Document Title	General Requirements specific to Adaptive Platform
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	714

Document Status	published
Part of AUTOSAR Standard	Adaptive Platform
Part of Standard Release	R19-11

Document Change History

Date	Release	Changed by	Description
2019-11-28	R19-11	AUTOSAR Release Management	• More design guidelines added • Changed Document Status from Final to published
2019-03-29	19-03	AUTOSAR Release Management	• No content changes.
2018-10-31	18-10	AUTOSAR Release Management	• More details to clause 1 Scope of document given • Former chapter 4.3 on Design requirements putted below chapter 4.2 Non-functional requirements • Following requirements have been revised: [RS_AP_00111], [RS_AP_00113], [RS_AP_00114], [RS_AP_00115], [RS_AP_00122], [RS_AP_00120], [RS_AP_00121], [RS_AP_00124], [RS_AP_00125] • Following requirements have been deleted: [RS_AP_00117], [RS_AP_00118] • Following requirements have been added: [RS_AP_00127], [RS_AP_00128], [RS_AP_00129], [RS_AP_00130], [RS_AP_00131], [RS_AP_00132], [RS_AP_00134]

2018-03-29	18-03	AUTOSAR Release Management	• Text entry for Supporting Material for [RS_AP_00111] • Text entry for Supporting Material for [RS_AP_00114] only refers now to ISO/IEC 14882 • Description of [RS_AP_00115] revised • Description of [RS_AP_00116], [RS_AP_00117], [RS_AP_00118], [RS_AP_00120], [RS_AP_00121], [RS_AP_00124], [RS_AP_00125] revised (in general "all ara libraries" changed to "all functional clusters").
2017-10-27	17-10	AUTOSAR Release Management	• Minor fixes
2017-03-31	17-03	AUTOSAR Release Management	• Initial release

Disclaimer

Table of Contents

1 Scope of this document

The goal of this document is to define a common set of basic requirements that apply to all **SWS documents** of the Adaptive Platform. Adaptive applications and functional cluster internals does not need to comply to these requirements.

2 Conventions to be used

The representation of requirements in AUTOSAR documents follows the table specified in [TPS_STDT_00078], see Standardization Template, chapter Support for Traceability ([1]).

The verbal forms for the expression of obligation specified in [TPS_STDT_00053] shall be used to indicate requirements, see Standardization Template, chapter Support for Traceability ([1]).

3 Acronyms and Abbreviations

There are no acronyms and abbreviations relevant within this document that are not included in the [2, AUTOSAR glossary].

4 Requirements Specification

4.1 Functional overview

4.2 Non-functional Requirements

[RS_AP_00111] The AUTOSAR Adaptive Platform shall support source code portability for AUTOSAR Adaptive applications. ⌈

Type:	valid
Description:	The AUTOSAR Adaptive platform shall support source code portability.
Rationale:	Ensure reuse of existing IPs.
Dependencies:	–
Use Case:	Integration of Adaptive Applications developed on different implementations of Adaptive Platform.
Supporting Material:	Any implementation of the Adaptive Platform shall allow successful compilation and linking of an Adaptive Application that uses ARA only as specified in the standard. No changes to the source code, and no conditional compilation constructs shall be necessary for this, if the application only uses constructs from the designated minimum C++ language version. The implementation may provide proprietary, non-ARA interfaces, as long as they are not contradicting with the AP standard. However, an implementation shall not add declarations or definitions that are not specified in an SWS to the namespace ara or any of its sub-namespaces.

⌋*(RS_Main_00150)*

[RS_AP_00130] AUTOSAR Adaptive Platform shall represent a rich and modern programming environment. ⌈

Type:	valid
Description:	AUTOSAR Adaptive Platform shall represent a rich and modern programming environment
Rationale:	Programmer productivity is an important aspect of any software framework. By providing and using advanced types and APIs, productivity is improved, and the platform's attractiveness increases.
Dependencies:	–
Use Case:	Some of these advanced types and APIs might be originally designed by AUTOSAR, whereas others might be back-ported from more recent C++ standards than defined by [RS_AP_00114].
Supporting Material:	–

