Document Title	Specification of Core Types for Adaptive Platform
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	903

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

Document Change History

Date	Release	Changed by	Description
2019-11-28	R19-11	AUTOSAR Release Management	• 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

Core Types defines common classes and functionality used by multiple Functional Clusters as part of their public interfaces.

2 Acronyms and Abbreviations

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

Abbreviation / Acronym:	**Description:**
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] ValueOrError and ValueOrNone types
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0786r1.pdf

[4] Explanation of ara::com API
AUTOSAR_EXP_ARAComAPI

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

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

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

[8] 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 (“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_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_00013] [SWS_CORE_00014] [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] [SWS_CORE_00323] [SWS_CORE_00325]

Requirement	Description	Satisfied by
		[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] [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]

Requirement	Description	Satisfied by
		[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_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] [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]

Requirement	Description	Satisfied by
		[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_00865] [SWS_CORE_00866] [SWS_CORE_00867] [SWS_CORE_01030] [SWS_CORE_01031] [SWS_CORE_01033] [SWS_CORE_01096] [SWS_CORE_01201] [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]

Requirement	Description	Satisfied by
		[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] [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]

Requirement	Description	Satisfied by
		[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] [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]

Requirement	Description	Satisfied by
		[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_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_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]
[RS_AP_00132]	noexcept behavior of API functions	[SWS_CORE_00050] [SWS_CORE_00051] [SWS_CORE_00052]

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

7 Functional Specification

This chapter describes the concepts that are introduced with this Functional Cluster, which only consists of data types and helper functions. Particular emphasis is put on error handling.

7.1 Error handling

7.1.1 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`).

7.1.2 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.

7.1.3 Handling of unsuccessful operations in the Adaptive Platform

The types of unsuccessful operations defined in section 7.1.1 ("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.

7.1.3.1 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.

As ara::core::ErrorCode is fully constexpr-capable, creation of this temporary instance is usually free of any runtime cost (assuming that the ErrorDomain subclass has also been made constexpr-capable, see below).

7.1.3.2 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 occured 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.

7.1.3.3 Result

The ara::core::Result type follows the ValueOrError concept from the C++ proposal p0786 [3]. 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.

7.1.3.4 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 7.1.3.3, 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.

7.1.4 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.

7.1.5 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 7.1.3.2). 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.

7.1.6 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.

Defining an own error domain firstly consists of defining an enumeration describing all known error situations of the new software module. Then, a new class must be created that derives from ara::core::ErrorDomain and defines all the mandatory members. One of these members is a type alias (called Errc) for the enumeration; another member is a type alias (called Exception) for an exception base class for this new error domain.

In addition, a global accessor function for the new error domain class should be defined. For an error domain class MyErrorDomain, the accessor function should be named something like GetMyErrorDomain. This accessor function should return a reference to a single global instance of that class. This accessor function should be fully constexpr-capable; this in turn implies that the ErrorDomain subclass also should be constexpr-constructible.

And finally, a global factory function ara::core::MakeErrorCode should 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.

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.

