Document Title	Explanation of Adaptive and Classic Platform Software Architectural Decisions
Document Category	Explanation (EXP)
Document UID	1078
Document Status	published
Part of AUTOSAR Standard	Foundation
Part of Standard Release	R26-11

Document Change History			
Date	**Release**	**Changed by**	**Description**
2026-11-26	R26-11	AUTOSAR Release Management	• Added decision on guidelines for hardware abstraction in AP • Removed obsolete decision on the V2X stack in CP
2025-11-27	R25-11	AUTOSAR Release Management	• Added architectural decision for release R25-11
2024-11-27	R24-11	AUTOSAR Release Management	• Added architectural decisions for release R24-11 • Clarified the use of the `final` specifier in "Final specifier for types and virtual member functions" • Removed obsolete decisions
2023-11-23	R23-11	AUTOSAR Release Management	• Added architectural decisions for release R23-11 • Clarified the expected handling of errors in architectural decision "Harmonized error handling for lost daemon connection" • Adapted architectural decision " Granularity of diagnostics" due to the removal of structural dependencies between Software Clusters
2022-11-24	R22-11	AUTOSAR Release Management	• Initial release

Disclaimer

This work (specification and/or software implementation) and the material contained in it, as released by AUTOSAR, is for the purpose of information only. AUTOSAR and the companies that have contributed to it shall not be liable for any use of the work.

The material contained in this work is protected by copyright and other types of intellectual property rights. The commercial exploitation of the material contained in this work requires a license to such intellectual property rights.

This work may be utilized or reproduced without any modification, in any form or by any means, for informational purposes only. For any other purpose, no part of the work may be utilized or reproduced, in any form or by any means, without permission in writing from the publisher.

The work has been developed for automotive applications only. It has neither been developed, nor tested for non-automotive applications.

The word AUTOSAR and the AUTOSAR logo are registered trademarks.

Table of Contents

1 Introduction 5
1.1 Objectives . . . 5
1.2 Scope . . . 5
2 Definition of Terms and Acronyms 6
2.1 Acronyms and Abbreviations . . . 6
2.2 Definition of Terms . . . 6
3 Related Documentation 7
3.1 References . . . 7
4 Overview 8
5 Architectural Decisions 9
5.1 Common Decisions . . . 9
5.1.1 Influence of PRS document changes on AP and CP . . . 9
5.1.2 Guidelines on standardizing SW functionalities . . . 10
5.2 Adaptive Platform . . . 10
5.2.1 Dynamic memory allocation . . . 11
5.2.2 Final specifier for types and virtual member functions . . . 12
5.2.3 Usage of out parameters . . . 12
5.2.4 Usage of named constructors for exception-less object creation . . . 13
5.2.5 Introduction of a monotonic clock API . . . 14
5.2.6 Responsibilities of State Management, Execution Management, and Platform Health Management . . . 15
5.2.7 Use of local proxy objects for shared access to objects . . . 18
5.2.8 Functional Clusters shall standardize their production errors . . . 19
5.2.9 Default arguments are not allowed in virtual functions . . . 19
5.2.10 Assert that only APIs from properly initialized functional clusters can be called . . . 20
5.2.11 The AUTOSAR Runtime for Adaptive Applications shall define only interfaces that are intended to be used by AUTOSAR applications and other Functional Clusters . . . 20
5.2.12 AUTOSAR Runtime for Adaptive Applications APIs should follow the C++ Core Guidelines . . . 21
5.2.13 Harmonized error handling for lost daemon connection . . . 22
5.2.14 Granularity of diagnostics . . . 23
5.2.15 Potentially throwing constructors . . . 24
5.2.16 The scope for restarting processes is a FunctionGroup . . . 25
5.2.17 Platform-independent development of Software Clusters of category APPLICATION_LAYER . . . 26
5.2.18 Functional Clusters shall standardize their logging/tracing . . . 27
5.2.19 Guidance whether to define a service or a C++ interface . . . 28
5.2.20 Support only functional dependencies between Software Clusters . . . 29

5.2.21 The introduction of virtual functions requires approval 30
5.2.22 Guidelines for Extension Interfaces 31
5.2.23 Messages for unrecoverable errors 32
5.2.24 No named constructors for abstract classes 33
5.2.25 Modeling of the interaction of application-layer software with Functional Clusters 34
5.2.26 Extent of allowed behavioral specification in API table description field 35
5.2.27 Namespace for AUTOSAR Adaptive Platform Extension Interfaces . 36
5.2.28 Guidelines on documenting re-specified external APIs in AUTOSAR 37
5.2.29 Guidelines on hardware abstraction 38
5.3 Classic Platform 39
5.3.1 The ordering of structure elements is a binding part of the standard 39
5.3.2 Types of standardized header files 40
5.3.3 Guidance for incompatible API changes 41
5.3.4 Handling of Time in the AUTOSAR Classic Platform 42
5.3.5 Providing configurable notification functions in BSW modules 43

1 Introduction

This explanatory document provides additional information on architectural decisions made for the AUTOSAR standards.

1.1 Objectives

The main objective of this document is to provide a documentation of architectural decisions made for the AUTOSAR standards that makes such decisions comprehensible and reviewable in the future and ultimately get more maintainable standards.

1.2 Scope

This document covers decisions made for the software architecture of AUTOSAR standards. The main audience of this document are architects of the AUTOSAR standards as well as members of other working groups.

2 Definition of Terms and Acronyms

2.1 Acronyms and Abbreviations

Abbreviation / Acronym	Description
API	Application Programming Interface
STL	Standard Template Library

2.2 Definition of Terms

Term	Description
Adaptive Application	See [1, AUTOSAR Glossary].
Execution Management	A `Functional Cluster` in the AUTOSAR Adaptive Platform. See [2, EXP_SWArchitecture] for an overview.
Functional Cluster	See [1, AUTOSAR Glossary]. [2, EXP_SWArchitecture] provides an overview of all Functional Clusters in the AUTOSAR Adaptive Platform.
Platform Health Management	A `Functional Cluster` in the AUTOSAR Adaptive Platform. See [2, EXP_SWArchitecture] for an overview.
Process	See [1, AUTOSAR Glossary].
State Management	A `Functional Cluster` in the AUTOSAR Adaptive Platform. See [2, EXP_SWArchitecture] for an overview.
Software Cluster	See [1, AUTOSAR Glossary] and [2, EXP_SWArchitecture].
Thread	See [1, AUTOSAR Glossary].
Watchdog	An external component that supervises execution of the AUTOSAR Adaptive Platform. See [2, EXP_SWArchitecture] for an overview.

3 Related Documentation

This document provides an overview of the architectural decisions that have been made for the AUTOSAR standards and their rationale. A high-level overview of the architecture of the AUTOSAR standards is provided in [3, EXP_LayeredSoftwareArchitecture] (AUTOSAR Classic Platform) as well as [4, EXP_PlatformDesign] and [2, EXP_SWArchitecture] (AUTOSAR Adaptive Platform).

3.1 References

[1] Glossary
AUTOSAR_FO_TR_Glossary

[2] Explanation of Adaptive Platform Software Architecture
AUTOSAR_AP_EXP_SWArchitecture

[3] Layered Software Architecture
AUTOSAR_CP_EXP_LayeredSoftwareArchitecture

[4] Explanation of Adaptive Platform Design
AUTOSAR_AP_EXP_PlatformDesign

[5] Dynamic Memory Allocation and Fragmentation
https://www.researchgate.net/publication/295010953_Dynamic_Memory_-Allocation_and_Fragmentation

[6] Dynamic Memory Allocation on Real-Time Linux
https://static.lwn.net/images/conf/rtlws-2011/proc/Jianping.pdf

[7] TLSF: a new dynamic memory allocator for real-time systems
https://doi.org/10.1109/EMRTS.2004.1311009

[8] The Memory Fragmentation Problem: Solved?
https://doi.org/10.1145/286860.286864

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

[10] Specification of Adaptive Platform Core
AUTOSAR_AP_SWS_Core

[11] General Requirements specific to Adaptive Platform
AUTOSAR_AP_RS_General

4 Overview

Architectural decisions refer to design choices that address architecturally significant requirements, particularly those related to non-functional or quality attributes. These decisions are typically regarded as challenging to make and costly to alter and therefore also alternative possibilities are considered / presented. Specifically, they impact multiple modules or functional clusters within the AUTOSAR standards.