⌋*(RS_Main_00420)*

4.2.1 Design Requirements

[RS_AP_00114] C++ interface shall be compatible with C++11. ⌈

Type:	valid
Description:	The interface of AUTOSAR Adaptive Platform shall be compatible with C++11.
Rationale:	The interface of AUTOSAR Adaptive platform is designed to be compatible with C++11 due to high availability of C++11 compiler for embedded devices. Nevertheless projects are free to use newer C++ version like C++14. Adaptive Platform vendors may restrict their package to a newer C++ version (e.g. to support newer build systems).
Dependencies:	RS_Main_00513
Use Case:	To manage the complexity of the application development, the Adaptive platform shall support object-oriented programming. C++ is the programming language which supports object-oriented programming and is best suited for performance-critical and real-time applications.
Supporting Material:	ISO/IEC 14882

⌋*(RS_Main_00513)*

[RS_AP_00115] Namespaces. ⌈

Type:	valid
Description:	The namespace of Adaptive Platform in global scope shall be "ara". Within "ara" namespace each Functional Cluster shall have exactly one own namespace with its shortname (defined in [3]). No other namespaces directly below "ara" are allowed. All names shall use lower-case only. Underscores may be used.
Rationale:	Harmonized look and feel.
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋*(RS_Main_00500, RS_Main_00150)*

[RS_AP_00116] Header file name. ⌈

Type:	valid
Description:	All Functional Clusters shall provide a self-contained header file for each public class (except scoped enum and exceptions). The header file name shall be derived from the class name. All header file names shall have the extensions .h.
Rationale:	Harmonized look and feel.
Dependencies:	–
Use Case:	–
Supporting Material:	Google C++ Style Guide: `https://google.github.io/styleguide/cppguide.html`

⌋*(RS_Main_00500, RS_Main_00150)*

[RS_AP_00122] Type names. ⌈

Type:	valid
Description:	For all Functional Clusters the name of their public types - classes, structs, type aliases, and type template parameters • shall be standardized in upper camel case. • underscores shall not be used. Except for fixed width integer types, postfix _t shall not be used. • capitalized acronyms shall be used as single words. Further the following exception is given: **exception:** all requirements and expectations that the C++ language standard or the C++ standard library place on the naming of certain symbols shall be heeded for all types and functions. Examples: nested type definitions that help with template metaprogramming such as value_type, size_type etc.
Rationale:	–
Dependencies:	–
Use Case:	Harmonized look and feel.
Supporting Material:	**CamelCase:** see [4] **STL:** see [5] **Google C++ Style Guide:** see [6]

⌋*(RS_Main_00500, RS_Main_00150)*

[RS_AP_00120] Method and Function names. ⌈

Type:	valid
Description:	For all Functional Clusters, the name of their public methods and functions shall use upper camel case. Further underscores shall not be used. Capitalized acronyms shall be used as single words. Further the following exceptions are given: **exception 1:** any function that fundamentally replicates a function which has been defined by an external standard (including, but not limited to, the C++ standard) shall keep that external standard's naming rules for that function, and for all symbols associated with it, including any external functions that are highly integrated with it. **exception 2:** all requirements and expectations that the C++ language standard or the C++ standard library place on the naming of certain symbols shall be heeded for all functions.
Rationale:	For the exceptions mentioned above the following rationals are given: **Rational for exception 2:** Certain special member functions and types cannot adopt the principal AUTOSAR naming rules, because their naming is defined by the C++ standard. Amongst these are: all operator functions, begin()/end() and all their variations, and virtual functions inherited from base classes of the C++ standard library.
Dependencies:	–
Use Case:	–
Supporting Material:	**CamelCase:** see [4] **STL:** see [5] **Google C++ Style Guide:** see [6]

⌋*(RS_Main_00500, RS_Main_00150)*

[RS_AP_00121] Parameter names. ⌈

Type:	valid
Description:	For all Functional Clusters, the name of parameters in public methods shall use lower camel case. Further underscores shall not be used. Capitalized acronyms shall be used as single words.
Rationale:	Harmonized look and feel.
Dependencies:	–
Use Case:	–
Supporting Material:	CamelCase: see [4]

