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

Dassault Ref No: DGT 2041087-A
Thales DMS Ref No: 69398926-035 --

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

0	Introduction	7
1	Scope	8
2	Warning	8
3	Normative References	8
4	Definitions	9
5	Abbreviations	9
6	Component to Language Mapping	10
6.1	Overview of interfaces	10
6.2	Overview of files	12
6.3	Component Interface Template	14
6.4	Container Interface Template	17
6.5	Container Types Template	23
6.6	User Component Context Template	23
6.7	Guards	24
7	Parameters	26
8	Component Context	27
8.1	User Component Context	28
9	Types	30
9.1	Libraries	30
9.2	Basic Types	30
9.3	Derived Types	31
9.3.1	Simple Types	32
9.3.2	Enumerations	32
9.3.3	Records	32
9.3.4	Variant Records	33
9.3.5	Fixed Arrays	33
9.3.6	Variable Arrays	34
9.4	Predefined Abstract Types	34
9.4.1	Function execution return status	34
9.4.2	Component and executable identifiers	34
9.4.3	Write access mode	35
9.4.4	Time management	35
9.4.5	Logs	36

9.4.6	**Error management**	**37**
9.4.7	**Pinfo management**	**38**
9.4.8	**Lifecycle management**	**38**
9.5	**Constants**	**39**
9.6	**Predefined constants**	**39**
9.7	**Functions defined on types**	**39**
9.7.1	**Initialization functions**	**39**
9.7.2	**Comparison functions**	**40**
10	**Component Interface**	**40**
10.1	**Operations**	**40**
10.1.1	**Request-Response**	**41**
10.1.2	**Versioned Data Updated**	**42**
10.1.3	**Event Received**	**42**
10.2	**Component Lifecycle**	**43**
10.2.1	**Initialize_Received**	**43**
10.2.2	**Start_Received**	**43**
10.2.3	**Stop_Received**	**43**
10.2.4	**Shutdown_Received**	**44**
10.2.5	**Reset_Received**	**44**
10.3	**Supervisor components**	**44**
10.4	**Error notification for fault handler components**	**45**
11	**Container Interface**	**45**
11.1	**Operations**	**45**
11.1.1	**Request Response**	**45**
11.1.2	**Versioned Data**	**47**
11.1.3	**Event**	**52**
11.2	**Properties**	**52**
11.2.1	**Get Value**	**52**
11.2.2	**Expressing Property Values**	**52**
11.2.3	**Example of Defining and Using Properties**	**52**
11.3	**Logging and Fault Management**	**53**
11.3.1	**Alternative 1: using fixed interfaces**	**53**
11.3.2	**Alternative 2: using flex interfaces**	**54**
11.4	**Time Services**	**55**
11.4.1	**Get_Relative_Local_Time**	**55**
11.4.2	**Get_UTC_Time**	**55**
11.4.3	**Get_Absolute_System_Time**	**56**

11.4.4 Get_Relative_Local_Time_Resolution 56
11.4.5 Get_UTC_Time_Resolution 56
11.4.6 Get_Absolute_System_Time_Resolution 57
11.5 Triggers 57
11.5.1 Trigger_Set 57
11.5.2 Trigger_Cancel 58
11.6 Persistent Information management (PINFO) 58
11.6.1 PINFO read 58
11.6.2 PINFO write 58
11.6.3 PINFO seek 59
11.7 Save Warm Start Context 59
11.8 Supervisor components 60
11.8.1 Supervision of executables 60
11.8.2 Supervision of components 60
11.8.3 Supervision variables 61
12 Container Types 61
12.1 Versioned Data Handles 61
13 Default Values 62
14 External Interface 62
15 External Components 63
16 TriggerManager Components 63
17 Reference C Header 64
18 Compatibility with ECOA Options 69

Figures

Figure 1 Generic C Files Organization 13

Tables

Table 1 Component and Container Interfaces 10
Table 2 Filename Mapping 13
Table 3 Method of Passing Parameters 26
Table 4 C Basic Type Mapping 30
Table 5 C Predefined Constants 31

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 9899:1999(E)) language binding for the component and container APIs that facilitate communication between the component instances and their container in an ECOA® system.

This language binding is fully compliant with the generic software interfaces specified in [Architecture Specification Part 4].

The C language binding includes the prototypes of the mandadory and optional functions described in [Architecture Specification Part 4]. A platform is compatible with the C language binding provided that:

- It covers at least mandatory functions,
- It complies with prototypes of optional features when present.

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 0 describes the Container Interface;
- Section 0 describes the Container Types;
- Section 13 describes Default Values
- Section 13 describes the External Interface;
- Section 15 describes the External Components;
- Section 0 describes PeriodicTriggerManager 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. ECOA® is a registered trademark.
PINFO	Persistent Information
UTC	Coordinated Universal Time
XML	eXtensible Markup Language

6 Component to Language Mapping

6.1 Overview of interfaces

Erreur ! Source du renvoi introuvable. 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**
	Is_Initialized	**OPTIONAL**	**YES [BINDING OPTION EXTENDED VD API]**
	Release_All_Data_Handles	**OPTIONAL**	**YES [BINDING OPTION EXTENDED VD API]**

Category	Abstract API Name	Level	Covered by the ECOA C Language Binding
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**	
	Get_Absolute_System_ Time_Resolution	**OPTIONAL**	
Triggers	Trigger_Set	**MANDATORY**	**YES MANDATORY**

Category	Abstract API Name	Level	Covered by the ECOA C Language Binding
	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

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

With structured languages such as C, the Component Interface will be composed of a set of functions corresponding to each entry-point of the Component Implementation. The declaration of these functions will be accessible in a header file called #component_impl_name#.h. The names of these functions shall begin with the prefix "#component_impl_name#__".

The Container Interface will be composed of a set of functions corresponding to the required operations. The declaration of these functions will be accessible in a header file called #component_impl_name#_container.h. The names of these functions shall begin with the prefix "#component_impl_name#_container__".

The Container Types will be composed of the types which the Component Implementation needs in order to declare, use and store various handles. The declaration of these types will be accessible in a header file called #component_impl_name#_container_types.h. The names of these types shall begin with the prefix "#component_impl_name#_container__".

It is important to ensure that the names of these functions and types do not clash within a single executable. One way to achieve this is for each component supplier to define the component implementation name prefixed by a unique identifier. In this way they can manage the uniqueness of their own components, and the mixing of different supplier components within an executable is possible.

