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

Document Status	Final
Part of AUTOSAR Standard	Adaptive Platform
Part of Standard Release	18-10

Document Change History			
Date	**Release**	**Changed by**	**Description**
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] Guidelines for the use of the C++14 language in critical and safety-related systems
AUTOSAR_RS_CPP14Guidelines

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

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

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

[7] N4762: Working Draft, Standard for Programming Language C++
http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2018/n4762.pdf

[8] ISO/IEC TS 19571:2016, Programming Languages – Technical specification for C++ extensions for concurrency
http://www.iso.org

[9] Explanation of ara::com API
AUTOSAR_EXP_ARAComAPI

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_00128]	Use of exceptions in API.	[SWS_CORE_00001] [SWS_CORE_00002]
[RS_AP_00130]	AUTOSAR Adaptive Platform shall represent a rich and modern programming environment.	[SWS_CORE_00010] [SWS_CORE_00012] [SWS_CORE_00013] [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_00200] [SWS_CORE_00211] [SWS_CORE_00212] [SWS_CORE_00221]

Requirement	Description	Satisfied by
		[SWS_CORE_00231] [SWS_CORE_00232] [SWS_CORE_00241] [SWS_CORE_00242] [SWS_CORE_00243] [SWS_CORE_00244] [SWS_CORE_00280] [SWS_CORE_00290] [SWS_CORE_00321] [SWS_CORE_00322] [SWS_CORE_00323] [SWS_CORE_00325] [SWS_CORE_00326] [SWS_CORE_00327] [SWS_CORE_00328] [SWS_CORE_00329] [SWS_CORE_00330] [SWS_CORE_00331] [SWS_CORE_00332] [SWS_CORE_00333] [SWS_CORE_00334] [SWS_CORE_00335] [SWS_CORE_00336] [SWS_CORE_00340] [SWS_CORE_00341] [SWS_CORE_00342] [SWS_CORE_00343] [SWS_CORE_00344] [SWS_CORE_00345] [SWS_CORE_00346] [SWS_CORE_00349] [SWS_CORE_00350] [SWS_CORE_00351] [SWS_CORE_00352] [SWS_CORE_00353] [SWS_CORE_00354] [SWS_CORE_00361] [SWS_CORE_00400] [SWS_CORE_00411] [SWS_CORE_00412] [SWS_CORE_00421] [SWS_CORE_00431] [SWS_CORE_00432] [SWS_CORE_00441] [SWS_CORE_00442] [SWS_CORE_00443] [SWS_CORE_00444] [SWS_CORE_00480]

Requirement	Description	Satisfied by
		[SWS_CORE_00490] [SWS_CORE_00501] [SWS_CORE_00511] [SWS_CORE_00512] [SWS_CORE_00513] [SWS_CORE_00514] [SWS_CORE_00515] [SWS_CORE_00516] [SWS_CORE_00517] [SWS_CORE_00518] [SWS_CORE_00601] [SWS_CORE_00611] [SWS_CORE_00612] [SWS_CORE_00613] [SWS_CORE_00701] [SWS_CORE_00711] [SWS_CORE_00712] [SWS_CORE_00721] [SWS_CORE_00722] [SWS_CORE_00723] [SWS_CORE_00724] [SWS_CORE_00725] [SWS_CORE_00726] [SWS_CORE_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_00761] [SWS_CORE_00762] [SWS_CORE_00763] [SWS_CORE_00765] [SWS_CORE_00766] [SWS_CORE_00767] [SWS_CORE_00768]

Requirement	Description	Satisfied by
		[SWS_CORE_00796] [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] [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]

Requirement	Description	Satisfied by
		[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_02001] [SWS_CORE_03001] [SWS_CORE_03296] [SWS_CORE_03301] [SWS_CORE_03302] [SWS_CORE_03303] [SWS_CORE_03304] [SWS_CORE_03305] [SWS_CORE_03306] [SWS_CORE_03307] [SWS_CORE_03308] [SWS_CORE_03309] [SWS_CORE_03310] [SWS_CORE_03311] [SWS_CORE_03312] [SWS_CORE_03313] [SWS_CORE_03314] [SWS_CORE_03315] [SWS_CORE_03316] [SWS_CORE_03317] [SWS_CORE_03318] [SWS_CORE_03319] [SWS_CORE_03320] [SWS_CORE_03321] [SWS_CORE_03322] [SWS_CORE_03323] [SWS_CORE_04011]

Requirement	Description	Satisfied by
		[SWS_CORE_04012] [SWS_CORE_04013] [SWS_CORE_04021] [SWS_CORE_04022] [SWS_CORE_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]
[RS_Main_00011]	AUTOSAR shall support the development of reliable systems	[SWS_CORE_10001] [SWS_CORE_10002]
[RS_Main_00320]	AUTOSAR shall provide formats to specify system development	[SWS_CORE_08001] [SWS_CORE_08021] [SWS_CORE_08022] [SWS_CORE_08029] [SWS_CORE_08031] [SWS_CORE_08041] [SWS_CORE_08042] [SWS_CORE_08043] [SWS_CORE_08044] [SWS_CORE_08045] [SWS_CORE_08046] [SWS_CORE_08090]

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 errors

During execution of an implementation of Adaptive Platform APIs, different error conditions might be detected and need to be handled and/or reported. Based on the nature and the handling of these errors, the following two types of errors can be distinguished:

Fatal Error: This kind of error cannot be handled properly by the Adaptive Application and thus leads to termination of the Adaptive Application. Typical examples for this kind of errors are:

- `Unchecked Exceptions` (according to [3, AUTOSAR CPP14 guidelines]), thrown by internal methods used to implement Adaptive Platform APIs (e.g., due to out of memory situations)
- errors which prevent any further operation of the respective process (e.g., encrypted communication channel not available)
- errors which are effectively configuration or programming errors (e.g., a violated pre- or post-condition of a function)

Non-Fatal Error: This kind of error can and should be handled by the Adaptive Application and usually does not lead to termination of the Adaptive Application. In many cases, this kind of error is even explicitly modeled in the Manifest by means of `Checked Errors` (see section 7.1.3).
An example might be a failure to read data from a network socket.

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 case of failure, i.e. exception-thrown)

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 Checked versus Unchecked Exceptions

In [3, AUTOSAR CPP14 guidelines], there is a strict separation between Checked Exception and Unchecked Exception. These terms are inspired by the Java Programming Language's use of exceptions.

Unchecked Exceptions can occur in almost any API call (except for API calls that are marked with noexcept, of course); they are never formally modeled in the AUTOSAR Manifest and are implementation specific. An Unchecked Exception represents an error from which a program typically cannot recover. It is an instance of Fatal Error, as defined in section 7.1.1.

In contrast, Checked Exceptions only occur in specific API calls, and their occurrence is formally specified in the API. A Checked Exception represents an error that is expected and from which the application can reasonably be expected to recover. It is an instance of Non-Fatal Error, as defined in section 7.1.1.

7.1.4 Error handling in the Adaptive Platform

In contrast to released versions of this document prior to R18-10 of the Adaptive Platform, Checked Exception errors are not directly reported via C++ exceptions. Therefore, they will henceforth be referenced as Checked Errors.

A Checked Error can be viewed as a generalization of Checked Exception; it is an abstract "error" that "contains" both an error code and an exception, and it can be "realized" as either of them. The same holds for Unchecked Error and Unchecked Exception.

Within the Adaptive Platform, synchronous Checked Errors are reported via ara::core::Result objects containing an ara::core::ErrorCode. Asynchronous Checked Errors are reported via ara::core::Future objects that effectively contain ara::core::Result objects.

In case a Fatal Error is detected during execution of Adaptive Platform APIs, the following error handling shall be performed:

[SWS_CORE_00001] Handling of Fatal Errors ⌈ The incident shall be logged (if logging is enabled for the respective Functional Cluster of the Adaptive Platform implementation), and the operation shall be terminated by either:

- raising an Unchecked Exception (which in turn shall lead to the respective process to be terminated by a call to std::terminate(), possibly after execution of cleanup code), or by
- explicitly terminating the process abnormally (e.g., by calling std::terminate or std::abort)

⌋*(RS_AP_00128)*

In case a Non-Fatal Error is detected durign execution of Adaptive Platform APIs, the following error handling shall be performed:

[SWS_CORE_00002] Handling of Non-Fatal Errors ⌈ The error shall be returned from the function as an instance of ara::core::Result. ⌋*(RS_AP_00128)*

7.1.4.1 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.

In addition to that, `ara::core::ErrorCode` can also optionally contain a user-defined message string. This is stored in the form of a non-owning pointer, so the lifetime of this string must match or exceed that of the `ara::core::ErrorCode` object. Typically, a string literal should be used, to ensure it has static storage duration.

`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 user-defined message and the `support data value` members are 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 user-defined message or `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).

7.1.4.2 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::GetFutureDomain`; 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 `PosixErrorDomain` (containing the set of errors defined by the POSIX standard; these are equivalent to `std::errc` from [4]) 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`.

7.1.4.3 Result

The `ara::core::Result` type follows the `ValueOrError` concept from the C++ proposal p0786 [5]. 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`.

7.1.4.4 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 7.1.4.3, 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.

7.1.5 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 workflow. No API function will ever throw a `Checked Exception`; it will always return a `Checked Error` in the form of a `ara::core::Result` return value instead. The class `ara::core::Result` then allows “converting” an `ara::core::ErrorCode` into an exception.

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.

Any `Unchecked Exception`s thrown from code from such a compiler will be transformed into calls to `std::terminate`. As an `Unchecked Error` is not assumed to be recoverable, termination of the application is the logical consequence. Orderly shutdown of the application in this case can be done through a C++ `terminate handler`.

7.1.6 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 an additional layer of exception base types, one for each error domain.

For error domains that are modelled in ARXML, the C++ language binding will generate an exception class in addition to the ErrorDomain subclass (which is described in section 7.1.4.2). 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.

