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

Document Status	Final
Part of AUTOSAR Standard	Adaptive Platform
Part of Standard Release	18-03

Document Change History

Date	Release	Changed by	Description
2018-03-29	18-03	AUTOSAR Release Management	• Initial release

Disclaimer

Table of Contents

1 Introduction 4

2 Related documentation 4

2.1 Input documents & related standards and norms 4

3 Requirements Tracing 4

4 Type specification 5

4.1 `Future` and `Promise` 5

4.2 `Optional` data type 10

1 Introduction

Core types defines common classes and functionality used by multiple functional clusters as part of their public interfaces.

2 Related documentation

2.1 Input documents & related standards and norms

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

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

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

3 Requirements Tracing

The following tables reference the requirements specified in <CITATIONS_OF_CONTRIBUTED_DOCUMENTS> and links to the fulfillment of these. Please note that if column “Satisfied by” is empty for a specific requirement this means that this requirement is not fulfilled by this document.

Requirement	Description	Satisfied by
[RS_AP_00114]	C++ binding shall be compatible with C++11.	[SWS_Core_00320] [SWS_Core_00321] [SWS_Core_00322] [SWS_Core_00323] [SWS_Core_00324] [SWS_Core_00325] [SWS_Core_00326] [SWS_Core_00327] [SWS_Core_00328] [SWS_Core_00329] [SWS_Core_00330] [SWS_Core_00331] [SWS_Core_00332] [SWS_Core_00340] [SWS_Core_00341] [SWS_Core_00342] [SWS_Core_00343] [SWS_Core_00344] [SWS_Core_00345] [SWS_Core_00346] [SWS_Core_00347] [SWS_Core_00348] [SWS_Core_01033] [SWS_Core_01034] [SWS_Core_01035] [SWS_Core_01036] [SWS_Core_01037] [SWS_Core_01038] [SWS_Core_01039] [SWS_Core_01040] [SWS_Core_01041] [SWS_Core_01042] [SWS_Core_01043]
[Req_Id_1]	No description	[SWS_XYZ_00001]
[Req_Id_2]	No description	[SWS_XYZ_00001]

4 Type specification

4.1 `Future` and `Promise`

The following section describes the `Future` and `Promise` class templates used in ara::core to provide and retrieve the results of method calls. Whenever there is a mention of a standard C++11 item (class, class template, enum or function) such as `std::future` or `std::promise`, the implied source material is [1]. Whenever there is a mention of an experimental C++ item such as `std::experimental::future::IsReady`, the implied source material is [2].

Futures are technically referred to as "asynchronous return objects", and promises are referred to as "asynchronous providers". Their interaction is made possible by a "shared state". The "shared state" concept is described in [1], section 30.6.4. The

description also applies to the shared state behind `ara::core Future` and `Promise`, with the following amendments:

- ", as used by async when policy is launch::deferred" is removed from paragraph 2.
- Paragraph 10, referring to "`promise::set_value_at_thread_exit`", is removed.

[SWS_Core_00320] **`FutureStatus`** ⌈ This class shall provide an enumeration `FutureStatus` which contains an operation status for timed Wait functions of `ara::core::Future`.

```
enum class FutureStatus : uint8_t {
    kReady,
    kTimeout
};
```

⌋*(RS_AP_00114)*
Note: The meaning of the values is the same as that of the corresponding ones in `std::future_status`. **Note:** The value `std::future_status::deferred` has no correspondent.

[SWS_Core_00321] **`Future` class template** ⌈ This class shall provide a `Future` class template which provides a way to check and retrieve results of method calls.

```
template<typename T>
class Future {
  // Default constructor
  Future() noexcept;
  // Move constructor
  Future(Future&&) noexcept;
  // Default copy constructor deleted
  Future(const Future&) = delete;
  // Specialized unwrapping constructor
  Future(Future<Future<T>>&&) noexcept;

  ~Future();

  // Move assignment operator
  Future& operator=(Future&&) noexcept;
  // Default copy assignment operator deleted
  Future& operator=(const Future&) = delete;

  // Returns the result
  T Get();

  // Check if the Future has any shared state
  bool Valid() const noexcept;

  // Block until the shared state is ready.
  void Wait() const;
```

```
  // Wait for a specified relative time.
  template< class Rep, class Period >
  FutureStatus WaitFor(
    const std::chrono::duration<Rep,Period>& timeout_duration) const;

  // Wait until a specified absolute time.
  template <class Clock, class Duration>
  FutureStatus WaitUntil(
    const std::chrono::time_point<Clock,Duration>& abs_time) const;

  // Set a continuation for when the shared state is ready.
  template <typename F>
  auto Then(F&& func) -> Future<decltype(func(std::move(*this)))>;

  // Return true only when the shared state is ready.
  bool IsReady() const;
};
```