A dedicated structure named #component_impl_name#__context, and called Component Context structure in the rest of the document will be generated by the ECOA toolchain in the Component Container header (#component_impl_name#_container.h) and shall be extended by the Component implementer to contain all the user variables of the Component. This structure will be allocated by the container before Component Instance start-up and passed to the Component Instance in each activation entry-point (i.e. received events, received requests or received asynchronous responses).

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

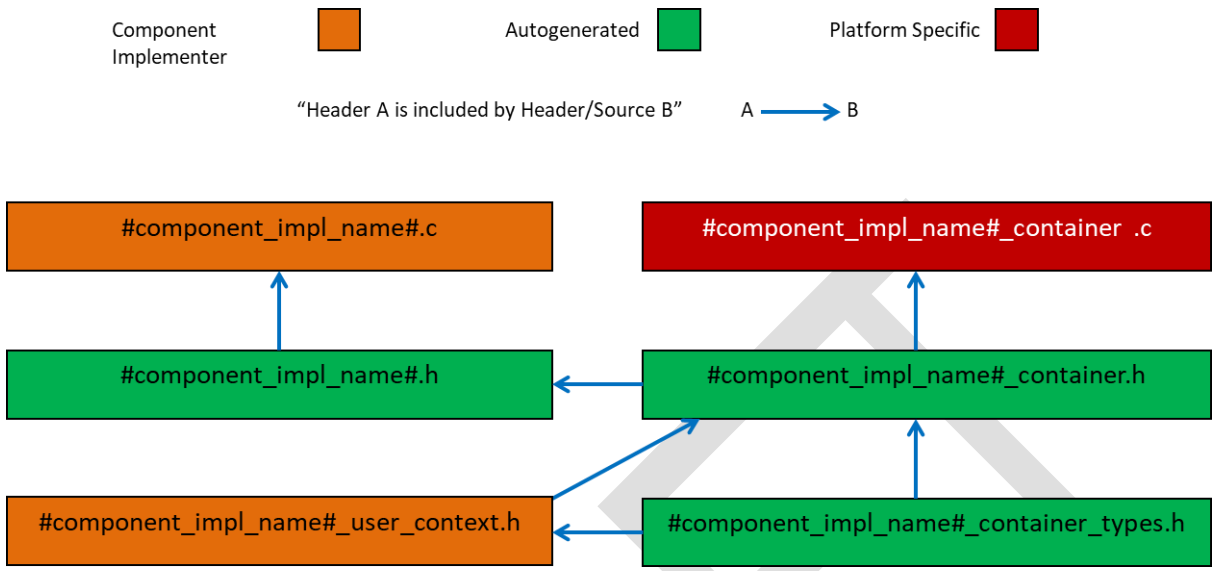

Figure 1 Generic C Files Organization

Complementary files are necessary in case external communications are defined, either using optional External Interfaces or using External Components. See §14 and §15.

Table 2 Filename Mapping

Filename	Use
#component_impl_name#.h	Component Interface declaration (entry points provided by the component and callable by the container)
#component_impl_name#.c	Component Implementation (implements the component interface)
#component_impl_name#_container.h	Container Interface declaration (functions provided by the container and callable by the component) Component Context type declaration
#component_impl_name#_container.c	Container Implementation: This source (.c) 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.

Filename	Use
#component_impl_name#_container_types.h	Container Types declaration (container-level data types usable by the component) These types are related to the Container for a Component Implementation and are declared with the **#component_impl_name#_container** prefix.
#component_impl_name#_user_context.h	User extensions to Component Context. These types are related to the Component Implementation and are declared with the **#component_impl_name#** prefix.
#component_impl_name#_External_Interface.h	External interface provided to non-ECOA software by a component container when EXTERNAL_INTERFACE option is defined.
#component_impl_name#_External_Component_Interface.h	External thread interface of an External Component.
#library#.h	Types defined from a data type library.
#library#_initialize.h	Initialization functions for #library# types.
#library#_compare.h	Comparison functions for #library# types.
epsilon_definition.h	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.h	Definition of asset ids to be used in software interfaces that require component or executable identifiers.

Templates for the files in Table 2 are provided in the following sections and in sections 14 and 15.

6.3 Component Interface Template

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

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

#if defined(__cplusplus)
extern "C" {
#endif /* __cplusplus */
```

```
/* Standard Types */

#include <ECOA.h>
/* Additionally created types */
#include #additionally_created_types#
/* Include container header */
#include "#component_impl_name#_container.h"
/* Include container types */
#include "#component_impl_name#_container_types.h"

void #component_impl_name#__INITIALIZE__received
   (#component_impl_name#__context* context);

void #component_impl_name#__START__received
   (#component_impl_name#__context* context);
```

```
/* CONDITIONAL BLOCK : present only if component attribute hasReset = true*/
void #component_impl_name#__RESET__received
   (#component_impl_name#__context* context);
/* END OF CONDITIONAL BLOCK */
```

```
void #component_impl_name#__STOP__received
   (#component_impl_name#__context* context);

void #component_impl_name#__SHUTDOWN__received
   (#component_impl_name#__context* context);

/* Event operation handlers specifications */
#list_of_event_operations_specifications#

/* Request-Response operation handlers specifications */
#list_of_request_response_operations_specifications#

/* Versioned Data Notifying operation handlers specifications */
#list_of_versioned_data_notifying_operations_specifications#
```

```
/* 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 */
```

```
/* END OF CONDITIONAL BLOCK */
```

```
/* 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 */
```

```
#if defined(__cplusplus)
}
#endif /* __cplusplus */

#endif  /* #COMPONENT_IMPL_NAME#_H */
```

```
/*
 * @file #component_impl_name#.c
 * Component Interface for Component #component_impl_name#
 * This file can be considered a template with the operation stubs
 * auto generated by the ECOA toolset and filled in by the component
 * developer.
 */

/* Include component interface header */
#include "#component_impl_name#.h"

/* 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#
```

```
/* Lifecycle operation handlers */
#list_of_lifecycle_operations#
```

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

```
#if defined(OPTION_SUPERVISION)

/* On state change handler if this component is a Supervisor */
#on_state_change_operations#

#endif /* OPTION_SUPERVISION */
```

6.4 Container Interface Template

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

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

#if defined(__cplusplus)
extern "C" {
#endif /* __cplusplus */

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