Architectural decisions may also provide general guidelines how to design modules / functional clusters. In such cases the architectural decision provides a rational and the guideline will be added to the respective RS document.

This chapter provides an overview of the organization and structure of architectural decisions listed in this document. All architectural decisions are structured as a table (see table 4.1 for a template). The architectural decisions are organized into sections according to the platform they apply to.

Applies to	A list of AUTOSAR platforms to which this architectural decision applies to.
Decision	The decision itself. The impact or direct consequences (for example, changes to interfaces) of the decision are not documented. Changes to the specifications are made during the roll-out process after the decision has been made.
Rationale	A rationale for the decision.
Category	Category of the decision.
Application affected	States if the decision has an direct impact on existing applications.
Assumptions	Lists the assumptions that have been made before making the decision itself. These assumptions are documented in order to be able to review decisions in the future and check if some assumptions probably no longer hold.
Constraints	Provides an overview of the constraints that were identified to have an impact on possible solutions. The constraints are also documented in order to be reference points for future reviews of the decision.
Alternatives	Lists the alternatives that were considered and a rationale why they are worse than the decision that has been made.
Remarks	Lists remarks on the decisions.
Related requirements	Lists requirements related to the decision.
Release	First AUTOSAR release that contained the documented decision.

Table 4.1: Template for Architectural Decisions

5 Architectural Decisions

5.1 Common Decisions

This chapter lists architectural decisions that have been made for the AUTOSAR Adaptive Platform and Classic Platform.

5.1.1 Influence of PRS document changes on AP and CP

Applies to	AP, CP
Decision	If multiple protocol versions shall be supported by AUTOSAR, they shall be standardized in one PRS document in the same release. Each platform can define the level of support by itself. One approach to document the different levels of support can be the use of chapter 4 of the SWS to describe the limitations. (Alternative 3)
Rationale	We see use cases where different versions of a protocol are used on the different platforms, e.g. AP might support the "old" and the "new" version whereas CP only supports the "old" version. The same applies to "features" of protocols of the same protocol version.
Category	None
Application affected	None
Assumptions	No assumptions were made.
Constraints	AUTOSAR follows a trunk-based development approach without any bugfix branches. This means current PRS document versions simply replace older ones. There is no maintenance of older PRS document versions.
Alternatives	**Allow reference to an older AUTOSAR release** Allow reference to an older AUTOSAR release like in the DLT v2 example.
	Support several versions in the same AUTOSAR release Support several versions of a PRS document in the same AUTOSAR release. Introduce "variant-aware" traceability to express different levels of support by AP and CP.
	Support several versions in a single document If multiple protocol versions shall be supported by AUTOSAR, they shall be standardized in one PRS document in the same release. Each platform can define the level of support by itself. One approach to document the different levels of support can be the use of chapter 4 of the SWS to describe the limitations.
	Support only one version in the same AUTOSAR release Avoid any ambiguity and allow only one PRS version supported by both AP and CP within one AUTOSAR release.
Remarks	No remarks.
Related requirements	None



△

Release	R22-11

5.1.2 Guidelines on standardizing SW functionalities

Applies to	AP, CP
Decision	Criteria which favor a standardization: • Function is reusable by multiple OEMs and multiple projects • Function abstracts standardized communication protocol(s) • Function provides a functionality defined by other standards towards application(s) • Function abstracts direct access to commonly used hardware • Function is required by multiple ECUs/Machines within a car Criteria which discourage standardization: • Function has potential for competitive advantage of OEM/T1 business • Function implements mainly OEM-specific requirement(s) • Function is already established in the market -implementations are available. A potential standardization should also consider the Layered Software Architecture [3] (CP) / Platform Design [4] (AP).
Rationale	No rationale provided.
Category	None
Application affected	Yes
Assumptions	No assumptions were made.
Constraints	No constraints were identified.
Alternatives	No alternatives were considered.
Remarks	There are no generally applicable rules for assigning a SW function into the architecture. Therefore, this architectural decision should be understood as a guide to support such assignment decisions.
Related requirements	No related requirements.
Release	R24-11

5.2 Adaptive Platform

This section lists architectural decisions that have been made for the AUTOSAR Adaptive Platform only.

5.2.1 Dynamic memory allocation

Applies to	AP
Decision	The use of dynamic memory allocation by Adaptive Applications and Functional Clusters is allowed and assumed upon designing the AUTOSAR Adaptive Platform standard.
Rationale	The use of dynamic memory allocation is essentially indispensable as the AUTOSAR Adaptive Platform standard employs C++ as the language for its API. As the AUTOSAR Adaptive Platform standard will be used in safety-related systems, dynamic memory allocation can cause non-deterministic behavior. Two typical issues are the fragmentation and non-deterministic allocation/de-allocation processing time. Memory allocators designed for non-safety-critical systems often exhibit such issues, as they are more or less designed for memory efficiency and/or average processing performance. These issues can be controlled by using deterministic memory allocators. Memory allocation is a well-studied area and various techniques have been reported (Refer to references below for some examples). Multiple AUTOSAR partners within the architecture group reportedly have such deterministic memory allocators implemented and have been used in mass-production systems. Note that such allocators should replace the default `malloc()/free()` implementations provided in the standard C library, that sits underneath the C++ runtime library providing `new()/delete()` and also STL that AUTOSAR Adaptive Platform also uses. This frees applications from providing its own custom deterministic allocators and installing it to custom-allocator-aware classes. Please refer to [5], [6], [7], and [8] for further information on memory fragmentation and memory allocation in real-time systems.
Category	Safety
Application affected	No
Assumptions	Platform vendors and/or compiler vendors can replace the default memory allocation/deallocation functions to use deterministic versions of those functions during critical phases of the runtime when such determinism is required for safety purposes.
Constraints	During certain phases of the runtime determinism is required. These are the phases in which the allocators need to be replaced with deterministic versions.
Alternatives	**Do not use dynamic memory allocation** Not using dynamic memory allocation is not an alternative for using C++. **Limit dynamic memory allocation to certain phases** Disallow dynamic memory allocation during certain phases of the runtime in which determinism is required. This makes it very difficult to run complex code during these phases.
Remarks	No remarks.

▽

△

Related requirements	• [RS_AP_00129] Public types defined by functional clusters shall be designed to allow implementation without dynamic memory allocation
Release	R20-11

5.2.2 Final specifier for types and virtual member functions

Applies to	AP
Decision	`Adaptive Runtime` 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	Making classes and `virtual` functions `final` that are not intended to be used as base classes or to be overwritten expresses the design (in particular the class hierarchy) more explicitly. This will avoid problems such as • to derive from a class that is not prepared for sub-classing, • to inadvertently create a new `virtual` function instead of overwriting a function from the base class due to a slightly different signature.
Category	None
Application affected	No
Assumptions	A clear expression of the intended design of the public AUTOSAR Runtime for Adaptive Applications class hierarchy.
Constraints	No constraints were identified.
Alternatives	**Ensure proper use of AUTOSAR types by code review** The alternative is to have a code review of the application code using AUTOSAR types. This is far out-of-scope of AUTOSAR. Therefore, it is not a real alternative.
Remarks	No remarks.
Related requirements	• [RS_AP_00140] Usage of "final specifier" in ara types
Release	R20-11

5.2.3 Usage of out parameters

Applies to	AP
Decision	Out parameters can be used for in-place modifications but shall not be used for returning values.

