# Contents

1. Introduction 8
2. Acronyms and Abbreviations 9
3. Related documentation 10
   1. Input documents & related standards and norms 10
4. Constraints and assumptions 11
   1. Limitations 11
   2. Applicability to car domains 11
5. Dependencies to other modules 12
6. Requirements Tracing 13
7. [Requirements Specification](#_bookmark0) 25
   1. [General requirements for all Functional Clusters](#_bookmark1) 25
      1. [Initialize/Deinitialize](#_bookmark2) 27
   2. [Functional Specification](#_bookmark3) 28
      1. [Error handling](#_bookmark4) 28
         1. [Types of unsuccessful operations](#_bookmark5) 28
         2. [Traditional error handling in C and C++](#_bookmark9) 28
         3. [Handling of unsuccessful operations in the Adaptive](#_bookmark10) [Platform](#_bookmark10) 29
         4. [Facilities for Error Handling](#_bookmark12) 30
            1. [ErrorCode](#_bookmark13) 30
            2. [ErrorDomain](#_bookmark14) 32
            3. [Result](#_bookmark16) 33
            4. [Future and Promise](#_bookmark17) 33
         5. [Duality of ErrorCode and exceptions](#_bookmark18) 34
         6. [Exception hierarchy](#_bookmark19) 34
         7. [Creating new error domains](#_bookmark20) 35
            1. [Error condition enumeration](#_bookmark21) 35
            2. [Exception base class](#_bookmark23) 36
            3. [ErrorDomain subclass](#_bookmark25) 36
            4. [Non-member ErrorDomain subclass accessor](#_bookmark27) [function](#_bookmark27) 38
            5. [Non-member MakeErrorCode overload](#_bookmark28) 38
            6. [C++ pseudo code example](#_bookmark29) 39
         8. [AUTOSAR error domains](#_bookmark30) 39
      2. [Async signal safety](#_bookmark32) 40
      3. [Explicit Operation Abortion](#_bookmark33) 40
         1. [AbortHandler](#_bookmark35) 41
         2. [SIGABRT handler](#_bookmark36) 42
      4. [Advanced data types](#_bookmark37) 43
         1. [AUTOSAR types](#_bookmark38) 43
            1. [InstanceSpecifier](#_bookmark39) 43
            2. [ScaleLinearAndTexttable](#_bookmark41) 44
         2. [Types derived from the base C++ standard](#_bookmark42) 44
            1. [Array](#_bookmark43) 44
            2. [Vector](#_bookmark44) 45
            3. [Map](#_bookmark45) 45
            4. [String and BasicString](#_bookmark46) 45
            5. [SteadyClock](#_bookmark47) 45

[Definitions of terms](#_bookmark48) 45

[Clocks in the Adaptive Platform](#_bookmark53) 46

* + - 1. [Types derived from newer C++ standards](#_bookmark54) 47
         1. [Optional](#_bookmark55) 47
         2. [Variant](#_bookmark56) 48
         3. [StringView](#_bookmark57) 48
         4. [Span](#_bookmark58) 48
         5. [ara::core::Byte](#_bookmark59) 48
    1. [Initialization and Shutdown](#_bookmark61) 50

1. API specification 52
   1. C++ language binding 52
      1. ErrorDomain data type 52
      2. ErrorCode data type 56
         1. ErrorCode non-member operators 59
      3. Exception data type 60
      4. Result data type 62
         1. Result<void, E> template specialization 75
         2. Non-member function overloads 85
      5. Core Error Domain 89
         1. CORE error codes 89
         2. CoreException type 89
         3. CoreErrorDomain type 90
         4. GetCoreErrorDomain accessor function 92
         5. MakeErrorCode overload for CoreErrorDomain . 92
      6. Future and Promise data types 93
         1. future\_errc enumeration 93
         2. FutureException type 94
         3. FutureErrorDomain type 94
         4. FutureErrorDomain accessor function 96
         5. MakeErrorCode overload for FutureErrorDomain 97
         6. future\_status enumeration 97
         7. Future data type 98
            1. Future<void, E> template specialization . 103
         8. Promise data type 108
            1. Promise<void, E> template specialization . 112
      7. Array data type 116
         1. Class Array 116
         2. Non-member functions 126
         3. Tuple interface 129
      8. Vector data type 131
      9. Map data type 134
      10. Optional data type 135
      11. Variant data type 136
      12. StringView data type 137
      13. String data types 137
      14. Span data type 146
      15. SteadyClock data type 165
      16. InstanceSpecifier data type 167
      17. ScaleLinearAndTexttable data type 172
      18. Generic helpers 183
          1. ara::core::Byte 183
          2. In-place disambiguation tags 184
             1. in\_place\_t tag 184
             2. in\_place\_type\_t tag 185
             3. in\_place\_index\_t tag 186
          3. Non-member container access 186
      19. Initialization and Shutdown 190
      20. Abnormal process termination 190
2. Mentioned Manifest Elements 193
3. Interfaces to other Functional Clusters (informative) 195
   1. Overview 195
   2. Interface Tables 195
      1. Functional Cluster initialization 195
4. History of Specification Items 196
   1. Specification Item History of this document compared to AUTOSAR

R20-11. 196

* + 1. Added Traceables in R21-11 196
    2. Changed Traceables in R21-11 197
    3. Deleted Traceables in R21-11 208
  1. Specification Item History of this document compared to AUTOSAR

R19-11. 208

* + 1. Added Traceables in R20-11 208
    2. Changed Traceables in R20-11 211
    3. Deleted Traceables in R20-11 219
  1. Specification Item History of this document compared to AUTOSAR

R19-03. 219

* + 1. Added Traceables in R19-11 219
    2. Changed Traceables in R19-11 226
    3. Deleted Traceables in R19-11 226

# Requirements Specification

## General requirements for all Functional Clusters

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

**[SWS\_CORE\_90001] Include folder structure** *[*All #include directives in header files that refer to ARA libraries shall be written in the form

#include "ara/fc/header.h"

with “ara” as the first path element, “fc” being the remaining directory path of the imple- mentation’s *installed* header file, starting with the Functional Cluster short name, and “header.h” being the filename of the header file.*♩(RS\_AP\_00116, RS\_AP\_00111)*

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

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

#include "ara/exec/execution\_client.h"

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

**[SWS\_CORE\_90002] Prevent multiple inclusion of header file** *[*All public header files shall prevent multiple inclusion by using #include guards that are likely to be system-wide unique.*♩(RS\_AP\_00111)*

While uniqueness can generally not be guaranteed, the likelihood of collisions can be decreased with a naming scheme that is regular and results in long symbol names.

