Document Title	Specification of the Adaptive Core
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	903

Document Status	published
Part of AUTOSAR Standard	Adaptive Platform
Part of Standard Release	R20-11

Document Change History

Date	Release	Changed by	Description
2020-11-30	R20-11	AUTOSAR Release Management	• Add specifications about “Explicit Operation Abortion” • Add specification about reserved symbol prefixes • Add specification of class SteadyClock • Add section about async signal safety of ARA APIs • Extend error domain scope with vendor-defined error domains • Add specifications about defining own error domains • Various extensions and fixes to the C++ data types • Incorporate contents of SWS_General • Rename document into “Adaptive Core”

2019-11-28	R19-11	AUTOSAR Release Management	• Rework error handling definitions • Add specifications of BasicString and Byte, and add overloads and template specializations for ErrorCode, Result, Future, and Promise • Add bits about validity of InstanceSpecifier arguments, and rework the specification of its construction mechanism • Rework ErrorCode to get rid of “User Message” and make “SupportDataType” implementation-defined • Replace PosixErrorDomain with CoreErrorDomain • Rename FutureErrorDomain accessor function • Changed Document Status from Final to published
2019-03-29	19-03	AUTOSAR Release Management	• Add specification of the template specialization Result<void, E>
2018-10-31	18-10	AUTOSAR Release Management	• Add chapter 2 with acronyms • Add chapter 4 with limitations of the current specifications • Add chapter 5 with dependencies to other modules • Add chapter 7 • Add classes representing the approach to error handling to chapter 8 • Adapt classes Future and Promise to the error handling approach • Add global functions for initialization and shutdown of the framework • Add class InstanceSpecifier to chapter 8 • Add more types and functions from the C++ standard
2018-03-29	18-03	AUTOSAR Release Management	• Initial Release

Disclaimer

Table of Contents

1 Introduction

This document defines basic requirements that apply to all Functional Clusters of the Adaptive Platform.

To aid in this, it also defines functionality that applies to the entire framework, including a set of common data types used by multiple Functional Clusters as part of their public interfaces.

2 Acronyms and Abbreviations

The glossary below includes acronyms and abbreviations relevant to Adaptive Core that are not included in the [1, AUTOSAR glossary].

Term	Description
Explicit Operation Abortion	Immediate abortion of an API call, which is initiated by calling `ara::core::Abort`, usually as a consequence of the detection of a `Violation`.
UUID	*Universally Unique Identifier*, a 128-bit number used to identify information in computer systems

3 Related documentation

3.1 Input documents & related standards and norms

[1] Glossary
AUTOSAR_TR_Glossary

[2] Specification of Operating System Interface
AUTOSAR_SWS_OperatingSystemInterface

[3] Functional Cluster Shortnames
AUTOSAR_TR_FunctionalClusterShortnames

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

[5] ISO/IEC 14882:2011, Information technology – Programming languages – C++
http://www.iso.org

[6] ValueOrError and ValueOrNone types
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0786r1.pdf

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

[8] Specification of Execution Management
AUTOSAR_SWS_ExecutionManagement

[9] Explanation of ara::com API
AUTOSAR_EXP_ARAComAPI

[10] N4659: Working Draft, Standard for ProgrammingLanguage C++
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf

[11] N4820: Working Draft, Standard for Programming Language C++
http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2019/n4820.pdf

[12] N3857: Improvements to std::future<T> and Related APIs
https://isocpp.org/files/papers/N3857.pdf

4 Constraints and assumptions

4.1 Limitations

- The specification of some data types (Array, Map, Optional, String, StringView, Variant) mentions “supporting constructs”, but lacks a precise scope definition of this term.
- The specification of some data types (Map, Vector, String) is lacking a comprehensive definition of memory allocation behavior; it currently only describes it as “implementation-defined”.
- Chapter 7.2 (“Functional Specification”) describes some behavior informally that should rather be given as specification items.

4.2 Applicability to car domains

No restrictions to applicability.

5 Dependencies to other modules

This Functional Cluster only depends on [2], in particular the C++11 standard library.

6 Requirements Tracing

The following tables reference the requirements specified in <CITATIONS_OF_CONTRIBUTED_DOCUMENTS> and links to the fulfillment of these. Please note that if column “Satisfied by” is empty for a specific requirement this means that this requirement is not fulfilled by this document.

Requirement	Description	Satisfied by
[RS_AP_00111]	The AUTOSAR Adaptive Platform shall support source code portability for AUTOSAR Adaptive applications.	[SWS_CORE_90001] [SWS_CORE_90002] [SWS_CORE_90003]
[RS_AP_00116]	Header file name.	[SWS_CORE_90001]
[RS_AP_00127]	Usage of ara::core types.	[SWS_CORE_00052]
[RS_AP_00128]	Error reporting.	[SWS_CORE_00002]
[RS_AP_00130]	AUTOSAR Adaptive Platform shall represent a rich and modern programming environment.	[SWS_CORE_00010] [SWS_CORE_00011] [SWS_CORE_00013] [SWS_CORE_00014] [SWS_CORE_00016] [SWS_CORE_00040] [SWS_CORE_00110] [SWS_CORE_00121] [SWS_CORE_00122] [SWS_CORE_00123] [SWS_CORE_00131] [SWS_CORE_00132] [SWS_CORE_00133] [SWS_CORE_00134] [SWS_CORE_00135] [SWS_CORE_00136] [SWS_CORE_00137] [SWS_CORE_00138] [SWS_CORE_00151] [SWS_CORE_00152] [SWS_CORE_00153] [SWS_CORE_00154] [SWS_CORE_00321] [SWS_CORE_00322]

Requirement	Description	Satisfied by
		[SWS_CORE_00323] [SWS_CORE_00325] [SWS_CORE_00326] [SWS_CORE_00327] [SWS_CORE_00328] [SWS_CORE_00329] [SWS_CORE_00330] [SWS_CORE_00331] [SWS_CORE_00332] [SWS_CORE_00333] [SWS_CORE_00334] [SWS_CORE_00335] [SWS_CORE_00336] [SWS_CORE_00337] [SWS_CORE_00340] [SWS_CORE_00341] [SWS_CORE_00342] [SWS_CORE_00343] [SWS_CORE_00344] [SWS_CORE_00345] [SWS_CORE_00346] [SWS_CORE_00349] [SWS_CORE_00350] [SWS_CORE_00351] [SWS_CORE_00352] [SWS_CORE_00353] [SWS_CORE_00354] [SWS_CORE_00355] [SWS_CORE_00356] [SWS_CORE_00361] [SWS_CORE_00400] [SWS_CORE_00411] [SWS_CORE_00412] [SWS_CORE_00421] [SWS_CORE_00431] [SWS_CORE_00432] [SWS_CORE_00441] [SWS_CORE_00442] [SWS_CORE_00443] [SWS_CORE_00444] [SWS_CORE_00480] [SWS_CORE_00490] [SWS_CORE_00501] [SWS_CORE_00512] [SWS_CORE_00513] [SWS_CORE_00514] [SWS_CORE_00515] [SWS_CORE_00516]

Requirement	Description	Satisfied by
		[SWS_CORE_00518] [SWS_CORE_00519] [SWS_CORE_00571] [SWS_CORE_00572] [SWS_CORE_00601] [SWS_CORE_00611] [SWS_CORE_00612] [SWS_CORE_00613] [SWS_CORE_00614] [SWS_CORE_00701] [SWS_CORE_00711] [SWS_CORE_00712] [SWS_CORE_00721] [SWS_CORE_00722] [SWS_CORE_00723] [SWS_CORE_00724] [SWS_CORE_00725] [SWS_CORE_00726] [SWS_CORE_00727] [SWS_CORE_00731] [SWS_CORE_00732] [SWS_CORE_00733] [SWS_CORE_00734] [SWS_CORE_00735] [SWS_CORE_00736] [SWS_CORE_00742] [SWS_CORE_00743] [SWS_CORE_00744] [SWS_CORE_00745] [SWS_CORE_00751] [SWS_CORE_00752] [SWS_CORE_00753] [SWS_CORE_00754] [SWS_CORE_00755] [SWS_CORE_00756] [SWS_CORE_00757] [SWS_CORE_00758] [SWS_CORE_00759] [SWS_CORE_00761] [SWS_CORE_00762] [SWS_CORE_00763] [SWS_CORE_00764] [SWS_CORE_00765] [SWS_CORE_00766] [SWS_CORE_00767] [SWS_CORE_00768] [SWS_CORE_00769] [SWS_CORE_00770]

Requirement	Description	Satisfied by
		[SWS_CORE_00771] [SWS_CORE_00772] [SWS_CORE_00773] [SWS_CORE_00780] [SWS_CORE_00781] [SWS_CORE_00782] [SWS_CORE_00783] [SWS_CORE_00784] [SWS_CORE_00785] [SWS_CORE_00786] [SWS_CORE_00787] [SWS_CORE_00788] [SWS_CORE_00789] [SWS_CORE_00796] [SWS_CORE_00801] [SWS_CORE_00811] [SWS_CORE_00812] [SWS_CORE_00821] [SWS_CORE_00823] [SWS_CORE_00824] [SWS_CORE_00825] [SWS_CORE_00826] [SWS_CORE_00827] [SWS_CORE_00831] [SWS_CORE_00834] [SWS_CORE_00835] [SWS_CORE_00836] [SWS_CORE_00842] [SWS_CORE_00843] [SWS_CORE_00844] [SWS_CORE_00845] [SWS_CORE_00851] [SWS_CORE_00852] [SWS_CORE_00853] [SWS_CORE_00855] [SWS_CORE_00857] [SWS_CORE_00858] [SWS_CORE_00861] [SWS_CORE_00863] [SWS_CORE_00864] [SWS_CORE_00865] [SWS_CORE_00866] [SWS_CORE_00867] [SWS_CORE_00868] [SWS_CORE_00869] [SWS_CORE_00870] [SWS_CORE_01030] [SWS_CORE_01031]

Requirement	Description	Satisfied by
		[SWS_CORE_01033] [SWS_CORE_01096] [SWS_CORE_01201] [SWS_CORE_01210] [SWS_CORE_01211] [SWS_CORE_01212] [SWS_CORE_01213] [SWS_CORE_01214] [SWS_CORE_01215] [SWS_CORE_01216] [SWS_CORE_01217] [SWS_CORE_01218] [SWS_CORE_01219] [SWS_CORE_01220] [SWS_CORE_01241] [SWS_CORE_01242] [SWS_CORE_01250] [SWS_CORE_01251] [SWS_CORE_01252] [SWS_CORE_01253] [SWS_CORE_01254] [SWS_CORE_01255] [SWS_CORE_01256] [SWS_CORE_01257] [SWS_CORE_01258] [SWS_CORE_01259] [SWS_CORE_01260] [SWS_CORE_01261] [SWS_CORE_01262] [SWS_CORE_01263] [SWS_CORE_01264] [SWS_CORE_01265] [SWS_CORE_01266] [SWS_CORE_01267] [SWS_CORE_01268] [SWS_CORE_01269] [SWS_CORE_01270] [SWS_CORE_01271] [SWS_CORE_01272] [SWS_CORE_01280] [SWS_CORE_01281] [SWS_CORE_01282] [SWS_CORE_01283] [SWS_CORE_01284] [SWS_CORE_01285] [SWS_CORE_01290] [SWS_CORE_01291] [SWS_CORE_01292]

Requirement	Description	Satisfied by
		[SWS_CORE_01293] [SWS_CORE_01294] [SWS_CORE_01295] [SWS_CORE_01296] [SWS_CORE_01301] [SWS_CORE_01390] [SWS_CORE_01391] [SWS_CORE_01392] [SWS_CORE_01393] [SWS_CORE_01394] [SWS_CORE_01395] [SWS_CORE_01396] [SWS_CORE_01400] [SWS_CORE_01496] [SWS_CORE_01601] [SWS_CORE_01696] [SWS_CORE_01900] [SWS_CORE_01901] [SWS_CORE_01911] [SWS_CORE_01912] [SWS_CORE_01913] [SWS_CORE_01914] [SWS_CORE_01915] [SWS_CORE_01916] [SWS_CORE_01917] [SWS_CORE_01918] [SWS_CORE_01919] [SWS_CORE_01920] [SWS_CORE_01921] [SWS_CORE_01931] [SWS_CORE_01941] [SWS_CORE_01942] [SWS_CORE_01943] [SWS_CORE_01944] [SWS_CORE_01945] [SWS_CORE_01946] [SWS_CORE_01947] [SWS_CORE_01948] [SWS_CORE_01949] [SWS_CORE_01950] [SWS_CORE_01951] [SWS_CORE_01952] [SWS_CORE_01961] [SWS_CORE_01962] [SWS_CORE_01963] [SWS_CORE_01964] [SWS_CORE_01965] [SWS_CORE_01966]

Requirement	Description	Satisfied by
		[SWS_CORE_01967] [SWS_CORE_01968] [SWS_CORE_01969] [SWS_CORE_01970] [SWS_CORE_01971] [SWS_CORE_01972] [SWS_CORE_01973] [SWS_CORE_01974] [SWS_CORE_01975] [SWS_CORE_01976] [SWS_CORE_01977] [SWS_CORE_01978] [SWS_CORE_01979] [SWS_CORE_01980] [SWS_CORE_01981] [SWS_CORE_01990] [SWS_CORE_01991] [SWS_CORE_01992] [SWS_CORE_01993] [SWS_CORE_01994] [SWS_CORE_02001] [SWS_CORE_03000] [SWS_CORE_03001] [SWS_CORE_03296] [SWS_CORE_03301] [SWS_CORE_03302] [SWS_CORE_03303] [SWS_CORE_03304] [SWS_CORE_03305] [SWS_CORE_03306] [SWS_CORE_03307] [SWS_CORE_03308] [SWS_CORE_03309] [SWS_CORE_03310] [SWS_CORE_03311] [SWS_CORE_03312] [SWS_CORE_03313] [SWS_CORE_03314] [SWS_CORE_03315] [SWS_CORE_03316] [SWS_CORE_03317] [SWS_CORE_03318] [SWS_CORE_03319] [SWS_CORE_03320] [SWS_CORE_03321] [SWS_CORE_03322] [SWS_CORE_03323] [SWS_CORE_04011]

Requirement	Description	Satisfied by
		[SWS_CORE_04012] [SWS_CORE_04013] [SWS_CORE_04021] [SWS_CORE_04022] [SWS_CORE_04023] [SWS_CORE_04031] [SWS_CORE_04032] [SWS_CORE_04033] [SWS_CORE_04110] [SWS_CORE_04111] [SWS_CORE_04112] [SWS_CORE_04113] [SWS_CORE_04120] [SWS_CORE_04121] [SWS_CORE_04130] [SWS_CORE_04131] [SWS_CORE_04132] [SWS_CORE_04200] [SWS_CORE_05200] [SWS_CORE_05211] [SWS_CORE_05212] [SWS_CORE_05221] [SWS_CORE_05231] [SWS_CORE_05232] [SWS_CORE_05241] [SWS_CORE_05242] [SWS_CORE_05243] [SWS_CORE_05244] [SWS_CORE_05280] [SWS_CORE_05290] [SWS_CORE_06221] [SWS_CORE_06222] [SWS_CORE_06223] [SWS_CORE_06225] [SWS_CORE_06226] [SWS_CORE_06227] [SWS_CORE_06228] [SWS_CORE_06229] [SWS_CORE_06230] [SWS_CORE_06231] [SWS_CORE_06232] [SWS_CORE_06233] [SWS_CORE_06234] [SWS_CORE_06235] [SWS_CORE_06236] [SWS_CORE_06237] [SWS_CORE_06340] [SWS_CORE_06341]

Requirement	Description	Satisfied by
		[SWS_CORE_06342] [SWS_CORE_06343] [SWS_CORE_06344] [SWS_CORE_06345] [SWS_CORE_06349] [SWS_CORE_06350] [SWS_CORE_06351] [SWS_CORE_06352] [SWS_CORE_06353] [SWS_CORE_06354] [SWS_CORE_06355] [SWS_CORE_06356] [SWS_CORE_06401] [SWS_CORE_06411] [SWS_CORE_06412] [SWS_CORE_06413] [SWS_CORE_06414] [SWS_CORE_06431] [SWS_CORE_06432] [SWS_CORE_10100] [SWS_CORE_10101] [SWS_CORE_10102] [SWS_CORE_10103] [SWS_CORE_10104] [SWS_CORE_10105] [SWS_CORE_10106] [SWS_CORE_10107] [SWS_CORE_10108] [SWS_CORE_10109] [SWS_CORE_10110] [SWS_CORE_10200] [SWS_CORE_10201] [SWS_CORE_10202] [SWS_CORE_10300] [SWS_CORE_10400] [SWS_CORE_10900] [SWS_CORE_10901] [SWS_CORE_10902] [SWS_CORE_10903] [SWS_CORE_10910] [SWS_CORE_10911] [SWS_CORE_10912] [SWS_CORE_10930] [SWS_CORE_10931] [SWS_CORE_10932] [SWS_CORE_10933] [SWS_CORE_10934] [SWS_CORE_10950]

Requirement	Description	Satisfied by
		[SWS_CORE_10951] [SWS_CORE_10952] [SWS_CORE_10953] [SWS_CORE_10980] [SWS_CORE_10981] [SWS_CORE_10982] [SWS_CORE_10990] [SWS_CORE_10991] [SWS_CORE_10999] [SWS_CORE_11200] [SWS_CORE_11800] [SWS_CORE_11801] [SWS_CORE_12402] [SWS_CORE_12403] [SWS_CORE_12404] [SWS_CORE_12405] [SWS_CORE_12406] [SWS_CORE_12407]
[RS_AP_00132]	noexcept behavior of API functions	[SWS_CORE_00050] [SWS_CORE_00051] [SWS_CORE_00052] [SWS_CORE_00053]
[RS_AP_00134]	noexcept behavior of class destructors	[SWS_CORE_08029]
[RS_AP_00136]	Usage of string types.	[SWS_CORE_00052] [SWS_CORE_08032]
[RS_AP_00137]	Connecting run-time interface with model.	[SWS_CORE_08032]
[RS_AP_00139]	Return type of synchronous function calls.	[SWS_CORE_00002]
[RS_AP_00140]	Usage of "final specifier" in ara types.	[SWS_CORE_00501] [SWS_CORE_08001] [SWS_CORE_10932]
[RS_AP_00142]	Handling of unsuccessful operations.	[SWS_CORE_00002] [SWS_CORE_00003] [SWS_CORE_00004] [SWS_CORE_00005]
[RS_Main_00011]	AUTOSAR shall support the development of reliable systems	[SWS_CORE_10001] [SWS_CORE_10002]
[RS_Main_00150]	AUTOSAR shall support the deployment and reallocation of AUTOSAR Application Software	[SWS_CORE_08032]

Requirement	Description	Satisfied by
[RS_Main_00320]	AUTOSAR shall provide formats to specify system development	[SWS_CORE_08001] [SWS_CORE_08021] [SWS_CORE_08022] [SWS_CORE_08023] [SWS_CORE_08024] [SWS_CORE_08025] [SWS_CORE_08029] [SWS_CORE_08041] [SWS_CORE_08042] [SWS_CORE_08043] [SWS_CORE_08044] [SWS_CORE_08045] [SWS_CORE_08046] [SWS_CORE_08081] [SWS_CORE_08082]

7 Requirements Specification

7.1 General requirements for all Functional Clusters

The goal of this section 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.

[SWS_CORE_90001] Include folder structure ⌈All #include directives in header files that refer to ARA libraries shall be written in the form

```
#include "ara/fc/header.h"
```

with “ara” as the first path element, “fc” being the remaining directory path of the implementation’s *installed* header file, starting with the Functional Cluster short name, and “header.h” being the filename of the header file.⌋*(RS_AP_00116, RS_AP_00111)*

The Functional Cluster short names are defined in [3].

Example: Execution Management (short name “exec”) provides class `ExecutionClient`, which can be accessed with:

```
#include "ara/exec/execution_client.h"
```

The "..." form of #include statements shall be used, due to the recommendation given in [4, the C++14 standard] section 16.2.7.

[SWS_CORE_90002] Prevent multiple inclusion of header file ⌈All public header files shall prevent multiple inclusion by using #include guards that are likely to be system-wide unique.⌋*(RS_AP_00111)*

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_`.

[SWS_CORE_90003]{DRAFT} ⌈C/C++ symbols that start with `ARA` are reserved for use by AUTOSAR.⌋*(RS_AP_00111)*

The Adaptive Platform generally avoids the use of C/C++ preprocessor macros. However, in case macros are introduced at some later point in time, any such macro will 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.

7.2 Functional Specification

This section describes the concepts that are introduced with this Functional Cluster. Particular emphasis is put on error handling.

7.2.1 Error handling

7.2.1.1 Types of unsuccessful operations

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, the following types of unsuccessful operations are distinguished within the Adaptive Platform:

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 considered to be recoverable.

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

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 considered to be non-recoverable.

A `failed default allocation` is the inability of the framework's default memory allocation mechanism to satisfy an allocation request.

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`).

7.2.1.2 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 analyse 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.

7.2.1.3 Handling of unsuccessful operations in the Adaptive Platform

The types of unsuccessful operations defined in section 7.2.1.1 ("Types of unsuccessful operations") are to be treated in different ways.

[SWS_CORE_00002] Handling of Errors ⌈An `Error` shall be returned from the function as an instance of `ara::core::Result` or `ara::core::Future`.⌋*(RS_AP_00142, RS_AP_00139, RS_AP_00128)*

[SWS_CORE_00003] Handling of Violations ⌈If a `Violation` is detected, its occurrence shall be logged as a message of FATAL severity (if logging is enabled for

the respective Functional Cluster of the Adaptive Platform implementation), then the operation shall be terminated by either:

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

⌋(*RS_AP_00142*)

[SWS_CORE_00004] Handling of Corruptions ⌈If a `Corruption` is detected, it shall result in unsuccessful process termination, in an implementation-defined way.⌋(*RS_AP_00142*)
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.

[SWS_CORE_00005] Handling of failed default allocations ⌈A "failed default allocation" shall be treated the same as a `Violation`.⌋(*RS_AP_00142*)
Note: An error of a custom allocator is not subject to this definition.

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

7.2.1.3.1 ErrorCode

As its name implies, `ara::core::ErrorCode` is a form of error code; however, it is a class type, loosely modeled on `std::error_code`, and thus allows much more sophisticated handling of errors than the simple error codes as used in typical C APIs. It always contains a low-level `error code value` and a reference to an `error domain`.

The `error code value` is an enumeration, typically a scoped one. When stored into a `ara::core::ErrorCode`, it is type-erased into an integral type and thus handled similarly to a C-style error code. The `error domain` reference defines the context for which the `error code value` is applicable and thus provides some measure of type safety.

An `ara::core::ErrorCode` also contains a `support data value`, which *can* be defined by an implementation of the Adaptive Platform to give a vendor-specific additional piece of data about the error.

`ara::core::ErrorCode` instances are usually not created directly, but only via the forwarding form of the function `ara::core::Result::FromError`.

An `ara::core::ErrorCode` is not restricted to any known set of error domains. Its internal type erasure of the enumeration makes sure that it is a simple (i.e., non-templated) type which can contain arbitrary errors from arbitrary domains.

However, comparison of two `ara::core::ErrorCode` instances only considers the `error code value` and the `error domain` reference; the `support data value` member is not considered for checking equality. This is due to the way `ara::core::ErrorCode` instances are usually compared against a known set of errors for which to check:

```
ErrorCode ec = ...
if (ec == MyEnum::some_error)
  // ...
else if (ec == AnotherEnum::another_error)
  // ...
```

Each of these comparisons will create a temporary `ara::core::ErrorCode` object for the right-hand side of the comparison, and then compare `ec` against that. Such automatically created instances naturally do not contain any meaningful `support data value`.

This frequent creation of temporary `ara::core::ErrorCode` instances is expected to be so fast as to induce no noticable runtime cost. This is usually ensured by `ara::core::ErrorCode` being a *literal type*.

[SWS_CORE_10300]{DRAFT} **ErrorCode type properties** ⌈Class `ara::core::ErrorCode` shall be a *literal type*, as defined in section 3.9-10 [basic.types] of [5, the C++11 standard].⌋*(RS_AP_00130)*

7.2.1.3.2 ErrorDomain

`ara::core::ErrorDomain` is the abstract base class for concrete error domains that are defined within Functional Clusters or even Adaptive Applications. This class is loosely based on `std::error_category`, but differs significantly from it.

An error domain has an associated error code enumeration and an associated base exception type. Both these are usually defined in the same namespace as the `ara::core::ErrorDomain` subclass. For normalized access to these associated types, type aliases with standardized names are defined within the `ara::core::ErrorDomain` subclass. This makes the `ErrorDomain` subclass the root of all data about errors.

Identity of error domains is defined in terms of unique identifiers. AUTOSAR-defined error domains are given standardized identifiers; user-defined error domains are also required to define unique identifiers.

The `ara::core::ErrorDomain` class definition requires this unique identifier to be of unsigned 64 bit integer type (`std::uint64_t`). The range of possible values is large enough to apply UUID-like generation patterns (for `UID-64`) even if typical UUIDs have 128 bits and are thus larger than that. When a new error domain is created (either an AUTOSAR defined or an user defined one) an according `Id` shall be randomly generated, which represents this error domain. The uniqueness and standardization of such an `Id` per error domain is mandatory, since the exchange of information on oc-

cured errors between callee and caller (potentially located at different ECUs) is based on this `Id`.

Given this definition of identity of error domains, it usually makes sense to have only one single instance of each `ara::core::ErrorDomain` subclass. While new instances of these subclasses can be created by calling their constructors, the recommended way to gain access to these subclasses is to call their global accessor functions. For instance, the error domain class `ara::core::FutureErrorDomain` is referenced by calling `ara::core::GetFutureErrorDomain`; within any process space, this will always return a reference to the same global instance of this class.

For error domains that are modelled in ARXML (as ApApplicationErrorDomain), the C++ language binding will create a C++ class for each such ApApplicationErrorDomain. This C++ class will be a subclass of `ara::core::ErrorDomain`, and its name will follow a standard scheme.

`ara::core` has two pre-defined error domains, called `CoreErrorDomain` (containing the set of errors returned by non-`Future`/`Promise` facilities from the `ara::core` Functional Cluster) and `FutureErrorDomain` (containing errors equivalent to those defined by `std::future_errc`).

Application programmers usually do not interact with class `ara::core::ErrorDomain` or its subclasses directly; most access is done via `ara::core::ErrorCode`.

As `ara::core::ErrorDomain` subclasses are expected to be implicitly referred to from within constant (i.e. compile-time) expressions (typically involving `ara::core::ErrorCode`), they are expected to be *literal types*.

[SWS_CORE_10400]{DRAFT} **ErrorDomain type properties** ⌈Class `ara::core::ErrorDomain` and all its subclasses shall be *literal types*, as defined in section 3.9-10 [basic.types] of [5, the C++11 standard].⌋ *(RS_AP_00130)*

7.2.1.3.3 Result

The `ara::core::Result` type follows the `ValueOrError` concept from the C++ proposal p0786 [6]. It either contains a value (of type `ValueType`), or an error (of type `ErrorType`). Both `ValueType` and `ErrorType` are template parameters of `ara::core::Result`, and due to their templated nature, both value and error can be of any type. However, `ErrorType` is defaulted to `ara::core::ErrorCode`, and it is expected that this assignment is kept throughout the Adaptive Platform.

`ara::core::Result` acts as a "wrapper type" that connects the exception-less API approach using `ara::core::ErrorCode` with C++ exceptions. As there is a direct mapping between `ara::core::ErrorCode` and a domain-specific exception type, `ara::core::Result` allows to "transform" its embedded `ara::core::ErrorCode` into the appropriate exception type, by calling `ara::core::Result::ValueOrThrow`.

7.2.1.3.4 Future and Promise

`ara::core::Future` and its companion class `ara::core::Promise` are closely modeled on `std::future` and `std::promise`, but have been adapted to interoperate with `ara::core::Result`. Similar to `ara::core::Result` described in section 7.2.1.3.3, the class `ara::core::Future` either contains a value, or an error (the `Future` first has to be in "ready" state, though). Class `ara::core::Promise` has been adapted in two aspects: `Promise::set_exception` has been removed, and `Promise::SetError` has been introduced in its stead. For `ara::core::Future`, there is a new member function `Future::GetResult` that is similar to `Future::get`, but never throws an exception and returns a `ara::core::Result` instead.

Thus, `ara::core::Future` as return type allows the same dual approach to error handling as `ara::core::Result`, in that it either works exception-based (with `Future::get`), or exception-free (with `Future::GetResult`).

`ara::core::Result` is a type used for returning values or errors from a *synchronous* function call, whereas `ara::core::Future` is a type used for returning values or errors from an *asynchronous* function call.

7.2.1.4 Duality of ErrorCode and exceptions

By using the classes listed above, all APIs of the Adaptive Platform can be used with either an exception-based or an exception-less error handling workflow. However, no API function will ever treat an `Error` by throwing an exception directly; it will always return an error code in the form of a `ara::core::Result` or `ara::core::Future` return value instead. It is then possible for the caller to "transform" the `Error` into an exception, typically via the member function `ara::core::Result::ValueOrThrow`.

When working with a C++ compiler that does not support exceptions at all (or one that has been configured to disable them with an option such as g++'s `-fno-exceptions`), all API functions still show the same behavior. What *does* differ then is that `ara::core::Result::ValueOrThrow` is not defined – this member function is only defined when the compiler does support exceptions.

7.2.1.5 Exception hierarchy

The Adaptive Platform defines a base exception type `ara::core::Exception` for all exceptions defined in the standard. This exception takes a `ara::core::ErrorCode` object as mandatory constructor argument, similar to the way `std::system_error` takes a `std::error_code` argument for construction.

Below this exception base type, there is an additional layer of exception base types, one for each error domain.

For error domains that are modeled in ARXML, the C++ language binding will generate an exception class in addition to the ErrorDomain subclass (which is described in section 7.2.1.3.2). This exception class also conforms to a standard naming scheme: <shortname> of ApApplicationErrorDomain plus "Exception" suffix (this makes it distinguishable from the `ErrorDomain` subclass itself). It is located in the same namespace as the corresponding `ErrorDomain` subclass.

7.2.1.6 Creating new error domains

Any new software module with significant logical separation from all existing modules of the Adaptive Platform should define one or more own error domains.

An error domain consists of:

- an error condition enumeration
- an exception base class
- an `ara::core::ErrorDomain` subclass
- a global `ErrorDomain` subclass accessor function
- a global `MakeErrorCode` function overload

All these are to reside not in the `ara::core` namespace, but in the "target" one.

[SWS_CORE_10999]{DRAFT} **Custom error domain scope** ⌈The `ErrorDomain` subclass and the corresponding enumeration, exception base class, global accessor function, and the `MakeErrorCode` overload shall be defined in the same namespace as the software module for which they are being specified.⌋*(RS_AP_00130)*
Note: This is to help making sure that the C++ ADL mechanism works as expected by other parts of this standard.

An error domain defined in the way specified in this section is suitable to be used for the `ApApplicationErrorDomain` model element.

Throughout this section, the character sequence `<SN>` is a placeholder for the *shortname* of the `ApApplicationErrorDomain`.

7.2.1.6.1 Error condition enumeration

The error condition enumeration describes all known error conditions of the new software module. It should be reasonably fine-grained to allow users to differentiate error conditions that they might want to handle in different ways.

[SWS_CORE_10900]{DRAFT} **Error condition enumeration type** ⌈Each error domain shall define an error condition enum class with the base type `ara::core::ErrorDomain::CodeType` that holds all error conditions of that error domain.⌋*(RS_AP_00130)*

[SWS_CORE_10901]{DRAFT} **Error condition enumeration naming** ⌈Error domain error condition enumerations shall follow the naming scheme `<SN>Errc`, where `<SN>` is the shortname of the `ApApplicationErrorDomain`.⌋*(RS_AP_00130)*

[SWS_CORE_10902]{DRAFT} **Error condition enumeration contents** ⌈Error domain error condition enumerations shall not contain any values that indicate success.⌋ *(RS_AP_00130)*

[SWS_CORE_10903]{DRAFT} **Error condition enumeration numbers** ⌈Error domain error condition enumerations shall keep the number 0 unassigned.⌋*(RS_AP_00130)*

7.2.1.6.2 Exception base class

As a complement to the error condition enumeration, an exception base class for this error domain also needs to be defined. This exception base class is used for the "transformation" of an `ara::core::ErrorCode` object into an exception.

Additional exception types can be defined by the software module, but all these then derive from this base type.

[SWS_CORE_10910]{DRAFT} **ErrorDomain exception base type** ⌈Each error domain shall define an exception base type that is a subclass of `ara::core::Exception`.⌋*(RS_AP_00130)*

[SWS_CORE_10911]{DRAFT} **ErrorDomain exception base type naming** ⌈All error domain exception base types specified by [SWS_CORE_10910] shall follow the naming scheme `<SN>Exception`, where `<SN>` is the shortname of the `ApApplicationErrorDomain`.⌋*(RS_AP_00130)*

[SWS_CORE_10912]{DRAFT} **ErrorDomain exception type hierarchy** ⌈All additional exception types defined by a software module shall have the exception base type specified by [SWS_CORE_10910] as a base class.⌋*(RS_AP_00130)*

7.2.1.6.3 ErrorDomain subclass

Then, a new class is created that derives from `ara::core::ErrorDomain` and overrides all the pure virtual member functions. In addition to that, it also needs to define in its scope a type alias called `Errc` for the error condition enumeration, as well as another type alias called `Exception` for the exception base class for this new error domain.

[SWS_CORE_10930]{DRAFT} **ErrorDomain subclass type** ⌈Each error domain shall define a class type that derives publicly from `ara::core::ErrorDomain`.⌋ *(RS_AP_00130)*

[SWS_CORE_10931]{DRAFT} **ErrorDomain subclass naming** ⌈All subclasses of `ara::core::ErrorDomain` shall follow the naming scheme `<SN>ErrorDomain`, where `<SN>` is the shortname of the `ApApplicationErrorDomain`.⌋*(RS_AP_00130)*

[SWS_CORE_10932]{DRAFT} **ErrorDomain subclass non-extensibility** ⌈All subclasses of `ara::core::ErrorDomain` shall be `final`.⌋*(RS_AP_00130, RS_AP_00140)*

[SWS_CORE_10933]{DRAFT} **ErrorDomain subclass Errc symbol** ⌈All subclasses of `ara::core::ErrorDomain` shall contain in their scope a type alias called `Errc` that refers to the error condition enumeration defined by [SWS_CORE_10900].⌋*(RS_AP_00130)*

[SWS_CORE_10934]{DRAFT} **ErrorDomain subclass Exception symbol** ⌈All subclasses of `ara::core::ErrorDomain` shall contain in their scope a type alias called `Exception` that refers to the exception base type defined by [SWS_CORE_10910].⌋*(RS_AP_00130)*

All `ErrorDomain` subclasses are usable from within constant expressions, see [SWS_CORE_10400]. In particular, this includes that `ErrorDomain` subclasses can be defined as `constexpr` global variables.

In order to further ease working with error domains, all member functions of the `ErrorDomain` subclass are required to be `noexcept`, with the obvious exception of `ErrorDomain::ThrowAsException`.

[SWS_CORE_10950]{DRAFT} **ErrorDomain subclass member function property** ⌈With the exception of `ara::core::ErrorDomain::ThrowAsException`, all public member functions of all `ErrorDomain` subclasses shall be `noexcept`.⌋*(RS_AP_00130)*

The virtual member function `ErrorDomain::Name()` returns the shortname of the `ApApplicationErrorDomain`, mostly for logging purposes.

[SWS_CORE_10951]{DRAFT} **ErrorDomain subclass shortname retrieval** ⌈The return value of an error domain's `Name()` member function shall be equal to the shortname of the `ApApplicationErrorDomain`.⌋*(RS_AP_00130)*

Each error domain has an 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.

[SWS_CORE_10952]{DRAFT} **ErrorDomain subclass unique identifier retrieval** ⌈The return value of an error domain's `Id()` member function shall be a unique identifier that follows the rules defined by [SWS_CORE_00010].⌋*(RS_AP_00130)*

An `ErrorDomain` can "transform" an `ErrorCode` into an exception.

