Document Title	General Specification of Adaptive Platform
Document Category	Software specification (SWS)
Document UID	1142
Document Status	published
Part of AUTOSAR Standard	Adaptive Platform
Part of Standard Release	R26-11

Document Change History			
Date	**Release**	**Changed by**	**Description**
2026-11-26	R26-11	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 and functional overview 5
2 Definition of acronyms and abbreviations 6
2.1 Acronyms and abbreviations . . . 6
2.2 Definition of terms . . . 6
3 Related documentation 7
3.1 Input documents & related standards and norms . . . 7
4 Constraints and assumptions 8
5 Dependencies to other Functional Clusters 9
6 Requirements Tracing 10
7 Functional specification for all Functional Clusters 11
7.1 Functional Cluster names . . . 11
7.2 Functional Cluster header files . . . 12
7.3 ARA coding guidelines . . . 13
7.4 Undefined behavior . . . 14
7.5 Implementation-defined behavior . . . 14
7.6 Vendor-specific behavior . . . 15
7.7 Initialization/De-initialization . . . 15
7.7.1 Common aspects . . . 15
7.7.2 Initialization . . . 15
7.7.3 De-initialization . . . 16
7.8 Logging and Tracing . . . 16
7.9 Thread safety . . . 18
7.10 Async signal safety . . . 21
7.11 Error handling . . . 22
7.11.1 Traditional error handling in C and C++ . . . 22
7.11.2 Violation . . . 23
7.11.3 Corruption . . . 25
7.11.4 Failed default allocation . . . 25
7.11.5 Error . . . 26
7.11.6 Standardized Error domain Identifiers . . . 26
7.12 Reporting . . . 27
7.12.1 Security Events . . . 27
7.12.2 Log Messages . . . 28
7.12.2.1 FailedDefaultAllocation . . . 28
7.12.3 Violation Messages . . . 28
7.12.3.1 AraNotInitializedViolation . . . 29
7.12.3.2 InstanceSpecifierAlreadyInUseViolation . . . 29
7.12.3.3 UnresolvableInstanceSpecifierViolation . . . 30

7.12.3.4 InsufficientPermissionsViolation 31
7.12.4 Production Errors 31
8 API specification 32
9 Service Interfaces 33
10 Configuration 34
A Mentioned Manifest Elements 35
B Demands and constraints on Base Software (normative) 36
C Platform Extension Interfaces (normative) 37
D Not implemented requirements 38
E Change history of AUTOSAR traceable items 39
E.1 Traceable item history of this document according to AUTOSAR Release R26-11 39
E.1.1 Added Specification Items in R26-11 39
E.1.2 Changed Specification Items in R26-11 41
E.1.3 Deleted Specification Items in R26-11 41

1 Introduction and functional overview

The goal of this document is to define a common set of basic requirements that apply to all functional clusters of the Adaptive Platform. It adds a common part to the specifications and it needs to be respected by platform vendors.

2 Definition of acronyms and abbreviations

2.1 Acronyms and abbreviations

This document does not define any specific acronyms or abbreviations. A general list of acronyms and abbreviations is available in the AUTOSAR glossary [1].

2.2 Definition of terms

The table below contains all definitions of terms that are used in this document.

Abbreviation / Acronym:	Description:
`Violation`	The semantics of a `Violation` are defined in [AP_SWS_GENERAL_00021]
`Standardized Violation`	A `Violation` that is standardized in the AUTOSAR AP standard
`Non-Standardized Violation`	A `Violation` that is not standardized in the AUTOSAR AP standard
`Undefined behavior`	Behavior as defined in `[defns.undefined]` in [2], with any additional UB documented by AUTOSAR (see also [AP_SWS_GENERAL_90031])
`Implementation-defined behavior`	Behavior as defined in `[defns.impl.defined]` in [2]; details are deferred to and documented by the implementation (see also [AP_SWS_GENERAL_90032])
`Vendor-specific behavior`	Equivalent to `Implementation-defined behavior` in AUTOSAR APIs (see also [AP_SWS_GENERAL_90033])
`callable`	The related definitions in `[func.def]` of [2] apply to AUTOSAR APIs (see also [AP_SWS_GENERAL_13206])
`Functions implemented by the user of the API`	Functions with signatures defined by AUTOSAR and implemented by the user, such as callbacks or pure virtual functions (see also [AP_SWS_GENERAL_13212])
`Corruption`	Corruption of a system resource (e.g., stack or heap overflow or hardware memory flaw); it is non-recoverable (see also [AP_SWS_GENERAL_00022])
`Failed Default Allocation`	Inability of the default framework allocator to satisfy an allocation request; it is non-recoverable (see also [AP_SWS_GENERAL_00023])
`Functional Cluster Identifier (FCID)`	Identifier for the Functional Cluster portion of a log and trace message identifier (see also [AP_SWS_GENERAL_00055])
`Sub Message Identifier (SMID)`	Identifier for the sub-message portion of a log and trace message identifier (see also [AP_SWS_GENERAL_00055])

Table 2.1: Acronyms and abbreviations used in the scope of this Document

3 Related documentation

3.1 Input documents & related standards and norms

[1] Glossary
AUTOSAR_FO_TR_Glossary

[2] ISO/IEC 14882:2014, Information technology – Programming languages – C++
https://www.iso.org

[3] General Requirements specific to Adaptive Platform
AUTOSAR_AP_RS_General

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

[5] Explanation of Adaptive Platform Design
AUTOSAR_AP_EXP_PlatformDesign

[6] Specification of Log and Trace
AUTOSAR_AP_SWS_LogAndTrace

[7] Explanation of Adaptive Platform Software Architecture
AUTOSAR_AP_EXP_SWArchitecture

[8] Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7
http://pubs.opengroup.org/onlinepubs/9699919799/

4 Constraints and assumptions

This specification does not specify a functional cluster. This chapter does not apply.

5 Dependencies to other Functional Clusters

This specification does not specify a functional cluster. This chapter does not apply.

6 Requirements Tracing

The following tables reference the requirements specified in AUTOSAR RS General [3], and links to the fulfillment of these.

