| Document Title | Guidelines for using Adaptive Platform interfaces |
|---|---|
| **Document Owner** | AUTOSAR |
| **Document Responsibility** | AUTOSAR |
| **Document Identification No** | 929 |

| | |
|---|---|
| **Document Status** | published |
| **Part of AUTOSAR Standard** | Adaptive Platform |
| **Part of Standard Release** | R24-11 |

**Document Change History**

| Date | Release | Changed by | Description |
|---|---|---|---|
| 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.<br>• Various clean-ups. |
| 2023-11-23 | R23-11 | AUTOSAR Release Management | • Introduction of the Common Regulations, Diagnostics Management, and the Raw Data Streams chapter<br>• Overall improvement of the Execution Management, Update and Configuration Management, and Persistency chapters<br>• Overall update of the State Management chapter<br>• Removal of the Deterministic Execution Client in the Execution Management chapter<br>• 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<br>• 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<br>• Moderate changes in the State Management chapter<br>• Minor changes in the Persistency Chapter |
| 2019-11-28 | R19-11 | AUTOSAR Release Management | • Persistency and Platform Health Management chapters added<br>• 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**

# Contents

# 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

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`.

# 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()
  .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:

```
Result<String> res = some_function()
  .Bind([](int v){ return v + 1; })
  .Bind([](int v){ return std::to_string(v); })
  .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

```
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 together with the `#define ARA_VENDOR` (see [SWS_CORE_00093]) 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:

```
// Could also be directly defined as main
int adaptive_application_main(int argc, char* argv[])
{
    // TODO: Perform setup actions that do not rely on ARA APIs if
    // necessary

    // Initialize ara::*
    // ara::core::Initialize() might make use of vendor specific
    // command line arguments for configuration.
    // If this is the case ara::core::Initialize() will remove
    // 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().

```
/*
* Example: ExecutionClient instantiation, providing Termination Handler and Execution State report
*/
bool termination_requested{false};

auto executionClientResult = ara::exec::ExecutionClient::Create([&]() {
    termination_requested = true;
});

if (executionClientResult.HasValue()) {
    ara::exec::ExecutionClient executionClient = std::move(executionClientResult).Value();

    // 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 Unexpected 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::kFailed == error) {
29          logger.LogError() << "Initial machine state transition failed.";
30          /* ... */
31      } else if (ara::exec::ExecErrc::kCancelled == error) {
32          logger.LogInfo() << "Initial machine state transition has been
     cancelled.";
33          /* ... */
34      } else if (ara::exec::ExecErrc::kCommunicationError == 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  auto functionGroupState = ara::exec::FunctionGroupState(kFunctionGroup,
     kFunctionGroupState);
47  auto setStateResult = stateClient.SetState(functionGroupState).GetResult
     ();
48
49  if (setStateResult.HasValue()) {
50      logger.LogInfo() << "Transition to Function Group State succeeded.";
51      /* ... */
52  } else {
53      auto setStateError = setStateResult.Error();
54      logger.LogError() << "Failed to set Function Group State. Error code:
      " << setStateError;
55
56      if (ara::exec::ExecErrc::kMetaModelError != setStateError) {
57          auto getExecutionErrorResult = stateClient.GetExecutionError(
     functionGroupState);
58
59          if (getExecutionErrorResult.HasValue()) {
60              ara::exec::ExecutionErrorEvent event = std::move(
     getExecutionErrorResult).Value();
61              logger.LogError() << "Execution Error: " << event.
     executionError;
62              /* ... */
63          } else {
64              ara::core::ErrorCode errc = getExecutionErrorResult.Error();
65
66              if (errc == ara::exec::ExecErrc::kCommunicationError) {
```