[SWS_CORE_10953]{DRAFT} **Throwing ErrorCodes as exceptions** ⌈The type of an exception thrown by the `ErrorDomain` subclass's implementation of `ErrorDo-`

`main::ThrowAsException` shall derive from that `ErrorDomain` subclass's `Exception` type alias defined by [SWS_CORE_10934].⌋*(RS_AP_00130)*

7.2.1.6.4 Global ErrorDomain subclass accessor function

A global accessor function for the new error domain class is to be defined. For an error domain class `MyErrorDomain`, the accessor function is named `GetMyErrorDomain`. This accessor function returns a reference to a single global instance of that class. This accessor function shall be fully `constexpr`-capable; this in turn implies that the `ErrorDomain` subclass also shall be `constexpr`-constructible (see [SWS_CORE_10400]).

[SWS_CORE_10980]{DRAFT} **ErrorDomain subclass accessor function** ⌈For all subclasses of `ara::core::ErrorDomain`, there shall be a global `constexpr` function that returns a reference-to-const to a singleton instance of it.⌋*(RS_AP_00130)*

[SWS_CORE_10981]{DRAFT} **ErrorDomain subclass accessor function naming** ⌈All `ara::core::ErrorDomain` subclass accessor functions shall follow the naming scheme `Get<SN>ErrorDomain`, where `<SN>` is the shortname of the `ApApplicationErrorDomain`.⌋*(RS_AP_00130)*

[SWS_CORE_10982]{DRAFT} **ErrorDomain subclass accessor function** ⌈All `ara::core::ErrorDomain` subclass accessor functions shall have a return type of `const ErrorDomain&`.⌋*(RS_AP_00130)*

7.2.1.6.5 Global MakeErrorCode overload

And finally, a global factory function `MakeErrorCode` needs to be defined, which is implicitly used by the convenience constructors of class `ara::core::ErrorCode`. This factory function will make use of the global accessor function for the error domain subclass, and call the type-erased constructor of class `ara::core::ErrorCode`.

[SWS_CORE_10990]{DRAFT} **MakeErrorCode overload for new error domains** ⌈For all subclasses of `ara::core::ErrorDomain`, there shall be a `constexpr` overload of the global function `MakeErrorCode` that creates an `ara::core::ErrorCode` instance for a given error condition value within the `ara::core::ErrorDomain` subclass's error condition range.⌋*(RS_AP_00130)*

[SWS_CORE_10991]{DRAFT} **MakeErrorCode overload signature** ⌈All overloads of the global function `MakeErrorCode` shall have the following signature:

```
constexpr ErrorCode MakeErrorCode(<SN>Errc code, ErrorDomain::
    SupportDataType data) noexcept;
```

where `<SN>` is the shortname of the `ApApplicationErrorDomain`.⌋*(RS_AP_00130)*

7.2.1.6.6 C++ pseudo code example

The following C++ pseudo code illustrates how these definitions come together:

```
namespace my
{

enum class <SN>Errc : ara::core::ErrorDomain::CodeType
{
    // ...
};

class <SN>Exception : public ara::core::Exception
{
public:
    <SN>Exception(ara::core::ErrorCode err) noexcept;
};

class <SN>ErrorDomain final : public ara::core::ErrorDomain
{
public:
    using Errc = <SN>Errc;
    using Exception = <SN>Exception;

    constexpr <SN>ErrorDomain() noexcept;

    const char* Name() const noexcept override;
    const char* Message(ara::core::ErrorDomain::CodeType errorCode)
       const noexcept override;
    void ThrowAsException(const ara::core::ErrorCode& errorCode) const
       noexcept(false) override;
};

constexpr const ara::core::ErrorDomain& Get<SN>ErrorDomain() noexcept;

constexpr ara::core::ErrorCode MakeErrorCode(<SN>Errc code, ara::core::
   ErrorDomain::SupportDataType data) noexcept;

}  // namespace my
```

7.2.1.7 AUTOSAR error domains

The full range of unique error domain identifiers is partitioned into a range of AUTOSAR-specified IDs, another range of vendor-defined IDs, and another range of user-defined IDs.

User-defined IDs have their top-bit set to 0 and can use the remaining 63 bits to provide uniqueness. IDs with their top-bit set to 1 are reserved for AUTOSAR and stack vendor use.

[SWS_CORE_00010]{DRAFT} **Error domain identifier** ⌈All error domains shall have a system-wide unique identifier that is represented as a 64-bit unsigned integer value.⌋ *(RS_AP_00130)*

[SWS_CORE_00011]{DRAFT} **AUTOSAR error domain range** ⌈Error domain identifiers where bit #63 is set to 1 and bit #62 is set to 0 are reserved for AUTOSAR-defined error domains.⌋ *(RS_AP_00130)*

[SWS_CORE_00016]{DRAFT} **Vendor-defined error domain range** ⌈Error domain identifiers where the top 32 bits (i.e. bit #63..#32) are equal to 0xc000'0000 are reserved for vendor-specific error domains. Bits #31..#16 hold the vendor's numerical identifier, and bits #15..#0 can be used by each vendor for error domain identifiers.⌋ *(RS_AP_00130)*

[SWS_CORE_00013] The Future error domain ⌈There shall be an error domain `ara::core::FutureErrorDomain` for all errors originating from the interaction of the classes `ara::core::Future` and `ara::core::Promise`. It shall have the shortname `Future` and the identifier 0x8000'0000'0000'0013.⌋ *(RS_AP_00130)*

[SWS_CORE_00014] The Core error domain ⌈There shall be an error domain `ara::core::CoreErrorDomain` for errors originating from non-`Future`/`Promise` facilities of `ara::core`. It shall have the shortname `Core` and the identifier 0x8000'0000'0000'0014.⌋ *(RS_AP_00130)*

7.2.2 Async signal safety

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

[7, 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.

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

7.2.3 Explicit Operation Abortion

If a `Violation` has been detected by the implementation of an API function, [SWS_CORE_00003] mandates to abort this operation immediately. It allows two ways to do this; either by throwing certain kinds of exceptions (if the implementation supports C++ exceptions), or by calling `ara::core::Abort`.

Calling `ara::core::Abort` will result in an `Explicit Operation Abortion`, which usually leads to an `Unexpected Termination` as defined by [8]. This section defines the behavior of this mechanism.

Like `std::abort`, calling `ara::core::Abort` is meant to terminate the current process abnormally and immediately, without performing stack unwinding and without calling destructors of static objects.

[SWS_CORE_12402]{DRAFT} **"Noreturn" property for Abort** ⌈The function `ara::core::Abort` shall not return to its caller.⌋*(RS_AP_00130)*

[SWS_CORE_12403]{DRAFT} **Logging of Explicit Operation Abortion** ⌈Unless logging is deactivated for this Application, calling `ara::core::Abort` shall result in a log message with FATAL severity, which shall contain the string that has been passed to the function as argument, being output via `ara::log`.⌋*(RS_AP_00130)*

[SWS_CORE_12407]{DRAFT} **Thread-safety of Explicit Operation Abortion** ⌈While a call to `ara::core::Abort` is in progress, other calls to this function shall block the calling threads.⌋*(RS_AP_00130)*

`ara::core::Abort` provides a means to add a "hook" into the system, by calling `ara::core::SetAbortHandler`, similar to the way `std::atexit` allows to install a callback for the `std::exit` mechanism. Unlike `std::atexit`, however, it is only possible to set exactly *one* handler with `ara::core::SetAbortHandler`.

[SWS_CORE_12404]{DRAFT} **AbortHandler invocation** ⌈Calling `ara::core::Abort` shall invoke the `AbortHandler` – if it has been set – after the log message as per [SWS_CORE_12403] has been output.⌋*(RS_AP_00130)*

7.2.3.1 AbortHandler

This handler can be installed with `ara::core::SetAbortHandler`. It is invoked in turn when `ara::core::Abort` is called, and it may perform arbitrary operations and then has these four principal choices for its final statements: it can either

- terminate the process, or
- return from the function call, or
- defer function return by entering an infinite loop, or
- perform a non-local goto operation such as `std::longjmp`.

The use of non-local goto operations, including `std::longjmp`, is strongly discouraged and also expressively prohibited by MISRA, the AUTOSAR C++14 Coding Guidelines, and most other coding guidelines as well.

Similarly, deferring function return by entering an infinite loop is discouraged as well; while this still leads to the desired outcome that the *operation* which caused a `Viola-`

tion has been aborted, it will do so at the cost of "defunct'ing" the calling thread and risking the destabilization of the software, which already has encountered a Violation.

An AbortHandler that terminates the process is strongly advised to do so by calling std::abort. This will make sure that the Unexpected Termination is properly seen by Execution Management as an Abnormal Termination as well.

If the AbortHandler returns, or if no AbortHandler is defined at all, then the final action of ara::core::Abort is to call std::abort.

[SWS_CORE_12405]{DRAFT} **Final action without AbortHandler** ⌈If there is no custom ara::core::AbortHandler that has been installed with ara::core::SetAbortHandler, then the implementation of ara::core::Abort shall call std::abort().⌋*(RS_AP_00130)*

[SWS_CORE_12406]{DRAFT} **Final action with a returning AbortHandler** ⌈If there is a custom ara::core::AbortHandler that has been installed with ara::core::SetAbortHandler and it returns, then the implementation of ara::core::Abort shall call std::abort().⌋*(RS_AP_00130)*

7.2.3.2 SIGABRT handler

In addition to the ara::core::AbortHandler, or alternatively to it, the application can also influence this mechanism by installing a signal handler for SIGABRT.

The signal handler for SIGABRT has the same choices of actions as the ara::core::AbortHandler: it can terminate the process, return from the function call, defer function return by entering an infinite loop, or perform a non-local goto operation. The same caveats as for the ara::core::AbortHandler apply here: non-local goto operations and infinite loops should be avoided.

If the SIGABRT handler does not return, it should in general terminate abnormally with SIGABRT. To do this without entering an infinite loop, it should restore the default disposition of SIGABRT with std::signal(SIGABRT, SIG_DFL) and then re-raise SIGABRT with e.g. std::raise(SIGABORT).

This "second step" of influence that the SIGABRT handler provides allows applications that are already handling other synchronous signals such as SIGSEGV or SIGFPE to treat SIGABRT the same way.

7.2.4 Advanced data types

7.2.4.1 AUTOSAR types

7.2.4.1.1 InstanceSpecifier

Instances of `ara::core::InstanceSpecifier` are used to identify service port prototype instances within the AUTOSAR meta-model and are therefore used in the `ara::com` API and elsewhere. A detailed description and background can be found in [9] sections 6.1 (“Instance Identifiers”) and 9.4.4 (“Usage of meta-model identifiers within ara::com based application code”).

`ara::core::InstanceSpecifier` can conceptually be understood to be a wrapper for a string representation of a valid meta-model path. It is designed to be either constructed from a string representation via a factory method `ara::core::InstanceSpecifier::Create`, which provides an exception-free solution, or directly by using the constructor, which might throw an exception if the string representation is invalid.

[SWS_CORE_10200] Valid InstanceSpecifier representations ⌈The content of a valid `InstanceSpecifier` consists of a "/"-separated list of model element names starting from an `Executable` to the respective `PortPrototype` to which the `InstanceSpecifier` shall apply.⌋*(RS_AP_00130)*

[SWS_CORE_10201] Validation of meta-model paths ⌈The construction mechanisms of class `InstanceSpecifier` shall reject meta-model paths that are syntactically invalid according to the syntax rules defined in [SWS_CORE_10200].⌋*(RS_AP_00130)*

[SWS_CORE_10202] Construction of InstanceSpecifier objects ⌈APIs for construction of `InstanceSpecifier` objects shall be available in both potentially-throwing and non-throwing form.⌋*(RS_AP_00130)*

7.2.4.2 Types derived from the base C++ standard

In addition to AUTOSAR-devised data types, which are mentioned in the previous sections, the Adaptive Platform also contains a number of generic data types and helper functions.

Some types are already contained in [5, the C++11 standard]; however, types with almost identical behavior are re-defined within the `ara::core` namespace. The reason for this is that the memory allocation behavior of the `std::` types is often unsuitable for automotive purposes. Thus, the `ara::core` ones define their own memory allocation behavior, and perform some other necessary adaptions as well, including about the throwing of exceptions.

[SWS_CORE_00040]{DRAFT} **Errors originating from C++ standard classes** ⌈For the classes in ara::core specified below in terms of the corresponding classes of the C++ standard, all functions that are specified by [5, the C++11 standard], [10, the

C++17 standard], or [11, the draft C++20 standard] to throw any exceptions, are instead specified to be the cause of a `Violation` when they do so.⌋*(RS_AP_00130)*

Examples for such data types are: Array, Vector, Map, and String.

7.2.4.2.1 Array

This section describes the `ara::core::Array` type that represents a container which encapsulates fixed size arrays.

`ara::core::Array` is an almost-equivalent of `std::array`, and most type properties of `std::array` apply to `ara::core::Array` as well.

These differences to `std::array` are intended:

- `std::array::at` has been omitted (in order to avoid mandatory exception handling)

[SWS_CORE_11200]{DRAFT} **Array base behavior** ⌈`ara::core::Array` and all its member functions and supporting constructs shall behave identical to those of header `<array>` from [4, the C++14 standard], except for the differences specified in this document.⌋*(RS_AP_00130)*

7.2.4.2.2 SteadyClock

7.2.4.2.2.1 Definitions of terms

The C++ `std::chrono` library defines a number of concepts and types for handling time and durations. One of these concepts is that of a "clock" which is able to create snapshots of specific "time points". When talking about clocks and time points, the three qualities *resolution*, *precision*, and *accuracy* are distinguished within this document as follows:

- The `resolution` relates to the smallest increment that can be expressed with the clock's measurement data type.

  For clocks of the POSIX `clock_gettime` API, the `resolution` is implicitly defined as nanoseconds by the API's usage of `struct timespec` with its `timespec::tv_nsec` field.

  For C++ clocks of the `std::chrono` APIs, the `resolution` is variable.

- The `precision` of a clock is the smallest time interval that its timer is able to measure. The `precision` is implementation-defined and depends on the properties and capabilities of the physical machine as well as the operating system.

- The `accuracy` of a clock is the relation between the reported value and the truth.

In addition to that, the epoch is an important property of a clock as well, as it defines the base of the time range that can originate from a clock. Clocks that measure calendar time often use “Unix time”, which is given as number of seconds (without leap seconds) since the “Unix Epoch”, which is 1970-01-01, 00:00:00 UTC.

Clocks that place more emphasis on high precision often do not relate to calendar time at all, but generate timestamps as offsets from something like the power-up time of the system.

7.2.4.2.2.2 Clocks in the Adaptive Platform

The C++ std::chrono library defines a number of standard clocks. Amongst these is std::chrono::steady_clock, which represents a monotonic clock whose time points are strictly increasing with a fixed interval.

However, the C++ standard does not place any requirements on the resolution, precision, and accuracy of this clock. The undefinedness of its resolution can pose some difficulties for application programmers, but these can usually be solved by agreeing on a common – or minimum – resolution. The precision and accuracy are always dependent on the physical properties of the machine and of the operating system.

The Adaptive Platform defines ara::core::SteadyClock as a std::chrono-compatible clock with nanosecond resolution and a std::int64_t datatype. Its precision and accuracy are still implementation-defined and can be given as characteristic values of a concrete platform. Its epoch is the power-up time of the ECU. With these properties, timestamps generated by ara::core::SteadyClock will not overflow until 292 years after its epoch.

It is the standard clock of the Adaptive Platform and should be used for most timekeeping purposes.

The properties of ara::core::SteadyClock imply that a type alias to std::chrono::steady_clock is a conforming implementation of ara::core::SteadyClock, if std::chrono::steady_clock::period is equivalent to std::nano, and std::chrono::steady_clock::rep is a 64-bit signed integer type such as std::int64_t.

[SWS_CORE_11800]{DRAFT} **SteadyClock type requirements** ⌈Class ara::core::SteadyClock shall meet the requirements of *TrivialClock* from [5, the C++11 standard].⌋ *(RS_AP_00130)*

[SWS_CORE_11801]{DRAFT} **Epoch of SteadyClock** ⌈The epoch of ara::core::SteadyClock shall be the system start-up.⌋ *(RS_AP_00130)*

7.2.4.3 Types derived from newer C++ standards

These types have been defined in or proposed for a newer C++ standard, and the Adaptive Platform includes them into the `ara::core` namespace, usually because they are necessary for certain constructs of the Manifest.

Examples for such data types are: Optional, StringView, Span, and Variant.

7.2.4.3.1 ara::core::Byte

`ara::core::Byte` is a type that is able to hold a "byte" of the machine. It is an own type distinct from any other type.

The definitions of this section have been carefully set up in a way to make `std::byte` from [10, the C++17 standard] a conforming implementation, but also allow a class-based implementation with only C++11 means.

Unlike `std::byte` from [10, the C++17 standard], it is implementation-defined whether `ara::core::Byte` can be used for type aliasing without triggering Undefined Behavior.

[SWS_CORE_10100] Type property of ara::core::Byte ⌈The type `ara::core::Byte` shall not be an integral type. In particular, the value `std::is_integral<ara::core::Byte>::value` shall be 0.⌋*(RS_AP_00130)*

[SWS_CORE_10101] Size of type ara::core::Byte ⌈The size (in bytes) of an instance of type `ara::core::Byte` (determined with `sizeof(ara::core::Byte)`) shall be 1.⌋*(RS_AP_00130)*

[SWS_CORE_10102] Value range of type ara::core::Byte ⌈The value of an instance of type `ara::core::Byte` shall be constrained to the range `[0..std::numeric_limits<unsigned char>::max()]`.⌋*(RS_AP_00130)*

[SWS_CORE_10103] Creation of ara::core::Byte instances ⌈An instance of type `ara::core::Byte` shall be creatable from an integral type with brace-initialization syntax. This initialization shall also be possible when called in a constant expression. If the initializer value is outside the value range of type `ara::core::Byte` (see [SWS_CORE_10102]), the behavior is undefined.⌋*(RS_AP_00130)*

[SWS_CORE_10104] Default-constructed ara::core::Byte instances ⌈An instance of type `ara::core::Byte` shall be constructible without giving an initializer value. Such a variable definition shall incur no runtime cost, and the value of the instance shall have indeterminate content.⌋*(RS_AP_00130)*

[SWS_CORE_10105] Destructor of type ara::core::Byte ⌈The destructor of type `ara::core::Byte` shall be trivial.⌋*(RS_AP_00130)*

[SWS_CORE_10106] Implicit conversion from other types ⌈The type `ara::core::Byte` shall not be implicitly convertible from any other type.⌋*(RS_AP_00130)*

[SWS_CORE_10107] Implicit conversion to other types ⌈The type `ara::core::Byte` shall allow no implicit conversion to any other type, including `bool`.⌋*(RS_AP_00130)*

[SWS_CORE_10108] Conversion to unsigned char ⌈The type `ara::core::Byte` shall allow conversion to `unsigned char` with a `static_cast<>` expression. This conversion shall also be possible when called in a constant expression.⌋*(RS_AP_00130)*

[SWS_CORE_10109] Equality comparison for ara::core::Byte ⌈The type `ara::core::Byte` shall be comparable for equality with other instances of type `ara::core::Byte`. This comparison shall also be possible when called in a constant expression.⌋*(RS_AP_00130)*

[SWS_CORE_10110] Non-equality comparison for ara::core::Byte ⌈The type `ara::core::Byte` shall be comparable for non-equality with other instances of type `ara::core::Byte`. This comparison shall also be possible when called in a constant expression.⌋*(RS_AP_00130)*

8 API specification

8.1 C++ language binding

All symbols described in this chapter reside within the namespace `ara::core`. All symbols have `public` visibility unless otherwise noted.

8.1.1 `ErrorDomain` data type

This section describes the `ara::core::ErrorDomain` type that constitutes a base class for error domain implementations.

[SWS_CORE_00110]{DRAFT} ⌈

Kind:	class
Symbol:	ErrorDomain
Scope:	namespace ara::core
Syntax:	`class ErrorDomain {...};`
Header file:	#include "ara/core/error_domain.h"
Description:	Encapsulation of an error domain. An error domain is the controlling entity for ErrorCode's error code values, and defines the mapping of such error code values to textual representations. This class is a literal type, and subclasses are strongly advised to be literal types as well.

⌋(*RS_AP_00130*)

[SWS_CORE_00121]{DRAFT} ⌈

Kind:	type alias
Symbol:	IdType
Scope:	class ara::core::ErrorDomain
Derived from:	std::uint64_t
Syntax:	`using IdType = std::uint64_t;`
Header file:	#include "ara/core/error_domain.h"
Description:	Alias type for a unique ErrorDomain identifier type .

⌋(*RS_AP_00130*)

[SWS_CORE_00122]{DRAFT} ⌈

Kind:	type alias
Symbol:	CodeType
Scope:	class ara::core::ErrorDomain

▽

△

Derived from:	std::int32_t
Syntax:	`using CodeType = std::int32_t;`
Header file:	#include "ara/core/error_domain.h"
Description:	Alias type for a domain-specific error code value .

⌋(*RS_AP_00130*)

[SWS_CORE_00123]{DRAFT} ⌈

Kind:	type alias
Symbol:	SupportDataType
Scope:	class ara::core::ErrorDomain
Derived from:	<implementation-defined>
Syntax:	`using SupportDataType = <implementation-defined>;`
Header file:	#include "ara/core/error_domain.h"
Description:	Alias type for vendor-specific supplementary data .

⌋(*RS_AP_00130*)

[SWS_CORE_00131]{DRAFT} ⌈

Kind:	function
Symbol:	ErrorDomain(const ErrorDomain &)
Scope:	class ara::core::ErrorDomain
Syntax:	`ErrorDomain (const ErrorDomain &)=delete;`
Header file:	#include "ara/core/error_domain.h"
Description:	Copy construction shall be disabled. .

⌋(*RS_AP_00130*)

[SWS_CORE_00132]{DRAFT} ⌈

Kind:	function
Symbol:	ErrorDomain(ErrorDomain &&)
Scope:	class ara::core::ErrorDomain
Syntax:	`ErrorDomain (ErrorDomain &&)=delete;`
Header file:	#include "ara/core/error_domain.h"
Description:	Move construction shall be disabled. .

⌋(*RS_AP_00130*)

[SWS_CORE_00135]{DRAFT} ⌈

Kind:	function	
Symbol:	ErrorDomain(IdType id)	
Scope:	class ara::core::ErrorDomain	
Visibility:	protected	
Syntax:	`explicit constexpr ErrorDomain (IdType id) noexcept;`	
Parameters (in):	id	the unique identifier
Exception Safety:	noexcept	
Header file:	#include "ara/core/error_domain.h"	
Description:	Construct a new instance with the given identifier. Identifiers are expected to be system-wide unique.	

⌋(*RS_AP_00130*)

[SWS_CORE_00136]{DRAFT} ⌈

Kind:	function
Symbol:	~ErrorDomain()
Scope:	class ara::core::ErrorDomain
Visibility:	protected
Syntax:	`~ErrorDomain ()=default;`
Header file:	#include "ara/core/error_domain.h"
Description:	Destructor. This dtor is non-virtual (and trivial) so that this class can be a literal type. While this class has virtual functions, no polymorphic destruction is needed.

⌋(*RS_AP_00130*)

[SWS_CORE_00133]{DRAFT} ⌈

Kind:	function
Symbol:	operator=(const ErrorDomain &)
Scope:	class ara::core::ErrorDomain
Syntax:	`ErrorDomain& operator= (const ErrorDomain &)=delete;`
Header file:	#include "ara/core/error_domain.h"
Description:	Copy assignment shall be disabled. .

⌋(*RS_AP_00130*)

[SWS_CORE_00134]{DRAFT} ⌈

Kind:	function
Symbol:	operator=(ErrorDomain &&)
Scope:	class ara::core::ErrorDomain
Syntax:	`ErrorDomain& operator= (ErrorDomain &&)=delete;`

▽

△

Header file:	#include "ara/core/error_domain.h"
Description:	Move assignment shall be disabled. .

⌋(*RS_AP_00130*)

[SWS_CORE_00137]{DRAFT} ⌈

Kind:	function	
Symbol:	operator==(const ErrorDomain &other)	
Scope:	class ara::core::ErrorDomain	
Syntax:	`constexpr bool operator== (const ErrorDomain &other) const noexcept;`	
Parameters (in):	other	the other instance
Return value:	bool	true if other is equal to *this, false otherwise
Exception Safety:	noexcept	
Header file:	#include "ara/core/error_domain.h"	
Description:	Compare for equality with another ErrorDomain instance. Two ErrorDomain instances compare equal when their identifiers (returned by Id()) are equal.	

⌋(*RS_AP_00130*)

[SWS_CORE_00138]{DRAFT} ⌈

Kind:	function	
Symbol:	operator!=(const ErrorDomain &other)	
Scope:	class ara::core::ErrorDomain	
Syntax:	`constexpr bool operator!= (const ErrorDomain &other) const noexcept;`	
Parameters (in):	other	the other instance
Return value:	bool	true if other is not equal to *this, false otherwise
Exception Safety:	noexcept	
Header file:	#include "ara/core/error_domain.h"	
Description:	Compare for non-equality with another ErrorDomain instance.	

⌋(*RS_AP_00130*)

[SWS_CORE_00151]{DRAFT} ⌈

Kind:	function	
Symbol:	Id()	
Scope:	class ara::core::ErrorDomain	
Syntax:	`constexpr IdType Id () const noexcept;`	
Return value:	IdType	the identifier
Exception Safety:	noexcept	
Header file:	#include "ara/core/error_domain.h"	

▽

△

Description:	Return the unique domain identifier.

⌋(*RS_AP_00130*)

[SWS_CORE_00152]{DRAFT} ⌈

Kind:	function	
Symbol:	Name()	
Scope:	class ara::core::ErrorDomain	
Syntax:	`virtual const char* Name () const noexcept=0;`	
Return value:	const char *	the name as a null-terminated string, never nullptr
Exception Safety:	noexcept	
Header file:	#include "ara/core/error_domain.h"	
Description:	Return the name of this error domain. The returned pointer remains owned by class ErrorDomain and shall not be freed by clients.	

⌋(*RS_AP_00130*)

[SWS_CORE_00153]{DRAFT} ⌈

Kind:	function	
Symbol:	Message(CodeType errorCode)	
Scope:	class ara::core::ErrorDomain	
Syntax:	`virtual const char* Message (CodeType errorCode) const noexcept=0;`	
Parameters (in):	errorCode	the domain-specific error code
Return value:	const char *	the text as a null-terminated string, never nullptr
Exception Safety:	noexcept	
Header file:	#include "ara/core/error_domain.h"	
Description:	Return a textual representation of the given error code. It is a Violation if the errorCode did not originate from this error domain, and thus be subject to SWS_CORE_00003. The returned pointer remains owned by the ErrorDomain subclass and shall not be freed by clients.	

⌋(*RS_AP_00130*)

[SWS_CORE_00154]{DRAFT} ⌈

Kind:	function	
Symbol:	ThrowAsException(const ErrorCode &errorCode)	
Scope:	class ara::core::ErrorDomain	
Syntax:	`virtual void ThrowAsException (const ErrorCode &errorCode) const noexcept(false)=0;`	
Parameters (in):	errorCode	the ErrorCode

▽

△

Return value:	None
Exception Safety:	noexcept(false)
Header file:	#include "ara/core/error_domain.h"
Description:	Throw the given error as exception. This function will determine the appropriate exception type for the given ErrorCode and throw it. The thrown exception will contain the given ErrorCode.

⌋(*RS_AP_00130*)

8.1.2 `ErrorCode` data type

This section describes the `ara::core::ErrorCode` type which holds a domain-specific error.

[SWS_CORE_00501]{DRAFT} ⌈

Kind:	class
Symbol:	ErrorCode
Scope:	namespace ara::core
Syntax:	`class ErrorCode final {...};`
Header file:	#include "ara/core/error_code.h"
Description:	Encapsulation of an error code. An ErrorCode contains a raw error code value and an error domain. The raw error code value is specific to this error domain.

⌋(*RS_AP_00130*, *RS_AP_00140*)

[SWS_CORE_00512]{DRAFT} ⌈

Kind:	function	
Symbol:	ErrorCode(EnumT e, ErrorDomain::SupportDataType data=ErrorDomain::SupportDataType())	
Scope:	class ara::core::ErrorCode	
Syntax:	`template <typename EnumT>` `constexpr ErrorCode (EnumT e, ErrorDomain::SupportDataType data=ErrorDomain::SupportDataType()) noexcept;`	
Template param:	EnumT	an enum type that contains error code values
Parameters (in):	e	a domain-specific error code value
	data	optional vendor-specific supplementary error context data
Exception Safety:	noexcept	
Header file:	#include "ara/core/error_code.h"	
Description:	Construct a new ErrorCode instance with parameters. This constructor does not participate in overload resolution unless EnumT is an enum type.	

⌋(*RS_AP_00130*)

[SWS_CORE_00513]{DRAFT} ⌈

Kind:	function	
Symbol:	ErrorCode(ErrorDomain::CodeType value, const ErrorDomain &domain, ErrorDomain::SupportDataType data=ErrorDomain::SupportDataType())	
Scope:	class ara::core::ErrorCode	
Syntax:	`constexpr ErrorCode (ErrorDomain::CodeType value, const ErrorDomain &domain, ErrorDomain::SupportDataType data=ErrorDomain::SupportDataType()) noexcept;`	
Parameters (in):	value	a domain-specific error code value
	domain	the ErrorDomain associated with value
	data	optional vendor-specific supplementary error context data
Exception Safety:	noexcept	
Header file:	#include "ara/core/error_code.h"	
Description:	Construct a new ErrorCode instance with parameters.	

⌋(*RS_AP_00130*)

[SWS_CORE_00514]{DRAFT} ⌈

Kind:	function	
Symbol:	Value()	
Scope:	class ara::core::ErrorCode	
Syntax:	`constexpr ErrorDomain::CodeType Value () const noexcept;`	
Return value:	ErrorDomain::CodeType	the raw error code value
Exception Safety:	noexcept	
Header file:	#include "ara/core/error_code.h"	
Description:	Return the raw error code value.	

⌋(*RS_AP_00130*)

[SWS_CORE_00515]{DRAFT} ⌈

Kind:	function	
Symbol:	Domain()	
Scope:	class ara::core::ErrorCode	
Syntax:	`constexpr const ErrorDomain& Domain () const noexcept;`	
Return value:	const ErrorDomain &	the ErrorDomain
Exception Safety:	noexcept	
Header file:	#include "ara/core/error_code.h"	
Description:	Return the domain with which this ErrorCode is associated.	

⌋(*RS_AP_00130*)

[SWS_CORE_00516]{DRAFT} ⌈

Kind:	function	
Symbol:	SupportData()	
Scope:	class ara::core::ErrorCode	
Syntax:	`constexpr ErrorDomain::SupportDataType SupportData () const noexcept;`	
Return value:	ErrorDomain::SupportDataType	the supplementary error context data
Exception Safety:	noexcept	
Header file:	#include "ara/core/error_code.h"	
Description:	Return the supplementary error context data. The underlying type and the meaning of the returned value are implementation-defined.	

⌋(*RS_AP_00130*)

[SWS_CORE_00518]{DRAFT} ⌈

Kind:	function	
Symbol:	Message()	
Scope:	class ara::core::ErrorCode	
Syntax:	`StringView Message () const noexcept;`	
Return value:	StringView	the error message text
Exception Safety:	noexcept	
Header file:	#include "ara/core/error_code.h"	
Description:	Return a textual representation of this ErrorCode.	

⌋(*RS_AP_00130*)

[SWS_CORE_00519]{DRAFT} ⌈

Kind:	function
Symbol:	ThrowAsException()
Scope:	class ara::core::ErrorCode
Syntax:	`void ThrowAsException () const;`
Return value:	None
Header file:	#include "ara/core/error_code.h"
Description:	Throw this error as exception. This function will determine the appropriate exception type for this ErrorCode and throw it. The thrown exception will contain this ErrorCode.

⌋(*RS_AP_00130*)

8.1.2.1 `ErrorCode` global operators

[SWS_CORE_00571]{DRAFT} ⌈

Kind:	function	
Symbol:	operator==(const ErrorCode &lhs, const ErrorCode &rhs)	
Scope:	namespace ara::core	
Syntax:	`constexpr bool operator== (const ErrorCode &lhs, const ErrorCode &rhs) noexcept;`	
Parameters (in):	lhs	the left hand side of the comparison
	rhs	the right hand side of the comparison
Return value:	bool	true if the two instances compare equal, false otherwise
Exception Safety:	noexcept	
Header file:	#include "ara/core/error_code.h"	
Description:	Global operator== for ErrorCode. Two ErrorCode instances compare equal if the results of their Value() and Domain() functions are equal. The result of SupportData() is not considered for equality.	

⌋(*RS_AP_00130*)

[SWS_CORE_00572]{DRAFT} ⌈

Kind:	function	
Symbol:	operator!=(const ErrorCode &lhs, const ErrorCode &rhs)	
Scope:	namespace ara::core	
Syntax:	`constexpr bool operator!= (const ErrorCode &lhs, const ErrorCode &rhs) noexcept;`	
Parameters (in):	lhs	the left hand side of the comparison
	rhs	the right hand side of the comparison
Return value:	bool	true if the two instances compare not equal, false otherwise
Exception Safety:	noexcept	
Header file:	#include "ara/core/error_code.h"	
Description:	Global operator!= for ErrorCode. Two ErrorCode instances compare equal if the results of their Value() and Domain() functions are equal. The result of SupportData() is not considered for equality.	

⌋(*RS_AP_00130*)

8.1.3 `Exception` data type

This section describes the `ara::core::Exception` type that constitutes the base type for all exception types defined by the Adaptive Platform.

[SWS_CORE_00601]{DRAFT} ⌈

Kind:	class
Symbol:	Exception
Scope:	namespace ara::core
Base class:	std::exception
Syntax:	`class Exception : public exception {...};`
Header file:	#include "ara/core/exception.h"
Description:	Base type for all AUTOSAR exception types.

⌋(*RS_AP_00130*)

[SWS_CORE_00611]{DRAFT} ⌈

Kind:	function	
Symbol:	Exception(ErrorCode err)	
Scope:	class ara::core::Exception	
Syntax:	`explicit Exception (ErrorCode err) noexcept;`	
Parameters (in):	err	the ErrorCode
Exception Safety:	noexcept	
Header file:	#include "ara/core/exception.h"	
Description:	Construct a new Exception object with a specific ErrorCode.	

⌋(*RS_AP_00130*)

[SWS_CORE_00612]{DRAFT} ⌈

Kind:	function	
Symbol:	what()	
Scope:	class ara::core::Exception	
Syntax:	`const char* what () const noexcept override;`	
Return value:	const char *	a null-terminated string
Exception Safety:	noexcept	
Header file:	#include "ara/core/exception.h"	
Description:	Return the explanatory string. This function overrides the virtual function std::exception::what. All guarantees about the lifetime of the returned pointer that are given for std::exception::what are preserved.	

⌋(*RS_AP_00130*)

[SWS_CORE_00613]{DRAFT} ⌈

Kind:	function
Symbol:	Error()
Scope:	class ara::core::Exception
Syntax:	`const ErrorCode& Error () const noexcept;`

▽

△

Return value:	const ErrorCode &	reference to the embedded ErrorCode
Exception Safety:	noexcept	
Header file:	#include "ara/core/exception.h"	
Description:	Return the embedded ErrorCode that was given to the constructor.	

⌋(*RS_AP_00130*)

[SWS_CORE_00614]{DRAFT} ⌈

Kind:	function	
Symbol:	operator=(Exception const &other)	
Scope:	class ara::core::Exception	
Visibility:	protected	
Syntax:	`Exception& operator= (Exception const &other);`	
Parameters (in):	other	the other instance
Return value:	Exception &	*this
Header file:	#include "ara/core/exception.h"	
Description:	Copy assign from another instance. This function is "protected" in order to prevent some opportunities for accidental slicing.	

⌋(*RS_AP_00130*)

8.1.4 `Result` data type

This section describes the `ara::core::Result<T, E>` type (and its specialization for `T=void`) that contains a value of type T or an error of type E.

[SWS_CORE_00701]{DRAFT} ⌈

Kind:	class	
Symbol:	Result	
Scope:	namespace ara::core	
Syntax:	`template <typename T, typename E = ErrorCode>` `class Result final {...};`	
Template param:	typename T	the type of value
	typename E = ErrorCode	the type of error
Header file:	#include "ara/core/result.h"	
Description:	This class is a type that contains either a value or an error.	

⌋(*RS_AP_00130*)

[SWS_CORE_00711]{DRAFT} ⌈

Kind:	type alias
Symbol:	value_type
Scope:	class ara::core::Result
Derived from:	T
Syntax:	`using value_type = T;`
Header file:	#include "ara/core/result.h"
Description:	Type alias for the type T of values .

