Document Title	Guidelines for using Adaptive Platform interfaces
Document Category	Explanation (EXP)
Document UID	929
Document Status	published
Part of AUTOSAR Standard	Adaptive Platform
Part of Standard Release	R26-11

Document Change History

Date	Release	Changed by	Description
2026-11-26	R26-11	AUTOSAR Release Management	• Added the Cryptography chapter and ara::crypto interface guidance. • Added C++ compatibility guidelines for application developers. • Updated Execution Management error codes in code examples.
2025-11-27	R25-11	AUTOSAR Release Management	• Introduction of Sensor Interfaces • Updates of Persistency, RawDataStream, Diagnostics, Execution Management, and Platform Health Management
2024-11-27	R24-11	AUTOSAR Release Management	• Updates of Raw Data Stream, Platform Health Management, Update and Config Management, Adaptive Core chapters according to the respective SWS changes • Various clean-ups
2023-11-23	R23-11	AUTOSAR Release Management	• Introduction of the Common Regulations, Diagnostics Management, and the Raw Data Streams chapter • Overall improvement of the Execution Management, Update and Configuration Management, and Persistency chapters • Overall update of the State Management chapter • Removal of the Deterministic Execution Client in the Execution Management chapter • Minor updates of Adaptive Core and Platform Health Management chapters

▽

△

2022-11-24	R22-11	AUTOSAR Release Management	• A new chapter - Update and Configuraiton Management
2021-11-25	R21-11	AUTOSAR Release Management	• A new chapter "Common Regulations" added • Minor updates in the Persistency chapter
2020-11-30	R20-11	AUTOSAR Release Management	• The name of the chapter "Core Types" to "Adaptive Core" and some minor changes in the chapter • Moderate changes in the State Management chapter • Minor changes in the Persistency Chapter
2019-11-28	R19-11	AUTOSAR Release Management	• Persistency and Platform Health Management chapters added • Changed Document Status from Final to published
2019-03-29	19-03	AUTOSAR Release Management	• Clause 4 revised to reflect the updated design on State Management
2018-10-31	18-10	AUTOSAR Release Management	• Initial release

Disclaimer

This work (specification and/or software implementation) and the material contained in it, as released by AUTOSAR, is for the purpose of information only. AUTOSAR and the companies that have contributed to it shall not be liable for any use of the work.

The material contained in this work is protected by copyright and other types of intellectual property rights. The commercial exploitation of the material contained in this work requires a license to such intellectual property rights.

This work may be utilized or reproduced without any modification, in any form or by any means, for informational purposes only. For any other purpose, no part of the work may be utilized or reproduced, in any form or by any means, without permission in writing from the publisher.

The work has been developed for automotive applications only. It has neither been developed, nor tested for non-automotive applications.

The word AUTOSAR and the AUTOSAR logo are registered trademarks.

Table of Contents

1 Introduction to this document 7
1.1 Contents . . . 7
1.2 Prereads . . . 7
1.3 Relationship to other AUTOSAR specifications . . . 7
2 Common Regulations 8
2.1 Namespace . . . 8
2.2 Client Code Restrictions and Assumptions . . . 8
2.2.1 Reliance on Exact Function Signatures . . . 8
2.2.2 Reliance on Exact Overload Sets . . . 9
2.2.3 Template Metaprogramming and Compile-Time Introspection . . . 9
2.2.4 Detection of Members and Functions . . . 9
2.2.5 Reliance on Exact Template Definitions . . . 9
2.2.6 Dependence on Aggregate Properties . . . 10
2.2.7 Dependence on Implicit Conversions . . . 10
2.2.8 Use of Inheritance from Non-Interface Classes . . . 10
2.2.9 Overriding Undocumented Virtual Functions . . . 10
2.2.10 Reliance on Namespace Contents . . . 10
2.2.11 Dependence on Header Structure . . . 11
2.2.12 Reliance on Public Constants for Compile-Time Decisions . . . 11
2.2.13 Dependence on Exact Type Definitions . . . 11
2.2.14 Use of Deprecated API Elements . . . 11
3 Adaptive Core 12
3.1 Error handling . . . 12
3.1.1 ErrorCode . . . 12
3.1.2 Result . . . 12
3.1.2.1 Creation of a Result . . . 13
3.1.2.2 Retrieving values and errors . . . 13
3.1.2.3 Advanced topics . . . 14
3.1.3 Faults inside constructors . . . 15
3.1.4 Non-Recoverable Errors . . . 16
3.1.5 Standardized vs Vendor-specific ErrorCodes . . . 16
3.2 Startup and Shutdown . . . 17
3.3 Reserved symbols . . . 18
3.3.1 Preprocessor macros . . . 18
3.4 Thread-safety . . . 18
3.5 Async signal-safety . . . 18
4 Execution Management 19
4.1 Execution Client . . . 19
4.2 Exit code . . . 20
4.3 State Client . . . 20

5 State Management 24
5.1 Interaction with AUTOSAR Adaptive (Platform) Applications 24
5.1.1 Basic State Management functionality 24
5.1.2 State Management StateMachine functionality 24
6 Persistency 27
6.1 Overview 27
6.1.1 Key-Value Storage 27
6.1.2 File Storage 28
6.1.3 Configuration 28
6.2 Redundancy feature 28
6.3 Reset Storage 29
6.4 Update and Removal of Persistent Data 29
6.5 Example usage of the Key-Value Storage API 29
6.6 Example usage of File Storage API 31
7 Platform Health Management 33
7.1 Supervision and Checkpoints 33
7.2 Recovery actions 33
7.3 Watchdog handling 34
7.4 Shutdown functionality 34
8 Update and Configuration Management 35
8.1 Port configuration 35
8.2 Method deployments 35
8.3 Data transfer, message loss and block sizes 36
9 Raw Data Streams 38
9.1 Raw Data Streaming over Ethernet using IP based protocols (network layer) 38
9.1.1 Raw Data Streaming Interface 38
9.1.1.1 Introduction 38
9.1.1.2 Functional description 38
9.1.1.3 Class and Model 39
9.1.1.4 Methods of class RawDataStream 40
9.1.1.5 Security 43
9.1.2 Usage of RawDataStreaming 43
9.1.2.1 Sequence diagrams 43
9.1.2.2 Usage 45
9.2 Raw Data Streaming using IEEE1722 protocol (data link layer) 47
9.2.1 Raw Data Streaming Interface 47
9.2.1.1 Introduction 47
9.2.1.2 Functional description 47
9.2.1.3 Class and Model 48
9.2.1.4 Methods of class IEEE1722RawDataStreamConsumer/Producer 50

9.2.1.5 Security . . . 52
9.2.2 Usage of RawDataStreaming . . . 52
9.2.2.1 Usage . . . 52
9.3 Safety . . . 54
9.4 Hints for implementers . . . 54
10 Diagnostic Management 55
10.1 Diagnostic Communication Management . . . 55
10.2 Diagnostic Event Management . . . 55
10.3 Negative Return Codes . . . 56
10.4 Diagnostic communication over Internet Protocol . . . 56
10.5 Service oriented Vehicle Diagnostics . . . 57
11 Cryptography 58
11.1 Modeling and Mapping . . . 58
11.1.1 Design-time Modeling . . . 58
11.1.1.1 KeySlot Design Steps . . . 58
11.1.2 Integration-time Mapping . . . 59
11.1.3 Development . . . 60
11.1.4 Modeling Examples . . . 60
11.1.4.1 CryptoProvider . . . 60
11.1.4.2 KeySlot . . . 60
11.1.4.3 Certificate . . . 61
11.1.4.4 CertificateGroup . . . 61
11.2 CryptoAPIs . . . 62
11.2.1 Crypto Provider . . . 62
11.2.1.1 Key Management . . . 62
11.2.1.2 Cryptographic Transformations . . . 66
11.2.2 Certificate Handling (X.509 Provider) . . . 69
11.2.2.1 X.509 Provider Overview . . . 69
11.2.2.2 Certificate . . . 69
11.2.2.3 Certificate Signing Request (CSR) . . . 72
11.2.2.4 Certificate Revocation Lists (CRLs) . . . 72
11.2.2.5 Online Certificate Status Protocol (OCSP) . . . 73
12 Sensor Interfaces 74
12.1 Interface Configuration . . . 74
12.2 Data Transfer . . . 74
12.3 Service profile and data type . . . 74
13 Related Documentation 75

1 Introduction to this document

1.1 Contents

While SWS of FC is a specification for ARA interfaces, some of the interfaces require "guidelines" on how to use them. The guidelines are indeed related to the specification, but some are indirect and having such information within each SWS bloats SWS hence making it difficult for readers to grasp the usage. Another important perspective is that these guidelines are a kind of requirement against AA to follow, but SWS of FC are specification requirements for FCs. Therefore, it does not fit well to have these contents in SWS, and this is the purpose of this "Guidelines for using Adaptive Platform Interfaces."

The main contents of this document will be the guidelines for applications to follow as mentioned in the background above. Not necessarily all FCs will have contents in this document; they will be added when it deems valid.

The contents are organized per relevant topic, but in general, this will be grouped by FC, each having its independent chapter. Also, note that the contents may be provided in separate AUTOSAR AP documents. If this is the case, such documents will be listed or referenced from this guideline.

1.2 Prereads

This document is a supplementary document to the SWS of AP. Therefore, the relevant SWS of the topic in these guidelines should be read in parallel. Also, the first AP document to be read is [1], which gives the architectural overview of AP.

1.3 Relationship to other AUTOSAR specifications

Refer to Contents 1.1 and Prereads 1.2.

2 Common Regulations

2.1 Namespace

The namespace `ara` (and all namespaces below it) are reserved by AUTOSAR.

The behavior of an Adaptive Application is undefined if it adds declarations or definitions to namespace `ara` or to a namespace within namespace `ara`.

Note: the standardized namespaces could have been extended by the platform vendors to be able to implement the standard or to provide extensions.

Data type declarations usually are also available via forwarding header files.

All data types for a particular Functional Cluster are available in a single forwarding header file whose filename follows the pattern `<fc>_fwd.h`. For instance, forwarding declarations of all `ara::core` types are available in a header file `ara/core/core_fwd.h`.

2.2 Client Code Restrictions and Assumptions

AUTOSAR defines rules according to which API changes are considered source compatible. These compatibility guarantees apply only to normal use of the documented API. Client code that relies on implicit implementation details, language corner cases, or compile-time introspection may require adaptation even when an API change is classified as source compatible.

2.2.1 Reliance on Exact Function Signatures

Client code shall not rely on exact function signatures.

Changes such as adding default arguments or additional overloads may be source compatible for ordinary function calls.

Client code shall not rely on exact function signatures through constructs such as:

```
auto ptr = &Api::Function;
```

or

```
using FunctionType = void(*)(void);
```

Such code may become invalid when the API evolves.

2.2.2 Reliance on Exact Overload Sets

Client code shall not assume that the set of overloads for a function remains unchanged.

For example:

```
Function(0);
```

may resolve differently if additional overloads are introduced.

Applications should avoid ambiguous calls and should use explicit casts where necessary.

2.2.3 Template Metaprogramming and Compile-Time Introspection

Compatibility guarantees do not apply to code that performs compile-time analysis of API declaration.

Examples include:

```
decltype(&Api::Function)
```

```
std::is_invocable_v<...>
```

```
std::void_t<...>
```

```
requires(...) // as soon as AUTOSAR makes use of C++20 Concepts
```

Changes that affect such constructs are not considered source incompatible by this standard.

2.2.4 Detection of Members and Functions

Client code shall not depend on the presence or absence of specific members solely through detection idioms.

Example:

```
HasResetMethod<T>
```

A source-compatible API extension may invalidate such assumptions.

2.2.5 Reliance on Exact Template Definitions

Client code shall not depend on implementation details of public template definitions unless explicitly documented as part of the API.

Changes to template implementations may affect metaprogramming constructs while remaining source compatible under this standard.

2.2.6 Dependence on Aggregate Properties

Client code **should not** rely on a type remaining an aggregate.

Example:

```
Point p{1, 2};
```

Future API evolution may introduce constructors or other members that change aggregate behavior. *Note that this might be especially interesting with [changes introduced in C++20](https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1008r1.pdf) as soon as AUTOSAR updates the required C++ version.*

2.2.7 Dependence on Implicit Conversions

Client code should avoid reliance on implicit conversions where explicit alternatives exist.

Additional overloads or constructors may alter overload resolution.

Example:

```
// Instead ...
Function(0);

// ... prefer the following when a specific overload is intended:
Function(static_cast<int>(0));
```

2.2.8 Use of Inheritance from Non-Interface Classes

Client code should derive only from classes explicitly documented for inheritance.

Future source-compatible changes may introduce new virtual functions, members, or implementation details that affect derived classes.

2.2.9 Overriding Undocumented Virtual Functions

Client code shall only override virtual functions intended as extension points.

Functions not documented for customization may change in future versions.

2.2.10 Reliance on Namespace Contents

Client code shall not assume that a namespace contains only the currently available declarations.

New types, functions, aliases, or enumerators may be added as part of source-compatible API evolution.

2.2.11 Dependence on Header Structure

Client code should include the header that directly declares the required API element.

Transitive include relationships are not guaranteed to remain unchanged.

2.2.12 Reliance on Public Constants for Compile-Time Decisions

Client code should avoid compile-time branching based on undocumented API constants, traits, or helper types.

Such implementation details may evolve without being considered source incompatible.

2.2.13 Dependence on Exact Type Definitions

Client code shall not rely on implementation-specific details of exposed types beyond their documented interface.

Client code should not assume a specific inheritance hierarchy, nested type set, or collection of helper members *unless documented as part of the API*.

```
class Container
{
public:
    using value_type = int;
};

// Client code
Container::value_type value;
//  If value_type is not part of the documented API contract,
//  its removal in a newer AUTOSAR release is source compatible.
```

2.2.14 Use of Deprecated API Elements

Client code should migrate away from deprecated interfaces as early as practical.

Deprecated API elements remain source compatible during the deprecation period. Their eventual removal is considered a source-incompatible change.

3 Adaptive Core

3.1 Error handling

Handling errors is a crucial topic for any software development. For safety-critical software, it is even more important, because lives can depend on it. However, current standards for the development of safety-critical software places significant restrictions on the build toolchain, especially with regard to C++ exceptions. For ASIL applications, using C++ exceptions is usually not possible due to the lack of exceptions support with ASIL-certified C++ compilers.

The Adaptive Platform introduces a concept that enables error handling without C++ exceptions and defines a number of C++ data types to aid in this.

From an application programmer's point of view, the central types implementing this concept are ara::core::ErrorCode and ara::core::Result.

3.1.1 ErrorCode

An instance of ara::core::ErrorCode represents a specific error condition within a software. It is similar to std::error_code, but differs in significant aspects from it.

An ErrorCode always contains an enumeration value (type-erased into an integral type) and a reference to an error domain. The enumeration value describes the specific type of error, and the error domain reference defines the context where that error is applicable. Additional optional members are a user-defined message string and a vendor-defined supplementary error description value.

3.1.2 Result

Class ara::core::Result follows the "ValueOrError" concept from the C++ proposal p0786 (see `https://wg21.link/P0786`). It either contains a value or an error. ErrorType is defaulted to ara::core::ErrorCode, and it is enforced that only this type is allowed as the ErrorType template parameter for ara::core::Result.

Because the ErrorType is defaulted to ara::core::ErrorCode, most declarations of ara::core::Result only need to give the ValueType, e.g. ara::core::Result<int> for a Result type that contains either an int variable, or an ErrorCode.