```
#include "#component_impl_name#_user_context.h"

/* Incomplete definition of the technical (platform-dependent) part of the */
/* context (it will be defined privately by the container) */
struct #component_impl_name#__platform_hook;


/* Component Context structure declaration */
typedef struct
{
   /*
    * Other container technical data will be accessible through the pointer
    * defined here
    */
   struct #component_impl_name#__platform_hook *platform_hook;

   /* The type
    * #component_impl_name#_user_context shall be defined by the user
    * in the #component_impl_name#_user_context.h file to carry the component
    * implementation private data
    */
   #component_impl_name#_user_context user;

   /* CONDITIONAL BLOCK : present only if component attribute
   hasWarmStartContext = true */
       #if defined(OPTION_WARM_START_CONTEXT)

      /* When the optional warm start context is used, the type
       * #component_impl_name#_warm_start_context shall be defined by the user
       * in the #component_impl_name#_user_context.h file to carry the
       * component implementation private data
       */
      #component_impl_name#_warm_start_context warm_start;

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

} #component_impl_name#__context;


void #component_impl_name#_container__log_trace
   (#component_impl_name#__context* context,
```

```
    const ECOA__log log);

void #component_impl_name#_container__log_debug
   (#component_impl_name#__context* context,
    const ECOA__log log);

void #component_impl_name#_container__log_info
   (#component_impl_name#__context* context,
    const ECOA__log log);

void #component_impl_name#_container__log_warning
   (#component_impl_name#__context* context,
    const ECOA__log log);

void #component_impl_name#_container__raise_error
   (#component_impl_name#__context* context,
    const ECOA__log log);

void #component_impl_name#_container__raise_fatal_error
   (#component_impl_name#__context* context,
    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 */
```

```
/* CONDITIONAL BLOCK : present only if component attribute
needsLocalTime = true */
void #component_impl_name#_container__get_relative_local_time
   (#component_impl_name#__context* context,
    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 #component_impl_name#_container__get_UTC_time
   (#component_impl_name#__context* context,
    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
#component_impl_name#_container__get_absolute_system_time
   (#component_impl_name#__context* context,
    ECOA__global_time *absolute_system_time);
/* END OF CONDITIONAL BLOCK */


/* 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 #component_impl_name#_container__get_relative_local_time_resolution
   (#component_impl_name#__context* context,
    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 #component_impl_name#_container__get_UTC_time_resolution
   (#component_impl_name#__context* context,
    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 #component_impl_name#_container__get_absolute_system_time_resolution
   (#component_impl_name#__context* context,
    ECOA__duration *absolute_system_time_resolution);
/* END OF SUB-CONDITIONAL BLOCK */
```

```
#endif /* BINDING_OPTION_TIME_RESOLUTION */

/* END OF CONDITIONAL BLOCK */
```

```
/* 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#
```

```
/* Persistent Information management operations */
#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 attribute
hasWarmStartContext = true */

#if defined(OPTION_WARM_START_CONTEXT)

/* Optional API for saving the warm start context */
/* Context management operation */
#save_warm_start_context_call_specifications#

#endif /* OPTION_WARM_START_CONTEXT */

/* END OF CONDITIONAL BLOCK */
```

```
/* 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 defined(__cplusplus)
}
#endif /* __cplusplus */
```

```
#endif  /* #COMPONENT_IMPL_NAME#_CONTAINER_H */
```

6.5 Container Types Template

```
/* @file #component_impl_name#_container_types.h
 * Container Types header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */
#if !defined(#COMPONENT_IMPL_NAME#_CONTAINER_TYPES_H)
#define #COMPONENT_IMPL_NAME#_CONTAINER_TYPES_H

#if defined(__cplusplus)
extern "C" {
#endif /* __cplusplus */

#include <ECOA.h>

/* 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
 */

#if defined(__cplusplus)
}
#endif /* __cplusplus */

#endif  /* #COMPONENT_IMPL_NAME#_CONTAINER_TYPES_H */
```

6.6 User Component Context Template

```
/* @file #component_impl_name#_user_context.h
 * This is an example of a user defined User Component context
 */
#if !defined(#COMPONENT_IMPL_NAME#_USER_CONTEXT_H)
#define #COMPONENT_IMPL_NAME#_ USER_CONTEXT_H

#if defined(__cplusplus)
extern "C" {
#endif /* __cplusplus */

/* Standard Types */
#include <ECOA.h>
/* Additionally created types */
```

```
#include #additionally_created_types#
/* Container Types */
#include "#component_impl_name#_container_types.h"

/* User Component Context structure example */
typedef struct
{
   /* declare the User Component Context "local" data here */

} #component_impl_name#_user_context;
```

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

#if defined (OPTION_WARM_START_CONTEXT)
/* Warm Start Component Context structure example */
typedef struct
{
   /* declare the Warm Start Component Context data here */

} #component_impl_name#_warm_start_context;

#endif /* OPTION_WARM_START_CONTEXT */

/* END OF CONDITIONAL BLOCK */
```

```
#if defined(__cplusplus)
}
#endif /* __cplusplus */

#endif  /* #COMPONENT_IMPL_NAME#_USER_CONTEXT_H */
```

6.7 Guards

In C, all of the declarations within header files shall be surrounded within the following block to make the code compatible with C++, and to avoid multiple inclusions:

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

#if defined(__cplusplus)
```

```
extern "C" {
#endif /* __cplusplus */

/* all the declarations shall come here */


#if defined(__cplusplus)
}
#endif /* __cplusplus */

#endif  /* #macro_protection_name#_H */
```

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

7 Parameters

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

- Input parameters defined with a simple type (i.e. basic, enum or actual simple type) will be passed by value, output parameters defined with a simple type will be passed as pointers
- Input parameters defined with a complex type will be passed as pointers to a const; output parameters defined with a complex type will be passed as pointers.

Table 3 Method of Passing Parameters

	Input parameter	Output parameter
Simple type	By value	Pointer
Complex type	Pointer to const	Pointer

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

In the C language, the Component Context is a structure which holds both the user local data (called “User Component Context” and “Warm Start Context”) and infrastructure-level technical data (which is implementation dependant). The warm start context feature may be optionally selected in the Component Implementation model using option 'hasWarmStartContext'. The presence or absence of declarations of corresponding fields in Component code must be in accordance with selections made in the Component Implementation model. The structure is defined in the Container Interface.

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

The following shows the C syntax for the Component Context:

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

/* Container Types */
#include "#component_impl_name#_container_types.h"

/* User Context */
#include "#component_impl_name#_user_context.h"

/* Incomplete definition of the technical (platform-dependent) part of the */
/* context (it will be defined privately by the container) */
struct #component_impl_name#__platform_hook;

/* Component Context structure declaration */
typedef struct
{
   /*
    * Other container technical data will accessible through the pointer */
    * defined here
    */
   struct #component_impl_name#__platform_hook *platform_hook;

   /* The type
    * #component_impl_name#_user_context shall be defined by the user
    * in the #component_impl_name#_user_context.h file to carry the component
    * implementation private data and the attribute
    * #component_impl_name#_user_context user shall be declared as follows:
    */
   #component_impl_name#_user_context user;
```

```
/* CONDITIONAL BLOCK : present only if component attribute
hasWarmStartContext = true */
   /*
    * When the optional warm start context is used, the type
    * #component_impl_name#_warm_start_context shall be defined by the
    * user in the #component_impl_name#_user_context.h file to carry the
    * component implementation warm start private data and the attribute
    * #component_impl_name#_warm_start_context user shall be declared as
    * follows:
    */
    #component_impl_name#_warm_start_context warm_start;

/* END OF CONDITIONAL BLOCK */
```

```
} #component_impl_name#__context;
```

8.1 User Component Context

The following shows the C syntax for 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). The Component User Context header file is needed only if the user context and/or warm start context are used:

```
/* @file #component_impl_name#_user_context.h
 * This is an example of a user defined User Component context
 */

/* Container Types */
#include "#component_impl_name#_container_types.h"

/* User Component Context structure example */
typedef struct
{
   /* declare the User Component Context "local" data here */
   int myCounter;
} #component_impl_name#_user_context;

/* Warm Start context structure example */
typedef struct {

   /* declare the warm start data here */
   ECOA__boolean8 warm_start_valid; /* example of  validity flag */
   unsigned long my_data;
```

```
} #component_impl_name#_warm_start_context;
```

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

The following example illustrates the usage of the Component context in the entry-point corresponding to an event-received:

```
/* @file "#component_impl_name#.c"
 * Generic operation implementation example
 */

void #component_impl_name#__#operation_name#__received
    (#component_impl_name#__context* context)
{
    /* To be implemented by the component */

    /*
     * …
     * increments a local user defined counter:
     */
    context->user.myCounter++;
}
```

The user extensions to Component Context need to be known by the container in order to allocate the required memory area. This means that the component supplier is required to provide the associated header file. If the supplier does not want to divulge the original contents of the header file, then:

- It may be replaced by an array with a size equivalent to the original data; or
- Memory management may be dealt with internally to the code, using memory allocation functions[1]
- The size of the Component User Context and Warm Start Context may be declared in the bin-desc file related to the Component.

To extend the Component Context structure, the component implementer shall define the User Component Context structure, named #component_impl_name#_user_context, in a header file called #component_impl_name#_user_context.h. All the private data of the Component Implementation shall be added as members of this structure, and will be accessible within the "user" field of the Component Context.

The Component Context structure will be passed by the Container to the Component as the first parameter for each operation (i.e. received events, received requests or received asynchronous responses). The Component Context defines the instance of the Component being invoked by the operation. This structure shall be passed by the Component to all Container Interface API functions it can call.

1 The current ECOA Architecture Specification does not specify any memory allocation function. So, this case may lead to non-portable code.

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 library defined in `#library#.types.xml` shall be placed in a file called `#library#.h`

The complete name of the declaration of a variable name and type name will be computed by prefixing these names with the name of the library. In the C language, this naming rule will be used for each variable or type declaration to create the complete variable name, reflecting the library from which it is defined.

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

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

typedef #basic_type_name#
  #library#__#simple_type_name#;
```

9.2 Basic Types

The basic types, shown in Table 4, shall be located in the “ECOA” library and hence in ECOA.h which shall also contain definitions of the pre-defined constants, e.g. that define constants to represent the true and false values of the basic Boolean type, that are shown in Table 5.

Table 4 C Basic Type Mapping

ECOA Basic Type	C type
ECOA:boolean8	ECOA__boolean8
ECOA:int8	ECOA__int8
ECOA:char8	ECOA__char8
ECOA:byte	ECOA__byte
ECOA:int16	ECOA__int16
ECOA:int32	ECOA__int32
ECOA:int64	ECOA__int64
ECOA:uint8	ECOA__uint8
ECOA:uint16	ECOA__uint16
ECOA:uint32	ECOA__uint32
ECOA:uint64	ECOA__uint64

ECOA Basic Type	C type
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 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	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	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

Note that as namespaces are not supported in the C language, the actual name of a type (known as the complete type (see para. 9.1) and referred to here by prefixing `complete_`) will be computed by adding as a prefix the name of the library in which it is included.

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# #complete_simple_type_name#;
```

If the optional #minRange# or #maxRange# fields are set, the previous type definition must be followed by the minRange or maxRange constant declarations as follows:

```
#define #complete_simple_type_name#_minRange (#minrange_value#)
#define #complete_simple_type_name#_maxRange (#maxrange_value#)
```

9.3.2 Enumerations

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

```
typedef #basic_type_name# #complete_enum_type_name#;

#define #complete_enum_type_name#_#enum_value_name_1#   (#enum_value_value_1#)
#define #complete_enum_type_name#_#enum_value_name_2#   (#enum_value_value_2#)
#define #complete_enum_type_name#_#enum_value_name_3#   (#enum_value_value_3#)
/*…*/
#define #complete_enum_type_name#_#enum_value_name_n#   (#enum_value_value_n#)
```

Where:

`#basic_type_name#` is either `ECOA__boolean8`, `ECOA__int8`, `ECOA__char8`, `ECOA__byte`, `ECOA__int16`, `ECOA__int32`, `ECOA__int64`, `ECOA__uint8`, `ECOA__uint16`, `ECOA__uint32` or `ECOA__uint64`.

`#complete_enum_type_name#` is computed by prefixing the name of the type with the namespaces and using '__' as separator (see para. 9.1)

`#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

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#, the syntax 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#;
}   #complete_record_type_name#;
```

9.3.4 Variant Records

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#, the syntax is given below.

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

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

typedef struct{

    #complete_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#;

} #complete_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 macro called #complete_array_type_name#_MAXSIZE will be defined to specify the size of the array.