⌋(*RS_AP_00114*)

[SWS_Core_00322] **`Future` default constructor** ⌈ The `Future` constructor

```
1 Future() noexcept;
```

shall behave as the corresponding `std::future` constructor. ⌋(*RS_AP_00114*)

[SWS_Core_00323] **`Future` move constructor** ⌈ The `Future` constructor

```
1 Future(Future&&) noexcept;
```

shall behave as the corresponding `std::future` constructor. ⌋(*RS_AP_00114*)

[SWS_Core_00324] **`Future` unwrapping constructor** ⌈ The `Future` constructor

```
1 Future(Future<Future<T>>&&) noexcept;
```

shall behave as the corresponding `std::future` constructor. ⌋(*RS_AP_00114*)

[SWS_Core_00325] **Move assignment operator** ⌈ The `Future` operator

```
1 Future& operator=(Future&&) noexcept;
```

shall behave as the corresponding `std::future` operator. ⌋(*RS_AP_00114*)

[SWS_Core_00326] **`Future::get`** ⌈ The `Future` function

```
1 T Get();
```

shall behave as the corresponding `std::future` function. ⌋(*RS_AP_00114*)

[SWS_Core_00327] **`Future::Valid`** ⌈ The `Future` function

```
1 bool Valid() const noexcept;
```

shall behave as the corresponding `std::future` function. ⌋(*RS_AP_00114*)

[SWS_Core_00328] **`Future::Wait`** ⌈ The `Future` function

```
void Wait() const;
```

shall behave as the corresponding `std::future` function. ⌋*(RS_AP_00114)*

[SWS_Core_00329] **`Future::WaitFor`** ⌈ The `Future` function

```
template< class Rep, class Period >
FutureStatus WaitFor(
  const std::chrono::duration<Rep,Period>& timeout_duration) const;
```

shall behave as the corresponding `std::future` function. ⌋*(RS_AP_00114)*

[SWS_Core_00330] **`Future::WaitUntil`** ⌈ The `Future` function

```
template <class Clock, class Duration>
FutureStatus WaitUntil(
  const std::chrono::time_point<Clock,Duration>& abs_time) const;
```

shall behave as the corresponding `std::future` function. ⌋*(RS_AP_00114)*

[SWS_Core_00331] **`Future::Then`** ⌈ The `Future` function

```
template <typename F>
auto Then(F&& func) -> Future<decltype(func(std::move(*this)))>;
```

shall behave as the corresponding `std::experimental::future` function, but without performing *implicit unwrapping*. ⌋*(RS_AP_00114)*

[SWS_Core_00332] **`Future::IsReady`** ⌈ The `Future` function

```
bool IsReady() const;
```

shall behave as the corresponding `std::experimental::future` function. ⌋*(RS_AP_00114)*

[SWS_Core_00340] **`Promise` class template** ⌈ This class shall provide a `Promise` class template which provides a way to set a value or exception into the shared state.

```
template <class T>
class Promise {
public:
  // Default constructor
  Promise();
  // Default copy constructor deleted
  Promise(const Promise&) = delete;
  // Move constructor
  Promise(Promise&&) noexcept;

  ~Promise();

  // Default copy assignment operator deleted
  Promise& operator=(const Promise&) = delete;
  // Move assignment operator
```

```
  Promise& operator=(Promise&&) noexcept;

  // Return a Future with the same shared state.
  Future<T> GetFuture();

  // Store an exception in the shared state.
  void SetException(std::exception_ptr p);

  // Store a value in the shared state.
  void SetValue(const T& value);
  void SetValue(T&& value);

  // Set a handler to be called, upon future destruction.
  void SetFutureDtorHandler(std::function<void> handler);
};
```

⌋(*RS_AP_00114*)

[SWS_Core_00341] **`Promise` default constructor** ⌈ The `Promise` constructor

```
Promise();
```

shall behave as the corresponding `std::promise` constructor. ⌋(*RS_AP_00114*)

[SWS_Core_00342] **`Promise` move constructor** ⌈ The `Promise` constructor

```
Promise(Promise&&) noexcept;
```

shall behave as the corresponding `std::promise` constructor. ⌋(*RS_AP_00114*)

[SWS_Core_00343] **`Promise` move assignment operator** ⌈ The `Promise` operator

```
Promise& operator=(Promise&&) noexcept;
```

shall behave as the corresponding `std::promise` operator. Note: there is no `promise::Swap` function. ⌋(*RS_AP_00114*)

[SWS_Core_00344] **`Promise::GetFuture`** ⌈ The `Promise` function

```
Future<T> GetFuture();
```

shall behave as the corresponding `std::promise` function. ⌋(*RS_AP_00114*)