7.1.7 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:

```

// 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 errorCode) const noexcept(
        false) override;
};

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

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

7.1.8 AUTOSAR error domains

[SWS_CORE_00010] 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_00012] The POSIX error domain ⌈ There shall be an error domain `ara::core::PosixErrorDomain` for errors defined by POSIX.1 in terms of `errno` values. It shall have the shortname `Posix` and the identifier 0x8000'0000'0000'0012. ⌋*(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)*

7.2 Advanced data types

7.2.1 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] Errors originating from C++ standard classes ⌈ For the classes Array, Vector, Map, Variant, String, and StringView specified below in terms of the corresponding classes of the C++ standard, all functions that are specified by [4, the C++11 standard], [6, the C++17 standard], or [7, the draft C++20 standard] to throw any exceptions, are instead specified to generate an `Unchecked Error`, which shall be handled as described in [SWS_CORE_00001]. The exact type of a thrown exception is unspecified in these cases. ⌋*(RS_AP_00130)*

7.2.1.1 Types taken from the C++11 standard

These types are already contained in the [4, 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.

7.2.1.2 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.

7.3 Text encoding

The Adaptive Platform strongly encourages developers to use UTF-8 for all non-ASCII text – or, rather, because ASCII is a strict subset of UTF-8, to use UTF-8 for all text resources. The types `ara::core::String` and `ara::core::StringView` are well-suited for this purpose, as they allow arbitrary 8-bit code units as their content. Please note, however, that e.g. `ara::core::String::size()` follows C++ tradition and returns the number of *code units* (i.e. bytes for UTF-8), not anything like the number of *code points* or *grapheme clusters*.

There currently is no support within the `ara::core` namespace for "wide character" strings, including UTF-16-encoded ones. It is recommended to convert any textual data into a `ara::core::String` object with UTF-8 encoding for further processing.

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] ⌈ The class ara::core::ErrorDomain is defined in Table 8.1. ⌋ *(RS_AP_00130)*

Kind:	class
Base class:	None
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.

Table 8.1: class ara::core::ErrorDomain

[SWS_CORE_00121] ⌈ The type alias ara::core::ErrorDomain::IdType is defined in Table 8.2. ⌋ *(RS_AP_00130)*

Kind:	type alias
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 .

Table 8.2: type alias ara::core::ErrorDomain::IdType

[SWS_CORE_00122] ⌈ The type alias ara::core::ErrorDomain::CodeType is defined in Table 8.3. ⌋ *(RS_AP_00130)*

Kind:	type alias
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 .

Table 8.3: type alias ara::core::ErrorDomain::CodeType

[SWS_CORE_00123] ⌈ The type alias ara::core::ErrorDomain::SupportDataType is defined in Table 8.4. ⌋*(RS_AP_00130)*

Kind:	type alias
Scope:	class ara::core::ErrorDomain
Derived from:	std::int32_t
Syntax:	using ara::core::ErrorDomain::SupportDataType = std::int32_t;
Header file:	#include "ara/core/error_domain.h"
Description:	Alias type for vendor-specific supplementary data .

Table 8.4: type alias ara::core::ErrorDomain::SupportDataType

[SWS_CORE_00131] ⌈ The function ara::core::ErrorDomain::ErrorDomain is defined in Table 8.5. ⌋*(RS_AP_00130)*

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

Table 8.5: function ara::core::ErrorDomain::ErrorDomain

[SWS_CORE_00132] ⌈ The function ara::core::ErrorDomain::ErrorDomain is defined in Table 8.6. ⌋*(RS_AP_00130)*

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

Table 8.6: function ara::core::ErrorDomain::ErrorDomain

[SWS_CORE_00135] ⌈ The function ara::core::ErrorDomain::ErrorDomain is defined in Table 8.7. ⌋*(RS_AP_00130)*

Symbol:	ara::core::ErrorDomain::ErrorDomain(IdType id)	
Kind:	function	
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.	

Table 8.7: function ara::core::ErrorDomain::ErrorDomain

[SWS_CORE_00136] ⌈ The function ara::core::ErrorDomain::~ErrorDomain is defined in Table 8.8. ⌋*(RS_AP_00130)*

Symbol:	ara::core::ErrorDomain::~ErrorDomain()
Kind:	function
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.

Table 8.8: function ara::core::ErrorDomain::~ErrorDomain

[SWS_CORE_00133] ⌈ The function ara::core::ErrorDomain::operator= is defined in Table 8.9. ⌋*(RS_AP_00130)*

Symbol:	ara::core::ErrorDomain::operator=(ErrorDomain const &)
Kind:	function
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. .

Table 8.9: function ara::core::ErrorDomain::operator=

[SWS_CORE_00134] ⌈ The function ara::core::ErrorDomain::operator= is defined in Table 8.10. ⌋*(RS_AP_00130)*

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

Table 8.10: function ara::core::ErrorDomain::operator=

[SWS_CORE_00137] ⌈ The function ara::core::ErrorDomain::operator== is defined in Table 8.11. ⌋*(RS_AP_00130)*

Symbol:	ara::core::ErrorDomain::operator==(ErrorDomain const &other)	
Kind:	function	
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.	

Table 8.11: function ara::core::ErrorDomain::operator==

[SWS_CORE_00138] ⌈ The function ara::core::ErrorDomain::operator!= is defined in Table 8.12. ⌋*(RS_AP_00130)*

Symbol:	ara::core::ErrorDomain::operator!=(ErrorDomain const &other)	
Kind:	function	
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.	

Table 8.12: function ara::core::ErrorDomain::operator!=

[SWS_CORE_00151] ⌈ The function ara::core::ErrorDomain::Id is defined in Table 8.13. ⌋*(RS_AP_00130)*

Symbol:	ara::core::ErrorDomain::Id()	
Kind:	function	
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. the identifier	

Table 8.13: function ara::core::ErrorDomain::Id

[SWS_CORE_00152] ⌈ The function ara::core::ErrorDomain::Name is defined in Table 8.14. ⌋*(RS_AP_00130)*

Symbol:	ara::core::ErrorDomain::Name()	
Kind:	function	
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. the name as a null-terminated string, never nullptr	

Table 8.14: function ara::core::ErrorDomain::Name

[SWS_CORE_00153] ⌈ The function ara::core::ErrorDomain::Message is defined in Table 8.15. ⌋*(RS_AP_00130)*

Symbol:	ara::core::ErrorDomain::Message(CodeType errorCode)	
Kind:	function	
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. The return value is undefined if the errorCode did not originate from this error domain. The returned pointer remains owned by class ErrorDomain and shall not be freed by clients.	

Table 8.15: function ara::core::ErrorDomain::Message

[SWS_CORE_00154] ⌈ The function ara::core::ErrorDomain::ThrowAsException is defined in Table 8.16. ⌋*(RS_AP_00130)*

Symbol:	ara::core::ErrorDomain::ThrowAsException(ErrorCode const &errorCode)
Kind:	function
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
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.

Table 8.16: function ara::core::ErrorDomain::ThrowAsException

8.2 `ErrorCode` data type

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

[SWS_CORE_00501] ⌈ The class ara::core::ErrorCode is defined in Table 8.17. ⌋ *(RS_AP_00130)*

Kind:	class
Base class:	None
Syntax:	`class ErrorCode`
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.

Table 8.17: class ara::core::ErrorCode

[SWS_CORE_00511] ⌈ The function ara::core::ErrorCode::ErrorCode is defined in Table 8.18. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::ErrorCode::ErrorCode(EnumT e, ErrorDomain::SupportDataType data=0, char const *userMessage=nullptr)	
Kind:	function	
Scope:	class ara::core::ErrorCode	
Syntax:	`template <typename EnumT> constexpr ErrorCode (EnumT e, ErrorDomain::SupportDataType data=0, char const *userMessage=nullptr) 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
	userMessage	an optional user-defined custom static message text (null-terminated)
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. The lifetime of the text pointed to by userMessage must exceed that of all uses of Error Code::UserMessage() on this instance.	

Table 8.18: function ara::core::ErrorCode::ErrorCode

[SWS_CORE_00512] ⌈ The function ara::core::ErrorCode::ErrorCode is defined in Table 8.19. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::ErrorCode::ErrorCode(EnumT e, char const *userMessage, ErrorDomain::Support DataType data=0)	
Kind:	function	
Scope:	class ara::core::ErrorCode	
Syntax:	`template <typename EnumT>` `constexpr ErrorCode (EnumT e, char const *userMessage, Error Domain::SupportDataType data=0) noexcept;`	
Template param:	EnumT	an enum type that contains error code values
Parameters (in):	e	a domain-specific error code value
	userMessage	a user-defined custom static message text (null-terminated)
	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. The lifetime of the text pointed to by userMessage must exceed that of all uses of Error Code::UserMessage() on this instance. This constructor does not participate in overload resolution unless EnumT is an enum type.	

Table 8.19: function ara::core::ErrorCode::ErrorCode

[SWS_CORE_00513] ⌈ The function ara::core::ErrorCode::ErrorCode is defined in Table 8.20. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::ErrorCode::ErrorCode(ErrorDomain::CodeType value, ErrorDomain const &domain, ErrorDomain::SupportDataType data=0, char const *userMessage=nullptr)
Kind:	function
Scope:	class ara::core::ErrorCode
Syntax:	`constexpr ErrorCode (ErrorDomain::CodeType value, ErrorDomain const &domain, ErrorDomain::SupportDataType data=0, char const *user Message=nullptr) noexcept;`

▽

△

Parameters (in):	value	a domain-specific error code value
	domain	the ErrorDomain associated with value
	data	optional vendor-specific supplementary error context data
	userMessage	a user-defined custom static message text (null-terminated)
Exception Safety:	noexcept	
Header file:	#include "ara/core/error_code.h"	
Description:	Construct a new ErrorCode instance with parameters. The lifetime of the text pointed to by userMessage must exceed that of all uses of Error Code::UserMessage() on this instance.	

Table 8.20: function ara::core::ErrorCode::ErrorCode

[SWS_CORE_00514] ⌈ The function ara::core::ErrorCode::Value is defined in Table 8.21. ⌋*(RS_AP_00130)*

Symbol:	ara::core::ErrorCode::Value()	
Kind:	function	
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. the raw error code value	

Table 8.21: function ara::core::ErrorCode::Value

[SWS_CORE_00515] ⌈ The function ara::core::ErrorCode::Domain is defined in Table 8.22. ⌋*(RS_AP_00130)*

Symbol:	ara::core::ErrorCode::Domain()	
Kind:	function	
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. the ErrorDomain	

Table 8.22: function ara::core::ErrorCode::Domain

[SWS_CORE_00516] ⌈ The function ara::core::ErrorCode::SupportData is defined in Table 8.23. ⌋*(RS_AP_00130)*

Symbol:	ara::core::ErrorCode::SupportData()	
Kind:	function	
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 meaning of the returned value is implementation-defined. the supplementary error context data	

Table 8.23: function ara::core::ErrorCode::SupportData

[SWS_CORE_00517] ⌈ The function ara::core::ErrorCode::UserMessage is defined in Table 8.24. ⌋*(RS_AP_00130)*

Symbol:	ara::core::ErrorCode::UserMessage()	
Kind:	function	
Scope:	class ara::core::ErrorCode	
Syntax:	`constexpr StringView UserMessage () const noexcept;`	
Return value:	StringView	the user-specified message text, or an empty string if none was given
Exception Safety:	noexcept	
Header file:	#include "ara/core/error_code.h"	
Description:	Return the user-specified message text. the user-specified message text, or an empty string if none was given	

Table 8.24: function ara::core::ErrorCode::UserMessage

[SWS_CORE_00518] ⌈ The function ara::core::ErrorCode::Message is defined in Table 8.25. ⌋*(RS_AP_00130)*

Symbol:	ara::core::ErrorCode::Message()	
Kind:	function	
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. the error message text	

Table 8.25: function ara::core::ErrorCode::Message

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] ⌈ The class ara::core::Exception is defined in Table 8.26. ⌋ *(RS_AP_00130)*

Kind:	class
Base class:	exception
Syntax:	`class Exception : public exception`
Header file:	#include "ara/core/exception.h"
Description:	Base type for all AUTOSAR exception types.

Table 8.26: class ara::core::Exception

[SWS_CORE_00611] ⌈ The function ara::core::Exception::Exception is defined in Table 8.27. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::Exception::Exception(ErrorCode err)	
Kind:	function	
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.	

Table 8.27: function ara::core::Exception::Exception

[SWS_CORE_00612] ⌈ The function ara::core::Exception::what is defined in Table 8.28. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::Exception::what()	
Kind:	function	
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. a null-terminated string	

Table 8.28: function ara::core::Exception::what

[SWS_CORE_00613] ⌈ The function ara::core::Exception::Error is defined in Table 8.29. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::Exception::Error()	
Kind:	function	
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. reference to the embedded ErrorCode	

Table 8.29: function ara::core::Exception::Error

8.4 `Result` data type

This section describes the `ara::core::Result<T, E>` type that contains a value of type T or an error of type E.

[SWS_CORE_00701] ⌈ The class ara::core::Result is defined in Table 8.30. ⌋ *(RS_AP_00130)*

Kind:	class	
Base class:	None	
Syntax:	`template <typename T, typename E = ErrorCode>` `class Result`	
Template param:	T	typename
	E = ErrorCode	typename
Header file:	#include "ara/core/result.h"	
Description:	This class is a type that contains either a value or an error.	

Table 8.30: class ara::core::Result

[SWS_CORE_00711] ⌈ The type alias ara::core::Result::value_type is defined in Table 8.31. ⌋ *(RS_AP_00130)*

Kind:	type alias
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 .

Table 8.31: type alias ara::core::Result::value_type

[SWS_CORE_00712] ⌈ The type alias ara::core::Result::error_type is defined in Table 8.32. ⌋ *(RS_AP_00130)*

Kind:	type alias
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 .

Table 8.32: type alias ara::core::Result::error_type

[SWS_CORE_00721] ⌈ The function ara::core::Result::Result is defined in Table 8.33.
⌋(*RS_AP_00130*)

Symbol:	ara::core::Result::Result(T const &t)	
Kind:	function	
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).	

Table 8.33: function ara::core::Result::Result

[SWS_CORE_00722] ⌈ The function ara::core::Result::Result is defined in Table 8.34.
⌋(*RS_AP_00130*)

Symbol:	ara::core::Result::Result(T &&t)	
Kind:	function	
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).	

Table 8.34: function ara::core::Result::Result

[SWS_CORE_00723] ⌈ The function ara::core::Result::Result is defined in Table 8.35.
⌋(*RS_AP_00130*)

Symbol:	ara::core::Result::Result(E const &e)	
Kind:	function	
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).	

Table 8.35: function ara::core::Result::Result

[SWS_CORE_00724] ⌈ The function ara::core::Result::Result is defined in Table 8.36.
⌋*(RS_AP_00130)*

Symbol:	ara::core::Result::Result(E &&e)	
Kind:	function	
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).	

Table 8.36: function ara::core::Result::Result

[SWS_CORE_00725] ⌈ The function ara::core::Result::Result is defined in Table 8.37.
⌋*(RS_AP_00130)*

Symbol:	ara::core::Result::Result(Result const &other)	
Kind:	function	
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.	

Table 8.37: function ara::core::Result::Result

[SWS_CORE_00726] ⌈ The function ara::core::Result::Result is defined in Table 8.38.
⌋*(RS_AP_00130)*

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

Table 8.38: function ara::core::Result::Result

[SWS_CORE_00731] ⌈ The function ara::core::Result::FromValue is defined in Table 8.39. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Result::FromValue(T const &t)
Kind:	function
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).	

Table 8.39: function ara::core::Result::FromValue

[SWS_CORE_00732] ⌈ The function ara::core::Result::FromValue is defined in Table 8.40. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Result::FromValue(T &&t)	
Kind:	function	
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).	

Table 8.40: function ara::core::Result::FromValue