```
#define #complete_array_type_name#_MAXSIZE #max_number#
typedef #complete_data_type_name#
#complete_array_type_name#[#complete_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.

```
#define #complete_var_array_type_name#_MAXSIZE #max_number#
typedef struct {
   ECOA__uint32 current_size;
   #data_type_name# data[#complete_var_array_type_name#_MAXSIZE];
} #complete_var_array_type_name#;
```

9.4 Predefined Abstract Types

9.4.1 Function execution return status

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

```
typedef ECOA__uint32 ECOA__return_status;
#define ECOA__return_status_OK                          (0)
#define ECOA__return_status_FAILURE                     (1)
#define ECOA__return_status_INVALID_HANDLE              (2)
#define ECOA__return_status_DATA_NOT_INITIALIZED        (3)
#define ECOA__return_status_NO_DATA                     (4)
#define ECOA__return_status_INVALID_IDENTIFIER          (5)
#define ECOA__return_status_NO_RESPONSE                 (6)
#define ECOA__return_status_OPERATION_ALREADY_PENDING   (7)
#define ECOA__return_status_CLOCK_UNSYNCHRONIZED        (8)
#define ECOA__return_status_RESOURCE_NOT_AVAILABLE      (9)
#define ECOA__return_status_OPERATION_NOT_AVAILABLE     (10)
#define ECOA__return_status_INVALID_PARAMETER           (11)
```

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 an `ECOA:asset_id` is:

```
typedef ECOA__uint32 ECOA__asset_id;
```

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

```
/* File ECOA_Assets.h */

#include <ECOA.h>

typedef ECOA__uint32 ECOA__asset_id;
#if !defined(ECOA_ASSETS_H)

#define ECOA_ASSETS_H
#define ECOA_Assets__CMP_#component_instance_name1# (#CMP_ID1#)
#define ECOA_Assets__CMP_#component_instance_name2# (#CMP_ID2#)
#define ECOA_Assets__CMP_#component_instance_nameN# (#CMP_IDN#)

#define ECOA_Assets__EXE_#executable_name1# (#PD_ID1#)
#define ECOA_Assets__EXE_#executable_name2# (#PD_ID2#)
#define ECOA_Assets__EXE_#executable_nameN# (#PD_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:

```
typedef ECOA__uint32 ECOA__write_access_mode;
#define ECOA__write_access_mode_READ_AND_UPDATE        (0)
#define ECOA__write_access_mode_WRITE_ONLY             (1)
```

9.4.4 Time management

9.4.4.1 ECOA:hr_time

The binding for time is:

```
typedef struct
{
   ECOA__uint32 seconds;     /* Seconds */
   ECOA__uint32 nanoseconds; /* Nanoseconds*/
} ECOA__hr_time;
```

9.4.4.2 ECOA:global_time

Global time is represented as:

```
typedef struct
{
```

```
   ECOA__uint32 seconds;      /* Seconds */
   ECOA__uint32 nanoseconds; /* Nanoseconds*/
} ECOA__global_time;
```

9.4.4.3 ECOA:duration

Duration is represented as:

```
typedef struct
{
   ECOA__uint32 seconds;      /* Seconds */
   ECOA__uint32 nanoseconds; /* Nanoseconds*/
} ECOA__duration;
```

9.4.5 Logs

9.4.5.1 ECOA:log

The syntax for a log in C is:

```
#define ECOA__LOG_MAXSIZE 256

typedef struct {
   ECOA__uint32 current_size;
   ECOA__char8  data[ECOA__LOG_MAXSIZE];
} ECOA__log;
```

9.4.5.2 ECOA:information_category

The C syntax for this type is:

```
typedef ECOA__uint32 ECOA__information_category;
#define ECOA__information_category_NONE              (0)
#define ECOA__information_category_CRITICAL          (1)
#define ECOA__information_category_ERROR             (2)
#define ECOA__information_category_WARNING           (3)
#define ECOA__information_category_INFO              (4)
#define ECOA__information_category_DEBUG             (5)
#define ECOA__information_category_TRACE             (6)
```

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 ECOA__error_id;
```

9.4.6.2 ECOA:error_code

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

```
typedef ECOA__uint32 ECOA__error_code;
```

9.4.6.3 ECOA:asset_type

The C syntax for this type is:

```
typedef ECOA__uint32 ECOA__asset_type;
#define ECOA__asset_type_COMPONENT              (0)
#define ECOA__asset_type_EXECUTABLE             (1)
```

9.4.6.4 ECOA:error_type

The C syntax for this type is:

```
typedef ECOA__uint32 ECOA__error_type;
#define ECOA__error_type_RESOURCE_NOT_AVAILABLE         (0)
#define ECOA__error_type_UNAVAILABLE                    (1)
#define ECOA__error_type_MEMORY_VIOLATION               (2)
#define ECOA__error_type_NUMERICAL_ERROR                (3)
#define ECOA__error_type_ILLEGAL_INSTRUCTION            (4)
#define ECOA__error_type_STACK_OVERFLOW                 (5)
#define ECOA__error_type_DEADLINE_VIOLATION             (6)
#define ECOA__error_type_OVERFLOW                       (7)
#define ECOA__error_type_UNDERFLOW                      (8)
#define ECOA__error_type_ILLEGAL_INPUT_ARGS             (9)
#define ECOA__error_type_ILLEGAL_OUTPUT_ARGS            (10)
#define ECOA__error_type_ERROR                          (11)
#define ECOA__error_type_FATAL_ERROR                    (12)
#define ECOA__error_type_HARDWARE_FAULT                 (13)
#define ECOA__error_type_POWER_FAIL                     (14)
#define ECOA__error_type_COMMUNICATION_ERROR            (15)
#define ECOA__error_type_INVALID_CONFIG                 (16)
#define ECOA__error_type_INITIALISATION_PROBLEM         (17)
#define ECOA__error_type_CLOCK_UNSYNCHRONIZED           (18)
```

9.4.7 Pinfo management

9.4.7.1 ECOA:pinfo_filename

The syntax for a pinfo_filename in C is:

```
#define ECOA__PINFO_FILENAME_MAXSIZE 256

typedef struct {
   ECOA__uint32 current_size;
   ECOA__char8  data[ECOA__PINFO_FILENAME_MAXSIZE];
} ECOA__pinfo_filename;
```