```
67                     logger.LogError() << "Failed to get Execution Error.
    Error code: " << static_cast<ara::core::ErrorDomain::CodeType>(ara::exec
    ::ExecErrc::kCommunicationError);
68                     /* ... */
69                 }
70             }
71         }
72     }
```

# 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 or restarted

Some of these functions are critical. Therefor the access to the Trigger fields 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 newly introduced optional StateMachine approach of StateManagement the user is now 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

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 ExecutionError, coming from PlatformHealthManagement and/or ExecutionManagement

| Execution Error | Next State |
|---|---|
| 11 | Recovery |
| 12 | Startup |
| 111 | Recovery |
| 23 | Suspend |
| 24 | Off |
| ANY | Shutdown |

**Figure 5.3: StateMachine ErrorTable**

To keep the configuration of StateMachines simple, the StateMachines do not contain business / project logic. Therefore customer application(s), called SMControlApplication, are needed on top of StateManagement, which decide under which conditions to request which state from a certain StateMachine.

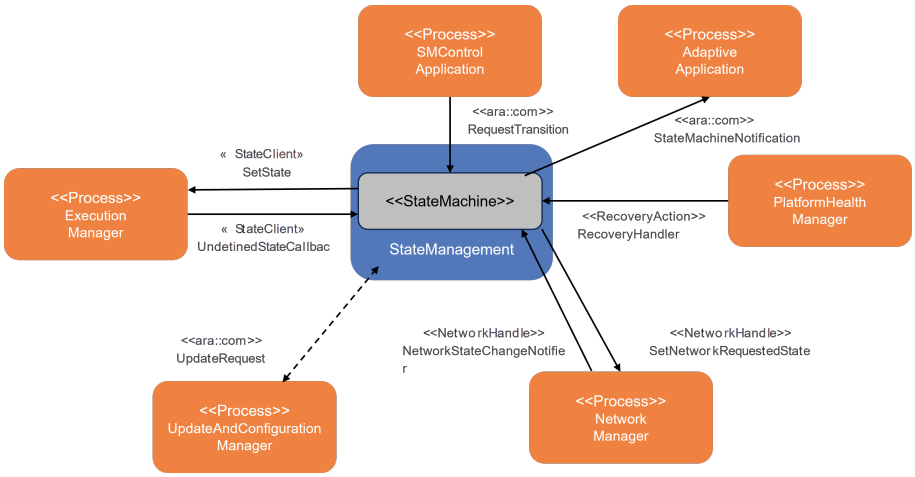


**Figure 5.4: StateMachine interactions**

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

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

To cover the interaction with UpdateAndConfigurationManagement each StateMachine has to have a number of mandatory states:

- PrepareUpdate, where - at least - all managed FunctionGroups and StateMachines shall be set to Off state, respectively stopped
- VerifyUpdate, where - at least - all managed FunctionGroups and StateMachines shall be set to Verify state, respectively started in Verify state
- PrepareRollback, where - at least - all managed FunctionGroups and StateMachines shall be set to Off state, respectively stopped

