Position = <integer> ;
    }
```

```
<autosar_rte> ::=
    {
        E2EPW_Usecase = AUTOSAR_RTE ;
        Port_Name = <string> ;
        VDP_Name = <string> ;
        Is_Opaque = <no_yes> ;
        Use_Call_By_Ref = <no_yes> ;
        Use_RTE_Update = <no_yes> ;
        Use_RTE_Instance = <no_yes> ;
    } ;

<autosar_no_rte> ::=
    {
        E2EPW_Usecase = AUTOSAR_NO_RTE ;
        Signal_Grp_ID = <integer> ;
        Is_Opaque = <no_yes> ;
    } ;

<j1939_can> ::=
    {
        E2EPW_Usecase = J1939_CAN ;
        PGN_Number = <integer> ;
        Is_Opaque = <no_yes> ;
    } ;

<direction> ::=
    RX | TX

<byte_order> ::=
    LITTLE_ENDIAN | BIG_ENDIAN

<bit_order> ::=
    DECREASING | INCREASING

<bit_counting> ::=
    SAWTOOTH | MONOTONE

<no_yes> ::=
    NO | YES

<string_list> ::=
    <string> {,<string>}

<data_id_mode> ::=
    BOTH | ALT | LOW | NIBBLE

<hex_list_16> ::=
      <hex>, <hex>, <hex>, <hex>
    , <hex>, <hex>, <hex>, <hex>
    , <hex>, <hex>, <hex>, <hex>
    , <hex>, <hex>, <hex>, <hex> ;

<sig_type> ::=
    UINT8 | UINT16 | UINT32 | SINT8 | SINT16 | SINT32 | BOOLEAN | UINT8N

<sig_prop> ::=
    NORMAL | CHK | SEQCNT | NIBBLE

<integer> ::=
    <digit>{<digit>}

<digit> ::=
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

```
<hex> ::=
    <hex_prefix><hex_digit>{<hex_digit>}

<hex_prefix> ::=
    0x | 0X

<hex_digit> ::=
    <digit> | a | b | c | d | e | f | A | B | C | D | E | F
```

5.2.2 Description of Elements

Although the **syntax** of the **E2EConfig file** is **defined** for **3 use cases** and several profiles, **only** the use case **AUTOSAR RTE** is supported by the delivered package. Depending on the delivered E2EPWG variant, different profiles may be supported. The E2EPWG checks the semantics for plausibility, but further checks must be done manually. They are described in the *E2E Protection Wrapper Safety Manual* [TT_E2EPW_SM] 68 . Also, the cause of validation errors is explained here.

The description of the semantics is structured in the same way as the BNF description of the syntax.

A **C-identifier** is a string that has some **limitations** regarding the contained characters and the length: It starts with a **letter** or an **underscore** and contains only letters, numbers, and underscores. It must not be identical to one of the **ANSI C** keywords. The **maximum length** of a C-identifier allowed in the **E2EConfig file** is **128** characters.

A **name identifier** is a string that consists only of letters, numbers, and underscores (max. 128 characters).

<E2EPW_configuration_file>	
`Version_Major`	Must match the corresponding values of the E2EPWG version.
`Version_Minor`	Must be smaller than or equal to the corresponding values of the E2EPWG version.
`Version_Patch`	Can differ from the corresponding value of the E2EPWG version.

<protected_area>	
`PDE_Name`	This parameter defines a **name identifier** for the **PDE**. This string is used as part of the file name of some of the generated files. It must not be longer than 128 characters. For the use case AUTOSAR RTE, `PDE_Name` must be equal to `VDP_Name`.
`PDE_Type`	This parameter defines the **data type** of the **PDE** data structure. This string is a **C-identifier** or begins with `struct` followed by a white space and a C-identifier. The C-identifier part must not be longer than 128 characters in both cases. `PDE_Type` must match the data type defined for and used by the application.

`Node_Name`	This parameter defines the **name identifier** that is used as a part of the file name of some of the generated files. It must not be longer than 128 characters. For use case `AUTOSAR RTE`, the SW-C type must be used (name of the software component).
`Direction`	This parameter defines the **direction** of the message flow. ▪ `TX` means that the PDE is **sent** by the application, therefore the E2EPW must provide a **write** function. ▪ `RX` means that the PDE is **received**, so the E2EPW must provide a **read** function.
`Byte_Order_CPU`	This parameter defines the **byte order** of the host CPU. ▪ `BIG_ENDIAN` means that the most significant byte is stored at the lowest address. ▪ `LITTLE_ENDIAN` means that the least significant byte is stored at the lowest address.
`Bit_Order`	This parameter defines the **ordering** of the **bits** within a byte. In AUTOSAR, this is used to define the order of bits that are transmitted. ▪ `DECREASING` means that the logical numeration of bits in a byte is decreasing, e.g., 7, 6, 5, 4, 3, 2, 1, 0 for the first byte of a sequence. ▪ `INCREASING` means that the logical numeration of bits in a byte is increasing, e.g., 0,1,2,3,4,5,6,7 for the first byte of a sequence.
`Bit_Counting`	This parameter defines the **order of significance** of bits over a transmitted data unit. ▪ `MONOTONE` means that the significance is either increasing or decreasing constantly, while ▪ `SAWTOOTH` means that there is a discontinuity between consecutive bytes. Only the following combinations of `Bit_Order` and `Bit_Counting` are supported: ▪ `DECREASING` with `SAWTOOTH` and ▪ `INCREASING` with `MONOTONE`.
`Unused_Bit_Value`	This parameter defines the **value** to which **unused bits** in the I-PDU must be set. The only valid values are **0** and **1**.
`Check_DeSerial`	If `YES`, a **deserialization check** is done when a PDE is received. The value can only be `YES` if `Direction` is `RX` and `Is_Opaque` is `NO`. In [AS_E2E_SWS] 68, this value is defined to be `YES` if `Direction` is `RX` and `Is_Opaque` is `NO`.
`Includes_H`	This parameter defines the **external include files** that must be included in the generated **header** files. Each file name must be unique within this list, and the length of each file name must not be greater than 260 characters. For use case `AUTOSAR RTE`, the value will contain `Rte_Type.h` and the header file of the application.
`Includes_C`	This parameter defines the **external include files** that must be

	included in the generated **source** files. Each file name must be unique within this list, and the length of each file name must not be greater than 260 characters. For use case `AUTOSAR RTE`, all necessary files are included in `Includes_H`.

<profile_p01>	
`Category`	This parameter defines the **name** of the **profile**, in this case **P01**. Unsupported profiles will cause an error message that is reported by the E2EPWG.
`Data_Length`	This parameter defines the **length** of the **I-PDU** in bits. The minimum value is 16. The maximum value is 240. The value must be a **multiple of 8**.
`Data_ID`	The sender and the receiver share the same `Data_ID`, which is incorporated into the CRC calculation. In this way, it is possible to detect if the sender and/or receiver is the intended one or not. The `Data_ID` has **2 bytes**. For profile P01, the `Data_ID` is equivalent to the Application ID in [BMW_LAST_KOMM] 68.
`Data_ID_Mode`	This parameter defines how the `Data_ID` is incorporated into the CRC calculation. ▪ `BOTH` means that both bytes of the `Data_ID` are used for CRC calculation (lower byte first). ▪ `LOW` means that only the lower byte is used. ▪ `ALT` means that the lower byte is used when the counter is even, otherwise the higher byte is used. ▪ NIBBLE means that the lower 8 bits are used for implicit inclusion in CRC calculation, while additional 4 bits are explicitly placed in the transmitted I-PDU array as distinct signal.
`Max_Delta_Counter_Init`	This parameter defines the **initialization value** of the variable `Max_Delta_Counter` as specified in [TT_E2EPW_SM] 68. The value must be in the closed interval **`[0...13]`**.
`CRC_Offset`	This parameter determines, in bits, the offset of the CRC byte in the I-PDU array. It must be a **multiple of 8**. The `CRC_Offset` is the position of the least significant bit.
`Counter_Offset`	This parameter determines, in bits, the offset of the Counter in the I-PDU array. It must be a **multiple of 4**. The `Counter_Offset` is the position of the least significant bit.
`Data_ID_Nibble_Offset`	Determines the offset of the Data ID nibble in the PDU array in bits. The Data_ID_Nibble_Offset is the position of the least significant bit.
`Max_No_New_Or_Repeated_Data`	The threshold value of maximum consecutive data losses or repetitions before re-synchronization according to AUTOSAR SWS E2Elib is performed. The value must be in the closed interval

	`[0...15]`. Backward-compatibility to previous E2Elib profile versions without this feature is given by the default value **15**.
`Sync_Counter_Init`	After re-synchronization according to AUTOSAR SWS E2Elib is performed, Sync_Counter_Init determines how many SYNC states are returned on valid receptions before normal operation continues. The value must be in the closed interval `[0...255]`. Backward-compatibility to previous E2Elib profile versions without this features is given by the default value **0**.

<profile_p02>	
`Category`	This parameter defines the **name** of the **profile**, in this case **P02**. Unsupported profiles will cause an error message that is reported by the E2EPWG.
`Data_Length`	This parameter defines the **length** of the **I-PDU** in bits. The minimum value is 16. The maximum value is 2048. The value must be a **multiple of 8**.
`Data_ID_List`	This parameter defines a **list** of **16 values** in hexadecimal format. Each value must be in the range `0x00...0xff`. This is the list of **data IDs** as specified in [TT_E2EPW_SM] 68.
`Max_Delta_Counter_Init`	This parameter defines the **initialization value** of the variable `Max_Delta_Counter` as specified in [TT_E2EPW_SM] 68. The value must be in the closed interval `[0...14]`.
`Max_No_New_Or_Repeated_Data`	The threshold value of maximum consecutive data losses or repetitions before re-synchronization according to AUTOSAR SWS E2Elib is performed. The value must be in the closed interval `[0...16]`. Backward-compatibility to previous E2Elib profile versions without this feature is given by the default value **16**.
`Sync_Counter_Init`	After performing are-synchronzation according to AUTOSAR SWS E2Elisb, Sync_Counter_Init determines how many SYNC states are returned on valid receptions before normal operation continues. The value must be in the closed interval `[0...255]`. Backward-compatibility to previous E2Elib profile versions without this features is given by the default value **0**.
`Offset`	Additional offset of the CRC and sequence counter signals in the PDU array in bits. The value must be a **multiple of 8**. Backward-compatibility to previous E2Elib profile versions without this feature is given by the default value **0**.

<signal>	
`Signal_Name`	This parameter defines the **name** of the **signal**. The signal name is **unique** within the protected area. The signal name must be the

	same as the one that is defined in the data type definition of the data element used by the application. The value must be a valid C-identifier.
`Signal_Type`	This parameter defines the **data type** of the **signal**. It must be the same as in the data type definition of the data element used by the application. The value must be a valid C-identifier.
`Signal_ID`	This parameter defines the **unique numeric identifier** of a **signal**. For use case `AUTOSAR RTE`, the only requirement to the `Signal_ID` is that it must be in the range **`[0...65535]`** and be unique within the protected area.
`Signal_Property`	This parameter defines if the signal is a **normal** user **signal** (`Signal_Property` is `NORMAL`) or a **special signal** used for the protection of the protected area. There must be exactly **one signal** with `Signal_Property CHK` for the profiles **P01**, **P02** and exactly one with the `Signal_Property SEQCNT` for all profiles. Also, `Bit_Length`, `Bit_Position` and `Signal_Type` must meet the requirements and settings for the used profile. The allowed value for `Bit_Position` also depends on the `Byte_Order` of the signal.
`Byte_Order`	This parameter defines the **byte order** of the signal. Depending on this value and the values of `Bit_Length` and `Bit_Position`, the signal is **mapped** to the corresponding **I-PDU**.
`Bit_Length`	This parameter defines the **length** of the **signal** in **bits**. ▪ If the `Signal_Type` is `BOOLEAN`, `Bit_Length` must be **1**. ▪ If the `Signal_Type` is `UINT8N`, the signal length must be a **multiple of 8**.
`Bit_Position`	This parameter defines the **position** of the **signal** within the I-PDU. ▪ If `Byte_Order` is `LITTLE_ENDIAN`, `Bit_Position` defines the position of the least significant bit. ▪ If `Byte_Order` is `BIG_ENDIAN`, `Bit_Position` defines the position of the most significant bit. ▪ If the `Signal_Type` is `UINT8N`, this parameter specifies the least significant bit of the first byte.

There are further restrictions that must be met, some of them are provided in this section. However, all restrictions, if not explicitly stated in the safety manual, are also automatically checked by the E2EPWG. Therefore, this section is for information purposes only.

- Signals must not overlap each other or exceed the I-PDU length.
- The `Signal_Type` of both, the checksum and the sequence counter signal must be `UINT8`.

Profile 1 [20]

- The checksum signal must be byte-aligned (i.e., it can occupy any byte of the protected area). The `Bit_Length` must be 8.

- The sequence counter signal must be positioned in the 4 least or most significant bits of any byte of the protected area. The `Bit_Length` must be 4.
- The data ID nibble signal must be present if `Data_ID_Mode` is `NIBBLE` and be positioned in the 4 least or most significant bits of any byte of the protected area. The `Bit_Length` must be 4.

Profile 2[21]

- The checksum signal must be positioned in the 8 bits of the first byte of the protected area.
  - For `Byte_Order = BIG_ENDIAN`, the `Bit_Position` must be 7.
  - For `Byte_Order = LITTLE_ENDIAN`, the `Bit_Position` must be 0.
  - The `Bit_Length` must be 8.
- The sequence counter signal must be positioned in the 4 least significant bits of the second byte of the protected area.
  - For `Byte_Order = BIG_ENDIAN`, the `Bit_Position` must be 11.
  - For `Byte_Order = LITTLE_ENDIAN`, the `Bit_Position` must be 8.
  - The `Bit_Length` must be 4.