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:

```
typedef ECOA__uint32 ECOA__seek_whence_type;
#define ECOA__seek_whence_type_SEEK_SET (0)
#define ECOA__seek_whence_type_SEEK_CUR (1)
#define ECOA__seek_whence_type_SEEK_END (2)
```

9.4.8 Lifecycle management

9.4.8.1 ECOA:component_state

The C syntax for this type is:

```
typedef ECOA__uint32 ECOA__component_state;
#define ECOA__component_state_UNAVAILABLE          (0)
#define ECOA__component_state_IDLE                 (1)
#define ECOA__component_state_READY                (2)
#define ECOA__component_state_RUNNING              (3)
```

9.4.8.2 ECOA:component_command

The C syntax for this type is:

```
typedef ECOA__uint32 ECOA__component_command;
#define ECOA__ component_command_INIT               (0)
#define ECOA__component_command_START              (1)
#define ECOA__component_command_STOP               (2)
#define ECOA__component_command_RESET              (3)
#define ECOA__component_command_SHUTDOWN           (4)
#define ECOA__component_command_KILL               (5)
```

9.4.8.3 ECOA:executable_state

The C syntax for this type is:

```
typedef ECOA__uint32 ECOA__executable_state;
#define ECOA__executable_state_NOT_LAUNCHED          (0)
#define ECOA__executable_state_LAUNCHED              (1)
```

9.4.8.4 ECOA:executable_command

The C syntax for this type is:

```
typedef ECOA__uint32 ECOA__executable_command;
#define ECOA__executable_command_START               (0)
#define ECOA__executable_command_STOP                (1)
```

9.5 Constants

The syntax for the declaration of a Constant called "`#constant_name#`" in C is shown below. Note that the `#type_name#` is not used in the C binding. In addition, namespaces are not supported in the C language, so the name of the constant (known as the complete name (see para.9.1) and referred to here by prefixing `complete_`) will be computed by adding as a prefix the name of the library in which it is included.

```
#define #complete_constant_name# (#constant_value#)
```

where #constant_value# is either an integer or floating point value described by the XML description.

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:

```
#define #component_impl_name#_#operation_name#_SERVICE_MAXDEFERRED
#max_concurrent_requests#
```

It is defined in `#component_impl_name#.h` 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. This value shall be provided in the platform documentation.

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

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.h
 * This is an example of user defined epsilon values for comparison between
 * simple floating types
 */

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

10 Component Interface

10.1 Operations

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.

10.1.1 Request-Response

10.1.1.1 Request Received

The following is the C syntax for invoking a request received by a component instance when a response is required, where #component_impl_name# is the name of the component implementation receiving the request and #operation_name# is the operation name. 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#.h
 * Component Interface header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

void #component_impl_name#__#operation_name#__request_received
   (#component_impl_name#__context* context,
    const ECOA__uint32 ID,
    const #request_parameters#);
```

10.1.1.1.2 Request Received when immediate=true

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

void #component_impl_name#__#operation_name#__request_received
   (#component_impl_name#__context* context,
    const ECOA__uint32 ID,
    const #request_parameters#,
    #response_parameters#
);
```

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, where #component_impl_name# is the name of the component implementation receiving the response and #operation_name# is the operation name. (The reply to a synchronous request response is provided by the return of the original request).

```
/*
```

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

void #component_impl_name#__#operation_name#__response_received
   (#component_impl_name#__context* context,
    const ECOA__uint32 ID,
    const ECOA__return_status status,
    const #response_parameters#);
```

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 that reads an item of versioned data that new data has been written.

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

void #component_impl_name#__#operation_name#__updated
   (#component_impl_name#__context* context);
```

10.1.3 Event Received

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

```
/*
```

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

void #component_impl_name#__#operation_name#__received
   (#component_impl_name#__context* context,
    const #event_parameters#);
```

10.2 Component Lifecycle

The following operations are applicable to application, trigger and dynamic-trigger component instances.

10.2.1 Initialize_Received

The C syntax for an operation to initialise a component instance is:

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

void #component_impl_name#__INITIALIZE__received
   (#component_impl_name#__context* context);
```

10.2.2 Start_Received

The C syntax for an operation to start a component instance is:

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


void #component_impl_name#__START__received
   (#component_impl_name#__context* context);
```

10.2.3 Stop_Received

The C syntax for an operation to stop a component instance is:

```
/*
 * @file #component_impl_name#.h
 * Component Interface header for Component #component_impl_name#
```

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

void #component_impl_name#__STOP__received
   (#component_impl_name#__context* context);
```

10.2.4 Shutdown_Received

The C syntax for an operation to shutdown a component instance is:

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

void #component_impl_name#__SHUTDOWN__received
   (#component_impl_name#__context* context);
```

10.2.5 Reset_Received

This entry point is only available if the component implementation model has set the option ***hasReset***.

The C syntax for an operation to (functionally) reset a component instance is:

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

void #component_impl_name#__RESET__received
   (#component_impl_name#__context* context);
```

10.3 Supervisor components

The following C types and methods are 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#.h
 * Component Interface header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

#if defined(OPTION_SUPERVISION)
```

```
void #component_impl_name#__on_state_change
   (#component_impl_name#__context* context,
    const ECOA__asset_id componentInstanceId,
    const ECOA__component_state state,
    const ECOA__component_state previous_state);

#endif /* OPTION_SUPERVISION */
```

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 #fault_handler_component_impl_name#.h
 * Component Interface header for the Fault Handler Component
 * #fault_handler_component_impl_name#
 * Generated automatically from specification; do not modify here
 */

#if defined(OPTION_FAULT_HANDLER)

void #fault_handler_impl_name#__error_notification
   (#fault_handler_impl_name#__context* context,
    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 */
```

11 Container Interface

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.h
 * Container Interface header for Component #component_impl_name#
```

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

ECOA__return_status
  #component_impl_name#_container__#operation_name#__request_sync
   (#component_impl_name#__context* context,
    const #request_parameters#,
    #response_parameters#);
```

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.h
 * Container Interface header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