[SWS_CORE_00733] ⌈ The function ara::core::Result::FromValue is defined in Table 8.41. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Result::FromValue(Args &&...args)	
Kind:	function	
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, andthe first type of the expanded parameter pack is not a specialization of Result	

Table 8.41: function ara::core::Result::FromValue

[SWS_CORE_00734] ⌈ The function ara::core::Result::FromError is defined in Table 8.42. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Result::FromError(E const &e)	
Kind:	function	
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).	

Table 8.42: function ara::core::Result::FromError

[SWS_CORE_00735] ⌈ The function ara::core::Result::FromError is defined in Table 8.43. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Result::FromError(E &&e)	
Kind:	function	
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).	

Table 8.43: function ara::core::Result::FromError

[SWS_CORE_00736] ⌈ The function ara::core::Result::FromError is defined in Table 8.44. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Result::FromError(Args &&...args)	
Kind:	function	
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, andthe first type of the expanded parameter pack is not a specialization of Result	

Table 8.44: function ara::core::Result::FromError

[SWS_CORE_00741] ⌈ The function ara::core::Result::operator= is defined in Table 8.45. ⌋*()*

Symbol:	ara::core::Result::operator=(Result const &other)	
Kind:	function	
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.	

Table 8.45: function ara::core::Result::operator=

[SWS_CORE_00742] ⌈ The function ara::core::Result::operator= is defined in Table 8.46. ⌋*(RS_AP_00130)*

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

Table 8.46: function ara::core::Result::operator=

[SWS_CORE_00743] ⌈ The function ara::core::Result::EmplaceValue is defined in Table 8.47. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Result::EmplaceValue(Args &&...args)	
Kind:	function	
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.	

Table 8.47: function ara::core::Result::EmplaceValue

[SWS_CORE_00744] ⌈ The function ara::core::Result::EmplaceError is defined in Table 8.48. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Result::EmplaceError(Args &&...args)	
Kind:	function	
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.	

Table 8.48: function ara::core::Result::EmplaceError

[SWS_CORE_00745] ⌈ The function ara::core::Result::Swap is defined in Table 8.49. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Result::Swap(Result &other)	
Kind:	function	
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);`	

Parameters (inout):	other	the other instance
Return value:	None	
Exception Safety:	noexcept	
Header file:	#include "ara/core/result.h"	
Description:	Exchange the contents of this instance with those of other.	

Table 8.49: function ara::core::Result::Swap

[SWS_CORE_00751] ⌈ The function ara::core::Result::HasValue is defined in Table 8.50. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Result::HasValue()	
Kind:	function	
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. true if *this contains a value, false otherwise	

Table 8.50: function ara::core::Result::HasValue

[SWS_CORE_00752] ⌈ The function ara::core::Result::operator bool is defined in Table 8.51. ⌋*(RS_AP_00130)*

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

Table 8.51: function ara::core::Result::operator bool

[SWS_CORE_00753] ⌈ The function ara::core::Result::operator* is defined in Table 8.52. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Result::operator*()	
Kind:	function	
Scope:	class ara::core::Result	
Syntax:	`T const& ara::core::Result< T, E >::operator* () const ;`	
Return value:	T const &	a 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. a reference to the contained value	

Table 8.52: function ara::core::Result::operator*

[SWS_CORE_00754] ⌈ The function ara::core::Result::operator-> is defined in Table 8.53. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Result::operator->()	
Kind:	function	
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. a pointer to the contained value	

Table 8.53: function ara::core::Result::operator->

[SWS_CORE_00755] ⌈ The function ara::core::Result::Value is defined in Table 8.54. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Result::Value()	
Kind:	function	
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. a const reference to the contained value	

Table 8.54: function ara::core::Result::Value

[SWS_CORE_00756] ⌈ The function ara::core::Result::Value is defined in Table 8.55. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::Result::Value()	
Kind:	function	
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. an rvalue reference to the contained value	

Table 8.55: function ara::core::Result::Value

[SWS_CORE_00757] ⌈ The function ara::core::Result::Error is defined in Table 8.56. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::Result::Error()	
Kind:	function	
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. a const reference to the contained error	

Table 8.56: function ara::core::Result::Error

[SWS_CORE_00758] ⌈ The function ara::core::Result::Error is defined in Table 8.57. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::Result::Error()	
Kind:	function	
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. an rvalue reference to the contained error	

Table 8.57: function ara::core::Result::Error

[SWS_CORE_00761] ⌈ The function ara::core::Result::ValueOr is defined in Table 8.58. ⌋*(RS_AP_00130)*

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

Table 8.58: function ara::core::Result::ValueOr

[SWS_CORE_00762] ⌈ The function ara::core::Result::ValueOr is defined in Table 8.59. ⌋*(RS_AP_00130)*

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

Table 8.59: function ara::core::Result::ValueOr

[SWS_CORE_00763] ⌈ The function ara::core::Result::ErrorOr is defined in Table 8.60. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Result::ErrorOr(G &&defaultError)	
Kind:	function	
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.	

Table 8.60: function ara::core::Result::ErrorOr

[SWS_CORE_00765] ⌈ The function ara::core::Result::CheckError is defined in Table 8.61. ⌋*(RS_AP_00130)*

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

Table 8.61: function ara::core::Result::CheckError

[SWS_CORE_00766] ⌈ The function ara::core::Result::ValueOrThrow is defined in Table 8.62. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Result::ValueOrThrow()	
Kind:	function	
Scope:	class ara::core::Result	
Syntax:	`T const& ara::core::Result< T, E >::ValueOrThrow () const noexcept(false);`	
Return value:	T const &	the 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.

Table 8.62: function ara::core::Result::ValueOrThrow

[SWS_CORE_00767] ⌈ The function ara::core::Result::Resolve is defined in Table 8.63. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Result::Resolve(F &&f)	
Kind:	function	
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&);	

Table 8.63: function ara::core::Result::Resolve

[SWS_CORE_00768] ⌈ The function ara::core::Result::Bind is defined in Table 8.64. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::Result::Bind(F &&f)	
Kind:	function	
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:	auto	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.

Table 8.64: function ara::core::Result::Bind

[SWS_CORE_00796] ⌈ The function ara::core::swap is defined in Table 8.65. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::swap(Result< T, E > &lhs, Result< T, E > &rhs)	
Kind:	function	
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:	noexcept	
Header file:	#include "ara/core/result.h"	
Description:	Swap the contents of the two given arguments.	

Table 8.65: function ara::core::swap

8.5 Posix Error Domain

This section describes the `ara::core::PosixErrorDomain` type that derives from `ara::core::ErrorDomain` and contains the `errno`-based errors that can occur in a POSIX system. It is a reformulation of std::generic_category.

8.5.1 POSIX error codes

[SWS_CORE_00200] ⌈ The enum ara::core::PosixErrc is defined in Table 8.66. ⌋ *(RS_AP_00130)*

Kind:	enum	
Values:	address_family_not_supported= 97	address family not supported (EAFNOSUPPORT)
	address_in_use= 98	address in use (EADDRINUSE)
	address_not_available= 99	address not available (EADDRNOTAVAIL)

▽

△

	already_connected= 106	already conntected (EISCONN)
	argument_list_too_long= 7	argument list too long (E2BIG)
	argument_out_of_domain= 33	argument out of domain (EDOM)
	bad_address= 14	bad address (EFAULT)
	bad_file_descriptor= 9	bad file descriptor (EBADF)
	bad_message= 74	bad message (EBADMSG)
	broken_pipe= 32	broken pipe (EPIPE)
	connection_aborted= 103	connection aborted (ECONNABORTED)
	connection_already_in_progress= 114	connection already in progress (EALREADY)
	connection_refused= 111	connection refused (ECONNREFUSED)
	connection_reset= 104	connection reset (ECONNRESET)
	cross_device_link= 18	cross-device link (EXDEV)
	destination_address_required= 89	destination address required (EDESTADDRREQ)
	device_or_resource_busy= 16	device or resource busy (EBUSY)
	directory_not_empty= 39	directory not empty (ENOTEMPTY)
	executable_format_error= 8	executable format error (ENOEXEC)
	file_exists= 17	file exists (EEXIST)
	file_too_large= 27	file too large (EFBIG)
	filename_too_long= 36	filename too long (ENAMETOOLONG)
	function_not_supported= 38	function not supported (ENOSYS)
	host_unreachable= 113	host unreachable (EHOSTUNREACH)
	identifier_removed= 43	identifier removed (EIDRM)
	illegal_byte_sequence= 84	illegal byte sequence (EILSEQ)
	inappropriate_io_control_operation= 25	inappropriate io control operation (ENOTTY)
	interrupted= 4	interrupted (EINTR)
	invalid_argument= 22	invalid argument (EINVAL)
	invalid_seek= 29	invalid seek (ESPIPE)
	io_error= 5	I/O error (EIO)
	is_a_directory= 21	is a directory (EISDIR)
	message_size= 90	message size (EMSGSIZE)
	network_down= 100	network down (ENETDOWN)
	network_reset= 102	network reset (ENETRESET)
	network_unreachable= 101	network unreachable (ENETUNREACH)
	no_buffer_space= 105	no buffer space (ENOBUFS)
	no_child_process= 10	no child process (ECHILD)
	no_link= 67	no link (ENOLINK)
	no_lock_available= 37	no lock available (ENOLCK)
	no_message_available= 61	no message available (ENODATA)
	no_message= 42	no message (ENOMSG)
	no_protocol_option= 92	no protocol option (ENOPROTOOPT)

▽

△

	no_space_on_device= 28	no space on device (ENOSPC)
	no_stream_resources= 63	no stream resources (ENOSR)
	no_such_device_or_address= 6	no such device or address (ENXIO)
	no_such_device= 19	no such device (ENODEV)
	no_such_file_or_directory= 2	no such file or directory (ENOENT)
	no_such_process= 3	no such process (ESRCH)
	not_a_directory= 20	not a directory (ENOTDIR)
	not_a_socket= 88	not a socket (ENOTSOCK)
	not_a_stream= 60	not a stream (ENOSTR)
	not_connected= 107	not connected (ENOTCONN)
	not_enough_memory= 12	not enough memory (ENOMEM)
	not_supported= 202	not supported (ENOTSUP)
	operation_canceled= 125	operation canceled (ECANCELED)
	operation_in_progress= 115	operation in progress (EINPROGRESS)
	operation_not_permitted= 1	operation not permitted (EPERM)
	operation_not_supported= 95	operation not supported (EOPNOTSUPP)
	operation_would_block= 201	operation would block (EWOULDBLOCK)
	owner_dead= 130	owner dead (EOWNERDEAD)
	permission_denied= 13	permission denied (EACCES)
	protocol_error= 71	protocol error (EPROTO)
	protocol_not_supported= 93	protocol not supported (EPROTONOSUPPORT)
	read_only_file_system= 30	read-only file system (EROFS)
	resource_deadlock_would_occur= 35	resource deadlock would occur (EDEADLK)
	resource_unavailable_try_again= 11	resource unavailable, try again (EAGAIN)
	result_out_of_range= 34	result out of range (ERANGE)
	state_not_recoverable= 131	state not recoverable (ENOTRECOVERABLE)
	stream_timeout= 62	stream timeout (ETIME)
	text_file_busy= 26	text file busy (ETXTBSY)
	timed_out= 110	timed out (ETIMEDOUT)
	too_many_files_open_in_system= 23	too many files open in system (ENFILE)
	too_many_files_open= 24	too many files open (EMFILE)
	too_many_links= 31	too many links (EMLINK)
	too_many_symbolic_link_levels= 40	too many symbolic link levels (ELOOP)
	value_too_large= 75	value too large (EOVERFLOW)
	wrong_protocol_type= 91	wrong protocol type (EPROTOTYPE)
Header file:	#include "ara/core/posix_error_domain.h"	
Description:	An enumeration that defines all errors from POSIX.1.	

Table 8.66: enum ara::core::PosixErrc

8.5.2 `PosixException` type

[SWS_CORE_00211] ⌈ The class ara::core::PosixException is defined in Table 8.67. ⌋ *(RS_AP_00130)*

Kind:	class
Base class:	ara::core::Exception
Syntax:	`class PosixException : public Exception`
Header file:	#include "ara/core/posix_error_domain.h"
Description:	Exception type thrown for POSIX errors.

Table 8.67: class ara::core::PosixException

[SWS_CORE_00212] ⌈ The function ara::core::PosixException::PosixException is defined in Table 8.68. ⌋*(RS_AP_00130)*

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

Table 8.68: function ara::core::PosixException::PosixException

8.5.3 `PosixErrorDomain` type

[SWS_CORE_00221] ⌈ The class ara::core::PosixErrorDomain is defined in Table 8.69. ⌋*(RS_AP_00130)*

Kind:	class
Base class:	ara::core::ErrorDomain
Syntax:	`class PosixErrorDomain final : public ErrorDomain`
Header file:	#include "ara/core/posix_error_domain.h"
Description:	An error domain for POSIX.1 errors.

Table 8.69: class ara::core::PosixErrorDomain

[SWS_CORE_00231] ⌈ The type alias ara::core::PosixErrorDomain::Errc is defined in Table 8.70. ⌋*(RS_AP_00130)*

Kind:	type alias
Scope:	class ara::core::PosixErrorDomain
Derived from:	PosixErrc
Syntax:	using ara::core::PosixErrorDomain::Errc = PosixErrc;
Header file:	#include "ara/core/posix_error_domain.h"

▽

△

Description:	Alias for the error code value enumeration.

Table 8.70: type alias ara::core::PosixErrorDomain::Errc

[SWS_CORE_00232] ⌈ The type alias ara::core::PosixErrorDomain::Exception is defined in Table 8.71. ⌋*(RS_AP_00130)*

Kind:	type alias
Scope:	class ara::core::PosixErrorDomain
Derived from:	PosixException
Syntax:	using ara::core::PosixErrorDomain::Exception = PosixException;
Header file:	#include "ara/core/posix_error_domain.h"
Description:	Alias for the exception base class.

Table 8.71: type alias ara::core::PosixErrorDomain::Exception

[SWS_CORE_00241] ⌈ The function ara::core::PosixErrorDomain::PosixErrorDomain is defined in Table 8.72. ⌋*(RS_AP_00130)*

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

Table 8.72: function ara::core::PosixErrorDomain::PosixErrorDomain

[SWS_CORE_00242] ⌈ The function ara::core::PosixErrorDomain::Name is defined in Table 8.73. ⌋*(RS_AP_00130)*

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

Table 8.73: function ara::core::PosixErrorDomain::Name

[SWS_CORE_00243] ⌈ The function ara::core::PosixErrorDomain::Message is defined in Table 8.74. ⌋*(RS_AP_00130)*

Symbol:	ara::core::PosixErrorDomain::Message(ErrorDomain::CodeType errorCode)	
Kind:	function	
Scope:	class ara::core::PosixErrorDomain	
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/posix_error_domain.h"	
Description:	Translate an error code value into a text message.	

Table 8.74: function ara::core::PosixErrorDomain::Message

[SWS_CORE_00244] ⌈ The function ara::core::PosixErrorDomain::ThrowAsException is defined in Table 8.75. ⌋*(RS_AP_00130)*

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

Table 8.75: function ara::core::PosixErrorDomain::ThrowAsException

8.5.4 `GetPosixDomain` accessor function

[SWS_CORE_00280] ⌈ The function ara::core::GetPosixDomain is defined in Table 8.76. ⌋*(RS_AP_00130)*

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

Table 8.76: function ara::core::GetPosixDomain

8.5.5 `MakeErrorCode` overload for `PosixErrorDomain`

[SWS_CORE_00290] ⌈ The function ara::core::MakeErrorCode is defined in Table 8.77. ⌋*(RS_AP_00130)*

Symbol:	ara::core::MakeErrorCode(PosixErrc code, ErrorDomain::SupportDataType data, char const *message)	
Kind:	function	
Scope:	namespace ara::core	
Syntax:	`constexpr ErrorCode MakeErrorCode (PosixErrc code, ErrorDomain::SupportDataType data, char const *message) noexcept;`	
Parameters (in):	code	the PosixErrorDomain-specific error code value
	data	optional vendor-specific error data
	message	optional user-provided message string
Return value:	ErrorCode	a new ErrorCode instance
Exception Safety:	noexcept	
Header file:	#include "ara/core/posix_error_domain.h"	