⌋(*RS_AP_00130*)

[SWS_CORE_00712]{DRAFT} ⌈

Kind:	type alias
Symbol:	error_type
Scope:	class ara::core::Result
Derived from:	E
Syntax:	`using error_type = E;`
Header file:	#include "ara/core/result.h"
Description:	Type alias for the type E of errors .

⌋(*RS_AP_00130*)

[SWS_CORE_00721]{DRAFT} ⌈

Kind:	function	
Symbol:	Result(const T &t)	
Scope:	class ara::core::Result	
Syntax:	`Result (const T &t);`	
Parameters (in):	t	the value to put into the Result
Header file:	#include "ara/core/result.h"	
Description:	Construct a new Result from the specified value (given as lvalue).	

⌋(*RS_AP_00130*)

[SWS_CORE_00722]{DRAFT} ⌈

Kind:	function	
Symbol:	Result(T &&t)	
Scope:	class ara::core::Result	
Syntax:	`Result (T &&t);`	
Parameters (in):	t	the value to put into the Result
Header file:	#include "ara/core/result.h"	
Description:	Construct a new Result from the specified value (given as rvalue).	

⌋(*RS_AP_00130*)

[SWS_CORE_00723]{DRAFT} ⌈

Kind:	function	
Symbol:	Result(const E &e)	
Scope:	class ara::core::Result	
Syntax:	`explicit Result (const E &e);`	
Parameters (in):	e	the error to put into the Result
Header file:	#include "ara/core/result.h"	
Description:	Construct a new Result from the specified error (given as lvalue).	

⌋(*RS_AP_00130*)

[SWS_CORE_00724]{DRAFT} ⌈

Kind:	function	
Symbol:	Result(E &&e)	
Scope:	class ara::core::Result	
Syntax:	`explicit Result (E &&e);`	
Parameters (in):	e	the error to put into the Result
Header file:	#include "ara/core/result.h"	
Description:	Construct a new Result from the specified error (given as rvalue).	

⌋(*RS_AP_00130*)

[SWS_CORE_00725]{DRAFT} ⌈

Kind:	function	
Symbol:	Result(const Result &other)	
Scope:	class ara::core::Result	
Syntax:	`Result (const Result &other);`	
Parameters (in):	other	the other instance
Header file:	#include "ara/core/result.h"	
Description:	Copy-construct a new Result from another instance.	

⌋(*RS_AP_00130*)

[SWS_CORE_00726]{DRAFT} ⌈

Kind:	function	
Symbol:	Result(Result &&other)	
Scope:	class ara::core::Result	
Syntax:	`Result (Result &&other) noexcept(std::is_nothrow_move_constructible< T >::value &&std::is_nothrow_move_constructible< E >::value);`	
Parameters (in):	other	the other instance
Exception Safety:	conditionally noexcept	
Header file:	#include "ara/core/result.h"	

▽

△

Description:	Move-construct a new Result from another instance.

⌋(*RS_AP_00130*)

[SWS_CORE_00727]{DRAFT} ⌈

Kind:	function
Symbol:	~Result()
Scope:	class ara::core::Result
Syntax:	`~Result ();`
Header file:	#include "ara/core/result.h"
Description:	Destructor. This destructor is trivial if std::is_trivially_destructible<T>::value && std::is_trivially_ destructible<E>::value is true.

⌋(*RS_AP_00130*)

[SWS_CORE_00731]{DRAFT} ⌈

Kind:	function	
Symbol:	FromValue(const T &t)	
Scope:	class ara::core::Result	
Syntax:	`static Result FromValue (const T &t);`	
Parameters (in):	t	the value to put into the Result
Return value:	Result	a Result that contains the value t
Header file:	#include "ara/core/result.h"	
Description:	Build a new Result from the specified value (given as lvalue).	

⌋(*RS_AP_00130*)

[SWS_CORE_00732]{DRAFT} ⌈

Kind:	function	
Symbol:	FromValue(T &&t)	
Scope:	class ara::core::Result	
Syntax:	`static Result FromValue (T &&t);`	
Parameters (in):	t	the value to put into the Result
Return value:	Result	a Result that contains the value t
Header file:	#include "ara/core/result.h"	
Description:	Build a new Result from the specified value (given as rvalue).	

⌋(*RS_AP_00130*)

[SWS_CORE_00733]{DRAFT} ⌈

Kind:	function	
Symbol:	FromValue(Args &&... args)	
Scope:	class ara::core::Result	
Syntax:	`template <typename... Args>` `static Result FromValue (Args &&... args);`	
Template param:	Args...	the types of arguments given to this function
Parameters (in):	args	the arguments used for constructing the value
Return value:	Result	a Result that contains a value
Header file:	#include "ara/core/result.h"	
Description:	Build a new Result from a value that is constructed in-place from the given arguments. This function shall not participate in overload resolution unless: std::is_constructible<T, Args&&...>::value is true, and the first type of the expanded parameter pack is not T, and the first type of the expanded parameter pack is not a specialization of Result	

⌋(*RS_AP_00130*)

[SWS_CORE_00734]{DRAFT} ⌈

Kind:	function	
Symbol:	FromError(const E &e)	
Scope:	class ara::core::Result	
Syntax:	`static Result FromError (const E &e);`	
Parameters (in):	e	the error to put into the Result
Return value:	Result	a Result that contains the error e
Header file:	#include "ara/core/result.h"	
Description:	Build a new Result from the specified error (given as lvalue).	

⌋(*RS_AP_00130*)

[SWS_CORE_00735]{DRAFT} ⌈

Kind:	function	
Symbol:	FromError(E &&e)	
Scope:	class ara::core::Result	
Syntax:	`static Result FromError (E &&e);`	
Parameters (in):	e	the error to put into the Result
Return value:	Result	a Result that contains the error e
Header file:	#include "ara/core/result.h"	
Description:	Build a new Result from the specified error (given as rvalue).	

⌋(*RS_AP_00130*)

[SWS_CORE_00736]{DRAFT} ⌈

Kind:	function	
Symbol:	FromError(Args &&... args)	
Scope:	class ara::core::Result	
Syntax:	`template <typename... Args>` `static Result FromError (Args &&... args);`	
Template param:	Args...	the types of arguments given to this function
Parameters (in):	args	the arguments used for constructing the error
Return value:	Result	a Result that contains an error
Header file:	#include "ara/core/result.h"	
Description:	Build a new Result from an error that is constructed in-place from the given arguments. This function shall not participate in overload resolution unless: std::is_constructible<E, Args&&...>::value is true, and the first type of the expanded parameter pack is not E, and the first type of the expanded parameter pack is not a specialization of Result	

⌋(*RS_AP_00130*)

[SWS_CORE_00741]{DRAFT} ⌈

Kind:	function	
Symbol:	operator=(const Result &other)	
Scope:	class ara::core::Result	
Syntax:	`Result& operator= (const Result &other);`	
Parameters (in):	other	the other instance
Return value:	Result &	*this, containing the contents of other
Header file:	#include "ara/core/result.h"	
Description:	Copy-assign another Result to this instance.	

⌋()

[SWS_CORE_00742]{DRAFT} ⌈

Kind:	function	
Symbol:	operator=(Result &&other)	
Scope:	class ara::core::Result	
Syntax:	`Result& operator= (Result &&other) noexcept(std::is_nothrow_move_constructible< T >::value &&std::is_nothrow_move_assignable< T >::value &&std::is_nothrow_move_constructible< E >::value &&std::is_nothrow_move_assignable< E >::value);`	
Parameters (in):	other	the other instance
Return value:	Result &	*this, containing the contents of other
Exception Safety:	conditionally noexcept	
Header file:	#include "ara/core/result.h"	
Description:	Move-assign another Result to this instance.	

⌋(*RS_AP_00130*)

[SWS_CORE_00743]{DRAFT} ⌈

Kind:	function	
Symbol:	EmplaceValue(Args &&... args)	
Scope:	class ara::core::Result	
Syntax:	`template <typename... Args>` `void EmplaceValue (Args &&... args);`	
Template param:	Args...	the types of arguments given to this function
Parameters (in):	args	the arguments used for constructing the value
Return value:	None	
Header file:	#include "ara/core/result.h"	
Description:	Put a new value into this instance, constructed in-place from the given arguments.	

⌋(*RS_AP_00130*)

[SWS_CORE_00744]{DRAFT} ⌈

Kind:	function	
Symbol:	EmplaceError(Args &&... args)	
Scope:	class ara::core::Result	
Syntax:	`template <typename... Args>` `void EmplaceError (Args &&... args);`	
Template param:	Args...	the types of arguments given to this function
Parameters (in):	args	the arguments used for constructing the error
Return value:	None	
Header file:	#include "ara/core/result.h"	
Description:	Put a new error into this instance, constructed in-place from the given arguments.	

⌋(*RS_AP_00130*)

[SWS_CORE_00745]{DRAFT} ⌈

Kind:	function	
Symbol:	Swap(Result &other)	
Scope:	class ara::core::Result	
Syntax:	`void Swap (Result &other) noexcept(std::is_nothrow_move_constructible< T >::value &&std::is_nothrow_move_assignable< T >::value &&std::is_nothrow_move_constructible< E >::value &&std::is_nothrow_move_assignable< E >::value);`	
Parameters (inout):	other	the other instance
Return value:	None	
Exception Safety:	conditionally noexcept	
Header file:	#include "ara/core/result.h"	
Description:	Exchange the contents of this instance with those of other.	

⌋(*RS_AP_00130*)

[SWS_CORE_00751]{DRAFT} ⌈

Kind:	function	
Symbol:	HasValue()	
Scope:	class ara::core::Result	
Syntax:	`bool HasValue () const noexcept;`	
Return value:	bool	true if *this contains a value, false otherwise
Exception Safety:	noexcept	
Header file:	#include "ara/core/result.h"	
Description:	Check whether *this contains a value.	

⌋(*RS_AP_00130*)

[SWS_CORE_00752]{DRAFT} ⌈

Kind:	function	
Symbol:	operator bool()	
Scope:	class ara::core::Result	
Syntax:	`explicit operator bool () const noexcept;`	
Return value:	bool	true if *this contains a value, false otherwise
Exception Safety:	noexcept	
Header file:	#include "ara/core/result.h"	
Description:	Check whether *this contains a value.	

⌋(*RS_AP_00130*)

[SWS_CORE_00753]{DRAFT} ⌈

Kind:	function	
Symbol:	operator*()	
Scope:	class ara::core::Result	
Syntax:	`const T& operator* () const &;`	
Return value:	const T &	a const_reference to the contained value
Header file:	#include "ara/core/result.h"	
Description:	Access the contained value. This function's behavior is undefined if *this does not contain a value.	

⌋(*RS_AP_00130*)

[SWS_CORE_00759]{DRAFT} ⌈

Kind:	function	
Symbol:	operator*()	
Scope:	class ara::core::Result	
Syntax:	`T&& operator* () &&;`	
Return value:	T &&	an rvalue reference to the contained value

▽

△

Header file:	#include "ara/core/result.h"
Description:	Access the contained value. This function's behavior is undefined if *this does not contain a value.

⌋(*RS_AP_00130*)

[SWS_CORE_00754]{DRAFT} ⌈

Kind:	function	
Symbol:	operator->()	
Scope:	class ara::core::Result	
Syntax:	`const T* operator-> () const;`	
Return value:	const T *	a pointer to the contained value
Header file:	#include "ara/core/result.h"	
Description:	Access the contained value. This function's behavior is undefined if *this does not contain a value.	

⌋(*RS_AP_00130*)

[SWS_CORE_00755]{DRAFT} ⌈

Kind:	function	
Symbol:	Value()	
Scope:	class ara::core::Result	
Syntax:	`const T& Value () const &;`	
Return value:	const T &	a const reference to the contained value
Header file:	#include "ara/core/result.h"	
Description:	Access the contained value. The behavior of this function is undefined if *this does not contain a value.	

⌋(*RS_AP_00130*)

[SWS_CORE_00756]{DRAFT} ⌈

Kind:	function	
Symbol:	Value()	
Scope:	class ara::core::Result	
Syntax:	`T&& Value () &&;`	
Return value:	T &&	an rvalue reference to the contained value
Header file:	#include "ara/core/result.h"	
Description:	Access the contained value. The behavior of this function is undefined if *this does not contain a value.	

⌋(*RS_AP_00130*)

[SWS_CORE_00757]{DRAFT} ⌈

Kind:	function	
Symbol:	Error()	
Scope:	class ara::core::Result	
Syntax:	`const E& Error () const &;`	
Return value:	const E &	a const reference to the contained error
Header file:	#include "ara/core/result.h"	
Description:	Access the contained error. The behavior of this function is undefined if *this does not contain an error.	

⌋(*RS_AP_00130*)

[SWS_CORE_00758]{DRAFT} ⌈

Kind:	function	
Symbol:	Error()	
Scope:	class ara::core::Result	
Syntax:	`E&& Error () &&;`	
Return value:	E &&	an rvalue reference to the contained error
Header file:	#include "ara/core/result.h"	
Description:	Access the contained error. The behavior of this function is undefined if *this does not contain an error.	

⌋(*RS_AP_00130*)

[SWS_CORE_00770]{DRAFT} ⌈

Kind:	function	
Symbol:	Ok()	
Scope:	class ara::core::Result	
Syntax:	`Optional<T> Ok () const &;`	
Return value:	Optional< T >	an Optional with the value, if present
Header file:	#include "ara/core/result.h"	
Description:	Return the contained value as an Optional.	

⌋(*RS_AP_00130*)

[SWS_CORE_00771]{DRAFT} ⌈

Kind:	function	
Symbol:	Ok()	
Scope:	class ara::core::Result	
Syntax:	`Optional<T> Ok () &&;`	
Return value:	Optional< T >	an Optional with the value, if present
Header file:	#include "ara/core/result.h"	

▽

△

Description:	Return the contained value as an Optional.

⌋(*RS_AP_00130*)

[SWS_CORE_00772]{DRAFT} ⌈

Kind:	function	
Symbol:	Err()	
Scope:	class ara::core::Result	
Syntax:	`Optional<E> Err () const &;`	
Return value:	Optional< E >	an Optional with the error, if present
Header file:	#include "ara/core/result.h"	
Description:	Return the contained error as an Optional.	

⌋(*RS_AP_00130*)

[SWS_CORE_00773]{DRAFT} ⌈

Kind:	function	
Symbol:	Err()	
Scope:	class ara::core::Result	
Syntax:	`Optional<E> Err () &&;`	
Return value:	Optional< E >	an Optional with the error, if present
Header file:	#include "ara/core/result.h"	
Description:	Return the contained error as an Optional.	

⌋(*RS_AP_00130*)

[SWS_CORE_00761]{DRAFT} ⌈

Kind:	function	
Symbol:	ValueOr(U &&defaultValue)	
Scope:	class ara::core::Result	
Syntax:	`template <typename U>` `T ValueOr (U &&defaultValue) const &;`	
Template param:	U	the type of defaultValue
Parameters (in):	defaultValue	the value to use if *this does not contain a value
Return value:	T	the value
Header file:	#include "ara/core/result.h"	
Description:	Return the contained value or the given default value. If *this contains a value, it is returned. Otherwise, the specified default value is returned, static_cast'd to T.	

⌋(*RS_AP_00130*)

[SWS_CORE_00762]{DRAFT} ⌈

Kind:	function	
Symbol:	ValueOr(U &&defaultValue)	
Scope:	class ara::core::Result	
Syntax:	`template <typename U>` `T ValueOr (U &&defaultValue) &&;`	
Template param:	U	the type of defaultValue
Parameters (in):	defaultValue	the value to use if *this does not contain a value
Return value:	T	the value
Header file:	#include "ara/core/result.h"	
Description:	Return the contained value or the given default value. If *this contains a value, it is returned. Otherwise, the specified default value is returned, static_cast'd to T.	

⌋(*RS_AP_00130*)

[SWS_CORE_00763]{DRAFT} ⌈

Kind:	function	
Symbol:	ErrorOr(G &&defaultError)	
Scope:	class ara::core::Result	
Syntax:	`template <typename G>` `E ErrorOr (G &&defaultError) const &;`	
Template param:	G	the type of defaultError
Parameters (in):	defaultError	the error to use if *this does not contain an error
Return value:	E	the error
Header file:	#include "ara/core/result.h"	
Description:	Return the contained error or the given default error. If *this contains an error, it is returned. Otherwise, the specified default error is returned, static_cast'd to E.	

⌋(*RS_AP_00130*)

[SWS_CORE_00764]{DRAFT} ⌈

Kind:	function	
Symbol:	ErrorOr(G &&defaultError)	
Scope:	class ara::core::Result	
Syntax:	`template <typename G>` `E ErrorOr (G &&defaultError) &&;`	
Template param:	G	the type of defaultError
Parameters (in):	defaultError	the error to use if *this does not contain an error
Return value:	E	the error
Header file:	#include "ara/core/result.h"	
Description:	Return the contained error or the given default error. If *this contains an error, it is std::move'd into the return value. Otherwise, the specified default error is returned, static_cast'd to E.	

⌋(*RS_AP_00130*)

[SWS_CORE_00765]{DRAFT} ⌈

Kind:	function	
Symbol:	CheckError(G &&error)	
Scope:	class ara::core::Result	
Syntax:	`template <typename G>` `bool CheckError (G &&error) const;`	
Template param:	G	the type of the error argument error
Parameters (in):	error	the error to check
Return value:	bool	true if *this contains an error that is equivalent to the given error, false otherwise
Header file:	#include "ara/core/result.h"	
Description:	Return whether this instance contains the given error. This call compares the argument error, static_cast'd to E, with the return value from Error().	

⌋(*RS_AP_00130*)

[SWS_CORE_00766]{DRAFT} ⌈

Kind:	function	
Symbol:	ValueOrThrow()	
Scope:	class ara::core::Result	
Syntax:	`const T& ValueOrThrow () const &noexcept(false);`	
Return value:	const T &	a const reference to the contained value
Exceptions:	<TYPE>	the exception type associated with the contained error
Header file:	#include "ara/core/result.h"	
Description:	Return the contained value or throw an exception. This function does not participate in overload resolution when the compiler toolchain does not support C++ exceptions.	

⌋(*RS_AP_00130*)

[SWS_CORE_00769]{DRAFT} ⌈

Kind:	function	
Symbol:	ValueOrThrow()	
Scope:	class ara::core::Result	
Syntax:	`T&& ValueOrThrow () &&noexcept(false);`	
Return value:	T &&	an rvalue reference to the contained value
Exceptions:	<TYPE>	the exception type associated with the contained error
Header file:	#include "ara/core/result.h"	
Description:	Return the contained value or throw an exception. This function does not participate in overload resolution when the compiler toolchain does not support C++ exceptions.	

⌋(*RS_AP_00130*)

[SWS_CORE_00767]{DRAFT} ⌈

Kind:	function	
Symbol:	Resolve(F &&f)	
Scope:	class ara::core::Result	
Syntax:	`template <typename F>` `T Resolve (F &&f) const;`	
Template param:	F	the type of the Callable f
Parameters (in):	f	the Callable
Return value:	T	the value
Header file:	#include "ara/core/result.h"	
Description:	Return the contained value or return the result of a function call. If *this contains a value, it is returned. Otherwise, the specified callable is invoked and its return value which is to be compatible to type T is returned from this function. The Callable is expected to be compatible to this interface: T f(const E&);	

⌋(*RS_AP_00130*)

[SWS_CORE_00768]{DRAFT} ⌈

Kind:	function	
Symbol:	Bind(F &&f)	
Scope:	class ara::core::Result	
Syntax:	`template <typename F>` `auto Bind (F &&f) const -> <see below>;`	
Template param:	F	the type of the Callable f
Parameters (in):	f	the Callable
Return value:	<see below>	a new Result instance of the possibly transformed type
Header file:	#include "ara/core/result.h"	
Description:	Apply the given Callable to the value of this instance, and return a new Result with the result of the call. The Callable is expected to be compatible to one of these two interfaces: Result<XXX, E> f(const T&); XXX f(const T&); meaning that the Callable either returns a Result<XXX> or a XXX directly, where XXX can be any type that is suitable for use by class Result. The return type of this function is decltype(f(Value())) for a template argument F that returns a Result type, and it is Result<decltype(f(Value())), E> for a template argument F that does not return a Result type. If this instance does not contain a value, a new Result<XXX, E> is still created and returned, with the original error contents of this instance being copied into the new instance.	

⌋(*RS_AP_00130*)

8.1.4.1 `Result<void, E>` template specialization

This section defines the interface of the `ara::core::Result` template specialization where the type T is “void”.

This specialization omits these member functions that are defined in the generic template:

- `operator->`

- `Bind`

In addition, a number of function overloads collapse to a single, no-argument one.

[SWS_CORE_00801]{DRAFT} ⌈

Kind:	class	
Symbol:	Result< void, E >	
Scope:	namespace ara::core	
Syntax:	`template <typename E>` `class Result< void, E > final {...};`	
Template param:	typename E	the type of error
Header file:	#include "ara/core/result.h"	
Description:	Specialization of class Result for "void" values.	

⌋(*RS_AP_00130*)

[SWS_CORE_00811]{DRAFT} ⌈

Kind:	type alias
Symbol:	value_type
Scope:	class ara::core::Result< void, E >
Derived from:	void
Syntax:	`using value_type = void;`
Header file:	#include "ara/core/result.h"
Description:	Type alias for the type T of values, always "void" for this specialization .

⌋(*RS_AP_00130*)

[SWS_CORE_00812]{DRAFT} ⌈

Kind:	type alias
Symbol:	error_type
Scope:	class ara::core::Result< void, E >
Derived from:	E
Syntax:	`using error_type = E;`
Header file:	#include "ara/core/result.h"
Description:	Type alias for the type E of errors .

⌋(*RS_AP_00130*)

[SWS_CORE_00821]{DRAFT} ⌈

Kind:	function
Symbol:	Result()
Scope:	class ara::core::Result< void, E >
Syntax:	`Result () noexcept;`
Exception Safety:	noexcept
Header file:	#include "ara/core/result.h"
Description:	Construct a new Result with a "void" value.

⌋(*RS_AP_00130*)

[SWS_CORE_00823]{DRAFT} ⌈

Kind:	function	
Symbol:	Result(const E &e)	
Scope:	class ara::core::Result< void, E >	
Syntax:	`explicit Result (const E &e);`	
Parameters (in):	e	the error to put into the Result
Header file:	#include "ara/core/result.h"	
Description:	Construct a new Result from the specified error (given as lvalue).	

⌋(*RS_AP_00130*)

[SWS_CORE_00824]{DRAFT} ⌈

Kind:	function	
Symbol:	Result(E &&e)	
Scope:	class ara::core::Result< void, E >	
Syntax:	`explicit Result (E &&e);`	
Parameters (in):	e	the error to put into the Result
Header file:	#include "ara/core/result.h"	
Description:	Construct a new Result from the specified error (given as rvalue).	

⌋(*RS_AP_00130*)

[SWS_CORE_00825]{DRAFT} ⌈

Kind:	function	
Symbol:	Result(const Result &other)	
Scope:	class ara::core::Result< void, E >	
Syntax:	`Result (const Result &other);`	
Parameters (in):	other	the other instance
Header file:	#include "ara/core/result.h"	
Description:	Copy-construct a new Result from another instance.	

⌋(*RS_AP_00130*)

[SWS_CORE_00826]{DRAFT} ⌈

Kind:	function	
Symbol:	Result(Result &&other)	
Scope:	class ara::core::Result< void, E >	
Syntax:	`Result (Result &&other) noexcept(std::is_nothrow_move_constructible< E >::value);`	
Parameters (in):	other	the other instance
Exception Safety:	conditionally noexcept	
Header file:	#include "ara/core/result.h"	
Description:	Move-construct a new Result from another instance.	

⌋(*RS_AP_00130*)

[SWS_CORE_00827]{DRAFT} ⌈

Kind:	function
Symbol:	~Result()
Scope:	class ara::core::Result< void, E >
Syntax:	`~Result ();`
Header file:	#include "ara/core/result.h"
Description:	Destructor. This destructor is trivial if std::is_trivially_destructible<E>::value is true.

⌋(*RS_AP_00130*)

[SWS_CORE_00831]{DRAFT} ⌈

Kind:	function	
Symbol:	FromValue()	
Scope:	class ara::core::Result< void, E >	
Syntax:	`static Result FromValue () noexcept;`	
Return value:	Result	a Result that contains a "void" value
Exception Safety:	noexcept	
Header file:	#include "ara/core/result.h"	
Description:	Build a new Result with "void" as value.	

⌋(*RS_AP_00130*)

[SWS_CORE_00834]{DRAFT} ⌈

Kind:	function	
Symbol:	FromError(const E &e)	
Scope:	class ara::core::Result< void, E >	
Syntax:	`static Result FromError (const E &e);`	
Parameters (in):	e	the error to put into the Result

▽

△

Return value:	Result	a Result that contains the error e
Header file:	#include "ara/core/result.h"	
Description:	Build a new Result from the specified error (given as lvalue).	

⌋(*RS_AP_00130*)

[SWS_CORE_00835]{DRAFT} ⌈

Kind:	function	
Symbol:	FromError(E &&e)	
Scope:	class ara::core::Result< void, E >	
Syntax:	`static Result FromError (E &&e);`	
Parameters (in):	e	the error to put into the Result
Return value:	Result	a Result that contains the error e
Header file:	#include "ara/core/result.h"	
Description:	Build a new Result from the specified error (given as rvalue).	

⌋(*RS_AP_00130*)

[SWS_CORE_00836]{DRAFT} ⌈

Kind:	function	
Symbol:	FromError(Args &&... args)	
Scope:	class ara::core::Result< void, E >	
Syntax:	`template <typename... Args>` `static Result FromError (Args &&... args);`	
Template param:	Args...	the types of arguments given to this function
Parameters (in):	args	the parameter pack used for constructing the error
Return value:	Result	a Result that contains an error
Header file:	#include "ara/core/result.h"	
Description:	Build a new Result from an error that is constructed in-place from the given arguments. This function shall not participate in overload resolution unless: std::is_constructible<E, Args&&...>::value is true, and the first type of the expanded parameter pack is not E, and the first type of the expanded parameter pack is not a specialization of Result	

⌋(*RS_AP_00130*)

[SWS_CORE_00841]{DRAFT} ⌈

Kind:	function	
Symbol:	operator=(const Result &other)	
Scope:	class ara::core::Result< void, E >	
Syntax:	`Result& operator= (const Result &other);`	
Parameters (in):	other	the other instance

▽

△

Return value:	Result &	*this, containing the contents of other
Header file:	#include "ara/core/result.h"	
Description:	Copy-assign another Result to this instance.	

⌋()

[SWS_CORE_00842]{DRAFT} ⌈

Kind:	function	
Symbol:	operator=(Result &&other)	
Scope:	class ara::core::Result< void, E >	
Syntax:	`Result& operator= (Result &&other) noexcept(std::is_nothrow_move_constructible< E >::value &&std::is_nothrow_move_assignable< E >::value);`	
Parameters (in):	other	the other instance
Return value:	Result &	*this, containing the contents of other
Exception Safety:	conditionally noexcept	
Header file:	#include "ara/core/result.h"	
Description:	Move-assign another Result to this instance.	

⌋(*RS_AP_00130*)

[SWS_CORE_00843]{DRAFT} ⌈

Kind:	function	
Symbol:	EmplaceValue(Args &&... args)	
Scope:	class ara::core::Result< void, E >	
Syntax:	`template <typename... Args>` `void EmplaceValue (Args &&... args) noexcept;`	
Template param:	Args...	the types of arguments given to this function
Parameters (in):	args	the arguments used for constructing the value
Return value:	None	
Exception Safety:	noexcept	
Header file:	#include "ara/core/result.h"	
Description:	Put a new value into this instance, constructed in-place from the given arguments.	

⌋(*RS_AP_00130*)

[SWS_CORE_00844]{DRAFT} ⌈

Kind:	function
Symbol:	EmplaceError(Args &&... args)
Scope:	class ara::core::Result< void, E >
Syntax:	`template <typename... Args>` `void EmplaceError (Args &&... args);`

▽

△

Template param:	Args...	the types of arguments given to this function
Parameters (in):	args	the arguments used for constructing the error
Return value:	None	
Header file:	#include "ara/core/result.h"	
Description:	Put a new error into this instance, constructed in-place from the given arguments.	

⌋(*RS_AP_00130*)

[SWS_CORE_00845]{DRAFT} ⌈

Kind:	function	
Symbol:	Swap(Result &other)	
Scope:	class ara::core::Result< void, E >	
Syntax:	`void Swap (Result &other) noexcept(std::is_nothrow_move_constructible< E >::value &&std::is_nothrow_move_assignable< E >::value);`	
Parameters (inout):	other	the other instance
Return value:	None	
Exception Safety:	conditionally noexcept	
Header file:	#include "ara/core/result.h"	
Description:	Exchange the contents of this instance with those of other.	

⌋(*RS_AP_00130*)

[SWS_CORE_00851]{DRAFT} ⌈

Kind:	function	
Symbol:	HasValue()	
Scope:	class ara::core::Result< void, E >	
Syntax:	`bool HasValue () const noexcept;`	
Return value:	bool	true if *this contains a value, false otherwise
Exception Safety:	noexcept	
Header file:	#include "ara/core/result.h"	
Description:	Check whether *this contains a value.	

⌋(*RS_AP_00130*)

[SWS_CORE_00852]{DRAFT} ⌈

Kind:	function	
Symbol:	operator bool()	
Scope:	class ara::core::Result< void, E >	
Syntax:	`explicit operator bool () const noexcept;`	
Return value:	bool	true if *this contains a value, false otherwise
Exception Safety:	noexcept	

▽

△

Header file:	#include "ara/core/result.h"
Description:	Check whether *this contains a value.

⌋(*RS_AP_00130*)

[SWS_CORE_00853]{DRAFT} ⌈

Kind:	function
Symbol:	operator*()
Scope:	class ara::core::Result< void, E >
Syntax:	`void operator* () const;`
Return value:	None
Header file:	#include "ara/core/result.h"
Description:	Do nothing. This function only exists for helping with generic programming. The behavior of this function is undefined if *this does not contain a value.

⌋(*RS_AP_00130*)

[SWS_CORE_00855]{DRAFT} ⌈

Kind:	function
Symbol:	Value()
Scope:	class ara::core::Result< void, E >
Syntax:	`void Value () const;`
Return value:	None
Header file:	#include "ara/core/result.h"
Description:	Do nothing. This function only exists for helping with generic programming. The behavior of this function is undefined if *this does not contain a value.

⌋(*RS_AP_00130*)

[SWS_CORE_00857]{DRAFT} ⌈

Kind:	function	
Symbol:	Error()	
Scope:	class ara::core::Result< void, E >	
Syntax:	`const E& Error () const &;`	
Return value:	const E &	a const reference to the contained error
Header file:	#include "ara/core/result.h"	
Description:	Access the contained error. The behavior of this function is undefined if *this does not contain an error.	

⌋(*RS_AP_00130*)

[SWS_CORE_00858]{DRAFT} ⌈

Kind:	function	
Symbol:	Error()	
Scope:	class ara::core::Result< void, E >	
Syntax:	`E&& Error () &&;`	
Return value:	E &&	an rvalue reference to the contained error
Header file:	#include "ara/core/result.h"	
Description:	Access the contained error. The behavior of this function is undefined if *this does not contain an error.	

⌋(*RS_AP_00130*)

[SWS_CORE_00868]{DRAFT} ⌈

Kind:	function	
Symbol:	Err()	
Scope:	class ara::core::Result< void, E >	
Syntax:	`Optional<E> Err () const &;`	
Return value:	Optional< E >	an Optional with the error, if present
Header file:	#include "ara/core/result.h"	
Description:	Return the contained error as an Optional.	

⌋(*RS_AP_00130*)

[SWS_CORE_00869]{DRAFT} ⌈

Kind:	function	
Symbol:	Err()	
Scope:	class ara::core::Result< void, E >	
Syntax:	`Optional<E> Err () &&;`	
Return value:	Optional< E >	an Optional with the error, if present
Header file:	#include "ara/core/result.h"	
Description:	Return the contained error as an Optional.	

⌋(*RS_AP_00130*)

[SWS_CORE_00861]{DRAFT} ⌈

Kind:	function	
Symbol:	ValueOr(U &&defaultValue)	
Scope:	class ara::core::Result< void, E >	
Syntax:	`template <typename U>` `void ValueOr (U &&defaultValue) const;`	
Template param:	U	the type of defaultValue
Parameters (in):	defaultValue	the value to use if *this does not contain a value

▽

△

Return value:	None
Header file:	#include "ara/core/result.h"
Description:	Do nothing. This function only exists for helping with generic programming.

⌋(*RS_AP_00130*)

[SWS_CORE_00863]{DRAFT} ⌈

Kind:	function	
Symbol:	ErrorOr(G &&defaultError)	
Scope:	class ara::core::Result< void, E >	
Syntax:	`template <typename G>` `E ErrorOr (G &&defaultError) const &;`	
Template param:	G	the type of defaultError
Parameters (in):	defaultError	the error to use if *this does not contain an error
Return value:	E	the error
Header file:	#include "ara/core/result.h"	
Description:	Return the contained error or the given default error. If *this contains an error, it is returned. Otherwise, the specified default error is returned, static_cast'd to E.	

⌋(*RS_AP_00130*)

[SWS_CORE_00864]{DRAFT} ⌈

Kind:	function	
Symbol:	ErrorOr(G &&defaultError)	
Scope:	class ara::core::Result< void, E >	
Syntax:	`template <typename G>` `E ErrorOr (G &&defaultError) &&;`	
Template param:	G	the type of defaultError
Parameters (in):	defaultError	the error to use if *this does not contain an error
Return value:	E	the error
Header file:	#include "ara/core/result.h"	
Description:	Return the contained error or the given default error. If *this contains an error, it is std::move'd into the return value. Otherwise, the specified default error is returned, static_cast'd to E.	

⌋(*RS_AP_00130*)

[SWS_CORE_00865]{DRAFT} ⌈

Kind:	function	
Symbol:	CheckError(G &&error)	
Scope:	class ara::core::Result< void, E >	
Syntax:	`template <typename G>` `bool CheckError (G &&error) const;`	
Template param:	G	the type of the error argument error
Parameters (in):	error	the error to check
Return value:	bool	true if *this contains an error that is equivalent to the given error, false otherwise
Header file:	#include "ara/core/result.h"	
Description:	Return whether this instance contains the given error. This call compares the argument error, static_cast'd to E, with the return value from Error().	

⌋(*RS_AP_00130*)

[SWS_CORE_00866]{DRAFT} ⌈

Kind:	function	
Symbol:	ValueOrThrow()	
Scope:	class ara::core::Result< void, E >	
Syntax:	`void ValueOrThrow () const noexcept(false);`	
Return value:	None	
Exceptions:	<TYPE>	the exception type associated with the contained error
Header file:	#include "ara/core/result.h"	
Description:	Return the contained value or throw an exception. This function does not participate in overload resolution when the compiler toolchain does not support C++ exceptions.	

⌋(*RS_AP_00130*)

[SWS_CORE_00867]{DRAFT} ⌈

Kind:	function	
Symbol:	Resolve(F &&f)	
Scope:	class ara::core::Result< void, E >	
Syntax:	`template <typename F>` `void Resolve (F &&f) const;`	
Template param:	F	the type of the Callable f
Parameters (in):	f	the Callable
Return value:	None	
Header file:	#include "ara/core/result.h"	
Description:	Do nothing or call a function. If *this contains a value, this function does nothing. Otherwise, the specified callable is invoked. The Callable is expected to be compatible to this interface: void f(const E&); This function only exists for helping with generic programming.	

⌋(*RS_AP_00130*)

[SWS_CORE_00870]{DRAFT} ⌈

Kind:	function	
Symbol:	Bind(F &&f)	
Scope:	class ara::core::Result< void, E >	
Syntax:	`template <typename F>` `auto Bind (F &&f) const -> <see below>;`	
Template param:	F	the type of the Callable f
Parameters (in):	f	the Callable
Return value:	<see below>	a new Result instance of the possibly transformed type
Header file:	#include "ara/core/result.h"	
Description:	Call the given Callable, and return a new Result with the result of the call. The Callable is expected to be compatible to one of these two interfaces: Result<XXX, E> f(); XXX f(); meaning that the Callable either returns a Result<XXX, E> or a XXX directly, where XXX can be any type that is suitable for use by class Result. The return type of this function is decltype(f()) for a template argument F that returns a Result type, and it is Result<decltype(f()), E> for a template argument F that does not return a Result type. If this instance does not contain a value, a new Result<XXX, E> is still created and returned, with the original error contents of this instance being copied into the new instance.	