ECOA__return_status
  #component_impl_name#_container__#operation_name#__request_async
   (#component_impl_name#__context* context,
    ECOA__uint32* ID,
    const #request_parameters#);
```

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.h
 * Container Interface header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
*/
```

```
ECOA__return_status
  #component_impl_name#_container__#operation_name#__response_send
   (#component_impl_name#__context* context,
    const ECOA__uint32 ID,
    const #response_parameters#);
```

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 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

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.h
 * Container Interface header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */
#if defined(BINDING_OPTION_REQUEST_CANCEL)

ECOA__return_status
  #component_impl_name#_container__#operation_name#__request_cancel
   (#component_impl_name#__context* context,
    const ECOA__uint32 ID);

#endif /* BINDING_OPTION_REQUEST_CANCEL */
```

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.h
 * Container Interface header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

#include "#component_impl_name#_container_types.h"

ECOA__return_status
  #component_impl_name#_container__#operation_name#__get_read_access
   (#component_impl_name#__context* context,
    #component_impl_name#_container__#operation_name#_handle* data_handle);
```

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.h
 * Container Interface header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

#include "#component_impl_name#_container_types.h"

ECOA__return_status
  #component_impl_name#_container__#operation_name#__release_read_access
   (#component_impl_name#__context* context,
    #component_impl_name#_container__#operation_name#_handle* data_handle);
```

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.h
 * Container Interface header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

#include "#component_impl_name#_container_types.h"

ECOA__return_status
  #component_impl_name#_container__#operation_name#__get_write_access
   (#component_impl_name#__context* context,
    #component_impl_name#_container__#operation_name#_handle* data_handle);
```

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

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

#include "#component_impl_name#_container_types.h"

#if defined(BINDING_OPTION_SELECTED_WRITE_ACCESS)

ECOA__return_status
  #component_impl_name#_container__#operation_name#__get_selected_write_access
   (#component_impl_name#__context* context,
    #component_impl_name#_container__#operation_name#_handle* data_handle,
    const ECOA__write_access_mode mode);

#endif /* BINDING_OPTION_SELECTED_WRITE_ACCESS */
```

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.h
 * Container Interface header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

#include "#component_impl_name#_container_types.h"

ECOA__return_status
  #component_impl_name#_container__#operation_name#__cancel_write_access
   (#component_impl_name#__context* context,
    #component_impl_name#_container__#operation_name#_handle* data_handle);
```

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.h
 * Container Interface header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

#include "#component_impl_name#_container_types.h"

ECOA__return_status
   #component_impl_name#_container__#operation_name#__publish_write_access
   (#component_impl_name#__context* context,
    #component_impl_name#_container__#operation_name#_handle* data_handle);
```

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

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.h
 * Container Interface header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

#include "#component_impl_name#_container_types.h"

#if defined(BINDING_OPTION_EXTENDED_VD_API)

ECOA__boolean8
   #component_impl_name#_container__#operation_name#__is_initialized
   (#component_impl_name#__context* context);

#endif /* BINDING_OPTION_EXTENDED_VD_API */
```

11.1.2.8 Release All Data Handles

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.h
 * Container Interface header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

#include "#component_impl_name#_container_types.h"

#if defined(BINDING_OPTION_EXTENDED_VD_API)

void
#component_impl_name#_container__#operation_name#__release_all_data_handles
   (#component_impl_name#__context* context);

#endif /* BINDING_OPTION_EXTENDED_VD_API */
```

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.h
 * Container Interface header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

void #component_impl_name#_container__#operation_name#__send
   (#component_impl_name#__context* context,
    const #event_parameters# );
```

11.2 Properties

This section describes the syntax for the Get_Value operation to request the component properties whose values are fulfilled by the Infrastructure based on elements described in the component implementation XML file.

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.h
 * Container Interface header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

void #component_impl_name#_container__get_#property_name#_value
   (#component_impl_name#__context* context,
    #property_type_name#* value);
```

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

11.3.1.1 Log_Trace

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

void #component_impl_name#_container__log_trace
   (#component_impl_name#__context* context,
    const ECOA__log log);
```

11.3.1.2 Log_Debug

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

void #component_impl_name#_container__log_debug
   (#component_impl_name#__context* context,
    const ECOA__log log);
```

11.3.1.3 Log_Info

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

void #component_impl_name#_container__log_info
   (#component_impl_name#__context* context,
    const ECOA__log log);
```

11.3.1.4 Log_Warning

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

void #component_impl_name#_container__log_warning
   (#component_impl_name#__context* context,
    const ECOA__log log);
```

11.3.1.5 Raise_Error

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

void #component_impl_name#_container__raise_error
   (#component_impl_name#__context* context,
    const ECOA__log log,
    const ECOA__error_code error_code);
```

11.3.1.6 Raise_Fatal_Error

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

void #component_impl_name#_container__raise_fatal_error
   (#component_impl_name#__context* context,
    const ECOA__log log,
    const ECOA__error_code error_code);
```

11.3.2 Alternative 2: using flex interfaces

11.3.2.1 Flex_Log

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

void #component_impl_name#_container__flex_log
```

```
   (#component_impl_name#__context* context,
   ECOA__information_category category,
   const ECOA__char8* log_string, ...);
```

11.3.2.2 Flex_Raise_Fatal_Error

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

void #component_impl_name#_container__flex_raise_fatal_error
   (#component_impl_name#__context* context,
   ECOA__information_category category,
   const ECOA__char8* fatal_error_string, ...);
```

11.4 Time Services

11.4.1 Get_Relative_Local_Time

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

void #component_impl_name#_container__get_relative_local_time
   (#component_impl_name#__context* context,
    ECOA__hr_time *relative_local_time);
```

11.4.2 Get_UTC_Time

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

#if defined(OPTION_UTC_TIME)

ECOA__return_status #component_impl_name#_container__get_UTC_time
   (#component_impl_name#__context* context,
    ECOA__global_time *utc_time);

#endif /* OPTION_UTC_TIME */
```

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.h"
 * Container Interface header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

ECOA__return_status #component_impl_name#_container__get_absolute_system_time
   (#component_impl_name#__context* context,
    ECOA__global_time *absolute_system_time);
```

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.h"
 * Container Interface header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

#if defined(OPTION_TIME_RESOLUTION)

void #component_impl_name#_container__get_relative_local_time_resolution
   (#component_impl_name#__context* context,
    ECOA__duration *relative_local_time_resolution);