▽

△

Rationale	Harmonized look and feel. C++ Core Guidelines [9]: “F.20: For "out" output values, prefer return values to output parameters. [...] A return value is self-documenting, whereas a `&` could be either in-out or out-only and is liable to be misused. This includes large objects like standard containers that use implicit move operations for performance and to avoid explicit memory management.”
Category	None
Application affected	No
Assumptions	Dynamic memory allocation is allowed for all cases in which the APIs are used, even when running time critical safety related code including ASIL D.
Constraints	In/out parameters, i.e. modifying an already existing parameter within a function is allowed. For example, a function that clears or writes to a buffer should receive that buffer as a non-const in/out parameter.
Alternatives	No alternatives were considered.
Remarks	No remarks.
Related requirements	• [RS_AP_00141] Usage of out parameters
Release	R20-11

5.2.4 Usage of named constructors for exception-less object creation

Applies to	AP
Decision	Exceptionless functions for creation of objects which returns an `ara::core::Result` should use the "named constructor idiom".
Rationale	Disadvantages of constructor token approach are avoided as follows: • The constructor token type is an implementation detail of a `Class` and should thus not be specified, or even accessible from outside. This makes the use of `auto` for obtaining a token mandatory because the token type cannot be referred to in any other way. • Moving the token’s content to the `SomeClass` instance has to be done very carefully to fulfill the always-successful guarantee, which can be tricky if multiple resources need to be acquired. • The token object is "destroyed" by `std::move`-ing its value into the `SomeClass` constructor (actually, it is to be in a "valid" but unspecific state according to the C++ standard), but it is easily possible to mistakenly use it again for attempting to create another instance, with undefined results.
Category	Safety
Application affected	Yes
Assumptions	No assumptions were made.
Constraints	No constraints were identified.

▽

△

Alternatives	**Constructor token approach** It was not considered due to the drawbacks described in the rationale of this decision.
	Regular constructor calls Regular constructor calls were not considered because regular constructors may throw exceptions and thus cannot be used in an exception-less design.
Remarks	No remarks.
Related requirements	No related requirements.
Release	R20-11

5.2.5 Introduction of a monotonic clock API

Applies to	AP
Decision	The AUTOSAR Runtime for Adaptive Applications shall provide its own monotonic `std::chrono::SteadyClock` representing the power-up time of the machine. The accuracy of this clock is defined by the platform vendor. The accuracy of this clock could be used as a characteristic value of the platform so that the projects could check whether this clock meets the project-specific requirements (e.g. time synchronization requires typically a clock with higher accuracy). The system start of the machine defines the epoch of the clock. So this clock represents the power-up time of the machine. Functional Clusters dealing with timestamps or clocks should use this clock as a basis.
Rationale	The timestamps used in the time synchronization cluster should be based on `std::chrono`. Time synchronization requires a monotonic clock with special accuracy.
Category	None
Application affected	Yes
Assumptions	The time synchronization cluster is typically a daemon-based architecture due to a single communication endpoint of the time sync messages. A standardized clock with a special accuracy as a common basis is required to synchronize the daemon with the library.
Constraints	No constraints were identified.
Alternatives	**Pass clock type as template argument** The used clock could also be passed as a template argument. But a standardized clock with a special accuracy as a common basis is required anyway in case the time synchronization cluster is daemon based.
Remarks	The monotonic clock API is realized by means of `ara::core::SteadyClock`.

▽

△

Related requirements	• [RS_AP_00130] AUTOSAR Adaptive Platform shall represent a rich and modern programming environment.
Release	R20-11

5.2.6 Responsibilities of State Management, Execution Management, and Platform Health Management

Applies to	AP
Decision	`State Management`, `Execution Management`, and `Platform Health Management` are the fundament/basis of the AUTOSAR Adaptive Platform. A failure in either `State Management`, `Platform Health Management`, or `Execution Management` process will typically lead to stop triggering the watchdog. `Platform Health Management` supervises `State Management` and `Execution Management`. `Platform Health Management` controls the watchdog and is thus in turn supervised by the hardware watchdog. Triggering of a Machine reset as a last resort should not be an option at all in case of a failing of an Adaptive Application supervision (i.e. apart from `Operating System` / `Execution Management` / `State Management` / `Platform Health Management`). A supervision failure in an Adaptive Application shall be reported to `State Management`. `State Management` may forward this failure based on the criticality to `Platform Health Management` to wrongly trigger or stop triggering the serviced watchdog. `Platform Health Management` performs a logical supervision of checkpoints within a process or between processes within a `Function Group`. `Platform Health Management` reports any supervision failures to `State Management`. `State Management` is responsible to perform recovery actions including a switch of the `Function Group State`, by delegating to the Adaptive Application, or, as a last resort, by advising `Platform Health Management` to perform a hardware reset. `Platform Health Management` is intended for supervision of safety-critical processes. Thus, `Platform Health Management` is an optional part of the AUTOSAR Adaptive Platform for non safety-critical applications. Processes shall never be restarted on their own because they may have unknown runtime dependencies. The relation between a `Process` and a `Function Group` is comparable to the relation between a `thread` and a `process`. `State Management` should always trigger a request (`Function Group State` change) to restart processes even in the simplistic/non-dependent cases. Thus, `Platform Health Management` does not have a direct interface to `Execution Management`. The unrecoverable state interface of `Platform Health Management` shall be removed.

▽

△

Rationale	The chosen solution leads to a simpler design of `Platform Health Management` with a single and well-defined responsibility. The chosen solution also adheres to the single responsibility principle for `State Management` (control system state) and `Execution Management` (control processes) as well. Recovery actions can be added by extension (open-closed principle) to `State Management`. There is no need to modify or configure `Platform Health Management`. Supervision failures may be handled by an Adaptive Application as well if `State Management` chooses to delegate recovery to the Adaptive Application.
Category	Safety
Application affected	Yes
Assumptions	• `State Management` is a mandatory part of the AUTOSAR Adaptive Platform. • Performance impact / delay of indirect reporting of supervision failures to an Adaptive Application via `State Management` is negligible in comparison to execution of reasonable recovery actions (such as starting processes).
Constraints	No constraints were identified.
Alternatives	**Failure recovery coordinated by `Platform Health Management`** Recovery in case of a systematic failure is coordinated by `Platform Health Management`. Several components (Adaptive Application, `Execution Management`, `State Management`, watchdog) are involved based on priorities. `Platform Health Management` coordinates the recovery in the following manner: 1. `Platform Health Management` asks the Adaptive Application to recover 2. In case of failure, `Platform Health Management` asks `Execution Management` to restart failed processes 3. In case of failure, `Platform Health Management` asks `State Management` to recover by switching the `Function Group State` 4. In case of failure, `Platform Health Management` stops triggering the watchdog and resets the `Machine` 5. In case of failure, `Platform Health Management` switches to unrecoverable state (not yet fully defined) This alternative was not considered due to not adhering to the single responsibility principle because several components are responsible for recovery actions. This solution would also require `Platform Health Management` to have application knowledge because it has to determine the appropriate `Function Group State` in step 3. Restarting single processes may not be appropriate (step 2) due to runtime dependencies.

▽

△

	Distributed failure recovery Recovery in case of a systematic failure is coordinated by `Platform Health Management` and `State Management`. Several components (Adaptive Application, `Execution Management`, watchdog) are involved based on priorities. `Platform Health Management` and `State Management` coordinate the recovery in the following manner: 1. `Platform Health Management` asks the Adaptive Application to recover 2. In case of failure, `Platform Health Management` asks `State Management` to coordinate recovery by restarting the application 3. `State Management` asks `Execution Management` to change state / switch to degraded state or safe state 4. In case of failure, `State Management` asks Adaptive Application to recover 5. In case step 2 failed due to application dependencies, `Platform Health Management` stops triggering the watchdog and resets the `Machine` This alternative was not considered due to not adhering to the single responsibility principle because `Platform Health Management` and `State Management` share responsibility for coordinating recovery actions.
Remarks	• According to ISO 26262, it has to be ensured that a reaction is triggered after a safety-relevant failure occurred. Therefore, `Platform Health Management` shall make sure that `State Management` receives the notification on a detected failure even if they communicate via an unreliable communication channel, for example, an inter-process communication mechanism. To achieve this, `Platform Health Management` should implement a timeout monitoring. If no response by `State Management` is received after a configurable timeout and number of tries, `Platform Health Management` shall trigger a reaction via hardware `Watchdog`. • For release R19-11 of the AUTOSAR Adaptive Platform, the configuration of `Platform Health Management` included rules for monitoring (`PhmSupervision`), arbitration and recovery actions. With this decision, `Platform Health Management` is only responsible for monitoring. The rules for monitoring (`PhmSupervision`) are unaffected. However, the responsibilities for arbitration and recovery actions are moved to `State Management`. In the current design, `State Management` is a piece of project-specific, coded software with only little configuration. The configuration for `State Management` should be extended to support arbitration and recovery actions as well. This will allow to validate such configurations based on standardized rules which is extremely hard to achieve on source code level.
Related requirements	No related requirements.
Release	R20-11

5.2.7 Use of local proxy objects for shared access to objects

Applies to	AP
Decision	Local proxy object(s) shall be used to provide shared access to object instance(s) via the AUTOSAR Runtime for Adaptive Applications interface.