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

```cpp

// How to define <ApApplicationErrorDomain.SN>?
// How to define enum values?

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

class <ApApplicationErrorDomain.SN>Exception : public ara::core::
    Exception
{
public:
    <ApApplicationErrorDomain.SN>Exception(ara::core::ErrorCode&& err);
};
```

```

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

  constexpr <ApApplicationErrorDomain.SN>ErrorDomain() noexcept;
  char const* Name() const noexcept override;
  char const* Message(ara::core::ErrorDomain::CodeType errorCode) const
       noexcept override;
  void ThrowAsException(ara::core::ErrorCode const& errorCode) const
       noexcept(false) override;
};

constexpr ErrorDomain const& Get<ApApplicationErrorDomain.SN>
    ErrorDomain();

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

7.1.7 AUTOSAR error domains

[SWS_CORE_00010]{DRAFT} **AUTOSAR error domain range** ⌈All error domains shall have a system-wide unique identifier that is represented as a 64-bit unsigned integer value. Identifiers that have their top bit (i.e. bit #63) set are reserved for AUTOSAR-defined error domains; all user-defined error domains shall have the top bit of their identifier set to 0.⌋*(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 Advanced data types

7.2.1 AUTOSAR types

7.2.1.1 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 [4] chapter 8.4.4.

`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)*

7.2.2 Types derived from the 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.

[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], [6, the C++17 standard], or [7, 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)*

7.2.2.1 Types taken from the C++11 standard

These types are already contained in the [5, 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.

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

7.2.2.2 Types taken 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: StringView, Span, Optional, and Variant.

7.2.2.2.1 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 [6, the C++17 standard] a conforming implementation, but also allow a class-based implementation with only C++11 means.

Unlike `std::byte` from [6, 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 byte 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 byte 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

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

8.1 `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:	ara::core::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:	ara::core::ErrorDomain::IdType
Scope:	class ara::core::ErrorDomain
Derived from:	std::uint64_t
Syntax:	`using ara::core::ErrorDomain::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:	ara::core::ErrorDomain::CodeType
Scope:	class ara::core::ErrorDomain
Derived from:	std::int32_t
Syntax:	`using ara::core::ErrorDomain::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:	ara::core::ErrorDomain::SupportDataType
Scope:	class ara::core::ErrorDomain
Derived from:	IMPLEMENTATION_DEFINED
Syntax:	`using ara::core::ErrorDomain::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:	ara::core::ErrorDomain::ErrorDomain(ErrorDomain const &)
Scope:	class ara::core::ErrorDomain
Syntax:	`ErrorDomain (ErrorDomain const &)=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:	ara::core::ErrorDomain::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:	ara::core::ErrorDomain::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:	ara::core::ErrorDomain::~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:	ara::core::ErrorDomain::operator=(ErrorDomain const &)
Scope:	class ara::core::ErrorDomain
Syntax:	`ErrorDomain& operator= (ErrorDomain const &)=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:	ara::core::ErrorDomain::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:	ara::core::ErrorDomain::operator==(ErrorDomain const &other)	
Scope:	class ara::core::ErrorDomain	
Syntax:	`constexpr bool operator== (ErrorDomain const &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:	ara::core::ErrorDomain::operator!=(ErrorDomain const &other)	
Scope:	class ara::core::ErrorDomain	
Syntax:	`constexpr bool operator!= (ErrorDomain const &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:	ara::core::ErrorDomain::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:	ara::core::ErrorDomain::Name()	
Scope:	class ara::core::ErrorDomain	
Syntax:	`virtual char const* Name () const noexcept=0;`	
Return value:	char const *	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:	ara::core::ErrorDomain::Message(CodeType errorCode)	
Scope:	class ara::core::ErrorDomain	
Syntax:	`virtual char const* Message (CodeType errorCode) const noexcept=0;`	
Parameters (in):	errorCode	the domain-specific error code
Return value:	char const *	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:	ara::core::ErrorDomain::ThrowAsException(ErrorCode const &errorCode)	
Scope:	class ara::core::ErrorDomain	
Syntax:	`virtual void ThrowAsException (ErrorCode const &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)*

8.2 `ErrorCode` data type

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

[SWS_CORE_00501]{DRAFT} ⌈

Kind:	class
Symbol:	ara::core::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:	ara::core::ErrorCode::ErrorCode(EnumT e, ErrorDomain::SupportDataType data=Error Domain::SupportDataType())	
Scope:	class ara::core::ErrorCode	

Syntax:	`template <typename EnumT>`
	`constexpr ErrorCode (EnumT e, ErrorDomain::SupportDataType data=Error Domain::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:	ara::core::ErrorCode::ErrorCode(ErrorDomain::CodeType value, ErrorDomain const &domain, ErrorDomain::SupportDataType data=ErrorDomain::SupportDataType())	
Scope:	class ara::core::ErrorCode	
Syntax:	`constexpr ErrorCode (ErrorDomain::CodeType value, ErrorDomain const &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:	ara::core::ErrorCode::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:	ara::core::ErrorCode::Domain()	
Scope:	class ara::core::ErrorCode	
Syntax:	`constexpr ErrorDomain const& Domain () const noexcept;`	
Return value:	ErrorDomain const &	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:	ara::core::ErrorCode::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:	ara::core::ErrorCode::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:	ara::core::ErrorCode::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)*

8.2.1 `ErrorCode` global operators

[SWS_CORE_00571]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::core::operator==(ErrorCode const &lhs, ErrorCode const &rhs)	
Scope:	namespace ara::core	
Syntax:	`constexpr bool operator== (ErrorCode const &lhs, ErrorCode const &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:	ara::core::operator!=(ErrorCode const &lhs, ErrorCode const &rhs)	
Scope:	namespace ara::core	
Syntax:	`constexpr bool operator!= (ErrorCode const &lhs, ErrorCode const &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)*

8.3 `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:	ara::core::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:	ara::core::Exception::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:	ara::core::Exception::what()	
Scope:	class ara::core::Exception	
Syntax:	`char const* what () const noexcept override;`	
Return value:	char const *	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:	ara::core::Exception::Error()	
Scope:	class ara::core::Exception	
Syntax:	`ErrorCode const& Error () const noexcept;`	
Return value:	ErrorCode const &	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)*

8.4 `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:	ara::core::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:	ara::core::Result::value_type
Scope:	class ara::core::Result
Derived from:	T
Syntax:	`using ara::core::Result< T, E >::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:	ara::core::Result::error_type
Scope:	class ara::core::Result
Derived from:	E
Syntax:	`using ara::core::Result< T, E >::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:	ara::core::Result::Result(T const &t)	
Scope:	class ara::core::Result	
Syntax:	`ara::core::Result< T, E >::Result (T const &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:	ara::core::Result::Result(T &&t)	
Scope:	class ara::core::Result	
Syntax:	`ara::core::Result< T, E >::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:	ara::core::Result::Result(E const &e)	
Scope:	class ara::core::Result	
