European Component Oriented Architecture (ECOA®) Collaboration Programme: Preliminary version of the ECOA C++ Language Binding

Dassault Ref No: DGT 2041088-A
Thales DMS Ref No: 69398927-035 --

Issue: 7

Prepared by
Dassault Aviation and Thales DMS

Note: *This specification is preliminary and is subject to further adjustments. Consequently, users are advised to exercise caution when relying on the information herein. No warranties are provided regarding the completeness or accuracy of the information in this preliminary version. The final version of the document will be released to reflect further improvements.*

Without prejudice to the property rights relating to the ECOA AS6 Standard, the information in this document relating to the changes envisaged for the transition from the ECOA AS6 Standard to the ECOA AS7 Standard is the intellectual property of Dassault Aviation and Thales DMS France SAS. The information set out in this document is provided solely on an ‘as is’ basis and co-developers of this specification make no warranties expressed or implied, including no warranties as to completeness, accuracy or fitness for purpose, with respect to any of the information.

This Page Intentionally Left Blank

This Page Intentionally Left Blank

Contents

Figures

Tables

0 Introduction

This Architecture Specification provides the specification for creating ECOA®-based systems. It describes the standardised programming interfaces and data-model that allow a developer to construct an ECOA®-based system. It uses terms defined in the Definitions (Architecture Specification Part 2). The details of the other documents comprising the rest of this Architecture Specification can be found in Section 3.

This document describes the C++ (ref ISO/IEC 14882:2003(E)) language binding for the component and container APIs that facilitate communication between the component instances and their container in an ECOA® system.

This document describes the API identified in ECOA Component Implementation models with the following information:

- APIType = "ECOA_C++"
- APIVersion = "7.1"

Warning: This document is not exhaustive regarding the Option-specific types and APIs.

This document is structured as follows:

- Section 6 describes the Component to Language Mapping;
- Section 7 describes the method of passing parameters;
- Section 8 describes the Component Context;
- Section 9 describes the basic types that are provided and the types that can be derived from them;
- Section 9.6 describes the Component Interface;
- Section 11 describes the Container Interface;
- Section 12 describes the Container Types;
- Section 13 describes the External Interface;
- Section **Erreur ! Source du renvoi introuvable.** describes the Default Values;
- Section 15 describes External Components;
- Section 16 describes PeriodicTriggerManager Components;
- Section **Erreur ! Source du renvoi introuvable.** describes DynamicTriggerManager Components;
- Section **Erreur ! Source du renvoi introuvable.** describes Supervisor Components;
- Section 17 provides a reference C++ header for the ECOA® library, usable in any C++ binding implementation;

1 Scope

This Architecture Specification specifies a uniform method for design, development and integration of software systems using a component oriented approach.

2 Warning

This specification represents the output of a research programme. Compliance with this specification shall not in itself relieve any person from any legal obligations imposed upon them. Product development shall rely on the BNAE publications of the ECOA standard.

3 Normative References

Architecture Specification Part 1	Dassault Ref No: DGT 2041078-A Thales DMS Ref No: 69398915-035 -- Issue 7 Architecture Specification Part 1 – Concepts
Architecture Specification Part 2	Dassault Ref No: DGT 2041081-A Thales DMS Ref No: 69398916-035 -- Issue 7 Architecture Specification Part 2 – Definitions
Architecture Specification Part 3	Dassault Ref No: DGT 2041082-A Thales DMS Ref No: 69398917-035 -- Issue 7 Architecture Specification Part 3 – Mechanisms
Architecture Specification Part 4	Dassault Ref No: DGT 2041083-A Thales DMS Ref No: 69398918-035 -- Issue 7 Architecture Specification Part 4 – Software Interface
Architecture Specification Part 5	Dassault Ref No: DGT 2041084-A Thales DMS Ref No: 69398919-035 -- Issue 7 Architecture Specification Part 5 – High Level Platform Requirements
Architecture Specification Part 6	Dassault Ref No: DGT 2041491-A Thales DMS Ref No: 69398920-035 -- Issue 7 Architecture Specification Part 6 – Options
Architecture Specification Part 7	Dassault Ref No: DGT 2041086-A Thales DMS Ref No: 69398925-035 -- Issue 7 Architecture Specification Part 7 – Metamodel

ISO/IEC 8652:1995(E) with COR.1:2000	Ada95 Reference Manual Issue 1
ISO/IEC 9899:1999(E)	Programming Languages – C
ISO/IEC 14882:2003(E)	Programming Languages C++
SPARK_LRM	The SPADE Ada Kernel (including RavenSPARK) Issue 7.3

4 Definitions

For the purpose of this standard, the definitions given in Architecture Specification Part 2 apply.

5 Abbreviations

API	Application Programming Interface
ECOA	European Component Oriented Architecture
ID	Identifier
PINFO	Persistent Information
UTC	Coordinated Universal Time
XML	eXtensible Markup Language

6 Component to Language Mapping

This section gives an overview of the Component and Container Interface APIs, in terms of the filenames and the overall structure of the files.

6.1 Overview of interfaces

The table below lists the Component and Container Interface APIs defined in the "generic" Software Interface document ([Architecture Specification Part 4], Table 1).

The “level” column specifies whether the interface is mandatory (i.e. required in any language binding), or optional.

The "implemented" column specifies whether the interface is implemented in the present language binding.

Table 1 Component and Container Interfaces

Category	Abstract API Name	Level	Covered by the ECOA C++ Language Binding
Events API	Event_Send	**MANDATORY**	**YES MANDATORY**
	Event_Received	**MANDATORY**	**YES MANDATORY**
Request Response API	Request_Sync	**MANDATORY**	**YES MANDATORY**
	Request_Async	**MANDATORY**	**YES MANDATORY**
	Request_Received	**MANDATORY**	**YES MANDATORY**
	Altenative 1 : Response_Received	**MANDATORY***	**YES MANDATORY**
	Altenative 2 : Response_Actually_Received Response_Not_Received		**NO**
	Response_Send	**MANDATORY**	**YES MANDATORY**
	Request_Cancel	**OPTIONAL**	**YES [BINDING OPTION REQUEST CANCEL]**
Versioned Data API	Get_Read_Access	**MANDATORY**	**YES MANDATORY**
	Release_Read_Access	**MANDATORY**	**YES MANDATORY**
	Updated	**MANDATORY**	**YES MANDATORY**
	Get_Write_Access	**MANDATORY****	**YES MANDATORY**
	Get_Selected_Write_Access		**YES [BINDING OPTION SELECTED WRITE ACCESS]**
	Cancel_Write_Access	**MANDATORY**	**YES MANDATORY**
	Publish_Write_Access	**MANDATORY**	**YES MANDATORY**

Category	Abstract API Name	Level	Covered by the ECOA C++ Language Binding
	Is_Initialized	**OPTIONAL**	**YES [BINDING OPTION EXTENDED VD API]**
	Release_All_Data_Handles	**OPTIONAL**	**YES [BINDING OPTION EXTENDED VD API]**
Properties API	Get_Value	**MANDATORY**	**YES MANDATORY**
Runtime Lifecycle API	Initialize_Received	**MANDATORY**	**YES MANDATORY**
	Start_Received	**MANDATORY**	**YES MANDATORY**
	Stop_Received	**MANDATORY**	**YES MANDATORY**
	Shutdown_Received	**MANDATORY**	**YES MANDATORY**
	Reset_Received	**MANDATORY**	**YES MANDATORY**
Supervisor Components	On_State_Change	**OPTIONAL**	**YES [OPTION SUPERVISION]**
	Get_Executable_Status	**OPTIONAL**	
	Executable_Command	**OPTIONAL**	
	Component_State_Command	**OPTIONAL**	
	Get_Component_Status	**OPTIONAL**	
	Get_Variable	**OPTIONAL**	
	Set_Variable	**OPTIONAL**	
Logging and Fault Management Services API	Altenative 1 : • Log_Debug • Log_Trace • Log_Info • Log_Warning • Raise_Error • Raise_Fatal_Error	**MANDATORY****	**YES MANDATORY**
	Alternative 2 : • Flex_Log • Flex_Raise_Fatal_Error		**YES [BINDING OPTION FLEX LOG]**
	Error_Notification	**OPTIONAL**	**YES [OPTION FAULT HANDLER]**
Time Services API	Get_Relative_Local_Time	**MANDATORY**	**YES MANDATORY**
	Get_UTC_Time	**OPTIONAL**	**YES [OPTION UTC TIME]**
	Get_Absolute_System_Time	**MANDATORY**	**YES MANDATORY**
	Get_Relative_Local_Time_Resolution	**OPTIONAL**	**YES [BINDING OPTION TIME RESOLUTION]**
	Get_UTC_Time_Resolution	**OPTIONAL**	

Category	Abstract API Name	Level	Covered by the ECOA C++ Language Binding
	Get_Absolute_System_ Time_Resolution	**OPTIONAL**	
Triggers	Trigger_Set	**MANDATORY**	**YES MANDATORY**
	Trigger_Cancel	**MANDATORY**	**YES MANDATORY**
Persistent Information (PINFO) Management	Read	**MANDATORY**	**YES MANDATORY**
	Write	**OPTIONAL**	**YES [OPTION PINFO WRITE]**
	Seek	**MANDATORY**	**YES MANDATORY**
Context Management	Save_Warm_Start_Context	**OPTIONAL**	**YES [OPTION WARM START CONTEXT]**
External Interface	External_Event_Received	**OPTIONAL**	**YES [OPTION EXTERNAL INTERFACE]**
External Components	External_Routine	**MANDATORY**	**YES MANDATORY**
	Start_External_Task	**MANDATORY**	**YES MANDATORY**
	Stop_External_Task	**MANDATORY**	**YES MANDATORY**

* it is mandatory to define exactly one of the API alternatives in a language binding.
** it is mandatory to define at least one of the API alternatives in a language binding.

6.2 Overview of files

A namespace is created called **`#component_impl_name#`** that will contain all items related to the Component Implementation.

Two objects (classes in C++) need to be created for Object-Oriented languages such as C++.

The first class is the Component class:

- It is a concrete class, which shall contain the user functional code to implement the required operations. The instance objects of this class, corresponding to each declared Component Instance, will be allocated by the container. There is a public member variable 'Container' that is a pointer set by the infrastructure to allow the Component to invoke Container operations.
- It is created by the Component Implementer.
- This document provides template specification of the following elements:
    - The **Component Interface**, which contains:
        - The declaration of methods that the component type is required to provide to the container,
        - User data of the Component Instance being declared within two attributes of this class corresponding to **user context** and **warm start context** structures.
    - The **Component Implementation** (i.e. the implementation of the **Component Interface**),
    - Associated with the **Component** class is the **Component User Context** (i.e. the declaration of the actual attributes contained in the user context and warm start context structures).

The second class is the **Container** class:

- It is a concrete class, which shall contain the platform specific functional code of the container.
- This class would usually be generated by an ECOA platform provider/integrator.
- This document is limited to the template specification of the following elements:
    - The **Container Interface** (i.e. the operations that the Container API offers to the Component), which the **Component Implementation** needs in order to call the ECOA infrastructure. This container interface is designed to conceal any platform specific implementation towards the Component Implementation (such as the actual contents of the platform hook),
    - Associated with the **Container** class are the **Container Types**, which the **Component Implementation** needs in order to declare, use and store various handles.
- The **Container Implementation** is platform dependent and is out of scope of this document. In order to be able to invoke component instances the container implementation would include a definition of the component interface.