The following #include guard naming scheme should be used by implementations for all header files that cover symbols within the ara namespace or a sub-namespace therein:

ARA\_<PATH>\_H\_

where <PATH> is the relative path name of the header file within the location of the implementation’s *installed* header files, starting with the Functional Cluster name (and omitting the file extension), and with all components of <PATH> separated by under- score ("\_") characters and containing only upper-case characters of the ASCII charac- ter set.

Example: The header file included with #include "ara/log/logger.h" should use the #include guard symbol ARA\_LOG\_LOGGER\_H\_.

**[SWS\_CORE\_90003]**{DRAFT} *[*C/C++ preprocessor symbols that start with ARA are reserved for use by AUTOSAR.*♩(RS\_AP\_00111)*

The Adaptive Platform generally avoids the use of C/C++ preprocessor macros. How- ever, in case macros are introduced at some later point in time, any such macro will start with the prefix ARA. Platform vendors should thus not define any symbols (both macros and C/C++ ones) with this prefix, lest they conflict with such future additions to the standard.

**[SWS\_CORE\_90004]**{DRAFT} **Implementation-defined declaration classifiers** *[*All APIs shall be implemented with the exact same declaration classifiers that are speci- fied, except for inline and friend, which may be added as necessary.*♩(RS\_AP\_- 00111)*

*Note: The order of declarations may be freely chosen.*

[4, The C++14 standard] defines in chapter 7.1 [dcl.spec] the specifiers that can be used in a declaration; these include, for instance, static, virtual, constexpr, inline and friend. An implementation that uses a different set of specifiers in its declaration of a specified API may be incompatible to the standard, or may allow non- standardized usage of that API, leading to portability concerns.

**[SWS\_CORE\_90005]**{DRAFT} **Custom declarations and definitions** *[*Implementation shall not add public declarations or definitions that are not specified in an SWS to the namespace ara or any of its direct sub-namespaces.*♩(RS\_AP\_00111)*

The Adaptive Platform is designed for source code portability. Wherefore any con- formant implementation of the Adaptive Platform allows a successful compilation and linking of an Adaptive Application that uses ARA only as specified in the standard. No changes to the source code, and no conditional compilation constructs will be neces- sary for this if the application only uses constructs from the designated minimum C++ language version. The implementation may provide proprietary, non-ARA interfaces, as long as they are not contradicting the AP standard.

**[SWS\_CORE\_90006]**{DRAFT} *[*If a constructor in the ara framework is called with wrong or invalid ara::core::InstanceSpecifier, the Functional Clus- ter implementation shall treat this as a [Violation](#_bookmark7) with a standardized log message "Invalid InstanceSpecifer >passed InstanceSpecifier< in ctor >ctor.shortname<".*♩(RS\_AP\_00111)*

The rationale to treat this as a [Violation](#_bookmark7) is that this is seen as an integration error which anyway cannot be handled by the caller of the API. Aborting execution is in line with the strategy to fail early.

Any other error check within the constructors is defined within the respective SWS.

### Initialize/Deinitialize

ara::core::Initialize allows a central initialization of all included shared li- braries of the ARA framework. This could include static initializers or the setup of daemon links (details are up to the platform vendor).

The general advice for application developers is to call ara::core::Initialize

right at the entry point of the application.

**[SWS\_CORE\_90020]**{DRAFT} *[*If functionality is called that depends on prior initial- ization via ara::core::Initialize and ara::core::Initialize has not been called, the Functional Cluster implementation shall treat this as a [Violation](#_bookmark7).*♩(RS\_- AP\_00111)*

The rationale to treat this as a [Violation](#_bookmark7) is that it cannot be handled by the caller of the API at the point in time where the error is detected. Aborting execution is the only way to signal this kind of systematic error and prevent later failures.

## Functional Specification

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

### Error handling

### Types of unsuccessful operations

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

**[SWS\_CORE\_00020]**{DRAFT} **Semantics of an Error** *[*An Error is the inability of an assumed-bug-free API function to fulfill its specified purpose; it is often a conse- quence of invalid and/or unexpected (i.e. possibly valid, but received in unexpected circumstances) input data. An [Error](#_bookmark6) is recoverable.*♩(RS\_AP\_00142)*

**[SWS\_CORE\_00021]**{DRAFT} **Semantics of a Violation** *[*A Violation is the con- sequence of failed pre- or post-conditions of internal state of the application framework. They are the Adaptive Platform’s analog to a failed assertion. A [Violation](#_bookmark7) is non- recoverable.*♩(RS\_AP\_00142)*

**[SWS\_CORE\_00022]**{DRAFT} **Semantics of a Corruption** *[*A Corruption is the consequence of the corruption of a system resource, e.g. stack or heap overflow, or a hardware memory flaw (including even, for instance, a detected bit flip). A [Corrup-](#_bookmark8) [tion](#_bookmark8) is non-recoverable.*♩(RS\_AP\_00142)*

**[SWS\_CORE\_00023]**{DRAFT} **Semantics of a Failed Default Allocation** *[*A Failed Default Allocation is the inability of the framework’s default memory allocation mechanism to satisfy an allocation request. A Failed Default Allocation is non-recoverable.*♩(RS\_AP\_00142)*

It is expected that a [Violation](#_bookmark7) or [Corruption](#_bookmark8) might occur during development of the framework, when new features are just coming together, but will not be experienced by a user (i.e. an application developer), unless there is something seriously wrong in the system’s environment (e.g. faulty hardware: [Corruption](#_bookmark8)), or basic assumptions about resource requirements are violated ([Violation](#_bookmark7)), or possibly the user runs the framework in a configuration that is not supported by its vendor ([Violation](#_bookmark7)).

### 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 inher- itance 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 gen- erally relies on them for error propagation.