Syntax:	`explicit ara::core::Result< T, E >::Result (E const &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:	ara::core::Result::Result(E &&e)	
Scope:	class ara::core::Result	
Syntax:	`explicit ara::core::Result< T, E >::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:	ara::core::Result::Result(Result const &other)	
Scope:	class ara::core::Result	
Syntax:	`ara::core::Result< T, E >::Result (Result const &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:	ara::core::Result::Result(Result &&other)	
Scope:	class ara::core::Result	
Syntax:	`ara::core::Result< T, E >::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:	ara::core::Result::~Result()
Scope:	class ara::core::Result
Syntax:	`ara::core::Result< T, E >::~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:	ara::core::Result::FromValue(T const &t)	
Scope:	class ara::core::Result	
Syntax:	`static Result ara::core::Result< T, E >::FromValue (T const &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:	ara::core::Result::FromValue(T &&t)	
Scope:	class ara::core::Result	
Syntax:	`static Result ara::core::Result< T, E >::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:	ara::core::Result::FromValue(Args &&... args)	
Scope:	class ara::core::Result	

Syntax:	`template <typename... Args>`
	`static Result ara::core::Result< T, E >::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:	ara::core::Result::FromError(E const &e)	
Scope:	class ara::core::Result	
Syntax:	`static Result ara::core::Result< T, E >::FromError (E const &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:	ara::core::Result::FromError(E &&e)	
Scope:	class ara::core::Result	
Syntax:	`static Result ara::core::Result< T, E >::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:	ara::core::Result::FromError(Args &&... args)	
Scope:	class ara::core::Result	

Syntax:	`template <typename... Args>`
	`static Result ara::core::Result< T, E >::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:	ara::core::Result::operator=(Result const &other)	
Scope:	class ara::core::Result	
Syntax:	`Result& ara::core::Result< T, E >::operator= (Result const &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:	ara::core::Result::operator=(Result &&other)	
Scope:	class ara::core::Result	
Syntax:	`Result& ara::core::Result< T, E >::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:	ara::core::Result::EmplaceValue(Args &&... args)
Scope:	class ara::core::Result

Syntax:	`template <typename... Args>`
	`void ara::core::Result< T, E >::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:	ara::core::Result::EmplaceError(Args &&... args)	
Scope:	class ara::core::Result	
Syntax:	`template <typename... Args>` `void ara::core::Result< T, E >::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:	ara::core::Result::Swap(Result &other)	
Scope:	class ara::core::Result	
Syntax:	`void ara::core::Result< T, E >::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:	ara::core::Result::HasValue()	
Scope:	class ara::core::Result	
Syntax:	`bool ara::core::Result< T, E >::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:	ara::core::Result::operator bool()	
Scope:	class ara::core::Result	
Syntax:	`explicit ara::core::Result< T, E >::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:	ara::core::Result::operator *()	
Scope:	class ara::core::Result	
Syntax:	`T const& ara::core::Result< T, E >::operator * () const &;`	
Return value:	T const &	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:	ara::core::Result::operator *()	
Scope:	class ara::core::Result	
Syntax:	`T&& ara::core::Result< T, E >::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:	ara::core::Result::operator->()	
Scope:	class ara::core::Result	
Syntax:	`T const* ara::core::Result< T, E >::operator-> () const;`	
Return value:	T const *	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:	ara::core::Result::Value()	
Scope:	class ara::core::Result	
Syntax:	`T const& ara::core::Result< T, E >::Value () const &;`	
Return value:	T const &	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:	ara::core::Result::Value()	
Scope:	class ara::core::Result	
Syntax:	`T&& ara::core::Result< T, E >::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:	ara::core::Result::Error()	
Scope:	class ara::core::Result	
Syntax:	`E const& ara::core::Result< T, E >::Error () const &;`	
Return value:	E const &	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:	ara::core::Result::Error()	
Scope:	class ara::core::Result	
Syntax:	`E&& ara::core::Result< T, 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_00761]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::core::Result::ValueOr(U &&defaultValue)	
Scope:	class ara::core::Result	

Syntax:	`template <typename U>`
	`T ara::core::Result< T, E >::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:	ara::core::Result::ValueOr(U &&defaultValue)	
Scope:	class ara::core::Result	

Syntax:	`template <typename U>`
	`T ara::core::Result< T, E >::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:	ara::core::Result::ErrorOr(G &&defaultError)	
Scope:	class ara::core::Result	

Syntax:	`template <typename G>`
	`E ara::core::Result< T, 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_00765]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::core::Result::CheckError(G &&error)	
Scope:	class ara::core::Result	

Syntax:	`template <typename G>`
	`bool ara::core::Result< T, E >::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:	ara::core::Result::ValueOrThrow()	
Scope:	class ara::core::Result	
Syntax:	`T const& ara::core::Result< T, E >::ValueOrThrow () const &noexcept(false);`	
Return value:	T const &	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:	ara::core::Result::ValueOrThrow()	
Scope:	class ara::core::Result	
Syntax:	`T&& ara::core::Result< T, E >::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:	ara::core::Result::Resolve(F &&f)	
Scope:	class ara::core::Result	

Syntax:	`template <typename F>`
	`T ara::core::Result< T, E >::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(E const&);	

⌋*(RS_AP_00130)*

[SWS_CORE_00768]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::core::Result::Bind(F &&f)	
Scope:	class ara::core::Result	
Syntax:	`template <typename F>` `auto ara::core::Result< T, E >::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(T const&); XXX f(T const&); 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)*

8.4.1 `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:	ara::core::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:	ara::core::Result< void, E >::value_type
Scope:	class ara::core::Result< void, E >
Derived from:	void
Syntax:	`using ara::core::Result< void, E >::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:	ara::core::Result< void, E >::error_type
Scope:	class ara::core::Result< void, E >
Derived from:	E
Syntax:	`using ara::core::Result< void, E >::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:	ara::core::Result< void, E >::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:	ara::core::Result< void, E >::Result(E const &e)	
Scope:	class ara::core::Result< void, E >	
Syntax:	`explicit Result (E const &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:	ara::core::Result< void, E >::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:	ara::core::Result< void, E >::Result(Result const &other)	
Scope:	class ara::core::Result< void, E >	
Syntax:	`Result (Result const &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:	ara::core::Result< void, E >::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:	ara::core::Result< void, E >::~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:	ara::core::Result< void, E >::FromValue()	
Scope:	class ara::core::Result< void, E >	
Syntax:	`static Result FromValue ();`	
Return value:	Result	a Result that contains a "void" value
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:	ara::core::Result< void, E >::FromError(E const &e)	
Scope:	class ara::core::Result< void, E >	
Syntax:	`static Result FromError (E const &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:	ara::core::Result< void, E >::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:	ara::core::Result< void, E >::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:	ara::core::Result< void, E >::operator=(Result const &other)	
Scope:	class ara::core::Result< void, E >	