ARA interfaces use ara::core::Result as the return type for functions that can encounter recoverable errors. This type can be used to either generate a C++ exception from the object if the user chooses to use exceptions, or retrieve error information via observer methods without using exceptions.

This section guides you on how to handle such Result objects returned from ARA interface in your application code, and also gives guidance on how to create new Result objects within your own Adaptive Application.

3.1.2.1 Creation of a Result

For creating a Result with an embedded value, there are constructors allowing implicit conversion from a ValueType. This makes defining a Result with a value quite straightforward:

```
Result<int> res1(42);
Result<int> res2 = 42;
```

Returning a value from a function declared to return a Result is similarly straightforward:

```
Result<int> myfunction()
{
  return 42;
}
```

Putting an error inside a Result requires calling an explicit constructor, e.g.:

```
ErrorCode ec = MyEnum::some_error;
Result<int> res2(ec);
```

Alternatively, construction of Result objects is also possible with static member functions, for instance:

```
Result<int> res1 = Result<int>::FromValue(42);
Result<int> res2 = Result<int>::FromError(ec);
```

These forms can be advantageous when ValueType or ErrorType are expensive to copy because they allow in-place construction. For instance, returning a Result containing an instance of BigClass which is constructed with two constructor arguments "a1" and "a2" could look like this:

```
return Result<BigClass>::FromValue(a1, a2);
```

For ErrorType, this also allows implicit construction of the ErrorCode instance, including an optional supplementary data value:

```
return Result<BigClass>::FromError(
  MyEnum::some\_error,              // ErrorCode enum value
  0x12345678                       // support data value
);
```

With this form of construction, only one constructor call is performed, unlike the regular (unnamed) constructor call, where at least two constructor calls are performed, because the pre-created value must then be copied or moved into the Result instance.

3.1.2.2 Retrieving values and errors

When trying to retrieve the value or error that is contained within a Result, one first has to consider which one of these (value or error) is available. In general, this is not known, so one has to take care to handle both cases.

When working without exceptions, the Result object is queried to check whether it contains a value or an error:

```
Result<int> some_function() { ... }

Result<int> res = some_function();
if (res.HasValue()) {
  int theValue = res.Value();
} else {
  ErrorCode const& ec = res.Error();
}
```

This code also works in a completely exception-free environment, including with a compiler that does not support exceptions at all.

When working with an exception-based workflow, the query code looks quite similar to regular exception-based code:

```
Result<int> some_function() { ... }

int theValue = some_function().ValueOrThrow();
```

Here, the Result object that is returned by some_function() is immediately reduced to its ValueType (int) by calling its ValueOrThrow() member function.

If the Result did contain an ErrorCode, this would immediately throw an exception type that corresponds to the embedded ErrorCode object.

Naturally, a try...catch block should be added at a suitable location in the code.

3.1.2.3 Advanced topics

The two basic methods for retrieving the embedded value or error are called just as such: Result::Value() and Result::Error(). However, when calling any of these, one has to be certain that the Result object does indeed contain what is implied by calling one of these functions. In the previous section, this was done by first calling Result::HasValue(), and calling Value() or Error() depending on the outcome of that call.

A more convenient way of accessing the embedded value has already also been mentioned in the previous section: By calling Result::ValueOrThrow, no if-statement is needed, and the invocation collapses into a single-line statement (excluding the try...catch block, which might exist elsewhere).

Other convenience methods exist, for instance Result::ValueOr, which retrieves the value, if if exists, or takes a default value otherwise (i.e., in case of any error), e.g.:

```
int res = some_function().ValueOr(42);
```

A generalization of Result::ValueOr is called Result::Resolve, which does not take a default value as an argument, but a Callable, which is to create the default value on-demand:

```
int res = some\_function()
```

```
2     .Resolve([](ErrorCode const& ec){ return 42; });
```

For this particular example, using Result::Resolve instead of Result::ValueOr does not make much sense. However, it can be advantageous when the default value is expensive to create. By using Result::Resolve, the default value is only created when it is needed.

Another convenience method is Result::Bind, which allows to transform the contained value into another value, or even into another type. For instance:

```
1   Result<String> res = some_function()
2     .Bind([](int v){ return v + 1; })
3     .Bind([](int v){ return std::to_string(v); })
4     .Bind([](String const& s) { return "'" + s + "'"); });
```

The first call to Result::Bind takes the int value contained in the Result object, adds one to it, puts that into a new Result object, and returns it. The second call to Result::Bind takes the incremented int value from the new Result object, converts it into a String, and returns a new Result<String> object with it. The third and final call to Result::Bind takes the String object contained in the new Result object, adds quote characters to it, and returns a new Result object with it.

If the Result does not contain a value, then none of these Callables are invoked, and the Result object is only type-converted but retains the original ErrorCode.

The Callables passed to Result::Bind must take a suitable type as a parameter and can return either a ValueType directly (as shown above, and either the same ValueType as before, or a new, different ValueType), or a Result<ValueType>.

3.1.3 Faults inside constructors

Constructors cannot return ara::core::Result objects. Therefore, constructors that may encounter recoverable errors will throw exceptions when they do so. The ARA API uses a technique that is similar to the named constructor idiom to support applications that do not use exception mechanisms. For each constructor that may throw exceptions as part of its defined behavior, a class provides an additional static method as an alternative to create objects. The method has the name Create and does not throw exceptions. It has the same parameters as the constructor.

For example, a class that provides the constructor

```
1   SomeClass::SomeClass(uint8_t i);
```

which may throw exceptions, also provides the static method

```
1   static ara::core::Result<SomeClass> SomeClass::Create(uint8_t i) noexcept
      ;
```

The method Create returns an ara::core::Result object, which either contains the new instance of the class or an error. Retrieving one or the other from the return value is similar to all other functions that return ara::core::Result objects (see Section 2.1.2).

Both the constructor itself and the method Create are available for constructors that may throw exceptions if the toolchain supports exceptions. However, if the toolchain does not support exceptions, only the method Create is available. Calling the constructor will result in a compilation error. Accordingly, code that uses the static method will compile on both kinds of toolchains. Code that uses the constructor will only compile on toolchains that support exceptions.

The static methods for creating instances of a class are only provided for constructors that may throw exceptions as part of their defined behavior. For other constructors, invariably use the constructor.

3.1.4 Non-Recoverable Errors

There can also be situations that are non-recoverable. The AUTOSAR AP defines three such categories in [2]:

- `Violation`
- `Failed Default Allocation`
- `Corruption`

They all lead to the termination of the process in which they occur.

Standardized `Violations` are implemented by the AUTOSAR AP implementation by a call to `ara::core::Abort` and `Non-Standardized Violations` as well as `Failed Default Allocations` may be implemented by a call to `ara::core::Abort`. The application can register abort handlers to do some required tasks before the process is terminated. `Non-Standardized Violations` and `Failed Default Allocations` can alternatively be realized by an AUTOSAR AP implementation by throwing an `Exception` that is not a subclass of `ara::core::Exception`. Such `Exceptions` are converted to a call to `std::terminate` if the API function in which it occurs is `noexcept`. In case they are not, it would be possible for an application to catch such `Exceptions`. This is not advised except for the purpose of calling `ara::core::Abort` because that would rely on the implementation-specific decision of how the handling of the concrete `Violation` is implemented. Catching such an `Exception` could also prevent the process from terminating in a situation in which its correct functioning can not be guaranteed anymore.

`Corruptions` lead to unsuccessful process termination in an implementation-defined way.

3.1.5 Standardized vs Vendor-specific ErrorCodes

AUTOSAR standardizes ErrorCodes for the standardized behavior.

It is possible for an API to return vendor-specific ErrorCodes. These must use their own ErrorDomain (see [SWS_CORE_00092]). These ErrorDomains contain the vendor ID, which enables a unique assignment.

A portable stack-vendor neutral application should consider this and treat these ErrorDomains as general errors (default case).

3.2 Startup and Shutdown

Before making use of ARA the `Application` has to call `ara::core::Initialize`. The general advice is to call `ara::core::Initialize` to initialize the platform right at the entry point of an Adaptive Application. That entry point could be either the standard main function or a user-defined function that is called from main (e.g., to support unit tests for that function). If necessary, actions that do not rely on ARA APIs can be performed prior to calling `ara::core::Initialize`, such as registering a SIGTERM handler or parsing command line arguments. However, only the ARA APIs that are listed in [SWS_CORE_15002] may be used prior to calling `ara::core::Initialize`.

At the end of the Adaptive Application lifecycle, ara::core::Deinitialize needs to be called prior to termination of the Adaptive Application after any ARA resources held by the Adaptive Application (including those that are in use by other threads of the Adaptive Application) have been freed.

Both ara::core::Initialize and ara::core::Deinitialize return a ara::core::Result<void> that needs to be checked because these operations may fail. There is no way for the Adaptive Application to recover from this situation (e.g., no retry is allowed) and the Adaptive Application needs to terminate with an error status (EXIT_FAILURE). The Adaptive Application may perform additional actions prior to termination if necessary. However, only the ARA APIs that are listed in [SWS_CORE_15002] may be used in this case.

The usual entry point function of an Adaptive Application looks like this:

```
1  // Could also be directly defined as main
2  int adaptive_application_main(int argc, char* argv[])
3  {
4      // TODO: Perform setup actions that do not rely on ARA APIs if
5      // necessary
6
7      // Initialize ara::*
8      // ara::core::Initialize() might make use of vendor specific
9      // command line arguments for configuration.
10     // If this is the case ara::core::Initialize() will remove
11     // such arguments from argc and argv.
12     if (!ara::core::Initialize(argc, argv)) {
13         // TODO: Add additional actions here if necessary.
14         // ARA APIs other than those listed in SWS_CORE_15002 cannot
15         // be used here.
16
```

```
17            return EXIT_FAILURE;
18        }
19
20        // All ARA APIs are now initialized and usable
21        ...
22
23        // TODO: Join threads and free any resources from ARA APIs if
24        // necessary.
25
26        if (!ara::core::Deinitialize()) {
27             // TODO: Add additional actions here if necessary.
28             // ARA APIs other than those listed in SWS_CORE_15002 cannot
29             // be used here.
30
31            return EXIT_FAILURE;
32        }
33
34        // TODO: Add additional actions here if necessary.
35        // ARA APIs other than those listed in SWS_CORE_15002 cannot be
36        // used here.
37
38        return EXIT_SUCCESS;
39    }
```

3.3 Reserved symbols

3.3.1 Preprocessor macros

The Adaptive Platform tries to avoid the use of C/C++ preprocessor macros. Such macros start with the prefix ARA. Macros with this prefix should thus not be defined by developers of an Adaptive Application.

3.4 Thread-safety

If the API is thread-safe, then thread-safety should be considered for any implementation of such functions provided by a vendor or user of the API (this affects e.g., callback or implementations of pure virtual functions like skeleton functions). The interface assumes adherence of implementations to the given thread-safety specifications.

3.5 Async signal-safety

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

4 Execution Management

Execution Management is a Functional Cluster responsible for all aspects of system execution management including platform initialization and startup / shutdown of processes.

Execution Management provides multiple dedicated interfaces to the processes as part of the AUTOSAR Runtime for Adaptive (ARA).

4.1 Execution Client

As mentioned in TPS_ManifestSpecification a Modelled Process is an instance of an Executable. On the AUTOSAR Adaptive Platform, a Modelled Process is realized at run-time as an OS process. As Execution Management is responsible for tracking the life cycle of a Modelled Process, a reporting process is expected to notify Execution Management about its own state. This is done by instantiation of an Execution Client (ara::exec::ExecutionClient) and reporting Running Execution State (ara::exec::ExecutionState::kRunning) via the ara::exec::ExecutionClient::ReportExecutionState class method. A process should typically report kRunning as soon as initialization has been completed. Delaying further the report according to service availability (service discovery over Communication Management) for example may introduce non-deterministic delays impacting other processes.

Please also note, that Execution Management is monitoring the start-up time of the process, which is measured until the report of the kRunning. If kRunning is not reported before the configured amount of time, Execution Management will consider this as a failure during the start-up and will terminate the process.

Execution Management will initiate process termination by sending the SIGTERM signal to a process. Handling of the SIGTERM signal should be done via the termination handler, which is defined by ExecutionClient::Create().

```
1  /*
2  * Example: ExecutionClient instantiation, providing Termination Handler
     and Execution State report
3  */
4  bool termination_requested{false};
5
6  auto executionClientResult = ara::exec::ExecutionClient::Create([&]() {
7      termination_requested = true;
8  });
9
10 if (executionClientResult.HasValue()) {
11     ara::exec::ExecutionClient executionClient = std::move(
     executionClientResult).Value();
12
13     // Do some application specific initialization ...
14     auto reportExecutionStateResult = executionClient.
     ReportExecutionState(ara::exec::ExecutionState::kRunning);
```

```
15
16      if (reportExecutionStateResult.HasValue()) {
17          logger.LogInfo() << "Report of running Execution State succeeded"
    ;
18
19          // Until termination has been requested
20          while (!termination_requested) {
21              // Do work ...
22          }
23      } else {
24          logger.LogError() << "Failed to report running Execution State.
    Error code: " << reportExecutionStateResult.Error();
25          // ...
26      }
27  } else {
28      logger.LogError() << "Failed to create ExecutionClient. Error code: "
     << executionClientResult.Error();
29      // ...
30  }
```

4.2 Exit code

On reception of the SIGTERM the process should initiate its own termination procedure. Execution Management will monitor the time needed for process termination. If the process does not terminate before the configured amount of time, this will be considered as a failure during the termination and the process will be forcibly terminated (e.g. via SIGKILL).

A graceful termination of the process is expected to report 0 (EXIT_SUCCESS). Any non 0 exit code are handled as Abnormal Termination.

4.3 State Client

State Management Functional Cluster is responsible for coordinating states of Function Groups to achieve a certain functionality (based on internal decisions, external or platform internal requests).

Function Groups provide the capability to coherently control group of processes. Each Function Group State (belonging to a Function Group) defines which processes shall be started/terminated/restarted. This is done when State Management requests Function Group State change activation from Execution Management.

The states of MachineFG are used to control machine life cycle (shutdown/restart) and therefore processes of platform level, while other Function Group States individually control processes which belong to functionally coherent user-level Applications.

The ara::exec::StateClient class provides multiple methods to State Management to request Function Group state changes. The undefinedStateCallback is a parameter of

the State Client constructor. Execution Management will invoke the undefinedStateCallback in case of unexpected termination of a process outside a state transition. In such a case, the related Function Group and ExecutionError are passed in the form of an ExecutionErrorEvent.

At Machine startup, Execution Management is responsible for self-initiating the transition of MachineFG to Startup state (once per machine life cycle). Assessing the transition success is possible using the ara::exec::StateClient::GetInitialMachineStateTransitionResult class method.

The ara::exec::StateClient::SetState class method allows State Management to request FunctionGroupState transitions. The method is asynchronous. The returned ara::core::Future can be used to check if the request has been successfully performed. The FunctionGroupState parameter has a specific constructor requiring fully qualified short names (Function Group and Function Group State) in form of ara::core::StringView parameters.