Rationale	Local proxy objects hide the implementation details of the shared access. The AUTOSAR Runtime for Adaptive Applications interface shall return a proxy object by value. The caller shall use the object as a local proxy for subsequent communication. Return by value is the most straightforward way to return data. This decision enforces harmonization of the AUTOSAR Runtime for Adaptive Applications interface. Stack vendors may freely choose how to implement the shared access inside the proxy class.
	An example for the use of a local proxy object by the caller is the following:
	`Result<void> myFunc() {`
	`Result<void> myFunc() {`
	`  Result<KeyValueStorage> kvsRes`
	`  = KeyValueStorage::Create(KVS_ID);`
	`  if (kvsRes) {`
	`  KeyValueStorage kvs = std::move(kvsRes).Value();`
	`  auto keyRes = kvs.GetAllKeys(); // Value semantics`
	`  // ...`
	`  } else {`
	`  return {std::move(kvsRes).Error()};`
	`  }`
	`}`

Category	None
Application affected	Yes
Assumptions	No assumptions were made.
Constraints	No constraints were identified.
Alternatives	**Use handles for shared access** The alternative of using proxy classes is the usage of handles. These handles would however reveal the implementation details of the shared access.
Remarks	No remarks.
Related requirements	• [RS_AP_00135] Avoidance of shared ownership
Release	R20-11

5.2.8 Functional Clusters shall standardize their production errors

Applies to	AP
Decision	Functional clusters shall standardize production errors for common use-cases demanded by the market. The standardization shall summarize all production errors by a standardized table in all SWS documents specifying production errors.
Rationale	Production errors are a fact. In order to be able to (semi-)automatically analyze them and react to them, they and their documentation/persistence and their healing needs to be standardized.
Category	None
Application affected	Yes
Assumptions	Conceptually production errors are taken over from the AUTOSAR Classic Platform. A differentiation between production errors and extended production errors is not necessary.
Constraints	No constraints were identified.
Alternatives	**Introduce interfaces for monitoring production errors** Functional clusters provide interfaces to allow applications to monitor production errors.
Remarks	None
Related requirements	None
Release	R21-11

5.2.9 Default arguments are not allowed in virtual functions

Applies to	AP
Decision	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 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.
Category	None
Application affected	Yes
Assumptions	No assumptions were made.
Constraints	No constraints were identified.
Alternatives	No alternatives were considered.
Remarks	No remarks.
Related requirements	• [RS_AP_00148] Default arguments are not allowed in virtual functions
Release	R21-11

5.2.10 Assert that only APIs from properly initialized functional clusters can be called

Applies to	AP
Decision	If functionality is called that depends on prior initialization via `ara::core::Initialize` and `ara::core::Initialize` has not been called, the functional cluster implementation shall treat this as a violation and shall follow `SWS_CORE_00003` from [10, Specification of Adaptive Platform Core].
Rationale	Calling APIs from uninitialized functional clusters that depend on prior initialization cannot perform properly. This results in undefined behavior. The problem is typically caused by misconfiguration or incomplete initialization at an earlier stage of the system startup. This cannot be handled by the caller of the API at the point in time where the error is detected. Aborting execution is the only way to signal this kind of systematic error and prevent later failures.
Category	None
Application affected	Yes
Assumptions	Parts of the system need to be initialized statically.
Constraints	No constraints were identified.
Alternatives	**Extend all APIs to report a specific error code** Extend every API that depends on prior initialization with a specific error code (e.g. kNotInitialized) and force callers to check this error code at every call (and let them abort themselves).
Remarks	No remarks.
Related requirements	None
Release	R21-11

5.2.11 The AUTOSAR Runtime for Adaptive Applications shall define only interfaces that are intended to be used by AUTOSAR applications and other Functional Clusters

Applies to	AP
Decision	It is explicitly prohibited to standardize implementation details, like: • Classes, base-classes, functions etc. that are not used on the application level or in platform extension APIs • Implementation inheritance in the public APIs • C++ SFINAE techniques of any kind • Private members of classes

▽

△

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.
Category	None
Application affected	Yes
Assumptions	No assumptions were made.
Constraints	No constraints were identified.
Alternatives	No alternatives were considered.
Remarks	No remarks.
Related requirements	• [RS_AP_00150] Provide only interfaces that are intended to be used by AUTOSAR applications and other Functional Clusters
Release	R21-11

5.2.12 AUTOSAR Runtime for Adaptive Applications APIs should follow the C++ Core Guidelines

Applies to	AP
Decision	AUTOSAR C++ APIs should follow the [9, C++ Core Guidelines]. The exceptions for hard-real-time systems shall apply. The AUTOSAR guidelines defined in RS-General 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, and more maintainable code. Specific guidelines for the automotive domain for C++ 14 are not available. When the upcoming version of the MISRA C++ standard is published, this decision/requirement may be replaced by a decision/requirement to follow MISRA C++.
Category	None
Application affected	Yes
Assumptions	No assumptions were made.
Constraints	Some exceptions apply like the exception-less handling of the ARA APIs.
Alternatives	No alternatives were considered.
Remarks	No remarks.
Related requirements	• [RS_AP_00151] C++ Core Guidelines
Release	R21-11

5.2.13 Harmonized error handling for lost daemon connection

Applies to	AP
Decision	If a functional cluster communicates with a remote peer (e.g. IPC communication to a daemon) adequate error cases for communication failures shall be identified (e.g. lost communication). These error cases shall be grouped (according to the same error recovery mechanism) and if the user of the API shall receive notification (e.g. by callbacks or returning error codes) for a particular group, a suitable notification mechanism shall be selected. Please note that there might be scenarios where the user of an API will not receive any notification by design (e.g. fire-and-forget methods). If an immediate action is required on error occurrence the type of action should be determined in the following way: • Functions that are currently defined with return type void (fire-and-forget methods) require no immediate action. Therefore, no return type and error code needs to be provisioned for such functions. The Adaptive Platform should defer the effects of such functions until the connection to the daemon has been (re-)established. Example: calling `Offer()` on a skeleton in `Diagnostic Management` should defer the internal registration of callbacks until the daemon connection has been (re-)established. • Synchronous functions (e.g. getters and setters) require immediate action. One of the following options shall be implemented for synchronous functions: – provision of error code, e.g. `kServiceNotAvailable` of type `ara::core::ErrorDomain::CodeType`. – mapping to functional status information inside the returned data structure (e.g. class object), which represent an error status • Asynchronous functions (e.g., functions that return a `ara::core::Future`) are a case-by-case decision based on the chance to be able to (re-)connect to the daemon within the usual time-bounds for these functions. If notification of the client is required as immediate action on error occurrence, the notification mechanism shall be based on the mechanisms in `ara::core::Future` or a client callback. A client callback uses registration of a state change callback handler before a client can make use of a service.
Rationale	The application needs to be informed in case of disrupted communication infrastructure in order to handle the error and take countermeasures (if any). The provided guide for choosing the type of action increases the usability of the Adaptive Platform APIs because the errors are signaled in a natural way based on the type of API. In addition, the error handling is partially done in the Adaptive Platform.
Category	None
Application affected	Yes

▽

△

Assumptions	The following assumptions were made: • The implementation does not depend on the type of communication interface, e.g. process local, ara::com or native IPC mechanisms are in scope of the decision. • There is no polling of communication status required by user of the API. • The cause of disconnected service shall be kept agnostic to the user of the API. • Connection oriented communication is out of scope due to inherent detection mechanisms of the protocol.
Constraints	No constraints were identified.
Alternatives	No alternatives were considered.
Remarks	No remarks.
Related requirements	None
Release	R21-11