Syntax:	`Result& operator= (Result const &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:	ara::core::Result< void, E >::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:	ara::core::Result< void, E >::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:	ara::core::Result< void, E >::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:	ara::core::Result< void, E >::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:	ara::core::Result< void, E >::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:	ara::core::Result< void, E >::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:	ara::core::Result< void, E >::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:	ara::core::Result< void, E >::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:	ara::core::Result< void, E >::Error()	
Scope:	class ara::core::Result< void, E >	
Syntax:	`E const& Error () const &;`	
Return value:	E const &	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:	ara::core::Result< void, E >::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_00861]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::core::Result< void, E >::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:	ara::core::Result< void, E >::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_00865]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::core::Result< void, E >::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:	ara::core::Result< void, E >::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:	ara::core::Result< void, E >::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(E const&); This function only exists for helping with generic programming.	

⌋*(RS_AP_00130)*

8.4.2 Global function overloads

[SWS_CORE_00780]{DRAFT} ⌈

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

Syntax:	`template <typename T, typename E>`
	`bool operator== (Result< T, E > const &lhs, Result< T, E > const &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:	ara::core::operator!=(Result< T, E > const &lhs, Result< T, E > const &rhs)	
Scope:	namespace ara::core	

Syntax:	`template <typename T, typename E>`
	`bool operator!= (Result< T, E > const &lhs, Result< T, E > const &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:	ara::core::operator==(Result< T, E > const &lhs, T const &rhs)	
Scope:	namespace ara::core	

Syntax:	`template <typename T, typename E>`
	`bool operator== (Result< T, E > const &lhs, T const &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:	ara::core::operator==(T const &lhs, Result< T, E > const &rhs)	
Scope:	namespace ara::core	

Syntax:	`template <typename T, typename E>`
	`bool operator== (T const &lhs, Result< T, E > const &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:	ara::core::operator!=(Result< T, E > const &lhs, T const &rhs)	
Scope:	namespace ara::core	

Syntax:	`template <typename T, typename E>`
	`bool operator!= (Result< T, E > const &lhs, T const &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:	ara::core::operator!=(T const &lhs, Result< T, E > const &rhs)	
Scope:	namespace ara::core	

Syntax:	`template <typename T, typename E>`
	`bool operator!= (T const &lhs, Result< T, E > const &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:	ara::core::operator==(Result< T, E > const &lhs, E const &rhs)	
Scope:	namespace ara::core	

Syntax:	`template <typename T, typename E>`
	`bool operator== (Result< T, E > const &lhs, E const &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:	ara::core::operator==(E const &lhs, Result< T, E > const &rhs)	
Scope:	namespace ara::core	

Syntax:	`template <typename T, typename E>`
	`bool operator== (E const &lhs, Result< T, E > const &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:	ara::core::operator!=(Result< T, E > const &lhs, E const &rhs)	
Scope:	namespace ara::core	
Syntax:	`template <typename T, typename E>` `bool operator!= (Result< T, E > const &lhs, E const &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:	ara::core::operator!=(E const &lhs, Result< T, E > const &rhs)	
Scope:	namespace ara::core	
Syntax:	`template <typename T, typename E>` `bool operator!= (E const &lhs, Result< T, E > const &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:	ara::core::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)*

8.5 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.

8.5.1 CORE error codes

[SWS_CORE_05200]{DRAFT} ⌈

Kind:	enumeration	
Symbol:	ara::core::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)*

8.5.2 `CoreException` type

[SWS_CORE_05211]{DRAFT} ⌈

Kind:	class
Symbol:	ara::core::CoreException
Scope:	namespace ara::core
Base class:	ara::core::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:	ara::core::CoreException::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)*

8.5.3 `CoreErrorDomain` type

[SWS_CORE_05221]{DRAFT} ⌈

Kind:	class
Symbol:	ara::core::CoreErrorDomain
Scope:	namespace ara::core
Base class:	ara::core::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:	ara::core::CoreErrorDomain::Errc
Scope:	class ara::core::CoreErrorDomain
Derived from:	CoreErrc
Syntax:	`using ara::core::CoreErrorDomain::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:	ara::core::CoreErrorDomain::Exception
Scope:	class ara::core::CoreErrorDomain
Derived from:	CoreException
Syntax:	`using ara::core::CoreErrorDomain::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:	ara::core::CoreErrorDomain::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:	ara::core::CoreErrorDomain::Name()	
Scope:	class ara::core::CoreErrorDomain	
Syntax:	`char const* Name () const noexcept override;`	
Return value:	char const *	"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:	ara::core::CoreErrorDomain::Message(ErrorDomain::CodeType errorCode)	
Scope:	class ara::core::CoreErrorDomain	
Syntax:	`char const* Message (ErrorDomain::CodeType errorCode) const noexcept override;`	
Parameters (in):	errorCode	the error code value
Return value:	char const *	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:	ara::core::CoreErrorDomain::ThrowAsException(ErrorCode const &errorCode)	
Scope:	class ara::core::CoreErrorDomain	
Syntax:	`void ThrowAsException (ErrorCode const &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)*

8.5.4 `GetCoreErrorDomain` accessor function

[SWS_CORE_05280]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::core::GetCoreErrorDomain()	
Scope:	namespace ara::core	
Syntax:	`constexpr ErrorDomain const& GetCoreErrorDomain () noexcept;`	
Return value:	ErrorDomain const &	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)*

8.5.5 `MakeErrorCode` overload for `CoreErrorDomain`

[SWS_CORE_05290]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::core::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)*

8.6 `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 [8].

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.

8.6.1 `future_errc` enumeration

[SWS_CORE_00400]{DRAFT} ⌈

Kind:	enumeration	
Symbol:	ara::core::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)*

8.6.2 `FutureException` type

[SWS_CORE_00411]{DRAFT} ⌈

Kind:	class
Symbol:	ara::core::FutureException
Scope:	namespace ara::core
Base class:	ara::core::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:	ara::core::FutureException::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)*

8.6.3 `FutureErrorDomain` type

[SWS_CORE_00421]{DRAFT} ⌈

Kind:	class
Symbol:	ara::core::FutureErrorDomain
Scope:	namespace ara::core
Base class:	ara::core::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:	ara::core::FutureErrorDomain::Errc
Scope:	class ara::core::FutureErrorDomain
Derived from:	future_errc
Syntax:	`using ara::core::FutureErrorDomain::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:	ara::core::FutureErrorDomain::Exception
Scope:	class ara::core::FutureErrorDomain
Derived from:	FutureException
Syntax:	`using ara::core::FutureErrorDomain::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:	ara::core::FutureErrorDomain::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:	ara::core::FutureErrorDomain::Name()	
Scope:	class ara::core::FutureErrorDomain	
Syntax:	`char const* Name () const noexcept override;`	
Return value:	char const *	"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:	ara::core::FutureErrorDomain::Message(ErrorDomain::CodeType errorCode)	
Scope:	class ara::core::FutureErrorDomain	
Syntax:	`char const* Message (ErrorDomain::CodeType errorCode) const noexcept override;`	
Parameters (in):	errorCode	the error code value
Return value:	char const *	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:	ara::core::FutureErrorDomain::ThrowAsException(ErrorCode const &errorCode)	
Scope:	class ara::core::FutureErrorDomain	
Syntax:	`void ThrowAsException (ErrorCode const &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)*

8.6.4 `FutureErrorDomain` accessor function