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

* with exceptions, it can be difficult to reason about a program’s control flow
* exceptions have much higher runtime cost than error codes (either in general, or only in the exception-thrown case)

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

The second one is even more critical in the context of developing safety-critical soft- ware. 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++, im- plies the use of dynamic memory allocation, which generally has non-predictable or even unbounded execution time. This makes exceptions currently unsuitable for devel- opment of certain safety-critical software in the automotive industry.

### Handling of unsuccessful operations in the Adaptive Platform

The types of unsuccessful operations defined in section [7.2.1.1](#_bookmark5) (“[Types of unsuccessful](#_bookmark5) [operations](#_bookmark5)”) are to be treated in different ways.

**[SWS\_CORE\_00002] Handling of Errors** *[*An [Error](#_bookmark6) shall be returned from the func- tion as an instance of ara::core::Result or ara::core::Future.*♩(RS\_AP\_- 00142, RS\_AP\_00139, RS\_AP\_00128)*

**[SWS\_CORE\_00003] Handling of Violations** *[*If a [Violation](#_bookmark7) is detected, then the operation shall be terminated by either:

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

*♩(RS\_AP\_00142)*

**[SWS\_CORE\_00004] Handling of Corruptions** *[*If a [Corruption](#_bookmark8) is detected, it shall result in unsuccessful process termination, in an implementation-defined way.*♩(RS\_- AP\_00142)*

*Note: It can either be abnormal or normal unsuccessful termination, depending on the implementation’s ability to detect the* [*Corruption*](#_bookmark8) *and to react to it by cleaning up resources.*

**[SWS\_CORE\_00005] Handling of failed default allocations** *[*A "failed default allo- cation" shall be treated the same as a [Violation](#_bookmark7).*♩(RS\_AP\_00142)*

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

### Facilities for Error Handling

For handling [Errors](#_bookmark6), there are a number of data types defined that help in dealing with them. These are described in the following subsections.

### 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 do- main.

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 addi- tional piece of data about the error.

**[SWS\_CORE\_10302]**{DRAFT} **Semantics of ErrorCode** *[*The type ara::core::- ErrorCode provides a class interface for storing an error condition. It shall contain these properties:

error code value: an integral representation of a low-level error code

error domain: reference to the context for which the *error code value* is applicable

support data value: an optional vendor-specific additional piece of data about the error

*♩(RS\_AP\_00119)*

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

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

However, comparison of two ara::core::ErrorCode instances only considers the error code value and the error domain reference; the support data value member is not considered for checking equality. This is due to the way ara::core:-

:ErrorCode instances are usually compared against a known set of errors for which to check:

1 ErrorCode ec = ...

2 if (ec == MyEnum::some\_error)

3 // ...

4 else if (ec == AnotherEnum::another\_error)

5 // ...

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 au- tomatically created instances naturally do not contain any meaningful support data value.

**[SWS\_CORE\_10301]**{DRAFT} **Comparison of ara::core::ErrorCode instances** *[*Any comparison of two ara::core::ErrorCode instances shall consider only the following members:

error code value

error domain

*♩(RS\_AP\_00119)*

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

**[SWS\_CORE\_10300] ErrorCode type properties** *[*Class ara::core::ErrorCode shall be a *literal type*, as defined in section 3.9-10 [basic.types] of [4, the C++14 stan- dard].*♩(RS\_AP\_00130)*

### 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::Error- Domain subclass. This makes the ErrorDomain subclass the root of all data about errors.

**[SWS\_CORE\_10303]**{DRAFT} **Semantics of ErrorDomain** *[*The type ara::core:-

:ErrorDomain defines a context for a set of error conditions.*♩(RS\_AP\_00119)*

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 (ei- ther an AUTOSAR defined or an user defined one) an according Id shall be randomly generated, which represents this error domain. The uniqueness and standardization of such an Id per error domain is mandatory, since the exchange of information on oc- cured errors between callee and caller (potentially located at different ECUs) is based on this Id.

**[SWS\_CORE\_10401]**{DRAFT} **Identity of ErrorDomains** *[*Two instances of ara:-

:core::ErrorDomain shall compare equal if and only if their unique identifiers are the same.*♩(RS\_AP\_00119)*

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 in- stances of these subclasses can be created by calling their constructors, the recom- mended way to gain access to these subclasses is to call their non-member accessor functions. For instance, the error domain class ara::core::FutureErrorDomain is referenced by calling ara::core::GetFutureErrorDomain; within any process space, this will always return a reference to the same global instance of this class.

For error domains that are modeled in ARXML (as ApApplicationErrorDomain), the C++ language binding will create a C++ class for each such ApApplication- ErrorDomain. 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 ara::core::CoreErrorDo- main (containing the set of errors returned by non-Future/Promise facilities from the

ara::core Functional Cluster) and ara::core::FutureErrorDomain (containing errors equivalent to those defined by std::future\_errc).

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

As ara::core::ErrorDomain subclasses are expected to be implicitly referred to from within constant (i.e. compile-time) expressions (typically involving ara::core:-

:ErrorCode), they are expected to be *literal types*.

**[SWS\_CORE\_10400] ErrorDomain type properties** *[*Class ara::core::Error- Domain and all its subclasses shall be *literal types*, as defined in section 3.9-10 [ba- sic.types] of [4, the C++14 standard].*♩(RS\_AP\_00130)*

### 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 di- rect mapping between ara::core::ErrorCode and a domain-specific exception type, ara::core::Result allows to “transform” its embedded ara::core::Er- rorCode into the appropriate exception type, by calling ara::core::Result::- ValueOrThrow.

**[SWS\_CORE\_10600]**{DRAFT} **Semantics of ara::core::Result** *[*The type ara::- core::Result shall provide a means to handle both return values and errors from synchronous function calls in an exception-less way, by providing an encapsulated return type which may be either:

a value *V*, where *V* may be any C++ type; or

an error *E*, where *E* may be any C++ type; default is ara::core::ErrorCode.