Requirement	Description	Satisfied by
[RS_AP_00111]	Source Code Portability Support	[AP_SWS_GENERAL_00055] [AP_SWS_GENERAL_90001] [AP_SWS_GENERAL_90002] [AP_SWS_GENERAL_90003] [AP_SWS_GENERAL_90004] [AP_SWS_GENERAL_90005] [AP_SWS_GENERAL_90007]
[RS_AP_00114]	Compatibility with the ISO 14882 C++ standard	[AP_SWS_GENERAL_90031] [AP_SWS_GENERAL_90032] [AP_SWS_GENERAL_90033]
[RS_AP_00115]	Public namespaces	[AP_SWS_GENERAL_90025]
[RS_AP_00116]	Header file name	[AP_SWS_GENERAL_90001]
[RS_AP_00128]	Error reporting	[AP_SWS_GENERAL_00002]
[RS_AP_00130]	AUTOSAR Adaptive Platform shall represent a rich and modern programming environment	[AP_SWS_GENERAL_13206] [AP_SWS_GENERAL_90023] [AP_SWS_GENERAL_90024]
[RS_AP_00139]	Return type of synchronous function calls	[AP_SWS_GENERAL_00002]
[RS_AP_00142]	Handling of unsuccessful operations	[AP_SWS_GENERAL_00002] [AP_SWS_GENERAL_00003] [AP_SWS_GENERAL_00004] [AP_SWS_GENERAL_00005] [AP_SWS_GENERAL_00006] [AP_SWS_GENERAL_00007] [AP_SWS_GENERAL_00020] [AP_SWS_GENERAL_00021] [AP_SWS_GENERAL_00022] [AP_SWS_GENERAL_00023] [AP_SWS_GENERAL_00090] [AP_SWS_GENERAL_13000] [AP_SWS_GENERAL_13006] [AP_SWS_GENERAL_13007] [AP_SWS_GENERAL_13018] [AP_SWS_GENERAL_13215]
[RS_AP_00156]	Naming conventions for L&T Context ID	[AP_SWS_GENERAL_00056]
[RS_AP_00163]	Reentrancy of Functions	[AP_SWS_GENERAL_13210]
[RS_AP_00164]	Thread-safety of Functions	[AP_SWS_GENERAL_13200] [AP_SWS_GENERAL_13201] [AP_SWS_GENERAL_13202] [AP_SWS_GENERAL_13203] [AP_SWS_GENERAL_13204] [AP_SWS_GENERAL_13207] [AP_SWS_GENERAL_13211] [AP_SWS_GENERAL_13212] [AP_SWS_GENERAL_13213] [AP_SWS_GENERAL_13214]
[RS_AP_00165]	Concurrent Use of Different Objects	[AP_SWS_GENERAL_13205]
[RS_AP_00181]	Guideline for the use of [[nodiscard]]	[AP_SWS_GENERAL_90008]

Table 6.1: Requirements Tracing

7 Functional specification for all Functional Clusters

This chapter defines a common set of basic requirements that apply to all Functional Clusters of the Adaptive Platform. It adds a common part to the specifications and it needs to be respected by platform vendors.

7.1 Functional Cluster names

AUTOSAR Functional Clusters are assigned a standardized "name" e.g. `com` together with a reserved `Functional Cluster namespace` ([RS_AP_00115]) below the top-level namespace. Table [AP_SWS_GENERAL_90025] shows the reserved Functional Cluster settings for each.

[AP_SWS_GENERAL_90025] AUTOSAR-standardized Functional Cluster names

Status: DRAFT
Upstream requirements: RS_AP_00115

⌈

Name	*Description*	*C++ namespace*
aag	Automotive API Gateway	ara::aag
adi	Sensor Interfaces	ara::adi
apbsw	Adaptive Platform Base Software	n/a
com	Communication Management	ara::com
core	Core	ara::core
crypto	Cryptography	ara::crypto
diag	Diagnostics	ara::diag
exec	Execution Management	ara::exec
fw	Firewall	ara::fw
idsm	Intrusion Detection System Manager	ara::idsm
log	Log and Trace	ara::log
nm	Network Management	ara::nm
osi	Operating System Interface	ara::osi
per	Persistency	ara::per
phm	Platform Health Management	ara::phm
rds	Raw Data Stream	ara::rds
rper	Remote Persistency	ara::rper
shwa	Safe Hardware Acceleration	ara::shwa
sm	State Management	ara::sm
tsync	Time Synchronization	ara::tsync
ucm	Update and Configuration Management	ara::ucm
vucm	Vehicle Update and Configuration Management	ara::vucm

⌋

7.2 Functional Cluster header files

[AP_SWS_GENERAL_90001] Include folder structure

Upstream requirements: RS_AP_00116, RS_AP_00111
Previous identifiers: SWS_CORE_90001

⌈All `#include` directives in header files that refer to ARA libraries shall be written in the form: `#include "ara/fc/header.h"` where:

- `ara`: reserved; as per [RS_AP_00115]
- `fc`: reserved; is the remaining directory path of the implementation's *installed* header file, starting with the Functional Cluster name as per [AP_SWS_GENERAL_90025] e.g. `com`
- `header.h`: the filename of the header file as per [RS_AP_00116]

⌋

Example: for class `ara::exec::ExecutionClient` present in `#include "ara/exec/execution_client.h"`

The "..." form of `#include` statements shall be used, due to the recommendation given in [2, C++14] `[cpp.include] Par. 7.`

[AP_SWS_GENERAL_90002] Prevent multiple inclusion of header file

Upstream requirements: RS_AP_00111
Previous identifiers: SWS_CORE_90002

⌈All public header files shall prevent multiple inclusion by using `#include` guards that are likely to be system-wide unique.⌋

While uniqueness can generally not be guaranteed, the likelihood of collisions can be decreased with a naming scheme that is regular and results in long symbol names. The following `#include` guard naming scheme should be used by implementations for all header files that cover symbols within the `ara::` namespace or a sub-namespace therein: `ARA_<PATH>_H_`

Where <PATH> is the relative path name of the header file within the location of the implementation's *installed* header files, starting with the Functional Cluster name (and omitting the file extension), and with all components of <PATH> separated by underscore ("_") characters and containing only upper-case characters of the ASCII character set.

Example: The header file included with `#include "ara/log/logger.h"` should use the `#include` guard symbol `ARA_LOG_LOGGER_H_`.

7.3 ARA coding guidelines

[AP_SWS_GENERAL_90003] Reservation of ARA preprocessor macros

Upstream requirements: RS_AP_00111
Previous identifiers: SWS_CORE_90003

⌈C/C++ preprocessor symbols that start with `ARA` are reserved for use by AUTOSAR.⌋

The Adaptive Platform tries to avoid the use of C/C++ preprocessor macros. Such macros start with the prefix `ARA`. Platform vendors should thus not define any symbols (both macros and C/C++ ones) with this prefix, lest they conflict with such future additions to the standard.

[AP_SWS_GENERAL_90004] Consistency of Implementation-defined declaration classifiers

Upstream requirements: RS_AP_00111
Previous identifiers: SWS_CORE_90004

⌈All APIs shall be implemented with the exact same declaration classifiers that are specified, except for `inline` and `friend`, which may be added as necessary.⌋

Note: The order of declarations may be freely chosen.