⌋*(RS_AP_00130)*

8.1.4.2 Global function overloads

[SWS_CORE_00780]{DRAFT} ⌈

Kind:	function	
Symbol:	operator==(const Result< T, E > &lhs, const Result< T, E > &rhs)	
Scope:	namespace ara::core	
Syntax:	`template <typename T, typename E>` `bool operator== (const Result< T, E > &lhs, const Result< T, E > &rhs);`	
Parameters (in):	lhs	the left hand side of the comparison
	rhs	the right hand side of the comparison
Return value:	bool	true if the two instances compare equal, false otherwise
Header file:	#include "ara/core/result.h"	
Description:	Compare two Result instances for equality. A Result that contains a value is unequal to every Result containing an error. A Result is equal to another Result only if both contain the same type, and the value of that type compares equal.	

⌋*(RS_AP_00130)*

[SWS_CORE_00781]{DRAFT} ⌈

Kind:	function	
Symbol:	operator!=(const Result< T, E > &lhs, const Result< T, E > &rhs)	
Scope:	namespace ara::core	
Syntax:	`template <typename T, typename E>` `bool operator!= (const Result< T, E > &lhs, const Result< T, E > &rhs);`	
Parameters (in):	lhs	the left hand side of the comparison
	rhs	the right hand side of the comparison
Return value:	bool	true if the two instances compare unequal, false otherwise
Header file:	#include "ara/core/result.h"	
Description:	Compare two Result instances for inequality. A Result that contains a value is unequal to every Result containing an error. A Result is equal to another Result only if both contain the same type, and the value of that type compares equal.	

⌋(*RS_AP_00130*)

[SWS_CORE_00782]{DRAFT} ⌈

Kind:	function	
Symbol:	operator==(const Result< T, E > &lhs, const T &rhs)	
Scope:	namespace ara::core	
Syntax:	`template <typename T, typename E>` `bool operator== (const Result< T, E > &lhs, const T &rhs);`	
Parameters (in):	lhs	the Result instance
	rhs	the value to compare with
Return value:	bool	true if the Result's value compares equal to the rhs value, false otherwise
Header file:	#include "ara/core/result.h"	
Description:	Compare a Result instance for equality to a value. A Result that contains no value is unequal to every value. A Result is equal to a value only if the Result contains a value of the same type, and the values compare equal.	

⌋(*RS_AP_00130*)

[SWS_CORE_00783]{DRAFT} ⌈

Kind:	function	
Symbol:	operator==(const T &lhs, const Result< T, E > &rhs)	
Scope:	namespace ara::core	
Syntax:	`template <typename T, typename E>` `bool operator== (const T &lhs, const Result< T, E > &rhs);`	
Parameters (in):	lhs	the value to compare with
	rhs	the Result instance
Return value:	bool	true if the Result's value compares equal to the lhs value, false otherwise
Header file:	#include "ara/core/result.h"	

▽

△

Description:	Compare a Result instance for equality to a value. A Result that contains no value is unequal to every value. A Result is equal to a value only if the Result contains a value of the same type, and the values compare equal.

⌋(*RS_AP_00130*)

[SWS_CORE_00784]{DRAFT} ⌈

Kind:	function	
Symbol:	operator!=(const Result< T, E > &lhs, const T &rhs)	
Scope:	namespace ara::core	
Syntax:	`template <typename T, typename E>` `bool operator!= (const Result< T, E > &lhs, const T &rhs);`	
Parameters (in):	lhs	the Result instance
	rhs	the value to compare with
Return value:	bool	true if the Result's value compares unequal to the rhs value, false otherwise
Header file:	#include "ara/core/result.h"	
Description:	Compare a Result instance for inequality to a value. A Result that contains no value is unequal to every value. A Result is equal to a value only if the Result contains a value of the same type, and the values compare equal.	

⌋(*RS_AP_00130*)

[SWS_CORE_00785]{DRAFT} ⌈

Kind:	function	
Symbol:	operator!=(const T &lhs, const Result< T, E > &rhs)	
Scope:	namespace ara::core	
Syntax:	`template <typename T, typename E>` `bool operator!= (const T &lhs, const Result< T, E > &rhs);`	
Parameters (in):	lhs	the value to compare with
	rhs	the Result instance
Return value:	bool	true if the Result's value compares unequal to the lhs value, false otherwise
Header file:	#include "ara/core/result.h"	
Description:	Compare a Result instance for inequality to a value. A Result that contains no value is unequal to every value. A Result is equal to a value only if the Result contains a value of the same type, and the values compare equal.	

⌋(*RS_AP_00130*)

[SWS_CORE_00786]{DRAFT} ⌈

Kind:	function	
Symbol:	operator==(const Result< T, E > &lhs, const E &rhs)	
Scope:	namespace ara::core	
Syntax:	`template <typename T, typename E>` `bool operator== (const Result< T, E > &lhs, const E &rhs);`	
Parameters (in):	lhs	the Result instance
	rhs	the error to compare with
Return value:	bool	true if the Result's error compares equal to the rhs error, false otherwise
Header file:	#include "ara/core/result.h"	
Description:	Compare a Result instance for equality to an error. A Result that contains no error is unequal to every error. A Result is equal to an error only if the Result contains an error of the same type, and the errors compare equal.	

⌋(*RS_AP_00130*)

[SWS_CORE_00787]{DRAFT} ⌈

Kind:	function	
Symbol:	operator==(const E &lhs, const Result< T, E > &rhs)	
Scope:	namespace ara::core	
Syntax:	`template <typename T, typename E>` `bool operator== (const E &lhs, const Result< T, E > &rhs);`	
Parameters (in):	lhs	the error to compare with
	rhs	the Result instance
Return value:	bool	true if the Result's error compares equal to the lhs error, false otherwise
Header file:	#include "ara/core/result.h"	
Description:	Compare a Result instance for equality to an error. A Result that contains no error is unequal to every error. A Result is equal to an error only if the Result contains an error of the same type, and the errors compare equal.	

⌋(*RS_AP_00130*)

[SWS_CORE_00788]{DRAFT} ⌈

Kind:	function	
Symbol:	operator!=(const Result< T, E > &lhs, const E &rhs)	
Scope:	namespace ara::core	
Syntax:	`template <typename T, typename E>` `bool operator!= (const Result< T, E > &lhs, const E &rhs);`	
Parameters (in):	lhs	the Result instance
	rhs	the error to compare with
Return value:	bool	true if the Result's error compares unequal to the rhs error, false otherwise
Header file:	#include "ara/core/result.h"	

▽

△

Description:	Compare a Result instance for inequality to an error. A Result that contains no error is unequal to every error. A Result is equal to an error only if the Result contains an error of the same type, and the errors compare equal.

⌋(*RS_AP_00130*)

[SWS_CORE_00789]{DRAFT} ⌈

Kind:	function	
Symbol:	operator!=(const E &lhs, const Result< T, E > &rhs)	
Scope:	namespace ara::core	
Syntax:	`template <typename T, typename E>` `bool operator!= (const E &lhs, const Result< T, E > &rhs);`	
Parameters (in):	lhs	the error to compare with
	rhs	the Result instance
Return value:	bool	true if the Result's error compares unequal to the lhs error, false otherwise
Header file:	#include "ara/core/result.h"	
Description:	Compare a Result instance for inequality to an error. A Result that contains no error is unequal to every error. A Result is equal to an error only if the Result contains an error of the same type, and the errors compare equal.	

⌋(*RS_AP_00130*)

[SWS_CORE_00796]{DRAFT} ⌈

Kind:	function	
Symbol:	swap(Result< T, E > &lhs, Result< T, E > &rhs)	
Scope:	namespace ara::core	

Syntax:	`template <typename T, typename E>`
	`void swap (Result< T, E > &lhs, Result< T, E > &rhs)`
	`noexcept(noexcept(lhs.Swap(rhs)));`	

Parameters (in):	lhs	one instance
	rhs	another instance
Return value:	None	
Exception Safety:	conditionally noexcept	
Header file:	#include "ara/core/result.h"	
Description:	Swap the contents of the two given arguments.	

⌋(*RS_AP_00130*)

8.1.5 Core Error Domain

This section describes the `ara::core::CoreErrorDomain` type that derives from `ara::core::ErrorDomain` and contains the errors that can originate from within the CORE Functional Cluster.

8.1.5.1 CORE error codes

[SWS_CORE_05200]{DRAFT} ⌈

Kind:	enumeration	
Symbol:	CoreErrc	
Scope:	namespace ara::core	
Underlying type:	ErrorDomain::CodeType	
Syntax:	`enum class CoreErrc : ErrorDomain::CodeType {...};`	
Values:	kInvalidArgument= 22	an invalid argument was passed to a function
	kInvalidMetaModelShortname= 137	given string is not a valid model element shortname
	kInvalidMetaModelPath= 138	missing or invalid path to model element
Header file:	#include "ara/core/core_error_domain.h"	
Description:	An enumeration that defines all errors of the CORE Functional Cluster.	

⌋(*RS_AP_00130*)

8.1.5.2 `CoreException` type

[SWS_CORE_05211]{DRAFT} ⌈

Kind:	class
Symbol:	CoreException
Scope:	namespace ara::core
Base class:	Exception
Syntax:	`class CoreException : public Exception {...};`
Header file:	#include "ara/core/core_error_domain.h"
Description:	Exception type thrown for CORE errors.

⌋(*RS_AP_00130*)

[SWS_CORE_05212]{DRAFT} ⌈

Kind:	function	
Symbol:	CoreException(ErrorCode err)	
Scope:	class ara::core::CoreException	
Syntax:	`explicit CoreException (ErrorCode err) noexcept;`	
Parameters (in):	err	the ErrorCode
Exception Safety:	noexcept	
Header file:	#include "ara/core/core_error_domain.h"	
Description:	Construct a new CoreException from an ErrorCode.	

⌋(*RS_AP_00130*)

8.1.5.3 `CoreErrorDomain` type

[SWS_CORE_05221]{DRAFT} ⌈

Kind:	class
Symbol:	CoreErrorDomain
Scope:	namespace ara::core
Base class:	ErrorDomain
Syntax:	`class CoreErrorDomain final : public ErrorDomain {...};`
Unique ID:	0x8000'0000'0000'0014
Header file:	#include "ara/core/core_error_domain.h"
Description:	An error domain for errors originating from the CORE Functional Cluster .

⌋(*RS_AP_00130*)

[SWS_CORE_05231]{DRAFT} ⌈

Kind:	type alias
Symbol:	Errc
Scope:	class ara::core::CoreErrorDomain
Derived from:	CoreErrc
Syntax:	`using Errc = CoreErrc;`
Header file:	#include "ara/core/core_error_domain.h"
Description:	Alias for the error code value enumeration.

⌋(*RS_AP_00130*)

[SWS_CORE_05232]{DRAFT} ⌈

Kind:	type alias
Symbol:	Exception
Scope:	class ara::core::CoreErrorDomain
Derived from:	CoreException
Syntax:	`using Exception = CoreException;`
Header file:	#include "ara/core/core_error_domain.h"
Description:	Alias for the exception base class.

⌋(*RS_AP_00130*)

[SWS_CORE_05241]{DRAFT} ⌈

Kind:	function
Symbol:	CoreErrorDomain()
Scope:	class ara::core::CoreErrorDomain

▽

△

Syntax:	`constexpr CoreErrorDomain () noexcept;`
Exception Safety:	noexcept
Header file:	#include "ara/core/core_error_domain.h"
Description:	Default constructor.

⌋(*RS_AP_00130*)

[SWS_CORE_05242]{DRAFT} ⌈

Kind:	function	
Symbol:	Name()	
Scope:	class ara::core::CoreErrorDomain	
Syntax:	`const char* Name () const noexcept override;`	
Return value:	const char *	"Core"
Exception Safety:	noexcept	
Header file:	#include "ara/core/core_error_domain.h"	
Description:	Return the "shortname" ApApplicationErrorDomain.SN of this error domain.	

⌋(*RS_AP_00130*)

[SWS_CORE_05243]{DRAFT} ⌈

Kind:	function	
Symbol:	Message(ErrorDomain::CodeType errorCode)	
Scope:	class ara::core::CoreErrorDomain	
Syntax:	`const char* Message (ErrorDomain::CodeType errorCode) const noexcept override;`	
Parameters (in):	errorCode	the error code value
Return value:	const char *	the text message, never nullptr
Exception Safety:	noexcept	
Header file:	#include "ara/core/core_error_domain.h"	
Description:	Translate an error code value into a text message.	

⌋(*RS_AP_00130*)

[SWS_CORE_05244]{DRAFT} ⌈

Kind:	function	
Symbol:	ThrowAsException(const ErrorCode &errorCode)	
Scope:	class ara::core::CoreErrorDomain	
Syntax:	`void ThrowAsException (const ErrorCode &errorCode) const override;`	
Parameters (in):	errorCode	the ErrorCode instance
Return value:	None	
Header file:	#include "ara/core/core_error_domain.h"	

▽

△

Description:	Throw the exception type corresponding to the given ErrorCode.

⌋(*RS_AP_00130*)

8.1.5.4 `GetCoreErrorDomain` accessor function

[SWS_CORE_05280]{DRAFT} ⌈

Kind:	function	
Symbol:	GetCoreErrorDomain()	
Scope:	namespace ara::core	
Syntax:	`constexpr const ErrorDomain& GetCoreErrorDomain () noexcept;`	
Return value:	const ErrorDomain &	the CoreErrorDomain
Exception Safety:	noexcept	
Header file:	#include "ara/core/core_error_domain.h"	
Description:	Return a reference to the global CoreErrorDomain.	

⌋(*RS_AP_00130*)

8.1.5.5 `MakeErrorCode` overload for `CoreErrorDomain`

[SWS_CORE_05290]{DRAFT} ⌈

Kind:	function	
Symbol:	MakeErrorCode(CoreErrc code, ErrorDomain::SupportDataType data)	
Scope:	namespace ara::core	
Syntax:	`constexpr ErrorCode MakeErrorCode (CoreErrc code, ErrorDomain::SupportDataType data) noexcept;`	
Parameters (in):	code	the CoreErrorDomain-specific error code value
	data	optional vendor-specific error data
Return value:	ErrorCode	a new ErrorCode instance
Exception Safety:	noexcept	
Header file:	#include "ara/core/core_error_domain.h"	
Description:	Create a new ErrorCode within CoreErrorDomain. This function is used internally by constructors of ErrorCode. It is usually not used directly by users.	

⌋(*RS_AP_00130*)

8.1.6 `Future` and `Promise` data types

This section describes the `Future` and `Promise` class templates used in `ara::core` to provide and retrieve the results of asynchronous method calls.

Whenever there is a mention of a standard C++11 item (class, class template, enum or function) such as `std::future` or `std::promise`, the implied source material is [5]. Whenever there is a mention of an experimental C++ item such as `std::experimental::future::is_ready`, the implied source material is [12].

Futures are technically referred to as "asynchronous return objects", and Promises are referred to as "asynchronous providers". Their interaction is made possible by a "shared state". The "shared state" concept is described in [5], section 30.6.4. The description also applies to the shared state behind `ara::core::Future` and `ara::core::Promise`, with the following changes:

- The text *", as used by async when policy is `launch::deferred`"* is removed from paragraph 2.
- Paragraph 10, referring to *"`promise::set_value_at_thread_exit`"*, is removed.

Class `ara::core::Future` and `ara::core::Promise` are closely modeled on `std::future` and `std::promise`. Consequently, the behavior of `ara::core::Future` and `ara::core::Promise` is expected to be same as that of `std::future` and `std::promise` from [5, the C++11 standard] and the corresponding `std::experimental::` classes from [12], except for the deviations from the `std::` classes that result from the integration with `ara::core::Result`.

8.1.6.1 `future_errc` enumeration

[SWS_CORE_00400]{DRAFT} ⌈

Kind:	enumeration	
Symbol:	future_errc	
Scope:	namespace ara::core	
Underlying type:	int32_t	
Syntax:	`enum class future_errc : int32_t {...};`	
Values:	broken_promise= 101	the asynchronous task abandoned its shared state
	future_already_retrieved= 102	the contents of the shared state were already accessed
	promise_already_satisfied= 103	attempt to store a value into the shared state twice
	no_state= 104	attempt to access Promise or Future without an associated state
Header file:	#include "ara/core/future_error_domain.h"	

▽

△

Description:	Specifies the types of internal errors that can occur upon calling Future::get or Future::Get Result. These definitions are equivalent to the ones from std::future_errc.

⌋(*RS_AP_00130*)

8.1.6.2 `FutureException` type

[SWS_CORE_00411]{DRAFT} ⌈

Kind:	class
Symbol:	FutureException
Scope:	namespace ara::core
Base class:	Exception
Syntax:	`class FutureException : public Exception {...};`
Header file:	#include "ara/core/future_error_domain.h"
Description:	Exception type thrown by Future and Promise classes.

⌋(*RS_AP_00130*)

[SWS_CORE_00412]{DRAFT} ⌈

Kind:	function	
Symbol:	FutureException(ErrorCode err)	
Scope:	class ara::core::FutureException	
Syntax:	`explicit FutureException (ErrorCode err) noexcept;`	
Parameters (in):	err	the ErrorCode
Exception Safety:	noexcept	
Header file:	#include "ara/core/future_error_domain.h"	
Description:	Construct a new FutureException from an ErrorCode.	

⌋(*RS_AP_00130*)

8.1.6.3 `FutureErrorDomain` type

[SWS_CORE_00421]{DRAFT} ⌈

Kind:	class
Symbol:	FutureErrorDomain
Scope:	namespace ara::core
Base class:	ErrorDomain
Syntax:	`class FutureErrorDomain final : public ErrorDomain {...};`
Unique ID:	0x8000'0000'0000'0013
Header file:	#include "ara/core/future_error_domain.h"
Description:	Error domain for errors originating from classes Future and Promise. .

⌋(*RS_AP_00130*)

[SWS_CORE_00431]{DRAFT} ⌈

Kind:	type alias
Symbol:	Errc
Scope:	class ara::core::FutureErrorDomain
Derived from:	future_errc
Syntax:	`using Errc = future_errc;`
Header file:	#include "ara/core/future_error_domain.h"
Description:	Alias for the error code value enumeration.

⌋(*RS_AP_00130*)

[SWS_CORE_00432]{DRAFT} ⌈

Kind:	type alias
Symbol:	Exception
Scope:	class ara::core::FutureErrorDomain
Derived from:	FutureException
Syntax:	`using Exception = FutureException;`
Header file:	#include "ara/core/future_error_domain.h"
Description:	Alias for the exception base class.

⌋(*RS_AP_00130*)

[SWS_CORE_00441]{DRAFT} ⌈

Kind:	function
Symbol:	FutureErrorDomain()
Scope:	class ara::core::FutureErrorDomain
Syntax:	`constexpr FutureErrorDomain () noexcept;`
Exception Safety:	noexcept
Header file:	#include "ara/core/future_error_domain.h"
Description:	Default constructor.

⌋(*RS_AP_00130*)

[SWS_CORE_00442]{DRAFT} ⌈

Kind:	function	
Symbol:	Name()	
Scope:	class ara::core::FutureErrorDomain	
Syntax:	`const char* Name () const noexcept override;`	
Return value:	const char *	"Future"
Exception Safety:	noexcept	
Header file:	#include "ara/core/future_error_domain.h"	
Description:	Return the "shortname" ApApplicationErrorDomain.SN of this error domain.	

⌋(*RS_AP_00130*)

[SWS_CORE_00443]{DRAFT} ⌈

Kind:	function	
Symbol:	Message(ErrorDomain::CodeType errorCode)	
Scope:	class ara::core::FutureErrorDomain	
Syntax:	`const char* Message (ErrorDomain::CodeType errorCode) const noexcept override;`	
Parameters (in):	errorCode	the error code value
Return value:	const char *	the text message, never nullptr
Exception Safety:	noexcept	
Header file:	#include "ara/core/future_error_domain.h"	
Description:	Translate an error code value into a text message.	

⌋(*RS_AP_00130*)

[SWS_CORE_00444]{DRAFT} ⌈

Kind:	function	
Symbol:	ThrowAsException(const ErrorCode &errorCode)	
Scope:	class ara::core::FutureErrorDomain	
Syntax:	`void ThrowAsException (const ErrorCode &errorCode) const noexcept(false) override;`	
Parameters (in):	errorCode	the ErrorCode instance
Return value:	None	
Exception Safety:	noexcept(false)	
Header file:	#include "ara/core/future_error_domain.h"	
Description:	Throw the exception type corresponding to the given ErrorCode.	

⌋(*RS_AP_00130*)

8.1.6.4 `FutureErrorDomain` accessor function

[SWS_CORE_00480]{DRAFT} ⌈

Kind:	function	
Symbol:	GetFutureErrorDomain()	
Scope:	namespace ara::core	
Syntax:	`constexpr const ErrorDomain& GetFutureErrorDomain () noexcept;`	
Return value:	const ErrorDomain &	reference to the FutureErrorDomain instance
Exception Safety:	noexcept	
Header file:	#include "ara/core/future_error_domain.h"	
Description:	Obtain the reference to the single global FutureErrorDomain instance.	

⌋(*RS_AP_00130*)

8.1.6.5 `MakeErrorCode` overload for `FutureErrorDomain`

[SWS_CORE_00490]{DRAFT} ⌈

Kind:	function	
Symbol:	MakeErrorCode(future_errc code, ErrorDomain::SupportDataType data)	
Scope:	namespace ara::core	
Syntax:	`constexpr ErrorCode MakeErrorCode (future_errc code, ErrorDomain::SupportDataType data) noexcept;`	
Parameters (in):	code	an enumeration value from future_errc
	data	a vendor-defined supplementary value
Return value:	ErrorCode	the new ErrorCode instance
Exception Safety:	noexcept	
Header file:	#include "ara/core/future_error_domain.h"	
Description:	Create a new ErrorCode for FutureErrorDomain with the given support data type.	

⌋(*RS_AP_00130*)

8.1.6.6 `future_status` enumeration

[SWS_CORE_00361]{DRAFT} ⌈

Kind:	enumeration	
Symbol:	future_status	
Scope:	namespace ara::core	
Underlying type:	uint8_t	
Syntax:	`enum class future_status : uint8_t {...};`	
Values:	ready	the shared state is ready
	timeout	the shared state did not become ready before the specified timeout has passed

▽

△

Header file:	#include "ara/core/future.h"
Description:	Specifies the state of a Future as returned by wait_for() and wait_until(). These definitions are equivalent to the ones from std::future_status. However, no item equivalent to std::future_status::deferred is available here. The numerical values of the enum items are implementation-defined.

⌋(*RS_AP_00130*)

8.1.6.7 `Future` data type

[SWS_CORE_00321]{DRAFT} ⌈

Kind:	class	
Symbol:	Future	
Scope:	namespace ara::core	
Syntax:	`template <typename T, typename E = ErrorCode>` `class Future final {...};`	
Template param:	typename T	the type of values
	typename E = ErrorCode	the type of errors
Header file:	#include "ara/core/future.h"	
Description:	Provides ara::core specific Future operations to collect the results of an asynchronous call.	

⌋(*RS_AP_00130*)

[SWS_CORE_00322]{DRAFT} ⌈

Kind:	function
Symbol:	Future()
Scope:	class ara::core::Future
Syntax:	`Future () noexcept=default;`
Exception Safety:	noexcept
Header file:	#include "ara/core/future.h"
Description:	Default constructor. This function shall behave the same as the corresponding std::future function.

⌋(*RS_AP_00130*)

[SWS_CORE_00334]{DRAFT} ⌈

Kind:	function
Symbol:	Future(const Future &)
Scope:	class ara::core::Future

▽

△

Syntax:	`Future (const Future &)=delete;`
Header file:	#include "ara/core/future.h"
Description:	Copy constructor shall be disabled.

⌋(*RS_AP_00130*)

[SWS_CORE_00323]{DRAFT} ⌈

Kind:	function	
Symbol:	Future(Future &&other)	
Scope:	class ara::core::Future	
Syntax:	`Future (Future &&other) noexcept;`	
Parameters (in):	other	the other instance
Exception Safety:	noexcept	
Header file:	#include "ara/core/future.h"	
Description:	Move construct from another instance. This function shall behave the same as the corresponding std::future function.	

⌋(*RS_AP_00130*)

[SWS_CORE_00333]{DRAFT} ⌈

Kind:	function
Symbol:	~Future()
Scope:	class ara::core::Future
Syntax:	`~Future ();`
Header file:	#include "ara/core/future.h"
Description:	Destructor for Future objects. This function shall behave the same as the corresponding std::future function.

⌋(*RS_AP_00130*)

[SWS_CORE_00335]{DRAFT} ⌈

Kind:	function
Symbol:	operator=(const Future &)
Scope:	class ara::core::Future
Syntax:	`Future& operator= (const Future &)=delete;`
Header file:	#include "ara/core/future.h"
Description:	Copy assignment operator shall be disabled.

⌋(*RS_AP_00130*)

[SWS_CORE_00325]{DRAFT} ⌈

Kind:	function	
Symbol:	operator=(Future &&other)	
Scope:	class ara::core::Future	
Syntax:	`Future& operator= (Future &&other) noexcept;`	
Parameters (in):	other	the other instance
Return value:	Future &	*this
Exception Safety:	noexcept	
Header file:	#include "ara/core/future.h"	
Description:	Move assign from another instance. This function shall behave the same as the corresponding std::future function.	

⌋(*RS_AP_00130*)

[SWS_CORE_00326]{DRAFT} ⌈

Kind:	function	
Symbol:	get()	
Scope:	class ara::core::Future	
Syntax:	`T get ();`	
Return value:	T	value of type T
Errors:	Domain:error	the error that has been put into the corresponding Promise via Promise::SetError
Header file:	#include "ara/core/future.h"	
Description:	Get the value. This function shall behave the same as the corresponding std::future function. This function does not participate in overload resolution when the compiler toolchain does not support C++ exceptions.	

⌋(*RS_AP_00130*)

[SWS_CORE_00336]{DRAFT} ⌈

Kind:	function	
Symbol:	GetResult()	
Scope:	class ara::core::Future	
Syntax:	`Result<T, E> GetResult () noexcept;`	
Return value:	Result< T, E >	a Result with either a value or an error
Exception Safety:	noexcept	
Errors:	Domain:error	the error that has been put into the corresponding Promise via Promise::SetError
Header file:	#include "ara/core/future.h"	
Description:	Get the result. Similar to get(), this call blocks until the value or an error is available. However, this call will never throw an exception.	

⌋(*RS_AP_00130*)

[SWS_CORE_00327]{DRAFT} ⌈

Kind:	function	
Symbol:	valid()	
Scope:	class ara::core::Future	
Syntax:	`bool valid () const noexcept;`	
Return value:	bool	true if the Future is usable, false otherwise
Exception Safety:	noexcept	
Header file:	#include "ara/core/future.h"	
Description:	Checks if the Future is valid, i.e. if it has a shared state. This function shall behave the same as the corresponding std::future function.	

⌋(*RS_AP_00130*)

[SWS_CORE_00328]{DRAFT} ⌈

Kind:	function	
Symbol:	wait()	
Scope:	class ara::core::Future	
Syntax:	`void wait () const;`	
Return value:	None	
Header file:	#include "ara/core/future.h"	
Description:	Wait for a value or an error to be available. This function shall behave the same as the corresponding std::future function.	

⌋(*RS_AP_00130*)

[SWS_CORE_00329]{DRAFT} ⌈

Kind:	function	
Symbol:	wait_for(const std::chrono::duration< Rep, Period > &timeoutDuration)	
Scope:	class ara::core::Future	
Syntax:	`template <typename Rep, typename Period>` `future_status wait_for (const std::chrono::duration< Rep, Period > &timeoutDuration) const;`	
Parameters (in):	timeoutDuration	maximal duration to wait for
Return value:	future_status	status that indicates whether the timeout hit or if a value is available
Header file:	#include "ara/core/future.h"	
Description:	Wait for the given period, or until a value or an error is available. This function shall behave the same as the corresponding std::future function.	

⌋(*RS_AP_00130*)

[SWS_CORE_00330]{DRAFT} ⌈

Kind:	function	
Symbol:	wait_until(const std::chrono::time_point< Clock, Duration > &deadline)	
Scope:	class ara::core::Future	
Syntax:	`template <typename Clock, typename Duration>` `future_status wait_until (const std::chrono::time_point< Clock, Duration > &deadline) const;`	
Parameters (in):	deadline	latest point in time to wait
Return value:	future_status	status that indicates whether the time was reached or if a value is available
Header file:	#include "ara/core/future.h"	
Description:	Wait until the given time, or until a value or an error is available. This function shall behave the same as the corresponding std::future function.	

⌋(*RS_AP_00130*)

[SWS_CORE_00331]{DRAFT} ⌈

Kind:	function	
Symbol:	then(F &&func)	
Scope:	class ara::core::Future	
Syntax:	`template <typename F>` `auto then (F &&func) -> Future< <see below> >;`	
Parameters (in):	func	a callable to register
Return value:	Future< <see below> >	a new Future instance for the result of the continuation
Header file:	#include "ara/core/future.h"	
Description:	Register a callable that gets called when the Future becomes ready. When func is called, it is guaranteed that get() and GetResult() will not block. func may be called in the context of this call or in the context of Promise::set_value() or Promise::SetError() or somewhere else. The return type of then depends on the return type of func (aka continuation). Let U be the return type of the continuation (i.e. a type equivalent to std::result_of<std::decay<F>::type(Future<T,E>)>::type). If U is Future<T2,E2> for some types T2, E2, then the return type of then() is Future<T2,E2>. This is known as implicit Future unwrapping. If U is Result<T2,E2> for some types T2, E2, then the return type of then() is Future<T2,E2>. This is known as implicit Result unwrapping. Otherwise it is Future<U,E>.	

⌋(*RS_AP_00130*)

[SWS_CORE_00337]{DRAFT} ⌈

Kind:	function	
Symbol:	then(F &&func, ExecutorT &&executor)	
Scope:	class ara::core::Future	
Syntax:	`template <typename F, typename ExecutorT>` `auto then (F &&func, ExecutorT &&executor) -> Future< <see below> >;`	
Template param:	F	the type of the func argument

▽

△

	ExecutorT	the type of the executor argument
Parameters (in):	func	a callable to register
	executor	the execution context in which to execute the Callable func
Return value:	Future< <see below> >	a new Future instance for the result of the continuation
Header file:	#include "ara/core/future.h"	
Description:	Register a callable that gets called when the Future becomes ready. When func is called, it is guaranteed that get() and GetResult() will not block. func is called in the context of the provided execution context executor. The return type of depends on the return type of func (aka continuation). Let U be the return type of the continuation (i.e. a type equivalent to std::result_of<std::decay<F>::type(Future<T,E>)>::type). If U is Future<T2,E2> for some types T2, E2, then the return type of then() is Future<T2,E2>. This is known as implicit Future unwrapping. If U is Result<T2,E2> for some types T2, E2, then the return type of then() is Future<T2,E2>. This is known as implicit Result unwrapping. Otherwise it is Future<U,E>.	

⌋(*RS_AP_00130*)

[SWS_CORE_00332]{DRAFT} ⌈

Kind:	function	
Symbol:	is_ready()	
Scope:	class ara::core::Future	
Syntax:	`bool is_ready () const;`	
Return value:	bool	true if the Future contains a value or an error, false otherwise
Header file:	#include "ara/core/future.h"	
Description:	Return whether the asynchronous operation has finished. If this function returns true, get(), GetResult() and the wait calls are guaranteed not to block. The behavior of this function is undefined if valid() returns false.	

⌋(*RS_AP_00130*)

8.1.6.7.1 `Future<void, E>` template specialization

This section defines the interface of the `ara::core::Future<T,E>` template specialization where the type T is `void`.

[SWS_CORE_06221]{DRAFT} ⌈

Kind:	class
Symbol:	Future< void, E >
Scope:	namespace ara::core

▽

△

Syntax:	`template <typename E>` `class Future< void, E > final {...};`	
Template param:	typename E	the type of error
Header file:	#include "ara/core/future.h"	
Description:	Specialization of class Future for "void" values.	

⌋(*RS_AP_00130*)

[SWS_CORE_06222]{DRAFT} ⌈

Kind:	function
Symbol:	Future()
Scope:	class ara::core::Future< void, E >
Syntax:	`Future () noexcept;`
Exception Safety:	noexcept
Header file:	#include "ara/core/future.h"
Description:	Default constructor. This function shall behave the same as the corresponding std::future function.

⌋(*RS_AP_00130*)

[SWS_CORE_06234]{DRAFT} ⌈

Kind:	function
Symbol:	Future(const Future &other)
Scope:	class ara::core::Future< void, E >
Syntax:	`Future (const Future &other)=delete;`
Header file:	#include "ara/core/future.h"
Description:	Copy constructor shall be disabled.

⌋(*RS_AP_00130*)

[SWS_CORE_06223]{DRAFT} ⌈

Kind:	function	
Symbol:	Future(Future &&other)	
Scope:	class ara::core::Future< void, E >	
Syntax:	`Future (Future &&other) noexcept;`	
Parameters (in):	other	the other instance
Exception Safety:	noexcept	
Header file:	#include "ara/core/future.h"	
Description:	Move construct from another instance. This function shall behave the same as the corresponding std::future function.	

⌋(*RS_AP_00130*)

[SWS_CORE_06233]{DRAFT} ⌈

Kind:	function
Symbol:	~Future()
Scope:	class ara::core::Future< void, E >
Syntax:	`~Future ();`
Header file:	#include "ara/core/future.h"
Description:	Destructor for Future objects. This function shall behave the same as the corresponding std::future function.

⌋(*RS_AP_00130*)

[SWS_CORE_06235]{DRAFT} ⌈

Kind:	function
Symbol:	operator=(const Future &other)
Scope:	class ara::core::Future< void, E >
Syntax:	`Future& operator= (const Future &other)=delete;`
Header file:	#include "ara/core/future.h"
Description:	Copy assignment operator shall be disabled.

⌋(*RS_AP_00130*)

[SWS_CORE_06225]{DRAFT} ⌈

Kind:	function	
Symbol:	operator=(Future &&other)	
Scope:	class ara::core::Future< void, E >	
Syntax:	`Future& operator= (Future &&other) noexcept;`	
Parameters (in):	other	the other instance
Return value:	Future &	*this
Exception Safety:	noexcept	
Header file:	#include "ara/core/future.h"	
Description:	Move assign from another instance. This function shall behave the same as the corresponding std::future function.	

⌋(*RS_AP_00130*)

[SWS_CORE_06226]{DRAFT} ⌈

Kind:	function	
Symbol:	get()	
Scope:	class ara::core::Future< void, E >	
Syntax:	`void get ();`	
Return value:	None	
Errors:	Domain:error	the error that has been put into the corresponding Promise via Promise::SetError

▽

△

Header file:	#include "ara/core/future.h"
Description:	Get the value. This function shall behave the same as the corresponding std::future function. This function does not participate in overload resolution when the compiler toolchain does not support C++ exceptions.

⌋(*RS_AP_00130*)

[SWS_CORE_06236]{DRAFT} ⌈

Kind:	function	
Symbol:	GetResult()	
Scope:	class ara::core::Future< void, E >	
Syntax:	`Result<void, E> GetResult () noexcept;`	
Return value:	Result< void, E >	a Result with either a value or an error
Exception Safety:	noexcept	
Errors:	Domain:error	the error that has been put into the corresponding Promise via Promise::SetError
Header file:	#include "ara/core/future.h"	
Description:	Get the result. Similar to get(), this call blocks until the value or an error is available. However, this call will never throw an exception.	

⌋(*RS_AP_00130*)

[SWS_CORE_06227]{DRAFT} ⌈

Kind:	function	
Symbol:	valid()	
Scope:	class ara::core::Future< void, E >	
Syntax:	`bool valid () const noexcept;`	
Return value:	bool	true if the Future is usable, false otherwise
Exception Safety:	noexcept	
Header file:	#include "ara/core/future.h"	
Description:	Checks if the Future is valid, i.e. if it has a shared state. This function shall behave the same as the corresponding std::future function.	

⌋(*RS_AP_00130*)

[SWS_CORE_06228]{DRAFT} ⌈

Kind:	function
Symbol:	wait()
Scope:	class ara::core::Future< void, E >

▽

△

Syntax:	`void wait () const;`
Return value:	None
Header file:	#include "ara/core/future.h"
Description:	Wait for a value or an error to be available. This function shall behave the same as the corresponding std::future function.