Description:	Create a new ErrorCode within PosixErrorDomain. This function is used internally by constructors of ErrorCode. It is usually not used directly by users. The lifetime of the text pointed to by message must exceed that of all uses of ErrorCode::UserMessage() on the new instance.	

Table 8.77: function ara::core::MakeErrorCode

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 [4]. 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 [4], 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] ⌈ The enum ara::core::future_errc is defined in Table 8.78. ⌋ *(RS_AP_00130)*

Kind:	enum	
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.	

Table 8.78: enum ara::core::future_errc

8.6.2 `FutureException` type

[SWS_CORE_00411] ⌈ The class ara::core::FutureException is defined in Table 8.79. ⌋ *(RS_AP_00130)*

Kind:	class
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.

Table 8.79: class ara::core::FutureException

[SWS_CORE_00412] ⌈ The function ara::core::FutureException::FutureException is defined in Table 8.80. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::FutureException::FutureException(ErrorCode err)	
Kind:	function	
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.	

Table 8.80: function ara::core::FutureException::FutureException

8.6.3 `FutureErrorDomain` type

[SWS_CORE_00421] ⌈ The class ara::core::FutureErrorDomain is defined in Table 8.81. ⌋*(RS_AP_00130)*

Kind:	class
Base class:	ara::core::ErrorDomain
Syntax:	`class FutureErrorDomain final : public ErrorDomain`
Header file:	#include "ara/core/future_error_domain.h"
Description:	Error domain for errors originating from classes Future and Promise.

Table 8.81: class ara::core::FutureErrorDomain

[SWS_CORE_00431] ⌈ The type alias ara::core::FutureErrorDomain::Errc is defined in Table 8.82. ⌋*(RS_AP_00130)*

Kind:	type alias
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.

Table 8.82: type alias ara::core::FutureErrorDomain::Errc

[SWS_CORE_00432] ⌈ The type alias ara::core::FutureErrorDomain::Exception is defined in Table 8.83. ⌋*(RS_AP_00130)*

Kind:	type alias
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.

Table 8.83: type alias ara::core::FutureErrorDomain::Exception

[SWS_CORE_00441] ⌈ The function ara::core::FutureErrorDomain::FutureErrorDomain is defined in Table 8.84. ⌋*(RS_AP_00130)*

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

▽

△

Description:	Default constructor.

Table 8.84: function ara::core::FutureErrorDomain::FutureErrorDomain

[SWS_CORE_00442] ⌈ The function ara::core::FutureErrorDomain::Name is defined in Table 8.85. ⌋*(RS_AP_00130)*

Symbol:	ara::core::FutureErrorDomain::Name()	
Kind:	function	
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. "Future"	

Table 8.85: function ara::core::FutureErrorDomain::Name

[SWS_CORE_00443] ⌈ The function ara::core::FutureErrorDomain::Message is defined in Table 8.86. ⌋*(RS_AP_00130)*

Symbol:	ara::core::FutureErrorDomain::Message(ErrorDomain::CodeType errorCode)	
Kind:	function	
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.	

Table 8.86: function ara::core::FutureErrorDomain::Message

[SWS_CORE_00444] ⌈ The function ara::core::FutureErrorDomain::ThrowAsException is defined in Table 8.87. ⌋*(RS_AP_00130)*

Symbol:	ara::core::FutureErrorDomain::ThrowAsException(ErrorCode const &errorCode)	
Kind:	function	
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
Header file:	#include "ara/core/future_error_domain.h"
Description:	Throw the exception type corresponding to the given ErrorCode.

Table 8.87: function ara::core::FutureErrorDomain::ThrowAsException

8.6.4 `GetFutureDomain` accessor function

[SWS_CORE_00480] ⌈ The function ara::core::GetFutureDomain is defined in Table 8.88. ⌋*(RS_AP_00130)*

Symbol:	ara::core::GetFutureDomain()	
Kind:	function	
Scope:	namespace ara::core	
Syntax:	`constexpr ErrorDomain const& GetFutureDomain () 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. reference to the FutureErrorDomain instance	

Table 8.88: function ara::core::GetFutureDomain

8.6.5 `MakeErrorCode` overload for `FutureErrorDomain`

[SWS_CORE_00490] ⌈ The function ara::core::MakeErrorCode is defined in Table 8.89. ⌋*(RS_AP_00130)*

Symbol:	ara::core::MakeErrorCode(future_errc code, ErrorDomain::SupportDataType data, char const *message)	
Kind:	function	
Scope:	namespace ara::core	
Syntax:	`constexpr ErrorCode MakeErrorCode (future_errc code, ErrorDomain::SupportDataType data, char const *message) noexcept;`	
Parameters (in):	code	an enumeration value from future_errc
	data	a vendor-defined supplementary value
	message	a user-defined context message (can be nullptr)
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 and message. The lifetime of the text pointed to by message must exceed that of all uses of ErrorCode::UserMessage() on the new instance.	

Table 8.89: function ara::core::MakeErrorCode

8.6.6 `future_status` enumeration

[SWS_CORE_00361] ⌈ The enum ara::core::future_status is defined in Table 8.90. ⌋ *(RS_AP_00130)*

Kind:	enum	
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.	

Table 8.90: enum ara::core::future_status

8.6.7 `Future` data type

[SWS_CORE_00321] ⌈ The class ara::core::Future is defined in Table 8.91. ⌋ *(RS_AP_00130)*

Kind:	class	
Base class:	None	
Syntax:	`template <typename T, typename E = ErrorCode>` `class Future`	
Template param:	T	typename
	E = ErrorCode	typename
Header file:	#include "ara/core/future.h"	
Description:	Provides ara::core specific Future operations to collect the results of an asynchronous call.	

Table 8.91: class ara::core::Future

[SWS_CORE_00322] ⌈ The function ara::core::Future::Future is defined in Table 8.92. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::Future::Future()
Kind:	function
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.

Table 8.92: function ara::core::Future::Future

[SWS_CORE_00334] ⌈ The function ara::core::Future::Future is defined in Table 8.93. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::Future::Future(Future const &)
Kind:	function
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.

Table 8.93: function ara::core::Future::Future

[SWS_CORE_00323] ⌈ The function ara::core::Future::Future is defined in Table 8.94.
⌋*(RS_AP_00130)*

Symbol:	ara::core::Future::Future(Future &&other)	
Kind:	function	
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.	

Table 8.94: function ara::core::Future::Future

[SWS_CORE_00333] ⌈ The function ara::core::Future::~Future is defined in Table 8.95. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Future::~Future()
Kind:	function
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.

Table 8.95: function ara::core::Future::~Future

[SWS_CORE_00335] ⌈ The function ara::core::Future::operator= is defined in Table 8.96. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Future::operator=(Future const &)
Kind:	function
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.

Table 8.96: function ara::core::Future::operator=

[SWS_CORE_00325] ⌈ The function ara::core::Future::operator= is defined in Table 8.97. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Future::operator=(Future &&other)	
Kind:	function	
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.	

Table 8.97: function ara::core::Future::operator=

[SWS_CORE_00326] ⌈ The function ara::core::Future::get is defined in Table 8.98. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Future::get()	
Kind:	function	
Scope:	class ara::core::Future	
Syntax:	`T ara::core::Future< T, E >::get ();`	
Return value:	T	value of type T Domain:error the error that has been put into the corresponding Promise via Promise::Set Error
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. value of type T Domain:error the error that has been put into the corresponding Promise via Promise::SetError	

Table 8.98: function ara::core::Future::get

[SWS_CORE_00336] ⌈ The function ara::core::Future::GetResult is defined in Table 8.99. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Future::GetResult()	
Kind:	function	
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 Domain:error the error that has been put into the corresponding Promise via Promise::SetError
Exception Safety:	noexcept	
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. a Result with either a value or an error Domain:error the error that has been put into the corresponding Promise via Promise::SetError	

Table 8.99: function ara::core::Future::GetResult

[SWS_CORE_00327] ⌈ The function ara::core::Future::valid is defined in Table 8.100. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Future::valid()	
Kind:	function	
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. true if the Future is usable, false otherwise	

Table 8.100: function ara::core::Future::valid

[SWS_CORE_00328] ⌈ The function ara::core::Future::wait is defined in Table 8.101. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Future::wait()
Kind:	function
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.

Table 8.101: function ara::core::Future::wait

[SWS_CORE_00329] ⌈ The function ara::core::Future::wait_for is defined in Table 8.102. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Future::wait_for(std::chrono::duration< Rep, Period > const &timeoutDuration)	
Kind:	function	
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.	

Table 8.102: function ara::core::Future::wait_for

[SWS_CORE_00330] ⌈ The function ara::core::Future::wait_until is defined in Table 8.103. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Future::wait_until(std::chrono::time_point< Clock, Duration > const &deadline)	
Kind:	function	
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.	

Table 8.103: function ara::core::Future::wait_until

[SWS_CORE_00331] ⌈ The function ara::core::Future::then is defined in Table 8.104. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Future::then(F &&func)	
Kind:	function	
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:	auto	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>.

Table 8.104: function ara::core::Future::then

[SWS_CORE_00332] ⌈ The function ara::core::Future::is_ready is defined in Table 8.105. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Future::is_ready()	
Kind:	function	
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. true if the Future contains a value or an error, false otherwise	

Table 8.105: function ara::core::Future::is_ready

8.6.8 `Promise` data type

[SWS_CORE_00340] ⌈ The class ara::core::Promise is defined in Table 8.106. ⌋*(RS_AP_00130)*

Kind:	class	
Base class:	None	
Syntax:	`template <typename T , typename E = ErrorCode>` `class Promise`	
Template param:		typename T
	= ErrorCode	typename E
Header file:	#include "ara/core/future.h"	
Description:	ara::core specific variant of std::promise class	

Table 8.106: class ara::core::Promise

[SWS_CORE_00341] ⌈ The function ara::core::Promise::Promise is defined in Table 8.107. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Promise::Promise()
Kind:	function
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.

Table 8.107: function ara::core::Promise::Promise

[SWS_CORE_00350] ⌈ The function ara::core::Promise::Promise is defined in Table 8.108. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Promise::Promise(Promise const &)	
Kind:	function	
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.	

Table 8.108: function ara::core::Promise::Promise

[SWS_CORE_00342] ⌈ The function ara::core::Promise::Promise is defined in Table 8.109. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Promise::Promise(Promise &&other)	
Kind:	function	
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.	

Table 8.109: function ara::core::Promise::Promise

[SWS_CORE_00349] ⌈ The function ara::core::Promise::~Promise is defined in Table 8.110. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Promise::~Promise()
Kind:	function
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.

Table 8.110: function ara::core::Promise::~Promise

[SWS_CORE_00351] ⌈ The function ara::core::Promise::operator= is defined in Table 8.111. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Promise::operator=(Promise const &)
Kind:	function
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.

Table 8.111: function ara::core::Promise::operator=

[SWS_CORE_00343] ⌈ The function ara::core::Promise::operator= is defined in Table 8.112. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Promise::operator=(Promise &&other)	
Kind:	function	
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.	

Table 8.112: function ara::core::Promise::operator=

[SWS_CORE_00352] ⌈ The function ara::core::Promise::swap is defined in Table 8.113. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Promise::swap(Promise &other)	
Kind:	function	
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.

Table 8.113: function ara::core::Promise::swap

[SWS_CORE_00344] ⌈ The function ara::core::Promise::get_future is defined in Table 8.114. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Promise::get_future()	
Kind:	function	
Scope:	class ara::core::Promise	
Syntax:	`Future<T, E> ara::core::Promise< T, E >::get_future ();`	
Return value:	Future< T, E >	a Future for types T and E
Header file:	#include "ara/core/promise.h"	
Description:	Return the associated Future for type T and E. 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. a Future for types T and E	

Table 8.114: function ara::core::Promise::get_future

[SWS_CORE_00345] ⌈ The function ara::core::Promise::set_value is defined in Table 8.115. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Promise::set_value(T const &value)	
Kind:	function	
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 result into the Future. This function shall behave the same as the corresponding std::promise function.	

Table 8.115: function ara::core::Promise::set_value

[SWS_CORE_00346] ⌈ The function ara::core::Promise::set_value is defined in Table 8.116. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Promise::set_value(T &&value)
Kind:	function
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 the result into the Future. This function shall behave the same as the corresponding std::promise function.	

Table 8.116: function ara::core::Promise::set_value

[SWS_CORE_00353] ⌈ The function ara::core::Promise::SetError is defined in Table 8.117. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Promise::SetError(E &&error)	
Kind:	function	
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 Future.	

Table 8.117: function ara::core::Promise::SetError

[SWS_CORE_00354] ⌈ The function ara::core::Promise::SetError is defined in Table 8.118. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Promise::SetError(E const &error)	
Kind:	function	
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 Future.	

Table 8.118: function ara::core::Promise::SetError

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] `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 [4] section 23.3. All supporting symbols shall be contained within namespace `ara::core`.

[SWS_CORE_01296] `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] `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 [4] section 23.3.6, except that the default value for the `Allocator` template argument is implementation-defined.

[SWS_CORE_01390] 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] 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] 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] 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] 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] 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] `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] `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 [4] 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] `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] `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] `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] class bad_optional_access ⌈ No class named `bad_optional_access` is defined in the `ara::core` namespace. ⌋*(RS_AP_00130)*

[SWS_CORE_01096] `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] `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] `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] `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 type

This section describes the `ara::core::String` type that represents sequences of characters.

This class is closely modeled on `std::string` with a number of additions which come from [6].

[SWS_CORE_03001] `String` class ⌈ The namespace `ara::core` shall provide a class `String`:

```
class String { ... };
```

⌋*(RS_AP_00130)*

All members of this class and supporting constructs (such as global relational operators) shall behave identical to those of `std::string` in header `<string>` from [4] section 21.3, except that this class's Allocator is implementation-defined. All supporting symbols shall be contained within namespace `ara::core`.

[SWS_CORE_03301] Implicit conversion to `StringView` ⌈ An operator shall be defined that provides implicit conversion to `StringView`:

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

⌋*(RS_AP_00130)*

[SWS_CORE_03302] Constructor from `StringView` ⌈ A constructor shall be defined that accepts a `StringView` argument by value:

```
explicit String(StringView sv);
```

⌋(*RS_AP_00130*)

[SWS_CORE_03303] Constructor from implicit `StringView` ⌈ A constructor shall be defined that accepts any type that is implicitly convertible to `StringView`:

```
template <typename T>
String(T const& t, size_type pos, size_type n);
```

⌋(*RS_AP_00130*)

[SWS_CORE_03304] operator= from `StringView` ⌈ An operator= member function shall be defined that accepts a `StringView` argument by value:

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

⌋(*RS_AP_00130*)

[SWS_CORE_03305] Assignment from `StringView` ⌈ A member function shall be defined that allows assignment from `StringView`:

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

⌋(*RS_AP_00130*)

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

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

⌋(*RS_AP_00130*)

[SWS_CORE_03307] operator+ from `StringView` ⌈ An operator+= member function shall be defined that accepts a `StringView` argument by value:

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

⌋(*RS_AP_00130*)

[SWS_CORE_03308] Concatenation of `StringView` ⌈ A member function shall be defined that allows concatenation of a `StringView`:

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

⌋(*RS_AP_00130*)

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

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

⌋(*RS_AP_00130*)

[SWS_CORE_03310] Insertion of `StringView` ⌈ A member function shall be defined that allows insertion of a `StringView`:

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

⌋*(RS_AP_00130)*

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

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

⌋*(RS_AP_00130)*

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

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

⌋*(RS_AP_00130)*

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

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

⌋*(RS_AP_00130)*

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

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

⌋*(RS_AP_00130)*

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

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

⌋*(RS_AP_00130)*

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

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

⌋*(RS_AP_00130)*

[SWS_CORE_03317] Forward-find of character set within a `StringView` ⌈ A member function shall be defined 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;
```