[2, C++14] `[dcl.spec]` defines the specifiers that can be used in a declaration; these include, for instance, `static`, `virtual`, `constexpr`, `inline` and `friend`. An implementation that uses a different set of specifiers in its declaration of a specified API may be incompatible to the standard, or may allow non-standardized usage of that API, leading to portability concerns.

[AP_SWS_GENERAL_90008] Addition of `[[nodiscard]]`

Upstream requirements: RS_AP_00181

⌈Implementations may add `[[nodiscard]]` to standardized elements if deemed beneficial.⌋

Note: The standard is relatively restrictive with its use of `[[nodiscard]]`. Implementations may want to use it in more places, e.g., for `ara::core::Result` or `ara::core::Future`. See also: [RS_AP_00181].

[AP_SWS_GENERAL_90005] Custom declarations and definitions

Upstream requirements: RS_AP_00111
Previous identifiers: SWS_CORE_90005

⌈Implementation shall not add public declarations or definitions that are not specified in an SWS to the namespace `ARA` or any of its direct sub-namespaces, except in the sub-namespace `::internal` which is reserved for vendor-specific use.⌋

The Adaptive Platform is designed for source code portability. Wherefore any conformant implementation of the Adaptive Platform allows a 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 will be necessary for this if the application only uses constructs from the designated minimum C++ language version. The implementation may provide proprietary, non-ARA interfaces, as long as they are not contradicting the AP standard.

[AP_SWS_GENERAL_90007] Potentially throwing constructors

Upstream requirements: RS_AP_00111
Previous identifiers: SWS_CORE_90007

⌈Constructors that may throw exceptions shall not participate in overload resolution when C++ exceptions are disabled in the compiler toolchain.⌋

See ArcDecision in FO_EXP_SWArchitecturalDecisions [4] *Potentially throwing constructors*.

7.4 Undefined behavior

[AP_SWS_GENERAL_90031] Undefined behavior in AUTOSAR APIs

Upstream requirements: RS_AP_00114
Previous identifiers: SWS_CORE_90031

⌈

- Generally: as per `[defns.undefined]` in [2]
- Additionally: Where known, UB is documented by AUTOSAR. The effects of UB are not documented and no guarantees are given about how the API shall execute during UB. Nevertheless, documented UB in AUTOSAR is not an error.

⌋

7.5 Implementation-defined behavior

[AP_SWS_GENERAL_90032] Implementation-defined behavior in AUTOSAR APIs

Upstream requirements: RS_AP_00114
Previous identifiers: SWS_CORE_90032

⌈

- Generally: as per `[defns.impl.defined]` in [2]
- Additionally: The precise behavior of IDB in an API is unknown and undocumented by AUTOSAR. The effects are deferred by AUTOSAR to the (out-of-scope) implementation where they shall be well-documented.

⌋

7.6 Vendor-specific behavior

[AP_SWS_GENERAL_90033] Vendor-specific behavior in AUTOSAR APIs

Upstream requirements: RS_AP_00114
Previous identifiers: SWS_CORE_90033

⌈Vendor-specific behavior is equivalent to Implementation-defined behavior ([AP_SWS_GENERAL_90032]) since any "vendor-specifics" are implicitly "implementation-defined".⌋

7.7 Initialization/De-initialization

7.7.1 Common aspects

For this Functional Cluster, the implementation may initialize the ARA framework up front to avoid first-use delays, or keep initialization empty when relying on “initialize on first use”. In both cases, and as required by [SWS_CORE_15002], the following ara::core types and functions are usable before framework initialization and after de-initialization:

Specifically:

- ara::core::ErrorCode and all its member functions and supporting constructs (including non-member operators)
- ara::core::StringView and all its member functions and supporting constructs (including non-member operators)
- ara::core::Result and all its member functions and supporting constructs, except for ara::core::Result::ValueOrThrow
- ara::core::ErrorDomain and all its member functions and its sub-classes, as long as they adhere to [SWS_CORE_10400], but excluding <Prefix>ErrorDomain::ThrowAsException
- ara::core::Initialize
- ara::core::Abort
- ara::core::SetAbortHandler
- ara::core::AddAbortHandler

7.7.2 Initialization

ara::core::Initialize allows a central initialization of all included shared libraries of the ARA framework. This could include initialization of internal data or the setup of daemon links (details are up to the platform vendor).

The general advice for application developers is to call `ara::core::Initialize` right at the entry point of the application.

The `ara::core::Initialize` function can be used to:

- initialization of ARA framework specific data structures
- initialization of system resources
- spawning of background threads

If command line argument injection is used in `ara::core::Initialize`, the relevant command line arguments are removed transparently by this function to allow Application's command line parsing of the other arguments (See [SWS_CORE_15006])

The behavior before initialization of the Adaptive Platform with `ara::core::Initialize` of functions that are not explicitly supported according to [SWS_CORE_15002] or explicitly not supported according to [SWS_CORE_90022] is implementation-defined.

7.7.3 De-initialization

`ara::core::Deinitialize` allows a central deinitialization of all included shared libraries of the ARA framework. This could include the orderly shutdown of background threads or the release of system resources (details are up to the platform vendor).

If a functionality (other than the ones mentioned in [SWS_CORE_15002]) is called after `ara::core::Deinitialize` has been called, the behavior is implementation-defined (see [SWS_CORE_90022]).

Rationale: A check for de-initialization would require runtime checks and semaphores to verify the platform state in each API call. Making this check mandatory would have a significant negative performance impact.

7.8 Logging and Tracing

The Adaptive Platform Foundation provides libraries that are integrated into adaptive application processes (see [5] "Figure: 'Overview Log and Trace'" and [6] "Figure: 'Architecture overview'"). Each function cluster has its own "Context ID" to distinguish these ARA libraries from the application logging and from each other. Further information on Context ID can be found in [6] in chapter: "Context ID".