Figure 1 shows the relationship between classes mentioned above, whilst Table 2 shows the filename mappings.

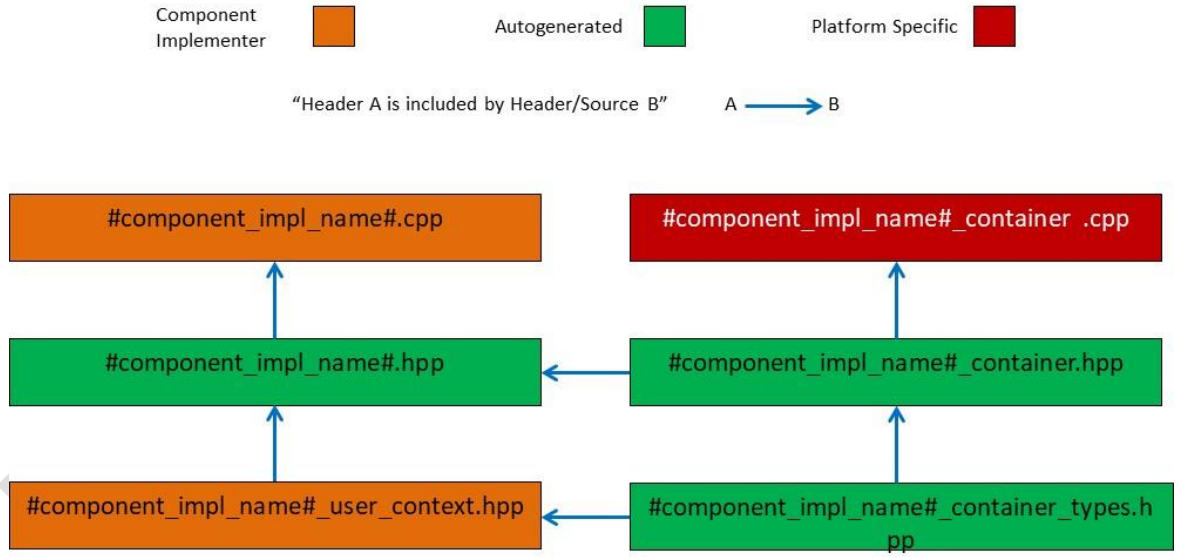

Figure 1 C++ Files Organization

Complementary files are necessary in case external communications are defined, either using optional External Interfaces or using External Components. See §**Erreur ! Source du renvoi introuvable.** and §**Erreur ! Source du renvoi introuvable.**.

Table 2 Filename Mapping

Filename	Use
#component_impl_name#.hpp	**Component Interface**: This header (.hpp) contains the declaration of the entry points provided by the component and callable by the container.
#component_impl_name#.cpp	**Component Implementation:** This source (.cpp) implements the Component Interface.
#component_impl_name#_container.hpp	**Component Interface**: This header (.hpp) contains the declaration of the functions provided by the container and callable by the component.
#component_impl_name#_container_types.hpp	**Container Types**: This header (.hpp) contains the declaration of container-level data types usable by the component. These types are related to the Container for a Component Implementation.
#component_impl_name#_container.cpp	**Container Implementation**: This source (.cpp) implements the Container Interface. It is out of scope of this document, as it is platform dependent. The Container may actually be a collection of source files depending upon the platform implementation
#component_impl_name#_user_context.hpp	**Component User Context** User extensions to Component Context. These types are related to the Component Implementation.
#component_impl_name#_ External_Interface.hpp	External interface provided to non-ECOA software by a component container when EXTERNAL_INTERFACE option is defined.
#component_impl_name#_ External_Component_Interface.hpp	External thread interface of an External Component.
#library#.hpp	Types defined from a data type library.
#library#_initialize.hpp	Initialization functions for #library# types.
#library#_compare.hpp	Comparison functions for #library# types.
epsilon_definition.hpp	Optional file to be added by users. Definition of epsilon values to be applied for floating values comparison in the library (see §9.7.2)
ECOA_Assets.hpp	Definition of asset ids to be used in software interfaces that require component or executable identifiers.

The ECOA infrastructure is responsible for allocating the appropriate Containers and Component objects; a pointer to the Container object shall be stored in the Component public member variable 'Container' by the infrastructure. This pointer to the Container shall remain valid while the Component Implementation object is active.

Templates for the files in Table 2 are provided in the following sections.

6.3 Component Interface Template

The following is a minimal Component Interface example:

- It defines all operations available to be invoked on a component,
- It declares a user context and a warm start context,
- It declares a pointer to a container instance, allowing to invoke the ECOA infrastructure.

```
/*
 * @file #component_impl_name#.hpp
 * Component Interface class header for Component #component_impl_name#
 * The user shall write this concrete class corresponding to the
 * Component Implementation itself.
 */

#if !defined(#COMPONENT_IMPL_NAME#_HPP)
#define #COMPONENT_IMPL_NAME#_HPP

/* Standard Types */
#include <ECOA.hpp>
/* Additionally created types */
#include #additionally_created_types#
/* Include container header */
#include "#component_impl_name#_container.hpp"
/* Include container types */
#include "#component_impl_name#_container_types.hpp"
/* Include user context */
#include "#component_impl_name#_user_context.hpp"

namespace #component_impl_name#
{

class Component
{
   public:

      void INITIALIZE__received();

      void START__received();
```

```
    void STOP__received();
```

```
/* CONDITIONAL BLOCK : present only if component attribute hasReset = true*/
   void RESET__received();
/* END OF CONDITIONAL BLOCK */
```

```
    void RESET__received();

    void SHUTDOWN__received();

    // the Component Implementation shall hold a Container pointer
    // which is passed within the constructor
    Container* container;

    // User data (which does not belong to the warm start context)
    // for this component implementation may be declared here within a standard
    // structure:
    user_context user;

    #if defined(OPTION_WARM_START_CONTEXT)
    // Optional Warm Start data for this component implementation may be
    // declared here as a single attribute named 'warm_start' which may be of
    // a user defined type.
       warm_start_context warm_start;
    #endif /* OPTION_SUPERVISION */

    // All operations for this Component implementation will be
    // declared as public concrete methods here

    // The following describes the Component API generated:
    /* Event operation handlers */
    #list_of_event_operations#

    /* Request-Response operation handlers */
    #list_of_request_response_operations#

   /* Versioned Data Notifying operation handlers */
    #list_of_versioned_data_notifying_operations#
```

```
/* CONDITIONAL BLOCK : present only if component is a Fault Handler
 *(isFaultHandler = true) */
#if defined(OPTION_FAULT_HANDLER)
```

```
#error_notification_operation_specifications#

#endif /* OPTION_FAULT_HANDLER */
```

```
/* CONDITIONAL BLOCK : present only if component is a supervisor
 *(componentKind = SUPERVISOR) */
#if defined(OPTION_SUPERVISION)

#on_state_change_operation_specifications#

#endif /* OPTION_SUPERVISION */

/* END OF CONDITIONAL BLOCK */
```

```
}; /* Component */

extern "C" {

   Component* #component_impl_name#__new_instance();

}

} /* #component_impl_name# */

#endif  /* #COMPONENT_IMPL_NAME#_HPP */
```

The inclusion of “extern C” at the end of the header file, above, avoids a static dependency between the generated code and the application code.

The following is an outline of a Component Implementation:

```
/*
 * @file #component_impl_name#.cpp
 * Component Implementation class for Component #component_impl_name#
 * The following code illustrates an example of a constructor method
 * and a Received Event entry-point
 */

namespace #component_impl_name#
{
```

```
extern "C" {

   Component* #component_impl_name#__new_instance()
   {
       return new Component();
   }
}

void Component::#operation_name#__received()
{
   /* To be implemented by the component */

   /* uses the container pointer to send an event called myDummyEvent
    * with no parameter
    */
   this->container->myDummyEvent__send();
   /*
    * …
    * increments a local user defined counter:
    */
   this->user.myCounter++;
}

} /* #component_impl_name# */
```

6.4 Container Interface Template

The following concrete class definition will define all container operations which a component can invoke.

```
/*
 * @file #component_impl_name#_container.hpp
 * Container Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */
#if !defined(#COMPONENT_IMPL_NAME#_CONTAINER_HPP)
#define #COMPONENT_IMPL_NAME#_CONTAINER_HPP

/* Standard Types */
#include <ECOA.hpp>
/* Additionally created types */
#include #additionally_created_types#
/* Container Types */
#include "#component_impl_name#_container_types.hpp"
```

```
namespace #component_impl_name#
{

class Container
{
   public:

      /* Logging and fault management services API */
      void log_trace
         (const ECOA::log &log);

      void log_debug
         (const ECOA::log &log);

      void log_info
         (const ECOA::log &log);

      void log_warning
         (const ECOA::log &log);

      void raise_error
         (const ECOA::log &log);

      void raise_fatal_error
         (const ECOA::log &log);

      #if defined(BINDING_OPTION_FLEX_LOG)

      #flex_log_call_specifications#
      #flex_raise_fatal_error_call_specifications#

      #endif /* BINDING_OPTION_FLEX_LOG */

      /* Time services API (generated on request) */
```

```
/* CONDITIONAL BLOCK : present only if component attribute
needsLocalTime = true */
      void get_relative_local_time
         (ECOA::hr_time &relative_local_time);
/* END OF CONDITIONAL BLOCK */

/* CONDITIONAL BLOCK : present only if component attribute
needsUTCTime = true */
```

```
      #if defined(OPTION_UTC_TIME)

      ECOA::return_status get_UTC_time (ECOA::global_time &utc_time);
      #endif /* OPTION_UTC_TIME */
/* END OF CONDITIONAL BLOCK */

/* CONDITIONAL BLOCK : present only if component attribute
needsSystemTime = true */
      ECOA::return_status get_absolute_system_time
         (ECOA::global_time &absolute_system_time);
/* END OF CONDITIONAL BLOCK */
```

```
   /* Time resolution services API (generated on request) */
```

```
/* CONDITIONAL BLOCK : present only if component attribute
needsTimeResolution = true */

#if defined(BINDING_OPTION_TIME_RESOLUTION)
```

```
/* SUB-CONDITIONAL BLOCK : present only if component attribute
needsLocalTime = true */
      void get_relative_local_time_resolution
         (ECOA::duration &relative_local_time_resolution);
/* END OF SUB-CONDITIONAL BLOCK */

/* SUB-CONDITIONAL BLOCK : present only if component attribute
needsUTCTime = true */
       #if defined(UTC_TIME)
       void get_UTC_time_resolution (ECOA::duration &utc_time_resolution);
       #endif /* OPTION_UTC_TIME */
/* END OF SUB-CONDITIONAL BLOCK */

/* SUB-CONDITIONAL BLOCK : present only if component attribute
needsSystemTime = true */
      void get_absolute_system_time_resolution
         (ECOA::duration &absolute_system_time_resolution);
/* END OF SUB-CONDITIONAL BLOCK */
```

```
#endif /* BINDING_OPTION_TIME_RESOLUTION */

/* END OF CONDITIONAL BLOCK */
```

```
    /* Optional API for saving the warm start context */
```

```
/* CONDITIONAL BLOCK : present only if component attribute
hasWarmStartContext = true */
      #if defined(OPTION_WARM_START_CONTEXT)
      void save_warm_start_context();
      #endif /* OPTION_WARM_START_CONTEXT */
/* END OF CONDITIONAL BLOCK */
```

```
    // All the operations for this Container interface will be declared as
    // public concrete methods here in the order that the container operations
    // are defined in the XML
```

```
    /* Triggers call specifications */
    #trigger_operations_call_specifications#

    /* Event operation call specifications */
    #event_operation_call_specifications#

    /* Request-response call specifications */
    #MAX_CONCURRENT_REQUESTS_specifications#
    #request_sync_call_specifications#
    #request_async_call_specifications#
    #response_send_call_specifications#

    #if defined(BINDING_OPTION_REQUEST_CANCEL)
    #request_cancel_call_specifications#
    #endif /* BINDING_OPTION_REQUEST_CANCEL */

    /* Versioned data call specifications */
    #get_read_access_call_specifications#
    #release_read_access_call_specifications#
    #get_write_access_call_specifications

    #if defined(BINDING_OPTION_SELECTED_WRITE_ACCESS)
    #get_selected_write_access_call_specifications#
    #endif /* BINDING_OPTION_SELECTED_WRITE_ACCESS */

    #cancel_write_access_call_specifications#
```

```
    #publish_write_access_call_specifications#
```

```
    #if defined(BINDING_OPTION_EXTENDED_VD_API)
    #get_is_initialized_call_specifications#
    #release_all_data_handles_call_specifications#
    #endif /* BINDING_OPTION_EXTENDED_VD_API */

    /* Functional parameters call specifications */
    #properties_call_specifications#

    // PINFO APIs
    #PINFO_read_call_specifications#
    #PINFO_seek_call_specifications#

    #if defined(OPTION_PINFO_WRITE)
    #PINFO_write_call_specifications#
    #endif /* OPTION PINFO_WRITE */
```

```
/* CONDITIONAL BLOCK : present only if component is a supervisor
 *(componentKind = SUPERVISOR) */

      #if defined(OPTION_SUPERVISION)

      /* Supervision call specifications */
      #supervision_of_executables_call_specifications#
      #supervision_of_components_call_specifications#
      #supervision_variables_call_specifications#

       #endif /* OPTION_SUPERVISION */

/* END OF CONDITIONAL BLOCK */
```

```
    // If this is a Fault Handler component then an additional API
    // may be defined according to ECOA available extensions

    // Other container technical data will accessible through the incomplete
    // structure defined here:
    struct platform_hook;

    // The constructor of the Container shall have the following signature:
    Container(platform_hook* hook);
```

```
   private:

      // private data for this container implementation is declared as a
      // private struct within the implementation
      platform_hook *hook;

}; /* Container */

} /* #component_impl_name# */

#endif  /* #COMPONENT_IMPL_NAME#_CONTAINER_HPP */
```