In some failure cases (process startup timeout, failed authenticity check or unexpected process termination for example), the FunctionGroupState will be set to the Undefined FunctionGroupState. The ara::exec::StateClient::GetExecutionError class method allows to retrieve the associated execution error (configured in the Execution Manifest) . The ExecutionError can be used for error recovery actions. Please note that the API returns values (ExecutionErrors) only if the Function Group is in an Undefined Function Group State. Otherwise, the API returns an error (meaning "API should not be used for this case").

```
1  /*
2  * Example: StateClient instantiation
3  */
4  auto stateClientResult = ara::exec::StateClient::Create([&logger](const
    ara::exec::ExecutionErrorEvent& event) {
5      logger.LogError() << "Unexpected termination. Function Group: " <<
    event.functionGroup << ", Execution Error: " << event.executionError;
6      /* ... */
7  });
8
9  if (stateClientResult.HasValue()) {
10     ara::exec::StateClient stateClient = std::move(stateClientResult).
    Value();
11     /* ... */
12 } else {
13     logger.LogError() << "Failed to create StateClient. Error code: " <<
    stateClientResult.Error();
14     /* ... */
15 }
16
17 /*
18 * Example: get initial machine state transition result
19 */
20 auto getInitialMachineStateTransitionResult = stateClient.
    GetInitialMachineStateTransitionResult().GetResult();
21
22 if (getInitialMachineStateTransitionResult.HasValue()) {
```

```
23      logger.LogInfo() << "Initial machine state transition succeeded.";
24      /* ... */
25  } else {
26      auto error = getInitialMachineStateTransitionResult.Error();
27
28      if (ara::exec::ExecErrc::kOperationFailed == error) {
29          logger.LogError() << "Initial machine state transition failed.";
30          /* ... */
31      } else if (ara::exec::ExecErrc::kOperationCanceled == error) {
32          logger.LogInfo() << "Initial machine state transition has been
    cancelled.";
33          /* ... */
34      } else if (ara::exec::ExecErrc::kNoCommunication == error) {
35          logger.LogError() << "Failed to get initial machine state
    transition result.";
36          /* ... */
37      } else {
38          logger.LogError() << "General error while getting initial machine
     state transition result.";
39          /* ... */
40      }
41  }
42
43  /*
44  * Example: Perform Function Group State transition
45  */
46  // Instance Specifier from model
47  ara::core::InstanceSpecifier instspec{...}
48
49  // Create the FunctionGroup object using the constructor
50  ara::exec::FunctionGroup functionGroup(instspec);
51
52  auto functionGroupState = ara::exec::FunctionGroupState(functionGroup,
    functionGroupState);
53  auto setStateResult = stateClient.SetState(functionGroupState).GetResult
    ();
54
55  if (setStateResult.HasValue()) {
56      logger.LogInfo() << "Transition to Function Group State succeeded.";
57      /* ... */
58  } else {
59      auto setStateError = setStateResult.Error();
60      logger.LogError() << "Failed to set Function Group State. Error code:
     " << setStateError;
61
62      if (ara::exec::ExecErrc::kInvalidMetaModelIdentifier != setStateError
    ) {
63          auto getExecutionErrorResult = stateClient.GetExecutionError(
    functionGroupState);
64
65          if (getExecutionErrorResult.HasValue()) {
66              ara::exec::ExecutionErrorEvent event = std::move(
    getExecutionErrorResult).Value();
67              logger.LogError() << "Execution Error: " << event.
    executionError;
68              /* ... */
```

```
69            } else {
70                ara::core::ErrorCode errc = getExecutionErrorResult.Error();
71
72                if (errc == ara::exec::ExecErrc::kNoCommunication) {
73                    logger.LogError() << "Failed to get Execution Error.
    Error code: " << static_cast<ara::core::ErrorDomain::CodeType>(ara::exec
    ::ExecErrc::kNoCommunication);
74                    /* ... */
75                }
76            }
77        }
78    }
```

5 State Management

5.1 Interaction with AUTOSAR Adaptive (Platform) Applications

5.1.1 Basic State Management functionality

As State Management is available in two different designs (StateMachine approach or project-specific "hand-written" code), two different approaches for the interfaces are available as well:

For the StateMachine approach a service interface to trigger a state change and another service interface for being notified about the current state of the StateMachine is available per StateMachine. For the project-specific approach no interface to applications is defined, as implementer can specify the needed interface by its own. The defined service interface, as well as the project-specific interface should at least support the following functionality:

- FunctionGroups can be requested to be set to a dedicated state
- (Partial) Networks can be requested to be de- / activated
- The machine can be requested to be shutdown, restarted or suspended to RAM

Some of these functions are critical. Therefor access to those interfaces has to be secured properly by Integrator via Identity and Access Management not to change the internal state of State Management (and therefor the depending effects) accidently.

5.1.2 State Management StateMachine functionality

With the optional StateMachine approach of StateManagement the user is able to configure a certain set of actions to be taken when a StateMachine state is entered. Currently the following actions are possible to configure:

- FunctionGroups can be requested to be set to a dedicated state
- (Partial) Networks can be requested to be set to FullCom or NoCom
- StateMachine(s) can be started or stopped
- Synchronization between actions is possible
- Sleeping between actions is possible
- Coordination of suspend to RAM is possible

Please note that the list of actions might be extended it future AUTOSAR releases.

Beside the actions it is for sure possible to configure the amount of states for a certain StateMachine. To complete StateMachine definition two more tables have to be configured:

- TransitionRequestTable, which contains the possible transitions based on user input triggers (from SMControlApplication) and / or changes in NMHandle states

Transition Request	Current State	Next State
1001	Off	On
1000	On	Off
1002	Recovery	Off
1001	Startup	Off
1000	Suspend	On
1000	Recovery	Off

Figure 5.1: StateMachine TransitionRequestTable

Transition Request	Current State	Next State
Nh1_FullCom	Off	Camera Active
Nh1_NoCom	Camera Active	Off

Figure 5.2: Extended StateMachine TransitionRequestTable

- ErrorRecoveryTable, which contains unconditional transitions, based on ErrorCode, coming from PlatformHealthManagement, ExecutionManagement and / or StateManagement itself.

Error Code	Next State	
11	Recovery	
12	Startup	
111	Recovery	
23	Suspend	
24	Off	On
ANY	Shutdown	

Figure 5.3: StateMachine ErrorTable

In the StateMachine approach, each StateMachine encapsulates the configuration specific to the project it serves. It defines how the Machine shall be started, shut down, suspended, or restarted. Additionally, it governs the behavior of individual Processes based on their assigned FunctionGroups, and may also manage associated NmNetworkHandles. While the integrator is responsible for modeling these configurations within the StateMachines to reflect project requirements, the business logic that determines when a StateMachine shall transition to a particular state - such as initiating a shutdown - resides outside the StateManagement scope and is delegated to the SmControlApplication. This separation of concerns between configuration and con-

trol enables a robust and flexible architecture, allowing business logic to be cleanly implemented within the SmControlApplication while maintaining a clear and reusable StateMachine structure.

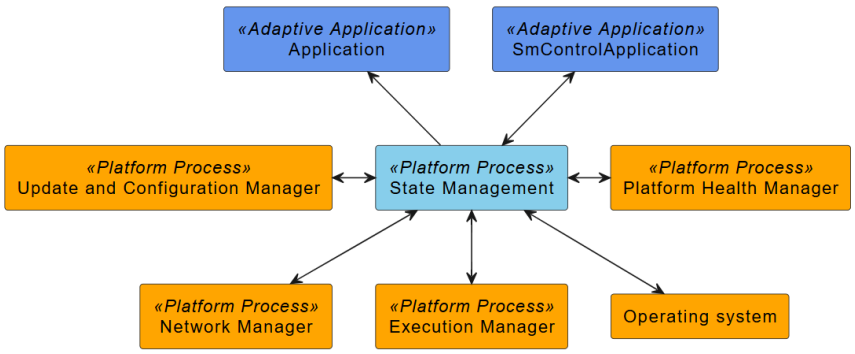


Figure 5.4: StateMachine interactions

To differentiate between the pure customer written StateManagement and the StateMachine approach, which is provided by stack vendor new interfaces were introduced.

- StateMachineService::RequestState, to be used by SMControlApplication to request a certain state
- UpdateAllowedService::UpdateAllowed, which will be used by SMControlApplication to signal StateManagement if under current conditions an update is allowed or not
- StateMachineNotification::CurrentState which is used to inform Adaptive Applications about the current state of a StateMachine

The number of StateMachines, which can be configured by the user is not limited.

6 Persistency

6.1 Overview

Persistency is one of the foundation clusters of the adaptive AUTOSAR platform which provides an API to the application to store and retrieve user data. It supports two different storage mechanisms: Key-Value Storage and File Storage.

Both storage mechanisms might use a file system of the operating system, and in this case rely on this file system to be able to synchronize changes. This has to be ensured by a proper integration of the file system, e.g. by using appropriate mount options.

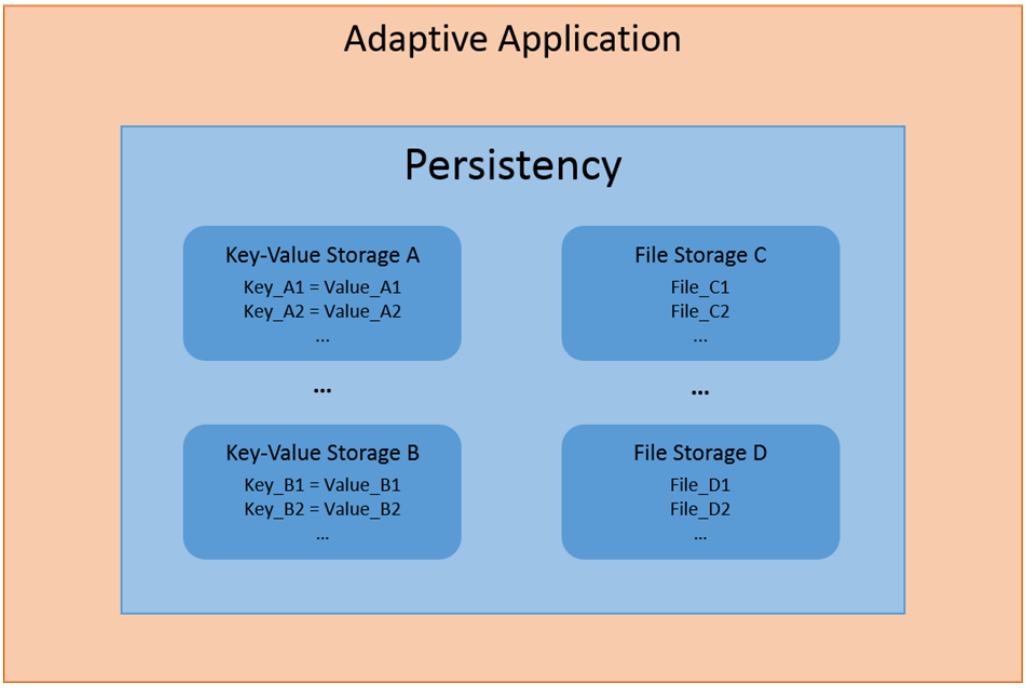


Figure 6.1: Persistency functionality overview

6.1.1 Key-Value Storage

This is a simple key based database that helps the user to store their data in the database.

Key-Value Storages support a simple transactional context that starts when the Key-Value Storage is opened, and is stopped and restarted when SyncToStorage or DiscardPendingChanges is called, and finally stopped when the Key-Value Storage is closed. It is well suited for a medium to large number of not too large data items.

6.1.2 File Storage

This is a simple file based storage where the data is stored in files inside a folder. It supports storing huge amounts of data.

6.1.3 Configuration

An application needs to design Key-Value Storages and File Storages as a port interfaces in order to access the Key-Value Storage and File Storage features. After designing the Persistency port interfaces, further configuration information will be provided during the target configuration stage (e.g. storage location, redundancy CRC, redundancy M out of N, etc.) either in the target configuration section of the manifest or in the APMC of Persistency.

Based on the logging implementation inside Persistency, the cluster can log the run time-related warnings errors and fatal problems.

Note: AUTOSAR_AP_SWS_Persistency does not specify the above details as that is implementation specific.

6.2 Redundancy feature

This feature ensures persistent data safety for both Key-Value Storages and File Storages. There are three possible ways to ensure data safety (integrity).

1. CRC
2. Hashes
3. M out of N approach

It is a configurable parameter. Based on the project need, either one or a combination of them can be configured. With respect to CRC, all the AUTOSAR CRCs are supported to configure. In addition to the detection of integrity M out of N approach helps to recover the data if there are sufficient redundant copies available.

Persistency also allows the application to install a callback that will report any problems detected in any of the storages, even if the problem could be corrected because the redundancy was sufficient. This callback can be used in safety critical application, or when the application wants to monitor the health of the hardware.

Recover Storage: It is part of the Redundancy Feature. If integrity checks fail for a Key-Value Storage or a File Storage or a file or a key-value pair, the user can use RecoverKeyValueStorage/RecoverFileStorage/RecoverFile/RecoverKey APIs to recover the data based on a best effort recovery mechanism. Upon invocation of the Recover APIs, a valid Key-Value Storage/File Storage/file/key-value pair could be retrieved which might have lost some key-value pairs/files as it is a best effort recovery mechanism.

6.3 Reset Storage

This feature resets a Key-Value Storage/File Storage/file/key-value pair to the initial state, containing only key-value pairs/files which were deployed from the manifest, with their initial values and it will fail with a kResourceBusyError when the Key-Value Storage/FileStorage/file/key-value pair is currently open/accessed and with a kInitValueNotAvailableError when target configuration does not define an initial content for the key-value pair/file.

Reset APIs:

- ResetPersistency(): Reset all Key-Value Storages and File Storages to the initial state with key-value pairs and files deployed from manifest
- ResetKeyValueStorage(): Reset Key-Value Storage to the initial state with key-value pairs deployed from manifest
- ResetAllFiles(): Reset the whole File Storage, including all files from the deployed content from manifest
- ResetFile(): Reset a single file to its initial content which was deployed from the manifest
- ResetKey(): Reset a single Key-Value Pair to its initial content which was deployed from the manifest

6.4 Update and Removal of Persistent Data

Persistency will perform the specific action (update/rollback/remove) on the persistent data of the application, either when UpdatePersistency() is called, or when the storages of Persistency are opened.

All Persistency files and Key-Value Storages will be updated after a new manifest was installed, which can also happen separately from the application at run time. The application can poll for such changes using CheckForManifestUpdate().

An application may also register callback functions (RegisterDataUpdateIndication(), RegisterApplicationDataUpdateCallback()) that are called after the update of any Key-Value Storage and File Storage. These callback functions will be called from the context of UpdatePersistency(), OpenKeyValueStorage(), or OpenFileStorage().

6.5 Example usage of the Key-Value Storage API

The configuration needs to map the InstanceSpecifier of the Key-Value Storage to a specific location during target configuration of the Key-Value Storage.

Write operation sequence

1. Open the Key-Value Storage with the ara::core::InstanceSpecifier

```
ara::core::Result<SharedHandle<KeyValueStorage>> kvsRes = ara::per
::OpenKeyValueStorage(kDatabaseNameIS);
```

   Parse the ara::core::Result to check success or error in case of failure of the OpenKeyValueStorage operation.

2. Parse the KeyValueStorage object (kvs) from ara::core::Result

```
SharedHandle<KeyValueStorage> db = std::move(kvsRes).Value();
```

3. Invoke the SetValue with key and value that needs to be persisted in the Key-Value Storage

```
ara::core::Result<void> result = db->SetValue(kDoubleKeyName,
DoubleValue);
```

   Parse the result to check the status of the write operation

   Hint: In order to effectively use the underlying storage device it is designed that all the user requests are stored intermediately in the RAM and the data will be persisted to file system only after invoking the below sync call. Hence it is suggested that after opening the Key-Value Storage, perform multiple SetValue() operations then persist the data finally via sync call.