<autosar_rte>	
`E2EPW_Usecase`	Value must be `AUTOSAR_RTE`.
`Port_Name`	This parameter defines the **name identifier** of the **communication port**. This parameter is used as a part of the names of some generated functions.
`VDP_Name`	This parameter defines the **name identifier** of the **VDP**. This parameter is used as a part of the names of some generated functions. The parameter `VDP_Name` must be the `DataElement` name used by the RTE.
`Is_Opaque`	This parameter defines if the PDE is provided as I-PDU at application level. If `YES`, the application is responsible for marshaling or unmarshaling the DE. The DE is provided as byte array to the protection wrapper. Also, the lower layers must be configured to handle the DE as I-PDU.
`Use_Call_By_Ref`	For complex data types, the RTE expects that DEs are provided by reference. If `Direction` is `RX`, the RTE always expects the DE to be provided by reference.
`USE_RTE_Update`	If `YES`, the E2EPW calls the RTE function `Rte_IsUpdated_<p>_<o> ()`, where `<p>` is the port and `<o>` the `VariableDataElement`. The value must be `NO`, if the `AUTOSAR RTE` does not support this function.
`Use_RTE_Instance`	Must be `YES` if the SW-C supports multiple RTE instances (multiple instantiation). Otherwise, the value must be `NO`.

5.2.3 File Content Checks

This Section only lists a selection of important checks revealing missing files or files that cannot be compiled. For detailed instructions on how to check a **E2EConfig file**, refer to the *E2E Protection Wrapper Safety Manual* [TT_E2EPW_SM] 68.

- The parameter combination `PDE_Name`, `Node_Name`, `Port_Name` and `Direction` must be unique within the **E2EConfig file**. Also, the parameter combination `VDP_Name`, `Node_Name`, `Port_Name`, and `Direction` must be unique within the **E2EConfig file**.
- The `PDE_Name` needs not be unique, but make sure by review that PAs with the same `PDE_Name` have the same signal definition if marshaling is required.
- Make sure that file names generated by the E2EPWG do not exceed the limitations of the file system or the operating system.
- Most Windows applications are bound to a restrictive limitation: Using the standard file access functions restricts the total path length to **260 characters**, including the drive letter prefix and the null termination character. This also refers to relative paths, and can only be mitigated by path substitution using the DOS command `subst`.
- The **files** generated by the E2EPWG are named according to the following patterns.

  These files contain the protection wrapper interface functions:
  - `E2EPW_`*`<Node_Name>`*`_`*`<Port_Name>`*`_`*`<PDE_Name>`*`_`*`<Direction>`*`.h`
  - `E2EPW_`*`<Node_Name>`*`_`*`<Port_Name>`*`_`*`<PDE_Name>`*`_`*`<Direction>`*`.c`

  These files contain the marshaling functions if required:
  - `E2EPW_Marshal_<PDE_Name>.h`
  - `E2EPW_Marshal_<PDE_Name>.c`

  These files contain the deserialization check function if required:
  - `E2EPW_CheckDeserial_`*`<PDE_Name>`*`.h`
  - `E2EPW_CheckDeserial_`*`<PDE_Name>`*`.c`
- It must be ensured that the **overall file name length** does **not exceed** the **260 character** limit for the **absolute path** length. Otherwise it will not be possible to create some files, and the E2EPWG will exit with an error message.
- The generated code contains **memory mapping defines**, which require a corresponding counterpart in the *`MemMap.h`* include file. Otherwise, the *`MemMap.h`* include file will prompt the compiler to display a warning or error message. There are **three types** of **defines** used in the generated code:
  1. Defines with the **prefix *`E2EPW_`***, which are provided in the *`E2EPW_MemMap.inc`* file and should be incorporated into the *`MemMap.h`* include file of the system, and
  2. **SW-C-specific** defines.

     The SW-C specific defines are:
     - *`<swc>`*`_START_SEC_CODE`
     - *`<swc>`*`_STOP_SEC_CODE`
     - *`<swc>`*`_START_SEC_CONST_UNSPECIFIED`
     - *`<swc>`*`_STOP_SEC_CONST_UNSPECIFIED`

- `<swc>_START_SEC_VAR_INIT_UNSPECIFIED`
- `<swc>_STOP_SEC_VAR_INIT_UNSPECIFIED`
- `<swc>_START_SEC_VAR_NOINIT_UNSPECIFIED`
- `<swc>_STOP_SEC_VAR_NOINIT_UNSPECIFIED`
- `<swc>_START_SEC_VAR_ZERO_INIT_UNSPECIFIED`
- `<swc>_STOP_SEC_VAR_ZERO_INIT_UNSPECIFIED`

where *`<swc>`* is the value of the configuration field `Node_Name`.

These are assumed to be provided in a file `<swc>_MemMap.h`, which is generated by the RTE generator or must be created manually.

3. The defines for software component independent code (marshaling and check-deserialization check) are:
   - `E2EPW_START_SEC_CODE_LIB`
   - `E2EPW_STOP_SEC_CODE_LIB`.

6 Generated Code

6.1 API

This Section describes the API of the generated functions.

Note: When referring to a **byte number** within a multi-byte value, the number describes **bytes of increasing order**. **Byte 0** is therefore of least significance.

The **function names** use the following **placeholders**:

Placeholder	Config field	Description
`<node>`	`Node_Name`	The node name. Also referred to as SW-C name.
`<o>`	`VDP_Name`	The `VariableDataPrototype` (`VariableDataElement`) of the messages through a port.
`<p>, <port>`	`Port_Name`	The port name.
`<pde>`	`PDE_Name`	The PDE name.
`<profile>`	`Category`	`P01` or `P02`, depending on the E2Elib profile used.
`<pde-type>`	`PDE_Type`	The data type of the PDE.
`<rxtx>`	`Direction`	The direction (`RX` or `TX`) in lowercase.

6.1.1 Initialization

Syntax	`void E2EPW_WriteInit_<p>_<o>x ()`
Reentrancy	reentrant
Parameters	none
Return value	none
Description	This function initializes the E2Elib communication state for a DE defined by `<p>` and `<o>` on the sender side.

Syntax	`void E2EPW_ReadInit_<p>_<o> ()`
Reentrancy	reentrant
Parameters	none
Return value	none
Description	This function initializes the E2Elib communication state for a DE defined by `<p>` and `<o>` on the receiver side.

6.1.2 Status

Syntax	`E2E_<profile>ProtectStateType * E2EPW_Get_ProtectState_<p>_<o> ()`	
Reentrancy	reentrant	
Parameters	none	
Return value	`E2E_<profile>ProtectStateType *`	pointer to `E2E_<profile>ProtectStateType`
Description	Returns a pointer to the current E2Elib communication state on the sender side.	

Syntax	`E2E_<profile>CheckStateType * E2EPW_Get_CheckState_<p>_<o> ()`	
Reentrancy	reentrant	
Parameters	none	
Return value	`E2E_<profile>CheckReceiverStateType*`	pointer to `E2E_<profile>CheckStateType`
Description	Returns a pointer to the current E2Elib communication state on the receiver side.	

6.1.3 Transmission and Reception

The following function is provided for protected transmission with E2EPW:

Syntax	`uint32 E2EPW_Write_<p>_<o>`

	([Rte_Instance instance,] <type> [*] AppData)	
Reentrancy	not reentrant	
Parameters (in)	`instance`	The `instance` value is passed to `Rte_Write_<p>_<o> ()`. This is an optional parameter, which depends on the configuration option `Use_RTE_Instance`.
	`AppData`	This parameter defines the DE to be protected and transmitted. Its type, `<type>`, depends on the configuration of the DE. This parameter can be called: ▪ **by value** (`AppData` is passed by value) or ▪ **by reference** (a pointer to `AppData` is passed).
Return value	`uint32`	**`Byte 0`**, the return-code of `E2E_Protect ()`: ▪ `E2E_E_INPUTERR_NULL` ▪ `E2E_E_INPUTERR_WRONG` ▪ `E2E_E_OK` (default) **`Byte 1`**, the return-code of `Rte_Write_<p>_<o> ()`: ▪ `RTE_E_COM_STOPPED` ▪ `RTE_E_SEG_FAULT` ▪ `RTE_E_OK` (default) **`Byte 2`**, the result of the `AppData=NULL` check: ▪ `E2E_E_INPUTERR_NULL` ▪ `E2E_E_OK` (default) **`Byte 3`**, always 0

Note: The default value is the value that is used when the corresponding function/check is not called (e.g., in case of a previous error).

The following function is provided for protected reception with E2EPW:

Syntax	`uint32 E2EPW_Read_<p>_<o>`
	`( [Rte_Instance instance, ] <type> [*] AppData)`	

Reentrancy	not reentrant	
Parameters (in)	`instance`	The `instance` value is passed to `Rte_Read_<p>_<o> ()`. This is an optional parameter, which depends on the configuration option `Use_RTE_Instance`.
	`AppData`	This parameter defines the DE to be protected and transmitted. Its type, `<type>`, depends on the configuration of the DE.

Return value	`uint32`	**`Byte 0`**, the return-code of `E2E_Check ()`:
		▪ `E2E_E_INPUTERR_NULL`
		▪ `E2E_E_INPUTERR_WRONG`
		▪ `E2E_E_OK (default)`
		
		`Byte 1`, the return-code of `Rte_Read_<p>_<o> ()`:
		▪ `RTE_E_INVALID`
		`RTE_E_MAX_AGE_EXCEEDED`
		`RTE_E_NEVER_RECEIVED`
		`RTE_E_UNCONNECTED`
		`RTE_E_OK` (default)
		
		`Byte 2`, the result of the `AppData=NULL` check:
		`E2E_E_INPUTERR_NULL`
		`E2E_E_OK`
		
		`Byte 3`/`Bit 7`, deserial check:
		▪ `1 (E2EPW_DESERIAL_ERR)`
		▪ `0` (default)
		
		`Byte 3`/`Bit 0...6`, `State->Status`:
		▪ `E2E_<profile>STATUS_OK: 0x00`
		▪ `E2E_<profile>STATUS_NONEWDATA: 0x01`
		▪ `E2E_<profile>STATUS_WRONGCRC: 0x02`
		▪ `E2E_<profile>STATUS_SYNC: 0x03`
		▪ `E2E_<profile>STATUS_INITIAL: 0x04`
		▪ `E2E_<profile>STATUS_REPEATED: 0x08`
		▪ `E2E_<profile>STATUS_OKSOMELOST: 0x20`
		▪ `E2E_<profile>STATUS_WRONGSEQUENCE 0x40`

Note: The default value is the value that is used when the corresponding function/check is not called (e.g., in case of a previous error).

6.1.4 Usage Example Code

6.1.4.1 *Application Sample Code*

This is an application sample code to be used with use case AUTOSAR RTE[23] and profile 2[21].

The following code example is a guideline only and must be adapted to the respective application design. In particular, the error handling depends on the application and on how and when to use the received data in case of errors.

The placeholders defined in the previous sections are used here again with the same meaning.

Necessary include files:

```
#include "E2EPW_<node>_<port>_<pde>.h"
..
```

Convenience Code

```
#define RETURNCODE_E2E        (x)  ((uint32)(x) & 0xFF)
#define RETURNCODE_RTE        (x) (((uint32)(x) & 0xFF00) >> 8)
#define RETURNCODE_APPDATA    (x) (((uint32)(x) & 0xFF0000) >> 16)
#define RETURNCODE_E2ESTATUS  (x) (((uint32)(x) & 0x7F000000) >> 24)
#define RETURNCODE_DESER      (x) (((uint32)(x) & 0x80000000) >> 31)


const uint32 e2epw_read_status_ok_u32 =
       (0<<31)                      /* Deserial-Check */
    | ((E2E_P02STATUS_OK & 0x7F)<<24)  /* ReceiverStatus */
    |  (E2E_E_OK<<16)               /* Protection Wrapper */
    |  (RTE_E_OK<<8)                /* Rte_Read/Rte_Write */
    |  (E2E_E_OK);                  /* E2E_P02Check/E2E_P02Protect */
const uint32 e2epw_write_status_ok_u32 =
       (0<<31)                      /* Deserial-Check */
    |  (E2E_E_OK<<16)               /* Protection Wrapper */
    |  (RTE_E_OK<<8)                /* Rte_Read/Rte_Write */
    |  (E2E_E_OK);                  /* E2E_P02Check/E2E_P02Protect */
```

Code for Initialization

```
/* initialize all tx */
E2EPW_WriteInit_<p>_<o> ();
…
/* initialize all rx */
E2EPW_ReadInit_<p>_<o> ();
…
```

Code for Transmission and Check of Return Values

```
uint32       r_tx_u32;
Rte_Instance inst;
<type>       AppData;
<type> *     AppDataPtr = &AppData;


/* set AppData */
...


/* send AppData */
r_tx_u32 = E2EPW_Write_<p>_<o> ( inst, AppDataPtr);


if (r_tx_u32 != e2epw_write_status_ok_u32)
{
    /* something went wrong - find out what */
    if (RETURNCODE_APPDATA (r_tx_u32) != E2E_E_OK)
    {
        /* AppDataPtr was NULL */
    }
    if (RETURNCODE_E2E (r_tx_u32) != E2E_E_OK)
    {
        /* E2Elib reported some error */
    }
    if (RETURNCODE_RTE (r_tx_u32) != RTE_E_OK)
    {
        /* RTE reported some error */
    }
    ..
}
```

Code for Reception and Check of Return Values

```
uint32       r_rx_u32;
Rte_Instance inst;
<type>       AppData;
<type> *     AppDataPtr = &AppData;
...

/* receive AppData */
r_rx_u32 = E2EPW_Read_<p>_<o> (inst, AppDataPtr);


if (r_rx_u32 != e2epw_read_status_ok_u32)
{
    /* something went wrong - find out what */
    if (RETURNCODE_APPDATA (r_rx_u32) != E2E_E_OK)
    {
        /* AppDataPtr was NULL */
    }
    if (RETURNCODE_E2E (r_rx_u32) != E2E_E_OK)
    {
        /* E2Elib reported some error */
    }
    if (RETURNCODE_RTE (r_rx_u32) != RTE_E_OK)
    {
        /* RTE reported some error */
    }
    if (RETURNCODE_DESER (r_rx_u32) != 0)
    {
        /* E2EPW detected some deserialization error caused by the COM */
    }
    if (RETURNCODE_E2ESTATUS (r_rx_u32) != E2E_P02STATUS_OK)
    {
        /* E2Elib detected some status other than ok */
        switch (RETURNCODE_E2ESTATUS (r_rx_u32))
        {
            case E2E_P02STATUS_NONEWDATA:       /* no new message was available */
                                 break;
            case E2E_P02STATUS_WRONGCRC:        /* CRC was wrong */
                                 break;
            case E2E_P02STATUS_SYNC:            /* sync in progress */
                                 break;
            case E2E_P02STATUS_INITIAL:         /* first data received  */
                                 break;
            case E2E_P02STATUS_REPEATED:        /* message was repeated */
                                 break;
             case E2E_P02STATUS_OKSOMELOST:       /* some messages were lost, but it's
ok */
                                 break;
            case E2E_P02STATUS_WRONGSEQUENCE:    /* message had invalid sequence count
*/
                                 break;
        }
        }
}
```

Note: For debugging, all values of `E2E_P02CheckStatesType ()` may be relevant. Then the function `E2EPW_Get_CheckState_<p>_<o> ()` can be used to retrieve state information.