⌋(*RS_AP_00130*)

[SWS_CORE_06229]{DRAFT} ⌈

Kind:	function	
Symbol:	wait_for(const std::chrono::duration< Rep, Period > &timeoutDuration)	
Scope:	class ara::core::Future< void, E >	
Syntax:	`template <typename Rep, typename Period>` `future_status wait_for (const std::chrono::duration< Rep, Period > &timeoutDuration) const;`	
Parameters (in):	timeoutDuration	maximal duration to wait for
Return value:	future_status	status that indicates whether the timeout hit or if a value is available
Header file:	#include "ara/core/future.h"	
Description:	Wait for the given period, or until a value or an error is available. This function shall behave the same as the corresponding std::future function.	

⌋(*RS_AP_00130*)

[SWS_CORE_06230]{DRAFT} ⌈

Kind:	function	
Symbol:	wait_until(const std::chrono::time_point< Clock, Duration > &deadline)	
Scope:	class ara::core::Future< void, E >	
Syntax:	`template <typename Clock, typename Duration>` `future_status wait_until (const std::chrono::time_point< Clock, Duration > &deadline) const;`	
Parameters (in):	deadline	latest point in time to wait
Return value:	future_status	status that indicates whether the time was reached or if a value is available
Header file:	#include "ara/core/future.h"	
Description:	Wait until the given time, or until a value or an error is available. This function shall behave the same as the corresponding std::future function.	

⌋(*RS_AP_00130*)

[SWS_CORE_06231]{DRAFT} ⌈

Kind:	function	
Symbol:	then(F &&func)	
Scope:	class ara::core::Future< void, E >	
Syntax:	`template <typename F>` `auto then (F &&func) -> Future< <see below> >;`	
Parameters (in):	func	a callable to register
Return value:	Future< <see below> >	a new Future instance for the result of the continuation
Header file:	#include "ara/core/future.h"	
Description:	Register a callable that gets called when the Future becomes ready. When func is called, it is guaranteed that get() and GetResult() will not block. func may be called in the context of this call or in the context of Promise::set_value() or Promise::SetError() or somewhere else. The return type of then depends on the return type of func (aka continuation). Let U be the return type of the continuation (i.e. a type equivalent to std::result_of<std::decay<F>::type(Future<T,E>)>::type). If U is Future<T2,E2> for some types T2, E2, then the return type of then() is Future<T2,E2>. This is known as implicit Future unwrapping. If U is Result<T2,E2> for some types T2, E2, then the return type of then() is Future<T2,E2>. This is known as implicit Result unwrapping. Otherwise it is Future<U,E>.	

⌋(*RS_AP_00130*)

[SWS_CORE_06237]{DRAFT} ⌈

Kind:	function	
Symbol:	then(F &&func, ExecutorT &&executor)	
Scope:	class ara::core::Future< void, E >	
Syntax:	`template <typename F, typename ExecutorT>` `auto then (F &&func, ExecutorT &&executor) -> Future< <see below> >;`	
Template param:	F	the type of the func argument
	ExecutorT	the type of the executor argument
Parameters (in):	func	a callable to register
	executor	the execution context in which to execute the Callable func
Return value:	Future< <see below> >	a new Future instance for the result of the continuation
Header file:	#include "ara/core/future.h"	
Description:	Register a callable that gets called when the Future becomes ready. When func is called, it is guaranteed that get() and GetResult() will not block. func is called in the context of the provided execution context executor. The return type of depends on the return type of func (aka continuation). Let U be the return type of the continuation (i.e. a type equivalent to std::result_of<std::decay<F>::type(Future<T,E>)>::type). If U is Future<T2,E2> for some types T2, E2, then the return type of then() is Future<T2,E2>. This is known as implicit Future unwrapping. If U is Result<T2,E2> for some types T2, E2, then the return type of then() is Future<T2,E2>. This is known as implicit Result unwrapping. Otherwise it is Future<U,E>.	

⌋(*RS_AP_00130*)

[SWS_CORE_06232]{DRAFT} ⌈

Kind:	function	
Symbol:	is_ready()	
Scope:	class ara::core::Future< void, E >	
Syntax:	`bool is_ready () const;`	
Return value:	bool	true if the Future contains a value or an error, false otherwise
Header file:	#include "ara/core/future.h"	
Description:	Return whether the asynchronous operation has finished. If this function returns true, get(), GetResult() and the wait calls are guaranteed not to block. The behavior of this function is undefined if valid() returns false.	

⌋(*RS_AP_00130*)

8.1.6.8 `Promise` data type

[SWS_CORE_00340]{DRAFT} ⌈

Kind:	class	
Symbol:	Promise	
Scope:	namespace ara::core	
Syntax:	`template <typename T, typename E = ErrorCode>` `class Promise {...};`	
Template param:	typename T	the type of value
	typename E = ErrorCode	the type of error
Header file:	#include "ara/core/promise.h"	
Description:	ara::core specific variant of std::promise class	

⌋(*RS_AP_00130*)

[SWS_CORE_00341]{DRAFT} ⌈

Kind:	function
Symbol:	Promise()
Scope:	class ara::core::Promise
Syntax:	`Promise ();`
Header file:	#include "ara/core/promise.h"
Description:	Default constructor. This function shall behave the same as the corresponding std::promise function.

⌋(*RS_AP_00130*)

[SWS_CORE_00342]{DRAFT} ⌈

Kind:	function	
Symbol:	Promise(Promise &&other)	
Scope:	class ara::core::Promise	
Syntax:	`Promise (Promise &&other) noexcept;`	
Parameters (in):	other	the other instance
Exception Safety:	noexcept	
Header file:	#include "ara/core/promise.h"	
Description:	Move constructor. This function shall behave the same as the corresponding std::promise function.	

⌋(*RS_AP_00130*)

[SWS_CORE_00350]{DRAFT} ⌈

Kind:	function
Symbol:	Promise(const Promise &)
Scope:	class ara::core::Promise
Syntax:	`Promise (const Promise &)=delete;`
Header file:	#include "ara/core/promise.h"
Description:	Copy constructor shall be disabled.

⌋(*RS_AP_00130*)

[SWS_CORE_00349]{DRAFT} ⌈

Kind:	function
Symbol:	~Promise()
Scope:	class ara::core::Promise
Syntax:	`~Promise ();`
Header file:	#include "ara/core/promise.h"
Description:	Destructor for Promise objects. This function shall behave the same as the corresponding std::promise function.

⌋(*RS_AP_00130*)

[SWS_CORE_00343]{DRAFT} ⌈

Kind:	function	
Symbol:	operator=(Promise &&other)	
Scope:	class ara::core::Promise	
Syntax:	`Promise& operator= (Promise &&other) noexcept;`	
Parameters (in):	other	the other instance
Return value:	Promise &	*this

▽

△

Exception Safety:	noexcept
Header file:	#include "ara/core/promise.h"
Description:	Move assignment. This function shall behave the same as the corresponding std::promise function.

⌋(*RS_AP_00130*)

[SWS_CORE_00351]{DRAFT} ⌈

Kind:	function
Symbol:	operator=(const Promise &)
Scope:	class ara::core::Promise
Syntax:	`Promise& operator= (const Promise &)=delete;`
Header file:	#include "ara/core/promise.h"
Description:	Copy assignment operator shall be disabled.

⌋(*RS_AP_00130*)

[SWS_CORE_00352]{DRAFT} ⌈

Kind:	function	
Symbol:	swap(Promise &other)	
Scope:	class ara::core::Promise	
Syntax:	`void swap (Promise &other) noexcept;`	
Parameters (in):	other	the other instance
Return value:	None	
Exception Safety:	noexcept	
Header file:	#include "ara/core/promise.h"	
Description:	Swap the contents of this instance with another one's. This function shall behave the same as the corresponding std::promise function.	

⌋(*RS_AP_00130*)

[SWS_CORE_00344]{DRAFT} ⌈

Kind:	function	
Symbol:	get_future()	
Scope:	class ara::core::Promise	
Syntax:	`Future<T, E> get_future ();`	
Return value:	Future< T, E >	a Future
Header file:	#include "ara/core/promise.h"	
Description:	Return the associated Future. The returned Future is set as soon as this Promise receives the result or an error. This method must only be called once as it is not allowed to have multiple Futures per Promise.	

⌋(*RS_AP_00130*)

[SWS_CORE_00345]{DRAFT} ⌈

Kind:	function	
Symbol:	set_value(const T &value)	
Scope:	class ara::core::Promise	
Syntax:	`void set_value (const T &value);`	
Parameters (in):	value	the value to store
Return value:	None	
Header file:	#include "ara/core/promise.h"	
Description:	Copy a value into the shared state and make the state ready. This function shall behave the same as the corresponding std::promise function.	

⌋(*RS_AP_00130*)

[SWS_CORE_00346]{DRAFT} ⌈

Kind:	function	
Symbol:	set_value(T &&value)	
Scope:	class ara::core::Promise	
Syntax:	`void set_value (T &&value);`	
Parameters (in):	value	the value to store
Return value:	None	
Header file:	#include "ara/core/promise.h"	
Description:	Move a value into the shared state and make the state ready. This function shall behave the same as the corresponding std::promise function.	

⌋(*RS_AP_00130*)

[SWS_CORE_00353]{DRAFT} ⌈

Kind:	function	
Symbol:	SetError(E &&error)	
Scope:	class ara::core::Promise	
Syntax:	`void SetError (E &&error);`	
Parameters (in):	error	the error to store
Return value:	None	
Header file:	#include "ara/core/promise.h"	
Description:	Move an error into the shared state and make the state ready.	

⌋(*RS_AP_00130*)

[SWS_CORE_00354]{DRAFT} ⌈

Kind:	function	
Symbol:	SetError(const E &error)	
Scope:	class ara::core::Promise	
Syntax:	`void SetError (const E &error);`	
Parameters (in):	error	the error to store
Return value:	None	
Header file:	#include "ara/core/promise.h"	
Description:	Copy an error into the shared state and make the state ready.	

⌋(*RS_AP_00130*)

[SWS_CORE_00355]{DRAFT} ⌈

Kind:	function	
Symbol:	SetResult(const Result< T, E > &result)	
Scope:	class ara::core::Promise	
Syntax:	`void SetResult (const Result< T, E > &result);`	
Parameters (in):	result	the result to store
Return value:	None	
Header file:	#include "ara/core/promise.h"	
Description:	Copy a Result into the shared state and make the state ready.	

⌋(*RS_AP_00130*)

[SWS_CORE_00356]{DRAFT} ⌈

Kind:	function	
Symbol:	SetResult(Result< T, E > &&result)	
Scope:	class ara::core::Promise	
Syntax:	`void SetResult (Result< T, E > &&result);`	
Parameters (in):	result	the result to store
Return value:	None	
Header file:	#include "ara/core/promise.h"	
Description:	Move a Result into the shared state and make the state ready.	

⌋(*RS_AP_00130*)

8.1.6.8.1 `Promise<void, E>` template specialization

This section defines the interface of the `ara::core::Promise<T,E>` template specialization where the type T is `void`.

[SWS_CORE_06340]{DRAFT} ⌈

Kind:	class	
Symbol:	Promise< void, E >	
Scope:	namespace ara::core	
Syntax:	`template <typename E>` `class Promise< void, E > final {...};`	
Template param:	typename E	the type of error
Header file:	#include "ara/core/promise.h"	
Description:	Specialization of class Promise for "void" values.	

⌋(*RS_AP_00130*)

[SWS_CORE_06341]{DRAFT} ⌈

Kind:	function	
Symbol:	Promise()	
Scope:	class ara::core::Promise< void, E >	
Syntax:	`Promise ();`	
Header file:	#include "ara/core/promise.h"	
Description:	Default constructor. This function shall behave the same as the corresponding std::promise function.	

⌋(*RS_AP_00130*)

[SWS_CORE_06342]{DRAFT} ⌈

Kind:	function	
Symbol:	Promise(Promise &&other)	
Scope:	class ara::core::Promise< void, E >	
Syntax:	`Promise (Promise &&other) noexcept;`	
Parameters (in):	other	the other instance
Exception Safety:	noexcept	
Header file:	#include "ara/core/promise.h"	
Description:	Move constructor. This function shall behave the same as the corresponding std::promise function.	

⌋(*RS_AP_00130*)

[SWS_CORE_06350]{DRAFT} ⌈

Kind:	function
Symbol:	Promise(const Promise &)
Scope:	class ara::core::Promise< void, E >
Syntax:	`Promise (const Promise &)=delete;`
Header file:	#include "ara/core/promise.h"

▽

△

Description:	Copy constructor shall be disabled.

⌋(*RS_AP_00130*)

[SWS_CORE_06349]{DRAFT} ⌈

Kind:	function
Symbol:	~Promise()
Scope:	class ara::core::Promise< void, E >
Syntax:	`~Promise ();`
Header file:	#include "ara/core/promise.h"
Description:	Destructor for Promise objects. This function shall behave the same as the corresponding std::promise function.

⌋(*RS_AP_00130*)

[SWS_CORE_06343]{DRAFT} ⌈

Kind:	function	
Symbol:	operator=(Promise &&other)	
Scope:	class ara::core::Promise< void, E >	
Syntax:	`Promise& operator= (Promise &&other) noexcept;`	
Parameters (in):	other	the other instance
Return value:	Promise &	*this
Exception Safety:	noexcept	
Header file:	#include "ara/core/promise.h"	
Description:	Move assignment. This function shall behave the same as the corresponding std::promise function.	

⌋(*RS_AP_00130*)

[SWS_CORE_06351]{DRAFT} ⌈

Kind:	function
Symbol:	operator=(const Promise &)
Scope:	class ara::core::Promise< void, E >
Syntax:	`Promise& operator= (const Promise &)=delete;`
Header file:	#include "ara/core/promise.h"
Description:	Copy assignment operator shall be disabled.

⌋(*RS_AP_00130*)

[SWS_CORE_06352]{DRAFT} ⌈

Kind:	function	
Symbol:	swap(Promise &other)	
Scope:	class ara::core::Promise< void, E >	
Syntax:	`void swap (Promise &other) noexcept;`	
Parameters (in):	other	the other instance
Return value:	None	
Exception Safety:	noexcept	
Header file:	#include "ara/core/promise.h"	
Description:	Swap the contents of this instance with another one's. This function shall behave the same as the corresponding std::promise function.	

⌋(*RS_AP_00130*)

[SWS_CORE_06344]{DRAFT} ⌈

Kind:	function	
Symbol:	get_future()	
Scope:	class ara::core::Promise< void, E >	
Syntax:	`Future<void, E> get_future ();`	
Return value:	Future< void, E >	a Future
Header file:	#include "ara/core/promise.h"	
Description:	Return the associated Future. The returned Future is set as soon as this Promise receives the result or an error. This method must only be called once as it is not allowed to have multiple Futures per Promise.	

⌋(*RS_AP_00130*)

[SWS_CORE_06345]{DRAFT} ⌈

Kind:	function
Symbol:	set_value()
Scope:	class ara::core::Promise< void, E >
Syntax:	`void set_value ();`
Return value:	None
Header file:	#include "ara/core/promise.h"
Description:	Make the shared state ready.

⌋(*RS_AP_00130*)

[SWS_CORE_06353]{DRAFT} ⌈

Kind:	function
Symbol:	SetError(E &&error)
Scope:	class ara::core::Promise< void, E >

▽

△

Syntax:	`void SetError (E &&error);`	
Parameters (in):	error	the error to store
Return value:	None	
Header file:	#include "ara/core/promise.h"	
Description:	Move an error into the shared state and make the state ready.	

⌋(*RS_AP_00130*)

[SWS_CORE_06354]{DRAFT} ⌈

Kind:	function	
Symbol:	SetError(const E &error)	
Scope:	class ara::core::Promise< void, E >	
Syntax:	`void SetError (const E &error);`	
Parameters (in):	error	the error to store
Return value:	None	
Header file:	#include "ara/core/promise.h"	
Description:	Copy an error into the shared state and make the state ready.	

⌋(*RS_AP_00130*)

[SWS_CORE_06355]{DRAFT} ⌈

Kind:	function	
Symbol:	SetResult(const Result< void, E > &result)	
Scope:	class ara::core::Promise< void, E >	
Syntax:	`void SetResult (const Result< void, E > &result);`	
Parameters (in):	result	the result to store
Return value:	None	
Header file:	#include "ara/core/promise.h"	
Description:	Copy a Result into the shared state and make the state ready.	

⌋(*RS_AP_00130*)

[SWS_CORE_06356]{DRAFT} ⌈

Kind:	function	
Symbol:	SetResult(Result< void, E > &&result)	
Scope:	class ara::core::Promise< void, E >	
Syntax:	`void SetResult (Result< void, E > &&result);`	
Parameters (in):	result	the result to store
Return value:	None	
Header file:	#include "ara/core/promise.h"	

▽

△

Description:	Move a Result into the shared state and make the state ready.

⌋(*RS_AP_00130*)

8.1.7 `Array` data type

This section describes the `ara::core::Array` type that represents a container which encapsulates fixed size arrays.

8.1.7.1 Class Array

[SWS_CORE_01201]{DRAFT} ⌈

Kind:	class	
Symbol:	Array	
Scope:	namespace ara::core	
Syntax:	`template <typename T, std::size_t N>` `class Array final {...};`	
Template param:	typename T	the type of element in the array
	std::size_t N	the number of elements in the array
Header file:	#include "ara/core/array.h"	
Description:	Encapsulation of fixed size arrays.	

⌋(*RS_AP_00130*)

[SWS_CORE_01210]{DRAFT} ⌈

Kind:	type alias
Symbol:	reference
Scope:	class ara::core::Array
Derived from:	T&
Syntax:	`using reference = T&;`
Header file:	#include "ara/core/array.h"
Description:	Alias type for a reference to an element.

⌋(*RS_AP_00130*)

[SWS_CORE_01211]{DRAFT} ⌈

Kind:	type alias
Symbol:	const_reference
Scope:	class ara::core::Array
Derived from:	const T&
Syntax:	`using const_reference = const T&;`
Header file:	#include "ara/core/array.h"
Description:	Alias type for a const_reference to an element.

⌋(*RS_AP_00130*)

[SWS_CORE_01212]{DRAFT} ⌈

Kind:	type alias
Symbol:	iterator
Scope:	class ara::core::Array
Derived from:	T*
Syntax:	`using iterator = T*;`
Header file:	#include "ara/core/array.h"
Description:	The type of an iterator to elements.

⌋(*RS_AP_00130*)

[SWS_CORE_01213]{DRAFT} ⌈

Kind:	type alias
Symbol:	const_iterator
Scope:	class ara::core::Array
Derived from:	const T*
Syntax:	`using const_iterator = const T*;`
Header file:	#include "ara/core/array.h"
Description:	The type of a const_iterator to elements.

⌋(*RS_AP_00130*)

[SWS_CORE_01214]{DRAFT} ⌈

Kind:	type alias
Symbol:	size_type
Scope:	class ara::core::Array
Derived from:	std::size_t
Syntax:	`using size_type = std::size_t;`
Header file:	#include "ara/core/array.h"
Description:	Alias for the type of parameters that indicate an index into the Array.

⌋(*RS_AP_00130*)

[SWS_CORE_01215]{DRAFT} ⌈

Kind:	type alias
Symbol:	difference_type
Scope:	class ara::core::Array
Derived from:	std::ptrdiff_t
Syntax:	`using difference_type = std::ptrdiff_t;`
Header file:	#include "ara/core/array.h"
Description:	Alias for the type of parameters that indicate a difference of indexes into the Array.

⌋(*RS_AP_00130*)

[SWS_CORE_01216]{DRAFT} ⌈

Kind:	type alias
Symbol:	value_type
Scope:	class ara::core::Array
Derived from:	T
Syntax:	`using value_type = T;`
Header file:	#include "ara/core/array.h"
Description:	Alias for the type of elements in this Array.

⌋(*RS_AP_00130*)

[SWS_CORE_01217]{DRAFT} ⌈

Kind:	type alias
Symbol:	pointer
Scope:	class ara::core::Array
Derived from:	T*
Syntax:	`using pointer = T*;`
Header file:	#include "ara/core/array.h"
Description:	Alias type for a pointer to an element.

⌋(*RS_AP_00130*)

[SWS_CORE_01218]{DRAFT} ⌈

Kind:	type alias
Symbol:	const_pointer
Scope:	class ara::core::Array
Derived from:	const T*
Syntax:	`using const_pointer = const T*;`
Header file:	#include "ara/core/array.h"
Description:	Alias type for a pointer to a const element.

⌋(*RS_AP_00130*)

[SWS_CORE_01219]{DRAFT} ⌈

Kind:	type alias
Symbol:	reverse_iterator
Scope:	class ara::core::Array
Derived from:	std::reverse_iterator<iterator>
Syntax:	`using reverse_iterator = std::reverse_iterator<iterator>;`
Header file:	#include "ara/core/array.h"
Description:	The type of a reverse_iterator to elements.

⌋(*RS_AP_00130*)

[SWS_CORE_01220]{DRAFT} ⌈

Kind:	type alias
Symbol:	const_reverse_iterator
Scope:	class ara::core::Array
Derived from:	std::reverse_iterator<const_iterator>
Syntax:	`using const_reverse_iterator = std::reverse_iterator<const_iterator>;`
Header file:	#include "ara/core/array.h"
Description:	The type of a const_reverse_iterator to elements.

⌋(*RS_AP_00130*)

[SWS_CORE_01241]{DRAFT} ⌈

Kind:	function	
Symbol:	fill(const T &u)	
Scope:	class ara::core::Array	
Syntax:	`void fill (const T &u);`	
Parameters (in):	u	the value
Return value:	None	
Header file:	#include "ara/core/array.h"	
Description:	Assign the given value to all elements of this Array.	

⌋(*RS_AP_00130*)

[SWS_CORE_01242]{DRAFT} ⌈

Kind:	function	
Symbol:	swap(Array< T, N > &other)	
Scope:	class ara::core::Array	
Syntax:	`void swap (Array< T, N > &other) noexcept(noexcept(swap(std::declval< T & >(), std::declval< T & >())));`	
Parameters (inout):	other	the other Array
Return value:	None	

▽

△

Exception Safety:	conditionally noexcept
Header file:	#include "ara/core/array.h"
Description:	Exchange the contents of this Array with those of other. The noexcept specification shall make use of ADL for the swap() call.

⌋(*RS_AP_00130*)

[SWS_CORE_01250]{DRAFT} ⌈

Kind:	function	
Symbol:	begin()	
Scope:	class ara::core::Array	
Syntax:	`iterator begin () noexcept;`	
Return value:	iterator	the iterator
Exception Safety:	noexcept	
Header file:	#include "ara/core/array.h"	
Description:	Return an iterator pointing to the first element of this Array.	

⌋(*RS_AP_00130*)

[SWS_CORE_01251]{DRAFT} ⌈

Kind:	function	
Symbol:	begin()	
Scope:	class ara::core::Array	
Syntax:	`const_iterator begin () const noexcept;`	
Return value:	const_iterator	the const_iterator
Exception Safety:	noexcept	
Header file:	#include "ara/core/array.h"	
Description:	Return a const_iterator pointing to the first element of this Array.	

⌋(*RS_AP_00130*)

[SWS_CORE_01252]{DRAFT} ⌈

Kind:	function	
Symbol:	end()	
Scope:	class ara::core::Array	
Syntax:	`iterator end () noexcept;`	
Return value:	iterator	the iterator
Exception Safety:	noexcept	
Header file:	#include "ara/core/array.h"	
Description:	Return an iterator pointing past the last element of this Array.	

⌋(*RS_AP_00130*)

[SWS_CORE_01253]{DRAFT} ⌈

Kind:	function	
Symbol:	end()	
Scope:	class ara::core::Array	
Syntax:	`const_iterator end () const noexcept;`	
Return value:	const_iterator	the const_iterator
Exception Safety:	noexcept	
Header file:	#include "ara/core/array.h"	
Description:	Return a const_iterator pointing past the last element of this Array.	

⌋(*RS_AP_00130*)

[SWS_CORE_01254]{DRAFT} ⌈

Kind:	function	
Symbol:	rbegin()	
Scope:	class ara::core::Array	
Syntax:	`reverse_iterator rbegin () noexcept;`	
Return value:	reverse_iterator	the reverse_iterator
Exception Safety:	noexcept	
Header file:	#include "ara/core/array.h"	
Description:	Return a reverse_iterator pointing to the last element of this Array.	

⌋(*RS_AP_00130*)

[SWS_CORE_01255]{DRAFT} ⌈

Kind:	function	
Symbol:	rbegin()	
Scope:	class ara::core::Array	
Syntax:	`const_reverse_iterator rbegin () const noexcept;`	
Return value:	const_reverse_iterator	the const_reverse_iterator
Exception Safety:	noexcept	
Header file:	#include "ara/core/array.h"	
Description:	Return a const_reverse_iterator pointing to the last element of this Array.	

⌋(*RS_AP_00130*)

[SWS_CORE_01256]{DRAFT} ⌈

Kind:	function
Symbol:	rend()
Scope:	class ara::core::Array
Syntax:	`reverse_iterator rend () noexcept;`

▽

△

Return value:	reverse_iterator	the reverse_iterator
Exception Safety:	noexcept	
Header file:	#include "ara/core/array.h"	
Description:	Return a reverse_iterator pointing past the first element of this Array.	

⌋(*RS_AP_00130*)

[SWS_CORE_01257]{DRAFT} ⌈

Kind:	function	
Symbol:	rend()	
Scope:	class ara::core::Array	
Syntax:	`const_reverse_iterator rend () const noexcept;`	
Return value:	const_reverse_iterator	the const_reverse_iterator
Exception Safety:	noexcept	
Header file:	#include "ara/core/array.h"	
Description:	Return a const_reverse_iterator pointing past the first element of this Array.	

⌋(*RS_AP_00130*)

[SWS_CORE_01258]{DRAFT} ⌈

Kind:	function	
Symbol:	cbegin()	
Scope:	class ara::core::Array	
Syntax:	`const_iterator cbegin () const noexcept;`	
Return value:	const_iterator	the const_iterator
Exception Safety:	noexcept	
Header file:	#include "ara/core/array.h"	
Description:	Return a const_iterator pointing to the first element of this Array.	

⌋(*RS_AP_00130*)

[SWS_CORE_01259]{DRAFT} ⌈

Kind:	function	
Symbol:	cend()	
Scope:	class ara::core::Array	
Syntax:	`const_iterator cend () const noexcept;`	
Return value:	const_iterator	the const_iterator
Exception Safety:	noexcept	
Header file:	#include "ara/core/array.h"	
Description:	Return a const_iterator pointing past the last element of this Array.	

⌋(*RS_AP_00130*)

[SWS_CORE_01260]{DRAFT} ⌈

Kind:	function	
Symbol:	crbegin()	
Scope:	class ara::core::Array	
Syntax:	`const_reverse_iterator crbegin () const noexcept;`	
Return value:	const_reverse_iterator	the const_reverse_iterator
Exception Safety:	noexcept	
Header file:	#include "ara/core/array.h"	
Description:	Return a const_reverse_iterator pointing to the last element of this Array.	

⌋(*RS_AP_00130*)

[SWS_CORE_01261]{DRAFT} ⌈

Kind:	function	
Symbol:	crend()	
Scope:	class ara::core::Array	
Syntax:	`const_reverse_iterator crend () const noexcept;`	
Return value:	const_reverse_iterator	the const_reverse_iterator
Exception Safety:	noexcept	
Header file:	#include "ara/core/array.h"	
Description:	Return a const_reverse_iterator pointing past the first element of this Array.	

⌋(*RS_AP_00130*)

[SWS_CORE_01262]{DRAFT} ⌈

Kind:	function	
Symbol:	size()	
Scope:	class ara::core::Array	
Syntax:	`constexpr size_type size () const noexcept;`	
Return value:	size_type	N
Exception Safety:	noexcept	
Header file:	#include "ara/core/array.h"	
Description:	Return the number of elements in this Array.	

⌋(*RS_AP_00130*)

[SWS_CORE_01263]{DRAFT} ⌈

Kind:	function
Symbol:	max_size()
Scope:	class ara::core::Array

▽

△

Syntax:	`constexpr size_type max_size () const noexcept;`	
Return value:	size_type	N
Exception Safety:	noexcept	
Header file:	#include "ara/core/array.h"	
Description:	Return the maximum number of elements supported by this Array.	

⌋(*RS_AP_00130*)

[SWS_CORE_01264]{DRAFT} ⌈

Kind:	function	
Symbol:	empty()	
Scope:	class ara::core::Array	
Syntax:	`constexpr bool empty () const noexcept;`	
Return value:	bool	true if this Array contains 0 elements, false otherwise
Exception Safety:	noexcept	
Header file:	#include "ara/core/array.h"	
Description:	Return whether this Array is empty.	

⌋(*RS_AP_00130*)

[SWS_CORE_01265]{DRAFT} ⌈

Kind:	function	
Symbol:	operator[](size_type n)	
Scope:	class ara::core::Array	
Syntax:	`reference operator[] (size_type n);`	
Parameters (in):	n	the index into this Array
Return value:	reference	the reference
Header file:	#include "ara/core/array.h"	
Description:	Return a reference to the n-th element of this Array.	

⌋(*RS_AP_00130*)

[SWS_CORE_01266]{DRAFT} ⌈

Kind:	function	
Symbol:	operator[](size_type n)	
Scope:	class ara::core::Array	
Syntax:	`constexpr const_reference operator[] (size_type n) const;`	
Parameters (in):	n	the index into this Array
Return value:	const_reference	the const_reference
Header file:	#include "ara/core/array.h"	

▽

△

Description:	Return a const_reference to the n-th element of this Array.

⌋(*RS_AP_00130*)

[SWS_CORE_01267]{DRAFT} ⌈

Kind:	function	
Symbol:	front()	
Scope:	class ara::core::Array	
Syntax:	`reference front ();`	
Return value:	reference	the reference
Header file:	#include "ara/core/array.h"	
Description:	Return a reference to the first element of this Array. The behavior of this function is undefined if the Array is empty.	

⌋(*RS_AP_00130*)

[SWS_CORE_01268]{DRAFT} ⌈

Kind:	function	
Symbol:	front()	
Scope:	class ara::core::Array	
Syntax:	`constexpr const_reference front () const;`	
Return value:	const_reference	the reference
Header file:	#include "ara/core/array.h"	
Description:	Return a const_reference to the first element of this Array. The behavior of this function is undefined if the Array is empty.	

⌋(*RS_AP_00130*)

[SWS_CORE_01269]{DRAFT} ⌈

Kind:	function	
Symbol:	back()	
Scope:	class ara::core::Array	
Syntax:	`reference back ();`	
Return value:	reference	the reference
Header file:	#include "ara/core/array.h"	
Description:	Return a reference to the last element of this Array. The behavior of this function is undefined if the Array is empty.	

⌋(*RS_AP_00130*)

[SWS_CORE_01270]{DRAFT} ⌈

Kind:	function	
Symbol:	back()	
Scope:	class ara::core::Array	
Syntax:	`constexpr const_reference back () const;`	
Return value:	const_reference	the reference
Header file:	#include "ara/core/array.h"	
Description:	Return a const_reference to the last element of this Array. The behavior of this function is undefined if the Array is empty.	

⌋(*RS_AP_00130*)

[SWS_CORE_01271]{DRAFT} ⌈

Kind:	function	
Symbol:	data()	
Scope:	class ara::core::Array	
Syntax:	`pointer data () noexcept;`	
Return value:	pointer	the pointer
Exception Safety:	noexcept	
Header file:	#include "ara/core/array.h"	
Description:	Return a pointer to the first element of this Array.	

⌋(*RS_AP_00130*)

[SWS_CORE_01272]{DRAFT} ⌈

Kind:	function	
Symbol:	data()	
Scope:	class ara::core::Array	
Syntax:	`const_pointer data () const noexcept;`	
Return value:	const_pointer	the const_pointer
Exception Safety:	noexcept	
Header file:	#include "ara/core/array.h"	
Description:	Return a const_pointer to the first element of this Array.	

⌋(*RS_AP_00130*)

8.1.7.2 Global functions

[SWS_CORE_01290]{DRAFT} ⌈

Kind:	function	
Symbol:	operator==(const Array< T, N > &lhs, const Array< T, N > &rhs)	
Scope:	namespace ara::core	
Syntax:	`template <typename T, std::size_t N>` `bool operator== (const Array< T, N > &lhs, const Array< T, N > &rhs);`	
Template param:	T	the type of element in the Array
	N	the number of elements in the Array
Parameters (in):	lhs	the left-hand side of the comparison
	rhs	the right-hand side of the comparison
Return value:	bool	true if the Arrays are equal, false otherwise
Header file:	#include "ara/core/array.h"	
Description:	Return true if the two Arrays have equal content.	

⌋(*RS_AP_00130*)

[SWS_CORE_01291]{DRAFT} ⌈

Kind:	function	
Symbol:	operator!=(const Array< T, N > &lhs, const Array< T, N > &rhs)	
Scope:	namespace ara::core	
Syntax:	`template <typename T, std::size_t N>` `bool operator!= (const Array< T, N > &lhs, const Array< T, N > &rhs);`	
Template param:	T	the type of element in the Array
	N	the number of elements in the Array
Parameters (in):	lhs	the left-hand side of the comparison
	rhs	the right-hand side of the comparison
Return value:	bool	true if the Arrays are non-equal, false otherwise
Header file:	#include "ara/core/array.h"	
Description:	Return true if the two Arrays have non-equal content.	

⌋(*RS_AP_00130*)

[SWS_CORE_01292]{DRAFT} ⌈

Kind:	function	
Symbol:	operator<(const Array< T, N > &lhs, const Array< T, N > &rhs)	
Scope:	namespace ara::core	
Syntax:	`template <typename T, std::size_t N>` `bool operator< (const Array< T, N > &lhs, const Array< T, N > &rhs);`	
Template param:	T	the type of element in the Array
	N	the number of elements in the Array
Parameters (in):	lhs	the left-hand side of the comparison
	rhs	the right-hand side of the comparison
Return value:	bool	true if lhs is less than rhs, false otherwise

▽

△

Header file:	#include "ara/core/array.h"
Description:	Return true if the contents of lhs are lexicographically less than the contents of rhs.

⌋(*RS_AP_00130*)

[SWS_CORE_01293]{DRAFT} ⌈

Kind:	function	
Symbol:	operator>(const Array< T, N > &lhs, const Array< T, N > &rhs)	
Scope:	namespace ara::core	
Syntax:	`template <typename T, std::size_t N>` `bool operator> (const Array< T, N > &lhs, const Array< T, N > &rhs);`	
Template param:	T	the type of elemenr in the Array
	N	the number of elements in the Array
Parameters (in):	lhs	the left-hand side of the comparison
	rhs	the right-hand side of the comparison
Return value:	bool	true if rhs is less than lhs, false otherwise
Header file:	#include "ara/core/array.h"	
Description:	Return true if the contents of rhs are lexicographically less than the contents of lhs.	

⌋(*RS_AP_00130*)

[SWS_CORE_01294]{DRAFT} ⌈

Kind:	function	
Symbol:	operator<=(const Array< T, N > &lhs, const Array< T, N > &rhs)	
Scope:	namespace ara::core	
Syntax:	`template <typename T, std::size_t N>` `bool operator<= (const Array< T, N > &lhs, const Array< T, N > &rhs);`	
Template param:	T	the type of element in the Array
	N	the number of elements in the Array
Parameters (in):	lhs	the left-hand side of the comparison
	rhs	the right-hand side of the comparison
Return value:	bool	true if lhs is less than or equal to rhs, false otherwise
Header file:	#include "ara/core/array.h"	
Description:	Return true if the contents of lhs are lexicographically less than or equal to the contents of rhs.	

⌋(*RS_AP_00130*)

[SWS_CORE_01295]{DRAFT} ⌈

Kind:	function	
Symbol:	operator>=(const Array< T, N > &lhs, const Array< T, N > &rhs)	
Scope:	namespace ara::core	
Syntax:	`template <typename T, std::size_t N>` `bool operator>= (const Array< T, N > &lhs, const Array< T, N > &rhs);`	
Template param:	T	the type of element in the Array
	N	the number of elements in the Array
Parameters (in):	lhs	the left-hand side of the comparison
	rhs	the right-hand side of the comparison
Return value:	bool	true if rhs is less than or equal to lhs, false otherwise
Header file:	#include "ara/core/array.h"	
Description:	Return true if the contents of rhs are lexicographically less than or equal to the contents of lhs.	