⌋*(RS_Main_00500, RS_Main_00150)*

[RS_AP_00124] Variable names. ⌈

Type:	valid
Description:	For all Functional Clusters, the name of their public variables (like Common Variable names, Class Data Members and Struct Data Members) shall use lower camel case. Further underscores shall not be used. Capitalized acronyms shall be used as single words.
Rationale:	Harmonized look and feel.
Dependencies:	–
Use Case:	–
Supporting Material:	CamelCase: see [4]

⌋*(RS_Main_00500, RS_Main_00150)*

[RS_AP_00125] Enumerator and constant names. ⌈

Type:	valid
Description:	For all Functional Clusters, the name of public enumerations shall use upper camel case. The individual enumerators and constants shall be written with a leading "k" followed by upper camel case. Further underscores shall not be used. Capitalized acronyms shall be used as single words.
Rationale:	Harmonized look and feel.
Dependencies:	–
Use Case:	–
Supporting Material:	CamelCase: see [4]

⌋*(RS_Main_00500, RS_Main_00150)*

[RS_AP_00141] Usage of out parameters. ⌈

Type:	valid
Description:	Out parameters can be used for inplace modifications but shall not be used for returning values.
Rationale:	Harmonized look and feel.
Dependencies:	–
Use Case:	–
Supporting Material:	C++ Core Guidelines [7]: F.20: For "out" output values, prefer return values to output parameters.

⌋*(RS_Main_00150)*

[RS_AP_00119] Return values / application errors. ⌈

Type:	valid
Description:	All API function specifications shall give the exact list of errors (linked to the ErrorDomains which define them) that can originate from them, and which situations can cause which of those errors. Furthermore, for return values (especially integral, floating-point, enumeration, and string), the exact range of possible values shall be specified.
Rationale:	Harmonized look and feel.
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋*(RS_Main_00150)*

[RS_AP_00138] Return type of asynchronous function calls. ⌈

Type:	valid
Description:	Asynchronous function calls that need to return a value, or that can potentially fail should use ara::core::Future as return type.
Rationale:	Harmonized look and feel.
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋*(RS_Main_00150)*

[RS_AP_00139] Return type of synchronous function calls. ⌈

Type:	valid
Description:	Synchronous function calls that can potentially fail should use ara::core::Result as return type and use it for returning both values and errors.
Rationale:	Harmonized look and feel.
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋*(RS_Main_00150)*

[RS_AP_00142] Handling of unsuccessful operations. ⌈

Type:	valid
Description:	Functional Clusters shall differentiate recoverable unsuccessful operations from non-recoverable ones.
Rationale:	–
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋*(RS_Main_00010, RS_Main_00011)*

[RS_AP_00128] Error reporting. ⌈

Type:	valid
Description:	Interfaces shall be designed to report recoverable errors via a suitable return type, such as ara::core::Result or ara::core::Future.
Rationale:	Few compilers in the market allows to use exceptions in safety related projects.
Dependencies:	–
Use Case:	Safety-related projects
Supporting Material:	–

⌋*(RS_Main_00010, RS_Main_00012, RS_Main_00350)*

[RS_AP_00132] noexcept behavior of API functions ⌈

Type:	valid
Description:	Each library function having a wide contract that cannot throw or shall never throw should be marked as unconditionally noexcept.
Rationale:	–
Dependencies:	–
Use Case:	Safety-related projects
Supporting Material:	A function has a “wide contract” if it does not specify any undefined behavior. It therefore does not put any additional runtime constraints on its arguments, any object state, or any global state. The opposite of a “wide contract” is called a “narrow contract”. An example of a function with a wide contract would be `ara::core::Vector<T>::size()`. An example of a function with a narrow contract would be `ara::core::Vector<T>::front()`, because it has the precondition that the container must not be empty. This requirement is based on the “Adopted Guidelines” from the document N3279: see [8]

⌋*(RS_Main_00010, RS_Main_00012, RS_Main_00350)*

[RS_AP_00134] noexcept behavior of class destructors ⌈

Type:	valid
Description:	No class destructor should throw. They should use an explicitly supplied “noexcept” specifier.
Rationale:	–
Dependencies:	–
Use Case:	Safety-related projects
Supporting Material:	N3279: see [8]