[AP_SWS_GENERAL_00055] Functional Cluster Log and Trace messages

Upstream requirements: RS_AP_00111
Previous identifiers: SWS_CORE_00055

⌈Each ARA Functional Cluster shall use standardized Logging and Tracing settings from [AP_SWS_GENERAL_90024] when logging their messages via `ara::log`:

- "Log and Trace Context Identifier": for use with DLT (see [PRS_Dlt_01054])
- "Log and Trace Message Identifier range": (see [PRS_Dlt_01062])

⌋

For Log and Trace Message Identifiers, a Functional Cluster Identifier (FCID) specific partitioning should be used:

- FCID field: up to 255 Functional Clusters (8-bit field)
- SMID field: up to 4096 messages per Functional Cluster (12-bit field)

The fields are derived from `msgid` as follows:

- AUTOSAR-standardized Message ID range marker: `(msgid & 0xC0000000) == 0x80000000` (bit 31 = 1 and bit 30 = 0), as required by [PRS_Dlt_01062]
- FCID: `fcid = msgid & 0x000FF000`
- SMID: `smid = msgid & 0x00000FFF`
- Reserved part: `msgid & 0x3FF00000`

Bit allocation:

- bit 31: set to 1 for AUTOSAR standardized modeled Log and Trace messages ([PRS_Dlt_01062])
- bit 30: set to 0 for AUTOSAR standardized modeled Log and Trace messages ([PRS_Dlt_01062])
- bits 29–20: reserved
- bits 19–12: Functional Cluster Identifier (as defined by the Log and Trace Message Identifier ranges in [AP_SWS_GENERAL_90024])
- bits 11–0: Sub Message Identifier

Bit 31	Bit 30	Bits 29–20	Bits 19–12	Bits 11–0
1	0	Reserved	Functional Cluster Identifier (FCID)	Sub Message Identifier (SMID)

Table 7.1: Log and Trace Message Identifier field layout (AUTOSAR standardized modeled messages)

SMID usage:

- Log Messages: increment the SMID from 0 for each new log message within one Functional Cluster.
- Violation Messages: decrement the SMID from 4095 for each violation message within the same Functional Cluster.

This scheme ensures that log and violation messages use distinct value ranges and can be distinguished unambiguously.

[AP_SWS_GENERAL_00056] Log and Trace messages of SWS General

Upstream requirements: RS_AP_00156

⌈General log messages defined in SWS General shall use the same setting as for the functional cluster: `core`⌋

[AP_SWS_GENERAL_90024] AUTOSAR-standardized Functional Cluster Log and Trace settings

Status: DRAFT
Upstream requirements: RS_AP_00130

⌈

Name	*Log and Trace Context Identifier*	*Log and Trace Message Identifier range (lower)*	*Log and Trace Message Identifier range (upper)*
aag	#AAG	0x8001'1000	0x8001'1fff
apbsw	#BSW	0x8000'e000	0x8000'efff
com	#COM	0x8000'0000	0x8000'0fff
core	#COR	0x8000'1000	0x8000'1fff
crypto	#CRY	0x8000'2000	0x8000'2fff
diag	#DIA	0x8000'3000	0x8000'3fff
exec	#EXE	0x8000'4000	0x8000'4fff
fw	#FWX	0x8001'0000	0x8001'0fff
idsm	#IDS	0x8000'5000	0x8000'5fff
log	#LOG	0x8000'6000	0x8000'6fff
nm	#NMX	0x8000'7000	0x8000'7fff
osi	#OSI	0x8001'4000	0x8001'4fff
per	#PER	0x8000'8000	0x8000'8fff
phm	#PHM	0x8000'9000	0x8000'9fff
rds	#RDS	0x8000'f000	0x8000'ffff
rper	#RPR	0x8001'3000	0x8001'3fff
shwa	#SHA	0x8001'2000	0x8001'2fff
sm	#SMX	0x8000'a000	0x8000'afff
tsync	#TSY	0x8000'b000	0x8000'bfff
ucm	#UCM	0x8000'c000	0x8000'cfff
vucm	#VUM	0x8000'd000	0x8000'dfff

⌋

7.9 Thread safety

The [2, the C++14 standard] does not provide explicit criteria for defining thread-safety in functions, it is therefore necessary to provide an AUTOSAR definition to set the criteria during evaluation of whether an AUTOSAR function shall be thread-safe. The definitions are deliberately scoped to AUTOSAR APIs, and do not apply to non-AUTOSAR APIs where no guarantees are given.

[AP_SWS_GENERAL_13200] AUTOSAR definition of a thread-safe member function

Upstream requirements: RS_AP_00164
Previous identifiers: SWS_CORE_13200

⌈An AUTOSAR API non-static member function is thread-safe if the function can be called on the same object concurrently by multiple threads without the need for the caller to synchronize between the calls.⌋

Note: This definition also applies to destructors and assignment operators. They are usually not thread safe. That means concurrent assignments to the same object need to be synchronized by the user of the API.

[AP_SWS_GENERAL_13201] AUTOSAR definition of multiple thread-safe member functions

Upstream requirements: RS_AP_00164
Previous identifiers: SWS_CORE_13201

⌈For all AUTOSAR API member functions of an object that are marked thread-safe it shall be possible to call them on the same object concurrently by multiple threads without the need for the caller to synchronize between the calls.⌋

[AP_SWS_GENERAL_13202] AUTOSAR definition of a thread-safe non-member function

Upstream requirements: RS_AP_00164
Previous identifiers: SWS_CORE_13202

⌈An AUTOSAR API non-member function is thread-safe if the function can be called concurrently by multiple threads without the need for the caller to synchronize between the calls.⌋

[AP_SWS_GENERAL_13211] AUTOSAR definition of a thread-safe static member function

Upstream requirements: RS_AP_00164
Previous identifiers: SWS_CORE_13211

⌈An AUTOSAR API static member function is thread-safe if the function can be called concurrently by multiple threads without the need for the caller to synchronize between the calls.⌋

[AP_SWS_GENERAL_13203] Behavior of a concurrently executed non-thread-safe member function

Upstream requirements: RS_AP_00164
Previous identifiers: SWS_CORE_13203

⌈If a non-thread-safe member function is called concurrently on the same object to itself or other functions of that object, the behavior is undefined.⌋

Note: This applies even if the other function(s) are thread-safe.

[AP_SWS_GENERAL_13204] Behavior of a concurrently executed non-thread-safe non-member function

Upstream requirements: RS_AP_00164
Previous identifiers: SWS_CORE_13204

⌈If a non-thread-safe non-member function is called concurrently, the behavior is undefined.⌋

Note: Adaptive Applications should not call non-thread-safe functions from different threads concurrently.