4. Invoke the below API to persist data to the nonvolatile storage (Flash/Hard disk)

```
ara::core::Result<void> result = db->SyncToStorage();
```

   Parse the result to check the status of the sync operation

5. There is a possibility that a user can go to last sync state by calling the API DiscardPendingChanges() which will discard the transaction of syncing the locally stored key value pairs with the underlying data base.

```
ara::core::Result<void> result = db->DiscardPendingChanges();
```

Read operation sequence:

1. Open the Key-Value Storage with the instance specifier

```
ara::core::Result<SharedHandle<KeyValueStorage>> kvsRes = ara::per
::OpenKeyValueStorage(kDatabaseNameIS);
```

   This returns ara::core::Result which contains kvsobject or error in case of failure

2. Parse the Kvs object from ara::core::Result

```
SharedHandle<KeyValueStorage> db = std::move(kvsRes).Value();
```

3. Invoke the GetValue with key and value that needs to be retrieved from the Key-Value Storage

```
ara::core::Result<ara::core::String> first_value = db->GetValue<
String>(kStringKeyName);
```

Parse the result to check the status of the read operation and get the value assigned to a key

6.6 Example usage of File Storage API

Write operation

1. Open File Storage with the short name of the PortPrototype

```
ara::core::Result<SharedHandle<FileStorage>> fsRes =
OpenFileStorage(kFolderNameIS);
```

Parse the ara::core::Result to check success or error in case of failure

2. Parse the File Storage object from ara::core::Result

```
SharedHandle<FileStorage> fs = std::move(fsRes).Value();
```

3. Invoke the OpenFileWriteOnly with the file name which is the short name of the PortPrototype to get the WriteAccessor object

```
ara::core::Result<UniqueHandle<ReadWriteAccessor>> fileRes = fs->
OpenFileWriteOnly(kFileName);
```

4. Perform the formatted writing via overloading operator

```
(*(std::move(fileRes).Value())) << "Overwriting!";
```

Read operation

1. Open File Storage with the short name of the PortPrototype

```
ara::core::Result<SharedHandle<FileStorage>> fsRes =
OpenFileStorage(kFolderNameIS);
```

Parse the ara::core::Result to check success or error in case of failure

2. Parse the FileStorage object from ara::core::Result

```
SharedHandle<FileStorage> fs = std::move(fsRes).Value();
```

3. Invoke the OpenFileReadWrite() with the file name to get the ReadWriteAccessor object

```
ara::core::Result<UniqueHandle<ReadWriteAccessor>> fileRes = fs->
OpenFileReadWrite(kFileName);
```

4. Perform the read operation

```
UniqueHandle<ReadWriteAccessor> rwa = std::move(fileRes).Value();
ara::core::Result<ara::core::String> = rwa->ReadLine();
```

Read the value in the buffer until default delimiter.

7 Platform Health Management

7.1 Supervision and Checkpoints

Platform Health Management (PHM) supervises software entities, including application and platform software. The software developers provide a modelled configuration (manifest) that defines the correct order and/or timing constraints of the checkpoints. Software components report checkpoints when specific points of the code are reached, and PHM verifies their sequence and timing. If designed order or timing constraints are violated, PHM informs State Management to initiate recovery actions.

PHM also periodically services the watchdog; if failures are detected in Execution Management or State Management, PHM stops servicing the watchdog and triggers watchdog reset.

Usage of checkpoints is depending on the software design and the safety requirements. Whenever the correct order of execution or the timing constraints between specific points of code must be ensured, calls of the ReportCheckpoint API can be placed in the code accordingly and the correct order and timing must be modelled.

API:

```
 void ara::phm::SupervisedEntity::ReportCheckpoint (EnumT checkpointId)
noexcept;
```

7.2 Recovery actions

Recovery actions can be defined and assigned to specific supervisions.Platform Health Management calls the recovery handler of recovery actions in case of violations. Recovery handlers must be implemented project specific in State Management. If the Recovery Handler fails to handle the supervision failure in-time (recovery time out), then refreshing of the Watchdog will be stopped.

API:

```
Virtual ara::core::Future<void> ara::phm::RecoveryAction::RecoveryHandler(
    const ara::exec::ExecutionErrorEvent&executionError,TypeOfSupervision
    supervision)noexcept=0;
```

The recovery handler invocation needs to be enabled before by a call of RecoveryAction::Offer.

API:

```
ara::core::Result < void > ara::phm::RecoveryAction::Offer() noexcept;
```

The recovery handler invocation can be disabled by a call of RecoveryAction::StopOffer.

API:

```
void ara::phm::RecoveryAction::StopOffer() noexcept;
```

7.3 Watchdog handling

The Platform Health Management is the only Functional Cluster with an interface to the hardware watchdog. It periodically serves the hardware watchdog with configured cycle time if there are no failure with the supervision of software entities with the interface WatchdogInterface::AliveNotification.

API:

```
void apext::phm::WatchdogInterface::AliveNotification();
```

If the supervision of State Management or Execution Management failed or State Management cannot handle the supervision failure reported by PHM with the interface RecoveryAction::RecoveryHandler, PHM stops calling WatchdogInterface::AliveNotification and call WatchdogInterface::FireWatchdogReaction.

API:

```
void apext::phm::WatchdogInterface::FireWatchdogReaction();
```

7.4 Shutdown functionality

In the sense of a safe system, the integrator shall ensure that the applications are configured to be supervised by Platform Health Management are terminated before triggering shut down of the Platform Health Management. Please refer to [3].

8 Update and Configuration Management

8.1 Port configuration

UCM can be configured to share a network port with other applications and does not necessarily need a dedicated port.

8.2 Method deployments

Most UCM methods have no or small parameters or return values that fit into an unsegmented SOMEIP message (~1300 byte payload) and could therefore be deployed via UDP. Some have however potentially large parameters or return values and therefore need to be deployed via TCP or SOMEIP/TP.

UCM interface follows a request response design pattern and clients as well as service might end up in blocked state if message loss occurs on request or response message. SOMEIP stacks usually open one dedicated socket which stays open for as long as services are consumed by the client. Therefore, it is best to simply deploy all messages and also fields via TCP if possible to avoid these possible deadlocks. If the use of TCP is not possible, then refer to chapter 8.3 for choosing block sizes for SOMEIP/TP deployments and handle message loss by project specific application timeouts and retries. Alternatively, the vehicle network has to guarantee that UDP messages are always delivered to the destination, e.g. via special switches and ensuring load thresholds are not exceeded.

The following table shows which methods can be deployed via UDP because they fit inside the UDP MTU and which need to be deployed via SOMEIP/TP because they could exceed it with (x). As explained above, it is recommended to use a TCP deployment for all methods because the overhead is minimal, marked with (X).

Method	UDP	SOMEIP/TP	TCP
Activate	x		X
Cancel	x		X
DeleteTransfer	x		X
Finish	x		X
GetHistory		x	X
GetId	x		X
GetProgress	x		X
GetSwClusterChangeInfo		x	X
GetSwClusterInfo		x	X
GetSwClusterManifestInfo		x	X
GetSwPackages		x	X
ProcessSwPackage	x		X
RegisterSoftwarePackage	x		X
Resume	x		X

▽

△

RevertProcessedSwPackages	x		X
Rollback	x		X
Suspend	x		X
TransferData		x	X
TransferExit	x		X
TransferStart	x		X

A similar consideration can be done for the V-UCM master interface methods.

Method	UDP	SOMEIP/TP	TCP
AllowCampaign	x		X
Approve	x		X
CancelCampaign	x		X
DeleteTransfer	x		X
GetCampaignHistory		x	X
GetSwClusterInfo		x	X
GetSwPackageDescription		x	X
GetSwPackages		x	X
GetVehiclePackageDescription		x	X
PublishSafetyState	x		X
ReportUnsupportedSafetyConditions	x		X
SwPackageInventory		x	X
TransferData		x	X
TransferExit	x		X
TransferStart	x		X
TransferVehiclePackage	x		X
VehicleCheck	x		X

8.3 Data transfer, message loss and block sizes

UCM needs to transfer potentially large amounts of data over the ara::com service interface. The configuration of the TransferData method is crucial for the performance of the data transmission.

Non-trivial software packages exceed the segmentation size of a single SOMEIP message transmitted over UDP. UCM can set the maximum BlockSize that may be used by the client so that the message will not be segmented ($\sim$1440 byte). However, this will lead to poor transfer performance because each block has to be acknowledged in the application context before sending the next block.

If the platform supports method calls via TCP or SOMEIP/TP segmentation the BlockSize setting can be increased to reduce protocol header overhead and move the segmentation effort into lower layers.

In case of SOMEIP/TP the integrator has to consider that a message loss will lead to the loss of the complete SOMEIP message. The following formula shows how to calcu-

late the total probability of losing the segmented message $p_{\text{SOMEIP/TP}}$ from the probability of a single UDP message loss p_{UDP}, the maximum transmission unit MTU_{Eth}, the individual layers header sizes Header_i and the BlockSize. The individual loss probability is multiplied by the number of messages sent to get the total loss probability. Note: For IPv6 the header size is 40 byte, so the denominator would be 1440 instead of 1460.

$$p_{\text{SOMEIP/TP}} = p_{\text{UDP}} \left\lceil \frac{\text{TransferID}+\text{BlockSize}+\text{BlockCounter}}{\text{MTU}_{\text{Eth}}-\text{Header}_{\text{IP}}-\text{Header}_{\text{UDP}}-\text{Header}_{\text{SOMEIP/TP}}} \right\rceil$$

$$p_{\text{SOMEIP/TP}} = p_{\text{UDP}} \left\lceil \frac{8+\text{BlockSize}+8}{1500-20-8-12} \right\rceil = p_{\text{UDP}} \left\lceil \frac{16+\text{BlockSize}}{1460} \right\rceil$$

In case of TCP, message loss is handled by its retransmission features, so loss of an individual TCP message does not break the SOMEIP message. The block size is virtually unlimited, so you could transfer the complete software package in a single TransferData call. In addition to this, the segmentation is handled in kernel space with highly optimized implementations. However, establishing the connection in a three-way handshake and keeping the connection context on both sides consumes more resources compared to SOMEIP/TP, but is only done once per (IP address, TCP port) pair.

The following table shows a summary of pros and cons of the options.

	UDP	**SOMEIP/TP**	**TCP**
Benefits	Simple	No handshake required	Segmentation in kernel space Reliable message delivery
Limitations	Max blocksize ~1400 byte Segmentation in application layer	Loss of one UDP message causes loss of the complete SOMEIP message Segmentation in SOMEIP layer	Handshake required, once per (IP address, TCP port) pair
	Unreliable message delivery can cause deadlocks Application level timeout or special network equipment needed		

9 Raw Data Streams

The Raw Data Stream Functional Cluster provides standalone Communication APIs for processing raw binary data streams for communication with an external ECU, e.g. a sensor in an ADAS system or to support audio/video streaming. The following approaches are supported:

- Raw Data Streaming over Ethernet using IP based protocols (network layer), where a byte stream (TCP) bidirectional or datagram stream (UDP) unidirectional between a client and server is used
- Raw Data Streaming using IEEE1722 protocol (data link layer), where a datagram stream unidirectional from a producer and to an consumer via an IEEE1722 stream is used

9.1 Raw Data Streaming over Ethernet using IP based protocols (network layer)

9.1.1 Raw Data Streaming Interface

9.1.1.1 Introduction

The AUTOSAR Adaptive Platform Communication Management is based on Service Oriented communication. This is good for implementing platform independent and dynamic applications with a service-oriented design.

For ADAS applications, it is important to be able to transfer raw binary data streams over Ethernet efficiently between applications and sensors, where service oriented communication (e.g. SOME/IP, DDS) either creates unnecessary overhead for efficient communication, or the sensors do not even have the possibility to send anything but raw binary data.

The Raw Data Binary Stream API using IP based protocols provides a way to send and receive Raw Binary Data Streams, which are sequences of bytes, without any data type. They enable efficient communication with external sensors in a vehicle (e.g. sensor delivers video and map data in "Raw data" format). The communication is performed over a network using network layer sockets.

The Raw Data Streaming API is static, i.e. its is not generated. It is located in namespace `ara::rds`.

9.1.1.2 Functional description

The Raw Data Binary Stream API using IP based protocols can be used in both the client or the server side. The functionality of both client and server allow to send and receive. The only difference is that the server can wait for connections but cannot

actively connect to a client. On the other side, the client can connect to a server (that is already waiting for connections) but the client cannot wait for connections.

The usage of the Raw Data Binary Stream API using IP based protocols from AUTOSAR Adaptive Platform must follow this sequence:

- As client
  1. Connect: Establishes connection to sensor (not needed for UDP)
  2. ReadData/WriteData: Receives or sends data
  3. Shutdown: Connection is closed.
- As server
  1. WaitForConnection: Waits for incoming connections from clients (not needed for UDP)
  2. ReadData/WriteData: Receives or sends data
  3. Shutdown: Connection is closed and stops waiting for connections.

9.1.1.3 Class and Model

9.1.1.3.1 Class and signatures

The namespace `ara::rds` defines a RawDataStream class for reading and writing binary data streams over a network connection using network layer sockets. The client side is an object of the class `ara::rds::RawDataStreamClient` and the server side is `ara::rds::RawDataStreamServer`.

9.1.1.3.1.1 Constructor

The named constructor takes as input the instance specifier qualifying the network binding and parameters for the instance.

```
static ara::core::Result<RawDataStreamClient>
       Create (const ara::core::InstanceSpecifier& instance);

static ara::core::Result<RawDataStreamServer>
       Create (const ara::core::InstanceSpecifier& instance);
```

9.1.1.3.1.2 Destructor

Destructor of RawDataStream using IP based protocols. If the connection is still open, it will be shut down before destroying the RawDataStream object.Destructor of Raw-

DataStream using IP based protocols. If the connection is still open, it will be shut down before destroying the RawDataStream object.

```
~RawDataStreamClient();
~RawDataStreamServer();
```

9.1.1.3.2 Manifest Model

The manifest defines the parameters of the Raw Data Stream deployment on the network.

The `RawDataStreamMapping` defines the actual transport that raw data uses in the sub-classes of `EthernetRawDataStreamMapping`. It also defines which local- and remote network endpoints (IP addresses) and ports to use for the communication, and if unicast or multicast is used.

In principle, Raw Data Streaming using IP based protocols could use any IP based protocol, but currently only TCP and UDP are supported.

The local IP address is defined in the attribute `communicationConnector` (type `EthernetCommunicationConnector`), and the protocol and port is defined in the following attributes of the sub-class `EthernetRawDataStreamMapping` with type `PositiveInteger`:

- `localTcpPort`
- `localUdpPort`

At least one of the two previous attributes has to be defined. The `socketOption` attribute allows to specify non-formal network layer socket options that might only be valid for specific platforms. This is defined as an array of `string`s and the possible values are platform and vendor specific.

Remote credentials for the different use cases are defined in attributes `RawDataStreamUdpCredentials` and `RawDataStreamUdpTcpCredentials`. See TPS Manifest [4] for details.

The `EthernetRawDataStreamMapping` also has an attribute regarding security, where TLS secure communication properties for the Raw Data Stream connection can be defined:

- `tlsSecureComProps`

9.1.1.4 Methods of class RawDataStream

Detailed information about the methods of `ara::rds::RawDataStream` can be found in chapter `API Specification` for Raw Data Streaming using IP based protocols (network layer) of [5].

9.1.1.4.1 Timeout parameter

All Connect/WaitForConnection/Read/Write methods of RawDataStream clients and servers have an optional input parameter for the timeout. This argument defines the timeout of the method in milliseconds. The type is `std::chrono::milliseconds`.