The Container Interface defines an incomplete structure, the platform_hook, which is defined privately by the Container Implementation. This platform_hook holds infrastructure-level technical data (which is implementation dependant).

In the rest of the document, the C++ bindings corresponding to the operations are presented for the Component Interface and for the Container Interface.

The Container of a given Component, which shall implement all methods specific to that Component, is implemented by a concrete class (Container Implementation), which is not specified in this document since it is platform specific.

6.5 Container Types Template

The following header file will define all container data types which the container and the component can use.

The specification for these data types is provided in Section 11.8.

```
/*
 * @file #component_impl_name#_container_types.hpp
 * Container Types for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

#if !defined(#COMPONENT_IMPL_NAME#_CONTAINER_TYPES_HPP)
#define #COMPONENT_IMPL_NAME#_CONTAINER_TYPES_HPP

#include <ECOA.hpp>

namespace #component_impl_name# {

// The following describes the data types generated with regard to APIs:
// * For any Versioned Data Read Access: data_handle
// * For any Versioned Data Write Access: data_handle
```

```
} /* #component_impl_name# */

#endif  /* #COMPONENT_IMPL_NAME#_CONTAINER_TYPES_HPP */
```

6.6 User Component Context Template

The following header file will define all user data types for the user context and the optional warm start context.

The Component User Context header file is necessary as the user context is mandatory.

It includes the Container Types header file in order for instance to allow declaring versioned data handles in the user context (to be able to save them from one component entry point execution to another).

The following shows the C++ syntax for defining the Component User Context (including an example data item; myCounter) and the Component Warm Start Context (including an example data item myData and validity flag warm_start_valid).

```
/*
 * @file #component_impl_name#_user_context.hpp
 * User defined data types for user context and warm start context
 * for Component #component_impl_name#
 * This is an example of a user defined Component User context
 */

#if !defined(#COMPONENT_IMPL_NAME#_USER_CONTEXT_HPP)
#define #COMPONENT_IMPL_NAME#_USER_CONTEXT_HPP

/* Standard Types */
#include <ECOA.hpp>
/* Additionally created types */
#include #additionally_created_types#
/* Container Types */
#include "#component_impl_name#_container_types.hpp"

namespace #component_impl_name#
{

// User Component Context structure example
typedef struct
{
   // declare the Component User Context "local" data here
   unsigned int myCounter;

} user_context;
```

```
// Component Warm Start Context structure example
```

```
/* CONDITIONAL BLOCK : present only if component attribute
hasWarmStartContext = true */

#if defined (OPTION_WARM_START_CONTEXT)

typedef struct
{
   /* declare the Component Warm Start Context data here */
   ECOA::boolean8 warm_start_valid;
   unsigned long myData;
} warm_start_context;

#endif /* OPTION_WARM_START_CONTEXT */
/* END OF CONDITIONAL BLOCK */
```

```
} /* #component_impl_name# */

#endif /* #COMPONENT_IMPL_NAME#_USER_CONTEXT_HPP */
```

6.7 Guards

In C++ the declarations in the header files shall be surrounded within the following block to avoid multiple inclusions:

```
#if !defined(#macro_protection_name#_HPP)
#define #macro_protection_name#_HPP

/* all the declarations shall come here */

#endif  /* #macro_protection_name#_HPP */
```

Where #macro_protection_name# is the name of the header file in capital letters and without the .hpp extension.

7 Parameters

This section describes the manner in which parameters are passed in C++:

- Input parameters defined with a simple type are passed by value, output parameters defined with a simple type are passed by reference,
- Input parameters defined with a complex type are passed as a reference to a const; output parameters defined with a complex type are passed by reference.

Table 3 Method of Passing Parameters

	Input parameter	**Output parameter**
Simple type	By value	Reference
Complex type	Reference to Const	Reference

Within the API bindings, parameters will be passed as constant if the behaviour of the specific API warrants it. This will override the normal conventions defined above.

8 Component Context

8.1 Component User Context

In C++, the Component Context is a structure which holds the user local data (called “Component User Context” and “Component Warm Start Context”). User context feature is always defined while warm start context feature may be optionally selected in Component Type declarations using metamodel attributes. The presence or absence of declarations of corresponding fields in Component code must be in accordance with selections made in the Component Type declaration. These structures may be declared in the Component Interface as member variables within the Component Implementation class (one public variable for the user context and one public variable for the warm start context).

Additionally a pointer to the Container object is also stored as a public member variable within the Component Implementation class. This is required in order to enable the Component Instance object to call the methods of the Container object. The pointer to the Container object is assigned by the Container.

Any language type can be used within the contexts (including ECOA ones).

NOTE The Container pointer and the User/Warm_Start Context are declared as public attributes of the component implementation class in order to be accessible to the container.

```
/*
 * @file #component_impl_name#.hpp
 * Component Interface class header for Component #component_impl_name#
 * The user shall write this concrete class corresponding to the
 * Component Implementation itself.
 */

#include "#component_impl_name#_user_context.hpp"

#include "#component_impl_name#_container.hpp"

namespace #component_impl_name#
{

class Component
{
   public:
      // the Component Implementation shall hold a Container pointer
      Container* container;

      // User data (which does not belong to the warm start context)
      // for this component implementation are declared here within a standard
      // structure:
      user_context user;

      // When the optional warm start context is used, the type warm_start_context

      // warm_start shall be defined by the user
```

```
      // in the #component_impl_name#_user_context.hpp file
      // to carry the component non-volatile data
      warm_start_context warm_start;

      // all the operations for this Component implementation will be
      // declared as public concrete methods here

}; /* Component */

extern "C" {

   Component* #component_impl_name#__new_instance();

}

} /* #component_impl_name# */
```

Data declared within the Component User Context and the Component Warm Start Context can be of any type.

9 Types

This section describes the convention for creating type libraries, and how the ECOA basic types and derived types are represented in C++.

9.1 Libraries

The type definitions are contained within libraries: all types for specific libraries defined in `#library#.types.xml` shall be placed in a file called `#library#.hpp`

Below is an example of a simple type being defined within a namespace in C++.

```
/*
 * @file #library#.hpp
 * Data-type declaration file
 * Generated automatically from specification; do not modify here
 */

namespace #library# {
//…
     typedef #basic_type_name# #simple_type_name#;
//…
} /* #library#  */
```

9.2 Basic Types

Basic types in C++ shall be located in the “ECOA” library and hence in `ECOA.hpp`.

Table 4 C++ Basic Type Mapping

ECOA Basic Type	C++ type
ECOA:boolean8	ECOA::boolean8
ECOA:int8	ECOA::int8
ECOA:char8	ECOA::char8
ECOA:int16	ECOA::int16
ECOA:int32	ECOA::int32
ECOA:int64 (optional)	ECOA::int64
ECOA:uint8	ECOA::uint8
ECOA:byte	ECOA::byte
ECOA:uint16	ECOA::uint16
ECOA:uint32	ECOA::uint32
ECOA:uint64 (optional)	ECOA::uint64
ECOA:float32	ECOA::float32
ECOA:double64	ECOA::double64

The data-types in Table 4 are fully defined using the predefined constants shown in Table 5:

Table 5 C++ Predefined Constants

C++ Basic Type	C++ constant
ECOA::boolean8	ECOA::TRUE ECOA::FALSE
ECOA::int8	ECOA::INT8_MIN ECOA::INT8_MAX
ECOA::char8	ECOA::CHAR8_MIN ECOA::CHAR8_MAX
ECOA::byte	ECOA::BYTE_MIN ECOA::BYTE_MAX
ECOA::int16	ECOA::INT16_MIN ECOA::INT16_MAX
ECOA::int32	ECOA::INT32_MIN ECOA::INT32_MAX
ECOA::int64 (optional)	ECOA::INT64_MIN ECOA::INT64_MAX
ECOA::uint8	ECOA::UINT8_MIN ECOA::UINT8_MAX
ECOA::uint16	ECOA::UINT16_MIN ECOA::UINT16_MAX
ECOA::uint32	ECOA::UINT32_MIN ECOA::UINT32_MAX
ECOA::uint64 (optional)	ECOA::UINT64_MIN ECOA::UINT64_MAX
ECOA::float32	ECOA::FLOAT32_MIN ECOA::FLOAT32_MAX
ECOA::double64	ECOA::DOUBLE64_MIN ECOA::DOUBLE64_MAX

The data types described in the following sections are also defined in the ECOA library.