*♩(RS\_AP\_00119, RS\_AP\_00142, RS\_AP\_00128)*

### 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 interoper- ate with ara::core::Result. Similar to ara::core::Result described in section [7.2.1.4.3](#_bookmark16), the class ara::core::Future either contains a value, or an error (the Fu-

ture first has to be in “ready” state, though). Class ara::core::Promise has been adapted in two aspects: std::promise::set\_exception has been removed, and ara::core::Promise::SetError has been introduced in its stead. For ara::- core::Future, there is a new member function ara::core::Future::GetRe- sult that is similar to ara::core::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 er- ror handling as ara::core::Result, in that it either works exception-based (with ara::core::Future::get), or exception-free (with ara::core::Future::Ge- tResult).

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.

**[SWS\_CORE\_10800]**{DRAFT} **Semantics of ara::core::Future and ara::core::Promise** *[*The types ara::core::Future and ara::core::Promise shall provide a means to handle both return values and errors from asynchronous function calls in an exception-less way. Together, they provide a means to store a value type *T* or an error type *E* which may be asynchronously retrieved in a thread-safe manner at a later point in time.*♩(RS\_AP\_00138, RS\_AP\_00128)*

### Duality of ErrorCode and exceptions

By using the classes listed above, all APIs of the Adaptive Platform can be used with ei- ther an exception-based or an exception-less error handling workflow. However, no API function will ever treat an [Error](#_bookmark6) by throwing an exception directly; it will always return an error code in the form of a ara::core::Result or ara::core::Future return value instead. It is then possible for the caller to “transform” the [Error](#_bookmark6) into an excep- tion, typically via the member function ara::core::Result::ValueOrThrow.

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

### Exception hierarchy

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

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

For error domains that are modeled in ARXML, the C++ language binding will gener- ate an exception class in addition to the ErrorDomain subclass (which is described in section [7.2.1.4.2](#_bookmark14)). 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.

### Creating new error domains

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

An error domain consists of:

* an error condition enumeration
* an exception base class
* an ara::core::ErrorDomain subclass
* a non-member ErrorDomain subclass accessor function
* a non-member MakeErrorCode function overload

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

**[SWS\_CORE\_10999] Custom error domain scope** *[*The ara::core::Error- Domain subclass and the corresponding enumeration, exception base class, non- member accessor function, and the MakeErrorCode overload shall be defined in the same namespace as the software module for which they are being specified.*♩(RS\_- AP\_00130)*

*Note: This is to help making sure that the C++ ADL mechanism works as expected by other parts of this standard.*

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

Throughout this section, the character sequence <SN> is a placeholder for the *short- name* of the ApApplicationErrorDomain.

### Error condition enumeration

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

**[SWS\_CORE\_10900] Error condition enumeration type** *[*Each error domain shall define an error condition enum class with the base type ara::core::ErrorDo- main::CodeType that holds all error conditions of that error domain.*♩(RS\_AP\_00130)*

**[SWS\_CORE\_10901] Error condition enumeration naming** *[*Error domain error con- dition enumerations shall follow the naming scheme <SN>Errc, where <SN> is the shortname of the ApApplicationErrorDomain.*♩(RS\_AP\_00130)*

**[SWS\_CORE\_10902] Error condition enumeration contents** *[*Error domain error condition enumerations shall not contain any values that indicate success.*♩(RS\_AP\_- 00130)*

**[SWS\_CORE\_10903] Error condition enumeration numbers** *[*Error domain error condition enumerations shall keep the number 0 unassigned.*♩(RS\_AP\_00130)*

### Exception base class

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

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

**[SWS\_CORE\_10910] ErrorDomain exception base type** *[*Each error domain shall define an exception base type that is a subclass of ara::core::Exception.*♩(RS\_- AP\_00130)*

**[SWS\_CORE\_10911] ErrorDomain exception base type naming** *[*All error do- main exception base types specified by [[SWS\_CORE\_10910](#_bookmark24)] shall follow the naming scheme <SN>Exception, where <SN> is the shortname of the ApApplication- ErrorDomain.*♩(RS\_AP\_00130)*

**[SWS\_CORE\_10912]**{DRAFT} **ErrorDomain exception type hierarchy** *[*All addi- tional exception types defined by a software module shall have the exception base type specified by [[SWS\_CORE\_10910](#_bookmark24)] as a base class.*♩(RS\_AP\_00130)*

### ErrorDomain subclass

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

**[SWS\_CORE\_10930] ErrorDomain subclass type** *[*Each error domain shall define a class type that derives publicly from ara::core::ErrorDomain.*♩(RS\_AP\_00130)*

**[SWS\_CORE\_10931] ErrorDomain subclass naming** *[*All subclasses of ara::- core::ErrorDomain shall follow the naming scheme <SN>ErrorDomain, where

<SN> is the shortname of the ApApplicationErrorDomain.*♩(RS\_AP\_00130)*

**[SWS\_CORE\_10932] ErrorDomain subclass non-extensibility** *[*All subclasses of

ara::core::ErrorDomain shall be final.*♩(RS\_AP\_00130, RS\_AP\_00140)*

**[SWS\_CORE\_10933] ErrorDomain subclass Errc symbol** *[*All subclasses of ara:-

:core::ErrorDomain shall contain in their scope a type alias called Errc that refers to the error condition enumeration defined by [[SWS\_CORE\_10900](#_bookmark22)].*♩(RS\_AP\_00130)*

**[SWS\_CORE\_10934] ErrorDomain subclass Exception symbol** *[*All subclasses of ara::core::ErrorDomain shall contain in their scope a type alias called Excep- tion that refers to the exception base type defined by [[SWS\_CORE\_10910](#_bookmark24)].*♩(RS\_- AP\_00130)*

All ErrorDomain subclasses are usable from within constant expressions, see [[SWS\_CORE\_10400](#_bookmark15)]. In particular, this includes that ErrorDomain subclasses can be defined as constexpr global variables.

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

**[SWS\_CORE\_10950] ErrorDomain subclass member function property** *[*With the exception of ara::core::ErrorDomain::ThrowAsException, all public member functions of all ErrorDomain subclasses shall be noexcept.*♩(RS\_AP\_00130)*

The virtual member function ara::core::ErrorDomain::Name returns the short- name of the ApApplicationErrorDomain, mostly for logging purposes.

