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	R24-11

Document Change History			
Date	**Release**	**Changed by**	**Description**
2024-11-27	R24-11	AUTOSAR Release Management	• Clarifications in relation to AP stabilization – The use of noexcept – Thread Safety – Versioning of Service Interface – Definition of rollback semantics • MISRA C++ 2023 • Own namespace for Platform Extension interfaces
2023-11-23	R23-11	AUTOSAR Release Management	• Clarifications
2022-11-24	R22-11	AUTOSAR Release Management	• Naming conventions for L&T Context ID added • Clarifications • Uptracing to RS Main fixed
2021-11-25	R21-11	AUTOSAR Release Management	• Guidance on error handling added • More design guidelines added • the sub-namespace ::internal is reserved for vendor-specific use
2020-11-30	R20-11	AUTOSAR Release Management	• More design guidelines for special member functions added • Support of C++ 14 added
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

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

The glossary below includes acronyms and abbreviations relevant to AP_RS_General that are not included in the AUTOSAR Glossary ([2]).

Abbreviation / Acronym:	**Description:**
failure transparent	• operations are guaranteed to succeed. If an error occurs, it will be handled internally and not observed by the caller of the operation • it has no defined errors nor exceptions • it is noexcept and does not have a Result nor Future return type
conditionally noexcept	A noexcept specifier with an argument that conditionally evaluates to either true or false. E.g., *noexcept(std::is_nothrow_move_constructible<T>)* or *noexcept(noexcept(T(std::forward<Args>(args)...)))*
standard Violation	AUTOSAR defines standardized Violations applicable for multiple `FunctionalCluster`s in [3, AP SWS Core].

Table 3.1: Acronyms and abbreviations used in the scope of this Document

4 Requirements Specification

4.1 Functional overview

4.2 Non-functional Requirements

[RS_AP_00111] Source Code Portability Support

Upstream requirements: RS_Main_00150

⌈

Description:	The AUTOSAR Adaptive platform shall support source code portability for AUTOSAR Adaptive applications.
Rationale:	Allow reuse of existing Adaptive Applications in another project.
Dependencies:	–
Use Case:	Integration of Adaptive Applications developed on different implementations of Adaptive Platform.
Supporting Material:	Adaptive Platform allows 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 need to be 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 do not contradict the AP standard.

⌋

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

Upstream requirements: RS_Main_00420

⌈

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:	–

⌋

4.2.1 Design Requirements

[RS_AP_00114] C++ interface shall be compatible with C++14

Upstream requirements: RS_Main_00001, RS_Main_00060

⌈

Description:	The interface of AUTOSAR Adaptive Platform shall be compatible with C++14.
Rationale:	The interface of AUTOSAR Adaptive platform is designed to be compatible with C++14 due to high availability of C++14 compiler for embedded devices. Nevertheless projects are free to use newer C++ version like C++17. 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_AP_00151] C++ Core Guidelines

Upstream requirements: RS_Main_00011, RS_Main_00350

⌈

Description:	AUTOSAR C++ APIs should follow the "C++ Core Guidelines" of May 11, 2024. The exceptions for hard-real-time systems shall apply. The AUTOSAR guidelines defined in RS-General [4] and/or MISRA C++:2023 Guidelines [5] (see [RS_AP_00167]) shall overrule the "C++ Core Guidelines" in case of conflict. If a part of the AUTOSAR C++ API cannot follow the "C++ Core Guidelines" for some other reason, its specification shall state the rationale (how this is done in detail, shall be aligned with the Architecture group)
Rationale:	These guidelines are well accepted in the market. Their aim is to help C++ programmers writing simpler, more efficient, more maintainable code.
Dependencies:	–
Use Case:	–
Supporting Material:	"C++ Core Guidelines" of May 11, 2024 [6]

⌋

[RS_AP_00167] MISRA C++:2023

Upstream requirements: RS_Main_00011, RS_Main_00350

⌈

Description:	AUTOSAR C++ APIs shall follow the MISRA C++:2023 Guidelines for the use C++:17 in critical systems [5]. The AUTOSAR guidelines defined in RS-General [4] shall overrule the "MISRA C++:2023 Guidelines" [5] in case of conflict. If a part of the AUTOSAR C++ API cannot follow the " MISRA C++:2023 Guidelines" [5] for some other reason, its specification shall state the rationale (how this is done in detail, shall be aligned with the Architecture group)
Rationale:	–
Dependencies:	–
Use Case:	–
Supporting Material:	MISRA C++:2023 Guidelines for the use C+:17 in critical systems; Published in October 2023 [5]

⌋

[RS_AP_00150] Provide only interfaces that are intended to be used by AUTOSAR Applications and Functional Clusters

Upstream requirements: RS_Main_00011, RS_Main_00350

⌈

Description:	AUTOSAR interfaces shall not define implementation details such as: • Classes, functions etc. that are not used in the application level or in platform extension APIs • Implementation inheritance in the public APIs • C++ SFINAE techniques of any kind • Class members with access specifier: `private` (except where the class *'borrows'* the specification from the corresponding specification in ISO and that ISO specification has standardized `private` members. AUTOSAR is therefore mandated to define those `private` members) see [3]
Rationale:	Provide only narrow interfaces to avoid coupling to implementation details. Hide implementation details because by AUTOSAR definition the implementation details are on the platform vendor.
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋

[RS_AP_00115] Public namespaces

Upstream requirements: RS_Main_00500, RS_Main_00150

⌈

Description:	The `top-level C++ namespace` `"ara"` is reserved globally for use by AUTOSAR application interfaces. • Within this `"ara"` namespace each Functional Cluster shall have one own namespace named as per [SWS_CORE_90025]. A namespace not in [SWS_CORE_90025] is forbidden. • Sub-namespaces below the own namespace are allowed (except the namespace `internal` see [RS_AP_00154]) • All names shall use lower-case only. • Underscores may be used. Inside "service" Functional Clusters, the namespace applies only to `ServiceInterface`s
Rationale:	Harmonized look and feel.
Dependencies:	–
Use Case:	–

⌋

[RS_AP_00174] PlatformExtension namespaces

Upstream requirements: RS_Main_00500, RS_Main_00150

⌈

Description:	The `top-level C++ namespace` `"apext"` is reserved globally for use by AUTOSAR Platform Extension Interfaces. • Within this `"apext"` namespace each Functional Cluster "may" have one own namespace named as per [SWS_CORE_90025]. A namespace not in [SWS_CORE_90025] is forbidden. • Sub-namespaces below the own namespace are allowed (except the namespace `internal` see [RS_AP_00154]) • All names shall use lower-case only. • Underscores may be used.
Rationale:	Harmonized look and feel.
Dependencies:	[RS_AP_00115] defines the namespaces for the application interfaces.
Use Case:	–

⌋

[RS_AP_00154] Internal namespaces

Upstream requirements: RS_Main_00500, RS_Main_00150

⌈

Description:	Within each Functional Cluster's namespace, the sub-namespace `::internal` shall be reserved for vendor-specific use.
Rationale:	–
Dependencies:	–
Use Case:	–

⌋

[RS_AP_00116] Header file name

Upstream requirements: RS_Main_00500, RS_Main_00150

⌈

Description:	All Functional Clusters should provide a self-contained header file for each public class. The header file name shall be derived from the class name. All header file names shall have the extension `.h`. Additional information: • Scoped enums, non-member functions, and exceptions are not required to have a self-contained header file. • If including multiple classes in one header file is imperative for usability this is also allowed. The header file name in this case shall be derived from one of the included class names.
Rationale:	Harmonized look and feel.
Dependencies:	–
Use Case:	–
Supporting Material:	Google C++ Style Guide: `https://google.github.io/styleguide/cppguide.html`

⌋

[RS_AP_00122] Type names

Upstream requirements: RS_Main_00500, RS_Main_00150

⌈

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 [7] **STL:** see [8] **Google C++ Style Guide:** see [9]

⌋

[RS_AP_00120] Method and Function names

Upstream requirements: RS_Main_00500, RS_Main_00150

⌈

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 [7] **STL:** see [8] **Google C++ Style Guide:** see [9]

⌋

[RS_AP_00121] Parameter names

Upstream requirements: RS_Main_00500, RS_Main_00150

⌈

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 [7]

⌋

[RS_AP_00124] Variable names

Upstream requirements: RS_Main_00500, RS_Main_00150

⌈

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 [7]

⌋

[RS_AP_00125] Enumerator and constant names

Upstream requirements: RS_Main_00500, RS_Main_00150

⌈

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 [7]

⌋

[RS_AP_00141] Usage of out parameters

Upstream requirements: RS_Main_00150

⌈

Description:	Out parameters shall not be used for returning values except for "expensive" in-place modifications. An example for such an exception would be the repeated retrieval of very large values using the same buffer to avoid repeated memory allocations.
Rationale:	Harmonized look and feel.
Dependencies:	–
Use Case:	–

▽

△

Supporting Material:	• See Architectural Decision in FO_EXP_SWArchitecturalDecisions [10] *Usage of out parameters*. • C++ Core Guidelines [6]: F.20: For "out" output values, prefer return values to output parameters.

⌋

[RS_AP_00119] Return values / application errors

Upstream requirements: RS_Main_00150

⌈

Description:	All API function specifications shall give the exact list of standardized 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. Additional information: APIs can be extended with vendor-specific error codes. These are not part of the AUTOSAR SWS specifications.
Rationale:	Harmonized look and feel.
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋

[RS_AP_00138] Return type of asynchronous function calls

Upstream requirements: RS_Main_00150

⌈

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_AP_00139] Return type of synchronous function calls

Upstream requirements: RS_Main_00150

⌈

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_AP_00142] Handling of unsuccessful operations

Upstream requirements: RS_Main_00010, RS_Main_00011

⌈

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

⌋

[RS_AP_00153] Assignment operators should restrict "this" to lvalues

Upstream requirements: RS_Main_00420

⌈

Description:	All specifications of assignment operators should be declared with the ref-qualifier &.
Rationale:	Assigning to temporaries is rarely, if ever, useful, and is more likely the result of a programming mistake. Adding the "&" ref-qualifier lets the compiler detect and reject such code.
Dependencies:	–
Use Case:	Safety-related projects
Supporting Material:	HIC++ v4.0, see [11]

⌋

[RS_AP_00144] Availability of a named constructor

Upstream requirements: RS_Main_00010, RS_Main_00012, RS_Main_00350

⌈

Description:	If the construction of an object can fail in a way that is recoverable by the caller, the class shall have named constructors returning a Result in addition to its regular constructors. Unless other considerations apply, the name of a named constructor should be *Create*, and its arguments shall be the same as those of the corresponding regular constructor. Named constructors shall be marked as noexcept.
Rationale:	All objects should be valid after their construction.
Dependencies:	–
Use Case:	–

▽

△

Supporting Material:	• See Architectural Decision in FO_EXP_SWArchitecturalDecisions [10] *Usage of named constructors for exception-less object creation* • C++ Core Guidelines [6]: C.42: If a constructor cannot construct a valid object, throw an exception.