6.1.5 Differences to SW-C End-to-End Communication Protection Library

The implementation of the E2EPW is based on the guideline in [AS_E2E_SWS] (68), Annex B.

However, there are some deviations from the guideline.

Deviations are required/suggested due to:

- Incompleteness or errors in the guideline of the initial AUTOSAR Release 4.0.1/3.2.1, or
- Extensions for easier integration and application of the E2EPW API, or
- Backward-compatibility with previous versions of the E2EPW by TTTech.

The deviations are listed here:

Title	**API Extension**
AUTOSAR 3.2.1/4.0.1	AUTOSAR does not provide API functions to initialize the E2Elib communication state.
TTTech E2EPW 1.3	Additional functions: `E2EPW_Init_<p>_<o>_rx ()` and `E2EPW_Init_<p>_<o>_tx ()`
AUTOSAR 4.2.1	Init Functions defined as: `Std_ReturnType E2EPW_WriteInit_<p>_<o> ( Rte_Instance <instance> )`
TTTech E2EPW 2.0	Renamed functions to conform to AUTOSAR Release: 4.2.1: `void E2EPW_ReadInit_<p>_<o> (void)` and `void E2EPW_WriteInit_<p>_<o>_tx (void)`
Reason	**API extension:** Initialization shall be possible each time the application is reset. Renaming of initialization functions due to increase compatibility to AUTOSAR specification document. However, return value is not used, the parameter must not be present according to AUTOSAR specification.

Title	API Extension
AUTOSAR 3.2.1/4.0.1	AUTOSAR does not provide API functions for retrieving the current E2Elib communication state.

TTTech E2EPW 1.3	Additional functions:
	`E2E_<profile>SenderStateType *`
	`    E2EPW_Get_SenderState_<p>_<o> (void)`
	and
	`E2E_<profile>ReceiverStateType *`
	`    E2EPW_Get_ReceiverState_<p>_<o> (void)`
	Where `<profile>` is the short name of the profile.

AUTOSAR 4.2.1	-

TTTech E2EPW 2.0	Renamed additional functions to reflect renamed state type of E2Elib:
	`E2E_<profile>ProtectStateType *`
	`    E2EPW_Get_ProtectState_<p>_<o> (void)`
	and
	`E2E_<profile>CheckStateType *`
	`    E2EPW_Get_CheckState_<p>_<o> (void)`
	Where `<profile>` is the short name of the profile.

Reason	**API extension**: The AUTOSAR guideline does not include an API to retrieve the current state. **Note**: The internal State variable has been moved from function level to module level. The internal State variable cannot be read directly, but only by these functions.

Title	Communication State `State`.
AUTOSAR 3.2.1/4.0.1	The communication state is function-local (in `E2EPW_Read/Write_<p>_<o> ()`).
TTTech E2EPW 1.3	The communication state is defined at module level.
AUTOSAR 4.2.1	The communication state is defined at module level.
TTTech E2EPW 2.0	The communication state is defined at module level.

Title	Communication State `State`.
Reason	The Communication State must be accessible among several functions in the module, variables must be defined at module level to be memory-mapped using the MemMap approach of AUTOSAR.

Title	Deserialization return value of `E2EPW_Read_<p>_<o> ()` (see `E2E0265`)
AUTOSAR 3.2.1/4.0.1	AUTOSAR specifies **byte 3** of the return value to mark a deserialization error, which is **0** (OK) or **1** (failed).
TTTech E2EPW 1.3	The value of the deserialization error has been moved to **bit 7** of **byte 3**. Use **bits 0-6** for the value of `State->Status`.
AUTOSAR 4.2.1	The layout of the return value changed various times in previous AUTOSAR releases. No backward-compatibility between releases. Deserialization error is currently reported in **byte 1**.
TTTech E2EPW 2.0	The layout is backward-compatible to TTTech E2EPW 1.3.
Reason	**Usage**: With the `State->Status` value in the return value of function `E2EPW_Read_<p>_<o> ()`, the reception result can be easily analyzed. A call of function `E2EPW_Get_ReceiverState_<p>_<o> ()` is only required for special cases, like retrieving `State->LostData` if `State->Status` is `E2E_<profile>STATUS_OKSOMELOST`, where `<profile>` is the short name of the profile.

Title	Return value layout of `E2EPW_Write_<p>_<o> ()` and `E2EPW_Read_<p>_<o> ()`
AUTOSAR 3.2.1/4.0.1	AUTOSAR specifies the following for `E2EPW_Write_<p>_<o> ()`: **byte 0** is the return value of `E2E_<profile>Protect ()`. **byte 1** is the status of `Rte_Write_<p>_<o> ()`. **byte 2** is the status of `E2EPW_Write_<p>_<o> ()` internal checks. **byte 3** is `E2E_E_OK` (not used). and for `E2EPW_Read_<p>_<o> ()`: **byte 0** is the return value of `E2E_<profile>Check ()`.

Title	**Return value layout of `E2EPW_Write_<p>_<o> ()` and `E2EPW_Read_<p>_<o> ()`**
	byte 1 is the status of `Rte_Read_<p>_<o> ()`. **byte 2** is the status of `E2EPW_Read_<p>_<o> ()` internal checks. **byte 3** is the status of the deserialization check (bit extension) .
TTTech E2EPW 1.3	The value of the deserialization error has been moved to `bit 7` of `byte 3`. Use bits 0-6 for the value of `State->Status`.
AUTOSAR 4.2.1	The layout of the return value changed various times in previous AUTOSAR releases. No backward-compatibility between releases. The current layout for `E2EPW_Write_<p>_<o> ()`: **byte 0** is the status of `Rte_Write_<p>_<o> ()`. **byte 1** is the status of `E2EPW_Write_<p>_<o> ()` internal checks. **byte 2** is the return value of `E2E_<profile>Protect ()`. **byte 3** is `E2E_E_OK` (not used). and for `E2EPW_Read_<p>_<o> ()`: **byte 0** is the status of `Rte_Read_<p>_<o> ()`. **byte 1** is the status of `E2EPW_Read_<p>_<o> ()` internal checks, including deserialization (bit extension) check. **byte 2** is the return value of `E2E_<profile>Check ()`. **byte 3** is the value of `State->Status` (`E2E_<profile>CheckStatusType`).
TTTech E2EPW 2.0	The layout is backward-compatible to TTTech E2EPW 1.3.
Reason	Initially, the deviation was to fix severe flaws in the AUTOSAR specification. Now, backward-compatibility to previous versions of TTTech E2EPW is to be maintained.

Title	**Abortion in case of errors**
AUTOSAR 3.2.1/4.0.1	AUTOSAR defines that all steps in `E2EPW_Read/Write_<p>_<o> ()` are evaluated even in case of errors.
TTTech E2EPW 1.3	In case of an error, the generated function aborts and returns a proper error code. This includes the following kinds of errors:

Title	Abortion in case of errors
	• `NULL` check of parameter `AppData`, • `E2E_<profile>Protect/Check ()` returns an error, • `RTE_Write_<p>_<o> ()` returns an error or • deserialization check returns an error. Where `<profile>` is the short name of the profile. Note that the return-values of `RTE_Read_<p>_<o> ()` is not considered to be an error (in the manner that `E2EPW_Read_<p>_<o> ()` needs to abort).
AUTOSAR 4.2.1	Changed specification to abort in case of error, as in TTTech E2EPW 1.3
TTTech E2EPW 2.0	Behavior as in TTTech E2EPW 1.3 and therefore conform to AUTOSAR 4.2.1.
Reason	**Safer code:** Continuation does not make sense after an error was detected. Continuation in case of an error can cause other errors.

Title	Return Code Interpretation for `E2EPW_Read/Write_<p>_<o>`
AUTOSAR 3.2.1/4.0.1	All steps in `E2EPW_Read/Write_<p>_<o> ()` are done even if errors occur. Thus, each **byte 0...3** of the return value gets a corresponding value.
TTTech E2EPW 1.3	In case of error, `E2EPW_Read/Write_<p>_<o> ()` aborts and the result values of further steps still have default values. The return codes are interpreted as follows: **`E2EPW_Read_<p>_<o> ()`**: 1. If **byte 0** is unequal to `E2E_E_OK (0)`, then a severe error occurred in `E2E_<profile>Check ()`. All other bytes are irrelevant. 2. If **byte 2** is unequal to `E2E_E_OK (0)`, then a severe error occurred in the Protection Wrapper code. All other bytes are irrelevant. 3. If **bit 7** in **byte 3** is **1**, then the deserialization check failed. All other bytes are irrelevant. 4. If **byte 0** and **byte 2** are `E2E_E_OK` and **bit 7** of **byte 3** is **0**, then a. **byte 1** holds the return value of `Rte_Read_<p>_<o> ()` and b. **bits 0...6** in **byte 3** hold the communication status from

Title	Return Code Interpretation for `E2EPW_Read/Write_<p>_<o>`
	`E2E_<profile>Check ().` Both are to be interpreted by the application. **`E2EPW_Write_<p>_<o> ()`**: 1. If **byte 0** is unequal to `E2E_E_OK (0)`, then a severe error occurred in `E2E_<profile>Protect ()`. All other bytes are irrelevant. 2. If **byte 2** is unequal to `E2E_E_OK (0)`, then a severe error occurred in the Protection Wrapper code. All other bytes are irrelevant. 3. If **byte 0** and **byte 2** are `E2E_E_OK`, then **byte 1** holds the return value of `RTE_Write_<p>_<o> ()`, which is to be interpreted by the application. 4. **Byte 3** is always **0**.
AUTOSAR 4.2.1	Changed specification to abort in case of error, as in TTTech E2EPW 1.3.
TTTech E2EPW 2.0	Behavior as in TTTech E2EPW 1.3 and therefore conform to AUTOSAR 4.2.1.
Reason	Interpretation is consequence of abortion in case of severe errors.

Title	Variables ret0...ret3 in `E2EPW_Read/Write_<p>_<o> ()`
AUTOSAR 3.2.1/4.0.1	AUTOSAR defines a separate variable for each check within the read/write function.
TTTech E2EPW 1.3	The generated code holds an internal `uint32` variable, which represents the return value and is filled in case of an error. It is a merge of the variables `ret0..ret3`.
AUTOSAR 4.2.1	AUTOSAR specification more relaxed on the internal variables, semantically backward-compatible. Code Example and activity diagrams use more verbose names.
TTTech E2EPW 2.0	As in TTTech E2EPW 1.3.
Reason	**Simpler code**: Before each step, it must be evaluated whether an error has raised or not. A single `uint32` variable makes these evaluations simpler.

Title	`NULL_PTR` **check at begin of** `E2EPW_Read/Write_<p>_<o> ()`
AUTOSAR 3.2.1/4.0.1	AUTOSAR proposes a `NULL_PTR` check for `AppData` at the end of the function.
TTTech E2EPW 1.3	The `NULL` check is the first step to do in the functions.
AUTOSAR 4.2.1	Changed specification to check `AppData` at beginning and abort in case of error, as in TTTech E2EPW 1.3.
TTTech E2EPW 2.0	Behavior as in TTTech E2EPW 1.3 and therefore conform to AUTOSAR 4.2.1.
Reason	**Safer code**: Applying `AppData` before `NULL_PTR` check may cause other errors.

Title	**`E2E_USING_RTE_ISUPDATED`**
AUTOSAR 3.2.1/4.0.1	AUTOSAR uses a `#define` to control the setting of `State->NewDataAvailable`.

TTTech E2EPW 1.3	The configuration of `State->NewDataAvailable` is moved to the E2EConfig file.
	As a result, the generated code contains either
	`State->NewDataAvailable = Rte_IsUpdated_<p>_<o> ()`
	or
	`State->NewDataAvailable = TRUE`
	
	There is no #define anymore.