5.2.14 Granularity of diagnostics

Applies to	AP
Decision	Diagnostic entity shall be identical to the deployable unit within a vehicle. Deployable unit means from hardware units (ECUs), up to Software Clusters.
Rationale	AUTOSAR focused on the Software Cluster approach because it offers a more easy option to keep the two worlds consistent. A Software Cluster is the individual deployable unit from the OEM perspective. Therefore, it is easy to keep the offboard world consistent if the diagnostic has identical boundaries. The production and workshop systems are often bound to the physical device. Thus, many OEMs want to start also with this approach in Adaptive. Consequently, until there is no individual software setup with a car (e.g. because the installed options can be chosen by the driver itself) the offboard systems could be kept consistent by stringent workflows.
Category	None
Application affected	No
Assumptions	DM core doesn't mind if a further diagnostic server is installed (in the context of a new Software Cluster) or the current diagnostic server is just extended.
Constraints	Diagnostics is a (non-verbose) offboard-communication using external description to document the communication content. For the development of a vehicle the AUTOSAR DEXT is used; for the offboard world typically the ASAM ODX format is used, because it offers higher flexibility across different carlines. Today it is often already a challenge to keep the two worlds consistent. But with the dynamic deployment (offered by Adaptive Platform) it is even more challenging because in worst cases each vehicle has an individual setup of installed Software Clusters.

▽

△

Alternatives	None, because both options are requested by the market.
Remarks	No remarks.
Related requirements	None
Release	R21-11

5.2.15 Potentially throwing constructors

Applies to	AP
Decision	Constructors that may throw exceptions shall not participate in overload resolution when C++ exceptions are disabled in the compiler toolchain.
Rationale	Similar solution to other functions that use C++ exceptions as their error handling mechanism e.g., `ara::core::Result::ValueOrThrow()`
Category	None
Application affected	Yes
Assumptions	• There are use cases targeted by AUTOSAR, when C++ exceptions are disabled in the compiler toolchain. • By this decision the overload set might be changed, which may result in an unintended change to the program flow. Thus, the existence of two constructors of the same class that fulfill the following conditions: – one is potentially throwing, the other one `noexcept`, – both accept the same number of parameters – the corresponding parameters have to be convertible from the potentially throwing one to the `noexcept` one would be problematic. It is assumed that this situation will never occur because AUTOSAR follows [11, RS General] and users use C++ best practices, in particular [9, C++ Core Guidelines] C.164: Avoid implicit conversion operators.
Constraints	No constraints were identified.
Alternatives	**Assert that exception-throwing constructors cannot be used** Calling a constructor that may throw exceptions as part of its defined behavior shall result in a compilation error when C++ exceptions are disabled in the compiler toolchain. The compilation error shall result from a `static_assert` with the error message "This constructor requires exception support.". • (Con) This is not implementable. If a constructor is neither part of a class template, nor is the constructor a function template itself, a static assertion failure is triggered even if the constructor is not called anywhere in the code.

▽

△

	Constructors that may throw exceptions shall call abort instead of throwing an exception Constructors that may throw exceptions shall call abort instead of throwing an exception when C++ exceptions are disabled in the compiler toolchain. • (Pro) Constructors that may throw may be used even when C++ exceptions are disabled in the compiler toolchain if it can be precluded that an exception is thrown. • (Con) May be difficult to support by vendors, unless they make large-scale changes to their C++ standard library if it does not happen to follow the AR-specified style. • (Con) Unintended calls to such constructors are only detected at runtime and only in the case of an error.
	Implementation-specific behavior • (Con) Violates [RS_AP_00111]
	Declare all public constructors as `noexcept` All public constructors shall be declared as `noexcept`. Instead of public constructors that may throw, the named constructor idiom shall be used (even when C++ exceptions are enabled in the compiler toolchain). • (Pro) Unintended calls to constructors that may throw are detected at compile time. • (Con) Unnecessary restriction when C++ exceptions are enabled in the compiler toolchain.
Remarks	No remarks.
Related requirements	• [AP_SWS_GENERAL_90007] Potentially throwing constructors
Release	R21-11 (updated in R24-11)

5.2.16 The scope for restarting processes is a FunctionGroup

Applies to	AP
Decision	Applications can be restarted in the scope of a `FunctionGroup`. Ideally, the recovery of supervision errors should be handled in the own `FunctionGroup`. If the recovery cannot be handled within the own `FunctionGroup`, it has to be escalated within the `State Management`. There the coordination for the recovery should take place. This could typically be: • the shutdown/restart of multiple `FunctionGroup`s, • the start of other `FunctionGroup`s or • the restart of the entire `Machine`. The coordination of the restart of the entire `Machine` has to be coordinated within the `State Management` of the platform-core `Software Cluster`.

▽

△

Rationale	Software Clusters are independently deployable units. They could be added later to the same Machine and then should not harm other Software Clusters (freedom from interference between Software Clusters). Recovery shall always be tried within the Software Cluster.
Category	Safety
Application affected	No
Assumptions	The platform-core Software Cluster is the housekeeping initial Software Cluster which Execution Management, Platform Health Management, and State Management are a mandatory part of (if it is a safety relevant Machine).
Constraints	No constraints were identified.
Alternatives	**Restart individual application processes** Applications can be restarted in the scope of a Software Cluster. The Software Cluster is for deployment and not visible in runtime. Thus, it cannot be used in this context.
Remarks	No remarks.
Related requirements	None
Release	R21-11

5.2.17 Platform-independent development of Software Clusters of category APPLICATION_LAYER

Applies to	AP
Decision	Functional Cluster daemons and their startup coordination shall be part of Software Clusters of category PLATFORM_CORE or PLATFORM.
Rationale	This allows uniform and platform-independent integration of Software Clusters of category APPLICATION_LAYER. Consequently, it shall not be necessary to take care of the platform software when developing an Software Cluster of category APPLICATION_LAYER.
Category	None
Application affected	Yes
Assumptions	Market demand is to deliver Machines with pre-installed Adaptive Platform software.
Constraints	No constraints were identified.
Alternatives	**No limitation for allocation of platform software to Software Clusters** Do not make any limitations of platform software. This can lead to a non-uniform integration of the platform software.
Remarks	No remarks.
Related requirements	None

▽

△

Release	R21-11

5.2.18 Functional Clusters shall standardize their logging/tracing

Applies to	AP
Decision	`Functional Cluster`s shall standardize their logging/tracing for common use-cases demanded by the market. The standardization shall be for the non-verbose logging/tracing. If applicable it shall be summarized by two standardized tables (one for logging and a second for tracing) listing all standardized log-/trace messages.
Rationale	Standardized logging/tracing within `Functional Cluster`s allows a harmonized evaluation of logging/tracing on vehicle-level.
Category	None
Application affected	Yes
Assumptions	Logging/tracing is necessary for a variety of use cases (root cause analysis, auditing, debugging). Especially, in a distributed environment a harmonization is necessary to enable automated analysis.
Constraints	No constraints were identified.
Alternatives	**No standardized logging** Do not standardize logging at all.
Remarks	No remarks.
Related requirements	None
Release	R21-11

5.2.19 Guidance whether to define a service or a C++ interface

Applies to	AP
Decision	The decision for a service interface or a C++ library interface should be based on design criteria associated with usability of an interface for the API consumer, efficient usage of Adaptive Platform resources and required capabilities of the communication. In case of conflicting criteria an interface should be implemented by means of a library interface. The decision should consider the various design aspects. Criteria to favor a service based interface design: • Using modelled data types that can be used for code generation. • Support for various features of service oriented communications: A service interface offers elements such as method, event, trigger, field to satisfy certain types of communication patterns. In addition it is possible to aggregate any types of these elements in a single service interface. Such communication features are not offered via library interface. • **Support for flexible discovery of communication endpoints – if a service interface is implemented, consumer of the service does not have to care about location of service instances. Possibly a service might be deployed among different machines.** • Is focused on data transport. Criteria to favor a library based interface design: • Reduced effort in respect to configuration. • Reduced overhead on communication control - a library interface doesn't require maintenance of the communication channel between provider and consumer. Certain types of communication patterns might show better performance like infrequent exchange of data, peer-to-peer communication. • **Additional functionality beyond the pure data transport can be realized.**
Rationale	The quality requirements demand that "the use of the standard shall be as easy as possible for suppliers and application developers". If endpoint configuration, service discovery or remote calls are required, it is sensible to use the existing functionality for services instead of individual solutions. The quality requirements also demand that "the holistic approach shall not be broken (avoid different approaches in one standard)". C++ library interfaces are simpler and may be more efficient. They also leave more freedom for the implementation because they allow an implementation that runs in the process of the Adaptive Application. The quality requirements demand that "the specification shall allow for a run-time efficient implementation. Runtime efficiency refers to all resource consumption, CPU, RAM, ROM". Therefore, C++ library interfaces should be preferred if it is unsure whether a service interface is beneficial.
Category	None
Application affected	No
Assumptions	No assumptions were made.