If timeout is 0 or not specified the operation will block until it returns.

If timeout is specified is > 0 the method call will return a timeout error if the time to perform the operation exceeds the timeout limit.

9.1.1.4.2 Methods

The API methods are synchronous, so they will block until the method returns or until timeout is reached.

9.1.1.4.2.1 WaitForConnection

This method is available only in the server side of the Raw Data Stream.

The server side of the Raw Data Stream is ready to be connected from a client. No connection from clients can be established until this method is called in the server. It is only used if TCP is used. For UDP this operation will do nothing.

9.1.1.4.2.2 Connect

This method is available only in the client side of the Raw Data Stream.

This method initializes the socket and establishes a connection to the TCP server. In the case of UDP, no connection needs to be established, and the operation will do nothing. Incoming and outgoing packets are restricted to the specified local and remote addresses.

The socket configurations are specified in the manifest which is accessed through the InstanceSpecifier provided in the constructor.

```
ara::core:Result<void> Connect();
ara::core:Result<void> Connect(std::chrono::milliseconds timeout);
```

9.1.1.4.2.3 Shutdown

This method shuts down communication. It is available from both client and server sides of the Raw Data Stream.

```
ara::core:Result<void> Shutdown();
```

9.1.1.4.2.4 ReadData

This method reads bytes from the socket connection. The maximum number of bytes to read is provided with the parameter length. The timeout parameter is optional.

```
ara::core::Result<ReadDataResult> ReadData(size_t length);

ara::core::Result<ReadDataResult> ReadData(
        size_t length,
        std::chrono::milliseconds timeout);
```

If the operation worked, it returns a struct with a pointer to the memory containing the read data and the actual number of read bytes.

```
struct ReadDataResult{
        std::unique_ptr<ara::core::Byte[]> data;
        size_t numberOfBytes;
}
```

In case of an error it returns an `ara::core::ErrorCode` :

- Stream Not Connected: If the connection is not yet established (TCP only).
- Interrupted By Signal: The operation was interrupted by the system.
- Communication Timeout: No data was read until the timeout expiration.

9.1.1.4.2.5 WriteData

This method writes bytes to the socket connection. The data is provided as a buffer with the data parameter. The number of bytes to write is provided in the length parameter. An optional timeout parameter can also be used.

```
ara::core:Result<size_t> WriteData(
        std::unique_ptr<ara::core::Byte[]> data,
        size_t length);

ara::core:Result<size_t> WriteData(
        std::unique_ptr<ara::core::Byte[]> data,
        size_t length,
        std::chrono::milliseconds timeout);
```

If the operation worked, it will return the actual number of bytes written. In case of an error, it will return a `ara::core::ErrorCode`:

- Stream Not Connected: If the connection is not yet established (TCP only).
- Interrupted By Signal: The operation was interrupted by the system.
- Communication Timeout: No data was written until the timeout expiration.

9.1.1.5 Security

Raw Data Stream communication can be transported using TCP and UDP. Therefore different security mechanisms have to be available to secure the stream communication. Currently the security protocols TLS, DTLS, IPSec and MACsec are available.

Access control to Raw Data Streams can also be defined by the IAM.

All security functions are configurable in the deployment and mapping model of Raw Data Streaming Interface.

If sensor data must fulfill security requirements, security extensions have to be used.

9.1.2 Usage of RawDataStreaming

This chapter describes how RawDataStreams using IP based protocols can be used in an AUTOSAR Adaptive Platform application.

RawDataStreaming currently supports four use cases regarding configuration of unicast/multicast and UDP/TCP connections. These use cases are described in chapter "Raw Data Streaming using IP based protocols (network layer)" of [5].

The most common use case is "'1:1 TCP unicast"', so that is used as example in this tutorial.

9.1.2.1 Sequence diagrams

The diagram 9.1 shows the sequence when using the Raw Data Streaming API on the client side.

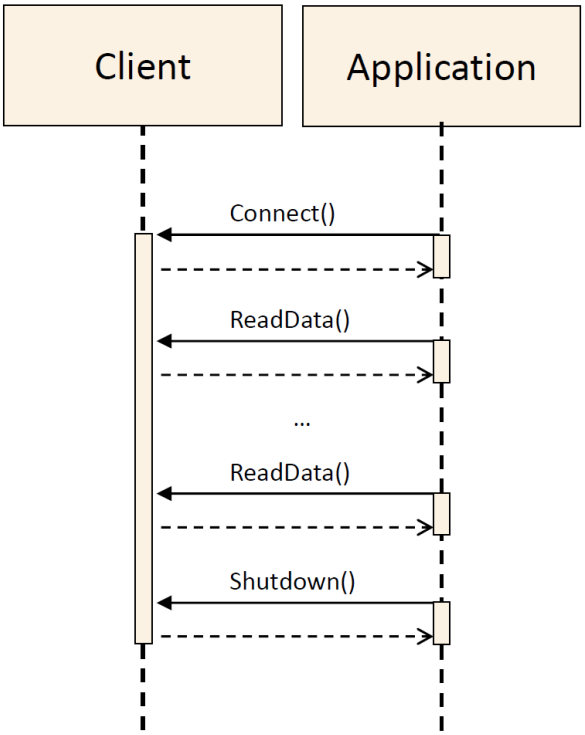


Figure 9.1: Client sequence diagram

The diagram 9.2 shows the sequence when using the Raw Data Streaming API on the server side.

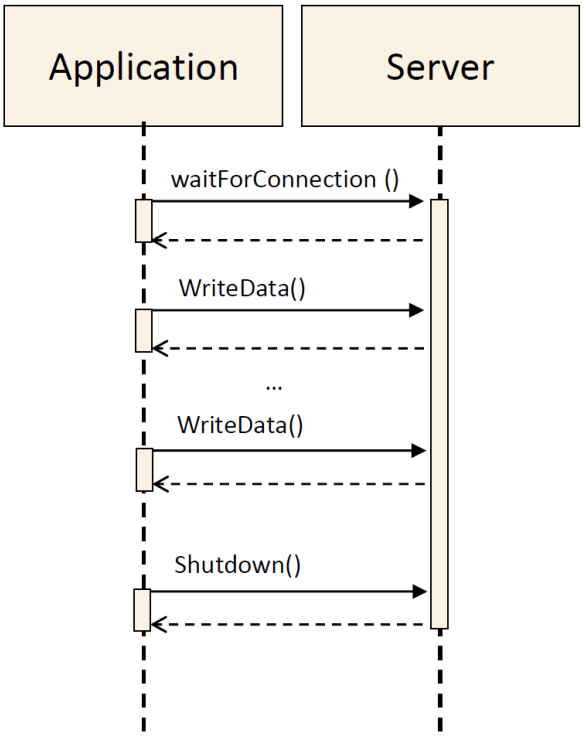


Figure 9.2: Client sequence diagram

Note that the sequences with a client that sends data and a server that reads data are also valid.

9.1.2.2 Usage

Since the Raw Data Streaming provides an API it is required to have the instances of the RawDataStreamServer or RawDataStreamClient and call the methods according to the sequences described in 9.1.2.1

9.1.2.2.1 Example of usage as server

The code 9.1 shows how to use the RawDataStreamServer for sending and receiving data.

```
// NOTE! For simplicity the example does not use ara::core::Result.

#include "ara/core/instance_specifier.h"
#include "raw_data_stream.h"
int main() {
    size_t rval;
    ara::rds::ReadDataResult result;

    // Instance Specifier from model
    ara::core::InstanceSpecifier instspec
    {...}

    // Create RawDataStream Server instance
    ara::core::Result<ara::rds::RawDataStreamServer>
        server_result{ara::rds::RawDataStreamServer::Create(instspec)};
    if (server_result.HasValue()) {
        ara::rds::RawDataStreamServer server{server_result.Value());
    }else
    {...};

    // Wait for incoming connections
    server.WaitForConnection();

    // Read data from the RawData stream in chunks of 10 bytes
    do{
        result = server.ReadData(10);
        rval = result.numberOfBytes;
        if (rval > 0) {
            // assumes the data is printable
            std::cout << "-->" << result.data.get() << std::endl;
        }
    } while (rval > 0);

    // Write data to the RawData stream in chunks of 16 bytes
    int i=0;
    do{
        std::unique_ptr<uint8_t> write_buf (new uint8_t[1024] \{....});
        rval = server.WriteData(std::move(write_buf), 16);
```

```
        ++i;
    }while (i<1000);

    // Shutdown RawDataStream connection
    server.Shutdown(); return 0;
}
```

Listing 9.1: Example of usage as server

9.1.2.2.2 Example of usage as client

The code 9.2 shows how to use the RawDataStreamClient for sending and receiving data.

```
// NOTE! For simplicity the example does not use ara::core::Result.

#include "ara/core/instance_specifier.h"
#include "raw_data_stream.h"
int main() {
    size_t rval;
    ara::rds::ReadDataResult result;

    // Instance Specifier from model
    ara::core::InstanceSpecifier instspec
    {...}

    // Create a RawDataStreamClient instance
    ara::core::Result<ara::rds::RawDataStreamClient>
        client_result{ara::rds::RawDataStreamClient::Create(instspec)};
    if (client_result.HasValue()) {
        ara::rds::RawDataStreamClient client{client_result.Value());
    }else
    {...};

    // Connect to RawDataStream Server
    client.Connect();

    // Write data to RawData stream in chunks of 40 bytes
    int i=0;
    do {
        std::unique_ptr<uint8_t> write_buf (new uint8_t[1024]{.....});
        rval = client.WriteData(std::move(write_buf), 40);
        ++i;
    } while (i<1000);

    // Read data from the RawData stream in chunks of 4 bytes
    do {
        result = client.ReadData(4);
        rval = result.numberOfBytes;
        if (rval > 0){
            // assumes the data is printable
            std::cout << "-->" << result.data.get() << std::endl;
        }
    } while (rval > 0);
```

```
41
42      // Shutdown RawDataStream connection
43      client.Shutdown(); return 0;
44  }
```

Listing 9.2: Example of usage as client

9.2 Raw Data Streaming using IEEE1722 protocol (data link layer)

9.2.1 Raw Data Streaming Interface

9.2.1.1 Introduction

The Adaptive Platform Communication Management is based on Service Oriented communication. This is good for implementing platform independent and dynamic applications with a service-oriented design.

For audio/video applications, it is important to be able to transfer raw binary data streams over Ethernet efficiently between applications and sensors, where service oriented communication (e.g. SOME/IP, DDS) either creates unnecessary overhead for efficient communication, or the sensors do not even have the possibility to send anything but raw binary data.

The Raw Data Binary Stream API using IEEE1722 protocol provides a way to send and receive Raw Binary Data Streams. Raw Binary Data Streams are sequences of datagrams. They enable efficient communication with external devices in a vehicle (e.g. an audio device, which produce an audio stream and transfer audio data in an IEEE1722 protocol defined format). The communication is performed via IEEE1722 streams over a network using data link layer sockets. The IEEE1722 protocol defines an AVTP-timestamp (a.k.a presentation time), which is used to support time sensitive use cases, where data is processed at the same point in time (synchronized) and with aligned processing rate (synchronized). Further details of the IEEE1722 protocol are available in [6, IEEE1722].

The Raw Data Streaming API used for streaming via the IEEE1722 protocol is static, i.e. its is not generated. It is located in namespace `ara::rds`.

9.2.1.2 Functional description

The Raw Data Binary Stream APIs used for streaming via the IEEE1722 protocol, implements functionality for applications which act as an IEEE1722 stream consumer to consume datagrams from an IEEE1722 stream, and applications which act as a IEEE1722 stream producer application to write (produce) datagrams to an IEEE1722 stream. IEEE1722 streams are identified with a system-wide unique stream id. An IEEE1722 stream with a dedicated stream id represents a communication channel where IEEE1722 stream consumer and producer are virtually connected to, since the

IEEE1722 protocol is an unidirectional and connection-less communication protocol. IEEE1722 streams are statically configured.

The API provides functionality for IEEE1722 stream consumers to read data (a stream of datagrams) from a IEEE1722 stream and for IEEE1722 stream producer to write data (a stream of datagrams) to an IEEE1722 steam. The API provides for IEEE1722 stream consumer and IEEE1722 stream producer to connect locally to a communication channel and to shutdown an established local communication connection. IEEE1722 stream consumer and IEEE1722 stream producer can be configured by an integrator by applying deployment information, containing e.g. stream id and destination MAC address.

The usage of the Raw Data Binary Stream API using IEEE1722 protocol from Adaptive Platform shall follow this sequence:

- As IEEE1722 stream consumer
  1. Connect: Establishes a local connection to the data link layer, to be able to read (consume) datagrams from an IEEE1722 stream
  2. ReadData: Requests to read (consume) received datagrams
  3. Shutdown: Disconnect a local connection from the data link layer
- As IEEE1722 stream producer
  1. Connect: Establishes a local connection to the data link layer to be able to write (produce) datagrams to an IEEE1722 stream
  2. WriteData: Requests to write (produce) datagrams, which are transmitted via an IEEE1722 stream
  3. Shutdown: Disconnect a local connection from the data link layer.

9.2.1.3 Class and Model

9.2.1.3.1 Class and signatures

The namespace `ara::rds` defines a RawDataStreamConsumer class for reading binary datagrams received via an IEEE1722 stream over a network connection using data link layer sockets. The namespace `ara::rds` defines a RawDataStreamProducer class for writing binary datagrams transmitted via an IEEE1722 stream over a network connection using data link layer sockets. The consumer side is an object of the class `ara::rds::IEEE1722RawDataStreamConsumer` and the producer side is an object of `ara::rds::IEEE1722RawDataStreamProducer`.

9.2.1.3.1.1 Constructor

The standard constructor or extended constructor is not used for the RawDataStream using IEEE1722 protocol. IEEE1722RawDataStreamConsumer and IEEE1722RawDataStreamProducer provide both a special member function "Create" to create an instance. This member function uses as input the instance specifier qualifying the network binding as instance parameter. Please note, the "Create" members are declared as static member function and therefore can be called without having and instance of the class.

```
IEEE1722RawDataStreamConsumer.Create(
    const ara::com::InstanceSpecifier& instance);
IEEE1722RawDataStreamProducer.Create(
    const ara::com::InstanceSpecifier& instance);
```

9.2.1.3.2 Destructor

Destructor of RawDataStream using IEEE1722 protocol. If the local connection to the data link layer is still open, it will be shut down before destroying the IEEE1722RawDataStreamConsumer / IEEE1722RawDataStreamProducer object.

```
~IEEE1722RawDataStreamConsumer();
~IEEE1722RawDataStreamProducer();
```

9.2.1.3.3 Manifest Model

The manifest defines the parameters of the Raw Data Stream deployment on the network.

The `RawDataStreamMapping` defines the actual transport that raw data uses in the sub-classes of `EthernetMacRawDataStreamMapping`. It also defines which IEEE1722 stream id (concatenation of `IEEE1722TpConnection.macAddress-StreamId` and `IEEE1722TpConnection.uniqueStreamId`) is used and the used IEEE1722 stream subtype (e.g. AAF (audio stream)).

In principle, RawDataStream using IEEE1722 protocol could use any IEEE1722 stream subtype defined in [6, IEEE1722]), but currently only 61883_IIDC, AAF, CRF, RVF, TSCF and NTSCF are supported. Please note, IEEE1722 subtype TSCF and NTSCF carry IEEE1722 encapsulated bus frames as ACF messages in the payload. Currently only CAN and LIN frames are supported by AUTOSAR to be transfered as ACF-messages.