[SWS_CORE_00480]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::core::GetFutureErrorDomain()	
Scope:	namespace ara::core	
Syntax:	`constexpr ErrorDomain const& GetFutureErrorDomain () noexcept;`	
Return value:	ErrorDomain const &	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)*

8.6.5 `MakeErrorCode` overload for `FutureErrorDomain`

[SWS_CORE_00490]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::core::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)*

8.6.6 `future_status` enumeration

[SWS_CORE_00361]{DRAFT} ⌈

Kind:	enumeration	
Symbol:	ara::core::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)*

8.6.7 `Future` data type

[SWS_CORE_00321]{DRAFT} ⌈

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

⌋*(RS_AP_00130)*

[SWS_CORE_00323]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::core::Future::Future(Future &&other)	
Scope:	class ara::core::Future	
Syntax:	`ara::core::Future< T, E >::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:	ara::core::Future::~Future()
Scope:	class ara::core::Future
Syntax:	`ara::core::Future< T, E >::~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:	ara::core::Future::operator=(Future const &)
Scope:	class ara::core::Future
Syntax:	`Future& ara::core::Future< T, E >::operator= (Future const &)=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:	ara::core::Future::operator=(Future &&other)	
Scope:	class ara::core::Future	
Syntax:	`Future& ara::core::Future< T, E >::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:	ara::core::Future::get()	
Scope:	class ara::core::Future	
Syntax:	`T ara::core::Future< T, E >::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:	ara::core::Future::GetResult()	
Scope:	class ara::core::Future	
Syntax:	`Result<T, E> ara::core::Future< 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:	ara::core::Future::valid()	
Scope:	class ara::core::Future	
Syntax:	`bool ara::core::Future< T, E >::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:	ara::core::Future::wait()
Scope:	class ara::core::Future
Syntax:	`void ara::core::Future< T, E >::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:	ara::core::Future::wait_for(std::chrono::duration< Rep, Period > const &timeoutDuration)	
Scope:	class ara::core::Future	

Syntax:	`template <typename Rep, typename Period>`
	`future_status ara::core::Future< T, E >::wait_for`
	`(std::chrono::duration< Rep, Period > const &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:	ara::core::Future::wait_until(std::chrono::time_point< Clock, Duration > const &deadline)	
Scope:	class ara::core::Future	
Syntax:	`template <typename Clock, typename Duration>` `future_status ara::core::Future< T, E >::wait_until` `(std::chrono::time_point< Clock, Duration > const &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:	ara::core::Future::then(F &&func)	
Scope:	class ara::core::Future	
Syntax:	`template <typename F>` `auto ara::core::Future< T, E >::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_00332]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::core::Future::is_ready()	
Scope:	class ara::core::Future	
Syntax:	`bool ara::core::Future< T, E >::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.

⌋*(RS_AP_00130)*

8.6.7.1 `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:	ara::core::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:	ara::core::Future< void, E >::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:	ara::core::Future< void, E >::Future(Future const &other)
Scope:	class ara::core::Future< void, E >

▽

△

Syntax:	`Future (Future const &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:	ara::core::Future< void, E >::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:	ara::core::Future< void, E >::~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:	ara::core::Future< void, E >::operator=(Future const &other)
Scope:	class ara::core::Future< void, E >
Syntax:	`Future& operator= (Future const &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:	ara::core::Future< void, E >::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:	ara::core::Future< void, E >::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:	ara::core::Future< void, E >::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:	ara::core::Future< void, E >::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:	ara::core::Future< void, E >::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:	ara::core::Future< void, E >::wait_for(std::chrono::duration< Rep, Period > const &timeout Duration)	
Scope:	class ara::core::Future< void, E >	

Syntax:	`template <typename Rep, typename Period>`
	`future_status wait_for (std::chrono::duration< Rep, Period > const &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:	ara::core::Future< void, E >::wait_until(std::chrono::time_point< Clock, Duration > const &deadline)	
Scope:	class ara::core::Future< void, E >	
Syntax:	`template <typename Clock, typename Duration>` `future_status wait_until (std::chrono::time_point< Clock, Duration > const &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:	ara::core::Future< void, E >::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_06232]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::core::Future< void, E >::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.

⌋(*RS_AP_00130*)

8.6.8 `Promise` data type

[SWS_CORE_00340]{DRAFT} ⌈

Kind:	class	
Symbol:	ara::core::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/future.h"	
Description:	ara::core specific variant of std::promise class	

⌋(*RS_AP_00130*)

[SWS_CORE_00341]{DRAFT} ⌈

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

⌋*(RS_AP_00130)*

[SWS_CORE_00349]{DRAFT} ⌈

Kind:	function
Symbol:	ara::core::Promise::~Promise()
Scope:	class ara::core::Promise
Syntax:	`ara::core::Promise< T, E >::~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:	ara::core::Promise::operator=(Promise &&other)	
Scope:	class ara::core::Promise	
Syntax:	`Promise& ara::core::Promise< T, E >::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:	ara::core::Promise::operator=(Promise const &)	
Scope:	class ara::core::Promise	
Syntax:	`Promise& ara::core::Promise< T, E >::operator= (Promise const &)=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:	ara::core::Promise::swap(Promise &other)	
Scope:	class ara::core::Promise	
Syntax:	`void ara::core::Promise< T, E >::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:	ara::core::Promise::get_future()	
Scope:	class ara::core::Promise	
Syntax:	`Future<T, E> ara::core::Promise< 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:	ara::core::Promise::set_value(T const &value)
Scope:	class ara::core::Promise

▽

△

Syntax:	`void ara::core::Promise< T, E >::set_value (T const &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:	ara::core::Promise::set_value(T &&value)	
Scope:	class ara::core::Promise	
Syntax:	`void ara::core::Promise< T, E >::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:	ara::core::Promise::SetError(E &&error)	
Scope:	class ara::core::Promise	
Syntax:	`void ara::core::Promise< T, E >::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:	ara::core::Promise::SetError(E const &error)	
Scope:	class ara::core::Promise	
Syntax:	`void ara::core::Promise< T, E >::SetError (E const &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)*

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

⌋*(RS_AP_00130)*

[SWS_CORE_06349]{DRAFT} ⌈

Kind:	function
Symbol:	ara::core::Promise< void, E >::~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:	ara::core::Promise< void, E >::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:	ara::core::Promise< void, E >::operator=(Promise const &)
Scope:	class ara::core::Promise< void, E >
Syntax:	`Promise& operator= (Promise const &)=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:	ara::core::Promise< void, E >::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:	ara::core::Promise< void, E >::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:	ara::core::Promise< void, E >::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:	ara::core::Promise< void, E >::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:	ara::core::Promise< void, E >::SetError(E const &error)	
Scope:	class ara::core::Promise< void, E >	
Syntax:	`void SetError (E const &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)*

8.7 `Array` data type

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

[SWS_CORE_01201]{DRAFT} **`Array` class template** ⌈The namespace `ara::core` shall provide a class template `Array`:

```
template <typename T, std::size_t N>
class Array { ... };
```

⌋*(RS_AP_00130)*

All members of this class and supporting constructs (such as global relational operators) shall behave identical to those of header `<array>` from [5] section 23.3. All supporting symbols shall be contained within namespace `ara::core`.

[SWS_CORE_01296]{DRAFT} **`swap` overload for `Array`** ⌈There shall be an overload of the `swap` function within the namespace `ara::core` for arguments of type `Array`. Its interface shall be equivalent to:

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

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

8.8 `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==(Vector<T, Allocator> const& lhs,
                Vector<T, Allocator> const& 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!=(Vector<T, Allocator> const& lhs,
                Vector<T, Allocator> const& 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<(Vector<T, Allocator> const& lhs,
               Vector<T, Allocator> const& 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<=(Vector<T, Allocator> const& lhs,
                Vector<T, Allocator> const& 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>(Vector<T, Allocator> const& lhs,
               Vector<T, Allocator> const& 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>=(Vector<T, Allocator> const& lhs,
                Vector<T, Allocator> const& 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*)

8.9 `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)*

8.10 `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 [6].

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 [6] 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 [6], 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)*

8.11 `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 [6] 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)*

8.12 `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 [6] 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`.

8.13 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 [6, 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 [6, 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 [6, 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(T const& t, size_type pos, size_type n);
```