⌋*(RS_Main_00010, RS_Main_00012, RS_Main_00350)*

[RS_AP_00133] noexcept behavior of move and swap operations ⌈

Type:	valid
Description:	If a library swap function, move-constructor, or move-assignment operator is conditionally-wide (i.e. can be proven to not throw by applying the noexcept operator) then it should be marked as conditionally noexcept. No other function should use a conditional noexcept specification.
Rationale:	–
Dependencies:	–
Use Case:	Safety-related projects
Supporting Material:	N3279: see [8]

⌋*()*

[RS_AP_00127] Usage of ara::core types. ⌈

Type:	valid
Description:	ARA interface shall use ara::core types instead of C++ standard types if ara::core provides the equivalent types.
Rationale:	–
Dependencies:	–
Use Case:	The ara::core types shall define common types in AP. Furthermore, it allows platform vendors to e.g. make use of own allocators for safety related projects.
Supporting Material:	–

⌋*(RS_Main_00010, RS_Main_00012, RS_Main_00350)*

[RS_AP_00129] Public types defined by functional clusters shall be designed to allow implementation without dynamic memory allocation. ⌈

Type:	valid
Description:	Public types defined by functional clusters shall be designed to allow implementation without dynamic memory allocation after the init-phase (i.e. after reaching Execution State Running of Execution Management).
Rationale:	–
Dependencies:	–
Use Case:	Safety related projects
Supporting Material:	Rule A18-5-7 of "Guidelines for the use of the C++14 language in critical and safety-related systems" [9]

⌋*(RS_Main_00010, RS_Main_00012, RS_Main_00350)*

[RS_AP_00135] Avoidance of shared ownership. ⌈

Type:	valid
Description:	APIs shall be designed in a way that the ownership of each data is unique. It should not span over the interface boundaries. Therefore usage of unique_ptr instead of shared_ptr shall be preferred. The ara::core::Future is preferred to synchronize asynchronous operations, instead of introducing own shared states.
Rationale:	It is conceptually simpler (defined ownership) and more predictable (you know who is responsible for destruction).
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋*(RS_Main_00010, RS_Main_00012, RS_Main_00350)*

[RS_AP_00136] Usage of string types. ⌈

Type:	valid
Description:	The default encoding of any string type (like ara::core::String or ara::core::StringView) in the ARA interfaces shall be UTF-8. In case the encoding is deviating from UTF-8, it shall be documented in the API definition (including the rationale as a note).
Rationale:	Harmonized usage
Dependencies:	–
Use Case:	Compatibility of strings in the platform
Supporting Material:	UTF-8: ISO/IEC 10646

⌋*(RS_Main_00010, RS_Main_00012, RS_Main_00350)*

[RS_AP_00137] Connecting run-time interface with model. ⌈

Type:	valid
Description:	Any reference of an API on application level to another element in the model shall refer to the other element using an ara::core::InstanceSpecifier. Modeling shall be done with PortPrototypes. No alternative methods of creating references to other elements in the model, such as FC-defined IDs are allowed.
Rationale:	Decoupling of interfaces and harmonized look and feel.
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋*(RS_Main_00160, RS_Main_00150, RS_Main_00513)*

[RS_AP_00140] Usage of "final specifier" in ara types. ⌈

Type:	valid
Description:	ARA types shall use the "final specifier", unless they are meant to be used as a base class.
Rationale:	Clear expression of the design (class hierarchy). Avoid problems that arise when deriving of a type which is not prepared for sub-classing.
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋*(RS_Main_00010, RS_Main_00012)*

5 Requirements Tracing

The following table references the requirements specified in [10] and links to the fulfillments of these.

