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

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

Document Change History

Date	Release	Changed by	Change Description
2018-10-31	18-10	AUTOSAR Release Management	• Initial release

Disclaimer

Table of 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 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 and Prereads.

2 Core Types

2.1 Error handling

Handling errors is a crucial topic for any software development. For safety-critical software, it is even more important, because lifes 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`.

2.1.1 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.

2.1.2 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. Due to their templated nature, both value and error can be of any type. However, *ErrorType* is defaulted to `ara::core::ErrorCode`, and it is expected that this assignment is kept throughout the Adaptive Platform.

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 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 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.

2.1.2.1 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 a custom error message and/or a support data value:

```
return Result<BigClass>::FromError(
     MyEnum::some_error,                // ErrorCode enum value
     "this operation did not work",     // custom error message
     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.

2.1.2.2 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 actually 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, in fact, 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.

2.1.2.3 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 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 actually 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 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 Execution Management

3.1 Execution State

The Execution State characterizes the internal lifecycle of any Process. Each Process needs to report changes in its Execution State to Execution Management, using the `ExecutionClient::ReportExecutionState()` interface (see [2]).

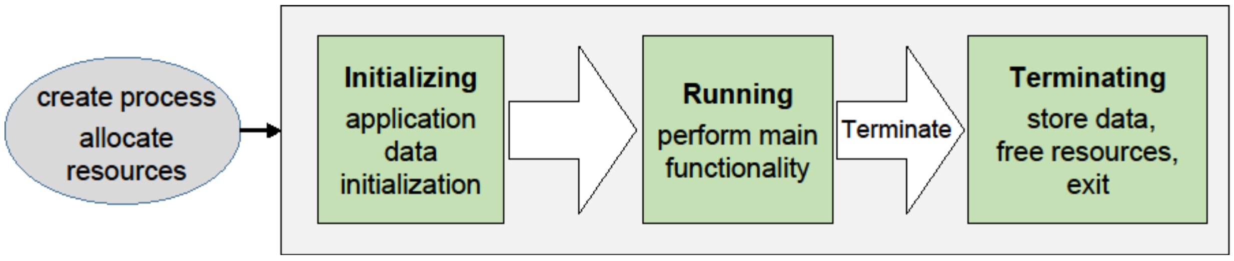
create process
allocate resources
Initializing
application data initialization
Running
perform main functionality
Terminate
Terminating
store data, free resources, exit

Figure 3-1 Execution States

Upon Process startup, Execution Management shall consider Process initialization complete when the state `kRunning` is reported (see [SWS_EM_01053]). Please note that Service Discovery can introduce nondeterministic delays and thus is advised to be done after reporting `kRunning` state; thus, the Process may not have completed all its initialization when the `kRunning` state is reported by the Process.

Execution Management initiates Process termination by sending the SIGTERM signal to a Process. On reception of SIGTERM, the Process shall acknowledge the state change request by reporting `kTerminating` to Execution Management (see [SWS_EM_01070).

In the case of a self-terminating Process, the Process shall initiate self-termination by reporting the `kTerminating` state to Execution Management (see [SWS_EM_01071).

After reporting `kTerminating`, the Process is expected to save persistent data and free all internally used resources. The Process indicates completion of the Terminating state by simply exiting (with an appropriate exit code). Execution Management does not require an explicit notification of actual Process termination by the process itself.

3.2 Deterministic Execution

Execution Management supports a fully deterministic multithreaded execution of a Process, so processing a given set of input data always produces a consistent output within a bounded time, i.e. the behavior is reproducible.

Expected use cases of the AUTOSAR Adaptive Platform where such determinism is required include redundant execution in a Software Lockstep framework for systems with high safety goals (up to ASIL D) and reuse of verified software. For more details see [2], section “Deterministic Execution”.

A Process that can be executed fully deterministically must be designed, implemented and integrated in a way such that it is independent of processor load caused by other functions and calculations, sporadic unrelated events, race conditions, deviating random numbers etc.

Non-deterministic behavior may arise from different reasons; for example insufficient computing resources, or uncoordinated access of data, potentially by multiple threads running on multiple processor cores. The order in which the threads access such data will affect the result, which makes it non-deterministic.

Full deterministic execution includes:

- Time Determinism: The output of the calculation is always produced before a given deadline. The resource demands of the Process need to be described in a standardized way, so the integrator can assign sufficient resources to the Process (see subsection “Real-Time Resources” in [2]).
- Data Determinism: Given the same input and internal state, the calculation always produces the same output. The rest of this section will describe how to achieve Data Determinism.

Execution Management} provides `DeterministicClient` library functions to support deterministic execution:

- Control of a process-internal cycle by wait point API `WaitForNextActivation()` ([SWS_EM_01301]). The Process shall execute one cycle when the API returns and then call the API again to wait for the next activation. A return value of the API controls the internal lifecycle (e.g. init, run, terminate) of the Process, which must be prepared accordingly ([SWS_EM_01302], [SWS_EM_01303] and [SWS_EM_01304]).
- A blocking deterministic worker pool API `RunWorkerPool()` ([SWS_EM_01305]) for execution of a set of container elements ([SWS_EM_01306]) which are processed in parallel or sequentially by the same worker runnable object (i.e. application function).
- APIs `GetActivationTime()` ([SWS_EM_01310]) and `GetNextActivationTime()` ([SWS_EM_01311]) to provide activation time stamps which don't change until the Process reaches its next wait point.
- API `GetRandom()` to provide random numbers ([SWS_EM_01308]). If used from within the worker pool, the random numbers are assigned to specific container elements to allow deterministic redundant execution.

To ensure deterministic behavior, only a “deterministic subset” of all available APIs may be used by the deterministic user Process, including the worker runnable objects:

- The Process is not allowed to create threads on its own by using normal POSIX mechanisms or access any other POSIX APIs directly, to avoid the risk of inducing indeterministic behavior.

- Only a “deterministic subset” of all available ara::com mechanisms are allowed to be used by the Process. A detailed list of such APIs and mechanisms will be provided at a later point in time.
- Only the following ara::exec interfaces may be used:
  - `DeterministicClient`
  - `ExecutionClient`
- No other ARA interfaces are allowed to be accessed by the user Process.

If the worker pool API `RunWorkerPool()` is used, the worker runnable object which processes the container elements, i.e. the jobs to be computed, needs to satisfy certain implementation rules to ensure Data Determinism:

- The runnable object is not allowed to exchange any information while it is running, i.e. it doesn't access data which can be altered by other instances of the runnable object to avoid race conditions.

  Rationale: The runnable object instances can physically run in parallel or sequentially in any order. Timing between individual workers is not guaranteed. The Operating System is scheduling threads individually. Concurrent influencing of the same data will result in indeterminate results.

- No locks and synchronization points except common joins for all workers by returning from `RunWorkerPool()` (e.g. no Semaphores/Mutexes, no locking/blocking).

  Rationale: locking/blocking makes Process runtime in-deterministic. Workers are provided to increase utilization of runtime. If synchronization is needed, a return from RunWorkerPool() is necessary.

The worker pool cannot be used to process multiple different tasks in parallel. The use of multiple potentially different explicit functions (worker runnable objects) could add unnecessary complexity and can lead to extremely heterogeneous runtime utilization, as each worker may have different computing time. This would complicate the planning of resource deployment, which is necessary for black-box integration.

Example of the implementation of Worker Pool Users, i.e. of a worker runnable object:

```
class MyWorker1
:   public DeterministicClient::WorkerrunnableBase<myContainer::
      value_type, MyWorker1>
{
public:
    void worker_runnable(myContainer::value_type& container_element,
      DeterministicClient::WorkerThread& t)
    {
      // Get a unique and deterministic pseudo-random number}
      uint64_t random_number = t.GetRandom();
    }
};
```

Worker-thread object:

```
class DeterministicClient::WorkerThread
{
   // returns a deterministic pseudo-random number}
   // which is unique for each container element}
   uint64_t GetRandom();

   ...
};
```

4 State Management

4.1 Component State

Component States are used to control Processes in a more fine grained way than it would be possible with Execution Management even without the need to unload and reload them from and to memory.

It is used e.g. to support 'late-wakeup': The Processes can continue their work immediately when a new wakeup reason is detected during shutdown.

Therefor the Executable has to register to State Management. It is done implicitly when the constructor the `ComponentClient::ComponentClient()` is called (see [2]).

A component is identified by State Management via the provided ara::core::string identifier in the `ComponentClient` constructor.

Two modes for the component state are supported

- Polling mode for safety critical components to have a deterministic behavior.
- Event mode for all other components.

The mode of the `ComponentClient` (polling or event-based) is specified by an additional parameter for the `ComponentClient` constructor

When the polling mode is selected the component has to use `ComponentClient::GetNewState()` to get next state from State Management

When the event mode is selected the component has to use `ComponentClient::SetStateUpdateHandler()` to provide an event-handler to State Management The Components are informed then via the given callback about needed state changes.

The states are given as ara::core::string. Predefined states are

- 'kOff'': Executable shall persist its data similar to when SigTerm is received, but Process remains in memory
- 'kFastOff': Similar to 'kOff', but only a subset of data shall be persisted (when needed at all). Used e.g. for fast shutdown in production diagnosis
- 'kOn': Executable works in regular manner
- 'khardReset': Used in diagnostic session when a hard reset is requested. Behavior is project specific
- 'ksoftReset': Used in diagnostic session when a soft reset is requested. Behavior is project specific

Further predefined states could be introduced in the future because other functionalities might have to be supported e.g. communication control states due to diagnostic communication control request needs.

Each component can decline to enter the requested stated due to its current needs e.g. the Bluetooth stack can decline to enter the 'kOff' state when a phonecall is ongoing.
Therefor State Management secures state requests with a project specific timeout and retry-count. To enable State Management to do so each Component has to report its state using `ComponentClient::ReportUpdatedState` .in a project specific time-slot.
When timeout and retry counts are exceeded State Management tries again to set the state with an enforcement flag.

To make State Management aware that a component is no longer available (don't care about timeouts any more), each component within a Process has to de-register from State Management. Therefore the destructor of the component interface has to be called (de-registration is done implicitly) `ComponentClient::~ComponentClient()` when components are no longer interested to receive component state updates e.g. a Process is terminated.

5 References

[1] Explanations of Adaptive Platform Design, AUTOSAR_EXP_PlatformDesign.pdf.

[2] Specification of Execution Management, AUTOSAR_SWS_StateManagement.pdf.