⌋

[RS_AP_00145] Availability of special member functions

Upstream requirements: RS_Main_00011

⌈

Description:	The rule of five shall apply. If it is necessary to define or =delete any copy, move, or destructor function, define or =delete them all. It is necessary to define own constructors, if the default (or implicit created) constructors will not create valid and fully initialized object.
Rationale:	Consistency.
Dependencies:	–
Use Case:	–
Supporting Material:	• See Architectural Decision in FO_EXP_SWArchitecturalDecisions [10] *Usage of named constructors for exception-less object creation* • C++ Core Guidelines [6]: – C.21: If you define or =delete any copy, move, or destructor function, define or =delete them all – C.41: A constructor should create a fully initialized object

⌋

[RS_AP_00146] Classes whose construction requires interaction by the ARA framework

Upstream requirements: RS_Main_00011

⌈

Description:	A class which is not intended to be constructable by application shall delete the default constructor. This does not apply to abstract base classes, as they are not constructable by definition.
Rationale:	To show the intent that this class is not intended to be constructable by the application.
Dependencies:	–
Use Case:	–
Supporting Material:	See Architectural Decision in FO_EXP_SWArchitecturalDecisions [10] *Usage of named constructors for exception-less object creation*

⌋

[RS_AP_00147] Classes that are created with an InstanceSpecifier as an argument are not copyable, but at most movable.

Upstream requirements: RS_Main_00011

⌈

Description:	Classes that are created with an `ara::core::InstanceSpecifier` as an argument shall: • set copy constructor and operator to deleted, • optionally have a non-throwing move constructor and operator (noexcept).
Rationale:	To only have one way to construct the object and register the internals.
Dependencies:	–
Use Case:	–
Supporting Material:	See Architectural Decision in FO_EXP_SWArchitecturalDecisions [10] *Usage of named constructors for exception-less object creation*

⌋

[RS_AP_00170] InstanceSpecifierMappingIntegrityViolation ⌈

Description:	Constructors or named constructors that have an InstanceSpecifier as an argument shall define the `standard Violation` `InstanceSpecifierMappingIntegrityViolation` [SWS_CORE_13003] defined in [3, AP SWS Core]. Additional information: The `standard Violation` text shall be taken from the `Violation` message itself (i.e. the Doxygen Tag has no message text)

▽

△

Rationale:	The rationale to treat this as a `Violation` is that this is seen as an integration error which anyway cannot be handled by the caller of the API. Aborting execution is in line with the strategy to fail early.
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋

[RS_AP_00171] PortInterfaceMappingViolation ⌈

Description:	Constructors or named constructors that have an InstanceSpecifier as an argument shall define the `standard Violation` `PortInterfaceMappingViolation` [SWS_CORE_13004] defined in [3, AP SWS Core]. Additional information: The `standard Violation` text shall be: "The type of mapping does not match the expected type of PortInterface: *{portInterfaceTypeName}* referenced by a *{mappingTypeName}*."
Rationale:	The rationale to treat this as a `Violation` is that this is seen as an integration error which anyway cannot be handled by the caller of the API. Aborting execution is in line with the strategy to fail early.
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋

[RS_AP_00172] ProcessMappingViolation ⌈

Description:	Constructors or named constructors that have an InstanceSpecifier as an argument shall define the `standard Violation` `ProcessMappingViolation` [SWS_CORE_13005] defined in [3, AP SWS Core]. Additional information: The `standard Violation` text shall be taken from the `Violation` message itself (i.e. the Doxygen Tag has no message text)
Rationale:	The rationale to treat this as a `Violation` is that this is seen as an integration error which anyway cannot be handled by the caller of the API. Aborting execution is in line with the strategy to fail early.
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋

[RS_AP_00173] InstanceSpecifierAlreadyInUseViolation ⌈

Description:	Constructors or named constructors that have an InstanceSpecifier as an argument shall define the `standard Violation` `InstanceSpecifierAlreadyInUseViolation` [SWS_CORE_13006] defined in [3, AP SWS Core]. Additional information: The `standard Violation` text shall be taken from the `Violation` message itself (i.e. the Doxygen Tag has no message text)
Rationale:	The rationale to treat this as a `Violation` is that this is seen as an integration error which anyway cannot be handled by the caller of the API. Aborting execution is in line with the strategy to fail early.
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋

[RS_AP_00157] Existence of a copy constructor and a copy assignment ⌈

Description:	Classes for which the copy operation can fail with a recoverable error shall not implement copy constructors nor copy assignments.
Rationale:	–
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋

[RS_AP_00127] Usage of ara::core types

Upstream requirements: RS_Main_00010, RS_Main_00012, RS_Main_00350

⌈

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_AP_00143] Use 32-bit integral types by default

Upstream requirements: RS_Main_00002

⌈

Description:	Type aliases to integral types, and scoped enum base types should prefer 32-bit types over 16-bit or 8-bit ones.
Rationale:	Many CPUs lack instructions to handle such types efficiently.
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋

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

Upstream requirements: RS_Main_00010, RS_Main_00012, RS_Main_00350

⌈

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:	Memory allocator used in the project needs to guarantee that memory allocation and deallocation are executed within defined time constraints that are appropriate for the response time constraints defined for the real-time system and its programs.
Dependencies:	–
Use Case:	Safety related projects
Supporting Material:	See Architectural Decision in FO_EXP_SWArchitecturalDecisions [10] *Dynamic memory allocation*.