⌋*(RS_AP_00130)*

[SWS_CORE_03318] Reverse-find of character set within a `StringView` ⌈ A member function shall be defined 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;
```

⌋*(RS_AP_00130)*

[SWS_CORE_03319] Forward-find of character set not within a `StringView` ⌈ A member function shall be defined 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;
```

⌋*(RS_AP_00130)*

[SWS_CORE_03320] Reverse-find of character set not within a `StringView` ⌈ A member function shall be defined 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;
```

⌋*(RS_AP_00130)*

[SWS_CORE_03321] Comparison with a `StringView` ⌈ A member function shall be defined that allows comparison with the contents of a `StringView`:

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

⌋*(RS_AP_00130)*

[SWS_CORE_03322] Comparison of subsequence with a `StringView` ⌈ A member function shall be defined 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;
```

⌋*(RS_AP_00130)*

[SWS_CORE_03323] Comparison of subsequence with a subsequence of a `StringView` ⌈ A member function shall be defined 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;
```

⌋(*RS_AP_00130*)

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

```
void swap(String& lhs, String& 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 21.7, but has been adapted in several ways:

- `Span::operator()` has been removed.
- All symbols from section 21.7.3.7 (`span.comparison`) have been removed.
- All symbols from section 21.7.3.8 (`span.objectrep`) have been removed.
- 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 slight performance degradation.
- `constexpr` has been removed 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] ⌈ The variable ara::core::dynamic_extent is defined in Table 8.119. ⌋(*RS_AP_00130*)

Kind:	variable
Type:	constexpr std::ptrdiff_t
Scope:	namespace ara::core
Syntax:	constexpr std::ptrdiff_t ara::core::dynamic_extent;
Header file:	#include "ara/core/span.h"
Description:	A constant for creating Spans with dynamic sizes. The constant is always set to std::ptrdiff_t(-1).

Table 8.119: variable ara::core::dynamic_extent

[SWS_CORE_01900] ⌈ The class ara::core::Span is defined in Table 8.120. ⌋ *(RS_AP_00130)*

Kind:	class	
Base class:	None	
Syntax:	`template <typename T, std::ptrdiff_t Extent = dynamic_extent> class Span`	
Public fields:	`constexpr index_type extent`	A constant reflecting the configured Extent of this Span.
Template param:	T	typename
	Extent = dynamic_extent	std::ptrdiff_t
Header file:	#include "ara/core/span.h"	
Description:	A view over a contiguous sequence of objects.	

Table 8.120: class ara::core::Span

[SWS_CORE_01911] ⌈ The type alias ara::core::Span::element_type is defined in Table 8.121. ⌋ *(RS_AP_00130)*

Kind:	type alias
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.

Table 8.121: type alias ara::core::Span::element_type

[SWS_CORE_01912] ⌈ The type alias ara::core::Span::value_type is defined in Table 8.122. ⌋ *(RS_AP_00130)*

Kind:	type alias
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.

Table 8.122: type alias ara::core::Span::value_type

[SWS_CORE_01913] ⌈ The type alias ara::core::Span::index_type is defined in Table 8.123. ⌋ *(RS_AP_00130)*

Kind:	type alias
Scope:	class ara::core::Span
Derived from:	std::ptrdiff_t
Syntax:	using ara::core::Span< T, Extent >::index_type = std::ptrdiff_t;
Header file:	#include "ara/core/span.h"
Description:	Alias for the type of parameters that indicate an index into the Span.

Table 8.123: type alias ara::core::Span::index_type

[SWS_CORE_01914] ⌈ The type alias ara::core::Span::difference_type is defined in Table 8.124. ⌋*(RS_AP_00130)*

Kind:	type alias
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.

Table 8.124: type alias ara::core::Span::difference_type

[SWS_CORE_01921] ⌈ The type alias ara::core::Span::size_type is defined in Table 8.125. ⌋*(RS_AP_00130)*

Kind:	type alias
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.

Table 8.125: type alias ara::core::Span::size_type

[SWS_CORE_01915] ⌈ The type alias ara::core::Span::pointer is defined in Table 8.126. ⌋*(RS_AP_00130)*

Kind:	type alias
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.

Table 8.126: type alias ara::core::Span::pointer

[SWS_CORE_01916] ⌈ The type alias ara::core::Span::reference is defined in Table 8.127. ⌋*(RS_AP_00130)*

Kind:	type alias
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.

Table 8.127: type alias ara::core::Span::reference

[SWS_CORE_01917] ⌈ The type alias ara::core::Span::iterator is defined in Table 8.128. ⌋*(RS_AP_00130)*

Kind:	type alias
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.

Table 8.128: type alias ara::core::Span::iterator

[SWS_CORE_01918] ⌈ The type alias ara::core::Span::const_iterator is defined in Table 8.129. ⌋*(RS_AP_00130)*

Kind:	type alias
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.

Table 8.129: type alias ara::core::Span::const_iterator

[SWS_CORE_01919] ⌈ The type alias ara::core::Span::reverse_iterator is defined in Table 8.130. ⌋*(RS_AP_00130)*

Kind:	type alias
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.

Table 8.130: type alias ara::core::Span::reverse_iterator

[SWS_CORE_01920] ⌈ The type alias ara::core::Span::const_reverse_iterator is defined in Table 8.131. ⌋*(RS_AP_00130)*

Kind:	type alias
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.