[AP_SWS_GENERAL_13205] Behavior of concurrently executed functions on different instances

Upstream requirements: RS_AP_00165
Previous identifiers: SWS_CORE_13205

⌈AUTOSAR API member functions can be called concurrently on different instances by multiple threads without the need for the caller to synchronize between the calls.⌋

[AP_SWS_GENERAL_13206] AUTOSAR `callable` definition

Upstream requirements: RS_AP_00130
Previous identifiers: SWS_CORE_13206

⌈The `callable` related definitions in `[func.def]` in [2] shall apply to AUTOSAR APIs when used.⌋

[AP_SWS_GENERAL_13207] Behavior of thread-safe `callable`

Upstream requirements: RS_AP_00164
Previous identifiers: SWS_CORE_13207

⌈The thread-safe `callable` may be called concurrently i.e. in the context of multiple threads or interleaved with multiple `ara::core::Future`/`ara::core::Promise` objects.⌋

[AP_SWS_GENERAL_13212] Definition of `functions implemented by the user of the API`

Upstream requirements: RS_AP_00164
Previous identifiers: SWS_CORE_13212

⌈`Functions implemented by the user of the API` are functions where the signature is defined by the AUTOSAR standard and that are implemented by the user of the API. These are either:

- `callable`s that are passed as callbacks by the user of the API
- pure virtual functions that are implemented by the user of the API

⌋

[AP_SWS_GENERAL_13213] Usage of thread-safe user-implemented functions

Upstream requirements: RS_AP_00164
Previous identifiers: SWS_CORE_13213

⌈The `ara` may assume that `functions implemented by the user of the API` that are defined to be thread-safe in the AUTOSAR standard can be called concurrently to themselves. This also applies to conditionally thread-safe user-implemented functions if their condition determines that they are thread-safe.⌋

[AP_SWS_GENERAL_13214] Usage of non-thread-safe user-implemented functions

Upstream requirements: RS_AP_00164
Previous identifiers: SWS_CORE_13214

⌈The `ara` shall call `functions implemented by the user of the API` that are defined to be non-thread-safe in the AUTOSAR standard non-concurrently (i.e. sequentially) to themselves. This also applies to conditionally thread-safe user-implemented functions if their condition determines that they are non-thread-safe.⌋

An example for the callback is the callback `ara::tsync::SynchronizedTimeBaseNotifier` [SWS_TS_14221]. It is defined to be non-thread-safe. That means the implementer of the function - who is the user of the time sync API - may implement that callback to be non-thread-safe. As a consequence, when that callback is registered via `ara::tsync::SynchronizedTimeBase Consumer::RegisterTimeLeapNotifier(SynchronizedTimeBaseNotifier notifier)` [SWS_TS_01014], the functional cluster implementation needs to make sure that this callback is only called sequentially ([AP_SWS_GENERAL_13214]).

If the callback would be specified to be thread-safe then the user needs to make sure that it is implemented that way. In that case the functional cluster implementation can call that callback concurrently ([AP_SWS_GENERAL_13213]).

The same also applies to pure virtual functions that are implemented by the user of the API and that are then called by the functional cluster implementation. This is a common situation when the Skeleton pattern is used (see also *Offer/StopOffer Pattern* in [7]). For example `ara::phm::RecoveryAction::RecoveryHandler` [SWS_PHM_01142] is defined to be thread-safe. That means when the user of the API implements that function it is expected that this implementation is thread-safe. The Platform Health Management functional cluster implementation may call this function concurrently.

7.10 Async signal safety

An *async-signal-safe* function is one that can be safely called from within a POSIX signal handler.

[8, The POSIX standard] defines a set of functions that are guaranteed to be async-signal-safe; all functions not on that list need to be assumed unsuitable to be called

within a signal handler. This includes all ARA APIs, as it is not specified (and in general not possible to determine) which other functions (whether from POSIX or from other standards or implementations) are called within them.

[AP_SWS_GENERAL_13210] ARA API usage within a signal handler

Upstream requirements: RS_AP_00163
Previous identifiers: SWS_CORE_13210

⌈Usage of any ARA API within a signal handler will result in undefined behavior of the application, unless otherwise specified.⌋

7.11 Error handling

During execution of an implementation of Adaptive Platform APIs, different abnormal conditions might be detected and need to be handled and/or reported. Based on their nature, in the following sub-chapters, the types of unsuccessful operations are distinguished within the Adaptive Platform and shall be treated in different ways.

7.11.1 Traditional error handling in C and C++

The C language largely relies on error codes for any kind of error handling. While it also has the `setjmp/longjmp` facility for performing "non-local gotos", its use for error handling is not widespread, mostly due to the difficulty of reliably avoiding resource leaks.

Error codes in C come in several flavors:

- return values
- out parameters
- error singletons (e.g. `errno`)

Typically, these error codes in C are plain `int` variables, making them a very low-level facility without any type safety.

C++ inherited these approaches to error handling from C (not least due to the inheritance of the C standard library as part of the C++ standard), but it also introduced exceptions as an alternative means of error propagation. There are many advantages of using exceptions for error propagation, which is why the C++ standard library generally relies on them for error propagation.

Notwithstanding the advantages of exceptions, error codes are still in widespread use in C++, even within the standard library. Some of that can be explained with concerns about binary compatibility with C, but many new libraries still prefer error codes to exceptions. Reasons for that include:

- with exceptions, it can be difficult to reason about a program's control flow

- exceptions have much higher runtime cost than error codes (either in general, or only in the exception-thrown case)

The first of these reasons concerns both humans and code analysis tools. Because exceptions are, in effect, a kind of hidden control flow, a C++ function that seems to contain only a single `return` statement might in fact have many additional function returns due to exceptions. That can make such a function hard to review for humans, but also hard to analyze for static code analysis tools.

The second one is even more critical in the context of developing safety-critical software. The specification of C++ exceptions pose significant problems for C++ compiler vendors that want their products be certified for development of safety-critical software. In fact, ASIL-certified C++ compilers generally do not support exceptions at all. One particular problem with exceptions is that exception handling, as specified for C++, implies the use of dynamic memory allocation, which generally has non-predictable or even unbounded execution time. This makes exceptions currently unsuitable for development of certain safety-critical software in the automotive industry.

7.11.2 Violation

[AP_SWS_GENERAL_00021] Semantics of a Violation

Upstream requirements: RS_AP_00142
Previous identifiers: SWS_CORE_00021

⌈A `Violation` is the consequence of failed pre- or post-conditions of an internal state of the application framework. They are the Adaptive Platform's analog to a failed assertion. A `Violation` is non-recoverable.⌋

Note: systematic errors can be considered as a failed pre-condition.

[AP_SWS_GENERAL_00090] Handling of `Standardized Violations`

Upstream requirements: RS_AP_00142
Previous identifiers: SWS_CORE_00090

⌈If a `Violation` is detected that is standardized in the AUTOSAR AP, then the operation shall be terminated by:

- explicitly terminating the process abnormally via a call to `Abort()`

In this case a message to support debugging shall be created according to the DLT message defined for the concrete `Violation`. It shall be output in a string format to the process' standard error stream. Additionally, it should be logged into the log sinks with the "Context Identifier" ([AP_SWS_GENERAL_90024]) of either:

- the functional cluster that detected the `Violation` for the affected process, or
- the `Execution Management`

Which Log and Trace Context Identifier ([AP_SWS_GENERAL_90024]) is chosen is implementation-defined.⌋

Note: The standard does not mandate the direct use of ara::log to allow for optimized solutions, in particular for a fast shutdown of the process without losing the message.