**[SWS\_CORE\_10951] ErrorDomain subclass shortname retrieval** *[*The return value of an error domain’s ara::core::ErrorDomain::Name member function shall be equal to the shortname of the ApApplicationErrorDomain.*♩(RS\_AP\_00130)*

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

**[SWS\_CORE\_10952] ErrorDomain subclass unique identifier retrieval** *[*The return value of an error domain’s ara::core::ErrorDomain::Id member function shall be a unique identifier that follows the rules defined by [[SWS\_CORE\_00010](#_bookmark31)].*♩(RS\_- AP\_00130)*

An ErrorDomain can “transform” an ErrorCode into an exception.

**[SWS\_CORE\_10953] Throwing ErrorCodes as exceptions** *[*The type of an excep- tion thrown by the ErrorDomain subclass’s implementation of ara::core::Error- Domain::ThrowAsException shall derive from that ErrorDomain subclass’s Ex- ception type alias defined by [[SWS\_CORE\_10934](#_bookmark26)].*♩(RS\_AP\_00130)*

### Non-member ErrorDomain subclass accessor function

A non-member accessor function for the new error domain class is to be defined. For an error domain class MyErrorDomain, the accessor function is named GetMyEr- rorDomain. This accessor function returns a reference to a single global instance of that class. This accessor function shall be fully constexpr-capable; this in turn implies that the ErrorDomain subclass also shall be constexpr-constructible (see [[SWS\_CORE\_10400](#_bookmark15)]).

**[SWS\_CORE\_10980] ErrorDomain subclass accessor function** *[*For all subclasses of ara::core::ErrorDomain, there shall be a non-member constexpr function that returns a reference-to-const to a singleton instance of it.*♩(RS\_AP\_00130)*

### [SWS\_CORE\_10981] ErrorDomain subclass accessor function naming *[*All ara:-

:core::ErrorDomain subclass accessor functions shall follow the naming scheme Get<SN>ErrorDomain, where <SN> is the shortname of the ApApplication- ErrorDomain.*♩(RS\_AP\_00130)*

**[SWS\_CORE\_10982] ErrorDomain subclass accessor function** *[*All ara::core:-

:ErrorDomain subclass accessor functions shall have a return type of const Er- rorDomain&.*♩(RS\_AP\_00130)*

### Non-member MakeErrorCode overload

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

**[SWS\_CORE\_10990] MakeErrorCode overload for new error domains** *[*For all sub- classes of ara::core::ErrorDomain, there shall be a constexpr overload of the non-member function MakeErrorCode that creates an ara::core::ErrorCode in- stance for a given error condition value within the ara::core::ErrorDomain sub- class’s error condition range.*♩(RS\_AP\_00130)*

**[SWS\_CORE\_10991] MakeErrorCode overload signature** *[*All overloads of the non- member function MakeErrorCode shall have the following signature:

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

where <SN> is the shortname of the ApApplicationErrorDomain.*♩(RS\_AP\_- 00130)*

### C++ pseudo code example

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

1 namespace my

2 {

3

4 enum class <SN>Errc : ara::core::ErrorDomain::CodeType

5 {

6 // ...

7 };

8

9 class <SN>Exception : public ara::core::Exception

10 {

11 public:

12 <SN>Exception(ara::core::ErrorCode err) noexcept;

13 };

14

15 class <SN>ErrorDomain final : public ara::core::ErrorDomain

16 {

17 public:

18 using Errc = <SN>Errc;

19 using Exception = <SN>Exception;

20

21 constexpr <SN>ErrorDomain() noexcept;

22

23 const char\* Name() const noexcept override;

24 const char\* Message(ara::core::ErrorDomain::CodeType errorCode) const noexcept override;

25 void ThrowAsException(const ara::core::ErrorCode& errorCode) const noexcept(false) override;

26 };

27

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

29

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

31

32 } // namespace my

### AUTOSAR error domains

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

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

**[SWS\_CORE\_00010] Error domain identifier** *[*All error domains shall have a system- wide unique identifier that is represented as a 64-bit unsigned integer value.*♩(RS\_AP\_- 00130)*

**[SWS\_CORE\_00011] AUTOSAR error domain range** *[*Error domain identifiers where bit #63 is set to 1 and bit #62 is set to 0 are reserved for AUTOSAR-defined error domains.*♩(RS\_AP\_00130)*

**[SWS\_CORE\_00016]**{DRAFT} **Vendor-defined error domain range** *[*Error domain identifiers where the top 32 bits (i.e. bit #63..#32) are equal to 0xc000’0000 are re- served for vendor-specific error domains. Bits #31..#16 hold the vendor’s numerical identifier, and bits #15..#0 can be used by each vendor for error domain identifiers.*♩ (RS\_AP\_00130)*

**[SWS\_CORE\_00013] The Future error domain** *[*There shall be an error domain ara::core::FutureErrorDomain for all errors originating from the interaction of the classes ara::core::Future and ara::core::Promise. It shall have the shortname Future and the identifier 0x8000’0000’0000’0013.*♩(RS\_AP\_00130)*

**[SWS\_CORE\_00014] The Core error domain** *[*There shall be an error domain ara::core::CoreErrorDomain for errors originating from non-Future/Promise facilities of ara::core. It shall have the shortname Core and the identifier 0x8000’0000’0000’0014.*♩(RS\_AP\_00130)*

### Async signal safety

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

[6, The POSIX standard] defines a set of functions that are guaranteed to be async- signal-safe; all functions not on that list need to be assumed unsuitable to be called within a signal handler. This includes all ARA APIs, as it is not specified (and in general not possible to determine) which other functions (whether from POSIX or from other standards or implementations) are called within them.

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

### Explicit Operation Abortion

If a [Violation](#_bookmark7) has been detected by the implementation of an API function, [[SWS\_CORE\_00003](#_bookmark11)] mandates to abort this operation immediately. It allows two ways to do this; either by throwing certain kinds of exceptions (if the implementation supports C++ exceptions), or by calling ara::core::Abort.

Calling ara::core::Abort will result in an Explicit Operation Abortion, which usually leads to an Unexpected Termination as defined by [7]. This section defines the behavior of this mechanism.

Like std::abort, calling ara::core::Abort is meant to terminate the current pro- cess abnormally and immediately, without performing stack unwinding and without call- ing destructors of static objects.