⌋(*RS_AP_00130*)

[SWS_CORE_01296]{DRAFT} ⌈

Kind:	function	
Symbol:	swap(Array< T, N > &lhs, Array< T, N > &rhs)	
Scope:	namespace ara::core	

Syntax:	`template <typename T, std::size_t N>`
	`void swap (Array< T, N > &lhs, Array< T, N > &rhs)`
	`noexcept(noexcept(lhs.swap(rhs)));`	

Template param:	T	the type of element in the Arrays
	N	the number of elements in the Arrays
Parameters (in):	lhs	the left-hand side of the call
	rhs	the right-hand side of the call
Return value:	None	
Exception Safety:	conditionally noexcept	
Header file:	#include "ara/core/array.h"	
Description:	Overload of std::swap for ara::core::Array.	

⌋(*RS_AP_00130*)

8.1.7.3 Tuple interface

These definitions implement the standard interface of tuple-like types for class `Array`.

The specializations of the `std::tuple_size` and `std::tuple_element` traits are put into the `std` namespace:

[SWS_CORE_01280]{DRAFT} ⌈

Kind:	struct	
Symbol:	tuple_size< ara::core::Array< T, N > >	
Scope:	namespace std	
Syntax:	`template <typename T, size_t N>` `struct tuple_size< ara::core::Array< T, N > > : public integral_constant< size_t, N > {...};`	
Template param:	typename T	the type of element in the Array
	size_t N	the number of elements in the Array
Header file:	#include "ara/core/array.h"	
Description:	Specialization of std::tuple_size for ara::core::Array. This specialization shall meet the C++14 UnaryTypeTrait requirements with a Base Characteristic of std::integral_constant<std::size_t, N>.	

⌋(*RS_AP_00130*)

[SWS_CORE_01281]{DRAFT} ⌈

Kind:	struct	
Symbol:	tuple_element< I, ara::core::Array< T, N > >	
Scope:	namespace std	
Syntax:	`template <size_t I, typename T, size_t N>` `struct tuple_element< I, ara::core::Array< T, N > > {...};`	
Template param:	size_t I	the index into the Array whose type is desired
	typename T	the type of element in the Array
	size_t N	the number of elements in the Array
Header file:	#include "ara/core/array.h"	
Description:	Specialization of std::tuple_element for ara::core::Array. The implementation shall flag the condition I >= N as a compile error.	

⌋(*RS_AP_00130*)

[SWS_CORE_01285]{DRAFT} ⌈

Kind:	type alias
Symbol:	type
Scope:	struct std::tuple_element< I, ara::core::Array< T, N > >
Derived from:	T
Syntax:	`using type = T;`
Header file:	#include "ara/core/array.h"
Description:	Alias for the type of the Array element with the given index.

⌋(*RS_AP_00130*)

The overloads of `std::get` are contained in the `ara::core` namespace; they can either be called explicitly (i.e. namespace-qualified), or be invoked via ADL.

For ADL lookup to work in C++14, `get` needs to be called without namespace qualification, similar to the way that `swap` is recommended to be called, e.g.:

```
using std::get;

ara::core::Array<int, 4> array = {1, 2, 3, 4};
int& e = get<0>(array);
```

[SWS_CORE_01282]{DRAFT} ⌈

Kind:	function	
Symbol:	get(Array< T, N > &a)	
Scope:	namespace ara::core	
Syntax:	`template <std::size_t I, typename T, std::size_t N>` `constexpr T& get (Array< T, N > &a) noexcept;`	
Template param:	I	the index into the Array whose element is desired
	T	the type of element in the Array
	N	the number of elements in the Array
Parameters (in):	a	the Array
Return value:	T &	a reference to the Ith element of the Array
Exception Safety:	noexcept	
Header file:	#include "ara/core/array.h"	
Description:	Overload of std::get for an lvalue mutable ara::core::Array. The implementation shall flag the condition I >= N as a compile error.	

⌋(*RS_AP_00130*)

[SWS_CORE_01283]{DRAFT} ⌈

Kind:	function	
Symbol:	get(Array< T, N > &&a)	
Scope:	namespace ara::core	
Syntax:	`template <std::size_t I, typename T, std::size_t N>` `constexpr T&& get (Array< T, N > &&a) noexcept;`	
Template param:	I	the index into the Array whose element is desired
	T	the type of element in the Array
	N	the number of elements in the Array
Parameters (in):	a	the Array
Return value:	T &&	an rvalue reference to the Ith element of the Array
Exception Safety:	noexcept	
Header file:	#include "ara/core/array.h"	
Description:	Overload of std::get for an rvalue ara::core::Array. The implementation shall flag the condition I >= N as a compile error.	

⌋(*RS_AP_00130*)

[SWS_CORE_01284]{DRAFT} ⌈

Kind:	function	
Symbol:	get(const Array< T, N > &a)	
Scope:	namespace ara::core	
Syntax:	`template <std::size_t I, typename T, std::size_t N>` `constexpr T const& get (const Array< T, N > &a) noexcept;`	
Template param:	I	the index into the Array whose element is desired
	T	the type of element in the Array
	N	the number of elements in the Array
Parameters (in):	a	the Array
Return value:	T const &	a const_reference to the Ith element of the Array
Exception Safety:	noexcept	
Header file:	#include "ara/core/array.h"	
Description:	Overload of std::get for an lvalue const ara::core::Array. The implementation shall flag the condition I >= N as a compile error.	

⌋(*RS_AP_00130*)

8.1.8 `Vector` data type

This section describes the `ara::core::Vector` type that represents a container which can change in size.

[SWS_CORE_01301]{DRAFT} **`Vector` class template** ⌈The namespace `ara::core` shall provide a class template `Vector`:

```
template <typename T, typename Allocator = /* implementation-defined */>
class Vector { ... };
```

⌋(*RS_AP_00130*)

All members of this class shall behave identical to those of `std::vector` from [5] section 23.3.6, except that the default value for the `Allocator` template argument is implementation-defined.

[SWS_CORE_01390]{DRAFT} **Global `operator==` for `Vector`** ⌈The namespace `ara::core` shall provide a function template `operator==` for `Vector`:

```
template <typename T, typename Allocator>
bool operator==(const Vector<T, Allocator>& lhs,
                const Vector<T, Allocator>& rhs);
```

⌋(*RS_AP_00130*)

[SWS_CORE_01391]{DRAFT} **Global `operator!=` for `Vector`** ⌈The namespace `ara::core` shall provide a function template `operator!=` for `Vector`:

```
template <typename T, typename Allocator>
bool operator!=(const Vector<T, Allocator>& lhs,
                const Vector<T, Allocator>& rhs);
```

⌋(*RS_AP_00130*)

[SWS_CORE_01392]{DRAFT} **Global `operator<` for `Vector`** ⌈The namespace `ara::core` shall provide a function template `operator<` for `Vector`:

```
template <typename T, typename Allocator>
bool operator<(const Vector<T, Allocator>& lhs,
               const Vector<T, Allocator>& rhs);
```

⌋(*RS_AP_00130*)

[SWS_CORE_01393]{DRAFT} **Global `operator<=` for `Vector`** ⌈The namespace `ara::core` shall provide a function template `operator<=` for `Vector`:

```
template <typename T, typename Allocator>
bool operator<=(const Vector<T, Allocator>& lhs,
                const Vector<T, Allocator>& rhs);
```

⌋(*RS_AP_00130*)

[SWS_CORE_01394]{DRAFT} **Global `operator>` for `Vector`** ⌈The namespace `ara::core` shall provide a function template `operator>` for `Vector`:

```
template <typename T, typename Allocator>
bool operator>(const Vector<T, Allocator>& lhs,
               const Vector<T, Allocator>& rhs);
```

⌋(*RS_AP_00130*)

[SWS_CORE_01395]{DRAFT} **Global `operator>=` for `Vector`** ⌈The namespace `ara::core` shall provide a function template `operator>=` for `Vector`:

```
template <typename T, typename Allocator>
bool operator>=(const Vector<T, Allocator>& lhs,
                const Vector<T, Allocator>& rhs);
```

⌋(*RS_AP_00130*)

[SWS_CORE_01396]{DRAFT} **`swap` overload for `Vector`** ⌈There shall be an overload of the `swap` function within the namespace `ara::core` for arguments of type `Vector`. Its interface shall be equivalent to:

```
template <typename T, typename Allocator>
void swap(Vector<T, Allocator>& lhs, Vector<T, Allocator>& rhs);
```

This function shall exchange the state of lhs with that of rhs.⌋(*RS_AP_00130*)

8.1.9 `Map` data type

This section describes the `ara::core::Map` type that represents a container which contains key-value pairs with unique keys.

[SWS_CORE_01400]{DRAFT} **`Map` class template** ⌈The namespace `ara::core` shall provide a class template `Map`:

```
template <
    typename K,
    typename V,
    typename C = std::less<K>,
    typename Allocator = /* implementation-defined */
>
class Map { ... };
```

⌋*(RS_AP_00130)*

All members of this class and supporting constructs (such as global relational operators) shall behave identical to those of `std::map` in header `<map>` from [5] section 23.4.2, except that the default value for the `Allocator` template argument is implementation-defined. All supporting symbols shall be contained within namespace `ara::core`.

[SWS_CORE_01496]{DRAFT} **`swap` overload for `Map`** ⌈There shall be an overload of the `swap` function within the namespace `ara::core` for arguments of type `Map`. Its interface shall be equivalent to:

```
template <
    typename K,
    typename V,
    typename C,
    typename Allocator
>
void swap(Map<K, V, C, Allocator>& lhs, Map<K, V, C, Allocator>& rhs);
```

This function shall exchange the state of lhs with that of rhs.⌋*(RS_AP_00130)*

8.1.10 `Optional` data type

This section describes the class template `ara::core::Optional` that provides access to optional record elements of a `Structure Implementation data type`. Whenever there is a mention of the standard C++17 item `std::optional`, the implied source material is [10].

The class template `ara::core::Optional` manages optional values, i.e. values that may or may not be present. The existence can be evaluated during both compile-time and runtime.
Note: Mandatory record elements are declared directly with the corresponding `ImplementationDataType` without using `ara::core::Optional`.

[SWS_CORE_01033]{DRAFT} **`Optional` class template** ⌈The namespace `ara::core` shall provide a class template `Optional`:

```
template <typename T>
class Optional { ... };
```

⌋(*RS_AP_00130*)

All members of this class and supporting constructs (such as global relational operators) shall behave identical to those of header `<optional>` from [10] section 23.6, with the exceptions as given below. All supporting symbols shall be contained within namespace `ara::core`.

[SWS_CORE_01030]{DRAFT} **`value` member function overloads** ⌈Contrary to the description in [10], no member functions with this name exist in `ara::core::Optional`.⌋(*RS_AP_00130*)

[SWS_CORE_01031]{DRAFT} **class bad_optional_access** ⌈No class named `bad_optional_access` is defined in the `ara::core` namespace.⌋(*RS_AP_00130*)

[SWS_CORE_01096]{DRAFT} **`swap` overload for `Optional`** ⌈There shall be an overload of the `swap` function within the namespace `ara::core` for arguments of type `Optional`. Its interface shall be equivalent to:

```
template <typename T>
void swap(Optional<T>& lhs, Optional<T>& rhs);
```

This function shall exchange the state of lhs with that of rhs.⌋(*RS_AP_00130*)

8.1.11 `Variant` data type

This section describes the `ara::core::Variant` type that represents a type-safe union.

[SWS_CORE_01601]{DRAFT} **`Variant` class template** ⌈The namespace `ara::core` shall provide a class template `Variant`:

```
template <typename... Ts>
class Variant { ... };
```

⌋(*RS_AP_00130*)

All members and supporting constructs (such as global relational operators) of this class shall behave identical to those of header `<variant>` from [10] section 23.7. All supporting symbols shall be contained within namespace `ara::core`.

[SWS_CORE_01696]{DRAFT} **`swap` overload for `Variant`** ⌈There shall be an overload of the `swap` function within the namespace `ara::core` for arguments of type `Variant`. Its interface shall be equivalent to:

```
template <typename... Ts>
void swap(Variant<Ts...>& lhs, Variant<Ts...>& rhs);
```

This function shall exchange the state of lhs with that of rhs.⌋(*RS_AP_00130*)

8.1.12 `StringView` data type

This section describes the `ara::core::StringView` type that constitutes a read-only view over a contiguous sequence of characters, the storage of which is owned by another object.

[SWS_CORE_02001]{DRAFT} **`StringView` class** ⌈The namespace `ara::core` shall provide a class `StringView`:

```
class StringView { ... };
```

⌋*(RS_AP_00130)*

All members of this class and supporting constructs (such as global relational operators) shall behave identical to those of header `<string_view>` from [10] section 24.4, except that non-`const` member functions are never declared with `constexpr`. (Note: This makes them compatible to C++11's semantics of `constexpr` member functions, which are always implicitly `const`.)
All supporting symbols shall be contained within namespace `ara::core`.

8.1.13 String data types

This section describes the `ara::core::String` type and its complement `ara::core::BasicString` which both represent sequences of characters.

These types are closely modeled on `std::string` and `std::basic_string` respectively from [5, the C++11 standard], with a number of additions coming from [10, the C++17 standard].

[SWS_CORE_03000]{DRAFT} **`BasicString` type** ⌈The namespace `ara::core` shall provide a template type `BasicString`:

```
template <typename Allocator = /* implementation-defined */>
class BasicString { ... };
```

All members of this class and supporting constructs (such as global relational operators) shall behave identical to those of `std::basic_string` in header `<string>` from [5, the C++11 standard] section 21.3, except that the default value for the `Allocator` template argument is implementation-defined. The character type is fixed to `char`, and the traits type is fixed to `std::char_traits<char>`. All supporting symbols shall be contained within namespace `ara::core`.⌋*(RS_AP_00130)*

[SWS_CORE_03001]{DRAFT} **`String` type** ⌈The namespace `ara::core` shall provide a type alias `String`:

```
using String = BasicString<>;
```

⌋*(RS_AP_00130)*

[SWS_CORE_03301]{DRAFT} **Implicit conversion to `StringView`** ⌈An operator shall be defined for `BasicString` that provides implicit conversion to `StringView`:

```
operator StringView() const noexcept;
```

This function shall behave the same as the corresponding `std::basic_string` function from [10, the C++17 standard].⌋*(RS_AP_00130)*

[SWS_CORE_03302]{DRAFT} **Constructor from `StringView`** ⌈A constructor shall be defined for `BasicString` that accepts a `StringView` argument by value:

```
explicit BasicString(StringView sv);
```

This function shall behave the same as the corresponding `std::basic_string` function from [10, the C++17 standard].⌋*(RS_AP_00130)*

[SWS_CORE_03303]{DRAFT} **Constructor from implicit `StringView`** ⌈A constructor shall be defined for `BasicString` that accepts any type that is implicitly convertible to `StringView`:

```
template <typename T>
BasicString(const T& t, size_type pos, size_type n,
            const Allocator& a = Allocator());
```

This function shall behave the same as the corresponding `std::basic_string` function from [10, the C++17 standard].⌋*(RS_AP_00130)*

[SWS_CORE_03304]{DRAFT} **operator= from `StringView`** ⌈An operator= member function shall be defined for `BasicString` that accepts a `StringView` argument by value:

```
BasicString& operator=(StringView sv);
```

This function shall behave the same as the corresponding `std::basic_string` function from [10, the C++17 standard].⌋*(RS_AP_00130)*

[SWS_CORE_03305]{DRAFT} **Assignment from `StringView`** ⌈A member function shall be defined for `BasicString` that allows assignment from `StringView`:

```
BasicString& assign(StringView sv);
```

This function shall behave the same as the corresponding `std::basic_string` function from [10, the C++17 standard].⌋*(RS_AP_00130)*

[SWS_CORE_03306]{DRAFT} **Assignment from implicit `StringView`** ⌈A member function shall be defined for `BasicString` that allows assignment from any type that is implicitly convertible to `StringView`:

```
template <typename T>
BasicString& assign(const T& t, size_type pos, size_type n = npos);
```

This function shall behave the same as the corresponding `std::basic_string` function from [10, the C++17 standard].⌋*(RS_AP_00130)*

[SWS_CORE_03307]{DRAFT} **operator+ from `StringView`** ⌈An operator+= member function shall be defined for `BasicString` that accepts a `StringView` argument by value:

```
BasicString& operator+=(StringView sv);
```

This function shall behave the same as the corresponding `std::basic_string` function from [10, the C++17 standard].⌋*(RS_AP_00130)*

[SWS_CORE_03308]{DRAFT} **Concatenation of `StringView`** ⌈A member function shall be defined for `BasicString` that allows concatenation of a `StringView`:

```
BasicString& append(StringView sv);
```

This function shall behave the same as the corresponding `std::basic_string` function from [10, the C++17 standard].⌋*(RS_AP_00130)*

[SWS_CORE_03309]{DRAFT} **Concatenation of implicit `StringView`** ⌈A member function shall be defined for `BasicString` that allows concatenation of any type that is implicitly convertible to `StringView`:

```
template <typename T>
BasicString& append(const T& t, size_type pos, size_type n = npos);
```

This function shall behave the same as the corresponding `std::basic_string` function from [10, the C++17 standard].⌋*(RS_AP_00130)*

[SWS_CORE_03310]{DRAFT} **Insertion of `StringView`** ⌈A member function shall be defined for `BasicString` that allows insertion of a `StringView`:

```
BasicString& insert(size_type pos, StringView sv);
```

This function shall behave the same as the corresponding `std::basic_string` function from [10, the C++17 standard].⌋*(RS_AP_00130)*

[SWS_CORE_03311]{DRAFT} **Insertion of implicit `StringView`** ⌈A member function shall be defined for `BasicString` that allows insertion of any type that is implicitly convertible to `StringView`:

```
template <typename T>
BasicString& insert(size_type pos1, const T& t,
                    size_type pos2, size_type n = npos);
```

This function shall behave the same as the corresponding `std::basic_string` function from [10, the C++17 standard].⌋*(RS_AP_00130)*

[SWS_CORE_03312]{DRAFT} **Replacement with `StringView`** ⌈A member function shall be defined for `BasicString` that allows replacement of a subsequence of *this with the contents of a `StringView`:

```
BasicString& replace(size_type pos1, size_type n1, StringView sv);
```

This function shall behave the same as the corresponding `std::basic_string` function from [10, the C++17 standard].⌋*(RS_AP_00130)*

[SWS_CORE_03313]{DRAFT} **Replacement with implicit `StringView`** ⌈A member function shall be defined for `BasicString` that allows replacement of a subsequence of *this with the contents of any type that is implicitly convertible to `StringView`:

```
template <typename T>
BasicString& replace(size_type pos1, size_type n1, const T& t,
                     size_type pos2, size_type n2 = npos);
```

This function shall behave the same as the corresponding `std::basic_string` function from [10, the C++17 standard].⌋*(RS_AP_00130)*

[SWS_CORE_03314]{DRAFT} **Replacement of iterator range with `StringView`** ⌈A member function shall be defined for `BasicString` that allows replacement of an iterator-bounded subsequence of *this with the contents of a `StringView`:

```
BasicString& replace(const_iterator i1, const_iterator i2, StringView sv);
```

This function shall behave the same as the corresponding `std::basic_string` function from [10, the C++17 standard].⌋*(RS_AP_00130)*

[SWS_CORE_03315]{DRAFT} **Forward-find a `StringView`** ⌈A member function shall be defined for `BasicString` that allows forward-searching for the contents of a `StringView`:

```
size_type find(StringView sv, size_type pos = 0) const noexcept;
```

This function shall behave the same as the corresponding `std::basic_string` function from [10, the C++17 standard].⌋*(RS_AP_00130)*

[SWS_CORE_03316]{DRAFT} **Reverse-find a `StringView`** ⌈A member function shall be defined for `BasicString` that allows reverse-searching for the contents of a `StringView`:

```
size_type rfind(StringView sv, size_type pos = npos) const noexcept;
```

This function shall behave the same as the corresponding `std::basic_string` function from [10, the C++17 standard].⌋*(RS_AP_00130)*

[SWS_CORE_03317]{DRAFT} **Forward-find of character set within a `StringView`** ⌈A member function shall be defined for `BasicString` that allows forward-searching for any of the characters within a `StringView`:

```
size_type find_first_of(StringView sv,
                        size_type pos = 0) const noexcept;
```

This function shall behave the same as the corresponding `std::basic_string` function from [10, the C++17 standard].⌋*(RS_AP_00130)*

[SWS_CORE_03318]{DRAFT} **Reverse-find of character set within a `StringView`** ⌈A member function shall be defined for `BasicString` that allows reverse-searching for any of the characters within a `StringView`:

```
size_type find_last_of(StringView sv,
                       size_type pos = npos) const noexcept;
```

This function shall behave the same as the corresponding `std::basic_string` function from [10, the C++17 standard].⌋*(RS_AP_00130)*

[SWS_CORE_03319]{DRAFT} **Forward-find of character set not within a `StringView`** ⌈A member function shall be defined for `BasicString` that allows forward-searching for any of the characters not contained in a `StringView`:

```
size_type find_first_not_of(StringView sv,
                            size_type pos = 0) const noexcept;
```

This function shall behave the same as the corresponding `std::basic_string` function from [10, the C++17 standard].⌋*(RS_AP_00130)*

[SWS_CORE_03320]{DRAFT} **Reverse-find of character set not within a `StringView`** ⌈A member function shall be defined for `BasicString` that allows reverse-searching for any of the characters not contained in a `StringView`:

```
size_type find_last_not_of(StringView sv,
                           size_type pos = npos) const noexcept;
```

This function shall behave the same as the corresponding `std::basic_string` function from [10, the C++17 standard].⌋*(RS_AP_00130)*

[SWS_CORE_03321]{DRAFT} **Comparison with a `StringView`** ⌈A member function shall be defined for `BasicString` that allows comparison with the contents of a `StringView`:

```
int compare(StringView sv) const noexcept;
```

This function shall behave the same as the corresponding `std::basic_string` function from [10, the C++17 standard].⌋*(RS_AP_00130)*

[SWS_CORE_03322]{DRAFT} **Comparison of subsequence with a `StringView`** ⌈A member function shall be defined for `BasicString` that allows comparison of a subsequence of *this with the contents of a `StringView`:

```
int compare(size_type pos1, size_type n1, StringView sv) const;
```

This function shall behave the same as the corresponding `std::basic_string` function from [10, the C++17 standard].⌋*(RS_AP_00130)*

[SWS_CORE_03323]{DRAFT} **Comparison of subsequence with a subsequence of a `StringView`** ⌈A member function shall be defined for `BasicString` that allows comparison of a subsequence of *this with the contents of a subsequence of any type that is implicitly convertible to `StringView`:

```
template <typename T>
int compare(size_type pos1, size_type n1, const T& t,
            size_type pos2, size_type n2 = npos) const;
```

This function shall behave the same as the corresponding `std::basic_string` function from [10, the C++17 standard].⌋*(RS_AP_00130)*

[SWS_CORE_03296]{DRAFT} **`swap` overload for `BasicString`** ⌈There shall be an overload of the `swap` function within the namespace `ara::core` for arguments of type `BasicString`. Its interface shall be equivalent to:

```
template <typename Allocator>
void swap(BasicString<Allocator>& lhs, BasicString<Allocator>& rhs);
```

This function shall exchange the state of lhs with that of rhs.⌋(*RS_AP_00130*)

8.1.14 `Span` data type

This section describes the `ara::core::Span` type that constitutes a view over a contiguous sequence of objects, the storage of which is owned by another object.

This specification is based on the draft standard of `std::span` from [11] section 22.7, but has been adapted in several ways:

- All symbols from section 22.7.3.8 (`span.tuple`) have been omitted.
- These class members have been omitted: `front()`, `back()`, `const_pointer`, `const_reference`.
- These "friend" functions have been omitted: `begin()` and `end()`.
- All references to `std::array` have been replaced with `ara::core::Array`; support for `std::array` still exists with the generic Container-based functions, with only a minuscule performance penalty.
- `constexpr` has been omitted from the assignment operator, because it would make the operator implicitly `const` in C++11.
- An additional type alias `Span::size_type` has been added.
- A number of global `MakeSpan` function overloads have been added.

[SWS_CORE_01901]{DRAFT} ⌈

Kind:	variable
Symbol:	dynamic_extent
Scope:	namespace ara::core
Type:	std::size_t
Syntax:	`constexpr std::size_t dynamic_extent = std::numeric_limits<std::size_t>::max();`
Header file:	#include "ara/core/span.h"
Description:	A constant for creating Spans with dynamic sizes. The constant is always set to std::numeric_limits<std::size_t>::max().

⌋(*RS_AP_00130*)

[SWS_CORE_01900]{DRAFT} ⌈

Kind:	class	
Symbol:	Span	
Scope:	namespace ara::core	
Syntax:	`template <typename T, std::size_t Extent = dynamic_extent>` `class Span {...};`	
Template param:	typename T	the type of elements in the Span
	std::size_t Extent = dynamic_extent	the extent to use for this Span
Header file:	#include "ara/core/span.h"	
Description:	A view over a contiguous sequence of objects.	

⌋(*RS_AP_00130*)

[SWS_CORE_01911]{DRAFT} ⌈

Kind:	type alias
Symbol:	element_type
Scope:	class ara::core::Span
Derived from:	T
Syntax:	`using element_type = T;`
Header file:	#include "ara/core/span.h"
Description:	Alias for the type of elements in this Span.

⌋(*RS_AP_00130*)

[SWS_CORE_01912]{DRAFT} ⌈

Kind:	type alias
Symbol:	value_type
Scope:	class ara::core::Span
Derived from:	typename std::remove_cv<element_type>::type
Syntax:	`using value_type = typename std::remove_cv<element_type>::type;`
Header file:	#include "ara/core/span.h"
Description:	Alias for the type of values in this Span.

⌋(*RS_AP_00130*)

[SWS_CORE_01913]{DRAFT} ⌈

Kind:	type alias
Symbol:	index_type
Scope:	class ara::core::Span
Derived from:	std::size_t
Syntax:	`using index_type = std::size_t;`
Header file:	#include "ara/core/span.h"

▽

△

Description:	Alias for the type of parameters that indicate an index into the Span.

⌋(*RS_AP_00130*)

[SWS_CORE_01914]{DRAFT} ⌈

Kind:	type alias
Symbol:	difference_type
Scope:	class ara::core::Span
Derived from:	std::ptrdiff_t
Syntax:	`using difference_type = std::ptrdiff_t;`
Header file:	#include "ara/core/span.h"
Description:	Alias for the type of parameters that indicate a difference of indexes into the Span.

⌋(*RS_AP_00130*)

[SWS_CORE_01921]{DRAFT} ⌈

Kind:	type alias
Symbol:	size_type
Scope:	class ara::core::Span
Derived from:	index_type
Syntax:	`using size_type = index_type;`
Header file:	#include "ara/core/span.h"
Description:	Alias for the type of parameters that indicate a size or a number of values.
Notes:	This is an AUTOSAR addition that is not contained in std::span.

⌋(*RS_AP_00130*)

[SWS_CORE_01915]{DRAFT} ⌈

Kind:	type alias
Symbol:	pointer
Scope:	class ara::core::Span
Derived from:	element_type*
Syntax:	`using pointer = element_type*;`
Header file:	#include "ara/core/span.h"
Description:	Alias type for a pointer to an element.

⌋(*RS_AP_00130*)

[SWS_CORE_01916]{DRAFT} ⌈

Kind:	type alias
Symbol:	reference
Scope:	class ara::core::Span
Derived from:	element_type&
Syntax:	`using reference = element_type&;`
Header file:	#include "ara/core/span.h"
Description:	Alias type for a reference to an element.

⌋(*RS_AP_00130*)

[SWS_CORE_01917]{DRAFT} ⌈

Kind:	type alias
Symbol:	iterator
Scope:	class ara::core::Span
Derived from:	implementation_defined
Syntax:	`using iterator = implementation_defined;`
Header file:	#include "ara/core/span.h"
Description:	The type of an iterator to elements. This iterator shall implement the concepts RandomAccessIterator, ContiguousIterator, and ConstexprIterator.

⌋(*RS_AP_00130*)

[SWS_CORE_01918]{DRAFT} ⌈

Kind:	type alias
Symbol:	const_iterator
Scope:	class ara::core::Span
Derived from:	implementation_defined
Syntax:	`using const_iterator = implementation_defined;`
Header file:	#include "ara/core/span.h"
Description:	The type of a const_iterator to elements. This iterator shall implement the concepts RandomAccessIterator, ContiguousIterator, and ConstexprIterator.

⌋(*RS_AP_00130*)

[SWS_CORE_01919]{DRAFT} ⌈

Kind:	type alias
Symbol:	reverse_iterator
Scope:	class ara::core::Span

▽

△

Derived from:	std::reverse_iterator<iterator>
Syntax:	`using reverse_iterator = std::reverse_iterator<iterator>;`
Header file:	#include "ara/core/span.h"
Description:	The type of a reverse_iterator to elements.

⌋(*RS_AP_00130*)

[SWS_CORE_01920]{DRAFT} ⌈

Kind:	type alias
Symbol:	const_reverse_iterator
Scope:	class ara::core::Span
Derived from:	std::reverse_iterator<const_iterator>
Syntax:	`using const_reverse_iterator = std::reverse_iterator<const_iterator>;`
Header file:	#include "ara/core/span.h"
Description:	The type of a const_reverse_iterator to elements.

⌋(*RS_AP_00130*)

[SWS_CORE_01931]{DRAFT} ⌈

Kind:	variable
Symbol:	extent
Scope:	class ara::core::Span
Type:	index_type
Syntax:	`constexpr index_type extent = Extent;`
Header file:	#include "ara/core/span.h"
Description:	A constant reflecting the configured Extent of this Span.

⌋(*RS_AP_00130*)

[SWS_CORE_01941]{DRAFT} ⌈

Kind:	function
Symbol:	Span()
Scope:	class ara::core::Span
Syntax:	`constexpr Span () noexcept;`
Exception Safety:	noexcept
Header file:	#include "ara/core/span.h"
Description:	Default constructor. This constructor shall not participate in overload resolution unless Extent <= 0 is true.

⌋(*RS_AP_00130*)

[SWS_CORE_01942]{DRAFT} ⌈

Kind:	function	
Symbol:	Span(pointer ptr, index_type count)	
Scope:	class ara::core::Span	
Syntax:	`constexpr Span (pointer ptr, index_type count);`	
Parameters (in):	ptr	the pointer
	count	the number of elements to take from ptr
Header file:	#include "ara/core/span.h"	
Description:	Construct a new Span from the given pointer and size. [ptr, ptr + count) shall be a valid range. If Extent is not equal to dynamic_extent, then count shall be equal to Extent.	

⌋(*RS_AP_00130*)

[SWS_CORE_01943]{DRAFT} ⌈

Kind:	function	
Symbol:	Span(pointer firstElem, pointer lastElem)	
Scope:	class ara::core::Span	
Syntax:	`constexpr Span (pointer firstElem, pointer lastElem);`	
Parameters (in):	firstElem	pointer to the first element
	lastElem	pointer to past the last element
Header file:	#include "ara/core/span.h"	
Description:	Construct a new Span from the open range between [firstElem, lastElem). [first, last) shall be a valid range. If @ extent is not equal to dynamic_extent, then (last - first) shall be equal to Extent.	

⌋(*RS_AP_00130*)

[SWS_CORE_01944]{DRAFT} ⌈

Kind:	function	
Symbol:	Span(element_type(&arr)[N])	
Scope:	class ara::core::Span	
Syntax:	`template <std::size_t N>` `constexpr Span (element_type(&arr)[N]) noexcept;`	
Template param:	N	the size of the raw array
Parameters (in):	arr	the raw array
Exception Safety:	noexcept	
Header file:	#include "ara/core/span.h"	
Description:	Construct a new Span from the given raw array. This constructor shall not participate in overload resolution unless: Extent == dynamic_extent || N == Extent is true, and std::remove_pointer<decltype(ara::core::data(arr))>::type(*)[] is convertible to T(*)[].	

⌋(*RS_AP_00130*)

[SWS_CORE_01945]{DRAFT} ⌈

Kind:	function	
Symbol:	Span(Array< value_type, N > &arr)	
Scope:	class ara::core::Span	
Syntax:	`template <std::size_t N>` `constexpr Span (Array< value_type, N > &arr) noexcept;`	
Template param:	N	the size of the Array
Parameters (in):	arr	the array
Exception Safety:	noexcept	
Header file:	#include "ara/core/span.h"	
Description:	Construct a new Span from the given Array. This constructor shall not participate in overload resolution unless: Extent == dynamic_extent || N == Extent is true, and std::remove_pointer<decltype(ara::core::data(arr))>::type(*)[] is convertible to T(*)[].	

⌋(*RS_AP_00130*)

[SWS_CORE_01946]{DRAFT} ⌈

Kind:	function	
Symbol:	Span(const Array< value_type, N > &arr)	
Scope:	class ara::core::Span	
Syntax:	`template <std::size_t N>` `constexpr Span (const Array< value_type, N > &arr) noexcept;`	
Template param:	N	the size of the Array
Parameters (in):	arr	the array
Exception Safety:	noexcept	
Header file:	#include "ara/core/span.h"	
Description:	Construct a new Span from the given const Array. This constructor shall not participate in overload resolution unless: Extent == dynamic_extent || N == Extent is true, and std::remove_pointer<decltype(ara::core::data(arr))>::type(*)[] is convertible to T(*)[].	

⌋(*RS_AP_00130*)

[SWS_CORE_01947]{DRAFT} ⌈

Kind:	function	
Symbol:	Span(Container &cont)	
Scope:	class ara::core::Span	
Syntax:	`template <typename Container>` `constexpr Span (Container &cont);`	
Template param:	Container	the type of container
Parameters (in):	cont	the container
Header file:	#include "ara/core/span.h"	

▽

△

Description:	Construct a new Span from the given container. [ara::core::data(cont), ara::core::data(cont) + ara::core::size(cont)) shall be a valid range. If Extent is not equal to dynamic_extent, then ara::core::size(cont) shall be equal to Extent. These constructors shall not participate in overload resolution unless: Container is not a specialization of Span, Container is not a specialization of Array, std::is_array<Container>::value is false, ara::core::data(cont) and ara::core::size(cont) are both well-formed, and std::remove_pointer<decltype(ara::core::data(cont))>::type(*)[] is convertible to T(*)[].

⌋(*RS_AP_00130*)

[SWS_CORE_01948]{DRAFT} ⌈

Kind:	function	
Symbol:	Span(const Container &cont)	
Scope:	class ara::core::Span	
Syntax:	`template <typename Container>` `constexpr Span (const Container &cont);`	
Template param:	Container	the type of container
Parameters (in):	cont	the container
Header file:	#include "ara/core/span.h"	
Description:	Construct a new Span from the given const container. [ara::core::data(cont), ara::core::data(cont) + ara::core::size(cont)) shall be a valid range. If Extent is not equal to dynamic_extent, then ara::core::size(cont) shall be equal to Extent. These constructors shall not participate in overload resolution unless: Container is not a specialization of Span, Container is not a specialization of Array, std::is_array<Container>::value is false, ara::core::data(cont) and ara::core::size(cont) are both well-formed, and std::remove_pointer<decltype(ara::core::data(cont))>::type(*)[] is convertible to T(*)[].	

⌋(*RS_AP_00130*)

[SWS_CORE_01949]{DRAFT} ⌈

Kind:	function	
Symbol:	Span(const Span &other)	
Scope:	class ara::core::Span	
Syntax:	`constexpr Span (const Span &other) noexcept=default;`	
Parameters (in):	other	the other instance
Exception Safety:	noexcept	
Header file:	#include "ara/core/span.h"	
Description:	Copy construct a new Span from another instance.	

⌋(*RS_AP_00130*)

[SWS_CORE_01950]{DRAFT} ⌈

Kind:	function	
Symbol:	Span(const Span< U, N > &s)	
Scope:	class ara::core::Span	
Syntax:	`template <typename U, std::size_t N>` `constexpr Span (const Span< U, N > &s) noexcept;`	
Template param:	U	the type of elements within the other Span
	N	the Extent of the other Span
Parameters (in):	s	the other Span instance
Exception Safety:	noexcept	
Header file:	#include "ara/core/span.h"	
Description:	Converting constructor. This ctor allows construction of a cv-qualified Span from a normal Span, and also of a dynamic_extent-Span<> from a static extent-one.	

⌋(*RS_AP_00130*)

[SWS_CORE_01951]{DRAFT} ⌈

Kind:	function	
Symbol:	~Span()	
Scope:	class ara::core::Span	
Syntax:	`~Span () noexcept=default;`	
Exception Safety:	noexcept	
Header file:	#include "ara/core/span.h"	
Description:	Destructor.	