9.3 Derived Types

9.3.1 Simple Types

The syntax for defining a Simple Type `#simple_type_name#` refined from a Basic Type `#basic_type_name#` in C++ is defined below.

```
typedef #basic_type_name# #simple_type_name#;
```

The syntax for defining a Simple Type `#simple_type_name_one#` refined from another Simple Type `#simple_type_name_two#` in C++ is defined below.

```
typedef #simple_type_name_two# #simple_type_name_one#;
```

`minRange` or `maxRange` constant definitions are optional. Where required they shall be provided after the type definition as follows:

```
static const #basic_type_name# #complete_simple_type_name#_minRange =
  #minrange_value#;
static const #basic_type_name# #complete_simple_type_name#_maxRange =
  #maxrange_value#;
```

`minRange` or `maxRange` can also reference existing constants (see Section **Erreur ! Source du renvoi introuvable.**) as follows:

```
static const #basic_type_name# #complete_simple_type_name#_minRange =
  #constant_name_one#;
static const #basic_type_name# #complete_simple_type_name#_maxRange =
  #constant_name_two#;
```

9.3.2 Enumerations

The C++ syntax for defining an enumerated type named #enum_type_name#, with a set of labels name from `#enum_value_name_1#` to `#enum_value_name_n#` and a set of optional values named `#enum_value_value_1#` … `#enum_value_value_n#`, the syntax is defined below.

```
struct #enum_type_name#
{
   #basic_type_name# value;
   enum  EnumValues {
      #enum_value_name_1# = #enum_value_value_1#,
      #enum_value_name_2# = #enum_value_value_2#,
      #enum_value_name_3# = #enum_value_value_3#,
      #enum_value_name_4# = #enum_value_value_4#,
      //…
      #enum_value_name_n# = #enum_value_value_n#
   };
   inline void operator = (#basic_type_name# i) { value = i; }
   inline operator #basic_type_name#() const { return value; }
   inline #enum_type_name#(EnumValues v):value(v) {}
```

```
   inline #enum_type_name#():value(#enum_value_name_1#) {}
};
```

Where:

- `#basic_type_name#` is `ECOA::boolean8`, `ECOA::int8`, `ECOA::char8`, `ECOA::byte`, `ECOA::int16`, `ECOA::int32`, `ECOA::int64`, `ECOA::uint8`, `ECOA::uint16`, `ECOA::uint32` or `ECOA::uint64`.
- `#enum_value_name_X#` is the name of a label
- `#enum_value_value_X#` is the optional value of a label
- `#enum_value_value_X#` is the optional value of the label. If not set, this value is computed from the previous label value, by adding 1 (or set to 0 if it is the first label of the enumeration).

9.3.3 Records

The syntax for a record type named `#record_type_name#` with a set of fields named `#field_name1#` to `#field_namen#` of given types `#data_type_1#` to `#data_type_n#` is given below.

The order of fields in the struct shall follow the order of fields used in the XML definition.

```
typedef struct
{
   #data_type_1# #field_name1#;
   #data_type_2# #field_name2#;
   //…
   #data_type_n# #field_namen#;
} #record_type_name#;
```

9.3.4 Variant Records

The syntax for a Variant Record named `#variant_record_type_name#` containing a set of fields (named `#field_name1#` to `#field_namen#`) of given types `#data_type_1#` to `#data_type_n#` and other optional fields (named `#optional_field_name1#` to `#optional_field_namen#`) of type (`#optional_type_name1#` to `#optional_type_namen#`) with selector `#selector_name#` is given below.

The order of fields in the struct shall follow the order of fields used in the XML definition.

```
/*
 *  #selector_type_name# can be of any simple basic type, or an enumeration
 */

typedef struct{

   #selector_type_name# #selector_name#;

   #data_type_1# #field_name1#; /* for each <field> element */
   #data_type_2# #field_name2#;
   //…
```

```
   #data_type_n# #field_namen#;

   union  {
      #optional_type_name1# #optional_field_name1#; /* for each <union> element */
      #optional_type_name2# #optional_field_name2#;
      //…
      #optional_type_namen# #optional_field_namen#;
   } u_#selector_name#;

} # variant_record_type_name#;
```

9.3.5 Fixed Arrays

The C++ syntax for a fixed array named `#array_type_name#` of maximum size `#max_number#` and element type of `#data_type_name#` is given below.

A constant called `#array_type_name#_MAXSIZE` is defined to specify the size of the array.

```
const ECOA::uint32 #array_type_name#_MAXSIZE = #max_number#;
typedef #data_type_name# #array_type_name#[#array_type_name#_MAXSIZE];
```

9.3.6 Variable Arrays

The C++ syntax for a variable array (named `#var_array_type_name#`) with maximum size `#max_number#`, elements with type `#data_type_name#` and a current size of current_size is given below.

```
const ECOA::uint32 #var_array_type_name#_MAXSIZE = #max_number#;
typedef struct {
   ECOA::uint32 current_size;
   #data_type_name# data[#var_array_type_name#_MAXSIZE];
} #var_array_type_name#;
```

9.4 Predefined Types

9.4.1 Function execution return_status

In C++ `ECOA:return_status` translates to `ECOA::return_status`, with the enumerated values shown below:

```
namespace ECOA {

   struct return_status
   {
      ECOA::uint32 value;
```

```
      enum EnumValues      {
         OK                          = 0,
         FAILURE                     = 1,
         INVALID_HANDLE              = 2,
         DATA_NOT_INITIALIZED        = 3,
         NO_DATA                     = 4,
         INVALID_IDENTIFIER          = 5,
         NO_RESPONSE                 = 6,
         OPERATION_ALREADY_PENDING   = 7,
         CLOCK_UNSYNCHRONIZED        = 8,
         RESOURCE_NOT_AVAILABLE      = 9,
         OPERATION_NOT_AVAILABLE     = 10,
         INVALID_PARAMETER           = 11
      };
      inline void operator = (ECOA::uint32 i) { value = i; }
      inline operator ECOA::uint32 () const { return value; }
      inline return_status (EnumValues v):value(v) {}
      inline return_status ():value(OK) {}
   };

} /* ECOA */
```

All Component or Container interfaces defined in the following sections, and offering a return status, shall at least manage values `ECOA::return_status::OK` and `ECOA::return_status::FAILURE`.

`ECOA::return_status::FAILURE` is the default return status applied when the status is not `ECOA::return_status::OK`.

9.4.2 Component and executable identifiers

In C++ the syntax for a `ECOA:asset_id` is:

```
typedef ECOA::uint32 asset_id;
```

In C++ the `ECOA:asset_id` definitions will be generated as constants declared in a file named ECOA_Assets.hpp using the following syntax:

```
/* File ECOA_Assets.hpp */

#include <ECOA.hpp>

#if !defined(ECOA_ASSETS_HPP)
#define ECOA_ASSETS_HPP
namespace ECOA_Assets {

      static const ECOA::asset_id CMP_#component_instance_name1# = #CMP_ID1#;
```

```
        static const ECOA::asset_id CMP_#component_instance_name2# = #CMP_ID2#;
        static const ECOA::asset_id CMP_#component_instance_nameN# = #CMP_IDN#;

        static const ECOA::asset_id EXE_#executable_name1# = #EXE_ID1#;
        static const ECOA::asset_id EXE_#executable_name2# = #EXE_ID2#;
        static const ECOA::asset_id EXE_#executable_nameN# = #EXE_IDN#;
}
#endif
```

9.4.3 Write access mode

In C++ `ECOA:write_access_mode` translates to `ECOA::write_access_mode`, with the enumerated values shown below:

```
struct write_access_mode
{
   ECOA::uint32 value;
   enum EnumValues
   {
      READ_AND_UPDATE   = 0,
      WRITE_ONLY        = 1
   };
   inline void operator = (ECOA::uint32 i) { value = i; }
   inline operator ECOA::uint32() const { return value; }
   inline write_access_mode (EnumValues v):value(v) {}
   inline write_access_mode ():value(READ_AND_UPDATE) {}
};
```

9.4.4 Time management

9.4.4.1 ECOA:hr_time

The binding for hr_time is:

```
namespace ECOA {

   typedef struct
   {
      ECOA::uint32 seconds;     /* Seconds */
      ECOA::uint32 nanoseconds; /* Nanoseconds*/
   } hr_time;

} /* ECOA */
```

9.4.4.2 ECOA:global_time

Global time is represented as:

```
namespace ECOA {

   typedef struct
   {
      ECOA::uint32 seconds;     /* Seconds */
      ECOA::uint32 nanoseconds; /* Nanoseconds*/
   } global_time;

} /* ECOA */
```

9.4.4.3 ECOA:duration

Duration is represented as:

```
namespace ECOA {

   typedef struct
   {
      ECOA::uint32 seconds;     /* Seconds */
      ECOA::uint32 nanoseconds; /* Nanoseconds*/
   } duration;

} /* ECOA */
```

9.4.5 Logs

9.4.5.1 ECOA:log

The syntax for a log in C++ is:

```
namespace ECOA {

   const ECOA::uint32 LOG_MAXSIZE = 256;

   typedef struct {
      ECOA::uint32 current_size;
      ECOA::char8  data[ECOA::LOG_MAXSIZE];
   } log;

} /* ECOA */
```

9.4.5.2 ECOA:information_category

The C++ syntax for this type is:

```
namespace ECOA {

struct information_category
{
   ECOA::uint32 value;
   enum EnumValues
   {
      NONE            = 0,
      CRITICAL        = 1,
      ERROR           = 1,
      WARNING         = 3,
      INFO            = 4,
      DEBUG           = 5,
      TRACE           = 6
   };
   inline void operator = (ECOA::uint32 i) { value = i; }
   inline operator ECOA::uint32() const { return value; }
   inline information_category  (EnumValues v):value(v) {}
   inline information_category  ():value(NONE) {}
};

} /* ECOA */
```

9.4.6 Error management

9.4.6.1 ECOA:error_id

In C++ the syntax for an `ECOA:error_id` is:

```
typedef ECOA::uint32 error_id;
```

9.4.6.2 ECOA:error_code

In C++ the syntax for an `ECOA:error_code` is:

```
typedef ECOA::uint32 error_code;
```

9.4.6.3 ECOA:asset_type

In C++ `ECOA:asset_type` translates to `ECOA::asset_type`, with the enumerated values shown below:

```
namespace ECOA {
```

```
struct asset_type
{
   ECOA::uint32 value;
   enum EnumValues
   {
      COMPONENT          = 0,
      EXECUTABLE         = 1
   };
   inline void operator = (ECOA::uint32 i) { value = i; }
   inline operator ECOA::uint32() const { return value; }
   inline asset_type (EnumValues v):value(v) {}
   inline asset_type ():value(COMPONENT) {}
};

} /* ECOA */
```

9.4.6.4 ECOA:error_type

In C++ `ECOA:error_type` translates to `ECOA::error_type`, with the enumerated values shown below:

```
namespace ECOA {

struct error_type
{
   ECOA::uint32 value;
   enum EnumValues
   {
      RESOURCE_NOT_AVAILABLE = 0,
      UNAVAILABLE            = 1,
      MEMORY_VIOLATION       = 2,
      NUMERICAL_ERROR        = 3,
      ILLEGAL_INSTRUCTION    = 4,
      STACK_OVERFLOW         = 5,
      DEADLINE_VIOLATION     = 6,
      OVERFLOW               = 7,
      UNDERFLOW              = 8,
      ILLEGAL_INPUT_ARGS     = 9,
      ILLEGAL_OUTPUT_ARGS    = 10,
      ERROR                  = 11,
      FATAL_ERROR            = 12,
      HARDWARE_FAULT         = 13,
      POWER_FAIL             = 14,
      COMMUNICATION_ERROR    = 15,
      INVALID_CONFIG         = 16,
      INITIALISATION_PROBLEM = 17,
```

```
      CLOCK_UNSYNCHRONIZED    = 18
   };

   inline void operator = (ECOA::uint32 i) { value = i; }
   inline operator ECOA::uint32() const { return value; }
   inline error_type (EnumValues v):value(v) {}
   inline error_type ():value(RESOURCE_NOT_AVAILABLE) {}
};

} /* ECOA */
```