⌋

[RS_AP_00135] Avoidance of shared ownership

Upstream requirements: RS_Main_00010, RS_Main_00012, RS_Main_00350

⌈

Description:	APIs shall be designed in a way that the ownership of each data is unique. This is achieved either by transferring ownership between caller and callee (e.g. by means of std::move) or by creating a copy of data at the receiver. In case of ownership transfer usage of unique_ptr instead of shared_ptr shall be used. In case of asynchronous operations the type `ara::core::Future` shall be used to avoid introduction of own shared states.
Rationale:	Unique ownership is conceptually simpler and more predictable (responsibility for destruction) to manage.
Dependencies:	–
Use Case:	–
Supporting Material:	See Architectural Decision in FO_EXP_SWArchitecturalDecisions [10] *Use of local proxy objects for shared access to objects*.

⌋

[RS_AP_00136] Usage of string types

Upstream requirements: RS_Main_00010, RS_Main_00012, RS_Main_00350

⌈

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_AP_00137] Connecting run-time interface with model

Upstream requirements: RS_Main_00150

⌈

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_AP_00158] IAM access violations ⌈

Description:	Access to modeled resources shall be granted only to the process that is assigned to the given InstanceSpecifier mapped to the resource, otherwise it shall be treated as a Violation.
Rationale:	
Dependencies:	PortPrototype Process-Mapping
Use Case:	
Supporting Material:	

⌋

[RS_AP_00140] Usage of "final specifier"

Upstream requirements: RS_Main_00010, RS_Main_00012

⌈

Description:	ARA types shall use the "final specifier", unless they are meant to be used as a base class. All virtual functions of a non-final class that are not intended to be overwritten by a user of the API shall be final.
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:	See Architectural Decision in FO_EXP_SWArchitecturalDecisions [10] *Types defined in the Adaptive Runtime for Applications should be final.*

⌋

Note: The child classes of `ara::core::Exception` specified in the individual functional clusters are themselves meant to be used as base classes.

[RS_AP_00161] Arguments with an extended lifetime

Upstream requirements: RS_Main_00010, RS_Main_00012

⌈

Description:	APIs shall document the lifetime of their arguments if they deviate from the standard lifetime (valid in the context of the function call). Additional information: If the lifetime has to be documented, the description shall start with "**Lifetime:**" in a new line. There is no need to document if the argument • is passing its ownership, or • the argument is only valid in the context of the function call.
Rationale:	–
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋

[RS_AP_00148] Default arguments are not allowed in virtual functions

Upstream requirements: RS_Main_00010, RS_Main_00012

⌈

Description:	Default arguments shall not be used at all in virtual functions.
Rationale:	The according RQ of the "C++ core guidelines" are too weak .. (they state, that it needs to be made sure that a default argument is always the same) ... this would lead to code duplication with dependencies and high risks of inconsistencies, which can easily lead to unexpected behavior.
Dependencies:	–
Use Case:	–
Supporting Material:	C++ Core Guidelines [6]: C.140: Do not provide different default arguments for a virtual function and an overrider

⌋

[RS_AP_00155] Avoidance of cluster-specific initialization functions

Upstream requirements: RS_Main_00011

⌈

Description:	If a cluster needs an explicit initialization/de-initialization, it shall use `ara::core::Initialize`/`ara::core::Deinitialize`.
Rationale:	Avoidance of cluster-specific initialization functions.
Dependencies:	–

▽

△

Use Case:	–
Supporting Material:	–

⌋

[RS_AP_00156] Naming conventions for L&T Context ID

Upstream requirements: RS_Main_00500

⌈

Description:	Context IDs short names shall be 4 characters long with a prefix "#" (U+0023) and predefined for each Functional Cluster.
Rationale:	All AUTOSAR-defined context IDs are restricted to 4 ASCII characters in order to remain compatible with v1 of the DLT protocol, which only supports context IDs of 4 chars length.
Dependencies:	See [SWS_CORE_90024] for the resulting names.
Use Case:	Avoid naming clashes.
Supporting Material:	See [PRS_Dlt_01054] for the reserved Prefix in [12, Log and Trace Protocol Specification]

⌋

4.2.2 Error handling

[RS_AP_00128] Error reporting

Upstream requirements: RS_Main_00010, RS_Main_00012, RS_Main_00350

⌈

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:	–

⌋

Guidelines on recoverable errors:

- avoid "general error"-kinds of errors, e.g. *kGeneralError, kGenericError, kInternalError* - always strive to describe concrete error conditions.

Note: It is recommended that error codes are recoverable and not too generic, but also not too specific. It is recommended to use the same error code for the same error reaction. e.g. lost daemon connection, link down, IPC corrupt, TCP/IP driver error, buffer overflow should be merged to the general communication error in ara::com.

- for error codes originating from a 3rd-party standard (e.g. ISO), prefer to take over those error code names as close to the original definitions as possible, even if that violates other of these guidelines (prefer to follow AR formatting, though, e.g. follow the *kCamelCase* formatting)
- avoid to define error codes for non-recoverable errors

[RS_AP_00160] Classification of Behavior for Recoverable Error

Upstream requirements: RS_Main_00010, RS_Main_00012, RS_Main_00350

⌈

Description:	Functions with standardized recoverable errors or exceptions shall use a doxygen tag to classify the error behavior into one of the following:"rollback semantics" or "no rollback semantics"
Rationale:	Defining a set of pre-defined types of recoverable errors is an essential aspect to enable harmonized error handling and avoid undefined behavior.
Dependencies:	–
Use Case:	• rollback semantics: Operations can fail, but are guaranteed to have no negative side effects, leaving original data values intact. • no rollback semantics: Failed operations can result in negative side effects, but all invariants are preserved and there are no resource leaks (including memory leaks). This case is assumed to be the default, e.g. due to lack of further knowledge
Supporting Material:	Appendix E: Standard-Library Exception Safety in [13] Safety Profiles: [14]

⌋

Guidelines on classification of recoverable error: Rollback semantics should be restricted to no negative side effects. This should be the default for all functions that are not explicitly failure transparent. In the following see examples of side effects in communication:

- Negative side effect by sending a message incorrectly or with corrupted content
- Negative side effect by involving active network communication (using ara::com methods) because this may impact the communication server
- No negative side effect: get..() method - there are no negative side effects on the server
- No side effect: e.g. transmission of a message is not started at all.