**[SWS\_CORE\_12402]**{DRAFT} **“Noreturn” property for Abort** *[*The function ara:-

:core::Abort shall not return to its caller.*♩(RS\_AP\_00130)*

**[SWS\_CORE\_12403]**{DRAFT} **Logging of Explicit Operation Abortion** *[*Calling ara::core::Abort shall result in a log message, which shall contain the string that has been passed to the function as argument, being output to the process’s standard error stream.*♩(RS\_AP\_00130)*

**[SWS\_CORE\_12407]**{DRAFT} **Thread-safety of Explicit Operation Abortion** *[*While a call to ara::core::Abort is in progress, other calls to this function shall block the calling threads.*♩(RS\_AP\_00130)*

ara::core::Abort provides a means to add a “hook” into the system, by calling ara::core::SetAbortHandler, similar to the way std::atexit allows to install a callback for the std::exit mechanism.

**[SWS\_CORE\_12404]**{DRAFT} **AbortHandler invocation** *[*Calling ara::core:-

:Abort shall invoke the AbortHandlers after the log message as per [[SWS\_CORE\_12403](#_bookmark34)] has been output, in the reverse order of installation.*♩(RS\_AP\_- 00130)*

### AbortHandler

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

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

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

Similarly, deferring function return by entering an infinite loop is discouraged as well; while this still leads to the desired outcome that the *operation* which caused a [Viola-](#_bookmark7) [tion](#_bookmark7) has been aborted, it will do so at the cost of “defunct’ing” the calling thread and risking the destabilization of the software, which already has encountered a [Viola-](#_bookmark7) [tion](#_bookmark7).

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

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

**[SWS\_CORE\_12405]**{DRAFT} **Final action without AbortHandler** *[*If there is no custom ara::core::AbortHandler that has been installed with ara::core::- SetAbortHandler or ara::core::AddAbortHandler, then the implementation of ara::core::Abort shall call std::abort().*♩(RS\_AP\_00130)*

**[SWS\_CORE\_12406]**{DRAFT} **Final action with returning AbortHandlers** *[*If there are custom ara::core::AbortHandlers that have been installed with ara::- core::SetAbortHandler or ara::core::AddAbortHandler and all of them re- turn, then the implementation of ara::core::Abort shall call std::abort().*♩ (RS\_AP\_00130)*

### SIGABRT handler

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

The signal handler for SIGABRT has the same choices of actions as the ara::core:-

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

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

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

### Advanced data types

* + - 1. **AUTOSAR types**
         1. **InstanceSpecifier**

Instances of ara::core::InstanceSpecifier are used to identify service port prototype instances within the AUTOSAR meta-model and are therefore used in the ara::com API and elsewhere. A detailed description and background can be found in [8] sections 6.1 (“Instance Identifiers”) and 9.4.4 (“Usage of meta-model identifiers within ara::com based application code”).

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

**[SWS\_CORE\_10200] Valid InstanceSpecifier representations - application in- teraction** *[*In case of application interaction and thus in the presence of PortPro- totypes the string representation of a valid ara::core::InstanceSpecifier consists of a "/"-separated list of model element shortNames starting from an Ex- ecutable via the RootSwComponentPrototype and several SwComponentPro- totypes to the respective PortPrototype to which the ara::core::Instance- Specifier shall apply.*♩(RS\_AP\_00130)*

Thus, in case of application interaction the content of a valid ara::core::In- stanceSpecifier adheres to the following pattern: Executable.shortName/RootSwComponentPrototype.shortName

/SwComponentPrototype.shortName/.../PortPrototype.shortName

**[SWS\_CORE\_10203] Valid InstanceSpecifier representations - functional clus- ter interaction** *[*In case of functional cluster interaction and thus in the absence of PortPrototypes the string representation of a valid ara::core::Instance- Specifier consists of a "/"-separated list of model element shortNames starting from a top-level ARPackage via contained sub-packages to the respective mapping element that is derived from FunctionalClusterInteractsWithFunctional- ClusterMapping (see TPS\_MANI\_03268 for further details).*♩(RS\_AP\_00130)*

Thus, in case of functional cluster interaction the content of a valid ara::core::- InstanceSpecifier adheres to the following pattern:

ARPackage.shortName/.../ARPackage.shortName

/FunctionalClusterInteractsWithFunctionalClusterMapping.shortName

**[SWS\_CORE\_10201] Validation of meta-model paths** *[*The construction mech- anisms of class ara::core::InstanceSpecifier shall reject meta-model paths that are syntactically invalid according to the syntax rules defined in [[SWS\_CORE\_10200](#_bookmark40)].*♩(RS\_AP\_00130)*

**[SWS\_CORE\_10202] Construction of InstanceSpecifier objects** *[*APIs for con- struction of ara::core::InstanceSpecifier objects shall be available in both potentially-throwing and non-throwing form.*♩(RS\_AP\_00130)*

### ScaleLinearAndTexttable

A ara::core::ScaleLinearAndTexttable type is a struct type that emulates an enumeration type with extended capabilities, such as those given in [9, the C++17 standard]

In particular, it can hold the values of the enumeration, but also any value of the under- lying type of the Enumeration Data Type with which it was defined.

### Types derived from the base C++ standard

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

Some types are already contained in [4, the C++14 standard]; however, types with al- most 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 allo- cation behavior, and perform some other necessary adaptions as well, including about the throwing of exceptions.

**[SWS\_CORE\_00040]**{DRAFT} **Errors originating from C++ standard classes** *[*For the classes in ara::core specified below in terms of the corresponding classes of the C++ standard, all functions that are specified by [4, the C++14 standard], [9, the C++17 standard], or [10, the draft C++20 standard] to throw any exceptions, are instead spec- ified to be the cause of a [Violation](#_bookmark7) when they do so.*♩(RS\_AP\_00130)*

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

### Array

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

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

These differences to std::array are intended:

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

**[SWS\_CORE\_11200] Array base behavior** *[*ara::core::Array and all its member functions and supporting constructs shall behave identical to those of std::array in header <array> from [4, the C++14 standard], except for the differences specified in this document.*♩(RS\_AP\_00130)*

### Vector

This section describes the ara::core::Vector type that represents a container of variable size.