9.4.7 Pinfo management

9.4.7.1 ECOA:pinfo_filename

The syntax for a pinfo_filename in C++ is:

```
namespace ECOA {

   const ECOA::uint32 PINFO_FILENAME_MAXSIZE = 256;

   typedef struct
   {
      ECOA::uint32 current_size;
      ECOA::char8  data[ECOA::PINFO_FILENAME_MAXSIZE];
   } pinfo_filename;

} /* ECOA */
```

9.4.7.2 ECOA:seek_whence_type

In C++ `ECOA:seek_whence_type` translates to `ECOA::seek_whence_type`, with the enumerated values shown below:

```
namespace ECOA {

struct seek_whence_type
{
   ECOA::uint32 value;
   enum EnumValues
   {
      SEEK_SET = 0,
      SEEK_CUR = 1,
      SEEK_END = 2
   };
```

```
   inline void operator = (ECOA::uint32 i) { value = i; }
   inline operator ECOA::uint32() const { return value; }
   inline seek_whence_type (EnumValues v):value(v) {}
   inline seek_whence_type ():value(SEEK_SET) {}
};

}/* ECOA */
```

9.4.8 Lifecycle management

9.4.8.1 ECOA:component_state

The C++ syntax for this type is:

```
namespace ECOA {

struct component_state
{
   ECOA::uint32 value;
   enum EnumValues
   {
      UNAVAILABLE = 0,
      IDLE = 1,
      READY = 2,
      RUNNING = 3
   };
   inline void operator = (ECOA::uint32 i) { value = i; }
   inline operator ECOA::uint32() const { return value; }
   inline component_state (EnumValues v):value(v) {}
   inline component_state ():value(UNAVAILABLE) {}
};

} /* ECOA */
```

9.4.8.2 ECOA:component_command

The C++ syntax for this type is:

```
namespace ECOA {

struct component_command
{
   ECOA::uint32 value;
   enum EnumValues
   {
```

```
      INIT = 0,
      START = 1,
      STOP = 2,
      RESET = 3,
      SHUTDOWN = 4,
      KILL = 5
   };
   inline void operator = (ECOA::uint32 i) { value = i; }
   inline operator ECOA::uint32() const { return value; }
   inline component_command (EnumValues v):value(v) {}
   inline component_command ():value(INIT) {}
};

}/* ECOA */
```

9.4.8.3 ECOA:executable_state

The C++ syntax for this type is:

```
namespace ECOA {

struct executable_state
{
   ECOA::uint32 value;
   enum EnumValues
   {
      NOT_LAUNCHED = 0,
      LAUNCHED     = 1
   };
   inline void operator = (ECOA::uint32 i) { value = i; }
   inline operator ECOA::uint32() const { return value; }
   inline executable_state (EnumValues v):value(v) {}
   inline executable_state ():value(NOT_LAUNCHED) {}
};

} /* ECOA */
```

9.4.8.4 ECOA:executable_command

The C++ syntax for this type is:

```
namespace ECOA {

struct executable_command
{
   ECOA::uint32 value;
```

```
   enum EnumValues
   {
      START = 0,
      STOP  = 1
   };
   inline void operator = (ECOA::uint32 i) { value = i; }
   inline operator ECOA::uint32() const { return value; }
   inline executable_command (EnumValues v):value(v) {}
   inline executable_command ():value(START) {}
};

} /* ECOA */
```

9.5 Constants

The syntax for declaring a Constant called `#constant_name#` of a type `#type_name#` in C++ is:

```
static const #type_name# #constant_name# = #constant_value#;
```

where `#constant_value#` is either an integer or a floating-point value as required by the XML description.

where `#type_name#` can be a Basic Type or a Simple Type.

9.6 Predefined constants

The constant `#component_impl_name#_#operation_name#_MAX_CONCURRENT_REQUESTS` is implemented in the present language binding with the following C language syntax:

```
static const #component_impl_name#::#operation_name#_SERVICE_MAXDEFERRED
#max_concurrent_requests#
```

It is defined in `#component_impl_name#.hpp` file.

9.7 Functions defined on types

9.7.1 Initialization functions

Each type defined in a library has an initialization function conforming to the following declaration:

```
void #library#::#type_name#_initialize (#library#_#type_name# *value);
```

This method initializes 'value' with a default value, which is defined as:

- For boolean types, the default value is false.
- For scalar types, the default value is 0.
- For enumerated types, the default value is the first value.
- For record types, the default value is made of the default value of each field.
- For variant record types, the default value is made of the default value of each field, and the selector has the default value of its type.

- For fixed array types, the default value is made of the default value of each element.
- For array types, the default value is the empty array.

9.7.2 Comparison functions

Each type defined in a library has a comparison function conforming to the following declaration:

```
ECOA::boolean8 #library#::#type_name#_equals (const #library#_#type_name# *value1,
const #library#_#type_name# *value2);
```

Comparison of values are made using a recursive approach for complex types, until comparing each leaf type (which is then a simple type) composing value1 and value2. The comparison function returns ECOA::TRUE if and only if all comparison functions applied on leaf types return ECOA::TRUE.

For simple types:

- Case of floating values : the comparison function returns ECOA::TRUE when value2 = value1±epsilon.
- All other cases : the comparison function returns ECOA::TRUE when values are strictly equal.

A default epsilon floating value shall be defined in the platform implementation.

It shall be possible to set a different epsilon value for each simple floating type using a dedicated user file: `epsilon_definition.hpp`.

This file is not generated by the ECOA platform: the user has to create it when required, in accordance with the following syntax:

```
/* @file epsilon_definition.hpp
 * This is an example of user defined epsilon values for comparison between
 * simple floating types
 */

#define #library#_#type_name#_epsilon #value#
```

10 Component Interface

This section contains details of the operations that comprise the component API i.e. the operations that can invoked by the container on a component.

10.1 Operations

10.1.1 Request-Response

The following is the C++ syntax for an operation used by the container to invoke a request received to a component instance when a response is required. The same syntax is applicable for both synchronous and asynchronous request-response operations.

10.1.1.1.1 Request Received when immediate=false

```
/*
 * @file #component_impl_name#.hpp
 * Component Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

namespace #component_impl_name#
{

class Component
{
   public:

      void #operation_name#__request_received
         (const ECOA::uint32 ID,
          const #request_parameters#);

}; /* Component */

} /* #component_impl_name# */
```

10.1.1.1.2 Request Received when immediate=true

```
/*
 * @file #component_impl_name#.hpp
 * Component Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

namespace #component_impl_name#
{

class Component
{
   public:

      void #operation_name#__request_received
         (const ECOA::uint32 ID,
          const #request_parameters#,
          #response_parameters#);

}; /* Component */
```

```
} /* #component_impl_name# */
```

10.1.1.2 End of an asynchronous Request

10.1.1.2.1 Response Received

The following is the C++ syntax for an operation used by the container to send the response to an asynchronous request response operation to the component instance that originally issued the request. (The reply to a synchronous request response is provided by the return of the original request).

```
/*
 * @file #component_impl_name#.hpp
 * Component Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

namespace #component_impl_name#
{

class Component
{
   public:

      void #operation_name#__response_received
         (const ECOA::uint32 ID,
          const ECOA::return_status status,
          const #response_parameters#) ;

}; /* Component */

} /* #component_impl_name# */
```

The "#response_parameters#" are the "out" parameters of the request-response operation, but are treated as inputs to the function and passed as "const" parameters, so they are not modified by the component.

The following return status values shall be managed:

- ECOA::return_status::OK
- ECOA::return_status::NO_RESPONSE
- ECOA::return_status::FAILURE

10.1.1.2.2 Optional Alternative

10.1.1.2.2.1 Response_Actually_Received

Not available in this binding.

10.1.1.2.2.2 Response_Not_Received

Not available in this binding.

10.1.2 Versioned Data Updated

The following is the C++ syntax that is used by the container to inform a component instance, which reads an item of versioned data, that new data has been written.

```
/*
 * @file #component_impl_name#.hpp
 * Component Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

namespace #component_impl_name#
{

class Component
{
   public:

      void #operation_name#__updated();

}; /* Component */

} /* #component_impl_name# */
```

10.1.3 Event Received

The following is the C++ syntax for an event received by a component instance.

```
/*
 * @file #component_impl_name#.hpp
 * Component Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

namespace #component_impl_name#
{

class Component
```

```
{
   public:

      void #operation_name#__received
         (const #event_parameters#);

}; /* Component */

} /* #component_impl_name# */
```

10.2 Component Lifecycle

The following C++ methods are applicable to application component instances to change their (lifecycle) state:

```
/*
 * @file #component_impl_name#.hpp
 * Component Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

namespace #component_impl_name#
{

class Component
{
   public:

      void INITIALIZE__received() ;

      void START__received();

      void STOP__received();

      void SHUTDOWN__received();

      void RESET__received();

}; /* Component */

} /* #component_impl_name# */
```

10.3 Supervisor components

The following C++ method is applicable to application component instances of kind SUPERVISOR, to be informed of lifecycle state changes of component instances of the application.

```
/*
 * @file #component_impl_name#.hpp
 * Component Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

namespace #component_impl_name#
{

class Component
{
   public:

      #if defined(OPTION_SUPERVISION)

      void on_state_change (
        const ECOA::asset_id componentInstanceId,
        const ECOA::component_state state,
        const ECOA::component_state previous_state);

      #endif /* OPTION_SUPERVISION */


}; /* Component */

} /* #component_impl_name# */
```

10.4 Error_notification for fault handler components

The C++ syntax for the container to report an error to a fault handler component is:

```
/*
 * @file #component_impl_name#.hpp
 * Component Interface class header for Fault handler #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

namespace #component_impl_name#
{

class Component
{
```

```
   public:

      #if defined(OPTION_FAULT_HANDLER)

      void error_notification
         (ECOA::error_id error_id,
          const ECOA::global_time& timestamp,
          ECOA::asset_id asset_id,
          ECOA::asset_type asset_type,
          ECOA::error_type error_type,
          ECOA::error_code error_code);

      #endif /* OPTION_FAULT_HANDLER */

}; /* Component */

} /* # component_impl_name # */
```

11 Container Interface

This section contains details of the operations that comprise the container API i.e. the operations that can be called by a component.