AUTOSAR 4.2.1	`Rte_IsUpdated_<p>_<o> ()` is no longer used (since 4.1.1), return value of `Rte_Read_<p>_<o> ()` is used instead now (non-configurable). If return is not `RTE_E_OK`, then `State->NewDataAvailable` is set to `FALSE`.
TTTech E2EPW 2.0	Backward-compatible to TTTech E2EPW 1.3. Return value of `Rte_Read_<p>_<o> ()` does not affect further evaluation, but is returned in the E2EPW return code. Therefore, while in AUTOSAR 4.2.1 the unavailability of new data by the RTE would be treated as "no new data", it will be treated as "repeated" here, if Rte_IsUpdated_`Rte_IsUpdated_<p>_<o> ()` is not used.
Reason	All configuration for generated code is gathered in the E2EConfig file. Simpler code.

Title	**Encapsulation of marshaling in separate function.**
AUTOSAR 3.2.1/4.0.1	AUTOSAR defines the marshaling code directly in the wrapper code.
TTTech E2EPW 1.3	`E2EPW_Read/Write_<p>_<o> ()` calls a separate function `E2EPW_Marshal_<pde> ()`.
AUTOSAR 4.2.1	No change to AUTOSAR 3.2.1/4.0.1.
TTTech E2EPW 2.0	Backward-compatible to TTTech E2EPW 1.3.
Reason	Simpler code. Improved structure. Code-sharing for same Signal Group (e.g. multiple Receivers) possible.

Title	**Encapsulation of deserialization check in separate function.**
AUTOSAR 3.2.1/4.0.1	AUTOSAR defines the code for the check directly in the wrapper code.
TTTech E2EPW 1.3	`E2EPW_Read/Write_<p>_<o> ()` calls a separate function `E2EPW_CheckDeserial_<pde> ()`.
AUTOSAR 4.2.1	No change to AUTOSAR 3.2.1/4.0.1.
TTTech E2EPW 2.0	Backward-compatible to TTTech E2EPW 1.3.
Reason	Simpler code. Improved structure. Code-sharing for same Signal Group (e.g. multiple Receivers) possible.

Title	**Variants for parameters of `E2EPW_Read/Write_<p>_<o> ()`**
AUTOSAR 3.2.1/4.0.1	AUTOSAR defines the parameters `Instance` and `AppData` (call by reference).
TTTech E2EPW 1.3	The following variants are configurable: `Instance` may be given as a parameter or not. However, it has no meaning to the functionality of the E2EPW code and is merely used as call parameter for `Rte_Read/Write_<p>_<o> ()`. `AppData` can be passed **call by value** or **call by reference**. For `E2EPW_Read_<p>_<o>` (), it is always **call by reference**.
AUTOSAR 4.2.1	Parameter `Instance` is specified in all API functions as optional,

Title	Variants for parameters of `E2EPW_Read/Write_<p>_<o> ()`
	additional specifications state that the Instance feature is explicitly not supported by E2E.
TTTech E2EPW 2.0	Backward-compatible to TTTech E2EPW 1.3.
Reason	Call signature of `E2EPW_Read/Write_<p>_<o> ()` is aligned to the specification of `Rte_Read/Write_<p>_<o> ()` in the AUTOSAR RTE document.

Title	Variables Config and ConfigVal in `E2EPW_Read_<p>_<o> ()` and `E2EPW_Write_<p>_<o> ()` are 'const'.
AUTOSAR 3.2.1/4.0.1	The variables `Config` and `ConfigVal` are declared `static` and non-constant.
TTTech E2EPW 1.3	`Config` and `ConfigVal` are defined `static` and `const`.
AUTOSAR 4.2.1	`Config` was renamed to `Config_<p>_<o>` and is defined as static and const. `ConfigVal` was removed from the code examples in 4.2.1 as it is not required to achieve the desired functionality.
TTTech E2EPW 2.0	Backward-compatible to TTTech E2EPW 1.3.
Reason	▪ `static` is by specification in [AS_E2E_SWS] 68 . ▪ `const` moves the variables from (often sparse) RAM to ROM. **Note:** The variables have been renamed to `Config_<p>_<o>` and `ConfigVal_<p>_<o>` to satisfy MISRA rules. This has no effect on the E2EPW API.

Title	Redundant Wrapper not implemented
AUTOSAR	Specified as an optional feature in AUTOSAR.
TTTech E2EPW 1.3	Redundant Wrapper (`E2EPW_Write1/2_<p>_<o> ()`, `E2EPW_Read1/2_<p>_<o> ()`), which is not implemented. Only single channel is implemented.
AUTOSAR 4.2.1	No change.
TTTech E2EPW 2.0	Backward-compatible to TTTech E2EPW 1.3.

Title	Redundant Wrapper not implemented
Reason	The Redundant Wrapper is not an appropriate solution to achieve the system-wide ASIL D level. Another solution (e.g., ASIL D HW) would be a better approach. However, redundant channels can be modeled on SW-C level.

Title	Direct use of opaque parameter in `E2EPW_Write_<p>_<o> ()`
AUTOSAR 3.2.1/4.0.1	When the parameter `AppData` is opaque, `E2EPW_Write_<p>_<o> ()` copies the array to a local variable `Data`. Then it protects `Data` and copies `CRC` and `counter` in `Data` back to `AppData` and passes `AppData` to `Rte_Write_<p>_<o> ()`.
TTTech E2EPW 1.3	When the parameter `AppData` is opaque, `E2EPW_Write_<p>_<o> ()` passes `AppData` directly to `E2E_<profile>Protect` and `Rte_Write_<p>_<o> ()`. No local variable is used. Side-effect: ▪ `AppData` is modified. ▪ After `E2EPW_Write_<p>_<o> ()` returned, `CRC` and `counter` are visible to the application.
AUTOSAR 4.2.1	No change.
TTTech E2EPW 2.0	Backward-compatible to TTTech E2EPW 1.3.
Reason	Improved runtime performance (no copy), less memory consumption (no copy).

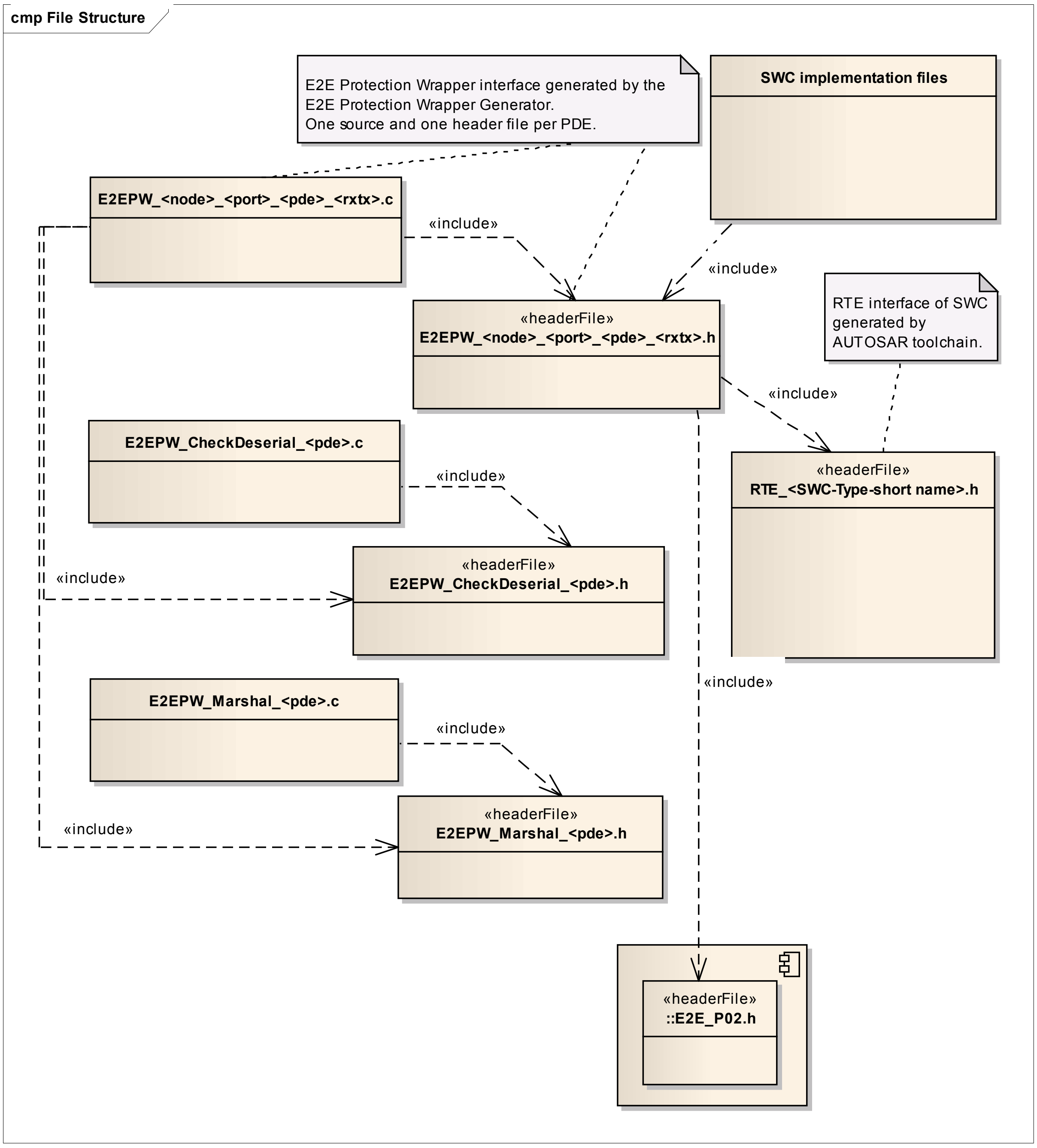
Title	The array `ppa_<port>_<vdp>_au8 []` is module-local.
AUTOSAR 3.2.1/4.0.1	The variable to store the marshaled version of a given PDE is local in `E2EPW_Read_<p>_<o> ()` and `E2EPW_Write_<p>_<o> ()`.
TTTech E2EPW 1.3	The variable is module-local and static.
AUTOSAR 4.2.1	The variable is still local in the code example.
TTTech E2EPW 2.0	No change to TTTech E2EPW 1.3.
Reason	`MemMap.h` is used for both, the function and the variable. `MemMap` sections cannot be stacked.

Title	Profile configuration is module-local.
AUTOSAR 3.2.1/4.0.1	The profile configuration constant `E2E_<profile>ConfigType` is local in `E2EPW_Read_<p>_<o> () and E2EPW_Write_<p>_<o> ()`.
TTTech E2EPW 1.3	The constant is module-local and static.
AUTOSAR 4.2.1	The profile configuration is defined as module-local as constant.
TTTech E2EPW 2.0	No change to TTTech E2EPW 1.3.
Reason	`MemMap.h` is used for both, the function and the variable. `MemMap` sections cannot be stacked.

Title	File Structure
AUTOSAR 3.2.1/4.0.1	A separate pair of files (`*.c` and `*.h`) for each SW-C, containing wrapper code for all protected data elements on this SW-C: • `E2EPW_<swc>.h` and • `E2EPW_<swc>.c`.
TTTech E2EPW 1.3	A separate pair of files (`*.c` and `*.h`) for each protected data element: • `E2EPW_<swc>_<p>_<o>.h` and • `E2EPW_<sws>_<p>_<o>.c`. The code for marshaling and deserialization check is also extracted into separate files: • `E2EPW_Marshal_<pde>.c` • `E2EPW_Marshal_<pde>.h` • `E2EPW_CheckDeserial_<pde>.c` • `E2EPW_CheckDeserial_<pde>.h` Where • `<swc>` is the `Node_Name` • `<p>` is the `Port_Name` • `<o>` is the `VDP_Name` • `<pde>` is the `PDE_Name`
AUTOSAR 4.2.1	The file structure has not changed since AUTOSAR 3.2.1/4.0.1.
TTTech E2EPW 2.0	No change to TTTech E2EPW 1.3.
Reason	Significantly simpler design of the E2EPWG. Shared code for marshaling/deserial check for the same PDE.

7 File Structure

The file structure of the generated files is shown in the figure below (for Profile 2[21]). For simplicity, inclusions of the files `StdTypes.h`, `MemMap.h`, `<swc>_MemMap.h` and of files stated in `Includes_H` and `Includes_C` are not shown.


cmp File Structure
E2E Protection Wrapper interface generated by the E2E Protection Wrapper Generator. One source and one header file per PDE.
SWC implementation files
E2EPW_<node>_<port>_<pde>_<rxtx>.c
«include»
«include»
«headerFile» E2EPW_<node>_<port>_<pde>_<rxtx>.h
RTE interface of SWC generated by AUTOSAR toolchain.
«include»
E2EPW_CheckDeserial_<pde>.c
«include»
«headerFile» RTE_<SWC-Type-short name>.h
«include»
«headerFile» E2EPW_CheckDeserial_<pde>.h
«include»
E2EPW_Marshal_<pde>.c
«include»
«include»
«headerFile» E2EPW_Marshal_<pde>.h
«headerFile» ::E2E_P02.h

File structure

8 Functional Specification

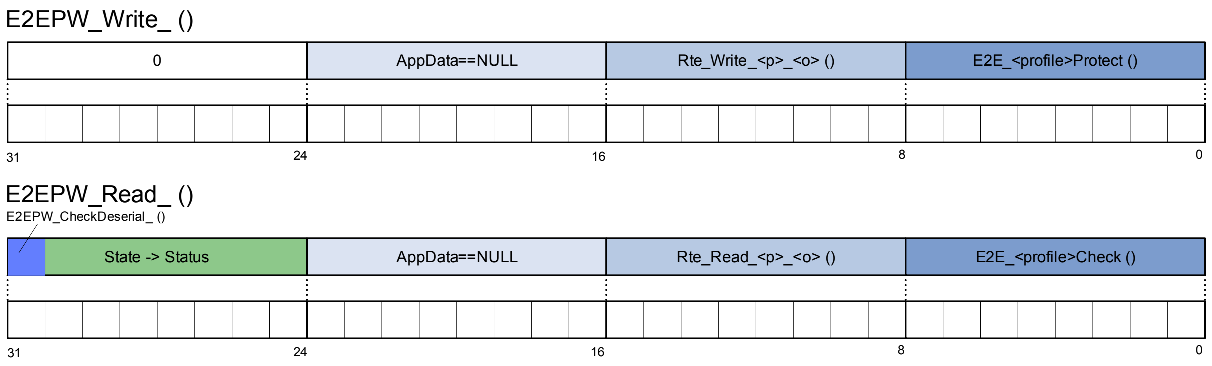
This Section contains the functional specification of the two main functions of the protection wrapper: **`E2EPW_Write_<p>_<o> ()`** and **`E2EPW_Read_<p>_<o> ()`**
.