Implementations will have `Violations` that are specific to their implementation approach. Thus it is expected that they will need to detect `Violations` that are not and cannot be standardized.

[AP_SWS_GENERAL_00003] Handling of `Non-Standardized Violations`

Upstream requirements: RS_AP_00142
Previous identifiers: SWS_CORE_00003

⌈If a `Violation` is detected that is not standardized in the AUTOSAR AP, then the operation shall be terminated by either:

- throwing an exception that is not a sub-class of `ara::core::Exception`
- explicitly terminating the process abnormally via a call to `Abort()`

In both cases a message to support debugging should be created.⌋

See also: [4] Messages for unrecoverable errors.

Note: There can be situations in which it might not be possible to generate the message. E.g., when the process is terminated through std::terminate in code that is not changeable by the implementer.

Note: `ara::core::Abort` is ideal for implementing the two previous requirements. In contrast to that, an interface to create a `Violation` implemented as an exception and associated logs is not standardized.

[AP_SWS_GENERAL_00006] Handling of exception-based Violations

Upstream requirements: RS_AP_00142
Previous identifiers: SWS_CORE_00006

⌈If a `Violation` is realized using an `Exception` (that is not a sub-class of `ara::core::Exception`), and the `noexcept` specifier condition ([2] `[except.spec]`) of the function in which it is raised, evaluates to `true`: the C++ runtime will invoke `std::terminate()` automatically.⌋

Note: This will be the case more often than not as AUTOSAR API functions typically evaluate to `noexcept(true)`.

7.11.3 Corruption

[AP_SWS_GENERAL_00022] Semantics of a Corruption

Upstream requirements: RS_AP_00142
Previous identifiers: SWS_CORE_00022

⌈A `Corruption` is the consequence of the corruption of a system resource, e.g. stack or heap overflow, or a hardware memory flaw (including even, for instance, a detected bit flip). A `Corruption` is non-recoverable.⌋

[AP_SWS_GENERAL_00004] Handling of Corruptions

Upstream requirements: RS_AP_00142
Previous identifiers: SWS_CORE_00004

⌈If a `Corruption` is detected, it shall result in unsuccessful process termination, in an implementation-defined way.⌋

Note: It can either be abnormal or normal unsuccessful termination, depending on the implementation's ability to detect the `Corruption` and to react to it by cleaning up resources.

It is expected that a `Violation` or `Corruption` might occur during development of the framework, when new features are just coming together, but will not be experienced by a user (i.e. an application developer), unless there is something seriously wrong in the system's environment (e.g. faulty hardware: `Corruption`), or basic assumptions about resource requirements are violated (`Violation`), or possibly the user runs the framework in a configuration that is not supported by its vendor (`Violation`).

7.11.4 Failed default allocation

[AP_SWS_GENERAL_00023] Semantics of a Failed Default Allocation

Upstream requirements: RS_AP_00142
Previous identifiers: SWS_CORE_00023

⌈A `Failed Default Allocation` is the inability of the framework's default memory allocation mechanism to satisfy an allocation request. A `Failed Default Allocation` is non-recoverable.⌋

[AP_SWS_GENERAL_00005] Handling of failed default allocations

Upstream requirements: RS_AP_00142
Previous identifiers: SWS_CORE_00005

⌈A `Failed Default Allocation` shall be treated the same as a `Non-Standardized Violation` (see [AP_SWS_GENERAL_00003]), except that the DLT message is `FailedDefaultAllocation` ([AP_SWS_GENERAL_13018]).⌋

Note: An error of a custom allocator is not subject to this definition.

[AP_SWS_GENERAL_00007] Handling of exception-based Failed Default Allocations

Upstream requirements: RS_AP_00142
Previous identifiers: SWS_CORE_00007

⌈If a `Failed Default Allocation` is realized using an `Exception` (that is not a sub-class of `ara::core::Exception`), and the `noexcept` specifier condition ([2] `[except.spec]`) of the function in which it is raised, evaluates to `true`: the C++ runtime will invoke `std::terminate()` automatically.⌋

Note: This will be the case more often than not as AUTOSAR API functions typically evaluate to `noexcept(true)`.

7.11.5 Error

[AP_SWS_GENERAL_00020] Semantics of an Error

Upstream requirements: RS_AP_00142
Previous identifiers: SWS_CORE_00020

⌈An `Error` is the inability of an assumed-bug-free API function to fulfill its specified purpose; it is often a consequence of invalid and/or unexpected (i.e. possibly valid, but received in unexpected circumstances) input data. An `Error` is recoverable.⌋

[AP_SWS_GENERAL_00002] Handling of Errors

Upstream requirements: RS_AP_00142, RS_AP_00139, RS_AP_00128
Previous identifiers: SWS_CORE_00002

⌈An `Error` shall be returned from the function as an instance of `ara::core::Result` or `ara::core::Future`.⌋

For handling `Errors`, there are a number of data types defined that help in dealing with them. These are described in the following sub-chapters.

7.11.6 Standardized Error domain Identifiers

Each error domain has a numeric identifier that is used to determine equality of error domains. The error domains that are pre-defined by the Adaptive Platform have standardized identifiers. Application-specific error domains should make sure their identifiers are system-wide unique.

The full range of unique error domain identifiers is partitioned into a range of:

- AUTOSAR-standardized (see [SWS_CORE_00011]) reserved by AUTOSAR; shown in [AP_SWS_GENERAL_90023]
- Vendor-specific Ids: (see [SWS_CORE_00016]); reserved by an AUTOSAR stack vendor

- User-defined Ids: MSB=0; defined via `ApApplicationErrorDomain.value`

Regardless of the origin of an error domain identifier, it shall be unique in the platform.