[SWS_Core_00345] **`Promise::SetValue`** ⌈ The `Promise` function

```
void SetValue(const T& value);
```

shall behave as the corresponding `std::promise` function. ⌋(*RS_AP_00114*)

[SWS_Core_00346] **`Promise::SetValue`, forwarding reference version** ⌈ The `Promise` function

```
void SetValue(T&& value);
```

shall behave as the corresponding `std::promise` function. ⌋(*RS_AP_00114*)

[SWS_Core_00347] **`Promise::SetException`** ⌈ The `Promise` function

```
1 void SetException(std::exception_ptr p);
```

shall behave as the corresponding `std::promise` function. ⌋(*RS_AP_00114*)

[SWS_Core_00348] **`Promise::SetFutureDtorHandler`** ⌈ The `Promise` function

```
1 void SetFutureDtorHandler(std::function<void> handler);
```

sets a handler to be called upon destruction of the `Future` associated with the `Promise`'s shared state.
Note: the destruction of the associated `Future` implies the value or exception set by the `Promise` cannot be received from that point on. ⌋(*RS_AP_00114*)

4.2 `Optional` data type

The following section describes the `Optional` class template `ara::core::Optional` used in ara::core to provide access to optional record elements of a `Structure Implementation data type`. Whenever there is a mention of the standard C++17 Item `std::optional`, the implied source material is [3].

The class template std::optional manages optional record elements, i.e. values that may or may not be present. Every instance of an optional record element either contains a value or does not. The existence can be evaluated during runtime.
Note: Mandatory record elements are declared directly with the corresponding `ImplementationDataType` without using `std::optional`.

[SWS_Core_01033] **`Optional` class template** ⌈ This class provides a way to check and set the availability of optional record elements.

```
template< class T >
class Optional {

  // Default constructor
  Optional() noexcept;
  // Move constructor
  Optional( Optional&& ) noexcept;
  // Copy constructor
  Optional( const Optional<T>& );

  ~Optional();

  // Move assignment operator
  Optional& operator=(Optional&&) noexcept;
  // Default copy assignment operator
  Optional& operator=(const Optional&);

  // Returns the value
  T& Value();
```

```
  // Check if the value is available
  bool HasValue();

  // Overload bool operator
  operator bool();

  // Destroy value and mark as unavailable
  void Reset();

};
```

⌋(*RS_AP_00114*)

[SWS_Core_01034] `Optional` **default constructor** ⌈ The `Optional` constructor

```
Optional();
```

shall behave as the `std::optional` constructor. ⌋(*RS_AP_00114*)

[SWS_Core_01035] `Optional` **move constructor** ⌈ The `Optional` move constructor

```
Optional( Optional&& ) noexcept;
```

shall behave as the `std::optional` move constructor. ⌋(*RS_AP_00114*)

[SWS_Core_01036] `Optional` **copy constructor** ⌈ The `Optional` copy constructor

```
Optional( const Optional& );
```

shall behave as the `std::optional` copy constructor. ⌋(*RS_AP_00114*)

[SWS_Core_01037] `Optional` **destructor** ⌈ The `Optional` destructor

```
~Optional();
```

shall behave as the `std::optional` destructor. ⌋(*RS_AP_00114*)

[SWS_Core_01038] `Optional` **move assignment operator** ⌈ The `Optional` move assignment operator

```
Optional& operator=(Optional&&) noexcept;
```

shall behave as the `std::optional` move assignment operator. ⌋(*RS_AP_00114*)

[SWS_Core_01039] `Optional` **default copy assignment operator** ⌈ The `Optional` default copy assignment operator

```
Optional& operator=(const Optional&);
```

shall behave as the `std::optional` default copy assignment operator. ⌋(*RS_AP_00114*)

[SWS_Core_01040] `Optional` **function to get contained value** ⌈ The `Optional` function to get contained value

```
T& Value();
```

If `*this` contains a value, returns a reference to the contained value.
Otherwise, throws a `std::bad_optional_access` exception. ⌋*(RS_AP_00114)*

[SWS_Core_01041] `Optional` function to check availability of contained value ⌈ The `Optional` checker function to check the availability of the contained value

```
bool HasValue();
```

true if `*this` contains a value, false if `*this` does not contain a value. ⌋ *(RS_AP_00114)*

[SWS_Core_01042] `Optional` bool operator ⌈ The `Optional` bool operator

```
operator bool();
```

true if `*this` contains a value, false if `*this` does not contain a value. ⌋ *(RS_AP_00114)*

[SWS_Core_01043] `Optional` Reset function ⌈ The `Optional` Reset function

```
void Reset();
```

If `*this` contains a value, destroy that value as if by `Value().T::~T()`. Otherwise, there are no effects.
`*this` does not contain a value after this call. ⌋*(RS_AP_00114)*