⌋(*RS_AP_00130*)

[SWS_CORE_01952]{DRAFT} ⌈

Kind:	function	
Symbol:	operator=(const Span &other)	
Scope:	class ara::core::Span	
Syntax:	`Span& operator= (const Span &other) noexcept=default;`	
Parameters (in):	other	the other instance
Return value:	Span &	*this
Exception Safety:	noexcept	
Header file:	#include "ara/core/span.h"	
Description:	Copy assignment operator.	
Notes:	This operator is not constexpr because that would make it implicitly const in C++11.	

⌋(*RS_AP_00130*)

[SWS_CORE_01961]{DRAFT} ⌈

Kind:	function	
Symbol:	first()	
Scope:	class ara::core::Span	
Syntax:	`template <std::size_t Count>` `constexpr Span<element_type, Count> first () const;`	
Template param:	Count	the number of elements to take over
Return value:	Span< element_type, Count >	the subspan
Header file:	#include "ara/core/span.h"	
Description:	Return a subspan containing only the first elements of this Span.	

⌋(*RS_AP_00130*)

[SWS_CORE_01962]{DRAFT} ⌈

Kind:	function	
Symbol:	first(index_type count)	
Scope:	class ara::core::Span	
Syntax:	`constexpr Span<element_type, dynamic_extent> first (index_type count) const;`	
Parameters (in):	count	the number of elements to take over
Return value:	Span< element_type, dynamic_extent >	the subspan
Header file:	#include "ara/core/span.h"	
Description:	Return a subspan containing only the first elements of this Span.	

⌋(*RS_AP_00130*)

[SWS_CORE_01963]{DRAFT} ⌈

Kind:	function	
Symbol:	last()	
Scope:	class ara::core::Span	
Syntax:	`template <std::size_t Count>` `constexpr Span<element_type, Count> last () const;`	
Template param:	Count	the number of elements to take over
Return value:	Span< element_type, Count >	the subspan
Header file:	#include "ara/core/span.h"	
Description:	Return a subspan containing only the last elements of this Span.	

⌋(*RS_AP_00130*)

[SWS_CORE_01964]{DRAFT} ⌈

Kind:	function	
Symbol:	last(index_type count)	
Scope:	class ara::core::Span	
Syntax:	`constexpr Span<element_type, dynamic_extent> last (index_type count) const;`	
Parameters (in):	count	the number of elements to take over
Return value:	Span< element_type, dynamic_extent >	the subspan
Header file:	#include "ara/core/span.h"	
Description:	Return a subspan containing only the last elements of this Span.	

⌋(*RS_AP_00130*)

[SWS_CORE_01965]{DRAFT} ⌈

Kind:	function	
Symbol:	subspan()	
Scope:	class ara::core::Span	
Syntax:	`template <std::size_t Offset, std::size_t Count = dynamic_extent> constexpr auto subspan () const -> Span< element_type, <see below> >;`	
Template param:	Offset	offset into this Span from which to start
	Count	the number of elements to take over
Return value:	Span< element_type, <see below> >	the subspan
Header file:	#include "ara/core/span.h"	
Description:	Return a subspan of this Span. The second template argument of the returned Span type is: Count != dynamic_extent ? Count : (Extent != dynamic_extent ? Extent - Offset : dynamic_extent)	

⌋(*RS_AP_00130*)

[SWS_CORE_01966]{DRAFT} ⌈

Kind:	function	
Symbol:	subspan(index_type offset, index_type count=dynamic_extent)	
Scope:	class ara::core::Span	
Syntax:	`constexpr Span<element_type, dynamic_extent> subspan (index_type offset, index_type count=dynamic_extent) const;`	
Parameters (in):	offset	offset into this Span from which to start
	count	the number of elements to take over
Return value:	Span< element_type, dynamic_extent >	the subspan
Header file:	#include "ara/core/span.h"	
Description:	Return a subspan of this Span.	

⌋(*RS_AP_00130*)

[SWS_CORE_01967]{DRAFT} ⌈

Kind:	function	
Symbol:	size()	
Scope:	class ara::core::Span	
Syntax:	`constexpr index_type size () const noexcept;`	
Return value:	index_type	the number of elements contained in this Span
Exception Safety:	noexcept	
Header file:	#include "ara/core/span.h"	
Description:	Return the size of this Span.	

⌋(*RS_AP_00130*)

[SWS_CORE_01968]{DRAFT} ⌈

Kind:	function	
Symbol:	size_bytes()	
Scope:	class ara::core::Span	
Syntax:	`constexpr index_type size_bytes () const noexcept;`	
Return value:	index_type	the number of bytes covered by this Span
Exception Safety:	noexcept	
Header file:	#include "ara/core/span.h"	
Description:	Return the size of this Span in bytes.	

⌋(*RS_AP_00130*)

[SWS_CORE_01969]{DRAFT} ⌈

Kind:	function	
Symbol:	empty()	
Scope:	class ara::core::Span	
Syntax:	`constexpr bool empty () const noexcept;`	
Return value:	bool	true if this Span contains 0 elements, false otherwise
Exception Safety:	noexcept	
Header file:	#include "ara/core/span.h"	
Description:	Return whether this Span is empty.	

⌋(*RS_AP_00130*)

[SWS_CORE_01970]{DRAFT} ⌈

Kind:	function
Symbol:	operator[](index_type idx)
Scope:	class ara::core::Span
Syntax:	`constexpr reference operator[] (index_type idx) const;`

▽

△

Parameters (in):	idx	the index into this Span
Return value:	reference	the reference
Header file:	#include "ara/core/span.h"	
Description:	Return a reference to the n-th element of this Span.	

⌋(*RS_AP_00130*)

[SWS_CORE_01971]{DRAFT} ⌈

Kind:	function	
Symbol:	data()	
Scope:	class ara::core::Span	
Syntax:	`constexpr pointer data () const noexcept;`	
Return value:	pointer	the pointer
Exception Safety:	noexcept	
Header file:	#include "ara/core/span.h"	
Description:	Return a pointer to the start of the memory block covered by this Span.	

⌋(*RS_AP_00130*)

[SWS_CORE_01972]{DRAFT} ⌈

Kind:	function	
Symbol:	begin()	
Scope:	class ara::core::Span	
Syntax:	`constexpr iterator begin () const noexcept;`	
Return value:	iterator	the iterator
Exception Safety:	noexcept	
Header file:	#include "ara/core/span.h"	
Description:	Return an iterator pointing to the first element of this Span.	

⌋(*RS_AP_00130*)

[SWS_CORE_01973]{DRAFT} ⌈

Kind:	function	
Symbol:	end()	
Scope:	class ara::core::Span	
Syntax:	`constexpr iterator end () const noexcept;`	
Return value:	iterator	the iterator
Exception Safety:	noexcept	
Header file:	#include "ara/core/span.h"	
Description:	Return an iterator pointing past the last element of this Span.	

⌋(*RS_AP_00130*)

[SWS_CORE_01974]{DRAFT} ⌈

Kind:	function	
Symbol:	cbegin()	
Scope:	class ara::core::Span	
Syntax:	`constexpr const_iterator cbegin () const noexcept;`	
Return value:	const_iterator	the const_iterator
Exception Safety:	noexcept	
Header file:	#include "ara/core/span.h"	
Description:	Return a const_iterator pointing to the first element of this Span.	

⌋(*RS_AP_00130*)

[SWS_CORE_01975]{DRAFT} ⌈

Kind:	function	
Symbol:	cend()	
Scope:	class ara::core::Span	
Syntax:	`constexpr const_iterator cend () const noexcept;`	
Return value:	const_iterator	the const_iterator
Exception Safety:	noexcept	
Header file:	#include "ara/core/span.h"	
Description:	Return a const_iterator pointing past the last element of this Span.	

⌋(*RS_AP_00130*)

[SWS_CORE_01976]{DRAFT} ⌈

Kind:	function	
Symbol:	rbegin()	
Scope:	class ara::core::Span	
Syntax:	`constexpr reverse_iterator rbegin () const noexcept;`	
Return value:	reverse_iterator	the reverse_iterator
Exception Safety:	noexcept	
Header file:	#include "ara/core/span.h"	
Description:	Return a reverse_iterator pointing to the last element of this Span.	

⌋(*RS_AP_00130*)

[SWS_CORE_01977]{DRAFT} ⌈

Kind:	function
Symbol:	rend()
Scope:	class ara::core::Span
Syntax:	`constexpr reverse_iterator rend () const noexcept;`

▽

△

Return value:	reverse_iterator	the reverse_iterator
Exception Safety:	noexcept	
Header file:	#include "ara/core/span.h"	
Description:	Return a reverse_iterator pointing past the first element of this Span.	

⌋(*RS_AP_00130*)

[SWS_CORE_01978]{DRAFT} ⌈

Kind:	function	
Symbol:	crbegin()	
Scope:	class ara::core::Span	
Syntax:	`constexpr const_reverse_iterator crbegin () const noexcept;`	
Return value:	const_reverse_iterator	the const_reverse_iterator
Exception Safety:	noexcept	
Header file:	#include "ara/core/span.h"	
Description:	Return a const_reverse_iterator pointing to the last element of this Span.	

⌋(*RS_AP_00130*)

[SWS_CORE_01979]{DRAFT} ⌈

Kind:	function	
Symbol:	crend()	
Scope:	class ara::core::Span	
Syntax:	`constexpr const_reverse_iterator crend () const noexcept;`	
Return value:	const_reverse_iterator	the reverse_iterator
Exception Safety:	noexcept	
Header file:	#include "ara/core/span.h"	
Description:	Return a const_reverse_iterator pointing past the first element of this Span.	

⌋(*RS_AP_00130*)

Some global factory functions for `ara::core::Span` allow to create instances without explicitly mentioning the template parameter type – this type is being deduced from the functions' arguments:

[SWS_CORE_01990]{DRAFT} ⌈

Kind:	function
Symbol:	MakeSpan(T *ptr, typename Span< T >::index_type count)
Scope:	namespace ara::core

▽

△

Syntax:	`template <typename T>` `constexpr Span<T> MakeSpan (T *ptr, typename Span< T >::index_type count);`	
Template param:	T	the type of elements
Parameters (in):	ptr	the pointer
	count	the number of elements to take from ptr
Return value:	Span< T >	the new Span
Header file:	#include "ara/core/span.h"	
Description:	Create a new Span from the given pointer and size.	

⌋(*RS_AP_00130*)

[SWS_CORE_01991]{DRAFT} ⌈

Kind:	function	
Symbol:	MakeSpan(T *firstElem, T *lastElem)	
Scope:	namespace ara::core	
Syntax:	`template <typename T>` `constexpr Span<T> MakeSpan (T *firstElem, T *lastElem);`	
Template param:	T	the type of elements
Parameters (in):	firstElem	pointer to the first element
	lastElem	pointer to past the last element
Return value:	Span< T >	the new Span
Header file:	#include "ara/core/span.h"	
Description:	Create a new Span from the open range between [firstElem, lastElem).	

⌋(*RS_AP_00130*)

[SWS_CORE_01992]{DRAFT} ⌈

Kind:	function	
Symbol:	MakeSpan(T(&arr)[N])	
Scope:	namespace ara::core	
Syntax:	`template <typename T, std::size_t N>` `constexpr Span<T, N> MakeSpan (T(&arr)[N]) noexcept;`	
Template param:	T	the type of elements
	N	the size of the raw array
Parameters (in):	arr	the raw array
Return value:	Span< T, N >	the new Span
Exception Safety:	noexcept	
Header file:	#include "ara/core/span.h"	
Description:	Create a new Span from the given raw array.	

⌋(*RS_AP_00130*)

[SWS_CORE_01993]{DRAFT} ⌈

Kind:	function	
Symbol:	MakeSpan(Container &cont)	
Scope:	namespace ara::core	
Syntax:	`template <typename Container>` `constexpr Span<typename Container::value_type> MakeSpan (Container &cont);`	
Template param:	Container	the type of container
Parameters (in):	cont	the container
Return value:	Span< typename Container::value_type >	the new Span
Header file:	#include "ara/core/span.h"	
Description:	Create a new Span from the given container.	

⌋(*RS_AP_00130*)

[SWS_CORE_01994]{DRAFT} ⌈

Kind:	function	
Symbol:	MakeSpan(const Container &cont)	
Scope:	namespace ara::core	
Syntax:	`template <typename Container>` `constexpr Span<typename Container::value_type const> MakeSpan (const Container &cont);`	
Template param:	Container	the type of container
Parameters (in):	cont	the container
Return value:	Span< typename Container::value_type const >	the new Span
Header file:	#include "ara/core/span.h"	
Description:	Create a new Span from the given const container.	

⌋(*RS_AP_00130*)

These global functions allow to “convert” a `Span<T>` into a `Span<Byte>`, thereby gaining access to the in-memory representation of the object referenced by a `Span` instance.

Unlike `std::byte` from [10, the C++17 standard], it is implementation-defined whether `ara::core::Byte` can be used for type aliasing without triggering Undefined Behavior. This may also affect `ara::core::as_bytes` and `ara::core::as_writable_bytes` in particular. Implementations usually provide a way to make this safe by loosening the aliasing restrictions of the C++ compiler.

[SWS_CORE_01980]{DRAFT} ⌈

Kind:	function	
Symbol:	as_bytes(Span< ElementType, Extent > s)	
Scope:	namespace ara::core	
Syntax:	`template <typename ElementType, std::size_t Extent> Span<const Byte, Extent == dynamic_extent ? dynamic_extent : sizeof(ElementType) * Extent> as_bytes (Span< ElementType, Extent > s) noexcept;`	
Parameters (in):	s	the input Span<T>
Return value:	Span< const Byte, Extent==dynamic_ extent ? dynamic_extent :sizeof(ElementType) *Extent >	a Span<const Byte>
Exception Safety:	noexcept	
Header file:	#include "ara/core/span.h"	
Description:	Return a read-only Span<Byte> over the object representation of the input Span<T>	

⌋(*RS_AP_00130*)

[SWS_CORE_01981]{DRAFT} ⌈

Kind:	function	
Symbol:	as_writable_bytes(Span< ElementType, Extent > s)	
Scope:	namespace ara::core	
Syntax:	`template <typename ElementType, std::size_t Extent> Span<Byte, Extent == dynamic_extent ? dynamic_extent : sizeof(ElementType) * Extent> as_writable_bytes (Span< ElementType, Extent > s) noexcept;`	
Parameters (in):	s	the input Span<T>
Return value:	Span< Byte, Extent==dynamic_extent ? dynamic_extent :sizeof(Element Type) *Extent >	a Span<Byte>
Exception Safety:	noexcept	
Header file:	#include "ara/core/span.h"	
Description:	Return a writable Span<Byte> over the object representation of the input Span<T>	

⌋(*RS_AP_00130*)

8.1.15 `SteadyClock` data type

[SWS_CORE_06401]{DRAFT} ⌈

Kind:	class
Symbol:	SteadyClock
Scope:	namespace ara::core
Syntax:	`class SteadyClock final {...};`
Header file:	#include "ara/core/steady_clock.h"

▽

△

Description:	This clock represents a monotonic clock. The time points of this clock cannot decrease as physical time moves forward and the time between ticks of this clock is constant.

⌋(*RS_AP_00130*)

[SWS_CORE_06412]{DRAFT} ⌈

Kind:	type alias
Symbol:	rep
Scope:	class ara::core::SteadyClock
Derived from:	std::int64_t
Syntax:	`using rep = std::int64_t;`
Header file:	#include "ara/core/steady_clock.h"
Description:	Arithmetic type representing the number of ticks in the clock's duration .

⌋(*RS_AP_00130*)

[SWS_CORE_06413]{DRAFT} ⌈

Kind:	type alias
Symbol:	period
Scope:	class ara::core::SteadyClock
Derived from:	std::nano
Syntax:	`using period = std::nano;`
Header file:	#include "ara/core/steady_clock.h"
Description:	A std::ratio type representing the tick period of the clock, in seconds .

⌋(*RS_AP_00130*)

[SWS_CORE_06411]{DRAFT} ⌈

Kind:	type alias
Symbol:	duration
Scope:	class ara::core::SteadyClock
Derived from:	std::chrono::duration<rep, period>
Syntax:	`using duration = std::chrono::duration<rep, period>;`
Header file:	#include "ara/core/steady_clock.h"
Description:	std::chrono::duration<rep, period>

⌋(*RS_AP_00130*)

[SWS_CORE_06414]{DRAFT} ⌈

Kind:	type alias
Symbol:	time_point
Scope:	class ara::core::SteadyClock
Derived from:	std::chrono::time_point<SteadyClock, duration>
Syntax:	`using time_point = std::chrono::time_point<SteadyClock, duration>;`
Header file:	#include "ara/core/steady_clock.h"
Description:	std::chrono::time_point<ara::core::SteadyClock>

⌋(*RS_AP_00130*)

[SWS_CORE_06431]{DRAFT} ⌈

Kind:	variable
Symbol:	is_steady
Scope:	class ara::core::SteadyClock
Type:	bool
Syntax:	`constexpr bool is_steady = true;`
Header file:	#include "ara/core/steady_clock.h"
Description:	steady clock flag, always true

⌋(*RS_AP_00130*)

[SWS_CORE_06432]{DRAFT} ⌈

Kind:	function	
Symbol:	now()	
Scope:	class ara::core::SteadyClock	
Syntax:	`static time_point now () noexcept;`	
Return value:	time_point	a time_point
Exception Safety:	noexcept	
Header file:	#include "ara/core/steady_clock.h"	
Description:	Return a time_point representing the current value of the clock.	

⌋(*RS_AP_00130*)

8.1.16 `InstanceSpecifier` data type

This section defines the `ara::core::InstanceSpecifier` type that describes the path to a meta model element.

[SWS_CORE_08001] ⌈

Kind:	class
Symbol:	InstanceSpecifier
Scope:	namespace ara::core
Syntax:	`class InstanceSpecifier final {...};`
Header file:	#include "ara/core/instance_specifier.h"
Description:	class representing an AUTOSAR Instance Specifier, which is basically an AUTOSAR shortname-path wrapper.

⌋(*RS_AP_00140*, *RS_Main_00320*)

[SWS_CORE_08021] ⌈

Kind:	function	
Symbol:	InstanceSpecifier(StringView metaModelIdentifier)	
Scope:	class ara::core::InstanceSpecifier	
Syntax:	`explicit InstanceSpecifier (StringView metaModelIdentifier);`	
Parameters (in):	metaModelIdentifier	stringified meta model identifier (short name path) where path separator is '/'. Lifetime of underlying string has to exceed the lifetime of the constructed InstanceSpecifier.
Exceptions:	CoreException	in case the given metaModelIdentifier is not a valid meta-model identifier/short name path.
Header file:	#include "ara/core/instance_specifier.h"	
Description:	throwing ctor from meta-model string	

⌋(*RS_Main_00320*)

[SWS_CORE_08022]{DRAFT} ⌈

Kind:	function	
Symbol:	InstanceSpecifier(const InstanceSpecifier &other)	
Scope:	class ara::core::InstanceSpecifier	
Syntax:	`InstanceSpecifier (const InstanceSpecifier &other);`	
Parameters (in):	other	the other instance
Header file:	#include "ara/core/instance_specifier.h"	
Description:	Copy constructor.	

⌋(*RS_Main_00320*)

[SWS_CORE_08023]{DRAFT} ⌈

Kind:	function
Symbol:	InstanceSpecifier(InstanceSpecifier &&other)
Scope:	class ara::core::InstanceSpecifier
Syntax:	`InstanceSpecifier (InstanceSpecifier &&other) noexcept;`

▽

△

Parameters (in):	other	the other instance
Exception Safety:	noexcept	
Header file:	#include "ara/core/instance_specifier.h"	
Description:	Move constructor.	

⌋(*RS_Main_00320*)

[SWS_CORE_08024]{DRAFT} ⌈

Kind:	function	
Symbol:	operator=(const InstanceSpecifier &other)	
Scope:	class ara::core::InstanceSpecifier	
Syntax:	`InstanceSpecifier& operator= (const InstanceSpecifier &other);`	
Parameters (in):	other	the other instance
Return value:	InstanceSpecifier &	*this
Header file:	#include "ara/core/instance_specifier.h"	
Description:	Copy assignment operator.	

⌋(*RS_Main_00320*)

[SWS_CORE_08025]{DRAFT} ⌈

Kind:	function	
Symbol:	operator=(InstanceSpecifier &&other)	
Scope:	class ara::core::InstanceSpecifier	
Syntax:	`InstanceSpecifier& operator= (InstanceSpecifier &&other);`	
Parameters (in):	other	the other instance
Return value:	InstanceSpecifier &	*this
Header file:	#include "ara/core/instance_specifier.h"	
Description:	Move assignment operator.	

⌋(*RS_Main_00320*)

[SWS_CORE_08029] ⌈

Kind:	function
Symbol:	~InstanceSpecifier()
Scope:	class ara::core::InstanceSpecifier
Syntax:	`~InstanceSpecifier () noexcept;`
Exception Safety:	noexcept
Header file:	#include "ara/core/instance_specifier.h"
Description:	Destructor.

⌋(*RS_AP_00134*, *RS_Main_00320*)

[SWS_CORE_08032] ⌈

Kind:	function	
Symbol:	Create(StringView metaModelIdentifier)	
Scope:	class ara::core::InstanceSpecifier	
Syntax:	`static Result<InstanceSpecifier> Create (StringView metaModel Identifier);`	
Parameters (in):	metaModelIdentifier	stringified form of InstanceSpecifier
Return value:	Result< InstanceSpecifier >	a Result, containing either a syntactically valid InstanceSpecifier, or an ErrorCode
Errors:	CoreErrc::kInvalidMetaModel Shortname	if any of the path elements of metaModelIdentifier is missing or contains invalid characters
	CoreErrc::kInvalidMetaModelPath	if the metaModelIdentifier is not a valid path to a model element
Header file:	#include "ara/core/instance_specifier.h"	
Description:	Create a new instance of this class.	

⌋(*RS_Main_00150*, *RS_AP_00137*, *RS_AP_00136*)

[SWS_CORE_08042] ⌈

Kind:	function	
Symbol:	operator==(const InstanceSpecifier &other)	
Scope:	class ara::core::InstanceSpecifier	
Syntax:	`bool operator== (const InstanceSpecifier &other) const noexcept;`	
Parameters (in):	other	InstanceSpecifier instance to compare this one with.
Return value:	bool	true in case both InstanceSpecifiers are denoting exactly the same model element, false else.
Exception Safety:	noexcept	
Header file:	#include "ara/core/instance_specifier.h"	
Description:	eq operator to compare with other InstanceSpecifier instance.	

⌋(*RS_Main_00320*)

[SWS_CORE_08043] ⌈

Kind:	function	
Symbol:	operator==(StringView other)	
Scope:	class ara::core::InstanceSpecifier	
Syntax:	`bool operator== (StringView other) const noexcept;`	
Parameters (in):	other	string representation to compare this one with.
Return value:	bool	true in case this InstanceSpecifiers is denoting exactly the same model element as other, false else.
Exception Safety:	noexcept	
Header file:	#include "ara/core/instance_specifier.h"	
Description:	eq operator to compare with other InstanceSpecifier instance.	

⌋(*RS_Main_00320*)

[SWS_CORE_08044] ⌈

Kind:	function	
Symbol:	operator!=(const InstanceSpecifier &other)	
Scope:	class ara::core::InstanceSpecifier	
Syntax:	`bool operator!= (const InstanceSpecifier &other) const noexcept;`	
Parameters (in):	other	InstanceSpecifier instance to compare this one with.
Return value:	bool	false in case both InstanceSpecifiers are denoting exactly the same model element, true else.
Exception Safety:	noexcept	
Header file:	#include "ara/core/instance_specifier.h"	
Description:	uneq operator to compare with other InstanceSpecifier instance.	

⌋(*RS_Main_00320*)

[SWS_CORE_08045] ⌈

Kind:	function	
Symbol:	operator!=(StringView other)	
Scope:	class ara::core::InstanceSpecifier	
Syntax:	`bool operator!= (StringView other) const noexcept;`	
Parameters (in):	other	string representation to compare this one with.
Return value:	bool	false in case this InstanceSpecifiers is denoting exactly the same model element as other, true else.
Exception Safety:	noexcept	
Header file:	#include "ara/core/instance_specifier.h"	
Description:	uneq operator to compare with other InstanceSpecifier string representation.	

⌋(*RS_Main_00320*)

[SWS_CORE_08046] ⌈

Kind:	function	
Symbol:	operator<(const InstanceSpecifier &other)	
Scope:	class ara::core::InstanceSpecifier	
Syntax:	`bool operator< (const InstanceSpecifier &other) const noexcept;`	
Parameters (in):	other	InstanceSpecifier instance to compare this one with.
Return value:	bool	true in case this InstanceSpecifiers is lexically lower than other, false else.
Exception Safety:	noexcept	
Header file:	#include "ara/core/instance_specifier.h"	
Description:	lower than operator to compare with other InstanceSpecifier for ordering purposes (f.i. when collecting identifiers in maps).	

⌋(*RS_Main_00320*)

[SWS_CORE_08041] ⌈

Kind:	function	
Symbol:	ToString()	
Scope:	class ara::core::InstanceSpecifier	
Syntax:	`StringView ToString () const noexcept;`	
Return value:	StringView	stringified form of InstanceSpecifier. Lifetime of the underlying string is only guaranteed for the lifetime of the underlying string of the StringView passed to the constructor.
Exception Safety:	noexcept	
Header file:	#include "ara/core/instance_specifier.h"	
Description:	method to return the stringified form of InstanceSpecifier	

⌋(*RS_Main_00320*)

[SWS_CORE_08081]{DRAFT} ⌈

Kind:	function	
Symbol:	operator==(StringView lhs, const InstanceSpecifier &rhs)	
Scope:	namespace ara::core	
Syntax:	`bool operator== (StringView lhs, const InstanceSpecifier &rhs) noexcept;`	
Parameters (in):	lhs	stringified form of a InstanceSpecifier
	rhs	an InstanceSpecifier
Return value:	bool	true in case rhs string representation equals lhs
Exception Safety:	noexcept	
Header file:	#include "ara/core/instance_specifier.h"	
Description:	Non-member function operator== to allow StringView on lhs.	

⌋(*RS_Main_00320*)

[SWS_CORE_08082]{DRAFT} ⌈

Kind:	function	
Symbol:	operator!=(StringView lhs, const InstanceSpecifier &rhs)	
Scope:	namespace ara::core	
Syntax:	`bool operator!= (StringView lhs, const InstanceSpecifier &rhs) noexcept;`	
Parameters (in):	lhs	stringified form of a InstanceSpecifier
	rhs	an InstanceSpecifier
Return value:	bool	true in case rhs string representation not equals lhs
Exception Safety:	noexcept	
Header file:	#include "ara/core/instance_specifier.h"	
Description:	Non-member function operator== to allow StringView on lhs.	

⌋(*RS_Main_00320*)

8.1.17 Generic helpers

8.1.17.1 ara::core::Byte

The exact setup of this type is implementation-defined; the specifications in section 7.2.4.3.1 ("ara::core::Byte") define the expected behavior.

[SWS_CORE_04200] ⌈

Kind:	type alias
Symbol:	Byte
Scope:	namespace ara::core
Derived from:	<implementation-defined>
Syntax:	`using Byte = <implementation-defined>;`
Header file:	#include "ara/core/utility.h"
Description:	A non-integral binary type.

⌋*(RS_AP_00130)*

8.1.17.2 In-place disambiguation tags

The data types `ara::core::in_place_t`, `ara::core::in_place_type_t`, and `ara::core::in_place_index_t` are disambiguation tags that can be passed to certain constructors of `ara::core::Optional` and `ara::core::Variant` to indicate that the contained type shall be constructed in-place, i.e. without any copy operation taking place.

They are equivalent to `std::in_place_t`, `std::in_place_type_t`, and `std::in_place_index_t` from [10], except that no variable templates are being defined, because they are not supported by [5, the C++11 standard]. All these symbols are provided here in order to give the necessary support for implementing `ara::core::Optional` and `ara::core::Variant` in a way that is highly compatible with the corresponding classes from [10, the C++17 standard].

8.1.17.2.1 `in_place_t` tag

[SWS_CORE_04011] ⌈

Kind:	struct
Symbol:	in_place_t
Scope:	namespace ara::core

▽

△

Syntax:	`struct in_place_t {...};`
Header file:	#include "ara/core/utility.h"
Description:	Denote an operation to be performed in-place. An instance of this type can be passed to certain constructors of ara::core::Optional to denote the intention that construction of the contained type shall be done in-place, i.e. without any copying taking place.

⌋(*RS_AP_00130*)

[SWS_CORE_04012] ⌈

Kind:	function
Symbol:	in_place_t()
Scope:	struct ara::core::in_place_t
Syntax:	`explicit in_place_t ()=default;`
Header file:	#include "ara/core/utility.h"
Description:	Default constructor.

⌋(*RS_AP_00130*)

[SWS_CORE_04013] ⌈

Kind:	variable
Symbol:	in_place
Scope:	namespace ara::core
Type:	in_place_t
Syntax:	`constexpr in_place_t in_place;`
Header file:	#include "ara/core/utility.h"
Description:	The singleton instance of in_place_t.

⌋(*RS_AP_00130*)

8.1.17.2.2 `in_place_type_t` tag

[SWS_CORE_04021] ⌈

Kind:	struct	
Symbol:	in_place_type_t	
Scope:	namespace ara::core	
Syntax:	`template <typename T>` `struct in_place_type_t {...};`	
Template param:	typename T	–

▽

△

Header file:	#include "ara/core/utility.h"
Description:	Denote a type-distinguishing operation to be performed in-place. An instance of this type can be passed to certain constructors of ara::core::Variant to denote the intention that construction of the contained type shall be done in-place, i.e. without any copying taking place.

⌋(*RS_AP_00130*)

[SWS_CORE_04022] ⌈

Kind:	function
Symbol:	in_place_type_t()
Scope:	struct ara::core::in_place_type_t
Syntax:	`explicit in_place_type_t ()=default;`
Header file:	#include "ara/core/utility.h"
Description:	Default constructor.

⌋(*RS_AP_00130*)

[SWS_CORE_04023]{DRAFT} ⌈

Kind:	variable	
Symbol:	in_place_type	
Scope:	namespace ara::core	
Type:	in_place_type_t< T >	
Syntax:	`template <typename T>` `constexpr in_place_type_t<T> in_place_type;`	
Template param:	typename T	the type to address
Header file:	#include "ara/core/utility.h"	
Description:	The singleton instances (one for each T) of in_place_type_t.	

⌋(*RS_AP_00130*)

8.1.17.2.3 `in_place_index_t` tag

[SWS_CORE_04031] ⌈

Kind:	struct
Symbol:	in_place_index_t
Scope:	namespace ara::core
Syntax:	`template <size_t I>` `struct in_place_index_t {...};`

▽

△

Template param:	size_t I	–
Header file:	#include "ara/core/utility.h"	
Description:	Denote an index-distinguishing operation to be performed in-place. An instance of this type can be passed to certain constructors of ara::core::Variant to denote the intention that construction of the contained type shall be done in-place, i.e. without any copying taking place.	

⌋(*RS_AP_00130*)

[SWS_CORE_04032] ⌈

Kind:	function
Symbol:	in_place_index_t()
Scope:	struct ara::core::in_place_index_t
Syntax:	`explicit in_place_index_t ()=default;`
Header file:	#include "ara/core/utility.h"
Description:	Default constructor.

⌋(*RS_AP_00130*)

[SWS_CORE_04033]{DRAFT} ⌈

Kind:	variable	
Symbol:	in_place_index	
Scope:	namespace ara::core	
Type:	in_place_index_t< I >	
Syntax:	`template <std::size_t I>` `constexpr in_place_index_t<I> in_place_index {};`	
Template param:	std::size_t I	the index to address
Header file:	#include "ara/core/utility.h"	
Description:	The singleton instances (one for each I) of in_place_index_t.	

⌋(*RS_AP_00130*)

8.1.17.3 Non-member container access

These global functions allow uniform access to the data and size properties of contiguous containers.

They are equivalent to `std::data`, `std::size`, and `std::empty` from [10].

[SWS_CORE_04110] ⌈

Kind:	function	
Symbol:	data(Container &c)	
Scope:	namespace ara::core	
Syntax:	`template <typename Container>` `constexpr auto data (Container &c) -> decltype(c.data());`	
Template param:	Container	a type with a data() method
Parameters (in):	c	an instance of Container
Return value:	decltype(c.data())	a pointer to the first element of the container
Header file:	#include "ara/core/utility.h"	
Description:	Return a pointer to the block of memory that contains the elements of a container.	

⌋(*RS_AP_00130*)

[SWS_CORE_04111] ⌈

Kind:	function	
Symbol:	data(const Container &c)	
Scope:	namespace ara::core	
Syntax:	`template <typename Container>` `constexpr auto data (const Container &c) -> decltype(c.data());`	
Template param:	Container	a type with a data() method
Parameters (in):	c	an instance of Container
Return value:	decltype(c.data())	a pointer to the first element of the container
Header file:	#include "ara/core/utility.h"	
Description:	Return a const_pointer to the block of memory that contains the elements of a container.	

⌋(*RS_AP_00130*)

[SWS_CORE_04112] ⌈

Kind:	function	
Symbol:	data(T(&array)[N])	
Scope:	namespace ara::core	
Syntax:	`template <typename T, std::size_t N>` `constexpr T* data (T(&array)[N]) noexcept;`	
Template param:	T	the type of array elements
	N	the number of elements in the array
Parameters (in):	array	reference to a raw array
Return value:	T *	a pointer to the first element of the array
Exception Safety:	noexcept	
Header file:	#include "ara/core/utility.h"	
Description:	Return a pointer to the block of memory that contains the elements of a raw array.	

⌋(*RS_AP_00130*)

[SWS_CORE_04113] ⌈

Kind:	function	
Symbol:	data(std::initializer_list< E > il)	
Scope:	namespace ara::core	
Syntax:	`template <typename E>` `constexpr const E* data (std::initializer_list< E > il) noexcept;`	
Template param:	E	the type of elements in the std::initializer_list
Parameters (in):	il	the std::initializer_list
Return value:	const E *	a pointer to the first element of the std::initializer_list
Exception Safety:	noexcept	
Header file:	#include "ara/core/utility.h"	
Description:	Return a pointer to the block of memory that contains the elements of a std::initializer_list.	

⌋(*RS_AP_00130*)

[SWS_CORE_04120] ⌈

Kind:	function	
Symbol:	size(const Container &c)	
Scope:	namespace ara::core	
Syntax:	`template <typename Container>` `constexpr auto size (const Container &c) -> decltype(c.size());`	
Template param:	Container	a type with a data() method
Parameters (in):	c	an instance of Container
Return value:	decltype(c.size())	the size of the container
Header file:	#include "ara/core/utility.h"	
Description:	Return the size of a container.	

⌋(*RS_AP_00130*)

[SWS_CORE_04121] ⌈

Kind:	function	
Symbol:	size(const T(&array)[N])	
Scope:	namespace ara::core	
Syntax:	`template <typename T, std::size_t N>` `constexpr std::size_t size (const T(&array)[N]) noexcept;`	
Template param:	T	the type of array elements
	N	the number of elements in the array
Parameters (in):	array	reference to a raw array
Return value:	std::size_t	the size of the array, i.e. N
Exception Safety:	noexcept	
Header file:	#include "ara/core/utility.h"	
Description:	Return the size of a raw array.	

⌋(*RS_AP_00130*)

[SWS_CORE_04130] ⌈

Kind:	function	
Symbol:	empty(const Container &c)	
Scope:	namespace ara::core	
Syntax:	`template <typename Container>` `constexpr auto empty (const Container &c) -> decltype(c.empty());`	
Template param:	Container	a type with a empty() method
Parameters (in):	c	an instance of Container
Return value:	decltype(c.empty())	true if the container is empty, false otherwise
Header file:	#include "ara/core/utility.h"	
Description:	Return whether the given container is empty.	

⌋(*RS_AP_00130*)

[SWS_CORE_04131] ⌈

Kind:	function	
Symbol:	empty(const T(&array)[N])	
Scope:	namespace ara::core	
Syntax:	`template <typename T, std::size_t N>` `constexpr bool empty (const T(&array)[N]) noexcept;`	
Template param:	T	the type of array elements
	N	the number of elements in the array
Parameters (in):	array	the raw array
Return value:	bool	false
Exception Safety:	noexcept	
Header file:	#include "ara/core/utility.h"	
Description:	Return whether the given raw array is empty. As raw arrays cannot have zero elements in C++, this function always returns false.	