**[SWS\_CORE\_11300]**{DRAFT} **Vector base behavior** *[*ara::core::Vector and all its member functions and supporting constructs shall behave identical to those of std::vector in header <vector> from [4, the C++14 standard], except for the dif- ferences specified in this document.*♩(RS\_AP\_00130)*

### Map

This section describes the ara::core::Map type that represents an associative con- tainer of variable size.

**[SWS\_CORE\_11400]**{DRAFT} **Map base behavior** *[*ara::core::Map and all its member functions and supporting constructs shall behave identical to those of std::map in header <map> from [4, the C++14 standard], except for the differences specified in this document.*♩(RS\_AP\_00130)*

### String and BasicString

This section describes the ara::core::String and ara::core::BasicString

types.

**[SWS\_CORE\_12000]**{DRAFT} **String base behavior** *[*ara::core::String, ara::core::BasicString and all their member functions and supporting con- structs shall behave identical to those of std::string and std::basic\_string in header <string> from [4, the C++14 standard], except for the differences specified in this document.*♩(RS\_AP\_00130)*

### SteadyClock 7.2.4.2.5.1 Definitions of terms

The C++ std::chrono library defines a number of concepts and types for handling time and durations. One of these concepts is that of a “clock” which is able to create snapshots of specific “time points”. When talking about clocks and time points, the

three qualities *resolution*, *precision*, and *accuracy* are distinguished within this docu- ment as follows:

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

For clocks of the POSIX clock\_gettime API, the [resolution](#_bookmark49) is implicitly de- fined as nanoseconds by the API’s usage of struct timespec with its time- spec::tv\_nsec field.

For C++ clocks of the std::chrono APIs, the [resolution](#_bookmark49) is variable.

The precision of a clock is the smallest time interval that its timer is able to measure. The [precision](#_bookmark50) is implementation-defined and depends on the prop- erties and capabilities of the physical machine as well as the operating system.

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

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

Clocks that place more emphasis on high [precision](#_bookmark50) often do not relate to calendar time at all, but generate timestamps as offsets from something like the power-up time of the system.

### Clocks in the Adaptive Platform

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

However, the C++ standard does not place any requirements on the [resolution](#_bookmark49), [precision](#_bookmark50), and [accuracy](#_bookmark51) of this clock. The undefinedness of its [resolution](#_bookmark49) can pose some difficulties for application programmers, but these can usually be solved by agreeing on a common – or minimum – [resolution](#_bookmark49). The [precision](#_bookmark50) and [ac-](#_bookmark51) [curacy](#_bookmark51) are always dependent on the physical properties of the machine and of the operating system.

The Adaptive Platform defines ara::core::SteadyClock as a std::chrono- compatible clock with nanosecond [resolution](#_bookmark49) and a std::int64\_t datatype. Its [precision](#_bookmark50) and [accuracy](#_bookmark51) are still implementation-defined and can be given as char- acteristic values of a concrete platform. Its [epoch](#_bookmark52) is the power-up time of the ECU. With these properties, timestamps generated by ara::core::SteadyClock will not overflow until 292 years after its [epoch](#_bookmark52).

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

The properties of ara::core::SteadyClock imply that a type alias to

std::chrono::steady\_clock is a conforming implementation of ara::core:-

:SteadyClock, if std::chrono::steady\_clock::period is equivalent to std::nano, and std::chrono::steady\_clock::rep is a 64-bit signed integer type such as std::int64\_t.

**[SWS\_CORE\_11800] SteadyClock type requirements** *[*Class ara::core::- SteadyClock shall meet the requirements of *TrivialClock* from [4, the C++14 stan- dard].*♩(RS\_AP\_00130)*

**[SWS\_CORE\_11801] Epoch of SteadyClock** *[*The [epoch](#_bookmark52) of ara::core::Steady- Clock shall be the system start-up.*♩(RS\_AP\_00130)*

### Types derived from newer C++ standards

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

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

### Optional

This section describes the ara::core::Optional type.

**[SWS\_CORE\_11000]**{DRAFT} **Optional base behavior** *[*ara::core::Optional and all its member functions and supporting constructs shall behave identical to those of std::optional in header <optional> from [9, the C++17 standard], except for the differences specified in this document.*♩(RS\_AP\_00130)*

Note: The value() function and the bad\_optional\_access exception defined in the C++ standard library are left out of this specification to provide an API that does not make use of exceptions. Use either has\_value or operator bool() to check if the ara::core::Optional contains a value before accessing the value with e.g., op- erator\*. Alternatively, use the value\_or functions to access the value and provide a default value in case the ara::core::Optional contains no value.

**[SWS\_CORE\_01030]**{DRAFT} **value member function overloads** *[*Contrary to the description in [9], no member functions with this name exist in ara::core::Op- tional.*♩(RS\_AP\_00130)*

**[SWS\_CORE\_01031]**{DRAFT} **class bad\_optional\_access** *[*No class named

bad\_optional\_access is defined in the ara::core namespace.*♩(RS\_AP\_00130)*

### Variant

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

**[SWS\_CORE\_11600]**{DRAFT} **Variant base behavior** *[*ara::core::Variant and all its member functions and supporting constructs shall behave identical to those of std::variant in header <variant> from [9, the C++17 standard], except for the differences specified in this document.*♩(RS\_AP\_00130)*

### StringView

This section describes the ara::core::StringView type that represents a read- only view over a contiguous sequence of characters whose storage is owned by an- other object.

**[SWS\_CORE\_12200]**{DRAFT} **StringView base behavior** *[*ara::core::- StringView and all its member functions and supporting constructs shall behave identical to those of std::string\_view in header <string\_view> from [9, the C++17 standard], except for the differences specified in this document.*♩(RS\_AP\_- 00130)*

### Span

ara::core::Span is a type that represents an abstraction over a linear sequence of values of a certain type. It is closely modeled on std::span from C++20, with deviations mostly coming from the lack of C++20’s “ranges” feature.

**[SWS\_CORE\_11900]**{DRAFT} **Span base behavior** *[*ara::core::Span and all its member functions and supporting constructs shall behave identical to those of std::span in header <span> from [10, the draft C++20 standard], except for the differences specified in this document.*♩(RS\_AP\_00130)*

### ara::core::Byte

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

The definitions of this section have been carefully set up in a way to make std::byte from [9, the C++17 standard] a conforming implementation, but also allow a class- based implementation with only C++14 means.