[AP_SWS_GENERAL_90023] AUTOSAR-standardized Functional Cluster Error Domain Identifiers

Status: DRAFT
Upstream requirements: RS_AP_00130

⌈

Name	*C++ Type*	*Error Domain Identifier*	*Error Domain Name*
com	ara::com::ComErrorDomain	0x8000'0000'0000'1267	Com
com	ara::com::ComE2eErrorDomain	0x8000'0000'0000'1268	ComE2e
com	ara::com::ComSecOcFvmError-Domain	0x8000'0000'0000'1271	ComSecOcFvm
core	ara::core::FutureErrorDomain	0x8000'0000'0000'0013	CoreFuture
core	ara::core::CoreErrorDomain	0x8000'0000'0000'0014	Core
crypto	ara::crypto::CryptoErrorDomain	0x8000'0000'0000'0801	Crypto
diag	ara::diag::DiagErrorDomain	0x8000'0000'0000'0401	Diag
diag	ara::diag::DiagReportingError-Domain	0x8000'0000'0000'0402	DiagReporting
diag	ara::diag::DiagOfferErrorDomain	0x8000'0000'0000'0403	DiagOffer
diag	ara::diag::DiagUdsNrcError-Domain	0x8000'0000'0000'0411	DiagUdsNrc
diag	ara::diag::DiagSovdErrorDomain	0x8000'0000'0000'0412	DiagSovd
exec	ara::exec::ExecErrorDomain	0x8000'0000'0000'0202	Exec
fw	ara::fw::FwErrorDomain	0x8000'0000'0000'0501	Fw
idsm	ara::idsm::IdsmErrorDomain	0x8000'0000'0000'1101	Idsm
nm	ara::nm::NmErrorDomain	0x8000'0000'0000'1001	Nm
per	ara::per::PerErrorDomain	0x8000'0000'0000'0101	Per
phm	ara::phm::PhmErrorDomain	0x8000'0000'0000'0602	Phm
rds	ara::rds::RdsErrorDomain	0x8000'0000'0000'1280	Rds
rper	ara::rper::RemotePerErrorDomain	0x8000'0000'0000'0102	RemotePer
shwa	ara::shwa::ShwaErrorDomain	0x8000'0000'0000'1302	Shwa
sm	ara::sm::SmErrorDomain	0x8000'0000'0000'0301	Sm
tsync	ara::tsync::TsyncErrorDomain	0x8000'0000'0000'0901	Tsync
tsync	ara::tsync::TsyncFvmErrorDomain	0x8000'0000'0000'0911	TsyncFvm
ucm	ara::ucm::UcmErrorDomain	0x8000'0000'0000'0701	Ucm
vucm	ara::vucm::VucmErrorDomain	0x8000'0000'0000'1301	Vucm

⌋

7.12 Reporting

7.12.1 Security Events

This functional cluster does not define any security events.

7.12.2 Log Messages

This section lists all non-verbose log messages (i.e., modelled DLT messages) defined by this functional cluster.

7.12.2.1 `FailedDefaultAllocation`

[AP_SWS_GENERAL_13018] Log Message FailedDefaultAllocation

Status: DRAFT
Upstream requirements: RS_AP_00142
Previous identifiers: SWS_CORE_13018

⌈

DIt-Message	FailedDefaultAllocation		
Description	Sent in case the affected process was terminated because of a failed default allocation		
MessageId	0x80001000		
MessageType Info	DLT_LOG_FATAL		
DIt-Argument	***ArgumentDescription***	***ArgumentType***	***ArgumentUnit***
modeledProcessId	Meta-model identifier of the process that caused the violation, i.e., its short name path with '/' as a separator	uint8 [encoding UTF-8]	
location	An implementation-defined identifier of the location where the violation was detected, for example {filename}:{linenumber}	uint8 [encoding UTF-8]	
message	Additional message that can describe the cause of the failed default allocation in more detail	uint8 [encoding UTF-8]	

⌋

7.12.3 Violation Messages

This section lists all violation messages (i.e., DLT messages logged for Violations according to [AP_SWS_GENERAL_00021]) defined by this functional cluster.
Please note that concrete implementations might additionally implement Non-Standardized Violations (see also [AP_SWS_GENERAL_00003]).

7.12.3.1 AraNotInitializedViolation

[AP_SWS_GENERAL_13007] Violation Message AraNotInitializedViolation

Status: DRAFT
Upstream requirements: RS_AP_00142
Previous identifiers: SWS_CORE_13007

⌈

Dlt-Message	AraNotInitializedViolation		
Description	Violation message that is sent in case a constructor or function checks for an initialized ara and identifies that ara is not initialized.		
MessageId	0x80001ff8		
MessageType Info	DLT_LOG_FATAL		
Dlt-Argument	***ArgumentDescription***	***ArgumentType***	***ArgumentUnit***
modeledProcessId	Meta-model identifier of the process that caused the violation, i.e., its short name path with '/' as a separator.	uint8 [encoding UTF-8]	
location	An implementation-defined identifier of the location where the violation was detected, for example {filename}:{linenumber}.	uint8 [encoding UTF-8]	
instanceSpecifier	InstanceSpecifier used to try to create the object. (only present in case this violation is created by a function/constructor that has an InstanceSpecifier as an argument)	uint8 [encoding UTF-8]	
functionName	Name of the function/constructor that was called that checks for an initialized ara.	uint8 [encoding UTF-8]	

⌋

7.12.3.2 InstanceSpecifierAlreadyInUseViolation

[AP_SWS_GENERAL_13006] Violation Message InstanceSpecifierAlreadyInUse Violation

Status: DRAFT
Upstream requirements: RS_AP_00142
Previous identifiers: SWS_CORE_13006

⌈

Dlt-Message	InstanceSpecifierAlreadyInUseViolation		
Description	Violation message that is sent in case a constructor in the ara framework was called with an Instance Specifier already in use in this process.		
MessageId	0x80001ff9		
MessageType Info	DLT_LOG_FATAL		
Dlt-Argument	***ArgumentDescription***	***ArgumentType***	***ArgumentUnit***
modeledProcessId	Meta-model identifier of the process that caused the violation, i.e., its short name path with '/' as a separator.	uint8 [encoding UTF-8]	

▽

△

location	An implementation-defined identifier of the location where the violation was detected, for example {filename}:{linenumber}.	uint8 [encoding UTF-8]	
instanceSpecifier	InstanceSpecifier used to try to create the object.	uint8 [encoding UTF-8]	
className	Name of the class that was instantiated.	uint8 [encoding UTF-8]	

⌋

7.12.3.3 UnresolvableInstanceSpecifierViolation

[AP_SWS_GENERAL_13215] Violation Message UnresolvableInstanceSpecifier Violation

Status: DRAFT
Upstream requirements: RS_AP_00142

⌈

DIt-Message	UnresolvableInstanceSpecifierViolation		
Description	InstanceSpecifier either cannot be resolved in the model in the context of your executable or process, or it refers to a model element that is neither a PortPrototype of the expected type nor a FunctionalCluster InteractsWithFunctionalClusterMapping of the expected type.		
MessageId	0x80001fe8		
MessageType Info	DLT_LOG_FATAL		
DIt-Argument	***ArgumentDescription***	***ArgumentType***	***ArgumentUnit***
modeledProcessId	Meta-model identifier of the process that caused the violation, i.e., its short name path with '/' as a separator	uint8 [encoding UTF-8]	
location	An implementation-defined identifier of the location where the violation was detected, for example {ClassName}:{MethodName}, {filename}:{linenumber}, etc.	uint8 [encoding UTF-8]	
instanceSpecifier	InstanceSpecifier used to try to create the object.	uint8 [encoding UTF-8]	
className	Name of the class that was instantiated.	uint8 [encoding UTF-8]	