▽

△

Constraints	No constraints were identified.
Alternatives	**Always use service interfaces** Advantages: • Same kind of interface for all Functional Clusters. Disadvantages: • Not always the most natural way for application developers. Unnecessary complexity and implementation restrictions if functionality of Communication Management is not required.
	Always use C++ library interfaces Advantages: • Same kind of interface for all Functional Clusters. Disadvantages: • Not always the most natural way for application developers. Would require individual solutions for service discovery and selection.
Remarks	An in-process implementation to be run in the process of the calling Adaptive Application is only possible for Functional Clusters with a C++ library interface. Functional Clusters with a service interface require a dedicated process. According to this decision, Network Management should provide a C++ library interface. Nevertheless, Network Management keeps using a service interface to maintain backward compatibility.
Related requirements	None
Release	R22-11

5.2.20 Support only functional dependencies between Software Clusters

Applies to	AP
Decision	Only functional dependencies between Software Clusters shall be supported.
Rationale	A Software Cluster is already a structural deployment entity and is technically the smallest unit that can be individually installed and updated on a Machine (by means of a Software Package). This means that also a delta-update (like updating only a single process within this Software Cluster) requires a new version of the Software Cluster.
Category	None
Application affected	No
Assumptions	No assumptions were made.
Constraints	No constraints were identified.

▽

△

Alternatives	**Support nested Software Clusters** The alternative of structurally nested Software Cluster was realized in AUTOSAR, but the market use-cases could also be realized via Software Cluster with their functional dependencies.
Remarks	Discontinue structurally nested Software Clusters (aka Sub-SWCL).
Related requirements	None
Release	R22-11

5.2.21 The introduction of virtual functions requires approval

Applies to	AP
Decision	Any change to the AUTOSAR Adaptive Platform APIs that introduces new virtual functions shall be presented to the architecture working group for approval.
Rationale	The AUTOSAR Adaptive Platform APIs are designed to be directly implemented by a stack vendor. For example, there are in general no abstract classes or virtual functions defined that a stack vendor has to implement. Thus, there is no need to define virtual functions in general. However, for some use cases such virtual functions may be required (for example callbacks that shall be implemented by an application). Such use cases will be collected and afterwards general design patterns should be derived from them.
Category	None
Application affected	No
Assumptions	The AUTOSAR Adaptive Platform APIs are designed to be directly implemented by a stack vendor (in general no abstract classes, no virtual functions that need to be implemented by a stack vendor).
Constraints	No constraints were identified.
Alternatives	No alternatives were considered.
Remarks	The roll-out shall not affect classes with virtual functions that are already specified in a released document.
Related requirements	None
Release	R22-11

5.2.22 Guidelines for Extension Interfaces

Applies to	AP
Decision	The Adaptive Platform shall support extensions of its behavior by means of standardized extension interfaces, so called Platform Extension Interfaces. An implementation of a Platform Extension Interface is provided e.g., by an OEM, an integrator, or other third-party application. Such extensions would be implemented in a programming language without any code generation support or any runtime configuration in the Manifest. The use of Platform Extension interfaces shall be limited to cases in which it is well justified to provide an implementation of a behavior rather than configuring a generic behavior via the Manifest. Platform Extension Interfaces that make use of the Plugin pattern (see [2] Chapter 8.5.5.3 *"Plugin"*) require review and approval by the architecture working group.
Rationale	The rationale for allowing Platform Extension Interfaces is a better usability of the Adaptive Platform standard. In particular, the level of fulfillment of following quality attributes is raised: • "The AUTOSAR Adaptive Platform Standard elements should be easy to use and hard to misuse." because in those cases in which Platform Extension interfaces are applicable they are more convenient to use. Providing patterns for Platform Extension interfaces (see [2] Chapter 8.5.5 *"Platform Extensions"*) contributes to fulfill the following quality attributes: • "The AUTOSAR Adaptive Platform Standard should document its decisions including their rationale and consequences." • "The AUTOSAR Adaptive Platform Standard should follow a holistic approach and avoid different approaches in one standard." Platform Extension interfaces do not interfere with the quality attribute "An application developer should not be able to supply a custom implementation for pre-defined platform functionality" because an implementation of a Platform Extension interfaces does provide functionality that is not provided by the platform itself.
Category	None
Application affected	No
Assumptions	It is assumed that a full customization of an Adaptive Platform stack implementation by means of the Manifest does not provide the best usability. For some variation points it is assumed to be easier to provide an implementation of a behavior rather than configuring a generic behavior. In such cases the Adaptive Platform needs to be extensible by means of standardized Platform Extension Interfaces that are implemented by an OEM, an integrator, or other third-party application.
Constraints	No constraints were identified.

▽

△

Alternatives	**Use Manifest only** This alternative would forbid any Platform Extension Interfaces. Any kind of variation in the behavior of the AUTOSAR Adaptive Platform needs to be configured via the Manifest. This alternative is not considered because for some variation points it is extremely complicated to configure a generic behavior rather than providing an implementation of the behavior itself.
Remarks	Supported patterns for Platform Extension Interfaces are described in [2] Chapter 8.5.5 *"Platform Extensions"*.
Related requirements	None
Release	R23-11

5.2.23 Messages for unrecoverable errors

Applies to	AP
Decision	Functional clusters should standardize their messages for violations. Other kinds of unrecoverable errors messages should be standardized by ara::core. In case of an unrecoverable error, (if possible) the message should be immediately delivered to the standard error stream of the affected process and to the log sinks as fatal log (like defined by ara::log for the affected process or the `Execution Management`). The implementation of this mechanism should minimize the delay to terminate the affected process. In order to support root cause analysis, the message should contain additional information like • type of error • source code position information • process information • additional context on the error
Rationale	Standardized messages within functional clusters support a common appearance of unrecoverable errors and a straightforward input to track root cause on system (e.g. vehicle) level. For these log messages the standardized logging capability of the diagnostic log and trace (DLT) functional cluster is not usable within the affected process for following reasons: • may cause a significant delay of the process abortion • DLT may not work properly after the error is detected • logging of violations may be required even if the application is not initialized (for using AP libraries) `Execution Management` functional cluster can take the task of logging, if DLT is not capable due to the aforementioned reasons. Standard error stream may be used as a fallback mechanism for analysis in case the logs could not be transmitted to the log sinks of ara::log. But this stream is not suitable to exchange log information between different processes. As a consequence, such mechanism is implementation specific.
Category	None

▽

△

Application affected	No
Assumptions	The following assumptions were made: • Log messages containing information about unrecoverable errors that occurred are useful for debugging. • Using ara::log for creating these logs could significantly delay the termination of the process, which may result in poor user experience.
Constraints	There are some potential constraints as follows: • Compliance with data protection may inhibit projects to reveal development related information (e.g. filenames) • Relevant log information is not at all or only partially available • Creation of log message not possible (e.g. when the process is terminated through std::terminate() call in code that cannot be modified by the implementer)
Alternatives	No alternatives were considered.
Remarks	No remarks.
Related requirements	None
Release	R24-11

5.2.24 No named constructors for abstract classes

Applies to	AP
Decision	Abstract classes that are intended for specialization by the user of the ARA shall not have recoverable errors in their constructors.
Rationale	Specifying a recoverable error would make a named constructor necessary. However, this pattern can not be applied to abstract classes where the concrete class is not known by the AUTOSAR Adaptive Platform stack. In the described situation the concrete class is user-defined. The stack has no knowledge of it. It therefore can not create an object of the type of the abstract class. Thus a named constructor is not implementable. If there are no recoverable errors during the construction the constructor can be declared `noexcept` and there is no need for the named constructor.
Category	None
Application affected	No
Assumptions	The following assumptions were made: • There are currently no situations in the ARA where an abstract class construction requires recoverable errors.
Constraints	No constraints were identified.