Unlike std::byte from [9, the C++17 standard], it is implementation-defined whether ara::core::Byte can be used for type aliasing without triggering Undefined Behav- ior.

**[SWS\_CORE\_10100] Type property of ara::core::Byte** *[*The type ara:-

:core::Byte shall not be an integral type. In particular, the value

std::is\_integral<ara::core::Byte>::value shall be 0.*♩(RS\_AP\_00130)*

**[SWS\_CORE\_10101] Size of type ara::core::Byte** *[*The size (in bytes) of an instance of type ara::core::Byte (determined with sizeof(ara::core::Byte)) shall be

1.*♩(RS\_AP\_00130)*

**[SWS\_CORE\_10102] Value range of type ara::core::Byte** *[*The value of an instance of type ara::core::Byte shall be constrained to the range [0..std::numeric\_limits<unsigned char>::max()].*♩(RS\_AP\_00130)*

**[SWS\_CORE\_10103] Creation of ara::core::Byte instances** *[*An instance of type ara::core::Byte shall be creatable from an integral type with brace-initialization syntax. This initialization shall also be possible when called in a constant expres- sion. If the initializer value is outside the value range of type ara::core::Byte (see [[SWS\_CORE\_10102](#_bookmark60)]), the behavior is undefined.*♩(RS\_AP\_00130)*

**[SWS\_CORE\_10104] Default-constructed ara::core::Byte instances** *[*An instance of type ara::core::Byte shall be constructible without giving an initializer value. Such a variable definition shall incur no runtime cost, and the value of the instance shall have indeterminate content.*♩(RS\_AP\_00130)*

**[SWS\_CORE\_10105] Destructor of type ara::core::Byte** *[*The destructor of type

ara::core::Byte shall be trivial.*♩(RS\_AP\_00130)*

**[SWS\_CORE\_10106] Implicit conversion from other types** *[*The type ara::- core::Byte shall not be implicitly convertible from any other type.*♩(RS\_AP\_00130)*

**[SWS\_CORE\_10107] Implicit conversion to other types** *[*The type ara::core:-

:Byte shall allow no implicit conversion to any other type, including bool.*♩(RS\_AP\_- 00130)*

**[SWS\_CORE\_10108] Conversion to unsigned char** *[*The type ara::core::Byte shall allow conversion to unsigned char with a static\_cast<> expression. This conversion shall also be possible when called in a constant expression.*♩(RS\_AP\_- 00130)*

**[SWS\_CORE\_10109] Equality comparison for ara::core::Byte** *[*The type ara:-

:core::Byte shall be comparable for equality with other instances of type ara:-

:core::Byte. This comparison shall also be possible when called in a constant expression.*♩(RS\_AP\_00130)*

**[SWS\_CORE\_10110] Non-equality comparison for ara::core::Byte** *[*The type ara::core::Byte shall be comparable for non-equality with other instances of type ara::core::Byte. This comparison shall also be possible when called in a constant expression.*♩(RS\_AP\_00130)*

### Initialization and Shutdown

This section describes the global initialization and shutdown of the ARA framework. Before the framework is initialized, and after the it is deinitialized, not all ARA function- ality may be available.

While it is usually possible for a framework implementation to initialize all parts of the framework in an “initialize on first use” fashion, this might not always be desirable, as it introduces potentially noticeable delays during runtime.

For this reason, there exist initialization and shutdown functions that may be used by the framework vendor to initialize/shutdown the framework to an extent that no lazy initialization during runtime is necessary.

On the other hand, another framework implementation might well have empty imple- mentations of these functions, e.g. if this framework chooses to fully adopt the “initialize on first use” idiom.

**[SWS\_CORE\_15003]**{DRAFT} **Startup and initialization of ARA** *[*The ara::- core::Initialize function shall initiate the start-up of the ARA framework, which might include (but is not limited to):

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

*♩(RS\_Main\_00011)*

**[SWS\_CORE\_15004]**{DRAFT} **Shutdown and de-initialization of ARA** *[*The ara:-

:core::Deinitialize function shall initiate the shutdown of the ARA framework, which might include (but is not limited to):

* orderly shutdown of spawned background threads
* deallocation of dynamically allocated memory
* deallocation of other system resources

*♩(RS\_Main\_00011)*

An error returned by ara::core::Deinitialize is the only way for the ARA to re- port 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 deinitial- ization without ara::log. A typical error case to be reported here is that the user is still holding some resource from the ARA.

Calling ara::core::Deinitialize while ARA APIs are still being called concur- rently results in undefined behavior of the application and the framework.

For a proper shutdown, it is also expected that ara::core::Deinitialize is called before the statically initialized data is destructed.

### [SWS\_CORE\_15001]{DRAFT} Handling of interaction with the ARA of an un-

**/deinitialized runtime** *[*A call to any ARA API (other than the ones mentioned in [[SWS\_CORE\_15002](#_bookmark62)]) without prior call to ara::core::Initialize shall be treated by the Functional Cluster implementation as a [Violation](#_bookmark7).*♩(RS\_AP\_00142)*

The rationale to treat this as a [Violation](#_bookmark7) is that such occurrences cannot be handled by the caller of the API at the point in time where the error is detected. Aborting exe- cution is the only way to signal this kind of systematic error and prevent later failures.

**[SWS\_CORE\_15002]**{DRAFT} **Special ara::core types to be used without initial- ization** *[*A small subset of ara::core types and functions shall be usable indepen- dently of initialization with ara::core::Initialize. These are:

* ara::core::ErrorCode and all its member functions and supporting con- structs (including non-member operators)
* ara::core::StringView and all its member functions and supporting con- structs (including non-member operators)
* ara::core::Result and all its member functions and supporting constructs, except for ara::core::Result::ValueOrThrow
* ara::core::ErrorDomain and all its member functions and its subclasses, as long as they adhere to [[SWS\_CORE\_10400](#_bookmark15)], but excluding <Prefix>Er- rorDomain::ThrowAsException.

*♩(RS\_AP\_00142)*

The rationale for the exception for this subset is the intended use before initialization and that these types are used as part of the initialization (ara::core::Result, ara::core::ErrorCode, ara::core::ErrorDomain).