This function shall behave the same as the corresponding `std::basic_string` function from [6, 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 [6, 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 [6, 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(T const& t, size_type pos, size_type n = npos);
```

This function shall behave the same as the corresponding `std::basic_string` function from [6, 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 [6, 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 [6, 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(T const& t, size_type pos, size_type n = npos);
```

This function shall behave the same as the corresponding `std::basic_string` function from [6, 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 [6, 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, T const& t,
                    size_type pos2, size_type n = npos);
```

This function shall behave the same as the corresponding `std::basic_string` function from [6, 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 [6, 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, T const& t,
                     size_type pos2, size_type n2 = npos);
```

This function shall behave the same as the corresponding `std::basic_string` function from [6, 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 [6, 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 [6, 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 [6, 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 [6, 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 [6, 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 [6, 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 [6, 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 [6, 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 [6, 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, T const& t,
            size_type pos2, size_type n2 = npos) const;
```

This function shall behave the same as the corresponding `std::basic_string` function from [6, 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)*

8.14 `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 [7] section 22.7, but has been adapted in several ways:

- All symbols from section 22.7.3.7 (`span.objectrep`) have been omitted.
- 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:	ara::core::dynamic_extent
Scope:	namespace ara::core
Type:	std::size_t
Syntax:	`constexpr std::size_t ara::core::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:	ara::core::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:	ara::core::Span::element_type
Scope:	class ara::core::Span
Derived from:	T
Syntax:	`using ara::core::Span< T, Extent >::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:	ara::core::Span::value_type
Scope:	class ara::core::Span
Derived from:	typename std::remove_cv<element_type>::type
Syntax:	`using ara::core::Span< T, Extent >::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:	ara::core::Span::index_type
Scope:	class ara::core::Span
Derived from:	std::size_t

▽

△

Syntax:	`using ara::core::Span< T, Extent >::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:	ara::core::Span::difference_type
Scope:	class ara::core::Span
Derived from:	std::ptrdiff_t
Syntax:	`using ara::core::Span< T, Extent >::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:	ara::core::Span::size_type
Scope:	class ara::core::Span
Derived from:	index_type
Syntax:	`using ara::core::Span< T, Extent >::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:	ara::core::Span::pointer
Scope:	class ara::core::Span
Derived from:	element_type*
Syntax:	`using ara::core::Span< T, Extent >::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:	ara::core::Span::reference
Scope:	class ara::core::Span
Derived from:	element_type&
Syntax:	`using ara::core::Span< T, Extent >::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:	ara::core::Span::iterator
Scope:	class ara::core::Span
Derived from:	implementation_defined
Syntax:	`using ara::core::Span< T, Extent >::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:	ara::core::Span::const_iterator
Scope:	class ara::core::Span
Derived from:	implementation_defined
Syntax:	`using ara::core::Span< T, Extent >::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:	ara::core::Span::reverse_iterator
Scope:	class ara::core::Span

▽

△

Derived from:	std::reverse_iterator<iterator>
Syntax:	`using ara::core::Span< T, Extent >::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:	ara::core::Span::const_reverse_iterator
Scope:	class ara::core::Span
Derived from:	std::reverse_iterator<const_iterator>
Syntax:	`using ara::core::Span< T, Extent >::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:	ara::core::Span::extent
Scope:	class ara::core::Span
Type:	index_type
Syntax:	`constexpr index_type ara::core::Span< T, Extent >::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:	ara::core::Span::Span()
Scope:	class ara::core::Span
Syntax:	`constexpr ara::core::Span< T, Extent >::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:	ara::core::Span::Span(pointer ptr, index_type count)	
Scope:	class ara::core::Span	
Syntax:	`constexpr ara::core::Span< T, Extent >::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:	ara::core::Span::Span(pointer firstElem, pointer lastElem)	
Scope:	class ara::core::Span	
Syntax:	`constexpr ara::core::Span< T, Extent >::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:	ara::core::Span::Span(element_type(&arr)[N])	
Scope:	class ara::core::Span	
Syntax:	`template <std::size_t N>` `constexpr ara::core::Span< T, Extent >::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:	ara::core::Span::Span(Array< value_type, N > &arr)	
Scope:	class ara::core::Span	
Syntax:	`template <std::size_t N>` `constexpr ara::core::Span< T, Extent >::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:	ara::core::Span::Span(Array< value_type, N > const &arr)	
Scope:	class ara::core::Span	
Syntax:	`template <std::size_t N>` `constexpr ara::core::Span< T, Extent >::Span (Array< value_type, N > const &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:	ara::core::Span::Span(Container &cont)	
Scope:	class ara::core::Span	
Syntax:	`template <typename Container>` `constexpr ara::core::Span< T, Extent >::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:	ara::core::Span::Span(Container const &cont)	
Scope:	class ara::core::Span	

Syntax:	`template <typename Container>`
	`constexpr ara::core::Span< T, Extent >::Span (Container const &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:	ara::core::Span::Span(Span const &other)	
Scope:	class ara::core::Span	

Syntax:	`constexpr ara::core::Span< T, Extent >::Span (Span const &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:	ara::core::Span::Span(Span< U, N > const &s)	
Scope:	class ara::core::Span	
Syntax:	`template <typename U, std::size_t N>` `constexpr ara::core::Span< T, Extent >::Span (Span< U, N > const &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:	ara::core::Span::~Span()
Scope:	class ara::core::Span
Syntax:	`ara::core::Span< T, Extent >::~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:	ara::core::Span::operator=(Span const &other)	
Scope:	class ara::core::Span	
Syntax:	`Span& ara::core::Span< T, Extent >::operator= (Span const &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:	ara::core::Span::first()	
Scope:	class ara::core::Span	
Syntax:	`template <std::size_t Count> constexpr Span<element_type, Count> ara::core::Span< T, Extent >::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:	ara::core::Span::first(index_type count)	
Scope:	class ara::core::Span	
Syntax:	`constexpr Span<element_type, dynamic_extent> ara::core::Span< T, 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:	ara::core::Span::last()	
Scope:	class ara::core::Span	
Syntax:	`template <std::size_t Count> constexpr Span<element_type, Count> ara::core::Span< T, Extent >::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:	ara::core::Span::last(index_type count)	
Scope:	class ara::core::Span	
Syntax:	`constexpr Span<element_type, dynamic_extent> ara::core::Span< T, 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:	ara::core::Span::subspan()	
Scope:	class ara::core::Span	
Syntax:	`template <std::size_t Offset, std::size_t Count = dynamic_extent> constexpr auto ara::core::Span< T, Extent >::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:	ara::core::Span::subspan(index_type offset, index_type count=dynamic_extent)	
Scope:	class ara::core::Span	
Syntax:	`constexpr Span<element_type, dynamic_extent> ara::core::Span< T, 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:	ara::core::Span::size()	
Scope:	class ara::core::Span	
Syntax:	`constexpr index_type ara::core::Span< T, Extent >::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:	ara::core::Span::size_bytes()	
Scope:	class ara::core::Span	
Syntax:	`constexpr index_type ara::core::Span< T, Extent >::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:	ara::core::Span::empty()	
Scope:	class ara::core::Span	
Syntax:	`constexpr bool ara::core::Span< T, Extent >::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:	ara::core::Span::operator[](index_type idx)
Scope:	class ara::core::Span

▽

△

Syntax:	`constexpr reference ara::core::Span< T, Extent >::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:	ara::core::Span::data()	
Scope:	class ara::core::Span	
Syntax:	`constexpr pointer ara::core::Span< T, Extent >::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:	ara::core::Span::begin()	
Scope:	class ara::core::Span	
Syntax:	`constexpr iterator ara::core::Span< T, Extent >::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:	ara::core::Span::end()
Scope:	class ara::core::Span