Requirement	Description	Satisfied by
[RS_Main_00010]	AUTOSAR shall support the development of safety related systems	[RS_AP_00127] [RS_AP_00128] [RS_AP_00129] [RS_AP_00132] [RS_AP_00134] [RS_AP_00135] [RS_AP_00136] [RS_AP_00140] [RS_AP_00142]
[RS_Main_00011]	AUTOSAR shall support the development of reliable systems	[RS_AP_00142]
[RS_Main_00012]	AUTOSAR shall provide a software platform to support the development of highly available systems	[RS_AP_00127] [RS_AP_00128] [RS_AP_00129] [RS_AP_00132] [RS_AP_00134] [RS_AP_00135] [RS_AP_00136] [RS_AP_00140]
[RS_Main_00150]	AUTOSAR shall support the deployment and reallocation of AUTOSAR Application Software	[RS_AP_00111] [RS_AP_00115] [RS_AP_00116] [RS_AP_00119] [RS_AP_00120] [RS_AP_00121] [RS_AP_00122] [RS_AP_00124] [RS_AP_00125] [RS_AP_00137] [RS_AP_00138] [RS_AP_00139] [RS_AP_00141]
[RS_Main_00160]	AUTOSAR shall provide means to describe interfaces of the entire system	[RS_AP_00137]
[RS_Main_00350]	AUTOSAR specifications shall be analyzable and support according methods to demonstrate the achievement of safety related properties	[RS_AP_00127] [RS_AP_00128] [RS_AP_00129] [RS_AP_00132] [RS_AP_00134] [RS_AP_00135] [RS_AP_00136]
[RS_Main_00420]	AUTOSAR shall use established software standards and consolidate de-facto standards for basic software functionality	[RS_AP_00130]

[RS_Main_00500]	AUTOSAR shall provide naming conventions	[RS_AP_00115] [RS_AP_00116] [RS_AP_00120] [RS_AP_00121] [RS_AP_00122] [RS_AP_00124] [RS_AP_00125]
[RS_Main_00513]	AUTOSAR shall support language bindings for different programming languages	[RS_AP_00114] [RS_AP_00137]

6 References

[1] Standardization Template
AUTOSAR_TPS_StandardizationTemplate

[2] Glossary
AUTOSAR_TR_Glossary

[3] Functional Cluster Shortnames
AUTOSAR_TR_FunctionalClusterShortnames

[4] Camel case
https://en.wikipedia.org/wiki/CamelCase

[5] Standard Template Library
https://en.wikipedia.org/wiki/Standard_Template_Library

[6] Cpp Styleguide
https://google.github.io/styleguide/cppguide.html#Type_Names

[7] C++ Core Guidelines
https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md

[8] Conservative use of noexcept in the Library
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3279.pdf

[9] Guidelines for the use of the C++14 language in critical and safety-related systems
AUTOSAR_RS_CPP14Guidelines

[10] Main Requirements
AUTOSAR_RS_Main

7 Change History of this Document

7.1 Change History of this document according to AUTOSAR Release 19-11

7.1.1 Added Traceables in 19-11

Number	Heading
[RS_AP_00133]	noexcept behavior of move and swap operations
[RS_AP_00135]	Avoidance of shared ownership.
[RS_AP_00136]	Usage of string types.
[RS_AP_00137]	Connecting run-time interface with model.
[RS_AP_00138]	Return type of asynchronous function calls.
[RS_AP_00139]	Return type of synchronous function calls.
[RS_AP_00140]	Usage of "final specifier" in ara types.
[RS_AP_00141]	Usage of out parameters.
[RS_AP_00142]	Handling of unsuccessful operations.

Table 7.1: Added Traceables in 19-11

7.1.2 Changed Traceables in 19-11

Number	Heading
[RS_AP_00115]	Namespaces.
[RS_AP_00116]	Header file name.
[RS_AP_00119]	Return values / application errors.
[RS_AP_00122]	Type names.
[RS_AP_00127]	Usage of ara::core types.
[RS_AP_00128]	Error reporting.
[RS_AP_00129]	Public types defined by functional clusters shall be designed to allow implementation without dynamic memory allocation.
[RS_AP_00132]	noexcept behavior of API functions
[RS_AP_00134]	noexcept behavior of class destructors

Table 7.2: Changed Traceables in 19-11

7.1.3 Deleted Traceables in 19-11

Number	Heading
[RS_AP_00113]	API specification shall comply with selected coding guidelines.
[RS_AP_00131]	Use of verbal forms to express requirement levels.

Table 7.3: Deleted Traceables in 19-11