⌋

7.12.3.4 InsufficientPermissionsViolation

[AP_SWS_GENERAL_13000] Violation Message InsufficientPermissionsViolation

Status: DRAFT
Upstream requirements: RS_AP_00142
Previous identifiers: SWS_CORE_13000

⌈

DIt-Message	InsufficientPermissionsViolation		
Description	Sent in case the caller had insufficient permissions for the requested operation.		
MessageId	0x80001fff		
MessageType Info	DLT_LOG_FATAL		
DIt-Argument	***ArgumentDescription***	***ArgumentType***	***ArgumentUnit***
modeledProcessId	Meta-model identifier of the process that caused the violation, i.e., its short name path with '/' as a separator.	uint8 [encoding UTF-8]	
location	An implementation-defined identifier of the location where the violation was detected, for example {filename}:{linenumber}.	uint8 [encoding UTF-8]	
message	Additional message that describes the cause of the access violation.	uint8 [encoding UTF-8]	

⌋

7.12.4 Production Errors

This functional cluster does not define any production errors (i.e., Diagnostic Events).

8 API specification

This specification does not specify a functional cluster. This chapter does not apply.

9 Service Interfaces

This specification does not specify a functional cluster. This chapter does not apply.

10 Configuration

This specification does not specify a functional cluster. This chapter does not apply.

A Mentioned Manifest Elements

This specification does not specify a functional cluster. This chapter does not apply.

B Demands and constraints on Base Software (normative)

This specification does not specify a functional cluster. This chapter does not apply.

C Platform Extension Interfaces (normative)

This specification does not specify a functional cluster. This chapter does not apply.

D Not implemented requirements

This functional cluster implements all functional requirements specified in the corresponding requirement specifications.

E Change history of AUTOSAR traceable items

This chapter provides an overview of the history of constraints and specification items. Please note that the lists in this chapter also include constraints and specification items that have been removed from the specification in a later version. These constraints and specification items do not appear as hyperlinks in the document.

E.1 Traceable item history of this document according to AUTOSAR Release R26-11

E.1.1 Added Specification Items in R26-11

Number	Heading
[AP_SWS_GENERAL_00002]	Handling of Errors
[AP_SWS_GENERAL_00003]	Handling of `Non-Standardized Violations`
[AP_SWS_GENERAL_00004]	Handling of Corruptions
[AP_SWS_GENERAL_00005]	Handling of failed default allocations
[AP_SWS_GENERAL_00006]	Handling of exception-based Violations
[AP_SWS_GENERAL_00007]	Handling of exception-based Failed Default Allocations
[AP_SWS_GENERAL_00020]	Semantics of an Error
[AP_SWS_GENERAL_00021]	Semantics of a Violation
[AP_SWS_GENERAL_00022]	Semantics of a Corruption
[AP_SWS_GENERAL_00023]	Semantics of a Failed Default Allocation
[AP_SWS_GENERAL_00055]	Functional Cluster Log and Trace messages
[AP_SWS_GENERAL_00056]	Log and Trace messages of SWS General
[AP_SWS_GENERAL_00090]	Handling of `Standardized Violations`
[AP_SWS_GENERAL_13000]	Violation Message InsufficientPermissionsViolation

▽

△

Number	Heading
[AP_SWS_GENERAL_13006]	Violation Message InstanceSpecifierAlreadyInUseViolation
[AP_SWS_GENERAL_13007]	Violation Message AraNotInitializedViolation
[AP_SWS_GENERAL_13018]	Log Message FailedDefaultAllocation
[AP_SWS_GENERAL_13200]	AUTOSAR definition of a thread-safe member function
[AP_SWS_GENERAL_13201]	AUTOSAR definition of multiple thread-safe member functions
[AP_SWS_GENERAL_13202]	AUTOSAR definition of a thread-safe non-member function
[AP_SWS_GENERAL_13203]	Behavior of a concurrently executed non-thread-safe member function
[AP_SWS_GENERAL_13204]	Behavior of a concurrently executed non-thread-safe non-member function
[AP_SWS_GENERAL_13205]	Behavior of concurrently executed functions on different instances
[AP_SWS_GENERAL_13206]	AUTOSAR `callable` definition
[AP_SWS_GENERAL_13207]	Behavior of thread-safe `callable`
[AP_SWS_GENERAL_13210]	ARA API usage within a signal handler
[AP_SWS_GENERAL_13211]	AUTOSAR definition of a thread-safe static member function
[AP_SWS_GENERAL_13212]	Definition of `functions implemented by the user of the API`
[AP_SWS_GENERAL_13213]	Usage of thread-safe user-implemented functions
[AP_SWS_GENERAL_13214]	Usage of non-thread-safe user-implemented functions
[AP_SWS_GENERAL_13215]	Violation Message UnresolvableInstanceSpecifierViolation
[AP_SWS_GENERAL_90001]	Include folder structure
[AP_SWS_GENERAL_90002]	Prevent multiple inclusion of header file
[AP_SWS_GENERAL_90003]	Reservation of ARA preprocessor macros
[AP_SWS_GENERAL_90004]	Consistency of Implementation-defined declaration classifiers

▽

△

Number	Heading
[AP_SWS_GENERAL_90005]	Custom declarations and definitions
[AP_SWS_GENERAL_90007]	Potentially throwing constructors
[AP_SWS_GENERAL_90008]	Addition of `[[nodiscard]]`
[AP_SWS_GENERAL_90023]	AUTOSAR-standardized Functional Cluster Error Domain Identifiers
[AP_SWS_GENERAL_90024]	AUTOSAR-standardized Functional Cluster Log and Trace settings
[AP_SWS_GENERAL_90025]	AUTOSAR-standardized Functional Cluster names
[AP_SWS_GENERAL_90031]	Undefined behavior in AUTOSAR APIs
[AP_SWS_GENERAL_90032]	Implementation-defined behavior in AUTOSAR APIs
[AP_SWS_GENERAL_90033]	Vendor-specific behavior in AUTOSAR APIs

Table E.1: Added Specification Items in R26-11

E.1.2 Changed Specification Items in R26-11

none

E.1.3 Deleted Specification Items in R26-11

none