8.1 Return values

The return values of the write and read function are a composition of several values.

Each step or function call within the write and read function results in a value that is incorporated into the return value of the function. As the further execution of the read or write function depends on the result of such a step or function call, the return value is referred to as status.

The status value is a 32-bit unsigned integer value which is composed as follows:



Note: The figures above show the logical bit and byte order. The generated code uses logical bit operations to maintain this logical representation regardless of the platform byte order.

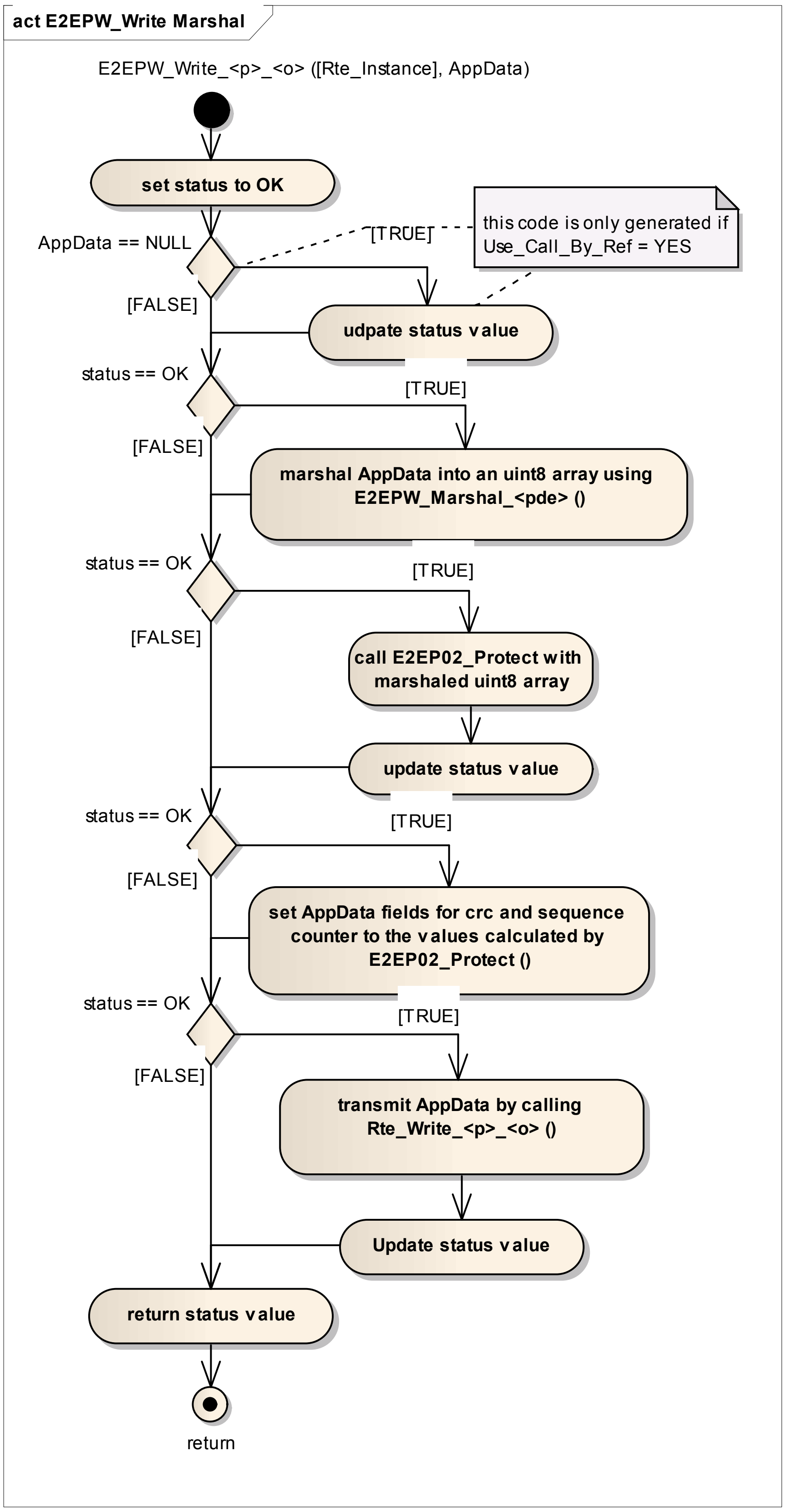
8.2 Function E2EPW_Write_<p>_<o> ()

Depending on the configuration, the **`E2EPW_Write_<p>_<o> ()`** function is generated differently.

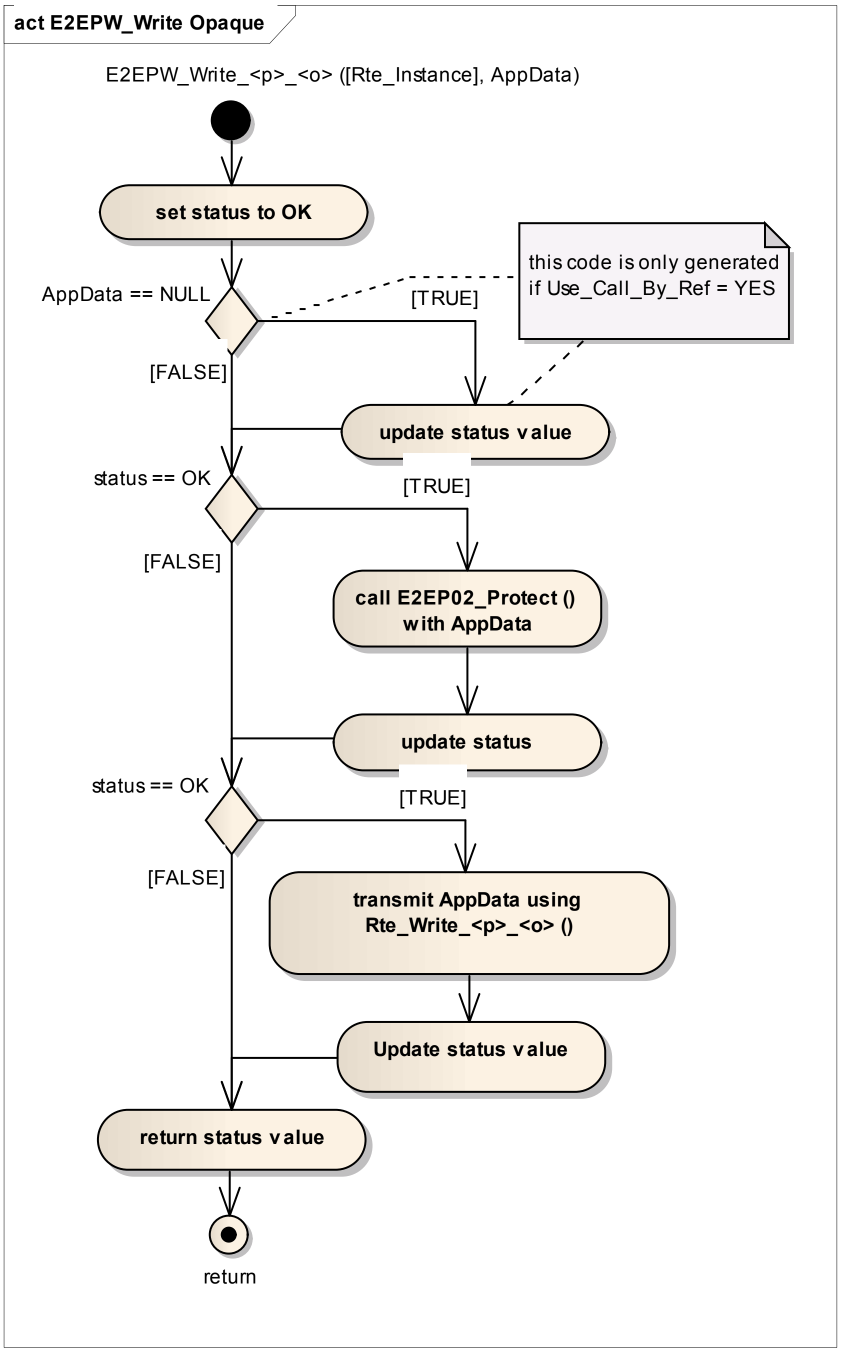
Options that affect the data flow are:

- `Use_Call_By_Ref`
- `IsOpaque`

The following figures show variants of the `E2EPW_Write_<p>_<o> ()` function:


act E2EPW_Write Marshal
E2EPW_Write_<p>_<o> ([Rte_Instance], AppData)
set status to OK
AppData == NULL
[TRUE]
this code is only generated if Use_Call_By_Ref = YES
[FALSE]
udpate status value
status == OK
[TRUE]
[FALSE]
marshal AppData into an uint8 array using E2EPW_Marshal_<pde> ()
status == OK
[TRUE]
[FALSE]
call E2EP02_Protect with marshaled uint8 array
update status value
status == OK
[TRUE]
[FALSE]
set AppData fields for crc and sequence counter to the values calculated by E2EP02_Protect ()
status == OK
[TRUE]
[FALSE]
transmit AppData by calling Rte_Write_<p>_<o> ()
Update status value
return status value
return

`E2EPW_Write_<p>_<o> ()` with `Use_Call_By_Ref = YES` and `Is_Opaque = NO`



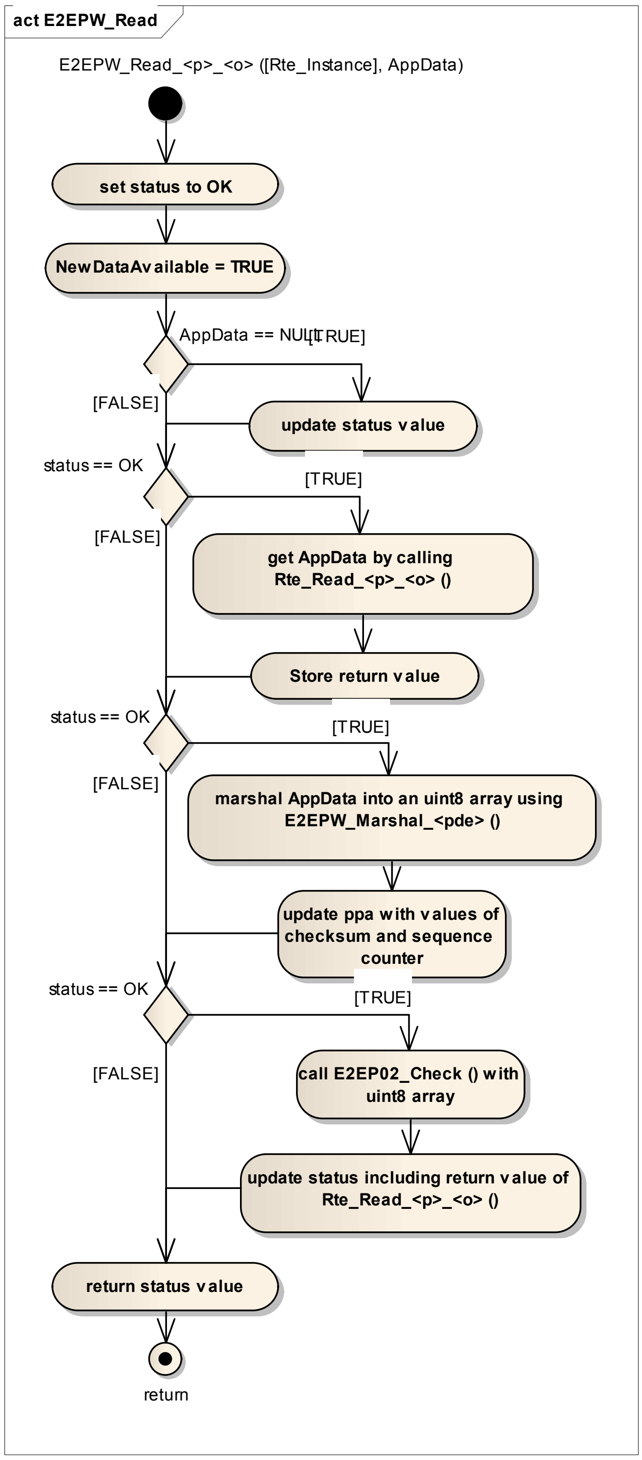
E2EPW_Write_<p>_<o> () with Use_Call_By_Ref = YES and Is_Opaque = YES