The `socketOption` attribute allows to specify non-formal data link layer socket options that might only be valid for specific platforms. This is defined as an array of `string`s and the possible values are platform and vendor specific.

9.2.1.4 Methods of class IEEE1722RawDataStreamConsumer/Producer

Detailed information about the methods of `ara::rds::RawDataStream` can be found in chapter for Raw Data Streaming using IEEE1722 protocol (data link layer) of [5].

9.2.1.4.1 Methods

The API methods are synchronous, so they will block until the method returns.

9.2.1.4.1.1 Connect

This method is available on IEEE1722 stream consumer as well as on IEEE1722 stream producer side of the Raw Data Stream using IEEE1722 protocol.

This method initializes the data link socket and establishes a local connection to data link layer. Incoming and outgoing packets are restricted to the specified IEEE stream id and the used IEEE1722 stream sub type.

The data link socket configurations are specified in the manifest which is accessed through the InstanceSpecifier provided in the constructor.

```
ara::core:Result<void> Connect();
```

In case of an error, it will return a `ara::core::ErrorCode`:

- kConnectionRefused: The target address was not listening for connections or refused the connection request.
- kStreamAlreadyConnected: The specified connection is already connected.
- kInterruptedBySignal: System error. Operation interrupted by system (POSIX EINTR).
- kConnectionCreationFailed: Permission to create a connection is denied. (POSIX EACCES)
- kGrantEnforcementError: Request was refused by Grant enforcement layer.

Please note: kConnectionRefused is used to indicated a local issue to set up a connection to data link layer if using with the IEEE1722 protocol.

9.2.1.4.1.2 Shutdown

This method is available on IEEE1722 stream consumer as well as on IEEE1722 stream producer side.

The method disconnects the instance of an IEEE1722RawDataStreamConsumer or IEEE1722RawDataStreamProducer from a data link socket connection.

```
ara::core:Result<void> Shutdown();
```

In case of an error, it will return a `ara::core::ErrorCode`:

- kConnectionRefused: The target address was not listening for connections or refused the connection request.
- kStreamAlreadyConnected: The specified connection is already connected.
- kInterruptedBySignal: System error. Operation interrupted by system (POSIX EINTR).
- kConnectionCreationFailed: Permission to create a connection is denied. (POSIX EACCES)
- kGrantEnforcementError: Request was refused by Grant enforcement layer.

9.2.1.4.1.3 ReadData

This method reads (consume) datagrams from the data link socket connection. The maximum number of datagrams to read is provided with the parameter maxNumberOfDatagrams.

```
ara::core::Result<ara::core::Vector< T > > ReadData(
    size_t maxNumberOfDatagrams);
```

If the operation worked, it returns a vector of datagrams of type T, where each vector element contains exact one datagram of type T. Type T has to be set to a defined API data type (e.g. IEEE1722DatagramAAF) which represents the IEEE1722 sub type (e.g. AAF) when creating an instance of class IEEE1722RawDataStreamConsumer.

In case of an error it returns an `ara::core::ErrorCode` :

- Stream Not Connected: The connection is locally not yet established to the data link layer.
- Interrupted By Signal: The operation was interrupted by the system.

9.2.1.4.1.4 WriteData

This method writes datagrams to the data link socket connection. The data is provided as vector with datagrams of type T, where each vector element contain exactly one datagram of type T. Type T has to be set a defined API data type (e.g. IEEE1722DatagramAAF) which represents the IEEE1722 sub type (e.g. AAF) when creating an instance of class IEEE1722RawDataStreamProducer.The number of datagrams to write is provided in the length of the vector object instance.

```
ara::core:Result<size_t> WriteData(
        ara::core::Vector< T > data;
```

If the operation worked, it will return the actual number of datagrams written. In case of an error, it will return a `ara::core::ErrorCode`:

- Stream Not Connected: The connection is locally not yet established to the data link layer..
- Interrupted By Signal: The operation was interrupted by the system.

9.2.1.5 Security

Raw Data Stream using IEEE1722 protocol could utilize secured communication by using MACsec protocol on the corresponding data link layer (see [7]).

Access control to Raw Data Stream using IEEE1722 protocol can be defined by the IAM.

9.2.2 Usage of RawDataStreaming

This chapter describes how RawDataStreams using IEEE1722 protocol can be used in an AUTOSAR Adaptive Platform application.

RawDataStreaming using IEEE1722 protocol currently supports to consume IEEE1722 stream data via an `IEEE1722RawDataStreamConsmer` and produce IEEE1722 stream data via an `IEEE1722RawDataStreamProducer`. These use cases are described in chapter `Raw Data Streaming` at chapter "Raw Data Streaming using IEEE1722 protocol (data link layer)" of [5].

9.2.2.1 Usage

This chapter describes examples for utilization of an IEEE1722RawDataStreamConsumer and IEEE1722RawDataStreamProducer.

9.2.2.1.1 Example of usage as IEEE1722RawDataStreamConsumer

The code 9.3 shows how to use the IEEE1722RawDataStreamConsumer receiving data.

```
1 // NOTE! For simplicity the example does not use ara::core::Result.
2
3 #include "ara/core/instance_specifier.h"
4 #include "raw_data_stream.h"
5 int main() {
6     size_t maxNumOfDatagrams;
7     ara::core::Result<ara::core::Vector<ara::rds::IEEE1722AAFDatagram> >
    consumedDatagramVector;
8
9     // Instance Specifier from model
```

```
    ara::core::InstanceSpecifier instspec
    {...}

    // Create a RawDataStreamConsumer instance of subtype IEEE1722AAFDatagram
    aafStreamConsumer = IEEE1722RawDataStreamConsumer<ara::rds::IEEE1722AAFDatagram>.Create{instspec};

    // Connect to the data link socket
    aafStreamConsumer.Connect();

    // Read datagrams of type IEEE1722AAFDatagram via the IEEE1722RawDataStreamConsumer
    consumedDatagramVector = aafStreamConsumer.ReadData(maxNumOfDatagrams);

    // Iterate over the result vector
    for (int i=0 ; i < consumedDatagramVector.size() ; i++)
    {
        // process received datagrams
    };

    // Shutdown data link socket connection
    aafStreamConsumer.Shutdown(); return 0;
}
```

Listing 9.3: Example of usage as IEEE1722RawDataStreamConsumer

9.2.2.1.2 Example of usage as IEEE1722RawDataStreamProducer

The code 9.4 shows how to use the IEEE1722RawDataStreamProducer for sending data.

```
// NOTE! For simplicity the example does not use ara::core::Result.

#include "ara/core/instance_specifier.h"
#include "raw_data_stream.h"
int main() {
    size_t countOfTransmittedDatagrams;
    size8_t countOfProducedDatagrams;
    ara::core::Result<ara::core::Vector<ara::rds::IEEE1722AAFDatagram> > producedDatagramVector;

    // 4 datagrams shall be produced
    countOfProducedDatagrams = 4;

    // Instance Specifier from model
    ara::core::InstanceSpecifier instspec
    {...}

    // Create a RawDataStreamProducer instance of subtype IEEE1722AAFDatagram
    aafStreamProducer = IEEE1722RawDataStreamProducer<ara::rds::IEEE1722AAFDatagram>.Create{instspec};

```

```
20      // Connect to the data link socket
21      aafStreamProducer.Connect();
22
23      // Produce datagrams of type IEEE1722AAFDatagram
24      for (int i=0 ; i < countOfProducedDatagrams ; i++)
25      {
26          // produce datagrams
27          ara::rds::IEEE1722AAFDatagram localDatagram;
28          localDatagram.getAudioData();
29
30          // add to datagram vector for transmission
31          producedDatagramVector.add(localDatagram)
32      };
33
34
35      // Write datagrams of type IEEE1722AAFDatagram via the
        IEEE1722RawDataStreamProducer
36      countOfTransmittedDatagrams = aafStreamProducer.WriteData(
        producedDatagramVector);
37
38      // Shutdown data link socket connection
39      aafStreamConsumer.Shutdown(); return 0;
40  }
```

Listing 9.4: Example of usage as IEEE1722RawDataStreamProducer

9.3 Safety

The RawDataStream interface only transmits raw data without any data type information. Therefore Raw Data Stream interface cannot provide any data protection, such as E2E protection. If it is required it must be implemented in the application that uses the RawDataStream interface.

9.4 Hints for implementers

Implementation of Raw Data Streaming interface should be independent from the underlying Sockets API (e.g. POSIX Sockets).

10 Diagnostic Management

The Diagnostic Management consists of several parts. One is the Diagnostic Communication Management, which includes the handling of sessions and security within Diagnostic Conversations (see Diagnostic Communication Management). Another part is the Diagnostic Event Management, which handles diagnostic events in the system, error management and error memory management. A further area is the DoIP, which implements a diagnostic protocol based on the Internet Protocol. The most recent topic is the Road vehicles – Service-oriented vehicle diagnostics (SOVD), which allows diagnostics via REST on HTTP.

10.1 Diagnostic Communication Management

A UDS request is always processed in the context of a Diagnostic Conversation. A single Diagnostic Server can handle multiple Diagnostic Conversations in parallel. With this the Diagnostic Conversation is one of the central elements of the Diagnostic Communication Management.

A Diagnostic Conversation depicts a conversation between a distinct Diagnostic Client and a Diagnostic Server instance. The Diagnostic Conversation is dynamically allocated during runtime of the Diagnostic Server instance and has a specific life-cycle, how it is started up, replaced during runtime and teared down again. Because of the beforehand characteristics a Diagnostic Conversation has to be prioritized in special cases.

For an incoming UDS request, the Diagnostic Server instance is identified via the target address of the UDS request, whereas the identification of the Diagnostic Client is transport layer specific.

Parallel processing of client requests are possible as long as all conversations are in default session. This means if all running conversations are in default session, the clients can be handled fully parallel. If one conversation is in a non-default session, only this conversation can be handled.

UDS services are partially implemented internally (within DM) and externally (via PortInterface to AA). This means, that Adaptive Applications may have the full control of interface or UDS service.

10.2 Diagnostic Event Management

Contrary to the Diagnostic Communication Management where actions are mostly done as a reaction of a request, the Diagnostic Event Management is acting independent of requests. For the Diagnostic Event Management it is also unknown if the DM daemon is currently running or not. Therefore a caching mechanism is implemented, which acts in the cases, that an Adaptive Application is reporting information even if the

DM daemon is not running or there is a temporary communication interruption between ara::diag interfaces and the DM daemon.

Diagnostic Events are used to monitor a specific entity in the system, what may result in a Diagnostic Trouble Code state change or creating a snapshot of the current state and store it in the event memory. A diagnostic monitor is a routine running inside an AA entity determining the proper functionality of a component. This monitoring function identifies a specific fault type (e.g. short-circuit to ground, missing signal, etc.) for a monitoring path. A monitoring path represents the physical system or a circuit, that is being monitored (e.g. sensor input). Each monitoring path is associated with exactly one diagnostic event.

Event combination defines the ability of the DM to map several events to one DTC. It is used to adapt different monitor results to one significant fault, which is clearly valuable in a service-station.

The DM provides also Enable Conditions to ignore a certain reporting of monitor statuses.

Debouncing of reported events is the capability of the DM to filter out undesirable noise reported by monitors. This is used to mature the result of the monitor.

Operation Cycles are a further feature of the Diagnostic Event Management, where the Diagnostic Trouble Code status may be altered and stored under certain conditions. A typical Operation Cycle is the Ignition on/off cycle, where a time-frame is represented when the ignition of the car is on or off.

Furthermore the Diagnostic Event Management includes the Event Memory and features, which may alter the event related data in the Event Memory. Typical features are storing and clearing of event related data, aging of a Diagnostic Trouble Code, reporting the passive or active status of events, event memory overflow reactions, as well as reporting of the current number and order of event memory entries.

10.3 Negative Return Codes

Negative Return Codes can be given back by an Adaptive Application and can be routed back to the Diagnostic Client as a consequence of a request. Some Services may return any ISO-defined Negative Return Codes, others may return only a subset of it. Negative Return Codes may be part of an Error Code Domain and have their own namespace and ID.

10.4 Diagnostic communication over Internet Protocol

One of the protocols the Diagnostic Management may use for the communication is the DoIP. Another one can use a proprietary implementation and can be introduced via a specific protocol mechanism. DoIP supports the identification of the vehicle, power

mode status, activation of certain ECUs and dispatching of messages based on the target address of a Diagnostic Client.

10.5 Service oriented Vehicle Diagnostics

Besides the DoIP the Diagnostic Management may also use Road vehicles – Service-oriented vehicle diagnostics (SOVD) protocol, where the Diagnostic Service instance represents an entity or sub-component in the system. The SOVD comprises an own set of services and APIs, which also need to be verified and validated. Furthermore a data conversion mechanism is specified, which converts between SOVD specific and AUTOSAR specific data types.

11 Cryptography

The FC Crypto offers to applications and other AUTOSAR Adaptive Platform Functional Clusters a standardized interface to perform cryptographic operations, manage cryptographic keys, and handle certificates.

This document describes the required steps during design, integration and development and explains the provided APIs by FC Crypto.

11.1 Modeling and Mapping

11.1.1 Design-time Modeling

The application designer has to model the needed resources for this application by creating the respective `PortInterface`:

- `CryptoProviderInterface` for Crypto Provider, that might be used by this application in keyless operation
- `CryptoKeySlotInterface` for persisted writable key used by this application
- `CryptoKeySlotClientInterface` for persisted Read-Only key used by this application
- `CryptoCertificateInterface` for persisted certificates used by this application
- `CryptoCertificateGroupInterface` for persisted certificate group used by this application

11.1.1.1 KeySlot Design Steps

There are two types of application interaction with KeySlot:

- As an **Owner**, the application will be able to use and modify the KeySlot contents.
- As a **User**, the application will only be able to use the KeySlot contents "Read-Only".

This is reflected in the following design steps:

The System Designer will create:

1. `CryptoKeySlotDesign` with the following:
    - Aggregate `CryptoKeySlotDesign.algorithmDescription` of type `CryptoKeySlotDesignDescription` in which the `CryptoKeySlotDesignDescription.cryptoAlgorithmId` and the `CryptoKeySlotDesignDescription.cryptoObjectType` will be defined.

- Define if this is a shared KeySlot or not through `CryptoKeySlotDesign.shared` boolean parameter.
- Reference **Owner** and **User** of type `CryptoKeySlotUsageDesign`.

2. `CryptoKeySlotUserDesign`, which aggregates the `CryptoKeySlotUsageDesign` in `CryptoKeySlotUserDesign.keySlotUsage`.

The Application Designer will create:

1. `CryptoKeySlotUsageDesignMapping`, which maps `CryptoKeySlotUsageDesignMapping.cryptoKeySlotUsageDesign` to `CryptoKeySlotUsageDesignMapping.portPrototype`.
2. `CryptoKeySlotUserDesignMapping`, which maps `CryptoKeySlotUserDesignMapping.cryptoKeySlotUserDesign` to `CryptoKeySlotUserDesignMapping.processDesign`.
3. The required `CryptoKeySlotInterface` and `CryptoKeySlotClientInterface` interfaces.

Note: The design steps and roles described in this document are provided for illustrative purposes only and are not strictly required. Also, the design modeling itself is not mandatory for the proper functioning of cryptographic operations in the deployment phase; however, it is intended to support and enhance the design.