In case there are functions that have no rollback semantics, they have to explicitly specify this per error code.

In case a function is failure transparent,

- operations are guaranteed to succeed. If an error occurs, it will be handled internally and not observed by the caller of the operation
- it has no defined errors nor exceptions
- it is `noexcept` and does not have a `ara::core::Result` nor `ara::core::Future` return type

Guidelines on the usage of error codes:

[RS_AP_00149] Error handling for non-initialized Functional Cluster

Upstream requirements: RS_Main_00010, RS_Main_00012, RS_Main_00350

⌈

Description:	Error codes for non-initialized Functional Cluster (i.e. when `ara::core::Initialize` has not been called) shall be avoided.
Rationale:	A call to a non-initialized API is treated as a violation because it is a systematic error.
Dependencies:	–
Use Case:	Safety-related projects
Supporting Material:	–

⌋

The foundation for the classification is what a user of the API has to consider. In the failure transparent case there are no possible error scenarios. So, the user can expect the function to always succeed. This is the most convenient case possible from a user perspective. The other two possible classes are defined per standardized error (which can be an exception or ErrorCode). In case a specific error has rollback semantics a user does not have to make special considerations while attempting to recover from the situation, because the state of the system did not change due to this failed call. Particularly, retrying the call can be done without issues. In contrast to this, in a situation without rollback semantics, additional considerations have to be made. The state of the system cannot be assumed to be the same as before the call. As a consequence, the user might have to manually trigger a more complex response.

For example if *ara::per::ResetKeyValueStorage* fails with the error *kAuthenticationFailed* this has rollback semantics. The state of the system is the same before as after this failed call. However, in the case of the error *kOutOfStorageSpace* on the same function it is not prescribed by the standard that the state must not have changed - hence the no rollback semantics classification. In that situation a user has to assume that some of the key-value pairs were reset while for others the reset could not be done due to missing physical storage space. In this case the user might have to release some storage that is not strictly required and only then retry to reset the Key-ValueStorage. Only after the call was successfully executed can a consistent state be assumed.

Guidelines on error naming:

- avoid "...Error" suffixes, e.g. *kBadSomethingError* - all these enum values are errors, there is no need to mention "Error" again
- prefer singular to plural form, e.g. prefer *kInvalidArgument* over *kInvalidArguments*, even if multiple arguments may be affected
- prefer to omit predicates, e.g. prefer *kSomethingNotValid* over *kSomething*Is*NotValid*
- prefer to omit verbs, e.g. prefer *kFileNotFound* over *kFile*Was*NotFound*
- American English has to be used: e.g. modeled(AE) over modelled(BE), canceled(AE) over cancelled(BE)
- get rid of redundant suffixes like "error" (e.g. by more fine-grained errors), or "is". Example: Communication*Is*Lost*Error* vs CommunicationLost (last should be used)
- verbs should be avoided. If it is unavoidable past is preferred, e.g. ...Failed and not ...Fails
- Prefer to phrase "failed-effort"-kind of error codes as "<Something>Failed", as opposed to e.g. "CouldNot<something>" or "FailedTo<something>"
- Prefer <Subject><Adverb> over <Adjective><Subject>, e.g. "ResourceBusy" rather than "BusyResource"

4.2.3 The use of noexcept

Since the error handling strategy in the AP is intended to be used in an exceptionless context many considerations that apply to "normal" C++ code and libraries do not apply to the AP. Therefore the guidance on the use of noexcept contradicts the strategy taken e.g., in the C++ STL.

Since errors are communicated through the `ara::core::Result` and `ara::core::Future` types as well as Violations instead of Exceptions, it is possible (even the default) to have functions with defined error conditions that still can not throw exceptions. This is the biggest difference in how the noexcept specifer needs to be interpreted depending on the context: In the context of AP noexcept does not mean that the function can not fail! Since there is the `ara::core::Result` and `ara::core::Future` error mechanism it is usually harmful to also allow exceptions. This is because it introduces inefficiencies. Also users of the API might have to also consider Exceptions which would make the API hard to use.

In contrast to the C++ standard, the AP specification tries to avoid undefined behavior wherever possible. One key mechanism for that are Violations ([SWS_CORE_00021]) that indicate nonrecoverable errors. Violations are often used in places where the C++ standard would define undefined behavior. That adds to gap between the two worlds, since a function with a defined violation can still have a wide contract and thus potentially be noexcept.

[RS_AP_00132] noexcept behavior of API functions

Status: OBSOLETE
Upstream requirements: RS_Main_00010, RS_Main_00012, RS_Main_00350

⌈

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.