11.1 Operations

11.1.1 Request Response

11.1.1.1 Synchronous Request

The C++ syntax for a component instance to perform a synchronous request response operation is:

```
/*
 * @file #component_impl_name#_container.hpp
 * Container Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

namespace #component_impl_name#
{

class Container
{
   public:

      ECOA::return_status #operation_name#__request_sync
         (const #request_parameters#,
```

```
          #response_parameters#);

}; /* Container */

} /* #component_impl_name# */
```

The following return status values shall be managed:

- ECOA::return_status::OK
- ECOA::return_status::NO_RESPONSE
- ECOA::return_status::INVALID_PARAMETER
- ECOA::return_status::FAILURE

11.1.1.2 Asynchronous Request

The C++ syntax for a component instance to perform an asynchronous request response operation is:

```
/*
 * @file #component_impl_name#_container.hpp
 * Container Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

#include "#component_impl_name#_container_types.hpp"

namespace #component_impl_name#
{

class Container
{
   public:

      ECOA::return_status #operation_name#__request_async
         (ECOA::uint32& ID,
          const #request_parameters#);

}; /* Container */

} /* #component_impl_name# */
```

The following return status values shall be managed:

- ECOA::return_status::OK
- ECOA::return_status::RESOURCE_NOT_AVAILABLE
- ECOA::return_status::INVALID_PARAMETER
- ECOA::return_status::FAILURE

11.1.1.3 Response Send

The C++ syntax, applicable to both synchronous and asynchronous request response operations, for sending a reply is:

```
/*
 * @file #component_impl_name#_container.hpp
 * Container Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

namespace #component_impl_name#
{

class Container
{
   public:

      ECOA::return_status #operation_name#__response_send
         (const ECOA::uint32 ID,
          const #response_parameters#);

}; /* Container */

} /* #component_impl_name# */
```

The "#response_parameters#[2]" are the "out" parameters of the request-response operation, but are treated as inputs to the function and passed as "const" parameters, so they are not modified by the container. The ID parameter is that which is passed in during the invocation of the request received operation.

The following return status values shall be managed:

- ECOA::return_status::OK
- ECOA::return_status::INVALID_IDENTIFIER
- ECOA::return_status::INVALID_PARAMETER
- ECOA::return_status::FAILURE

11.1.1.4 Request Cancel

If BINDING OPTION REQUEST CANCEL is available, the C++ syntax for a component instance to cancel the handling of a pending request (i.e. notify the infrastructure and the caller that no response will be sent for this request) is:

```
/*
 * @file #component_impl_name#_container.hpp
 * Container Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */
```

```
namespace #component_impl_name#
{

class Container
{
   public:

      #if defined(BINDING_OPTION_REQUEST_CANCEL)

      ECOA::return_status #operation_name#__request_cancel
         (const ECOA::uint32 ID);

      #endif /* BINDING_OPTION_REQUEST_CANCEL */

}; /* Container */

} /* #component_impl_name# */
```

The following return status values shall be managed:

- `ECOA::return_status::OK`
- `ECOA::return_status::FAILURE`

11.1.2 Versioned Data

This section contains the C++ syntax for versioned data operations, which allow a component instance to:

- Get (request) Read Access (mandatory)
- Release Read Access (mandatory)
- Get (request) Write Access (mandatory)
- Gest Selected Write Access (binding option)
- Cancel Write Access (without writing new data) (mandatory)
- Publish (write) new data (automatically releases write access) (mandatory)
- Know if a Data is Initialized (optional)
- Release All Data Handles (optional)

Note: the definition of versioned data handles involved in all `#operation_name#` is done in the Container Types header file, as specified in Section 12.1.

11.1.2.1 Get Read Access

```
/*
 * @file #component_impl_name#_container.hpp
 * Container Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
```

```
 */

#include "#component_impl_name#_container_types.hpp"

namespace #component_impl_name#
{

class Container
{
   public:

      ECOA::return_status #operation_name#__get_read_access
         (#operation_name#_handle& data_handle);
}; /* Container */

} /* #component_impl_name# */
```

The following return status values shall be managed:

- ECOA::return_status::OK
- ECOA::return_status::NO_DATA
- ECOA::return_status::INVALID_HANDLE
- ECOA::return_status::RESOURCE_NOT_AVAILABLE
- ECOA::return_status::FAILURE

11.1.2.2 Release Read Access

```
/*
 * @file #component_impl_name#_container.hpp
 * Container Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

#include "#component_impl_name#_container_types.hpp"

namespace #component_impl_name#
{

class Container
{
   public:

      ECOA::return_status #operation_name#__release_read_access
         (#operation_name#_handle& data_handle);

}; /* Container */
```

```
} /* #component_impl_name# */
```

The following return status values shall be managed:

- ECOA::return_status::OK
- ECOA::return_status::INVALID_HANDLE
- ECOA::return_status::FAILURE

11.1.2.3 Get Write Access

```
/*
 * @file #component_impl_name#_container.hpp
 * Container Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

#include "#component_impl_name#_container_types.hpp"

namespace #component_impl_name#
{
class Container
{
   public:

      ECOA::return_status #operation_name#__get_write_access
         (#operation_name#_handle& data_handle);

}; /* Container */

} /* #component_impl_name# */
```

The following return status values shall be managed:

- ECOA::return_status::OK
- ECOA::return_status::DATA_NOT_INITIALIZED
- ECOA::return_status::INVALID_HANDLE
- ECOA::return_status::RESOURCE_NOT_AVAILABLE
- ECOA::return_status::FAILURE

11.1.2.4 Get Selected Write Access

If BINDING OPTION SELECTED WRITE ACCESS is available, the C++ syntax for a component instance to get an optimized write access on a versioned data is:

```
/*
 * @file #component_impl_name#_container.hpp
 * Container Interface class header for Component #component_impl_name#
```

```
 * Generated automatically from specification; do not modify here
 */

#include "#component_impl_name#_container_types.hpp"

namespace #component_impl_name#
{
class Container
{
   public:

      #if defined(BINDING_OPTION_SELECTED_WRITE_ACCESS)

      ECOA::return_status #operation_name#__get_selected_write_access
         (#operation_name#_handle& data_handle,
          const ECOA::write_access_mode mode);

      #endif /* BINDING_OPTION_SELECTED_WRITE_ACCESS */

}; /* Container */

} /* #component_impl_name# */
```

The following return status values shall be managed when the function is defined:

- ECOA::return_status::OK
- ECOA::return_status::FAILURE

11.1.2.5 Cancel Write Access

```
/*
 * @file #component_impl_name#_container.hpp
 * Container Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

#include "#component_impl_name#_container_types.hpp"

namespace #component_impl_name#
{
class Container
{
   public:

      ECOA::return_status #operation_name#__cancel_write_access
```

```
        (#operation_name#_handle& data_handle);

}; /* Container */

} /* #component_impl_name# */
```

The following return status values shall be managed:

- ECOA::return_status::OK
- ECOA::return_status::INVALID_HANDLE
- ECOA::return_status::FAILURE

11.1.2.6 Publish Write Access

```
/*
 * @file #component_impl_name#_container.hpp
 * Container Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

#include "#component_impl_name#_container_types.hpp"

namespace #component_impl_name#
{

class Container
{
   public:

      ECOA::return_status #operation_name#__publish_write_access
         (#operation_name#_handle& data_handle);

}; /* Container */

} /* #component_impl_name# */
```

The following return status values shall be managed:

- ECOA::return_status::OK
- ECOA::return_status::INVALID_HANDLE
- ECOA::return_status::FAILURE

11.1.2.7 Is Initialized

If BINDING OPTION EXTENDED VD API is available, this function allows the component to know if a data has a value or not, i.e. if it has been initialized, either by a default value in the assembly, or by a write operation.

```
/*
```

```
 * @file #component_impl_name#_container.hpp
 * Container Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

#include "#component_impl_name#_container_types.hpp"

namespace #component_impl_name#
{

class Container
{
   public:

      #if defined(BINDING_OPTION_EXTENDED_VD_API)

      ECOA::boolean8 #operation_name#__is_initialized();

      #endif /* BINDING_OPTION_EXTENDED_VD_API */

}; /* Container */

} /* #component_impl_name# */
```

11.1.2.8 Release All Data Handles

If BINDING OPTION EXTENDED VD API is available, this function allows to release all data handles (read and write) obtained by the component. It can be used to simplify by ensuring at a given point in code that no handle is kept by the component.

```
/*
 * @file #component_impl_name#_container.hpp
 * Container Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

#include "#component_impl_name#_container_types.hpp"

namespace #component_impl_name#
{

class Container
{
   public:
```

```
      #if defined(BINDING_OPTION_EXTENDED_VD_API)

      void #operation_name#__release_all_data_handles();

      #endif /* BINDING_OPTION_EXTENDED_VD_API */

}; /* Container */

} /* #component_impl_name# */
```

11.1.3 Event

11.1.3.1 Send

The C++ syntax for a component instance to perform an event send operation is:

```
/*
 * @file #component_impl_name#_container.hpp
 * Container Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

namespace #component_impl_name#
{

class Container
{
   public:

      void #operation_name#__send
         (const #event_parameters#);

}; /* Container */

} /* #component_impl_name# */
```

11.2 Properties

11.2.1 Get Value

The syntax for Get_Value is shown below where:

- `#property_name#` is the name of the property used in the component definition,
- `#property_type_name#` is the name of the data-type of the property.

```
/*
```

```
 * @file #component_impl_name#_container.hpp
 * Container Interface class class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

namespace #component_impl_name#
{

class Container
{
   public:

      void get_#property_name#_value
         (#property_type_name#& value);

}; /* Container */

} /* #component_impl_name# */
```

11.2.2 Expressing Property Values

Not applicable to the C++ Binding.

11.2.3 Example of Defining and Using Properties

Not applicable to the C++ Binding.

11.3 Logging and Fault Management

11.3.1 Alternative 1: using fixed interfaces

This section describes the C++ syntax for the logging and fault management operations provided by the container. There are six operations:

- Trace: a detailed runtime trace to assist with debugging
- Debug: debug information
- Info: to log runtime events that are of interest e.g. changes of component state
- Warning: to report and log warnings
- Raise_Error: to report an error from which the application may be able to recover
- Raise_Fatal_Error: to raise a severe error from which the application cannot recover

```
/*
 * @file #component_impl_name#_container.hpp
 * Container Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */
```

```
namespace #component_impl_name#
{

class Container
{
   public:

      void log_trace
         (const ECOA::log& log);

      void log_debug
         (const ECOA::log &log);

      void log_info
         (const ECOA::log &log);

      void log_warning
         (const ECOA::log &log);

      void raise_error
         (const ECOA::log &log,
          const ECOA::error_code error_code);

      void raise_fatal_error
         (const ECOA::log &log,
          const ECOA::error_code error_code);

}; /* Container */

} /* #component_impl_name# */
```

11.3.2 Alternative 2: using flex interfaces

```
/*
 * @file #component_impl_name#_container.hpp
 * Container Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

namespace #component_impl_name#
{

class Container
```

```
{
   public:

      #if defined(BINDING_OPTION_FLEX_LOG)

      void flex_log
         (ECOA_information_category category,
          const ECOA::char8* log, ...);

      void flex_raise_fatal_error
         (ECOA_information_category category,
          const ECOA::char8* fatal_error_string, ...);

      #endif /* BINDING_OPTION_FLEX_LOG */


}; /* Container */

} /* #component_impl_name# */
```

11.4 Time Services

11.4.1 Get_Relative_Local_Time

```
/*
 * @file #component_impl_name#_container.hpp
 * Container Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

namespace #component_impl_name#
{

class Container
{
   public:

      void get_relative_local_time
         (ECOA::hr_time& relative_local_time);

}; /* Container */

} /* #component_impl_name# */
```

11.4.2 Get_UTC_Time

```
/*
 * @file #component_impl_name#_container.hpp
 * Container Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

namespace #component_impl_name#
{

class Container
{
   public:

      #if defined(OPTION_UTC_TIME)

      ECOA::return_status get_UTC_time
         (ECOA::global_time& utc_time);

      #endif /* OPTION_UTC_TIME */

}; /* Container */

} /* #component_impl_name# */
```