The request of the corresponding states will be done within StateManagement, following the needed sequence from UpdateAndConfigurationManagement.

Please note that the interaction with DiagnosticManagement for the StateMachine approach will be defined within future release of AUTOSAR.

# 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.

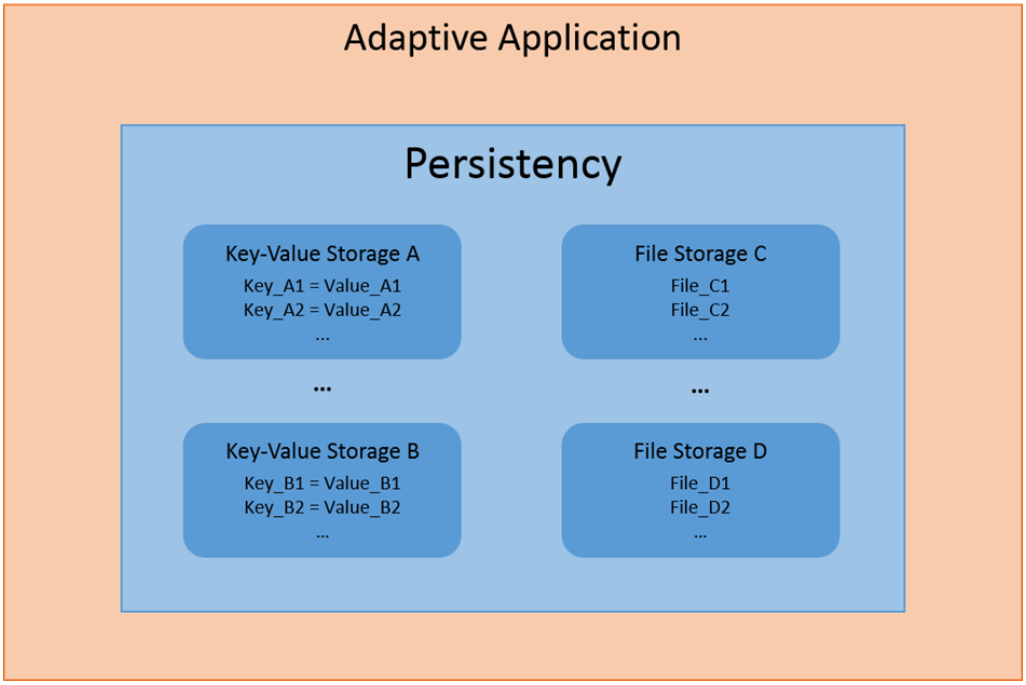


**Figure 6.1: Persistency functionality overview**

### 6.1.1 Key Value Storage

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

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 Storage and File Storage as a port interface 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 deployment stage (e.g. storage location, redundancy CRC, redundancy M out of N, etc.)

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

Note: AUTOSAR_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 Storage and File Storage. 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 Keys/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 deployment 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 deployment 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 offers supervision of software. The software developers provide a modelled configuration (manifest) defining correct order and/or timing constraints of the checkpoints that will be reported to PHM. The software reports checkpoints when a specific point of the code is reached. PHM supervises that the checkpoints are reached in the correct order and with the correct timing. If the designed order or timing constraints are missed, State Management is informed to trigger recovery actions.

Additionally, the watchdog is serviced and stopped triggering if a failure in EM or SM is detected.

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 timeout), then refreshing of the Watchdog will be stopped.

API:

```
virtual void ara::phm::RecoveryAction::RecoveryHandler (const ara::exec
::ExecutionErrorEvent &executionError, TypeOfSupervision supervision)=0;
```

## 7.3 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 $p_{\text{UDP}}$, the maximum transmission unit $\text{MTU}_{\text{Eth}}$, the individual layers header sizes $\text{Header}_i$ 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<br>Reliable message delivery |
| **Limitations** | Max blocksize ~1400 byte<br>Segmentation in application layer | Loss of one UDP message causes loss of the complete SOMEIP message<br>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 constructor takes as input the instance specifier qualifying the network binding and parameters for the instance.

```
RawDataStreamClient(const ara::com::InstanceSpecifier\& instance);
RawDataStreamServer(const ara::com::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 RawDataStream 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` from `ara::rds::RawErrorDomain`:

- 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.

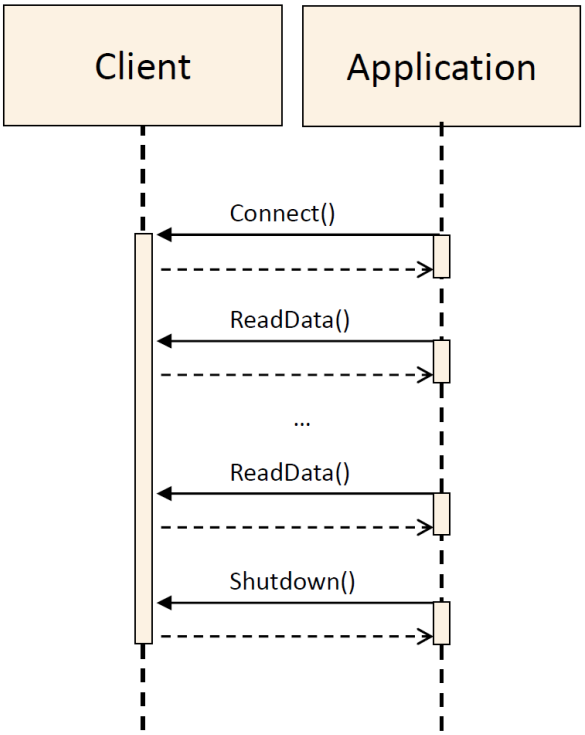


**Figure 9.1: Client sequence diagram**

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

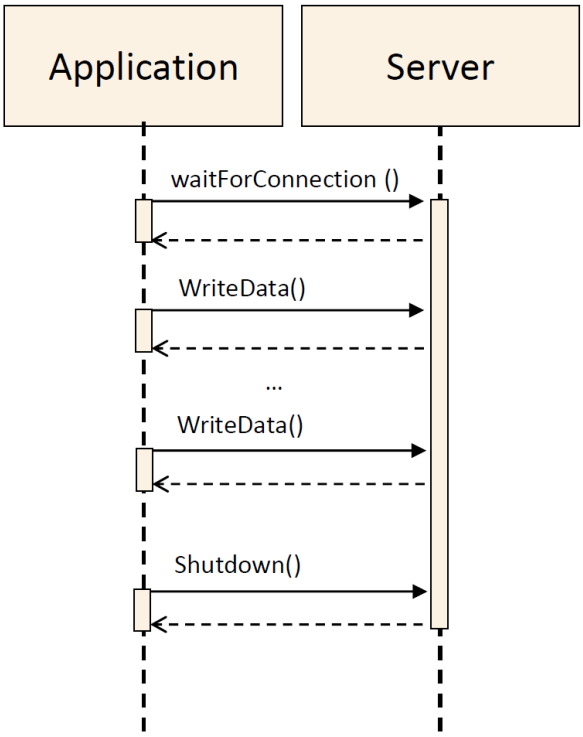


**Figure 9.2: Client sequence diagram**

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::com::raw::RawDataStream::ReadDataResult result;

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

    // Create RawDataStream Server instance
    ara::rds::RawDataStreamServer server{instspec};

    // 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
```

```
38     server.Shutdown(); return 0;
39 }
```

**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.

```
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 rval;
7      ara::com::raw::RawDataStream::ReadDataResult result;
8
9      // Instance Specifier from model
10     ara::core::InstanceSpecifier instspec
11     {...}
12
13     // Create a RawDataStreamClient instance
14     ara::rds::RawDataStreamClient client {instspec};
15
16     // Connect to RawDataStream Server
17     client.Connect();
18
19     // Write data to RawData stream in chunks of 40 bytes
20     int i=0;
21     do {
22         std::unique_ptr<uint8_t> write_buf (new uint8_t[1024]{.....});
23         rval = client.WriteData(std::move(write_buf), 40);
24         ++i;
25     } while (i<1000);
26
27     // Read data from the RawData stream in chunks of 4 bytes
28     do {
29         result = client.ReadData(4);
30         rval = result.numberOfBytes;
31         if (rval > 0){
32             // assumes the data is printable
33             std::cout << "-->" << result.data.get() << std::endl;
34         }
35     } while (rval > 0);
36
37     // Shutdown RawDataStream connection
38     client.Shutdown(); return 0;
39 }
```

**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.macAddressStreamId` 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` from `ara::rds::RawErrorDomain`:

- 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
10     ara::core::InstanceSpecifier instspec
11     {...}
12
13     // 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};

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

    // Produce datagrams of type IEEE1722AAFDatagram
    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 Service oriented Vehicle Diagnostics, which allows diagnostics wire i.e. web browser and consists of an entity-based communication paradigm.

## 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.

Interfaces are partially implemented internally and externally. This means, that Adaptive Applications may have the full control of interface or UDS service. Internal interfaces or services are implemented within a diagnostic stack and work without involvement of an external Adaptive Application.

## 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 an 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 proprietary and can be introduced wire specific transport

protocol mechanisms. 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 Service oriented Vehicle Diagnostics protocol, where the Diagnostic Service instance represents an entity or sub-component in the system. The Diagnostic Management implements interaction between Unified Diagnostic Services and the Service oriented Vehicle Diagnostics protocol. 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 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