Syntax:	`constexpr iterator ara::core::Span< T, Extent >::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:	ara::core::Span::cbegin()	
Scope:	class ara::core::Span	
Syntax:	`constexpr const_iterator ara::core::Span< T, Extent >::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:	ara::core::Span::cend()	
Scope:	class ara::core::Span	
Syntax:	`constexpr const_iterator ara::core::Span< T, Extent >::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:	ara::core::Span::rbegin()	
Scope:	class ara::core::Span	
Syntax:	`constexpr reverse_iterator ara::core::Span< T, Extent >::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:	ara::core::Span::rend()	
Scope:	class ara::core::Span	
Syntax:	`constexpr reverse_iterator ara::core::Span< T, Extent >::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:	ara::core::Span::crbegin()	
Scope:	class ara::core::Span	
Syntax:	`constexpr const_reverse_iterator ara::core::Span< T, Extent >::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:	ara::core::Span::crend()	
Scope:	class ara::core::Span	
Syntax:	`constexpr const_reverse_iterator ara::core::Span< T, Extent >::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*)

[SWS_CORE_01990]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::core::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:	ara::core::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:	ara::core::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:	ara::core::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:	ara::core::MakeSpan(Container const &cont)	
Scope:	namespace ara::core	
Syntax:	`template <typename Container>` `constexpr Span<typename Container::value_type const> MakeSpan (Container const &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)*

8.15 `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:	ara::core::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:	ara::core::InstanceSpecifier::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_08029] ⌈

Kind:	function	
Symbol:	ara::core::InstanceSpecifier::~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:	ara::core::InstanceSpecifier::Create(StringView metaModelIdentifier)	
Scope:	class ara::core::InstanceSpecifier	
Syntax:	`static Result<InstanceSpecifier> Create (StringView metaModelIdentifier);`	

▽

△

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:	ara::core::InstanceSpecifier::operator==(InstanceSpecifier const &other)	
Scope:	class ara::core::InstanceSpecifier	
Syntax:	`bool operator== (InstanceSpecifier const &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:	ara::core::InstanceSpecifier::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:	ara::core::InstanceSpecifier::operator!=(InstanceSpecifier const &other)	
Scope:	class ara::core::InstanceSpecifier	
Syntax:	`bool operator!= (InstanceSpecifier const &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:	ara::core::InstanceSpecifier::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:	ara::core::InstanceSpecifier::operator<(InstanceSpecifier const &other)	
Scope:	class ara::core::InstanceSpecifier	
Syntax:	`bool operator< (InstanceSpecifier const &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:	ara::core::InstanceSpecifier::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)*

8.16 Generic helpers

8.16.1 ara::core::Byte

The exact setup of this type is implementation-defined; the specifications in section 7.2.2.2.1 ("ara::core::Byte") define the expected behavior.

[SWS_CORE_04200] ⌈

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

⌋*(RS_AP_00130)*

8.16.2 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 [6], 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 [6, the C++17 standard].

8.16.2.1 `in_place_t` tag

[SWS_CORE_04011] ⌈

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

⌋*(RS_AP_00130)*

8.16.2.2 `in_place_type_t` tag

[SWS_CORE_04021] ⌈

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

⌋*(RS_AP_00130)*

8.16.2.3 `in_place_index_t` tag

[SWS_CORE_04031] ⌈

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

⌋*(RS_AP_00130)*

8.16.3 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 [6].

[SWS_CORE_04110] ⌈

Kind:	function	
Symbol:	ara::core::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:	ara::core::data(Container const &c)	
Scope:	namespace ara::core	

Syntax:	`template <typename Container>`
	`constexpr auto data (Container const &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:	ara::core::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:	ara::core::data(std::initializer_list< E > il)	
Scope:	namespace ara::core	

Syntax:	`template <typename E>`
	`constexpr E const* 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:	E const *	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:	ara::core::size(Container const &c)	
Scope:	namespace ara::core	

Syntax:	`template <typename Container>`
	`constexpr auto size (Container const &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:	ara::core::size(T const (&array)[N])	
Scope:	namespace ara::core	
Syntax:	`template <typename T, std::size_t N>` `constexpr std::size_t size (T const (&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:	ara::core::empty(Container const &c)	
Scope:	namespace ara::core	
Syntax:	`template <typename Container>` `constexpr auto empty (Container const &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:	ara::core::empty(T const (&array)[N])	
Scope:	namespace ara::core	
Syntax:	`template <typename T, std::size_t N>` `constexpr bool empty (T const (&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:	ara::core::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)*

8.17 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:	ara::core::Initialize()	
Scope:	namespace ara::core	
Syntax:	`Result<void> Initialize ();`	
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.
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:	ara::core::Deinitialize()	
Scope:	namespace ara::core	
Syntax:	`Result<void> Deinitialize ();`	
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.
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)*

8.18 Abnormal process termination

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

[SWS_CORE_00050]{DRAFT} ⌈

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

⌋*(RS_AP_00132)*

[SWS_CORE_00051]{DRAFT} ⌈

Kind:	function
Symbol:	ara::core::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]{DRAFT} ⌈

Kind:	function	
Symbol:	ara::core::Abort(char const *text)	
Scope:	namespace ara::core	
Syntax:	`void Abort (char const *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:	Terminate the current process abnormally. Before terminating, a log message with FATAL severity is being output, which includes the text given as argument to this function. This function will never return to its caller. The stack is not unwound: destructors of variables with automatic storage duration are not called. An application can add exactly one own "hook" into this mechanism by calling the function Set AbortHandler(). If such an Abort handler has been installed, it is called in turn when Abort() is invoked, but after the log message has been output. If there is no custom Abort handler, or if there is one and it returns, then the invocation of this function will terminate the process via std::abort(). Any call of this function that is performed while another call is already in progress will block the calling thread. The text argument is expected to point to a null-terminated string with static storage duration. Note: The type of the text argument is a raw pointer (instead of a more "modern" type such as StringView) in order to increase the chances that the function call succeeds even in situations when e.g. the stack has been corrupted.	

⌋*(RS_AP_00127, RS_AP_00132, RS_AP_00136)*

A 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.

A.1 Specification Item History of this document compared to AUTOSAR R19-03.

A.1.1 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]	
[SWS_CORE_00322]	
[SWS_CORE_00323]	
[SWS_CORE_00325]	
[SWS_CORE_00326]	
[SWS_CORE_00327]	
[SWS_CORE_00328]	
[SWS_CORE_00329]	

▽

△

Number	Heading
[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]	
[SWS_CORE_00518]	
[SWS_CORE_00519]	
[SWS_CORE_00571]	
[SWS_CORE_00572]	
[SWS_CORE_00611]	
[SWS_CORE_00612]	

▽

△

Number	Heading
[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]	
[SWS_CORE_00766]	
[SWS_CORE_00767]	
[SWS_CORE_00768]	
[SWS_CORE_00769]	
[SWS_CORE_00780]	

▽

△

Number	Heading
[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]	
[SWS_CORE_00863]	
[SWS_CORE_00865]	
[SWS_CORE_00866]	
[SWS_CORE_00867]	
[SWS_CORE_01941]	

▽

△

Number	Heading
[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]	
[SWS_CORE_01992]	
[SWS_CORE_01993]	
[SWS_CORE_01994]	
[SWS_CORE_03000]	`BasicString` type
[SWS_CORE_04012]	

▽

△

Number	Heading
[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]	
[SWS_CORE_06230]	
[SWS_CORE_06231]	
[SWS_CORE_06232]	
[SWS_CORE_06233]	
[SWS_CORE_06234]	

▽

△

Number	Heading
[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
[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

▽

△

Number	Heading
[SWS_CORE_10201]	Validation of meta-model paths
[SWS_CORE_10202]	Construction of InstanceSpecifier objects

Table A.1: Added Traceables in R19-11

A.1.2 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`
[SWS_CORE_03322]	Comparison of subsequence with a `StringView`
[SWS_CORE_03323]	Comparison of subsequence with a subsequence of a `StringView`

Table A.2: Changed Traceables in R19-11

A.1.3 Deleted Traceables in R19-11

Number	**Heading**
[SWS_CORE_00001]	Handling of Fatal Errors
[SWS_CORE_00012]	The POSIX error domain

Table A.3: Deleted Traceables in R19-11