⌋(*RS_AP_00130*)

[SWS_CORE_04132] ⌈

Kind:	function	
Symbol:	empty(std::initializer_list< E > il)	
Scope:	namespace ara::core	
Syntax:	`template <typename E>` `constexpr bool empty (std::initializer_list< E > il) noexcept;`	
Template param:	E	the type of elements in the std::initializer_list
Parameters (in):	il	the std::initializer_list
Return value:	bool	true if the std::initializer_list is empty, false otherwise
Exception Safety:	noexcept	
Header file:	#include "ara/core/utility.h"	
Description:	Return whether the given std::initializer_list is empty.	

⌋(*RS_AP_00130*)

8.1.18 Initialization and Shutdown

This section describes the global initialization and shutdown functions that initialize resp. deinitialize data structures and threads of the AUTOSAR Runtime for Adaptive Applications.

[SWS_CORE_10001]{DRAFT} ⌈

Kind:	function	
Symbol:	Initialize()	
Scope:	namespace ara::core	
Syntax:	`Result<void> Initialize () noexcept;`	
Return value:	ara::core::Result< void >	A Result object that indicates whether the AUTOSAR Adaptive Runtime for Applications was successfully initialized. Note that this is the only way for the ARA to report an error that is guaranteed to be available, e.g., in case ara::log failed to correctly initialize. The user is not expected to be able to recover from such an error. However, the user may have a project-specific way of recording errors during initialization without ara::log.
Exception Safety:	noexcept	
Header file:	#include "ara/core/initialization.h"	
Description:	Initializes data structures and threads of the AUTOSAR Adaptive Runtime for Applications. Prior to this call, no interaction with the ARA is possible. This call must be made inside of main(), i.e., in a place where it is guaranteed that static memory initialization has completed. Depending on the individual functional cluster specification, the calling application may have to provide additional configuration data (e.g., set an Application ID for Logging) or make additional initailization calls (e.g., start a FindService in ara::com) before other API calls to the respective functional cluster can be made. Such calls must be made after the call to Initialize(). Calls to ARA APIs made before static initialization has completed lead to undefinded behavior. Calls made after static initialization has completed but before Initialize() was called will be rejected by the functional cluster implementation with an error or, if no error to be reported is defined, lead to undefined behavior.	

⌋*(RS_Main_00011)*

[SWS_CORE_10002]{DRAFT} ⌈

Kind:	function	
Symbol:	Deinitialize()	
Scope:	namespace ara::core	
Syntax:	`Result<void> Deinitialize () noexcept;`	
Return value:	ara::core::Result< void >	A Result object that indicates whether the ARA was successfully destroyed. Typical error cases to be reported here are that the user is still holding some resource inside the ARA. Note that this Result is the only way for the ARA to report an error that is guaranteed to be available, e.g., in case ara::log has already been deinitialized. The user is not expected to be able to recover from such an error. However, the user may have a project-specific way of recording errors during deinitialization without ara::log.

▽

△

Exception Safety:	noexcept
Header file:	#include "ara/core/initialization.h"
Description:	Destroy all data structures and threads of the AUTOSAR Adaptive Runtime for Applications. After this call, no interaction with the ARA is possible. This call must be made inside of main(), i.e., in a place where it is guaranteed that the static initialization has completed and destruciton of statically initialized data has not yet started. Calls made to ARA APIs after a call to ara::core::Deinitialize() but before destruction of statically initialized data will be rejected with an error or, if no error is defined, lead to undefined behavior. Calls made to ARA APIs after the destruction of statically initialized data will lead to undefined behavior.

⌋(*RS_Main_00011*)

8.1.19 Abnormal process termination

This section describes the APIs that constitute the explicit abnormal termination facility.

[SWS_CORE_00053]{DRAFT} ⌈

Kind:	function
Symbol:	AbortHandlerPrototype()
Scope:	namespace ara::core
Syntax:	`void AbortHandlerPrototype () noexcept;`
Return value:	None
Exception Safety:	noexcept
Header file:	#include "ara/core/abort.h"
Description:	A function declaration with the correct prototype for SetAbortHandler(). This declaration exists only for providing a function type that includes "noexcept" and that acts as base type for a type alias, which is defined in SWS_CORE_00050. This compensates for the fact that the C++ standard (up to and including C++14) prohibits that "noexcept" appears in an alias-declaration. There is no implementation of this function.

⌋(*RS_AP_00132*)

[SWS_CORE_00050] ⌈

Kind:	type alias
Symbol:	AbortHandler
Scope:	namespace ara::core
Derived from:	decltype(&AbortHandlerPrototype)
Syntax:	`using AbortHandler = decltype(&AbortHandlerPrototype);`
Header file:	#include "ara/core/abort.h"
Description:	The type of a handler for SetAbortHandler().

⌋(*RS_AP_00132*)

[SWS_CORE_00051] ⌈

Kind:	function	
Symbol:	SetAbortHandler(AbortHandler handler)	
Scope:	namespace ara::core	
Syntax:	`AbortHandler SetAbortHandler (AbortHandler handler) noexcept;`	
Parameters (in):	handler	a custom Abort handler (or nullptr)
Return value:	AbortHandler	the previously installed Abort handler (or nullptr if none was installed)
Exception Safety:	noexcept	
Thread Safety:	thread-safe	
Header file:	#include "ara/core/abort.h"	
Description:	Set a custom global Abort handler function and return the previously installed one. By setting nullptr, the implementation may restore the default handler instead. This function can be called from multiple threads simultaneously; these calls are performed in an implementation-defined sequence.	

⌋(*RS_AP_00132*)

[SWS_CORE_00052] ⌈

Kind:	function	
Symbol:	Abort(const char *text)	
Scope:	namespace ara::core	
Syntax:	`void Abort (const char *text) noexcept;`	
Parameters (in):	text	a custom text to include in the log message being output
Return value:	None	
Exception Safety:	noexcept	
Thread Safety:	thread-safe	
Header file:	#include "ara/core/abort.h"	
Description:	Abort the current operation. This function will never return to its caller. The stack is not unwound: destructors of variables with automatic storage duration are not called.	

⌋(*RS_AP_00127*, *RS_AP_00132*, *RS_AP_00136*)

A Mentioned Manifest Elements

For the sake of completeness, this chapter contains a set of class tables representing meta-classes mentioned in the context of this document but which are not contained directly in the scope of describing specific meta-model semantics.

Chapter is generated.

Class	**ImplementationDataType**			
Package	M2::AUTOSARTemplates::CommonStructure::ImplementationDataTypes			
Note	Describes a reusable data type on the implementation level. This will typically correspond to a typedef in C-code. **Tags:**atp.recommendedPackage=ImplementationDataTypes			
Base	*ARElement*, *ARObject*, *AbstractImplementationDataType*, *AtpBlueprint*, *AtpBlueprintable*, *AtpClassifier*, *AtpType*, *AutosarDataType*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *Packageable Element*, *Referrable*			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
dynamicArray SizeProfile	String	0..1	attr	Specifies the profile which the array will follow in case this data type is a variable size array.
isStructWith Optional Element	Boolean	0..1	attr	This attribute is only valid if the attribute category is set to STRUCTURE. If set to True, this attribute indicates that the ImplementationDataType has been created with the intention to define at least one element of the structure as optional.
subElement (ordered)	ImplementationData TypeElement	*	aggr	Specifies an element of an array, struct, or union data type. The aggregation of ImplementionDataTypeElement is subject to variability with the purpose to support the conditional existence of elements inside a Implementation DataType representing a structure. **Stereotypes:** atpVariation **Tags:**vh.latestBindingTime=preCompileTime
symbolProps	SymbolProps	0..1	aggr	This represents the SymbolProps for the Implementation DataType. **Stereotypes:** atpSplitable **Tags:**atp.Splitkey=symbolProps.shortName
typeEmitter	NameToken	0..1	attr	This attribute is used to control which part of the AUTOSAR toolchain is supposed to trigger data type definitions.

Table A.1: ImplementationDataType

B Interfaces to other Functional Clusters (informative)

B.1 Overview

AUTOSAR decided not to standardize interfaces which are exclusively used between Functional Clusters (on platform-level only), to allow efficient implementations, which might depend e.g. on the used Operating System.

This chapter provides informative guidelines how the interaction between Functional Clusters looks like, by clustering the relevant requirements of this document to describe Inter-Functional Cluster (IFC) interfaces. In addition, the standardized public interfaces which are accessible by user space applications (see chapter 8 ("API specification")) can also be used for interaction between Functional Clusters.

The goal is to provide a clear understanding of Functional Cluster boundaries and interaction, without specifying syntactical details. This ensures compatibility between documents specifying different Functional Clusters and supports parallel implementation of different Functional Clusters. Details of the interfaces are up to the platform provider. Additional interfaces, parameters and return values can be added.

B.2 Interface Tables

B.2.1 Functional Cluster initialization

`ara::core::Initialize` and `ara::core::Deinitialize` initialize and deinitialize other Functional Clusters as necessary for the particular implementation. All Functional Clusters where this is necessary thus need to provide internal interfaces for their initialization and deinitialization.

C History of Specification Items

Please note that the lists in this chapter also include specification items that have been removed from the specification in a later version. These specification items do not appear as hyperlinks in the document.

C.1 Specification Item History of this document compared to AUTOSAR R19-11.

C.1.1 Added Traceables in R20-11

Number	Heading
[SWS_CORE_00011]	AUTOSAR error domain range
[SWS_CORE_00016]	Vendor-defined error domain range
[SWS_CORE_00053]	
[SWS_CORE_00337]	
[SWS_CORE_00355]	
[SWS_CORE_00356]	
[SWS_CORE_00614]	
[SWS_CORE_00764]	
[SWS_CORE_00770]	
[SWS_CORE_00771]	
[SWS_CORE_00772]	
[SWS_CORE_00773]	
[SWS_CORE_00864]	
[SWS_CORE_00868]	
[SWS_CORE_00869]	
[SWS_CORE_00870]	
[SWS_CORE_01210]	
[SWS_CORE_01211]	
[SWS_CORE_01212]	
[SWS_CORE_01213]	
[SWS_CORE_01214]	
[SWS_CORE_01215]	
[SWS_CORE_01216]	
[SWS_CORE_01217]	
[SWS_CORE_01218]	
[SWS_CORE_01219]	
[SWS_CORE_01220]	
[SWS_CORE_01241]	
[SWS_CORE_01242]	
[SWS_CORE_01250]	
[SWS_CORE_01251]	
[SWS_CORE_01252]	
[SWS_CORE_01253]	
[SWS_CORE_01254]	

▽

△

Number	Heading
[SWS_CORE_01255]	
[SWS_CORE_01256]	
[SWS_CORE_01257]	
[SWS_CORE_01258]	
[SWS_CORE_01259]	
[SWS_CORE_01260]	
[SWS_CORE_01261]	
[SWS_CORE_01262]	
[SWS_CORE_01263]	
[SWS_CORE_01264]	
[SWS_CORE_01265]	
[SWS_CORE_01266]	
[SWS_CORE_01267]	
[SWS_CORE_01268]	
[SWS_CORE_01269]	
[SWS_CORE_01270]	
[SWS_CORE_01271]	
[SWS_CORE_01272]	
[SWS_CORE_01280]	
[SWS_CORE_01281]	
[SWS_CORE_01282]	
[SWS_CORE_01283]	
[SWS_CORE_01284]	
[SWS_CORE_01285]	
[SWS_CORE_01290]	
[SWS_CORE_01291]	
[SWS_CORE_01292]	
[SWS_CORE_01293]	
[SWS_CORE_01294]	
[SWS_CORE_01295]	
[SWS_CORE_01980]	
[SWS_CORE_01981]	
[SWS_CORE_04023]	
[SWS_CORE_04033]	
[SWS_CORE_06237]	
[SWS_CORE_06355]	
[SWS_CORE_06356]	
[SWS_CORE_06401]	

▽

△

Number	Heading
[SWS_CORE_06411]	
[SWS_CORE_06412]	
[SWS_CORE_06413]	
[SWS_CORE_06414]	
[SWS_CORE_06431]	
[SWS_CORE_06432]	
[SWS_CORE_08022]	
[SWS_CORE_08023]	
[SWS_CORE_08024]	
[SWS_CORE_08025]	
[SWS_CORE_08081]	
[SWS_CORE_08082]	
[SWS_CORE_10300]	ErrorCode type properties
[SWS_CORE_10400]	ErrorDomain type properties
[SWS_CORE_10900]	Error condition enumeration type
[SWS_CORE_10901]	Error condition enumeration naming
[SWS_CORE_10902]	Error condition enumeration contents
[SWS_CORE_10903]	Error condition enumeration numbers
[SWS_CORE_10910]	ErrorDomain exception base type
[SWS_CORE_10911]	ErrorDomain exception base type naming
[SWS_CORE_10912]	ErrorDomain exception type hierarchy
[SWS_CORE_10930]	ErrorDomain subclass type
[SWS_CORE_10931]	ErrorDomain subclass naming
[SWS_CORE_10932]	ErrorDomain subclass non-extensibility
[SWS_CORE_10933]	ErrorDomain subclass Errc symbol
[SWS_CORE_10934]	ErrorDomain subclass Exception symbol
[SWS_CORE_10950]	ErrorDomain subclass member function property
[SWS_CORE_10951]	ErrorDomain subclass shortname retrieval
[SWS_CORE_10952]	ErrorDomain subclass unique identifier retrieval
[SWS_CORE_10953]	Throwing ErrorCodes as exceptions
[SWS_CORE_10980]	ErrorDomain subclass accessor function
[SWS_CORE_10981]	ErrorDomain subclass accessor function naming
[SWS_CORE_10982]	ErrorDomain subclass accessor function
[SWS_CORE_10990]	MakeErrorCode overload for new error domains
[SWS_CORE_10991]	MakeErrorCode overload signature
[SWS_CORE_10999]	Custom error domain scope
[SWS_CORE_11200]	Array base behavior

▽

△

Number	Heading
[SWS_CORE_11800]	SteadyClock type requirements
[SWS_CORE_11801]	Epoch of SteadyClock
[SWS_CORE_12402]	“Noreturn” property for Abort
[SWS_CORE_12403]	Logging of Explicit Operation Abortion
[SWS_CORE_12404]	AbortHandler invocation
[SWS_CORE_12405]	Final action without AbortHandler
[SWS_CORE_12406]	Final action with a returning AbortHandler
[SWS_CORE_12407]	Thread-safety of Explicit Operation Abortion
[SWS_CORE_90001]	Include folder structure
[SWS_CORE_90002]	Prevent multiple inclusion of header file
[SWS_CORE_90003]	

Table C.1: Added Traceables in R20-11

C.1.2 Changed Traceables in R20-11

Number	Heading
[SWS_CORE_00010]	Error domain identifier
[SWS_CORE_00050]	
[SWS_CORE_00051]	
[SWS_CORE_00052]	
[SWS_CORE_00110]	
[SWS_CORE_00121]	
[SWS_CORE_00122]	
[SWS_CORE_00123]	
[SWS_CORE_00131]	
[SWS_CORE_00132]	
[SWS_CORE_00133]	
[SWS_CORE_00134]	
[SWS_CORE_00135]	
[SWS_CORE_00136]	
[SWS_CORE_00137]	
[SWS_CORE_00138]	
[SWS_CORE_00151]	
[SWS_CORE_00152]	
[SWS_CORE_00153]	
[SWS_CORE_00154]	
[SWS_CORE_00321]	

▽

△

Number	Heading
[SWS_CORE_00322]	
[SWS_CORE_00323]	
[SWS_CORE_00325]	
[SWS_CORE_00326]	
[SWS_CORE_00327]	
[SWS_CORE_00328]	
[SWS_CORE_00329]	
[SWS_CORE_00330]	
[SWS_CORE_00331]	
[SWS_CORE_00332]	
[SWS_CORE_00333]	
[SWS_CORE_00334]	
[SWS_CORE_00335]	
[SWS_CORE_00336]	
[SWS_CORE_00340]	
[SWS_CORE_00341]	
[SWS_CORE_00342]	
[SWS_CORE_00343]	
[SWS_CORE_00344]	
[SWS_CORE_00345]	
[SWS_CORE_00346]	
[SWS_CORE_00349]	
[SWS_CORE_00350]	
[SWS_CORE_00351]	
[SWS_CORE_00352]	
[SWS_CORE_00353]	
[SWS_CORE_00354]	
[SWS_CORE_00361]	
[SWS_CORE_00400]	
[SWS_CORE_00411]	
[SWS_CORE_00412]	
[SWS_CORE_00421]	
[SWS_CORE_00431]	
[SWS_CORE_00432]	
[SWS_CORE_00441]	
[SWS_CORE_00442]	
[SWS_CORE_00443]	
[SWS_CORE_00444]	

▽

△

Number	Heading
[SWS_CORE_00480]	
[SWS_CORE_00490]	
[SWS_CORE_00501]	
[SWS_CORE_00512]	
[SWS_CORE_00513]	
[SWS_CORE_00514]	
[SWS_CORE_00515]	
[SWS_CORE_00516]	
[SWS_CORE_00518]	
[SWS_CORE_00519]	
[SWS_CORE_00571]	
[SWS_CORE_00572]	
[SWS_CORE_00601]	
[SWS_CORE_00611]	
[SWS_CORE_00612]	
[SWS_CORE_00613]	
[SWS_CORE_00701]	
[SWS_CORE_00711]	
[SWS_CORE_00712]	
[SWS_CORE_00721]	
[SWS_CORE_00722]	
[SWS_CORE_00723]	
[SWS_CORE_00724]	
[SWS_CORE_00725]	
[SWS_CORE_00726]	
[SWS_CORE_00727]	
[SWS_CORE_00731]	
[SWS_CORE_00732]	
[SWS_CORE_00733]	
[SWS_CORE_00734]	
[SWS_CORE_00735]	
[SWS_CORE_00736]	
[SWS_CORE_00741]	
[SWS_CORE_00742]	
[SWS_CORE_00743]	
[SWS_CORE_00744]	
[SWS_CORE_00745]	
[SWS_CORE_00751]	

▽

△

Number	Heading
[SWS_CORE_00752]	
[SWS_CORE_00753]	
[SWS_CORE_00754]	
[SWS_CORE_00755]	
[SWS_CORE_00756]	
[SWS_CORE_00757]	
[SWS_CORE_00758]	
[SWS_CORE_00759]	
[SWS_CORE_00761]	
[SWS_CORE_00762]	
[SWS_CORE_00763]	
[SWS_CORE_00765]	
[SWS_CORE_00766]	
[SWS_CORE_00767]	
[SWS_CORE_00768]	
[SWS_CORE_00769]	
[SWS_CORE_00780]	
[SWS_CORE_00781]	
[SWS_CORE_00782]	
[SWS_CORE_00783]	
[SWS_CORE_00784]	
[SWS_CORE_00785]	
[SWS_CORE_00786]	
[SWS_CORE_00787]	
[SWS_CORE_00788]	
[SWS_CORE_00789]	
[SWS_CORE_00796]	
[SWS_CORE_00801]	
[SWS_CORE_00811]	
[SWS_CORE_00812]	
[SWS_CORE_00821]	
[SWS_CORE_00823]	
[SWS_CORE_00824]	
[SWS_CORE_00825]	
[SWS_CORE_00826]	
[SWS_CORE_00827]	
[SWS_CORE_00831]	

▽

△

Number	Heading
[SWS_CORE_00834]	
[SWS_CORE_00835]	
[SWS_CORE_00836]	
[SWS_CORE_00841]	
[SWS_CORE_00842]	
[SWS_CORE_00843]	
[SWS_CORE_00844]	
[SWS_CORE_00845]	
[SWS_CORE_00851]	
[SWS_CORE_00852]	
[SWS_CORE_00853]	
[SWS_CORE_00855]	
[SWS_CORE_00857]	
[SWS_CORE_00858]	
[SWS_CORE_00861]	
[SWS_CORE_00863]	
[SWS_CORE_00865]	
[SWS_CORE_00866]	
[SWS_CORE_00867]	
[SWS_CORE_01201]	
[SWS_CORE_01296]	
[SWS_CORE_01390]	Global `operator==` for `Vector`
[SWS_CORE_01391]	Global `operator!=` for `Vector`
[SWS_CORE_01392]	Global `operator<` for `Vector`
[SWS_CORE_01393]	Global `operator<=` for `Vector`
[SWS_CORE_01394]	Global `operator>` for `Vector`
[SWS_CORE_01395]	Global `operator>=` for `Vector`
[SWS_CORE_01900]	
[SWS_CORE_01901]	
[SWS_CORE_01911]	
[SWS_CORE_01912]	
[SWS_CORE_01913]	
[SWS_CORE_01914]	
[SWS_CORE_01915]	
[SWS_CORE_01916]	
[SWS_CORE_01917]	
[SWS_CORE_01918]	

▽

△

Number	Heading
[SWS_CORE_01919]	
[SWS_CORE_01920]	
[SWS_CORE_01921]	
[SWS_CORE_01931]	
[SWS_CORE_01941]	
[SWS_CORE_01942]	
[SWS_CORE_01943]	
[SWS_CORE_01944]	
[SWS_CORE_01945]	
[SWS_CORE_01946]	
[SWS_CORE_01947]	
[SWS_CORE_01948]	
[SWS_CORE_01949]	
[SWS_CORE_01950]	
[SWS_CORE_01951]	
[SWS_CORE_01952]	
[SWS_CORE_01961]	
[SWS_CORE_01962]	
[SWS_CORE_01963]	
[SWS_CORE_01964]	
[SWS_CORE_01965]	
[SWS_CORE_01966]	
[SWS_CORE_01967]	
[SWS_CORE_01968]	
[SWS_CORE_01969]	
[SWS_CORE_01970]	
[SWS_CORE_01971]	
[SWS_CORE_01972]	
[SWS_CORE_01973]	
[SWS_CORE_01974]	
[SWS_CORE_01975]	
[SWS_CORE_01976]	
[SWS_CORE_01977]	
[SWS_CORE_01978]	
[SWS_CORE_01979]	
[SWS_CORE_01990]	
[SWS_CORE_01991]	

▽

△

Number	Heading
[SWS_CORE_01992]	
[SWS_CORE_01993]	
[SWS_CORE_01994]	
[SWS_CORE_03303]	Constructor from implicit `StringView`
[SWS_CORE_03306]	Assignment from implicit `StringView`
[SWS_CORE_03309]	Concatenation of implicit `StringView`
[SWS_CORE_03311]	Insertion of implicit `StringView`
[SWS_CORE_03313]	Replacement with implicit `StringView`
[SWS_CORE_03323]	Comparison of subsequence with a subsequence of a `StringView`
[SWS_CORE_04011]	
[SWS_CORE_04012]	
[SWS_CORE_04013]	
[SWS_CORE_04021]	
[SWS_CORE_04022]	
[SWS_CORE_04031]	
[SWS_CORE_04032]	
[SWS_CORE_04110]	
[SWS_CORE_04111]	
[SWS_CORE_04112]	
[SWS_CORE_04113]	
[SWS_CORE_04120]	
[SWS_CORE_04121]	
[SWS_CORE_04130]	
[SWS_CORE_04131]	
[SWS_CORE_04132]	
[SWS_CORE_04200]	
[SWS_CORE_05200]	
[SWS_CORE_05211]	
[SWS_CORE_05212]	
[SWS_CORE_05221]	
[SWS_CORE_05231]	
[SWS_CORE_05232]	
[SWS_CORE_05241]	
[SWS_CORE_05242]	
[SWS_CORE_05243]	
[SWS_CORE_05244]	
[SWS_CORE_05280]	

▽

△

Number	Heading
[SWS_CORE_05290]	
[SWS_CORE_06221]	
[SWS_CORE_06222]	
[SWS_CORE_06223]	
[SWS_CORE_06225]	
[SWS_CORE_06226]	
[SWS_CORE_06227]	
[SWS_CORE_06228]	
[SWS_CORE_06229]	
[SWS_CORE_06230]	
[SWS_CORE_06231]	
[SWS_CORE_06232]	
[SWS_CORE_06233]	
[SWS_CORE_06234]	
[SWS_CORE_06235]	
[SWS_CORE_06236]	
[SWS_CORE_06340]	
[SWS_CORE_06341]	
[SWS_CORE_06342]	
[SWS_CORE_06343]	
[SWS_CORE_06344]	
[SWS_CORE_06345]	
[SWS_CORE_06349]	
[SWS_CORE_06350]	
[SWS_CORE_06351]	
[SWS_CORE_06352]	
[SWS_CORE_06353]	
[SWS_CORE_06354]	
[SWS_CORE_08001]	
[SWS_CORE_08021]	
[SWS_CORE_08029]	
[SWS_CORE_08032]	
[SWS_CORE_08041]	
[SWS_CORE_08042]	
[SWS_CORE_08043]	
[SWS_CORE_08044]	
[SWS_CORE_08045]	

▽

△

Number	Heading
[SWS_CORE_08046]	
[SWS_CORE_10001]	
[SWS_CORE_10002]	
[SWS_CORE_10109]	Equality comparison for ara::core::Byte
[SWS_CORE_10110]	Non-equality comparison for ara::core::Byte

Table C.2: Changed Traceables in R20-11

C.1.3 Deleted Traceables in R20-11

none

C.2 Specification Item History of this document compared to AUTOSAR R19-03.

C.2.1 Added Traceables in R19-11

Number	Heading
[SWS_CORE_00003]	Handling of Violations
[SWS_CORE_00004]	Handling of Corruptions
[SWS_CORE_00005]	Handling of failed default allocations
[SWS_CORE_00014]	The Core error domain
[SWS_CORE_00050]	
[SWS_CORE_00051]	
[SWS_CORE_00052]	
[SWS_CORE_00131]	
[SWS_CORE_00132]	
[SWS_CORE_00133]	
[SWS_CORE_00134]	
[SWS_CORE_00135]	
[SWS_CORE_00136]	
[SWS_CORE_00137]	
[SWS_CORE_00138]	
[SWS_CORE_00151]	
[SWS_CORE_00152]	
[SWS_CORE_00153]	
[SWS_CORE_00154]	

▽

△

Number	Heading
[SWS_CORE_00322]	
[SWS_CORE_00323]	
[SWS_CORE_00325]	
[SWS_CORE_00326]	
[SWS_CORE_00327]	
[SWS_CORE_00328]	
[SWS_CORE_00329]	
[SWS_CORE_00330]	
[SWS_CORE_00331]	
[SWS_CORE_00332]	
[SWS_CORE_00333]	
[SWS_CORE_00334]	
[SWS_CORE_00335]	
[SWS_CORE_00336]	
[SWS_CORE_00341]	
[SWS_CORE_00342]	
[SWS_CORE_00343]	
[SWS_CORE_00344]	
[SWS_CORE_00345]	
[SWS_CORE_00346]	
[SWS_CORE_00349]	
[SWS_CORE_00350]	
[SWS_CORE_00351]	
[SWS_CORE_00352]	
[SWS_CORE_00353]	
[SWS_CORE_00354]	
[SWS_CORE_00412]	
[SWS_CORE_00441]	
[SWS_CORE_00442]	
[SWS_CORE_00443]	
[SWS_CORE_00444]	
[SWS_CORE_00480]	
[SWS_CORE_00490]	
[SWS_CORE_00512]	
[SWS_CORE_00513]	
[SWS_CORE_00514]	
[SWS_CORE_00515]	
[SWS_CORE_00516]	

▽

△

Number	Heading
[SWS_CORE_00518]	
[SWS_CORE_00519]	
[SWS_CORE_00571]	
[SWS_CORE_00572]	
[SWS_CORE_00611]	
[SWS_CORE_00612]	
[SWS_CORE_00613]	
[SWS_CORE_00721]	
[SWS_CORE_00722]	
[SWS_CORE_00723]	
[SWS_CORE_00724]	
[SWS_CORE_00725]	
[SWS_CORE_00726]	
[SWS_CORE_00727]	
[SWS_CORE_00731]	
[SWS_CORE_00732]	
[SWS_CORE_00733]	
[SWS_CORE_00734]	
[SWS_CORE_00735]	
[SWS_CORE_00736]	
[SWS_CORE_00741]	
[SWS_CORE_00742]	
[SWS_CORE_00743]	
[SWS_CORE_00744]	
[SWS_CORE_00745]	
[SWS_CORE_00751]	
[SWS_CORE_00752]	
[SWS_CORE_00753]	
[SWS_CORE_00754]	
[SWS_CORE_00755]	
[SWS_CORE_00756]	
[SWS_CORE_00757]	
[SWS_CORE_00758]	
[SWS_CORE_00759]	
[SWS_CORE_00761]	
[SWS_CORE_00762]	
[SWS_CORE_00763]	
[SWS_CORE_00765]	

▽

△

Number	Heading
[SWS_CORE_00766]	
[SWS_CORE_00767]	
[SWS_CORE_00768]	
[SWS_CORE_00769]	
[SWS_CORE_00780]	
[SWS_CORE_00781]	
[SWS_CORE_00782]	
[SWS_CORE_00783]	
[SWS_CORE_00784]	
[SWS_CORE_00785]	
[SWS_CORE_00786]	
[SWS_CORE_00787]	
[SWS_CORE_00788]	
[SWS_CORE_00789]	
[SWS_CORE_00796]	
[SWS_CORE_00821]	
[SWS_CORE_00823]	
[SWS_CORE_00824]	
[SWS_CORE_00825]	
[SWS_CORE_00826]	
[SWS_CORE_00827]	
[SWS_CORE_00831]	
[SWS_CORE_00834]	
[SWS_CORE_00835]	
[SWS_CORE_00836]	
[SWS_CORE_00841]	
[SWS_CORE_00842]	
[SWS_CORE_00843]	
[SWS_CORE_00844]	
[SWS_CORE_00845]	
[SWS_CORE_00851]	
[SWS_CORE_00852]	
[SWS_CORE_00853]	
[SWS_CORE_00855]	
[SWS_CORE_00857]	
[SWS_CORE_00858]	
[SWS_CORE_00861]	

▽

△

Number	Heading
[SWS_CORE_00863]	
[SWS_CORE_00865]	
[SWS_CORE_00866]	
[SWS_CORE_00867]	
[SWS_CORE_01941]	
[SWS_CORE_01942]	
[SWS_CORE_01943]	
[SWS_CORE_01944]	
[SWS_CORE_01945]	
[SWS_CORE_01946]	
[SWS_CORE_01947]	
[SWS_CORE_01948]	
[SWS_CORE_01949]	
[SWS_CORE_01950]	
[SWS_CORE_01951]	
[SWS_CORE_01952]	
[SWS_CORE_01961]	
[SWS_CORE_01962]	
[SWS_CORE_01963]	
[SWS_CORE_01964]	
[SWS_CORE_01965]	
[SWS_CORE_01966]	
[SWS_CORE_01967]	
[SWS_CORE_01968]	
[SWS_CORE_01969]	
[SWS_CORE_01970]	
[SWS_CORE_01971]	
[SWS_CORE_01972]	
[SWS_CORE_01973]	
[SWS_CORE_01974]	
[SWS_CORE_01975]	
[SWS_CORE_01976]	
[SWS_CORE_01977]	
[SWS_CORE_01978]	
[SWS_CORE_01979]	
[SWS_CORE_01990]	
[SWS_CORE_01991]	

▽

△

Number	Heading
[SWS_CORE_01992]	
[SWS_CORE_01993]	
[SWS_CORE_01994]	
[SWS_CORE_03000]	`BasicString` type
[SWS_CORE_04012]	
[SWS_CORE_04022]	
[SWS_CORE_04032]	
[SWS_CORE_04110]	
[SWS_CORE_04111]	
[SWS_CORE_04112]	
[SWS_CORE_04113]	
[SWS_CORE_04120]	
[SWS_CORE_04121]	
[SWS_CORE_04130]	
[SWS_CORE_04131]	
[SWS_CORE_04132]	
[SWS_CORE_04200]	
[SWS_CORE_05200]	
[SWS_CORE_05211]	
[SWS_CORE_05212]	
[SWS_CORE_05221]	
[SWS_CORE_05231]	
[SWS_CORE_05232]	
[SWS_CORE_05241]	
[SWS_CORE_05242]	
[SWS_CORE_05243]	
[SWS_CORE_05244]	
[SWS_CORE_05280]	
[SWS_CORE_05290]	
[SWS_CORE_06221]	
[SWS_CORE_06222]	
[SWS_CORE_06223]	
[SWS_CORE_06225]	
[SWS_CORE_06226]	
[SWS_CORE_06227]	
[SWS_CORE_06228]	
[SWS_CORE_06229]	

▽

△

Number	Heading
[SWS_CORE_06230]	
[SWS_CORE_06231]	
[SWS_CORE_06232]	
[SWS_CORE_06233]	
[SWS_CORE_06234]	
[SWS_CORE_06235]	
[SWS_CORE_06236]	
[SWS_CORE_06340]	
[SWS_CORE_06341]	
[SWS_CORE_06342]	
[SWS_CORE_06343]	
[SWS_CORE_06344]	
[SWS_CORE_06345]	
[SWS_CORE_06349]	
[SWS_CORE_06350]	
[SWS_CORE_06351]	
[SWS_CORE_06352]	
[SWS_CORE_06353]	
[SWS_CORE_06354]	
[SWS_CORE_08021]	
[SWS_CORE_08029]	
[SWS_CORE_08032]	
[SWS_CORE_08041]	
[SWS_CORE_08042]	
[SWS_CORE_08043]	
[SWS_CORE_08044]	
[SWS_CORE_08045]	
[SWS_CORE_08046]	
[SWS_CORE_10001]	
[SWS_CORE_10002]	
[SWS_CORE_10100]	Type property of ara::core::Byte
[SWS_CORE_10101]	Size of type ara::core::Byte
[SWS_CORE_10102]	Value range of type ara::core::Byte
[SWS_CORE_10103]	Creation of ara::core::Byte instances
[SWS_CORE_10104]	Default-constructed ara::core::Byte instances
[SWS_CORE_10105]	Destructor of type ara::core::Byte
[SWS_CORE_10106]	Implicit conversion from other types

▽

△

Number	Heading
[SWS_CORE_10107]	Implicit conversion to other types
[SWS_CORE_10108]	Conversion to unsigned char
[SWS_CORE_10109]	Equality comparison for byte ara::core::Byte
[SWS_CORE_10110]	Non-equality comparison for byte ara::core::Byte
[SWS_CORE_10200]	Valid InstanceSpecifier representations
[SWS_CORE_10201]	Validation of meta-model paths
[SWS_CORE_10202]	Construction of InstanceSpecifier objects

Table C.3: Added Traceables in R19-11

C.2.2 Changed Traceables in R19-11

Number	Heading
[SWS_CORE_00002]	Handling of Errors
[SWS_CORE_00040]	Errors originating from C++ standard classes
[SWS_CORE_03001]	`String` type
[SWS_CORE_03296]	`swap` overload for `BasicString`
[SWS_CORE_03301]	Implicit conversion to `StringView`
[SWS_CORE_03302]	Constructor from `StringView`
[SWS_CORE_03303]	Constructor from implicit `StringView`
[SWS_CORE_03304]	operator= from `StringView`
[SWS_CORE_03305]	Assignment from `StringView`
[SWS_CORE_03306]	Assignment from implicit `StringView`
[SWS_CORE_03307]	operator+ from `StringView`
[SWS_CORE_03308]	Concatenation of `StringView`
[SWS_CORE_03309]	Concatenation of implicit `StringView`
[SWS_CORE_03310]	Insertion of `StringView`
[SWS_CORE_03311]	Insertion of implicit `StringView`
[SWS_CORE_03312]	Replacement with `StringView`
[SWS_CORE_03313]	Replacement with implicit `StringView`
[SWS_CORE_03314]	Replacement of iterator range with `StringView`
[SWS_CORE_03315]	Forward-find a `StringView`
[SWS_CORE_03316]	Reverse-find a `StringView`
[SWS_CORE_03317]	Forward-find of character set within a `StringView`
[SWS_CORE_03318]	Reverse-find of character set within a `StringView`
[SWS_CORE_03319]	Forward-find of character set not within a `StringView`
[SWS_CORE_03320]	Reverse-find of character set not within a `StringView`
[SWS_CORE_03321]	Comparison with a `StringView`

▽

△

Number	Heading
[SWS_CORE_03322]	Comparison of subsequence with a `StringView`
[SWS_CORE_03323]	Comparison of subsequence with a subsequence of a `StringView`

Table C.4: Changed Traceables in R19-11

C.2.3 Deleted Traceables in R19-11

Number	Heading
[SWS_CORE_00001]	Handling of Fatal Errors
[SWS_CORE_00012]	The POSIX error domain

Table C.5: Deleted Traceables in R19-11