8.3 Function E2EPW_Read_<p>_<o> ()

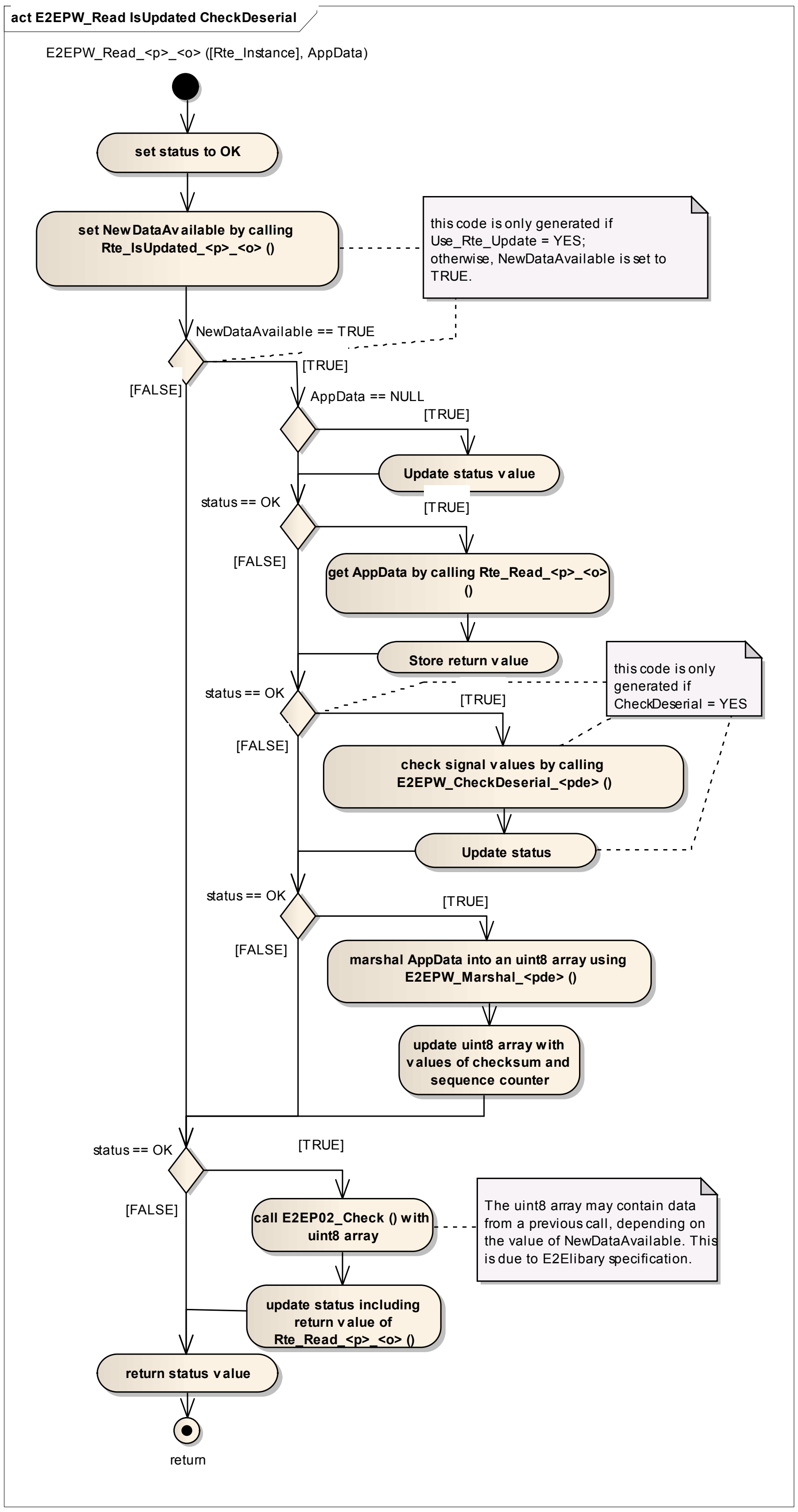
Depending on the configuration, the **`E2EPW_Read_<p>_<o> ()`** function is generated differently. Options that affect the data flow are:

- `CheckDeserial`
- `IsOpaque`
- `Use_Rte_Update`

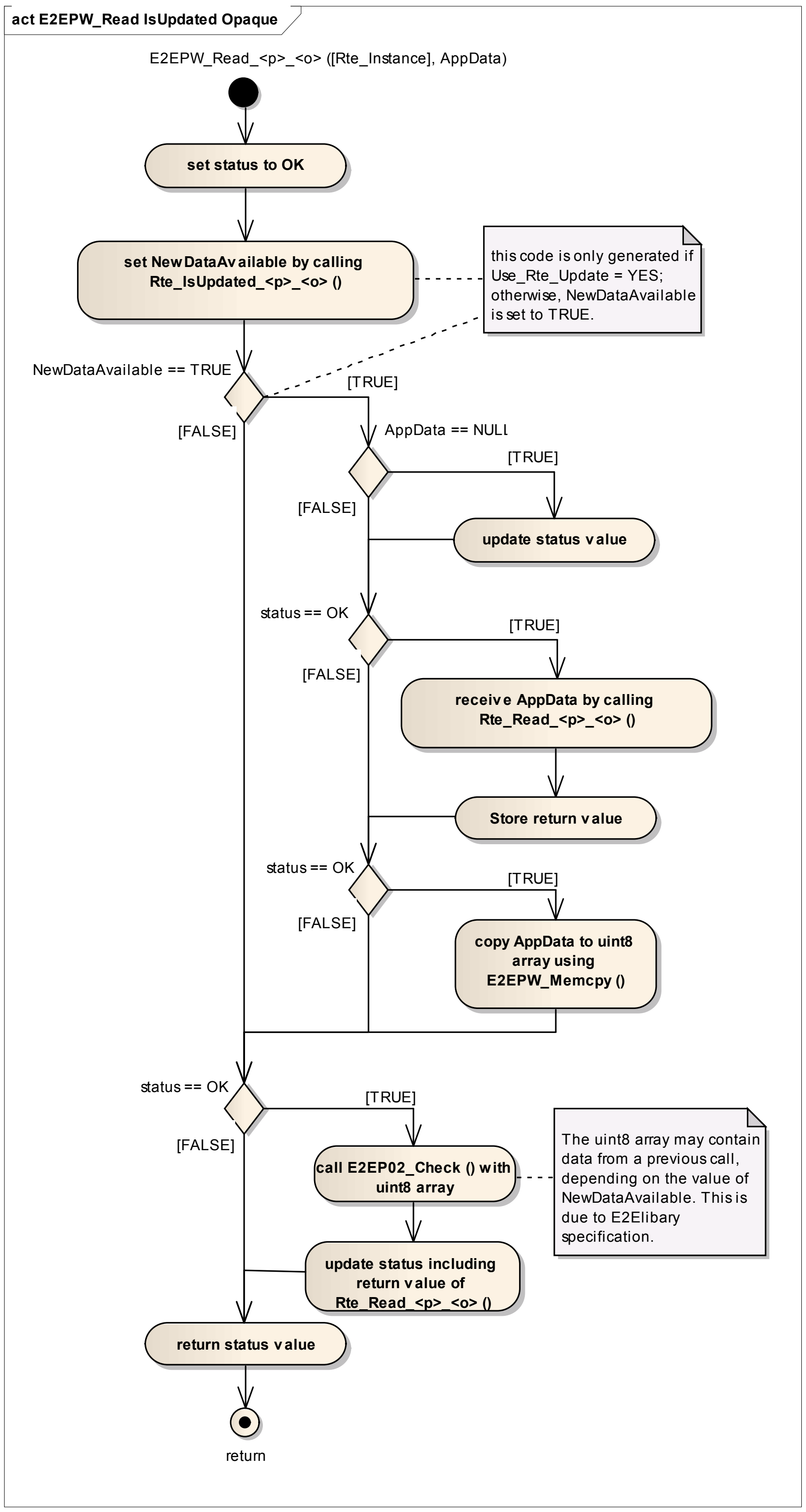
The following figures show variants of the `E2EPW_Read_<p>_<o> ()` function:


act E2EPW_Read
E2EPW_Read_<p>_<o> ([Rte_Instance], AppData)
set status to OK
NewDataAvailable = TRUE
AppData == NULL
[TRUE]
[FALSE]
update status value
status == OK
[TRUE]
[FALSE]
get AppData by calling Rte_Read_<p>_<o> ()
Store return value
status == OK
[TRUE]
[FALSE]
marshal AppData into an uint8 array using E2EPW_Marshal_<pde> ()
update ppa with values of checksum and sequence counter
status == OK
[TRUE]
[FALSE]
call E2EP02_Check () with uint8 array
update status including return value of Rte_Read_<p>_<o> ()
return status value
return

`E2EPW_Read_<p>_<o> ()`* with *`CheckDeserial=NO`*, *`Use_Rte_Update=NO`* and *`Is_Opaque=NO`


act E2EPW_Read IsUpdated CheckDeserial
E2EPW_Read_<p>_<o> ([Rte_Instance], AppData)
set status to OK
set NewDataAvailable by calling Rte_IsUpdated_<p>_<o> ()
this code is only generated if Use_Rte_Update = YES; otherwise, NewDataAvailable is set to TRUE.
NewDataAvailable == TRUE
[TRUE]
[FALSE]
AppData == NULL
[TRUE]
Update status value
status == OK
[TRUE]
[FALSE]
get AppData by calling Rte_Read_<p>_<o> ()
Store return value
this code is only generated if CheckDeserial = YES
status == OK
[TRUE]
[FALSE]
check signal values by calling E2EPW_CheckDeserial_<pde> ()
Update status
status == OK
[TRUE]
[FALSE]
marshal AppData into an uint8 array using E2EPW_Marshal_<pde> ()
update uint8 array with values of checksum and sequence counter
status == OK
[TRUE]
[FALSE]
call E2EP02_Check () with uint8 array
The uint8 array may contain data from a previous call, depending on the value of NewDataAvailable. This is due to E2Elibary specification.
update status including return value of Rte_Read_<p>_<o> ()
return status value
return

E2EPW_Read_<p>_<o> () **with** ***CheckDeserial = YES, Use_Rte_Update = YES*** **and** ***Is_Opaque = NO***


act E2EPW_Read IsUpdated Opaque
E2EPW_Read_<p>_<o> ([Rte_Instance], AppData)
set status to OK
set NewDataAvailable by calling Rte_IsUpdated_<p>_<o> ()
this code is only generated if Use_Rte_Update = YES; otherwise, NewDataAvailable is set to TRUE.
NewDataAvailable == TRUE
[TRUE]
[FALSE]
AppData == NULL
[TRUE]
[FALSE]
update status value
status == OK
[TRUE]
[FALSE]
receive AppData by calling Rte_Read_<p>_<o> ()
Store return value
status == OK
[TRUE]
[FALSE]
copy AppData to uint8 array using E2EPW_Memcpy ()
status == OK
[TRUE]
[FALSE]
call E2EP02_Check () with uint8 array
The uint8 array may contain data from a previous call, depending on the value of NewDataAvailable. This is due to E2Elibary specification.
update status including return value of Rte_Read_<p>_<o> ()
return status value
return

E2EPW_Read_<p>_<o> ()* with *CheckDeserial = NO*, *Use_Rte_Update = YES* and *Is_Opaque = YES

9 Environment Specifics

This Section explains the properties and restrictions of the build and runtime environment. They affect the Preprocessor, the E2EPWG and the E2EConfig file, and depend not only on the build environment, but also on the AUTOSAR revision, the use case, and the supported profiles of the protection wrapper generator and the preprocessor.

9.1 Vector DaVinci Developer/RTE Configurator for AUTOSAR 3.2

In this build environment, only the use case AUTOSAR RTE[23] is available. The Preprocessor and Generator support one or more of the E2Elib profiles P01[20] and P02[21], depending on the product license purchased.

The Preprocessor and Generator are integrated into this environment, and are automatically invoked when building the project. See the corresponding DaVinci user manuals for details.

9.1.1 Configuration Restrictions

There are several restrictions to the configuration when the Vector RTE for AUTOSAR 3.2 and AUTOSAR 4.0 is used:

- `IsOpaque` must always be `NO` (the RTE does not support opaque DEs)
- `Use_Call_By_Ref` must always be `YES` (only record-type DEs are supported by the RTE)
- `Use_Rte_Update` is `YES` by default. If the RTE in the given system does not support the function `Rte_Is_Updated_<p>_<o> ()`, then `Use_Rte_Update` must be `NO`. The preprocessor for this build environment generates E2EConfig files that conform to those restrictions. Restrictions relating to the preprocessor itself are described in the following Section.

9.1.2 Preprocessor Restrictions

- The preprocessor uses the name of the `VDP` (`VDP_Name`) for the `PDE_Name` and the SW-C name for the `Node_Name` in the E2EConfig file.

  The combination of the attributes `Node_Name` and `VDP_Name` in the configuration must be unique. As the `PDE_Name` can be chosen freely (it is only used in file names and internal functions such as `Marshal` and `CheckDeserial`), it can be changed manually in the E2EConfig file. With the DaVinci tools release R12, the DaVinci Developer supports SW-C-specific defines for the memory mapping (see also Other issues[52]). `Node_Name` must be the SW-C name because this identifier is used in the generated `MemMap` defines.
- The `Byte_Order_CPU` must be provided as command line parameter. The default value is `LITTLE_ENDIAN`. If the `Byte_Order_CPU` is not set to the correct value, signals with a length of more than 8 bits are not correctly marshaled.

9.1.3 E2EPW and RTE in a Safety-Related System

If a safety-relevant application developed according to ISO 26262 calls functions which are generated by a tool in QM quality, the *called* functions must be considered as safety-relevant as well, as they can potentially interfere with the *calling* application (e.g., by overwriting data, consuming execution time, etc.). Therefore, additional measures can be necessary to ensure **freedom from interference**. These can be applied at the function level (e.g., code reviews, tests, development process) or already at the system design level (**decoupling**).

In the present implementation, the generated E2EPW directly calls RTE functions which are generated by a tool in QM quality (i.e., the **DaVinci RTE Configurator**), which fits into the scenario described above. As the RTE functions are rather small and simple, both kinds of measures (code review and decoupling) are suitable, whereas the complex COM is almost impossible to be tackled without decoupling. The following example illustrates both cases.