⌋

[RS_AP_00133] noexcept behavior of move and swap operations

Status: OBSOLETE
Upstream requirements: RS_Main_00010, RS_Main_00012, RS_Main_00350

⌈

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 [15]

⌋

[RS_AP_00134] noexcept behavior of class destructors

Upstream requirements: RS_Main_00010, RS_Main_00012, RS_Main_00350

⌈

Description:	Class destructor shall not throw. They shall use an explicitly supplied “noexcept” specifier.
Rationale:	–
Dependencies:	–
Use Case:	Safety-related projects
Supporting Material:	N3279: see [15]

⌋

[RS_AP_00159] usage of “noexcept” specifier ⌈

Description:	Functions shall be annotated using the “noexcept” specifier **Additional Constraints:** • Functions should be `non-throwing` (i.e."noexcept(true)"), except functions that are intended to support error handling with exceptions. (Note: See list below the spec item) • Constructors that explicitly define exceptions shall be “noexcept(false)“. • Functions that are `failure transparent` shall be `non-throwing` (i.e."noexcept(true)"). • More specific requirements: – Functions that return an `ara::core::Result` or `ara::core::Future` shall be `non-throwing` (i.e."noexcept(true)"). – Default constructors, copy constructors, move constructors, copy assignment operators, move assignment operators, and other operators shall be `non-throwing` (i.e."noexcept(true)") if they exist. • Special case: Functions that are templated or are members of a templated class – May additionally also use `conditionally noexcept` in the “More specific requirements” cases. – Should still prefer the use of "noexcept(true)"
Rationale:	The use of a restrictive noexcept specifier wherever feasible restricts the to expected error behavior to the AP-specific exceptionless solutions and thus make the API more deterministic, more performant, and simpler to use.
Dependencies:	–
Use Case:	Safety-related projects
Supporting Material:	–

⌋

Examples of functions that are intend to support error handling with exceptions:

- `ara::core::Future::get`
- `ara::core::Result::ValueOrThrow`
- `ara::core::ErrorCode::ThrowAsException`
- `ara::core::ErrorDomain::ThrowAsException` and overriding functions

4.2.4 Thread Safety

[RS_AP_00163] Reentrancy of Functions ⌈

Description:	AUTOSAR API functions specified by Functional Clusters in Adaptive Platform shall be considered not reentrant unless stated otherwise.
Rationale:	In almost all cases where reentrancy is specified, a specification of thread-safety is sufficient, more appropriate, and easier to understand.
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋

[RS_AP_00164] Thread-safety of Functions ⌈

Description:	AUTOSAR API functions shall specify whether the function is thread-safe or not thread-safe by using either "thread-safe" or "not thread-safe" value in the API table describing the function.
Rationale:	–
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋

[RS_AP_00165] Concurrent Use of Different Objects ⌈

Description:	AUTOSAR APIs shall be designed such that different instances of AUTOSAR API classes can be used concurrently from different threads.
Rationale:	Usability, concurrency.
Dependencies:	–
Use Case:	–

▽

△

Supporting Material:	–

⌋

[RS_AP_00169] Thread-safety of Overriding Functions ⌈

Description:	Functions that override other functions specified in ara shall adhere to the thread-safety specification of the overridden function. Additional Information: That means if the overridden function is not thread-safe the overriding function can be not thread-safe or thread-safe. If the overridden function is thread-safe the overriding function has to be also thread-safe.
Rationale:	–
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋

[RS_AP_00162] Thread-safety of the Callback Function ⌈

Description:	Callback functions shall define the "Thread-safety" property in their API table.
	
	Additional Information:
	In the case of notification callbacks registered at runtime, the following pattern has to be used:
	
	`/// @arthreadsafety{%%%} <this is the thread-safety of how the callback has to be implemented>`
	`using ExampleCallback = std::function<void(void)>;`
	
	`/// @arthreadsafety{%%%} <this is the thread-safety of this ara-API>`
	`ara::core::Result<void> SetExampleCallback(ExampleCallback callback);`

Rationale:	–
Dependencies:	–
Use Case:	–
Supporting Material:	

⌋

4.2.5 Versioning of Service Interface API

On the Adaptive Platform the client and the provider of a service rely on a contract which covers the service interface and behavior. The interface and the behavior of a service may change over time. Therefore, service contract versioning shall be used to distinguish between the different versions of a service.

[RS_AP_00166] Versioning of ServiceInterfaces ⌈

Description:	Every Functional Cluster of the Adaptive Platform that provides a Service Interface shall also provide its version which consists of a majorVersion and a minorVersion numbers. The majorVersion number shall be increased when backwards-incompatible changes are introduced. The minorVersion number shall be increased when backwards-compatible changes are introduced.
Rationale:	–
Dependencies:	–
Use Case:	–
Supporting Material:	–

⌋

5 Requirements Tracing

The following table references the requirements specified in [16] and links to the fulfillments of these.