11.1.2 Integration-time Mapping

Each interface has to be mapped to a corresponding resource:

- Through `CryptoProviderToPortPrototypeMapping` each `CryptoProviderInterface` should be mapped to a `CryptoProvider`, which represents the Crypto Provider on the machine.
- Through `CryptoKeySlotToPortPrototypeMapping` each `CryptoKeySlotInterface` should be mapped to a `CryptoKeySlot`, which represents the key slot on the machine, which can be used and modified through this interface.
- Through `CryptoKeySlotToClientPortPrototypeMapping` each `CryptoKeySlotClientInterface` should be mapped to a `CryptoKeySlot`, which represents the key slot on the machine, which can be only used "Read-Only" through this interface.
- Through `CryptoCertificateToPortPrototypeMapping`, each `CryptoCertificateInterface` should be mapped to `CryptoCertificate`, which represents the certificate on the machine.
- Through `CryptoCertificateGroupToPortPrototypeMapping`, each `CryptoCertificateGroupInterface` should be mapped to `CryptoCertificateGroup`, which represents the certificate Group on the machine.

11.1.3 Development

The application developer can use each of the modelled interfaces by passing the `ara::core::InstanceSpecifier` representing the modelled `PortInterface` to an appropriate function of `ara::crypto`.

11.1.4 Modeling Examples

11.1.4.1 CryptoProvider

The Crypto Provider will aggregate all of its KeySlots.

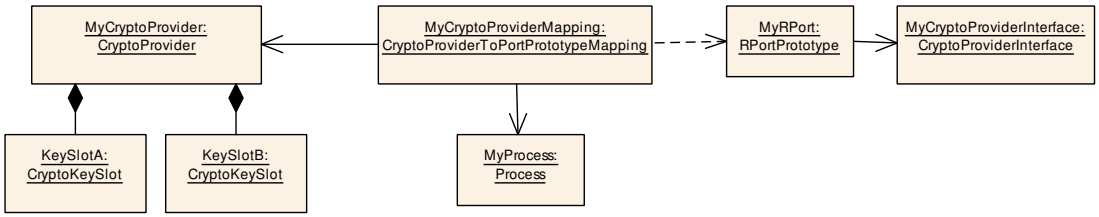


Figure 11.1: CryptoProvider Model Example

11.1.4.2 KeySlot

Each KeySlot will be linked to Owner and User applications.



Figure 11.2: Abstract KeySlot Model Example

11.1.4.2.1 Detailed KeySlot Model

The KeySlot **MyKeySlot** is shared between two applications A and B:

- **Application A**, who is running on **ProcessA**, is able to access the **MyKeySlot** through **KeySlotInterfaceA**, which makes Application A an Owner application, who has a **Read/Write access** to **MyKeySlot**.
- **Application B**, who is running on **ProcessB**, is able to access the **MyKeySlot** through **KeySlotClientInterfaceB**, which makes Application B a client application, who has a **Read-Only access** to **MyKeySlot**.

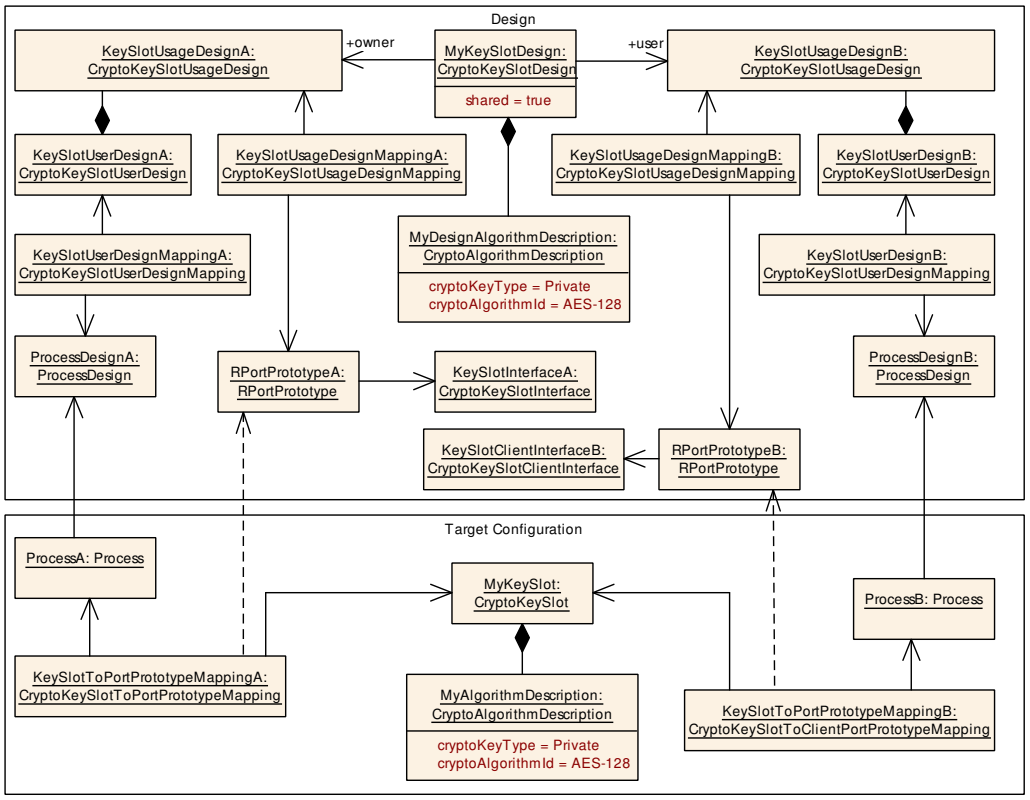


Figure 11.3: Detailed KeySlot Model Example

11.1.4.3 Certificate

The Certificate can be modeled as follows. The **writeAccess** boolean parameter defines if this application has write access to this certificate or not.

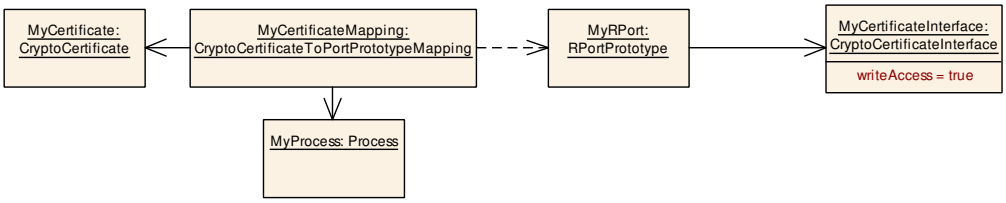


Figure 11.4: Certificate Model Example

11.1.4.4 CertificateGroup

The CertificateGroup can be modeled as follows. Optionally, a collection of individual certificates can be referenced to include some Certificates “by identity” into the CertificateGroup.

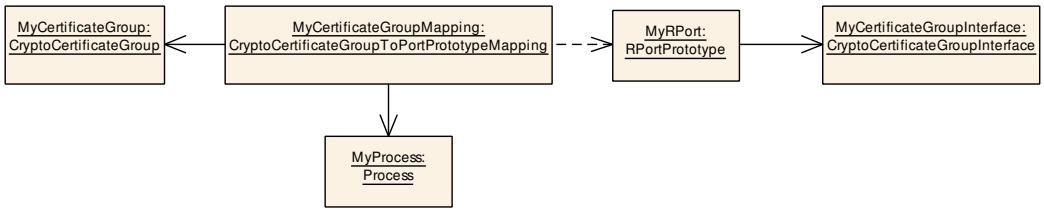


Figure 11.5: CertificateGroup Model Example

11.2 CryptoAPIs

11.2.1 Crypto Provider

CryptoProvider is a structural element that organizes cryptographic primitives. Every Crypto Provider typically represents either one hardware element, e.g., trusted platform module (TPM) or hardware security module (HSM), or one software element, e.g., cryptographic library.

FC Crypto provides two functions to instantiate a CryptoProvider:

- Keyless Operation

Instantiate a CryptoProvider by specifying the InstanceSpecifier of a CryptoProviderInterface, mainly used for keyless operation like hashing or RNG.

```
// The CryptoProvider mapped to CryptoProviderInstanceSpecifier
   instantiation
ara::crypto::cryp::CryptoProvider::Sptr CryptoProvider =
    LoadCryptoProvider(CryptoProviderInstanceSpecifier).ValueOrThrow();
```

- Key-Based Operation

Instantiate a CryptoProvider associated to KeySlot, mainly used for key-based operation like encryption, signing, etc.

```
// instantiate CryptoProvider, which is associated with this KeySlot
ara::crypto::cryp::CryptoProvider::Sptr CryptoProvider =
    KeySlot->GetCryptoProvider().ValueOrThrow();
```

11.2.1.1 Key Management

A KeySlot is an interface representing a storage location for cryptographic keys.

FC Crypto provides APIs to manage keys through various operations.

11.2.1.1.1 Load KeySlot

Load the KeySlot with either Read-Only or Read-Write permissions.

IAM feature can be used to control which applications are granted which type of access to the KeySlot, as defined in the configuration.

The access control will restrict the Read-Write access to CryptoKeySlot to the Processes, who have been referenced by a CryptoKeySlotToPortPrototypeMapping in the role process, which also references this CryptoKeySlot and a RPortPrototype of type CryptoKeySlotInterface.

The access control will give Read-Only access to CryptoKeySlot to the Processes, who have been referenced by CryptoKeySlotToClientPortPrototypeMapping in the role process, which also references this CryptoKeySlot and a RPortPrototype of type CryptoKeySlotClientInterface.

Any unauthorized access attempt shall be treated as a violation.

11.2.1.1.1.1 Writable KeySlot

KeySlot should have an Owner, who has read and write access to it and will be able to load it as modifiable KeySlot (WriteableKeySlot).

```
// instantiate a writable KeySlot by specifying the InstanceSpecifier of a
    CryptoKeySlotInterface.
cryp::WriteableKeySlot::Sptr modifiableKeySlot = LoadWriteableKeySlot(
    OwnerKeySlotInstanceSpecifier).ValueOrThrow();
```

11.2.1.1.1.2 Read-Only KeySlot

KeySlot might have several Users, who have Read-Only access to the KeySlot and will be only able to load it as Read-Only KeySlot (KeySlot).

```
// instantiate a Read-Only KeySlot by specifying the InstanceSpecifier of a
     CryptoKeySlotClientInterface or CryptoKeySlotInterface.
cryp::KeySlot::Sptr readOnlyKeySlot = LoadKeySlot(
    ClientKeySlotInstanceSpecifier).ValueOrThrow();
```

11.2.1.1.2 Update Writable KeySlot Content

CryptoObject provides API to be able to save its content in a KeySlot.

```
// Save the content of a CryptoObject sessionKey into a writable KeySlot
sessionKey->Save(modifiableKeySlot);
```

11.2.1.1.3 Delete Writable KeySlot Content

WritableKeySlot provides API to be able to clear its contents.

```
// Delete the content of a writable KeySlot
modifiableKeySlot->Clear();
```

11.2.1.1.4 Generate Cryptographic Keys

11.2.1.1.4.1 Symmetric Key

CryptoProvider provides API to be able to generate a symmetric key.

```
// Generate a symmetric key with AES-128 algorithm and allow MAC generation
    usage
cryp::SymmetricKey::Sptr sessionKey = CryptoProvider->GenerateSymmetricKey(
    "AES-128", kAllowMacGeneration).ValueOrThrow();
```

11.2.1.1.4.2 Asymmetric Key Pair

CryptoProvider provides API to be able to generate new public/private key pair and return the private key, from which we can get the public key.

```
// Generate an X25519 private key and allow key agreement usage
cryp::PrivateKey::Sptr gPrivateKey = CryptoProvider->GeneratePrivateKey("
    PRIVKEY/X25519", kAllowKeyAgreement).ValueOrThrow();

// Get the public key of this private key
cryp::PublicKey::Sptr gPublicKey = gPrivateKey->GetPublicKey().ValueOrThrow
    ();
```

11.2.1.1.5 Derive Cryptographic Keys

This is used to derive a key from an existing key and other key material.

```
// Create the Key derivation Function context for TLS 1.2 PRF master key
// derivation using DH and SHA-256
auto kdfCtx = CryptoProvider->CreateKeyDerivationFunctionCtx("KDF/
    TLS12_PRF_MASTER_KEY_DERIVE_DH/SHA-256").ValueOrThrow();

// Set the Key for the KDF context
kdfCtx->SetKey(premastersecret);

// use SetParameter to set the kdf specific parameters identified by their
    name in the first argument (in this case "Seed")
kdfCtx->SetParameter("Seed", seed);

// Derive the new key
auto derivedKeyMaterial = kdfCtx->DeriveKey("SYMKEY/GENERIC/384",
    kAllowMacGeneration | kAllowKeyDerivation).ValueOrThrow();
```

11.2.1.1.6 Import and Export Plaintext Cryptographic Keys

11.2.1.1.6.1 Export

To export CryptoObject in plaintext form.

```
// Export the key "CryptoObject" in plaintext kFormatRawValueOnly format
    and put the result in outBuffer
CryptoObject->ExportPublicly(outBuffer, kFormatRawValueOnly).ValueOrThrow()
    ;
```

11.2.1.1.6.2 Import

To import the provided Serialized data as a symmetric Key and set the allowed usages and the AlgId of this key.

```
// Import as symmetric key the provided plaintext data in serializedKey
    with
// kFormatRawValueOnly format to be used with AES-128 algorithm and allow
// persisting and allow MAC generation usage
auto symmetrickey = CryptoProvider->ImportSymmetricKey(serializedKey,
    kAllowPersist | kAllowMacGeneration, kFormatRawValueOnly, "AES-128").
    ValueOrThrow();
```

11.2.1.1.7 Import and Export Protected Cryptographic Keys

11.2.1.1.7.1 Export Secured Key

To export CryptoObject in an encrypted format using another KEK key-encryption-key.

```
// Export the key "toBeExportedKey" in secured format using KEK "kek" and
    algorithm algId and add the result in resultBuffer
auto size = ExportSecuredKey(algId, toBeExportedKey, resultBuffer, kek).
    ValueOrThrow();
```

11.2.1.1.7.2 Import Protected Key

To import an encrypted serializedData by decrypting it and then save it in KeySlot, so it can be later retrieved either from the KeySlot itself or via one of the provided API like GetPrivateKey, GetPublicKey, GetSymmetricKey or GetResponse.

```
// Create protected key import context for algorithm algID
auto protectedKeyImportingCtx = CryptoProvider->CreateProtectedKeyImportCtx
    (algID).ValueOrThrow();

// Set the transport key for the protected key import context
protectedKeyImportingCtx->SetKey(TransportKey);

// Import the protected key data serializedData into the modifiableKeySlot
protectedKeyImportingCtx->Import(serializedData, modifiableKeySlot);

// Get the imported private key
cryp::PrivateKey::Sptr gPrivateKey = protectedKeyImportingCtx->
    GetPrivateKey();
```

11.2.1.1.8 Allowed Usages

FC Crypto provides the ability to limit how the key can be used throughout bitmask variable `ara::crypto::AllowedUsageFlags`.

If the SetKey function of the SignerPrivateCtx is called with a private key with allow usage flags do not have the kAllowSigGeneration set, the function will return kUsageViolation error.

```
// Set key with a key with wrong allowed usage flags
auto status = SignerCtx->SetKey(prDecrptionKey);
if(status == kUsageViolation)
  // Error Handling
```

11.2.1.2 Cryptographic Transformations

A **CryptoProvider** implements a set of contexts, each of which represents a specific cryptographic transformation. The contexts can be created using the `Create*` functions e.g. `CreateKeyDerivationFunctionCtx` provided by the **CryptoProvider**.