The following return status values shall be managed:

- ECOA::return_status::OK
- ECOA::return_status::CLOCK_UNSYNCHRONIZED
- ECOA::return_status::FAILURE

11.4.3 Get_Absolute_System_Time

```
/*
 * @file #component_impl_name#_container.hpp
 * Container Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

namespace #component_impl_name#
{

class Container
{
   public:
```

```
      ECOA::return_status get_absolute_system_time
         (ECOA::global_time& absolute_system_time);

}; /* Container */

} /* #component_impl_name# */
```

The following return status values shall be managed:

- ECOA::return_status::OK
- ECOA::return_status::CLOCK_UNSYNCHRONIZED
- ECOA::return_status::FAILURE

11.4.4 Get_Relative_Local_Time_Resolution

```
/*
 * @file #component_impl_name#_container.hpp
 * Container Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

namespace #component_impl_name#
{

class Container
{
   public:

      #if defined(OPTION_TIME_RESOLUTION)

      void get_relative_local_time_resolution
         (ECOA::duration& relative_local_time_resolution);

      #endif /* OPTION_TIME_RESOLUTION */

}; /* Container */

} /* #component_impl_name# */
```

11.4.5 Get_UTC_Time_Resolution

```
/*
 * @file #component_impl_name#_container.hpp
 * Container Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
```

```
 */

namespace #component_impl_name#
{

class Container
{
   public:

      #if defined(OPTION_TIME_RESOLUTION)

      void get_UTC_time_resolution
         (ECOA::duration& utc_time_resolution);

      #endif /* OPTION_TIME_RESOLUTION */

}; /* Container */

} /* #component_impl_name# */
```

11.4.6 Get_Absolute_System_Time_Resolution

```
/*
 * @file #component_impl_name#_container.hpp
 * Container Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

namespace #component_impl_name#
{

class Container
{
   public:

      #if defined(OPTION_TIME_RESOLUTION)

      void get_absolute_system_time_resolution
         (ECOA::duration& absolute_system_time_resolution);

      #endif /* OPTION_TIME_RESOLUTION */

}; /* Container */
```

```
} /* #component_impl_name# */
```

11.5 Triggers

The C++ syntax for a component instance to set and cancel a trigger is given hereafter:

11.5.1 Trigger set

```
/*
 * @file #component_impl_name#_container.hpp
 * Container Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

namespace #component_impl_name#
{

class Container
{
   public:

      void #trigger_name#_set
         (const ECOA::duration& delay);

}; /* Container */

} /* #component_impl_name# */
```

The following return status values shall be managed:

- ECOA::return_status::OK
- ECOA::return_status::OPERATION_ALREADY_PENDING
- ECOA::return_status::FAILURE

11.5.2 Trigger cancel

```
/*
 * @file #component_impl_name#_container.hpp
 * Container Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

namespace #component_impl_name#
{

class Container
```

```
{
   public:

      void #trigger_name#_cancel();

}; /* Container */

} /* #component_impl_name# */
```

The following return status values shall be managed:

- ECOA::return_status::OK
- ECOA::return_status::FAILURE

11.6 Persistent Information management (PINFO)

11.6.1 PINFO read

The C++ syntax for a component instance to read persistent data (PINFO) is:

```
/*
 * @file #component_impl_name#_container.hpp
 * Container Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

namespace #component_impl_name#
{

class Container
{
   public:

      // the following method shall be implemented by the Container to allow
      // the component to read the corresponding persistent data #PINFOname#.
      ECOA::return_status read_#PINFOname#
         (ECOA::byte *memory_address,
          ECOA::uint32 in_size,
          ECOA::uint32 *out_size);

}; /* Container */

} /* #component_impl_name# */
```

The following return status values shall be managed:

- ECOA::return_status::OK
- ECOA::return_status::RESOURCE_NOT_AVAILABLE

- ECOA::return_status::INVALID_PARAMETER
- ECOA::return_status::FAILURE

11.6.2 PINFO write

When PINFO_WRITE option is defined, the C++ syntax for a component instance to write persistent data (PINFO) is:

```
/*
 * @file #component_impl_name#_container.hpp
 * Container Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

namespace #component_impl_name#
{

class Container
{
   public:

      #if defined(OPTION_PINFO_WRITE)

      // the following optional method shall be implemented by the Container
      // to allow the component to write in the corresponding
      // persistent data #PINFOname#.
      ECOA::return_status write_#PINFOname# (ECOA::byte *memory_address,
                                             ECOA::uint32 in_size,
                                             ECOA::uint32 *out_size);

      #endif /* OPTION_PINFO_WRITE */

}; /* Container */

} /* #component_impl_name# */
```

The following return status values shall be managed:

- ECOA::return_status::OK
- ECOA::return_status::RESOURCE_NOT_AVAILABLE
- ECOA::return_status::INVALID_PARAMETER
- ECOA::return_status::FAILURE

11.6.3 PINFO seek

The C++ syntax for a component instance to seek within persistent data (PINFO) is:

```
/*
 * @file #component_impl_name#_container.hpp
 * Container Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

namespace #component_impl_name#
{

class Container
{
   public:

      // the following method shall be implemented by the Container to allow
      // the component to seek within the corresponding persistent data
      // #PINFOname#.
      ECOA::return_status seek_#PINFOname#
         (ECOA::int32 offset,
          ECOA::seek_whence_type whence,
          ECOA::uint32 *new_position);

}; /* Container */

} /* #component_impl_name# */
```

The following return status values shall be managed:

- ECOA::return_status::OK
- ECOA::return_status::RESOURCE_NOT_AVAILABLE
- ECOA::return_status::INVALID_PARAMETER
- ECOA::return_status::FAILURE

11.7 Save Warm Start Context

When WARM_START_CONTEXT option is defined, the C++ syntax for a component instance to save its warm start (non-volatile) context is:

```
/*
 * @file #component_impl_name#_container.hpp
 * Container Interface class header for Component #component_impl_name#

 * Generated automatically from specification; do not modify here
```

```
 */

namespace #component_impl_name#
{

class Container
{
   public:

      #if defined(OPTION_WARM_START_CONTEXT)

      // the following optional method shall be implemented by the Container to
      // allow the component to save its warm start (non-volatile) context.
      void save_warm_start_context();

      #endif /* OPTION_WARM_START_CONTEXT */


}; /* Container */

} /* #component_impl_name# */
```

11.8 Supervisor components

This section is specific to [OPTION SUPERVISION].

When the component is of SUPERVISOR kind, the C++ syntax for a component instance to command and control executables, components, and variables, is described in the following sections.

11.8.1 Supervision of executables

```
/*
 * @file #component_impl_name#_container.hpp
 * Container Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

namespace #component_impl_name#
{

class Container
{
   public:

      #if defined(OPTION_SUPERVISION)
```

```
      ECOA::return_status get_executable_state(ECOA::asset_id id,
                                               ECOA::executable_state* state);

      void executable_command(ECOA::asset_id id,
                              ECOA::executable_command command);

      #endif /* OPTION_SUPERVISION */

}; /* Container */

} /* #component_impl_name# */
```

For these functions, the following return status values shall be managed:

- ECOA::return_status::OK
- ECOA::return_status::FAILURE

11.8.2 Supervision of components

```
/*
 * @file #component_impl_name#_container.hpp
 * Container Interface class header for Component #component_impl_name#

 * Generated automatically from specification; do not modify here
 */

namespace #component_impl_name#
{

class Container
{
   public:

      #if defined(OPTION_SUPERVISION)

       ECOA::return_status get_component_state(const ECOA::asset_id id,
                                               ECOA::component_state* state);

       ECOA::return_status component_state_command(const ECOA::asset_id id,
                                       const ECOA::component_command command);

      #endif /* OPTION_SUPERVISION */

}; /* Container */
```

```
} /* #component_impl_name# */
```

For these functions, the following return status values shall be managed:

- ECOA_::return_status::OK
- ECOA::return_status::FAILURE

11.8.3 Supervision variables

```
/*
 * @file #component_impl_name#_container.hpp
 * Container Interface class header for Component #component_impl_name#

 * Generated automatically from specification; do not modify here
 */

namespace #component_impl_name#
{

class Container
{
   public:

      #if defined(OPTION_SUPERVISION)

      ECOA::return_status set_variable_#variable_name#
                              (const #variable_type# *value);

      ECOA::return_status get_variable_#variable_name#
                              (#variable_type# *value);

      #endif /* OPTION_SUPERVISION */

}; /* Container */

} /* #component_impl_name# */
```

For these functions, the following return status values shall be managed:

- ECOA::return_status::OK
- ECOA::return_status::FAILURE

12 Container Types

This section contains details of the data types that comprise the container API i.e. the data types that can be used by a component.

12.1 Versioned Data Handles

This section contains the C++ syntax in order to define data handles for versioned data operations defined in the Container Interface.

```
/*
 * @file #component_impl_name#_container_types.hpp
 * Container Types for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

#define ECOA_VERSIONED_DATA_HANDLE_PRIVATE_SIZE 32

namespace #component_impl_name# {

/*
 * The following is the data handle structure associated to the data
 * operation called #operation_name# of data-type #type_name#
 */
typedef struct {
   /* pointer to the local copy of the data */
   #type_name#* data;
   /* stamp updated each time the data value is updated locally for that reader */
   ECOA::uint32 stamp;
   /* technical info associated with the data (opaque for the user, reserved */
   /* for the infrastructure) */
   ECOA::byte platform_hook[ECOA_VERSIONED_DATA_HANDLE_PRIVATE_SIZE];
} #operation_name#_handle;

} /* #component_impl_name# */
```

13 Default values

Not applicable to the C++ Binding.

14 External Interface

This section contains the C++ syntax for the ECOA external interface provided to non-ECOA software by the container when EXTERNAL_INTERFACE option is defined.

NOTE: The choice of the language for generating external APIs is made separately from the choice of the language for generating ECOA components APIs. The choice of supported languages is made depending on needs that are to be taken into account in platform procurement requirements.

```
/* @file #component_impl_name#_External_Interface.hpp
 * External Interface header for Component Implementation
 * #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

namespace #component_impl_name#_External_Interface
{
   public:

      #if defined(OPTION_EXTERNAL_INTERFACE)

      void #external_operation_name#(const #event_parameters#);

      #endif /* OPTION_EXTERNAL_INTERFACE */

} /* #component_impl_name#_External_Interface */
```

15 External Components

This section contains the C++ syntax for the "External" special component kind.

The C++ syntax for the external routine (code executed by the external thread) of an External Component is:

```
/*
 * @file #component_impl_name#.hpp
 * Component Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

namespace #component_impl_name#
{

class Component
{
   public:

      void external_routine();

}; /* Component */

} /* #component_impl_name# */
```

The C++ syntax for the functions to allow starting and stopping the external thread is:

```
/*
 * @file #component_impl_name#_container.hpp
 * Container Interface class header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

namespace #component_impl_name#
{

class Container
{
   public:

      ECOA::return_status start_external_task();

      ECOA::return_status stop_external_task();

}; /* Container */

} /* #component_impl_name# */
```

For these functions, when available, the following return status values shall be managed:

- ECOA::return_status::OK
- ECOA::return_status::OPERATION_NOT_AVAILABLE
- ECOA::return_status::FAILURE

Note that the external thread can be automatically started if [OPTION AUTO START EXTERNAL START] is selected.