▽

△

Alternatives	**Alternative 1** One alternative is to define a solution approach for implementing recoverable errors for constructors of abstract classes without relying on exceptions. Since there was no need identified for this, that alternative was not chosen.
Remarks	In case there arises a situation in which an abstract class construction requires recoverable errors, this arc decision and possible alternatives shall be discussed. As a result, an alternative solution might be found, the decision might be altered, or an exception might be granted.
Related requirements	None
Release	R24-11

5.2.25 Modeling of the interaction of application-layer software with Functional Clusters

Applies to	AP
Decision	`PortPrototype`s defined for the interaction of application-layer software with the Functional Cluster shall always be modeled as `RPortPrototype`s, irrespective of the `PortInterface` that types the `RPortPrototype`.
Rationale	All `PortPrototype`s that are created in a Software Component for the interaction with Functional Clusters are of one kind, irrespective of the interaction semantics and of the used `PortInterface`. This approach harmonizes and simplifies the modeling approach for the interaction of application-layer software with all Functional Clusters.
Category	None
Application affected	No
Assumptions	The following assumptions were made: • The current decision reflects the current status of the majority of the documents.
Constraints	No constraints were identified.
Alternatives	**The semantic of the interaction determines the type of the Port** In case the application is calling a method on the Functional Cluster then a `RPortPrototype` is used (e.g., reporting of a Diagnostic Event). In case that the application is providing a method that is used by the functional cluster then a `PPortPrototype` is used (e.g., DebouncingCounterCallback, DiagnosticRoutine). Disadvantage: Such an approach would require to design the port interfaces in a way that only one direction is applicable.

▽

△

Remarks	For backward compatibility reasons modeling approaches for the interaction of application-layer software with the Functional Cluster that were released before R23-11 may deviate from this decision. Please note that in contrast to the conventions on the AUTOSAR Classic Platform, the RPorts are only used on the application side of this communication relation. The PPorts on the Foundation Functional Cluster side are not modeled and therefore also the connection between the application and the Functional cluster is not modeled as well.
Related requirements	None
Release	R24-11

5.2.26 Extent of allowed behavioral specification in API table description field

Applies to	AP
Decision	"Trivial" behavioral definitions can be made in the description fields of the API tables in [10] Chapter 8 *"API specification"*, [10] Chapter 9 *"Service Interfaces"*, and [10] Appendix C *"Platform Extension Interfaces (normative)"*. Then this behavior does not need to be specified in [10] Chapter 7 *"Functional specification"*. Additional information: • What exactly can be considered "trivial" has to be decided case-by-case. • If this is done in the row "description", no additional behavioral specification shall be made in [10] Chapter 7 *"Functional specification"*. • This includes behavioral definitions in the following rows of the table: "description", "errors", "exceptions", and "violations".
Rationale	Having trivial behavior as part of the API tables saves an additional spec item in [10] Chapter 7 *"Functional specification"* that is difficult for a reader to find and consider (usability) and can become inconsistent to the content of the API table (maintainability).
Category	None
Application affected	No
Assumptions	The following assumptions were made: • The definition of behavior describes a function. Therefore it can be part of the "Description" fields in the API table. • Doc owners and reviewers make sensible judgement of what behavior can be considered "trivial".
Constraints	No constraints were identified.
Alternatives	**Alternative 1** Mandate all behavior specifications to be done in [10] Chapter 7 *"Functional specification"*.

▽

△

Remarks	All specification items (including their "Description" row) in [10] Chapter 8 *"API specification"*, [10] Chapter 9 *"Service Interfaces"*, and [10] Appendix C *"Platform Extension Interfaces (normative)"* are binding but might be incomplete. The complete picture only forms when also considering the specification items from [10] Chapter 7 *"Functional specification"*. If there is such a behavioral definition in the row "Description", it shall use one of the keywords defined by Requirements Management (e.g., "shall"). If there are spec items for the behavior in [10] Chapter 7 *"Functional specification"* for an item in [10] Chapter 8 *"API specification"*, [10] Chapter 9 *"Service Interfaces"*, or [10] Appendix C *"Platform Extension Interfaces (normative)"*, there shall not be a behavioral specification in the row "Description". However, it is still allowed and desired to make matter of fact statements to better describe the behavior (e.g., use "the function does...", instead of "the function shall..."). Examples: • API tables where the behavior is clear from the C++ semantics like constructors and destructors usually do not need separate specification items in [10] Chapter 7 *"Functional specification"* defining their behavior. • Errors that are simple to understand and sufficiently explained in the API table do not need a separate specification item in [10] Chapter 7 *"Functional specification"*, for example in `ara::core::InstanceSpecifier::Create` the error description for `kInvalidMetaModelPath`: if the metaModelIdentifier is not a valid path to a model element.
Related requirements	None
Release	R24-11

5.2.27 Namespace for AUTOSAR Adaptive Platform Extension Interfaces

Applies to	AP
Decision	All AUTOSAR Adaptive Platform Extension Interfaces shall belong under a top-level namespace `apext`, which stands for AUTOSAR Adaptive Platform Extension Interface. The namespace shall have a sub-namespace representing the Functional Cluster that specifies the Platform Extension Interface, for example, `apext::log`.
Rationale	The `ara` which stands for AUTOSAR Runtime for Adaptive Applications, is an interface for applications. The purpose of the AUTOSAR Adaptive Platform Extension Interfaces is to extend the platform's capabilities. Therefore, it should be in a separate top-level namespace to clearly distinguish the intention of the interface.
Category	None
Application affected	Yes

▽

△

Assumptions	The following assumptions were made: • Based on the definition of the Platform Extensions described in the [2, EXP_SWArchitecture] and the existing instances of such extensions in multiple Functional Clusters.
Constraints	No constraints were identified.
Alternatives	**Alternative 1** Not to define this namespace and mix it in the `ara` namespace.
Remarks	No remarks.
Related requirements	None
Release	R24-11

5.2.28 Guidelines on documenting re-specified external APIs in AUTOSAR

Applies to	AP
Decision	When documenting external APIs which are re-specified in AUTOSAR (e.g. AUTOSAR implementations of ISO C++), those AUTOSAR definitions should not duplicate the specification of: • function behavioral descriptions • function parameters • C++ function template parameters • return types IF they do not deviate from the external specification AUTOSAR provides doxygen processing tags to avoid this duplication. In place of the function behavioral description the following template general statement shall be used: "As per <fully-qualified-function-name> in <citation>". **Example:** "As per std::pmr::memory_resource::deallocate in [11]". Further examples can be seen in [SWS_CORE_06506] or [SWS_CORE_06507] If an AUTOSAR re-specified function is different in signature or behavior to its externally specified function, it is sufficient to document only the deltas. In place of the function behavioral description the following general statement shall be used: "As per <fully-qualified-function-name> in <citation> except for the following deviations: 1. signature deviation 1 2. signature deviation 2 **Example:** "As per std::pmr::memory_resource::deallocate in [11] except for the following deviations: 1. Function is noexcept 2. If unsuccessful: the returned pointer shall be nullptr

▽

△

	△ 3. ... Further examples can be seen in [SWS_CORE_06503] or [SWS_CORE_06504]
Rationale	1. Avoidance of potential copyright issues 2. Uniformity and consistence of AUTOSAR external references 3. Automatic maintenance and management of the AUTOSAR bibliography
Category	None
Application affected	No
Assumptions	No assumptions were made.
Constraints	No constraints were identified.
Alternatives	No alternatives were considered.
Remarks	No remarks.
Related requirements	None
Release	R25-11

5.2.29 Guidelines on hardware abstraction

Applies to	AP
Decision	Functional Clusters shall build on standardized interfaces for accessing hardware functionality rather than accessing hardware directly. There are two alternatives to achieve this: • **If the hardware functionality is project-independent:** AUTOSAR shall only define the interface towards the application. The stack vendor is responsible to select a proper interface of the Operating System or other libraries, e.g., POSIX, to access the hardware. AUTOSAR may suggest libraries or interfaces as part of section `Demands and Constraint on Base Software` and name the accessed hardware units. This option shall be used whenever possible. • **If the hardware functionality is project-dependent:** AUTOSAR shall define Platform Extension Interfaces in the Functional Cluster. An integrator is then responsible to provide a project-specific implementation. The interfaces for accessing hardware shall follow the guidelines for Platform Extension Interfaces defined in [2] Chapter 8.5.5 *"Platform Extensions"*, in particular the Callout pattern. Please note that the two alternatives are not exclusive. Each hardware functionality needs to be assessed individually. A Functional Cluster may provide a mixture of both.