Example:
A safety-related software component *SW-C A* sends a message using the E2EPW. As the RTE is generated by the RTE generator, the RTE code of SW-C A **must be reviewed manually** by the integrator to meet the requirements of ISO 26262. However, the function `Rte_Write_<p>_<o> ()` (see Functional specification[45]) calls some COM functions, which are also not developed according to ISO 26262. Therefore, the safety-related *SW-C A* sends the protected data element to another *SW-C B*. This *SW-C B* runs in a **different context** (i.e., in the same as the COM) and can call the COM directly. Thus the *SW-C B* is used as a proxy, and the E2EPW is only called in *SW-C A*. The RTE and the E2EPW must be configured accordingly.

9.2 Other Issues

- As of version 1.3, the E2EPWG uses the contents of the configuration field `Node_Name` to generate SW-C specific defines for the memory mapping. As of version 2.0.1, those must be defined in the corresponding `<swc>_MemMap.h` include file of the software component which are generated by the RTE gnerator. If not present, a compiler warning or error may be triggered.
- File names of the generated files can become very long. As the maximum length of most string-type fields in the E2EConfig file is 128 characters, it is possible that files cannot be generated due to file name or path length restrictions.

  For example, **Windows XP** and **Windows 7** restrict the **maximum total path length** to **260** characters (including the terminating null character). If `Node_Name`, `PDE_Name` and `VDP_Name` are very long, the generated files can easily exceed this length, especially if they are generated into a subdirectory. A possible solution is to choose a short `PDE_Name` and change it manually in the E2EConfig file.

10 Integration Notes

This Section contains further information and work instructions regarding the integration and integration testing of the **E2EPW** and the **E2Elib**.

It extends the requirements and comments of the *E2E Protection Wrapper Safety Manual* ([TT_E2EPW_SM] 68) and provides further explanations and code examples.

10.1 Checking the Tool Input

Make sure the settings in the Vector tools are as expected and the signals are mapped correctly to the signals on the bus.

This can be done by comparing the communication specification of the system specification with the settings in the Vector tools and the content of the generated E2EConfig file. A detailed description of the syntax and semantics of the E2EConfig file can be found in Section E2EConfig file 15.

10.2 Checking the Generated Files

Make sure that

- the E2EConfig file generated by the preprocessor is generated correctly (generation messages/errors on `stdout` – it is not sufficient to watch message on `stdout`, the E2EConfig file must be checked).
- the files generated by the E2EPWG (`pwg.exe`) are generated correctly (generation messages/errors on `stdout`).
- the files are compiled and linked correctly to the binary (`.elf` file, memory mapping; build environment messages/errors).

10.3 Performing an Integration Test

There are several ways to ensure that End-to-End Protection is integrated properly. The most common option is to build the target and perform an integration test by simulating the environment. For End-to-End Protection, this could be done by simulating the communication of the other ECUs on the communication network (restbus simulation 53). Here, the communication network is simulated, and messages are sent to and received from the target ECU. This can be done with or without a special test application, depending on the test requirements (unaltered target, test coverage, etc.), which also depends on the application using End-to-End Protection.

Another way to perform an integration test is a self-test using internal (intra-ECU) signaling 63. A sending and a receiving software component communicates with End-to-End Protection. After checking that End-to-End Protection is working when no fault is present, the sender produces sequences of messages simulating predefined faults.

10.3.1 Using Restbus Simulation

A common way to check the effectiveness of the End-to-End protection is to test it by simulating the bus communication. This also enables testing of the fault handling mechanisms of the receiving application using the End-to-End Protection. As the E2EPW only reports the status of the End-to-End protection, the test highly relies on the receiving application using the E2EPW and its behavior in case of communication faults.

The sender application does not get any information about the state of the End-to-End protection. The test of the sender side is therefore always straight forward: Send some predefined data, check if the checksum is correct regarding the sent data and the DataID, and check if the sequence counter is incremented with each message.

If there are no special requirements given by the application, a test for the End-to-End protection at the receiver shall systematically provoke the different communication status values the E2EPW can report:

E2Elib <profile> status defines	**Description**
`E2E_<profile>STATUS_INITIAL`	Initial (Message ok)
`E2E_<profile>STATUS_OK`	Message ok
`E2E_<profile>STATUS_WRONGCRC`	Wrong CRC
`E2E_<profile>STATUS_OKSOMELOST`	Some messages lost, but within tolerance
`E2E_<profile>STATUS_WRONGSEQUENCE`	Messages received in wrong sequence
`E2E_<profile>STATUS_REPEATED`	Message repeated
`E2E_<profile>STATUS_NONEWDATA`	No new message

By its design, the E2EPW only reports symptoms of communication faults, not their cause. Using additional information, the application can derive some possible causes, depending on assumptions regarding attributes of the system and communication behavior.

An example list of communication faults and the status reported by the E2EPW:

Status ***Fault***	**Wrong CRC**	**Messages lost**	**Invalid sequence**	**Message repeated**	**No new Message**
Bit flip	X				
Byte swap	X				
Wrong sender[1]	X				
Message(s) lost on bus		X	X		X
Message not sent (in time)		X			X
Message repeated at sender				X	

1 Refers to masquerading / message insertion

There are some faults that induce a characteristic sequence of status values. Some values depend on the system (communication) characteristics, others are reported in sequence:

For the scenario where messages are lost on the bus, a periodically checking receiver would yield a status depending on the following cases:

1. As long as no message has been received after a startup/restart, the status is `NONEWDATA`.
2. If a message has been received since the last startup/restart, then, depending on the number of lost messages and the configuration, the status might be either `messages lost` or `invalid sequence`, depending on the setting of `Max_Delta_Counter_Init`. Also, `message ok` or `message repeated` would also be possible if a multiple of 16 (15 for E2Elib profile 1) messages were lost consecutively. For the latter case, the sequence checking capabilities of End-to-End Protection would not be suitable if the system is expected to be capable of such a fault.

10.3.1.1 *Example Scenarios*

The following examples show how important the system attributes and assumptions are for the handling of communication faults.

Example 1:

Assuming a given system of a sender ECU and a receiver ECU, having the following properties:

1. The sender application is called periodically to send an end-to-end-protected message.
2. The sender communication stack periodically takes the last message sent by the sender application and sends it over the communication bus.
3. The receiver communication stack periodically checks for received messages on the bus.
4. The receiver application periodically queries the communication stack for received messages using the E2EPW.
5. All periods are equal; the tasks are in sync with each other.

The following fault is simulated: after message 3, the sender application misses its deadline for sending the end-to-end-protected message.

The result of the fault in the system: The sender communication stack does not get the next message (`counter=4`) and sends the last message (`counter=3`) again on the bus. The receiver receives the old message (`counter=3`) again. In the next communication cycle, the sender application sends the next message (`counter=5`) at the proper time. The sender communication stack takes the message (`counter=5`) and sends it on the bus.

Status values reported by the E2EPW at the receiver side:

Call Number	Sequence counter	Status
1	1	`E2E_<profile>STATUS_INITIAL`
2	2	`E2E_<profile>STATUS_OK`

Call Number	Sequence counter	Status
3	3	`E2E_<profile>STATUS_OK`
4	3	`E2E_<profile>STATUS_REPEATED`
5	5	`E2E_<profile>STATUS_OKSOMELOST`
6	6	`E2E_<profile>STATUS_OK`
…	…	…

Example 2:

In the next scenario, the same fault as in *Example 1* is introduced, but some of the above assumptions are extended:

- The sender communication stack uses an update bit (extends assumption 2).
- The receiver communication stack evaluates the update bit (extends assumption 3).
- The E2EPW checks for updated data (by using the RTE function `Rte_Is_Updated_<p>_<o> ()`; extends assumption 4).

Now the fault is perceived in a slightly different way:

Call Number	Sequence counter	Status
1	1	`E2E_<profile>STATUS_INITIAL`
2	2	`E2E_<profile>STATUS_OK`
3	3	`E2E_<profile>STATUS_OK`
4	3	`E2E_<profile>STATUS_NONEWDATA`
5	5	`E2E_<profile>STATUS_OKSOMELOST`
6	6	`E2E_<profile>STATUS_OK`
…	…	…

Here, instead of `Repeated`, `No new message` is reported.

Example 3:

As a next modification, let the sender communication stack use a queue for the sent messages:

Call Number	Sequence counter	Status
1	1	`E2E_<profile>STATUS_INITIAL`
2	2	`E2E_<profile>STATUS_OK`
3	3	`E2E_<profile>STATUS_OK`
4	3	`E2E_<profile>STATUS_NONEWDATA`
5	4	`E2E_<profile>STATUS_OK`

Call Number	Sequence counter	Status
6	5	`E2E_<profile>STATUS_OK`
…	…	…

Here, the sender misses the deadline for message 4, and as a result the messages are queued.The message with counter 4 is received in call 5, as there was no new message received before. By assuming that the receiving application was executed as specified, a timing violation can be derived by the receiving application when receiving message with counter 4 in call 5, when expecting message with counter 5.

Example 4:

Using the original assumptions, but with the following fault scenario:

After call 3, the receiver communication stack is not called before the receiver application tries to receive the next message.

The result of the fault on the system: the receiver application misses the deadline for message 4. It receives the old message with counter 3 again, and then the receiver communication stack is called and receives the new message with counter 4. Before the receiver application is called the next time, the receiver communication stack receives the next message, with counter 5.

Call Number	Sequence counter	Status
1	1	`E2E_<profile>STATUS_INITIAL`
2	2	`E2E_<profile>STATUS_OK`
3	3	`E2E_<profile>STATUS_OK`
4	3	`E2E_<profile>STATUS_REPEATED`
5	5	`E2E_<profile>STATUS_OKSOMELOST`
6	6	`E2E_<profile>STATUS_OK`
…	…	…

Here, the reported status values are the same as in the first scenario. The conclusion is that the cause of the perceived communication fault cannot be determined reliably without further information (which is out of the scope of the E2EPW).

10.3.1.2 *Integration Test Message Sequence*

In this Section, a small example integration test is provided using fixed sequences of messages to trigger all status values the E2EPW can report.

The following system properties are assumed:

1. The receiver communication stack periodically checks for received messages on the bus.
2. The receiver application periodically queries the communication stack for received messages using the E2EPW.
3. All periods are of equal duration; the tasks are ‘in sync’ with each other.
4. The receiver application uses the RTE function `Rte_IsUpdated_<p>_<o> ()` to determine if a new message is received since the last call of `Rte_Read_<p>_<o> ()`.
5. The E2Elib configuration has `Max_Delta_Counter_Init = 2`, `SyncCounterInit = 0`.

Each of the following tables is a message sequence. The message sequence of the first table is retrieved by sending valid messages and recording them. The message sequences of the other tables are modifications of the message sequence of the first table.

The meaning of the columns of the following tables is:

Call	The call number of the function `E2EPW_Read_<p>_<o> ().`
SC	The sequence counter value of the message sent over the bus by the E2EPW.
CRC	▪ `OK` if the CRC is as expected. ▪ `NOK` if the CRC is wrong.
Status	The expected status value reported by the E2EPW.
Description	A short description of the semantics of the simulated fault.
*****	For E2Elib profile 1 [20], there is no counter value 15. This line can be omitted.

Sequence of correct messages:

Call	SC	CRC	Status	Description
1	0	OK	E2E_<profile>STATUS_INITIAL	
2	1	OK	E2E_<profile>STATUS_OK	
3	2	OK	E2E_<profile>STATUS_OK	
4	3	OK	E2E_<profile>STATUS_OK	
5	4	OK	E2E_<profile>STATUS_OK	
6	5	OK	E2E_<profile>STATUS_OK	
7	6	OK	E2E_<profile>STATUS_OK	
8	7	OK	E2E_<profile>STATUS_OK	
9	8	OK	E2E_<profile>STATUS_OK	
10	9	OK	E2E_<profile>STATUS_OK	
11	10	OK	E2E_<profile>STATUS_OK	
12	11	OK	E2E_<profile>STATUS_OK	
13	12	OK	E2E_<profile>STATUS_OK	
14	13	OK	E2E_<profile>STATUS_OK	
15	14	OK	E2E_<profile>STATUS_OK	
16	15*	OK	E2E_<profile>STATUS_OK	

Deleted and repeated messages:

Call	SC	CRC	Status	Description
1	0	OK	E2E_<profile>STATUS_INITIAL	First message.