Requirement	Description	Satisfied by
[RS_Main_00001]	Real-Time System Software Platform	[RS_AP_00114]
[RS_Main_00002]	AUTOSAR shall provide a software platform for high performance computing platforms	[RS_AP_00143]
[RS_Main_00010]	Safety Mechanisms	[RS_AP_00127] [RS_AP_00128] [RS_AP_00129] [RS_AP_00132] [RS_AP_00133] [RS_AP_00134] [RS_AP_00135] [RS_AP_00136] [RS_AP_00140] [RS_AP_00142] [RS_AP_00144] [RS_AP_00148] [RS_AP_00149] [RS_AP_00160] [RS_AP_00161]
[RS_Main_00011]	Mechanisms for Reliable Systems	[RS_AP_00142] [RS_AP_00145] [RS_AP_00146] [RS_AP_00147] [RS_AP_00150] [RS_AP_00151] [RS_AP_00155] [RS_AP_00167]
[RS_Main_00012]	Highly Available Systems Support	[RS_AP_00127] [RS_AP_00128] [RS_AP_00129] [RS_AP_00132] [RS_AP_00133] [RS_AP_00134] [RS_AP_00135] [RS_AP_00136] [RS_AP_00140] [RS_AP_00144] [RS_AP_00148] [RS_AP_00149] [RS_AP_00160] [RS_AP_00161]
[RS_Main_00060]	Standardized Application Communication Interface	[RS_AP_00114]
[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_AP_00154] [RS_AP_00174]
[RS_Main_00350]	Documented Software Architecture	[RS_AP_00127] [RS_AP_00128] [RS_AP_00129] [RS_AP_00132] [RS_AP_00133] [RS_AP_00134] [RS_AP_00135] [RS_AP_00136] [RS_AP_00144] [RS_AP_00149] [RS_AP_00150] [RS_AP_00151] [RS_AP_00160] [RS_AP_00167]
[RS_Main_00420]	AUTOSAR shall use established software standards and consolidate de-facto standards for basic software functionality	[RS_AP_00130] [RS_AP_00153]
[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_AP_00154] [RS_AP_00156] [RS_AP_00174]

Table 5.1: Requirements Tracing

5.1 Trace Groups

Defined Trace Groups	
Identifier	**Included Requirements**
TG_AP_GeneralNonFunctional	[RS_AP_00111] [RS_AP_00114] [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_00127] [RS_AP_00128] [RS_AP_00129] [RS_AP_00130] [RS_AP_00132] [RS_AP_00133] [RS_AP_00134] [RS_AP_00135] [RS_AP_00136] [RS_AP_00137] [RS_AP_00138] [RS_AP_00139] [RS_AP_00140] [RS_AP_00141] [RS_AP_00142] [RS_AP_00143] [RS_AP_00144] [RS_AP_00145] [RS_AP_00146] [RS_AP_00147] [RS_AP_00148] [RS_AP_00149] [RS_AP_00150] [RS_AP_00151] [RS_AP_00153] [RS_AP_00154] [RS_AP_00155] [RS_AP_00156] [RS_AP_00157] [RS_AP_00158] [RS_AP_00159] [RS_AP_00160] [RS_AP_00161] [RS_AP_00162] [RS_AP_00163] [RS_AP_00164] [RS_AP_00165] [RS_AP_00166] [RS_AP_00167] [RS_AP_00169] [RS_AP_00170] [RS_AP_00171] [RS_AP_00172] [RS_AP_00173] [RS_AP_00174]

Table 5.2: Trace Groups of this document

6 References

[1] Standardization Template
AUTOSAR_FO_TPS_StandardizationTemplate

[2] Glossary
AUTOSAR_FO_TR_Glossary

[3] Specification of Adaptive Platform Core
AUTOSAR_AP_SWS_Core

[4] General Requirements specific to Adaptive Platform
AUTOSAR_AP_RS_General

[5] MISRA C++:2023: Guidelines for the use of C++17 in critical systems, ISBN 978-1911700104

[6] C++ Core Guidelines of May 11, 2024
https://github.com/isocpp/CppCoreGuidelines/blob/50afe02/CppCoreGuide-lines.md

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

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

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

[10] Explanation of Adaptive and Classic Platform Software Architectural Decisions
AUTOSAR_FO_EXP_SWArchitecturalDecisions

[11] High Integrity C++ (HIC++) v4.0
https://www.perforce.com/resources/qac/high-integrity-cpp-coding-standard

[12] Log and Trace Protocol Specification
AUTOSAR_FO_PRS_LogAndTraceProtocol

[13] The C++ Programming Language (PDF) (3rd ed.)

[14] Safety Profiles: Type-and-resource Safe programming in ISO Standard C++ (Doc. no. P2816R0)

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

[16] Main Requirements
AUTOSAR_FO_RS_Main

A Change History of this Document

A.1 Change History of this document according to AUTOSAR Release 24-11

A.1.1 Added Requirements in R24-11

Number	Heading
[RS_AP_00157]	Existence of a copy constructor and a copy assignment
[RS_AP_00158]	IAM access violations
[RS_AP_00159]	usage of "noexcept" specifier
[RS_AP_00160]	Classification of Behavior for Recoverable Error
[RS_AP_00161]	Arguments with an extended lifetime
[RS_AP_00162]	Thread-safety of the Callback Function
[RS_AP_00163]	Reentrancy of Functions
[RS_AP_00164]	Thread-safety of Functions
[RS_AP_00165]	Concurrent Use of Different Objects
[RS_AP_00166]	Versioning of ServiceInterfaces
[RS_AP_00167]	MISRA C++:2023
[RS_AP_00169]	Thread-safety of Overriding Functions
[RS_AP_00170]	InstanceSpecifierMappingIntegrityViolation
[RS_AP_00171]	PortInterfaceMappingViolation
[RS_AP_00172]	ProcessMappingViolation
[RS_AP_00173]	InstanceSpecifierAlreadyInUseViolation
[RS_AP_00174]	PlatformExtension namespaces

Table A.1: Added Requirements in R24-11

A.1.2 Changed Requirements in R24-11

Number	Heading
[RS_AP_00111]	Source Code Portability Support
[RS_AP_00115]	Public namespaces
[RS_AP_00116]	Header file name
[RS_AP_00119]	Return values / application errors
[RS_AP_00120]	Method and Function names
[RS_AP_00121]	Parameter names
[RS_AP_00122]	Type names