Table 8.131: type alias ara::core::Span::const_reverse_iterator

[SWS_CORE_01931] ⌈ The variable ara::core::Span::extent is defined in Table 8.132. ⌋*(RS_AP_00130)*

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

Table 8.132: variable ara::core::Span::extent

[SWS_CORE_01941] ⌈ The function ara::core::Span::Span is defined in Table 8.133. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Span::Span()
Kind:	function
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.

Table 8.133: function ara::core::Span::Span

[SWS_CORE_01942] ⌈ The function ara::core::Span::Span is defined in Table 8.134. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Span::Span(pointer ptr, index_type count)	
Kind:	function	
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.	

Table 8.134: function ara::core::Span::Span

[SWS_CORE_01943] ⌈ The function ara::core::Span::Span is defined in Table 8.135.
⌋*(RS_AP_00130)*

Symbol:	ara::core::Span::Span(pointer firstElem, pointer lastElem)	
Kind:	function	
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.	

Table 8.135: function ara::core::Span::Span

[SWS_CORE_01944] ⌈ The function ara::core::Span::Span is defined in Table 8.136.
⌋*(RS_AP_00130)*

Symbol:	ara::core::Span::Span(element_type(&arr)	
Kind:	function	
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, andstd::remove_pointer<decltype(ara::core::data(arr))>::type(*)[] is convertible to T(*)[].

Table 8.136: function ara::core::Span::Span

[SWS_CORE_01945] ⌈ The function ara::core::Span::Span is defined in Table 8.137.
⌋*(RS_AP_00130)*

Symbol:	ara::core::Span::Span(Array< value_type, N > &arr)	
Kind:	function	
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, andstd::remove_pointer<decltype(ara::core::data(arr))>::type(*)[] is convertible to T(*)[].	

Table 8.137: function ara::core::Span::Span

[SWS_CORE_01946] ⌈ The function ara::core::Span::Span is defined in Table 8.138.
⌋*(RS_AP_00130)*

Symbol:	ara::core::Span::Span(Array< value_type, N > const &arr)	
Kind:	function	
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, andstd::remove_pointer<decltype(ara::core::data(arr))>::type(*)[] is convertible to T(*)[].	

Table 8.138: function ara::core::Span::Span

[SWS_CORE_01947] ⌈ The function ara::core::Span::Span is defined in Table 8.139.
⌋*(RS_AP_00130)*

Symbol:	ara::core::Span::Span(Container &cont)	
Kind:	function	
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, andstd::remove_pointer<decltype(ara::core::data(cont))>::type(*)[] is convertible to T(*)[].	

Table 8.139: function ara::core::Span::Span

[SWS_CORE_01948] ⌈ The function ara::core::Span::Span is defined in Table 8.140.
⌋*(RS_AP_00130)*

Symbol:	ara::core::Span::Span(Container const &cont)	
Kind:	function	
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, andstd::remove_pointer<decltype(ara::core::data(cont))>::type(*)[] is convertible to T(*)[].	

Table 8.140: function ara::core::Span::Span

[SWS_CORE_01949] ⌈ The function ara::core::Span::Span is defined in Table 8.141.
⌋*(RS_AP_00130)*

Symbol:	ara::core::Span::Span(Span const &other)	
Kind:	function	
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.

Table 8.141: function ara::core::Span::Span

[SWS_CORE_01950] ⌈ The function ara::core::Span::Span is defined in Table 8.142. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Span::Span(Span< U, N > const &s)	
Kind:	function	
Scope:	class ara::core::Span	

Syntax:	`template <typename U, std::ptrdiff_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.	

Table 8.142: function ara::core::Span::Span

[SWS_CORE_01951] ⌈ The function ara::core::Span::~Span is defined in Table 8.143. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Span::~Span()
Kind:	function
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.

Table 8.143: function ara::core::Span::~Span

[SWS_CORE_01952] ⌈ The function ara::core::Span::operator= is defined in Table 8.144. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Span::operator=(Span const &other)	
Kind:	function	
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.	

Table 8.144: function ara::core::Span::operator=

[SWS_CORE_01961] ⌈ The function ara::core::Span::first is defined in Table 8.145. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::Span::first()	
Kind:	function	
Scope:	class ara::core::Span	
Syntax:	template <std::ptrdiff_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.	

Table 8.145: function ara::core::Span::first

[SWS_CORE_01962] ⌈ The function ara::core::Span::first is defined in Table 8.146. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::Span::first(index_type count)	
Kind:	function	
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.	

Table 8.146: function ara::core::Span::first

[SWS_CORE_01963] ⌈ The function ara::core::Span::last is defined in Table 8.147. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::Span::last()	
Kind:	function	
Scope:	class ara::core::Span	
Syntax:	`template <std::ptrdiff_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.	

Table 8.147: function ara::core::Span::last

[SWS_CORE_01964] ⌈ The function ara::core::Span::last is defined in Table 8.148. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::Span::last(index_type count)	
Kind:	function	
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.	

Table 8.148: function ara::core::Span::last

[SWS_CORE_01965] ⌈ The function ara::core::Span::subspan is defined in Table 8.149. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::Span::subspan()	
Kind:	function	
Scope:	class ara::core::Span	
Syntax:	`template <std::ptrdiff_t Offset, std::ptrdiff_t Count> 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:	auto	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)	

Table 8.149: function ara::core::Span::subspan

[SWS_CORE_01966] ⌈ The function ara::core::Span::subspan is defined in Table 8.150. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::Span::subspan(index_type offset, index_type count=dynamic_extent)	
Kind:	function	
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.	

Table 8.150: function ara::core::Span::subspan

[SWS_CORE_01967] ⌈ The function ara::core::Span::size is defined in Table 8.151. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::Span::size()	
Kind:	function	
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. the number of elements contained in this Span	

Table 8.151: function ara::core::Span::size

[SWS_CORE_01968] ⌈ The function ara::core::Span::size_bytes is defined in Table 8.152. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::Span::size_bytes()	
Kind:	function	
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. the number of bytes covered by this Span	

Table 8.152: function ara::core::Span::size_bytes

[SWS_CORE_01969] ⌈ The function ara::core::Span::empty is defined in Table 8.153. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::Span::empty()	
Kind:	function	
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. true if this Span contains 0 elements, false otherwise	

Table 8.153: function ara::core::Span::empty

[SWS_CORE_01970] ⌈ The function ara::core::Span::operator[] is defined in Table 8.154. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Span::operator[](index_type idx)	
Kind:	function	
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.	

Table 8.154: function ara::core::Span::operator[]

[SWS_CORE_01971] ⌈ The function ara::core::Span::data is defined in Table 8.155. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Span::data()	
Kind:	function	
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. the pointer	

Table 8.155: function ara::core::Span::data

[SWS_CORE_01972] ⌈ The function ara::core::Span::begin is defined in Table 8.156. ⌋*(RS_AP_00130)*

Symbol:	ara::core::Span::begin()	
Kind:	function	
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. the iterator	

Table 8.156: function ara::core::Span::begin

[SWS_CORE_01973] ⌈ The function ara::core::Span::end is defined in Table 8.157. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::Span::end()	
Kind:	function	
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. the iterator	

Table 8.157: function ara::core::Span::end

[SWS_CORE_01974] ⌈ The function ara::core::Span::cbegin is defined in Table 8.158. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::Span::cbegin()	
Kind:	function	
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. the const_iterator	

Table 8.158: function ara::core::Span::cbegin

[SWS_CORE_01975] ⌈ The function ara::core::Span::cend is defined in Table 8.159. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::Span::cend()	
Kind:	function	
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. the const_iterator	

Table 8.159: function ara::core::Span::cend

[SWS_CORE_01976] ⌈ The function ara::core::Span::rbegin is defined in Table 8.160. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::Span::rbegin()	
Kind:	function	
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. the reverse_iterator	

Table 8.160: function ara::core::Span::rbegin

[SWS_CORE_01977] ⌈ The function ara::core::Span::rend is defined in Table 8.161. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::Span::rend()	
Kind:	function	
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. the reverse_iterator	

Table 8.161: function ara::core::Span::rend

[SWS_CORE_01978] ⌈ The function ara::core::Span::crbegin is defined in Table 8.162. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::Span::crbegin()	
Kind:	function	
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. the const_reverse_iterator	

Table 8.162: function ara::core::Span::crbegin

[SWS_CORE_01979] ⌈ The function ara::core::Span::crend is defined in Table 8.163. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::Span::crend()	
Kind:	function	
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. the reverse_iterator	

Table 8.163: function ara::core::Span::crend

[SWS_CORE_01990] ⌈ The function ara::core::MakeSpan is defined in Table 8.164. ⌋ *(RS_AP_00130)*

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

Table 8.164: function ara::core::MakeSpan

[SWS_CORE_01991] ⌈ The function ara::core::MakeSpan is defined in Table 8.165. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::MakeSpan(T *firstElem, T *lastElem)	
Kind:	function	
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).	

Table 8.165: function ara::core::MakeSpan

[SWS_CORE_01992] ⌈ The function ara::core::MakeSpan is defined in Table 8.166. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::MakeSpan(T(&arr)	
Kind:	function	
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.	

Table 8.166: function ara::core::MakeSpan

[SWS_CORE_01993] ⌈ The function ara::core::MakeSpan is defined in Table 8.167. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::MakeSpan(Container &cont)	
Kind:	function	
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.	

Table 8.167: function ara::core::MakeSpan

[SWS_CORE_01994] ⌈ The function ara::core::MakeSpan is defined in Table 8.168. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::MakeSpan(Container const &cont)	
Kind:	function	
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.	

Table 8.168: function ara::core::MakeSpan

8.15 `InstanceSpecifier` data type

This section describes the `ara::core::InstanceSpecifier` type. `ara::core::InstanceSpecifier`s are used to identify service port prototype instances within the AUTOSAR meta-model and are therefore used in the `ara::com` API. A detailed description and background can be found in [9] chapter 8.4.4.

`ara::core::InstanceSpecifier` is basically a wrapper for a valid meta-model related string in the form: <shortname context1>/<shortname context2>/.../<shortname contextN>/<shortname of PortPrototype>. "Valid" is meant here according to the syntax: The string has to be a valid `shortname path` in terms of AUTOSAR meta model definitions, i.e. a row of valid AUTOSAR `shortname`s separated by "/" as path delimiter.

`ara::core::InstanceSpecifier` is designed to be either constructed from a string representation via a factory method `makeInstanceSpecifier`, which provides an exception-free solution by using a construction token pattern, or directly by using the `ctor` overload with a string parameter, which might throw an exception, if the string representation is invalid.