16 TriggerManager Components

There is no specific API for PeriodicTriggerManager components, since these components are entirely managed by the infrastructure.

Idem for DynamicTriggerManager Components (related to DYNAMIC_TRIGGER option).

17 Reference C++ Header

```
/*
 * @file ECOA.hpp
 */

/*  This is a compilable ISO C++ 98 specification of the generic ECOA       */
/*  types derived from the C++ binding specification.                       */
```

```
/*  The declarations of the types given below are taken from the            */
/*  standard, as are the enum types and the names of the others types.      */
/*  Unless specified as implementation dependent, the values specified in   */
/*  this appendix should be implemented as defined.                         */


#if !defined(ECOA_HPP)
#define ECOA_HPP

namespace ECOA {

   /* ECOA:boolean8 */
   typedef unsigned char boolean8;
   static const boolean8 TRUE  = 1;
   static const boolean8 FALSE = 0;

   /* ECOA:int8 */
   typedef char int8;
   static const int8 INT8_MIN = -127;
   static const int8 INT8_MAX = 127;

   /* ECOA:char8 */
   typedef char char8;
   static const char8 CHAR8_MIN = 0;
   static const char8 CHAR8_MAX = 127;

   /* ECOA:byte */
   typedef unsigned char byte;
   static const byte BYTE_MIN = 0;
   static const byte BYTE_MAX = 255;

   /* ECOA:int16 */
   typedef short int int16;
   static const int16 INT16_MIN = -32767;
   static const int16 INT16_MAX = 32767;

   /* ECOA:int32 */
   typedef int int32;
   static const int32 INT32_MIN = -2147483647L;
   static const int32 INT32_MAX = 2147483647L;

    /* ECOA:uint8 */
   typedef unsigned char uint8;
   static const uint8 UINT8_MIN = 0;
```

```
   static const uint8 UINT8_MAX = 255;

   /* ECOA:uint16 */
   typedef unsigned short int uint16;
   static const uint16 UINT16_MIN = 0;
   static const uint16 UINT16_MAX = 65535;

   /* ECOA:uint32 */
   typedef unsigned int uint32;
   static const uint32 UINT32_MIN = 0LU;
   static const uint32 UINT32_MAX = 4294967295LU;

#if defined (ECOA_INT64_SUPPORT)
   /* ECOA:int64 */
   typedef long long int int64;
   static const int64 INT64_MIN = -92233720368547758O7LL;
   static const int64 INT64_MAX = 9223372036854775807LL;
#endif /* ECOA_INT64_SUPPORT */

#if defined(ECOA_UINT64_SUPPORT)
   /* ECOA:uint64 */
   typedef unsigned long long int uint64;
   static const uint64 UINT64_MIN = 0LLU;
   static const uint64 UINT64_MAX = 18446744073709551615LLU;
#endif /* ECOA_UINT64_SUPPORT */

   /* ECOA:float32 */
   typedef float float32;
   static const float32 FLOAT32_MIN = -3.402823466e+38F;
   static const float32 FLOAT32_MAX = 3.402823466e+38F;

   /* ECOA:double64 */
   typedef double double64;
   static const double64 DOUBLE64_MIN = -1.7976931348623157e+308;
   static const double64 DOUBLE64_MAX = 1.7976931348623157e+308;

   /* ECOA:return_status */
   struct return_status
   {
      ECOA::uint32 value;
      enum EnumValues
      {
         OK                         = 0,
         FAILURE                    = 1,
```

```
         INVALID_HANDLE            = 2,
         DATA_NOT_INITIALIZED      = 3,
         NO_DATA                   = 4,
         INVALID_IDENTIFIER        = 5,
         NO_RESPONSE               = 6,
         OPERATION_ALREADY_PENDING = 7,
         CLOCK_UNSYNCHRONIZED      = 8,
         RESOURCE_NOT_AVAILABLE    = 9,
         OPERATION_NOT_AVAILABLE   = 10,
         INVALID_PARAMETER         = 11
      };
      inline void operator = (ECOA::uint32 i) { value = i; }
      inline operator ECOA::uint32 () const { return value; }
      inline return_status (EnumValues v):value(v) {}
      inline return_status ():value(OK) {}
   };

   /* Component and executable identifiers ECOA:asset_id */
typedef ECOA::uint32 asset_id;

   /* ECOA:write_access_mode */
   struct write_access_mode
   {
      ECOA::uint32 value;
      enum EnumValues
      {
         READ_AND_UPDATE   = 0,
         WRITE_ONLY        = 1
      };
      inline void operator = (ECOA::uint32 i) { value = i; }
      inline operator ECOA::uint32() const { return value; }
      inline write_access_mode (EnumValues v):value(v) {}
      inline write_access_mode ():value(READ_AND_UPDATE) {}
};

   /* ECOA:hr_time */
   typedef struct {
      ECOA::uint32 seconds;     /* Seconds */
      ECOA::uint32 nanoseconds; /* Nanoseconds*/
   } hr_time;

   /* ECOA:global_time */
   typedef struct {
      ECOA::uint32 seconds;     /* Seconds */
```

```
   ECOA::uint32 nanoseconds; /* Nanoseconds*/
} global_time;

/* ECOA:duration */
typedef struct {
   ECOA::uint32 seconds;     /* Seconds */
   ECOA::uint32 nanoseconds; /* Nanoseconds*/
} duration;

/* ECOA:log */
static const ECOA::uint32 LOG_MAXSIZE = 256;
typedef struct {
   ECOA::uint32 current_size;
   ECOA::char8  data[ECOA::LOG_MAXSIZE];
} log;

/* ECOA:information_category */
struct information_category
{
   ECOA::uint32 value;
   enum EnumValues
   {
      NONE           = 0,
      CRITICAL       = 1,
      ERROR          = 1,
      WARNING        = 3,
      INFO           = 4,
      DEBUG          = 5,
      TRACE          = 6
   };
   inline void operator = (ECOA::uint32 i) { value = i; }
   inline operator ECOA::uint32() const { return value; }
   inline information_category  (EnumValues v):value(v) {}
   inline information_category  ():value(NONE) {}
};

/* ECOA:error_id */
typedef ECOA::uint32 error_id;

/* ECOA:error_code */
typedef ECOA::uint32 error_code;

/* ECOA:asset_type */
struct asset_type
```

```
{
   ECOA::uint32 value;
   enum EnumValues
   {
      COMPONENT         = 0,
      EXECUTABLE        = 1
   };
   inline void operator = (ECOA::uint32 i) { value = i; }
   inline operator ECOA::uint32() const { return value; }
   inline asset_type (EnumValues v):value(v) {}
   inline asset_type ():value(COMPONENT) {}
};

/* ECOA:error_type */
struct error_type
{
   ECOA::uint32 value;
   enum EnumValues
   {
      RESOURCE_NOT_AVAILABLE = 0,
      UNAVAILABLE            = 1,
      MEMORY_VIOLATION       = 2,
      NUMERICAL_ERROR        = 3,
      ILLEGAL_INSTRUCTION    = 4,
      STACK_OVERFLOW         = 5,
      DEADLINE_VIOLATION     = 6,
      OVERFLOW               = 7,
      UNDERFLOW              = 8,
      ILLEGAL_INPUT_ARGS     = 9,
      ILLEGAL_OUTPUT_ARGS    = 10,
      ERROR                  = 11,
      FATAL_ERROR            = 12,
      HARDWARE_FAULT         = 13,
      POWER_FAIL             = 14,
      COMMUNICATION_ERROR    = 15,
      INVALID_CONFIG         = 16,
      INITIALISATION_PROBLEM = 17,
      CLOCK_UNSYNCHRONIZED   = 18
  };

   inline void operator = (ECOA::uint32 i) { value = i; }
   inline operator ECOA::uint32() const { return value; }
   inline error_type (EnumValues v):value(v) {}
   inline error_type ():value(RESOURCE_NOT_AVAILABLE) {}
```

```
};


const ECOA::uint32 PINFO_FILENAME_MAXSIZE = 256;

/* ECOA:pinfo_filename */
typedef struct
{
   ECOA::uint32 current_size;
   ECOA::char8  data[ECOA::PINFO_FILENAME_MAXSIZE];
} pinfo_filename;

/* ECOA:seek_whence_type */
struct seek_whence_type {
   ECOA::uint32 value;
   enum EnumValues
   {
      SEEK_SET = 0,
      SEEK_CUR = 1,
      SEEK_END = 2
   };
   inline void operator = (ECOA::uint32 i) { value = i; }
   inline operator ECOA::uint32() const { return value; }
   inline seek_whence_type (EnumValues v):value(v) {}
   inline seek_whence_type ():value(ECOA_SEEK_SET) {}
};


/* ECOA:component_state */
struct component_state
{
   ECOA::uint32 value;
   enum EnumValues
   {
      UNAVAILABLE = 0,
      IDLE = 1,
      READY = 2,
      RUNNING = 3
   };
   inline void operator = (ECOA::uint32 i) { value = i; }
   inline operator ECOA::uint32() const { return value; }
   inline component_state (EnumValues v):value(v) {}
   inline component_state ():value(UNAVAILABLE) {}
};
```

```
/* ECOA:component_command */
struct component_command
{
   ECOA::uint32 value;
   enum EnumValues
   {
      INIT = 0,
      START = 1,
      STOP = 2,
      RESET = 3,
      SHUTDOWN = 4,
      KILL = 5
   };
   inline void operator = (ECOA::uint32 i) { value = i; }
   inline operator ECOA::uint32() const { return value; }
   inline component_command (EnumValues v):value(v) {}
   inline component_command ():value(INIT) {}
};

/* ECOA:executable_state */
struct executable_state
{
   ECOA::uint32 value;
   enum EnumValues
   {
      NOT_LAUNCHED = 0,
      LAUNCHED     = 1
   };
   inline void operator = (ECOA::uint32 i) { value = i; }
   inline operator ECOA::uint32() const { return value; }
   inline executable_state (EnumValues v):value(v) {}
   inline executable_state ():value(NOT_LAUNCHED) {}
};

/* ECOA:executable_command */
struct executable_command
{
   ECOA::uint32 value;
   enum EnumValues
   {
      START = 0,
      STOP  = 1
```

```
      };
      inline void operator = (ECOA::uint32 i) { value = i; }
      inline operator ECOA::uint32() const { return value; }
      inline executable_command (EnumValues v):value(v) {}
      inline executable_command ():value(START) {}
   };


} /* ECOA */

#endif /* ECOA_HPP */
```

18 Compatibility with ECOA options

The following table indicates, for each optional functionnality defined in the ECOA Standard (taken from [Part 5] document), whether it is supported or not by this binding.

- YES: the option is supported by this binding
- NO: the option is not supported by this binding
- N/A: Not Applicable. The option has no impact on bindings.

Name of Option	Supported by this binding
[OPTION SUPERVISOR COMPONENTS]	YES
[OPTION ELI]	N/A
[OPTION FAULT HANDLING]	YES
[OPTION MULTI APP ASSEMBLY]	N/A
[OPTION DYNAMIC TRIGGER MANAGER]	N/A
[OPTION UINT64]	YES
[OPTION INT64]	YES
[OPTION PINFO WRITE]	YES
[OPTION WARM START CONTEXT]	YES
[OPTION AUTO START EXTERNAL TASK]	N/A
[OPTION SYSTEM TIME]	YES
[OPTION UTC TIME]	YES