▽

△

Number	Heading
[RS_AP_00124]	Variable names
[RS_AP_00125]	Enumerator and constant names
[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_00133]	noexcept behavior of move and swap operations
[RS_AP_00134]	noexcept behavior of class destructors
[RS_AP_00135]	Avoidance of shared ownership
[RS_AP_00140]	Usage of "final specifier"
[RS_AP_00141]	Usage of out parameters
[RS_AP_00144]	Availability of a named constructor
[RS_AP_00145]	Availability of special member functions
[RS_AP_00146]	Classes whose construction requires interaction by the ARA framework
[RS_AP_00147]	Classes that are created with an InstanceSpecifier as an argument are not copyable, but at most movable.
[RS_AP_00149]	Error handling for non-initialized Functional Cluster
[RS_AP_00150]	Provide only interfaces that are intended to be used by AUTOSAR Applications and Functional Clusters
[RS_AP_00151]	C++ Core Guidelines
[RS_AP_00155]	Avoidance of cluster-specific initialization functions
[RS_AP_00156]	Naming conventions for L&T Context ID

Table A.2: Changed Requirements in R24-11

A.1.3 Deleted Requirements in R24-11

Number	Heading
[RS_AP_00152]	Faults inside constructor.

Table A.3: Deleted Requirements in R24-11

A.2 Change History of this document according to AUTOSAR Release 23-11

A.2.1 Added Requirements in R23-11

none

A.2.2 Changed Requirements in R23-11

Number	Heading
[RS_AP_00115]	Public namespaces.
[RS_AP_00132]	noexcept behavior of API functions
[RS_AP_00144]	Availability of a named constructor.
[RS_AP_00147]	Classes that are created with an InstanceSpecifier as an argument are not copyable, but at most movable.
[RS_AP_00156]	Naming conventions for L&T Context ID.

Table A.4: Changed Requirements in R23-11

A.2.3 Deleted Requirements in R23-11

none

A.3 Change History of this document according to AUTOSAR Release 22-11

A.3.1 Added Requirements in R22-11

Number	Heading
[RS_AP_00156]	Naming conventions for L&T Context ID.

Table A.5: Added Requirements in R22-11

A.3.2 Changed Requirements in R22-11

Number	Heading
[RS_AP_00114]	C++ interface shall be compatible with C++14.
[RS_AP_00137]	Connecting run-time interface with model.
[RS_AP_00141]	Usage of out parameters.
[RS_AP_00143]	Use 32-bit integral types by default.
[RS_AP_00145]	Availability of special member functions.
[RS_AP_00146]	Classes whose construction requires interaction by the ARA framework.

▽

△

Number	Heading
[RS_AP_00147]	Classes which are created by an InstanceSpecifer shall not be copyable, but at most movable.

Table A.6: Changed Requirements in R22-11

A.3.3 Deleted Requirements in R22-11

none

A.4 Change History of this document according to AUTOSAR Release 21-11

A.4.1 Added Requirements in R21-11

Number	Heading
[RS_AP_00148]	Default arguments are not allowed in virtual functions.
[RS_AP_00149]	Guidance on error handling.
[RS_AP_00150]	Provide only interfaces that are intended to be used by AUTOSAR applications and other Functional Clusters.
[RS_AP_00151]	C++ Core Guidelines.
[RS_AP_00152]	Faults inside constructor.
[RS_AP_00153]	Assignment operators should restrict "this" to lvalues
[RS_AP_00154]	Internal namespaces.
[RS_AP_00155]	Avoidance of cluster-specific initialization functions.

Table A.7: Added Requirements in R21-11

A.4.2 Changed Requirements in R21-11

Number	Heading
[RS_AP_00111]	The AUTOSAR Adaptive Platform shall support source code portability for AUTOSAR Adaptive applications.
[RS_AP_00115]	Public namespaces.
[RS_AP_00129]	Public types defined by functional clusters shall be designed to allow implementation without dynamic memory allocation.
[RS_AP_00133]	noexcept behavior of move and swap operations

▽

△

Number	Heading
[RS_AP_00135]	Avoidance of shared ownership.
[RS_AP_00140]	Usage of "final specifier" in ara types.
[RS_AP_00141]	Usage of out parameters.
[RS_AP_00144]	Availability of a named constructor.
[RS_AP_00145]	Availability of special member functions.
[RS_AP_00146]	Classes whose construction requires interaction by the ARA framework.
[RS_AP_00147]	Classes which are created by an InstanceSpecifer shall not be copyable, but at most movable.

Table A.8: Changed Requirements in R21-11

A.4.3 Deleted Requirements in R21-11

none

A.5 Change History of this document according to AUTOSAR Release 20-11

A.5.1 Added Requirements in R20-11

Number	Heading
[RS_AP_00143]	Use 32-bit integral types by default.
[RS_AP_00144]	Availability of a named constructor.
[RS_AP_00145]	Availability of special member functions.
[RS_AP_00146]	Classes whose construction requires interaction by the ARA framework.
[RS_AP_00147]	Classes which are created by an InstanceSpecifer shall not be copyable, but at most movable.

Table A.9: Added Requirements in R20-11

A.5.2 Changed Requirements in R20-11

Number	Heading
[RS_AP_00114]	C++ interface shall be compatible with C++14.

▽

△

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

Table A.10: Changed Requirements in R20-11

A.5.3 Deleted Requirements in R20-11

none

A.6 Change History of this document according to AUTOSAR Release 19-11

A.6.1 Added Requirements 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 A.11: Added Requirements in 19-11

A.6.2 Changed Requirements 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.

▽

△

Number	Heading
[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 A.12: Changed Requirements in 19-11

A.6.3 Deleted Requirements 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 A.13: Deleted Requirements in 19-11