- Available Cryptographic Transformations

- Random Number Generator (RNG)
- Hashing
- Message Authentication Code (MAC)
- Digital signatures
- Symmetric Encryption & Decryption
- Asymmetric Encryption & Decryption
- Authenticated Encryption with Associated Data
- Key Derivation Function (KDF)
- Key Wrapping
- Key Encapsulation Mechanism (KEM)
- Key Agreement

In order to set a key, all keyed contexts provide a SetKey function that takes either a **KeySlot** or a **CryptoObject** as a parameter.

There are two possible approaches to process a cryptographic transformation:

- **Single-call**: Use only one API call to process the Crypto Transformation. It is intended to ease and optimize calculation of small data (e.g., SecOC protected PDUs).

- **Streaming**: The application typically first calls `Start()`, then calls the `Update()` functions multiple times with the data to be processed, and calls the `Finish()` function to finalize the calculation. It is intended to process large amounts of data (e.g., software update).

11.2.1.2.1 Example Scenarios

11.2.1.2.1.1 Generate MAC

Use a KeySlot as the source of the key and use a single call approach to generate the MAC. This can be used if we have a preshared persisted key and a small amount of data e.g. SecOC.

```
// use instance specifier kMacKeyInstanceSpecifier to load the key slot
   with Read-Only permission
auto readOnlyKeySlot = LoadKeySlot(kMacKeyInstanceSpecifier).ValueOrThrow()
   ;

// Use the readOnlyKeySlot to load the associated CryptoProvider
auto CryptoProvider = readOnlyKeySlot->GetCryptoProvider().ValueOrThrow();

// Create CMAC context
cryp::MessageAuthnCodeCtx::Sptr cmacCtx = CryptoProvider->
   CreateMessageAuthnCodeCtx("AES-128/CMAC").ValueOrThrow();

// Set readOnlyKeySlot as the key source for the context, which means that
   the key stored in the KeySlot will be used for CMAC generation
cmacCtx->SetKey(readOnlyKeySlot);

// Prepare data and output buffer
ReadOnlyMemRegion macDataSpan("asdf");
std::vector<std::uint8_t> digest(32);

// Generate CMAC with Single-call API for the data in macDataSpan and put
   the result in digest
cmacCtx->Generate(macDataSpan, digest, ara::core::nullopt);
```

11.2.1.2.1.2 Import Public Key and Verify Signature

Use a temporary “Non-persistent” public key, which is imported from an external source to verify a signature.

```
// use instance specifier iSpecify to load the CryptoProvider
cryp::CryptoProvider::Sptr CryptoProvider = LoadCryptoProvider(iSpecify).
   ValueOrThrow();

// Import serializedKey in DER format with allow sig verification usage
auto publicKey = CryptoProvider->ImportPublicKey(serializedKey,
   kAllowSigVerification, kFormatDerEncoded, ara::core::nullopt).
   ValueOrThrow();
```

```
// Create Verifier context for ECDSA with curve secp256r1 and SHA-256
auto verifierCtx = CryptoProvider->CreateVerifierPublicCtx("ECDSA/secp256r1
    /SHA2-256").ValueOrThrow();

// Set the imported public key as the key for the verifier context
verifierCtx->SetKey(publicKey);

// Verify the signature using single-call API
bool verificationStatus = verifierCtx->Verify(data, signature,
    kFormatDerEncoded).ValueOrThrow();
```

11.2.1.2.1.3 Streaming Encryption

Use a KeySlot as the source of the key and use a streaming approach, which is more suitable with large data to be encrypted.

```
// use instance specifier KeySlotInstanceSpecifier to load the
    readOnlyKeySlot with Read-Only permission
auto readOnlyKeySlot = LoadKeySlot(KeySlotInstanceSpecifier).ValueOrThrow()
    ;

// Use the readOnlyKeySlot to load the associated CryptoProvider
auto CryptoProvider = readOnlyKeySlot->GetCryptoProvider().ValueOrThrow();

// Create Symmetric Cipher context for AES-128 in CBC mode with PKCS7
    padding
auto Cipherctx = CryptoProvider->CreateSymmetricCipherCtx("AES-128/CBC/
    PKCS7").ValueOrThrow();

// Set readOnlyKeySlot as the key source for the context, which means that
    the key stored in the readOnlyKeySlot will be used for encryption
Cipherctx->SetKey(readOnlyKeySlot);

// Start the context with encryption transformation and IV through Start
    API
Cipherctx->Start(CryptoTransform::kEncrypt, iv);

// Pass the plaintext data in chunks through Update API and collect
    ciphertext
Cipherctx->Update(plaintextChunk, ciphertextChunk);
// ... more updates ...
Cipherctx->Update(plaintextChunk, ciphertextChunk);

// Finalize the encryption process through Finish API and collect remaining
     ciphertext
Cipherctx->Finish(plaintextChunk, ciphertextChunk);
```

11.2.1.2.1.4 Hash Calculation

Example of Keyless operation, in which we will load the CryptoProvider directly by its instance specifier, then create the hashing context, then use it to calculate the hash for the required data.

```
// use instance specifier CryptoProviderInstanceSpecifier to load the
    CryptoProvider
cryp::CryptoProvider::Sptr CryptoProvider = LoadCryptoProvider(
    CryptoProviderInstanceSpecifier).ValueOrThrow();

// Create Hash context for SHA-512
auto hashctx = CryptoProvider->CreateHashFunctionCtx("HASH/SHA-512").
    ValueOrThrow();

// Calculate the hash digest using single-call API without IV
hashctx->CalculateHashDigest(data, digest, ara::core::nullopt);
```

11.2.2 Certificate Handling (X.509 Provider)

11.2.2.1 X.509 Provider Overview

The X509Provider provides access to manage certificates and is responsible for:

- Persisting and loading X.509 certificates.
- Parsing and validating X.509 certificates and certificate chains.
- Creating Certificate Signing Requests.
- Parsing, validating and storing CRLs.
- Creating OCSP requests and parsing and validating responses.

Unlike CryptoProvider there is only one X.509 Provider instance on one machine, which can be loaded using the following API.

```
// Load X.509 certificate provider
ara::crypto::x509::X509Provider::Sptr  x509provider = LoadX509Provider().
    ValueOrThrow();
```

11.2.2.2 Certificate

11.2.2.2.1 Loading

We can load the Certificate or Certificate Group by passing the instanceSpecifier that represent it CryptoCertificateInterface or CryptoCertificateGroupInterface. Here the access control can be applied to restrict the access to Certificate or Certificate Group to the Processes, who have been referenced by a `CryptoCertificateToPort-PrototypeMapping` or `CryptoCertificateGroupToPortPrototypeMapping` in the role process, which also references the CryptoCertificate or CryptoCertificate-

Group and a RPortPrototype of type CryptoCertificateInterface or CertificateGroupInterface. Any unauthorized access attempt shall be treated as a violation.

11.2.2.2.1.1 Load Certificate

```
// load Certificate by passing its instance Specifier CertInstanceSpecifier
auto Cert = x509provider->LoadCertificate(CertInstanceSpecifier).
    ValueOrThrow();
```

11.2.2.2.1.2 Load Certificate Group

```
// load Certificate Group by passing its instance Specifier
// CertGroupInstanceSpecifier
ara::core::Vector<x509::Certificate::Sptrc> certificates = x509provider->
    LoadCertificateGroup(CertGroupInstanceSpecifier).ValueOrThrow();
```

11.2.2.2.2 Parsing

11.2.2.2.2.1 Parse Certificate

Parse a serialized representation of an X.509 certificate and create a certificate object.

```
// parse the provided certificate raw data certRawdata in DER encoding
    format
auto cert = x509provider->ParseCert(certRawdata, kFormatDerEncoded).
    ValueOrThrow();
```

11.2.2.2.2.2 Parse Certificate Chain

Parse a serialized representation of a certificate chain, creating a new Certificate instance for each parsed certificate. Certificates in the returned vector will be placed in the same order as provided in Certificate Chain serialized data.

```
// parse the provided certificate chain raw data certChainRawdata in DER
    encoding format
auto certChain = x509provider->ParseCertChain(certChainRawdata,
    kFormatDerEncoded).ValueOrThrow();
```

11.2.2.2.3 Validating

Validating the certificate will be handled through a certificate validation context in which we can set the root and intermediate certificates for the certificate chain and set a special certificate validation option like kIgnoreNotBefore, kIgnoreCrl, kOcspResponseCompleteChain or other and trigger the leaf certificate validation and get the result of this certificate validation.

```
// Create Certificate validation context
auto certValidationCtx = x509provider->CreateCertificateValidationCtx().
    ValueOrThrow();
```

```
// set trusted and untrusted certificates
certValidationCtx->AddTrustedCertificate(rootCert);
certValidationCtx->AddUntrustedCertificate(intermediateCert);

// set a certificate validation option from the once specified in
   CertificateValidationOptions or other user specific option
certValidationCtx->SetValidationOptions(x509::kOcspOff);

// validate certificate
auto validationResult = certValidationCtx->ValidateCertificate(leafCert).
   ValueOrThrow();

// get the certificate validation result
CertificateValidationStatus validationResult = validationResult->
   GetValidationStatus();

// get the certificate chain
auto chain = validationResult->GetCertificateChain();

// If it fails, get the failed certificate
auto failedCertificate = validationResult->GetFailedCertificate();
```

11.2.2.2.4 Persisting and Deleting

11.2.2.2.4.1 Persist Certificate

Persist certificate in certificate slot identified by its instanceSpecifier.

```
// persist certificateObject in the certificate slot identified by instance
    specifier certificateInstanceSpecifier
auto result = x509provider->ImportCertificate(certificateInstanceSpecifier,
    certificateObject);
```

11.2.2.2.4.2 Delete Certificate

Clear the certificate saved inside the certificate slot identified by its instance specifier. Return True if certificate deleted or False if there is no certificate saved in this slot.

```
// delete the save certificate in certificate slot identified by instance
   specifier certificateInstanceSpecifier
auto result = x509provider->ClearCertificateSlot(
   certificateInstanceSpecifier);
```

11.2.2.2.4.3 Persist Certificate in Group

Persist certificate inside certificate group identified by certificate group instanceSpecifier.

```
// persist certificateObject1 inside certificate group identified by
    instance specifier certificateGroupInstanceSpecifier
auto result = x509provider->ImportCertificateInGroup(
    certificateGroupInstanceSpecifier, certificateObject1);
```

11.2.2.2.4.4 Remove Certificate from Group

Remove a specific certificate from certificate group identified by certificate group instanceSpecifier.

```
// remove certificateObject1 from certificate group identified by instance
    specifier certificateGroupInstanceSpecifier
auto result = x509provider->RemoveCertificateFromGroup(
    certificateGroupInstanceSpecifier, certificateObject1);
```

11.2.2.3 Certificate Signing Request (CSR)

The following API creates and returns Certificate Signing Request CertSignRequest according to RFC 2986 by taking DN, Public key and optionally DER-encoded data and sign them with the already created signer private context SignerPrivateCtx, whose private key must correspond to the passed Public key.

```
// create CSR using public key csrPublicKey and DN subjectDn and additional
     DER encoded data additionalDerEncodedData and sign it with Signer
    Private Context csrSignerCtx
ara::crypto::x509::CertSignRequest::Sptrc pCsr = x509provider->
    CreateCertSignRequest(csrSignerCtx, subjectDn, csrPublicKey,
    additionalDerEncodedData);
```

11.2.2.4 Certificate Revocation Lists (CRLs)

11.2.2.4.1 Import CRL

Parse and persist a Certificate Revocation List (CRL) or Delta CRL and linked it with the CA certificate of this CRL identified by the passed InstanceSpecifier.

```
// parse and persist the provided crlData, whose format is
    kFormatDerEncoded and linked to the CA certificate identified by
    crlCaInstanceSpecifier
CertificateValidationStatus crlStatus = x509provider->ImportCrl(
    crlCaInstanceSpecifier, crlData, kFormatDerEncoded);
```

11.2.2.4.2 Delete CRL

Deletes a previously stored CRL identified by instance specifier.

```
// Delete a stored CRL identified by crlCaInstanceSpecifier.
x509provider->DeleteCrl(crlCaInstanceSpecifier);
```

11.2.2.4.3 CRL Validation

The validation against the imported CRL will be performed within the validation context that utilizes this CA certificate.

The validation against CRL can be ignored if we use the following API.

```
// set a certificate validation option to ignore the check against CRL
certValidationCtx->SetValidationOptions(x509::kIgnoreCrl);
```

11.2.2.5 Online Certificate Status Protocol (OCSP)

11.2.2.5.1 Create OCSP Request

The Following API can be used to create OCSP request according to RFC 6960 and return it as a serialized buffer.

```
// create OCSP request for certificate cert, whose issuer certificate is
    issuerCert and signed the request using the passed signer private
    context signerCtx
ara::core::Vector<std::uint8_t> requestOcsp = CreateOcspRequest(cert,
    issuerCert, signerCtx);
```

11.2.2.5.2 Parse and Use OCSP Response

This OCSP request can be sent to an OCSP responder, who will send back the OCSP response. Then the received OcspResponse can be parsed and can be used during certificate validation.

```
// Parse the received OCSP response, which is DER-Encoded
auto ocspResponse = x509provider->ParseOcspResponse(ocspResponseRaw,
    kFormatDerEncoded).ValueOrThrow();

// get the status of this parsed OCSP response
OcspResponseStatus StatusOcsp = ocspResponse->GetResponseStatus();

// add the parsed OCSP response to the certificate validation context
certValidationCtx->AddOcspResponse(ocspResponse, ara::core::nullopt);

// activate the validation option to use this OCSP response only for target
     certificate
certValidationCtx->SetValidationOptions(x509::kOcspResponseTargetOnly);
```

Note: The use of ValueOrThrow() in this document is for illustrative purposes only. It is not part of a standardized AUTOSAR error-handling approach. The actual error-handling strategy should be defined by the user.

12 Sensor Interfaces

12.1 Interface Configuration

In this release, there are four types of sensors including radar, lidar, USS and camera. All the services are independent and optional. During the design of an automated driving system, the supported service and corresponding version are selected. There is no capability negotiation for the specific service during runtime.

In addtion, there is also a set of optional signal in each defined service. To reduce development costs, especially in terms of functional safety, the set of optionals and thus the resulting sensor interface shall be fixed in the design time as well, which is transmitted via the Capability Vector in corresponding service.

12.2 Data Transfer

Each service is composed by a method of capability vector for optional signal indication and an event for data transmission. The data transmission is supposed to be periodical and unidirectional from the sensor to the service consumer. The consumer shall process the sensor data according to the timestamp in the event, not the event receive time.

12.3 Service profile and data type

In different use cases, the same type of sensor may have different performance, i.e. also possible to use different data type resolutions with the same capability setting. For example, the sensor with low profile may provide the data with int 8 and float 16, while the sensor with high profile may provide the data with int 16 and float 32. In this release, only one profile is supported with typical float. More profiles should be considered in future releases.

13 Related Documentation

[1] Explanation of Adaptive Platform Design
AUTOSAR_AP_EXP_PlatformDesign

[2] Specification of Adaptive Platform Core
AUTOSAR_AP_SWS_Core

[3] Specification of Platform Health Management
AUTOSAR_AP_SWS_PlatformHealthManagement

[4] Specification of Manifest
AUTOSAR_AP_TPS_ManifestSpecification

[5] Specification of Raw Data Stream
AUTOSAR_AP_SWS_RawDataStream

[6] IEEE Standard 1722-2016 - IEEE Standard for a Transport Protocol for Time-Sensitive Applications in Bridged Local Area Networks

[7] Explanation of MACsec and MKA Protocols implementation and configuration guidelines
AUTOSAR_AP_EXP_MACsec