[SWS_CORE_08001] ⌈ The class ara::core::InstanceSpecifier is defined in Table 8.169. ⌋*(RS_Main_00320)*

Kind:	class
Base class:	None
Syntax:	`class InstanceSpecifier`
Header file:	#include "ara/core/instance_specifier.h"
Description:	class representing an AUTOSAR Instance Specifier, which is basically an AUTOSAR shortname-path wrapper.

Table 8.169: class ara::core::InstanceSpecifier

[SWS_CORE_08021] ⌈ The function ara::core::InstanceSpecifier::InstanceSpecifier is defined in Table 8.170. ⌋*(RS_Main_00320)*

Symbol:	ara::core::InstanceSpecifier::InstanceSpecifier(StringView metaModelIdentifier)	
Kind:	function	
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:	InstanceSpecifierException	in case the given metamodel_identifier isn't a valid meta-model identifier/short name path.
Header file:	#include "ara/core/instance_specifier.h"	
Description:	throwing ctor from meta-model string	

Table 8.170: function ara::core::InstanceSpecifier::InstanceSpecifier

[SWS_CORE_08022] ⌈ The function ara::core::InstanceSpecifier::InstanceSpecifier is defined in Table 8.171. ⌋*(RS_Main_00320)*

Symbol:	ara::core::InstanceSpecifier::InstanceSpecifier(ConstructionToken &&token)	
Kind:	function	
Scope:	class ara::core::InstanceSpecifier	
Syntax:	`explicit InstanceSpecifier (ConstructionToken &&token) noexcept;`	
Parameters (in):	token	construction token obtained via Preconstruct.
Exception Safety:	noexcept	
Header file:	#include "ara/core/instance_specifier.h"	
Description:	non-throwing ctor for construction token pattern	

Table 8.171: function ara::core::InstanceSpecifier::InstanceSpecifier

[SWS_CORE_08029] ⌈ The function ara::core::InstanceSpecifier::~InstanceSpecifier is defined in Table 8.172. ⌋*(RS_Main_00320)*

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

Table 8.172: function ara::core::InstanceSpecifier::~InstanceSpecifier

[SWS_CORE_08031] ⌈ The function ara::core::InstanceSpecifier::Preconstruct is defined in Table 8.173. ⌋*(RS_Main_00320)*

Symbol:	ara::core::InstanceSpecifier::Preconstruct(StringView metaModelIdentifier)	
Kind:	function	
Scope:	class ara::core::InstanceSpecifier	
Syntax:	`static Result<ConstructionToken> Preconstruct (StringView metaModelIdentifier) noexcept;`	
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.
Return value:	Result< ConstructionToken >	a construction token from which an instance specifier can be constructed or an error
Exception Safety:	noexcept	
Header file:	#include "ara/core/instance_specifier.h"	
Description:	preconstruct method for InstanceSpecifier.	

Table 8.173: function ara::core::InstanceSpecifier::Preconstruct

[SWS_CORE_08042] ⌈ The function ara::core::InstanceSpecifier::operator== is defined in Table 8.174. ⌋*(RS_Main_00320)*

Symbol:	ara::core::InstanceSpecifier::operator==(InstanceSpecifier const &other)	
Kind:	function	
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.	

Table 8.174: function ara::core::InstanceSpecifier::operator==

[SWS_CORE_08043] ⌈ The function ara::core::InstanceSpecifier::operator== is defined in Table 8.175. ⌋*(RS_Main_00320)*

Symbol:	ara::core::InstanceSpecifier::operator==(StringView other)	
Kind:	function	
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.	

Table 8.175: function ara::core::InstanceSpecifier::operator==

[SWS_CORE_08044] ⌈ The function ara::core::InstanceSpecifier::operator!= is defined in Table 8.176. ⌋*(RS_Main_00320)*

Symbol:	ara::core::InstanceSpecifier::operator!=(InstanceSpecifier const &other)	
Kind:	function	
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.	

Table 8.176: function ara::core::InstanceSpecifier::operator!=

[SWS_CORE_08045] ⌈ The function ara::core::InstanceSpecifier::operator!= is defined in Table 8.177. ⌋*(RS_Main_00320)*

Symbol:	ara::core::InstanceSpecifier::operator!=(StringView other)	
Kind:	function	
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.	

Table 8.177: function ara::core::InstanceSpecifier::operator!=

[SWS_CORE_08046] ⌈ The function ara::core::InstanceSpecifier::operator< is defined in Table 8.178. ⌋*(RS_Main_00320)*

Symbol:	ara::core::InstanceSpecifier::operator<(InstanceSpecifier const &other)	
Kind:	function	
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).	

Table 8.178: function ara::core::InstanceSpecifier::operator<

[SWS_CORE_08041] ⌈ The function ara::core::InstanceSpecifier::ToString is defined in Table 8.179. ⌋ *(RS_Main_00320)*

Symbol:	ara::core::InstanceSpecifier::ToString()	
Kind:	function	
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 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.	

Table 8.179: function ara::core::InstanceSpecifier::ToString

[SWS_CORE_08090] ⌈ The function ara::core::MakeInstanceSpecifier is defined in Table 8.180. ⌋ *(RS_Main_00320)*

Symbol:	ara::core::MakeInstanceSpecifier(StringView metaModelIdentifier)	
Kind:	function	
Scope:	namespace ara::core	
Syntax:	`Result<InstanceSpecifier> MakeInstanceSpecifier (StringView metaModelIdentifier) noexcept;`	
Parameters (in):	metaModelIdentifier	stringified form of InstanceSpecifier. Lifetime of underlying string has to exceed the lifetime of the constructed InstanceSpecifier.
Return value:	Result< InstanceSpecifier >	a Result containing valid InstanceSpecifier or an ErrorCode
Exception Safety:	noexcept	
Header file:	#include "ara/core/instance_specifier.h"	

▽

△

Description:	A non-throwing helper method to create InstanceSpecifier from a string representation.

Table 8.180: function ara::core::MakeInstanceSpecifier

8.16 Generic helpers

8.16.1 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].

8.16.1.1 `in_place_t` tag

[SWS_CORE_04011] ⌈ The struct ara::core::in_place_t is defined in Table 8.181. ⌋ *(RS_AP_00130)*

Kind:	struct
Protection:	Public
Public fields:	None
Protected fields:	None
Private fields:	None
Location:	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.

Table 8.181: struct ara::core::in_place_t

[SWS_CORE_04012] ⌈ The function ara::core::in_place_t::in_place_t is defined in Table 8.182. ⌋ *(RS_AP_00130)*

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

Table 8.182: function ara::core::in_place_t::in_place_t

[SWS_CORE_04013] ⌈ The variable ara::core::in_place is defined in Table 8.183. ⌋ *(RS_AP_00130)*

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

Table 8.183: variable ara::core::in_place

8.16.1.2 `in_place_type_t` tag

[SWS_CORE_04021] ⌈ The struct ara::core::in_place_type_t is defined in Table 8.184. ⌋ *(RS_AP_00130)*

Kind:	struct
Protection:	Public
Public fields:	None
Protected fields:	None
Private fields:	None
Location:	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.

Table 8.184: struct ara::core::in_place_type_t

[SWS_CORE_04022] ⌈ The function ara::core::in_place_type_t::in_place_type_t is defined in Table 8.185. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::in_place_type_t::in_place_type_t()
Kind:	function
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.

Table 8.185: function ara::core::in_place_type_t::in_place_type_t

8.16.1.3 `in_place_index_t` tag

[SWS_CORE_04031] ⌈ The struct ara::core::in_place_index_t is defined in Table 8.186. ⌋*(RS_AP_00130)*

Kind:	struct
Protection:	Public
Public fields:	None
Protected fields:	None
Private fields:	None
Location:	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.

Table 8.186: struct ara::core::in_place_index_t

[SWS_CORE_04032] ⌈ The function ara::core::in_place_index_t::in_place_index_t is defined in Table 8.187. ⌋*(RS_AP_00130)*

Symbol:	ara::core::in_place_index_t::in_place_index_t()
Kind:	function
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.

Table 8.187: function ara::core::in_place_index_t::in_place_index_t

8.16.2 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] ⌈ The function ara::core::data is defined in Table 8.188. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::data(Container &c)	
Kind:	function	
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:	auto	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.	

Table 8.188: function ara::core::data

[SWS_CORE_04111] ⌈ The function ara::core::data is defined in Table 8.189. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::data(Container const &c)	
Kind:	function	
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:	auto	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.	

Table 8.189: function ara::core::data

[SWS_CORE_04112] ⌈ The function ara::core::data is defined in Table 8.190. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::data(T(&array)	
Kind:	function	
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.	

Table 8.190: function ara::core::data

[SWS_CORE_04113] ⌈ The function ara::core::data is defined in Table 8.191. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::data(std::initializer_list< E > il)	
Kind:	function	
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.	

Table 8.191: function ara::core::data

[SWS_CORE_04120] ⌈ The function ara::core::size is defined in Table 8.192. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::size(Container const &c)	
Kind:	function	
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:	auto	the size of the container
Header file:	#include "ara/core/utility.h"	
Description:	Return the size of a container.	

Table 8.192: function ara::core::size

[SWS_CORE_04121] ⌈ The function ara::core::size is defined in Table 8.193. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::size(T const (&array)	
Kind:	function	
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.

Table 8.193: function ara::core::size

[SWS_CORE_04130] ⌈ The function ara::core::empty is defined in Table 8.194. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::empty(Container const &c)	
Kind:	function	
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:	auto	true if the container is empty, false otherwise
Header file:	#include "ara/core/utility.h"	
Description:	Return whether the given container is empty.	

Table 8.194: function ara::core::empty

[SWS_CORE_04131] ⌈ The function ara::core::empty is defined in Table 8.195. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::empty(T const (&array)	
Kind:	function	
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.	

Table 8.195: function ara::core::empty

[SWS_CORE_04132] ⌈ The function ara::core::empty is defined in Table 8.196. ⌋ *(RS_AP_00130)*

Symbol:	ara::core::empty(std::initializer_list< E > il)	
Kind:	function	
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.	

Table 8.196: function ara::core::empty

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] ⌈ The function ara::core::Initialize is defined in Table 8.197. ⌋ *(RS_Main_00011)*

Symbol:	ara::core::Initialize()	
Kind:	function	
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. ▽	

▽

△

	△ 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.

Table 8.197: function ara::core::Initialize

[SWS_CORE_10002] ⌈ The function ara::core::Deinitialize is defined in Table 8.198. ⌋
(RS_Main_00011)

Symbol:	ara::core::Deinitialize()	
Kind:	function	
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. 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.	

Table 8.198: function ara::core::Deinitialize