Call	SC	CRC	Status	Description
2	1	OK	E2E_<profile>STATUS_OK	
3	-	-	E2E_<profile>STATUS_NONEWDATA	No new message received.
4	3	OK	E2E_<profile>STATUS_OKSOMELOST	One message lost (2).
5	4	OK	E2E_<profile>STATUS_OK	
6	5	OK	E2E_<profile>STATUS_OK	
7	5	OK	E2E_<profile>STATUS_REPEATED	Message 5 received again.
8	7	OK	E2E_<profile>STATUS_OKSOMELOST	One message lost (6).
9	8	OK	E2E_<profile>STATUS_OK	
10	9	OK	E2E_<profile>STATUS_OK	
11	9	OK	E2E_<profile>STATUS_REPEATED	Message 9 received again.
12	10	OK	E2E_<profile>STATUS_OK	Message 10 received at call 11.
13	11	OK	E2E_<profile>STATUS_OK	Message 11 received at call 12.
14	13	OK	E2E_<profile>STATUS_OKSOMELOST	One message lost (12).
15	14	OK	E2E_<profile>STATUS_OK	
16	15*	OK	E2E_<profile>STATUS_OK	

Wrong CRC due to different failure modes:

Cycle	SC	CRC	Status	Description
1	0	OK	E2E_<profile>STATUS_INITIAL	
2	1	OK	E2E_<profile>STATUS_OK	
3	2	NOK	E2E_<profile>STATUS_WRONGCRC	Bit flip in message.
4	3	OK	E2E_<profile>STATUS_OKSOMELOST	One message lost (2).
5	4	OK	E2E_<profile>STATUS_OK	
6	5	NOK	E2E_<profile>STATUS_WRONGCRC	Used different DataID.
7	6	OK	E2E_<profile>STATUS_OKSOMELOST	One message lost (5).
8	7	OK	E2E_<profile>STATUS_OK	
9	8	OK	E2E_<profile>STATUS_OK	
10	13	NOK	E2E_<profile>STATUS_WRONGCRC	Bit flip in sequence counter signal.
11	14	NOK	E2E_<profile>STATUS_WRONGCRC	Bit flip in sequence counter signal.
12	11	OK	E2E_<profile>STATUS_OKSOMELOST	Two messages lost (with sequence counter values 9 and

Cycle	SC	CRC	Status	Description
				10).
13	12	OK	E2E_<profile>STATUS_OK	
14	13	OK	E2E_<profile>STATUS_OK	
15	14	OK	E2E_<profile>STATUS_OK	
16	15*	OK	E2E_<profile>STATUS_OK	

Sequence error because of swap and sender reset:

Cycle	SC	CRC	Status	Description
1	0	OK	E2E_<profile>STATUS_INITIAL	
2	1	OK	E2E_<profile>STATUS_OK	
3	2	OK	E2E_<profile>STATUS_OK	
4	4	OK	E2E_<profile>STATUS_OKSOMELOST	One message lost (3).
5	3	OK	E2E_<profile>STATUS_WRONGSEQUENCE	Messages 3 and 4 swapped.
6	5	OK	E2E_<profile>STATUS_OK	
7	6	OK	E2E_<profile>STATUS_OK	
8	-	-	E2E_<profile>STATUS_NONEWDATA	No new message received.
9	0	OK	E2E_<profile>STATUS_WRONGSEQUENCE	Sender was reset.
10	1	OK	E2E_<profile>STATUS_WRONGSEQUENCE	
11	2	OK	E2E_<profile>STATUS_WRONGSEQUENCE	
12	3	OK	E2E_<profile>STATUS_WRONGSEQUENCE	
13	4	OK	E2E_<profile>STATUS_WRONGSEQUENCE	
14	5	OK	E2E_<profile>STATUS_WRONGSEQUENCE	
15	6	OK	E2E_<profile>STATUS_REPEATED	
16	7	OK	E2E_<profile>STATUS_OK	

Sequence error because of receiver timing violation:

Call	SC	CRC	Status	Description
1	0	OK	`E2E_<profile>STATUS_INITIAL`	
2	1	OK	`E2E_<profile>STATUS_OK`	
3	2	OK	`E2E_<profile>STATUS_OK`	
4	-	-	`E2E_<profile>STATUS_NONEWDATA`	No new data received (extra call of receiver task simulated).
5	-	-	`E2E_<profile>STATUS_NONEWDATA`	No new data received (extra call of receiver task simulated).
6	5	OK	`E2E_<profile>STATUS_OKSOMELOST`	Two messages lost (3, 4).
7	6	OK	`E2E_<profile>STATUS_OK`	
8	10	OK	`E2E_<profile>STATUS_WRONGSEQUENCE`	Three messages lost because `E2EPW_Read_<p>_<o> ()` was not called three times in a row. Only two losses are tolerated because of `Max_Delta_Counter_Init = 2`. As the sequence counter of the sender increases, but the receiver expects values 7, 8 or 9, the E2EPW reports an error.
9	11	OK	`E2E_<profile>STATUS_WRONGSEQUENCE`	
10	12	OK	`E2E_<profile>STATUS_WRONGSEQUENCE`	
11	13	OK	`E2E_<profile>STATUS_WRONGSEQUENCE`	
12	14	OK	`E2E_<profile>STATUS_WRONGSEQUENCE`	
13	15*	OK	`E2E_<profile>STATUS_WRONGSEQUENCE`	
14	0	OK	`E2E_<profile>STATUS_WRONGSEQUENCE`	
15	1	OK	`E2E_<profile>STATUS_WRONGSEQUENCE`	
16	2	OK	`E2E_<profile>STATUS_WRONGSEQUENCE`	
17	3	OK	`E2E_<profile>STATUS_WRONGSEQUENCE`	
18	4	OK	`E2E_<profile>STATUS_WRONGSEQUENCE`	
19	5	OK	`E2E_<profile>STATUS_WRONGSEQUENCE`	
20	6	OK	`E2E_<profile>STATUS_REPEATED`	Got the same sequence counter as last valid message.
21	7	OK	`E2E_<profile>STATUS_OK`	Finally got a valid sequence counter.

10.3.1.3 *Hints for Integration Test Setup*

Using the tables in the Section above, it is necessary to be able to check the correctness of protected messages as well as to generate correct and incorrect messages according to the test sequences.

One way to establish this is to build a test environment that provides the E2Elib functionality and generates and checks protected messages dynamically. If this is not possible, static message sequences can be used:

1. For each data element received using the E2EPW, configure another data element that is sent using the E2EPW.
2. Send 16 messages and record them. Make sure that this sequence of messages corresponds to the message sequence listed in Table Sequence of correct messages[59] and make sure that the CRC is correct.
3. Build the message sequences of the other tables by adapting the recorded messages.
4. Use the message sequences to test the E2EPW for the data elements it is configured to receive and check the reported return values.

Note: To get the message sequences of the other tables, the original message sequence must be adapted (reordering, bit-manipulation,...). For message 5 in Table Wrong CRC due to different failure modes[60], the CRC must be calculated with a different message ID. This can be done with the sent messages, but by modifying the configuration of the E2EPW to use a different message ID and record message 5 of the sequence.

10.3.2 Using Intra-ECU Signaling

Another way to test the E2EPW and the E2Elib is to set up a software component that checks the functionality by sending end-to-end-protected messages, manipulating them according to some fault assumptions, and evaluating the reported status of the E2EPW on the receiver side. This can be done at any time, possibly at startup, regularly during runtime, or after a software update.

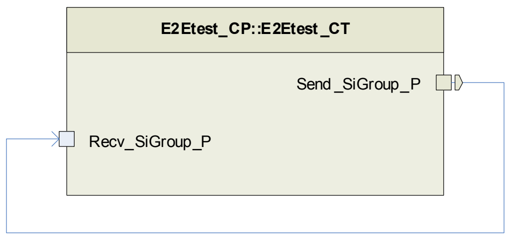
Note1: This test only covers the code of the E2Elib and the part of the E2EPW that is used for the protected messages sent/received. There is no test coverage for the actual E2EPW code used by the other applications, or for the code of the communication layers or the applications themselves.

Prerequisites: A sender and a receiver port using the E2EPW must be configured and connected. Both ports must use the same VDP.

Note 2: As those ports do not send to or receive from a communication bus, there is no tool support for adding End-to-End Protection (an I-PDU layout is needed for the marshaling). The configuration of End-to-End Protection must be done manually. A convenient way to do this is to reuse an existing data element and copy the signal definitions from there.

Note 3: The function `Rte_IsUpdated_<p>_<o> ()` must be emulated if it is used by `E2EPW_Read_<p>_<o> ()`, which is assumed in the message sequences described in this document. Therefore, the `Use_RteUpdate` must be `YES` in the E2EConfig file. The RTE generator must not generate the function, because the return value must be chosen according to the message sequence: Each time a message is sent by the sender, the function `Rte_IsUpdated_<p>_<o> ()` must return `TRUE` once. The user must provide this function.

As the RTE functions `Rte_Write_<p>_<o> ()` and `Rte_Read_<p>_<o> ()` only pass on pointers to the sent/received data structure, there is no need for multiple software components or even runnables for this test. However, transmission using multiple software components can be beneficial if memory partitioning is used, and is hence also subject to this test.



Software component setup

10.3.2.1 *Sending Correct Messages*

The **correct messages** sequence (see Table *Sequence of correct messages*[59]) is the simple part: The sender just sends messages using `E2EPW_Write_<p>_<o> ()`, and the receiver checks the reported status returned by `E2EPW_Read_<p>_<o> ()`.

10.3.2.2 *Sending Manipulated Messages*

For manipulated messages, the sender cannot use the function `E2EPW_Write_<p>_<o> ()` directly, because not all manipulation operations can be performed before or after the call of `E2EPW_Write_<p>_<o> ()` (e.g., the manipulation of the DataID used for the CRC calculation). Therefore, the sender must emulate the function `E2EPW_Write_<p>_<o> ()` and modify it according to the message sequence. The functional behavior of the E2EPW write function is as follows:

1. Build a byte array representing the I-PDU of the data element by calling the function `E2EPW_Marshal_<pde> ()`. The parameters are a pointer to the array and a pointer to the data element.
2. Call `E2E_<profile>Protect ()`, with the parameters being pointers to the `configuration` data structure, to the `status` data structure and to the I-PDU byte array.
3. Extract the counter and CRC value from the byte array (configuration-dependent position) and write them to the corresponding signals in the `data element` data structure.
4. Call `Rte_Write_<p>_<o> ()` with the data element as parameter.

To generate and send manipulated messages, the configuration and/or status of the E2Elib must be altered (sequence counter value, DataID, ...), and data manipulation may be necessary (to provoke a wrong CRC), before sending the data with `Rte_Write_<p>_<o> ()`. While the `status` data structure is available to the application, the `configuration` data structure is declared as static `const` in the `E2EPW_Write_<p>_<o> ()` function and therefore not accessible. The application must emulate `E2EPW_Write` with its own configuration data structure, which can then be altered if necessary.

The steps to send manipulated messages are:

1. Prepare the data element.
2. Call the function `E2EPW_Marshal_<pde> ()` with a pointer to the `data element` data structure and a pointer to a byte array in order to get a byte array representing the I-PDU of the data element.
3. Alter the `Counter` field of the `status` data structure to be `<Counter Value>` - 1 mod 16 (mod 15 for profile 1) according to the message sequence.
4. Alter the `configuration` data structure, if necessary (e.g., change the DataID field).
5. Call `E2E_<profile>Protect ()`, with the parameters being pointers to the `configuration` data structure, to the `status` data structure and to the I-PDU byte array.
6. Extract the counter and CRC value from the byte array according to the E2Elib configuration and write them to the signals in the `message` data structure.
7. Manipulate the `data element` data structure, if necessary (according to message sequence).
8. Call `Rte_Write_<p>_<o> ()` with a pointer to the `data element` data structure.
9. Maintain the value returned by `Rte_IsUpdated_<p>_<o> ()`.

11 Abbreviations

Abbreviation	Description
API	Application Programming Interface
ASIL	Application Safety Integrity Level
COM	Communication layer, a software layer that un/packs signal from a network PDU
CRC	Cyclic Redundancy Check
DE	Data Element
E2EConfig file	E2E Configuration File, which is the input for the E2EPWG
E2EPW	E2E Protection Wrapper
E2EPWG	E2E Protection Wrapper Generator
E2Elib	End-to-End Communication Protection Library
EBNF	Extended Backus-Naur-Form (see http://en.wikipedia.org/wiki/Extended_Backus Naur_Form)
ECU	Electronic Control Unit
QM	Quality Management (safety level according to company standards, but not ISO26262)
PA	Protected Area
PDE	Protected Data Element
I-PDU	Interaction Layer Protocol Data Unit
PGN	Parameter Group Number
RTE	Run-Time Environment
SC	Sequence Counter
SW-C, SWC	Software Component
VDP	Variable Data Prototype

12 Glossary

Term	Description
Communication stack	The Autosar software stack (in the context of E2EPW: Flexray or CAN stack).
Data Element (DE)	A C-like structure containing signal values. Used at application level. A DE is the smallest unit that can be transmitted/received with a single transmission/reception-function from the application view. If the transmission/reception of a Data Element is protected by the E2Elib, it is called a *Protected Data Element*.
(E2E)Protection Wrapper	A set of API functions that encapsulates the protection and check mechanisms of the E2Elib at application level: ▪ At sender side, there is an API function which adds CRC and counter to the data and passes it along to the next lower layer (RTE, COM or Transport layer). ▪ At reception side, there is an API function that receives data from the lower layer, checks its CRC and counter and returns a state telling if the data is correct or not.
Marshaling	The process of copying the DE (a C struct) to the I-PDU representation of the DE. The CRC evaluation is done on the I-PDU representation.
Protected Area	The E2EConfig file contains a list of Protected Areas. Each protected area holds all information that is required to generate wrapper code to send/receive a certain signal group using the COM layer: • information for the E2EPW API (e.g., the PDE name, node name, communication direction), • information for the calls of the COM API (e.g., signal group ID, opaque property), and • information on signals for marshaling (signal positions, signal lengths).
Protected Data Element	See *Data Element*.

13 References

[TT_E2EPW_SM] TTTech Automotive GmbH. *E2E Protection Wrapper Safety Manual*, SafetyManual_E2EPW.pdf, D-MSP-M-70-002

[AS_E2E_SWS] AUTOSAR. *Specification of SW-C End-to-End Communication Protection Library*, Version 1.2.0, Release 3.2.1, Annex B Section 12.1.

[TT_E2EP01_SM] TTTech Automotive GmbH, *E2Elib Profile-1 Safety Manual*, D-MSP-M-70-003, V 1.1.8, 2012

[TT_E2EP02_SM] TTTech Automotive GmbH, *E2Elib Profile-2 Safety Manual*, D-001-M-70-001, V 1.2.9, 2012

[BMW_LAST_KOMM] *Lastenheft Bordnetz-Kommunikation*, 10000235-000-09, BMW Group, 2009

[AS_COMABS_SWS] AUTOSAR Gbr, *Specification of Compiler Abstraction*, ID 051, V2.1.0, Rel. 3.2.1

[AS_MEM_SWS] AUTOSAR Gbr, *Specification of Memory Mapping*, ID 128, V1.2.1, Rel 3.2.1