▽

△

Rationale	This guideline provides most flexibility to the stack vendors in implementing hardware abstraction while still providing a standardized API to the application. The approach also minimizes the need for shared responsibilities between stack vendor and integrator that would be a consequence of Callouts.
Category	None
Application affected	No
Assumptions	No assumptions were made.
Constraints	No constraints were identified.
Alternatives	No alternatives were considered.
Remarks	No remarks.
Related requirements	None
Release	R26-11

5.3 Classic Platform

This section lists architectural decisions that have been made for the AUTOSAR Classic Platform only.

5.3.1 The ordering of structure elements is a binding part of the standard

Applies to	CP
Decision	The order of structure elements as defined by the SWS is considered as part of the standard. Implementation specific optimizations, e.g. a re-ordering of structure elements by size to avoid alignment gaps, are therefore not standard compliant.
Rationale	Object code interoperability could be jeopardized by deviating structure type definitions.
Category	None
Application affected	Yes
Assumptions	Structure elements are usually accessed via name, which means that the order shouldn't matter. There are however valid use-cases like the initialization of structures without designated initializers (e.g. `my_struct x = {0, 42}`) where no element names are involved at all.
Constraints	None
Alternatives	**No standardized order of structure elements** The order of structure elements in the SWS is not prescribed by the standard. An implementation is free to do any desired re-ordering.

▽

△

Remarks	In resource optimized implementations, structure elements are usually ordered by size to avoid alignment gaps. This helps to increase efficiency and reduces memory consumption. Nevertheless some structures defined in AUTOSAR do not follow this rule.
Related requirements	None
Release	R21-11

5.3.2 Types of standardized header files

Applies to	CP
Decision	There shall be only 3 types of headers: 1. The module header (e.g. `NvM.h`, `CanIf.h`, `EcuM.h`, ...) 2. The private header between two modules (e.g. `BswM_Sd.h`, `Adc_SchM.h`, `Dcm_Externals.h`, ...) 3. The shared header (e.g. `PlatformTypes.h`, `StandardTypes.h`, `Can_GeneralTypes.h`, `ComStackTypes.h`, ...) Any additional headers are no longer necessary and are dropped/removed from the SWS. This means that they are no longer standardized. An implementation is however free to have such headers for its own purpose. **Rules:** • All header files are self-contained • A module which uses types of another BSW in its own interface must consider moving such types into a shared header (Exception: types of service interfaces which are generated by the RTE and are available via `Rte_<Mip>.h`) • A library cannot have private headers by definition • Shared headers only consist of types and enums (No function prototypes...) • Shared headers do never depend on other module or private headers • For callouts to integration code or CDDs: The prototypes are available via `<Mip>_Externals.h` **Consequences:** • The tables for types and APIs (C interface) shall have a line "Available via" to indicate the name of the header which exports the type/function
Rationale	This is sufficient for an external view to answer the question which header is needed by a user.
Category	None
Application affected	No
Assumptions	None

▽

△

Constraints	None
Alternatives	**BSW implementation focused header file concept** Keep the current BSW implementation focused header file concept.
Remarks	None
Related requirements	None
Release	R21-11

5.3.3 Guidance for incompatible API changes

Applies to	CP

Decision	If a function from a BSW module requires an incompatible change, the change of the API name shall be based on this decision matrix:
	`[change]`
	`--> [API shall be renamed (==new API, old to obsolete)]`
	`---`
	`[Adding/removing of a parameter with change of behavior]`
	`--> [YES]`
	`[Adding/removing of a parameter without change of behavior]`
	`--> [NO: Direct change, "Bug", "Optimization"]`
	`[Changing an existing type / return type with change of behavior]`
	`--> [YES]`
	`[Changing an existing type / return type without change of behavior]`
	`--> [NO]`
	`[Major change of the behavior of a function without a change of the prototype]`
	`--> [YES]`
	
	If a new API replaces the old one, the old (obsoleted) API shall contain information which new API shall be used instead.
	A) For external APIs, that are not also used by other BSW modules, the following life cycle changes shall apply:
	1. Introduction of the new function AND setting the existing old one to "obsolete"
	2. In the release + 1: remove the old function
	B) For other APIs, which are mainly or exclusively used between the BSW modules, the change shall become immediately visible (direct change of the existing function, no "obsolete" setting)

Rationale	The approach provides the best backward compatibility rating and offers a migration time for users.
Category	None
Application affected	No
Assumptions	The changed function is a normal service function. Callouts (functions where the prototype is defined by the module, but not the code) may be handled differently.
Constraints	None

▽

△

Alternatives	**Directly change existing function** Instead of adding a new function the existing one can also be directly changed. • (Pro) If only i.e. arguments were added/removed, then the name of the function does not change • (Con) Does not support a migration phase for users
	Prepare function for future changes If it is already known that the function may change in the future then the arguments could be provided as tag/value pairs. • (Pro) Allows compatible extensions of arguments for future use cases • (Con) Requires variable length arguments ("...") which cause MISRA issues (?)
Remarks	The drawback of the decision is that the new function requires a new function name. For real bugs where the existing prototype can not support the already defined behavior ("does not work at all") a direct change without migration phase is preferable. If a C type is changed (e.g. a structure gets a new field) and such type is used in a prototype, the change of the type is considered compatible. So no mandatory change of the function prototype (e.g. function name) is needed.
Related requirements	None
Release	R22-11

5.3.4 Handling of Time in the AUTOSAR Classic Platform

Applies to	CP
Decision	The `Tm` module shall handle all use cases related to local time handling. This includes all cases where currently `Os` is used (e.g. service interface for time handling).
Rationale	This reduces overlap and ambiguity of existing time services in the AUTOSAR Classic Platform.
Category	None
Application affected	Yes
Assumptions	No assumptions were made.
Constraints	Global time synchronization still requires separate modules (e.g. `StbM`). Furthermore there are specific timing uses cases in `EcuM` which are not impacted by this decision.
Alternatives	**Integrate functionality in Os module** Remove the `Tm` module and integrate the functionality in the `Os` module.

▽

△

	Remove Tm module Remove the `Tm` module with no replacement of functionality.
	Integrate functionality in StbM module Remove the `Tm` module and integrate the functionality in the `StbM` module.
Remarks	No remarks.
Related requirements	None
Release	R23-11

5.3.5 Providing configurable notification functions in BSW modules

Applies to	CP
Decision	Use only one model element for configurable callback. Based on this element multiple API tables for caller and callee side can then be provided. Currently the APIs are modeled for each SWS separately. We could adapt the BSW UML model in a way that only the "calling SWS" does contain the model element for the API and the "providing" SWS contains just a reference. The artifact generator can then provide for each SWS an own table, which would be just a copy of the model element, just with an own spec item id.
Rationale	Use only one element to avoid inconsistent changes.
Category	None
Application affected	No
Assumptions	No assumptions were made.
Constraints	No constraints were identified.
Alternatives	**Alternative 1** The "providing" SWS shall only have a spec item in Chapter 8 (own subchapter) which references to the API in the "calling" SWS like "The `<own_module>` shall provide a notification function with name `<XYZ>` which comply to the API of `<ARTraceRef to API>` in `<calling_module>`." and the reference is checked by tooling.
	Alternative 2 Same as Alternative 1, but no formal spec item. Reference may be placed in Chapter 5.
Remarks	No remarks.
Related requirements	None
Release	R24-11