#endif /* OPTION_TIME_RESOLUTION */
```

11.4.5 Get_UTC_Time_Resolution

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

#if defined(OPTION_TIME_RESOLUTION)
```

```
void #component_impl_name#_container__get_UTC_time_resolution
   (#component_impl_name#__context* context,
    ECOA__duration *utc_time_resolution);

#endif /* OPTION_TIME_RESOLUTION */
```

11.4.6 Get_Absolute_System_Time_Resolution

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

#if defined(OPTION_TIME_RESOLUTION)

void #component_impl_name#_container__get_absolute_system_time_resolution
   (#component_impl_name#__context* context,
    ECOA__duration *absolute_system_time_resolution);

#endif /* OPTION_TIME_RESOLUTION */
```

11.5 Triggers

11.5.1 Trigger_Set

The C syntax for a component instance to set a trigger is:

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


ECOA__return_status #component_impl_name#_container__#trigger_name#_set
   (#component_impl_name#__context* context,
    const ECOA__duration delay);
```

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

The C syntax for a component instance to cancel a trigger is:

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

ECOA__return_status #component_impl_name#_container__#trigger_name#_cancel
   (#component_impl_name#__context* context);
```

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.h"
 * Container Interface header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

ECOA__return_status #component_impl_name#_container__read_#PINFOname#
   (#component_impl_name#__context* context,
    ECOA__byte *memory_address,
    ECOA__uint32 in_size,
    ECOA__uint32 *out_size);
```

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

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

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

```
#if defined(OPTION_PINFO_WRITE)

ECOA__return_status #component_impl_name#_container__write_#PINFOname#
   (#component_impl_name#__context* context,
    const ECOA__byte *memory_address,
    const ECOA__uint32 in_size,
    ECOA__uint32 *out_size);

#endif /* OPTION_PINFO_WRITE */
```

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.h"
 * Container Interface header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

ECOA__return_status #component_impl_name#_container__seek_#PINFOname#
   (#component_impl_name#__context* context,
    ECOA__int32 offset, ECOA__seek_whence_type whence,
    ECOA__uint32 *new_position);
```

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

The C syntax for a component instance to save its warm start (non-volatile) context is:

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

```
#if defined(OPTION_WARM_START_CONTEXT)

void #component_impl_name#_container__save_warm_start_context
   (#component_impl_name#__context* context);

#endif /* OPTION_WARM_START_CONTEXT */
```

11.8 Supervisor components

This section is specific to [OPTION SUPERVISION].

When the component is of the 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.h"
 * Container Interface header for Component
 * #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

ECOA__return_status #component_impl_name#_container__get_executable_state
   (#component_impl_name#__context* context,
    const ECOA__asset_id id,
    ECOA__executable_state* state);

ECOA__return_status #component_impl_name#_container__executable_command
   (#component_impl_name#__context* context,
    const ECOA__asset_id id,
    const ECOA__executable_command command);
```

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.h"
 * Container Interface header for Component
 * #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

ECOA__return_status #component_impl_name#_container__get_component_state
```

```
    (#component_impl_name#__context* context,
     const ECOA__asset_id id,
     ECOA__component_state* state);

ECOA__return_status #component_impl_name#_container__component_state_command
    (#component_impl_name#__context* context,
     const ECOA__asset_id id,
     const ECOA__component_command command);
```

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.h"
 * Container Interface header for Component
 * #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

ECOA__return_status
#component_impl_name#_container__get_variable_#variable_name#
    (#component_impl_name#__context* context,
     #variable_type# *value);

ECOA__return_status
#component_impl_name#_container__set_variable_#variable_name#
    (#component_impl_name#__context* context,
     const #variable_type# value);
```

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.h
 * Container Types header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

#define ECOA_VERSIONED_DATA_HANDLE_PRIVATE_SIZE 32

/*
 * 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];
} #component_impl_name#_container__#operation_name#_handle;
```

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.h"
 * External Interface header for Component Implementation
 * #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

#if defined(OPTION_EXTERNAL_INTERFACE)

void #component_impl_name#__#external_operation_name#
   (const #event_parameters#);
```

```
#endif /* OPTION_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 added in the Component interface template defined in §6.3 with the following syntax:

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


void #component_impl_name#__external_routine
   (#component_impl_name#__context* context);
```

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

```
/* @file "#component_impl_name#_External_Component_Interface.h"
* Container Interface header for Component
 * #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

ECOA__return_status #component_impl_name#_container__start_external_task
(#component_impl_name#__context* context);

ECOA__return_status  #component_impl_name#_container__stop_external_task
(#component_impl_name#__context* context);
```

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.h
 */

/*  This is a compilable ISO C99 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.                         */


#ifndef ECOA_H
#define ECOA_H

#if defined(__cplusplus)
extern "C" {
#endif /* __cplusplus */

/* ECOA:boolean8 */
typedef unsigned char ECOA__boolean8;
#define ECOA__TRUE  (1)
#define ECOA__FALSE (0)

/* ECOA:int8 */
typedef char ECOA__int8;
#define ECOA__INT8_MIN (-127)
#define ECOA__INT8_MAX ( 127)

/* ECOA:char8 */
typedef char ECOA__char8;
#define ECOA__CHAR8_MIN (0)
#define ECOA__CHAR8_MAX (127)

/* ECOA:byte */
typedef unsigned char ECOA__byte;
#define ECOA__BYTE_MIN (0)
#define ECOA__BYTE_MAX (255)

/* ECOA:int16 */
typedef short int ECOA__int16;
#define ECOA__INT16_MIN (-32767)
```

```
#define ECOA__INT16_MAX ( 32767)

/* ECOA:int32 */
typedef int ECOA__int32;
#define ECOA__INT32_MIN (-2147483647L)
#define ECOA__INT32_MAX ( 2147483647L)

/* ECOA:uint8 */
typedef unsigned char ECOA__uint8;
#define ECOA__UINT8_MIN (0)
#define ECOA__UINT8_MAX (255)

/* ECOA:uint16 */
typedef unsigned short int ECOA__uint16;
#define ECOA__UINT16_MIN (0)
#define ECOA__UINT16_MAX (65535)

/* ECOA:uint32 */
typedef unsigned int ECOA__uint32;
#define ECOA__UINT32_MIN (0LU)
#define ECOA__UINT32_MAX (4294967295LU)

/* ECOA:float32 */
typedef float ECOA__float32;
#define ECOA__FLOAT32_MIN (-3.402823466e+38F)
#define ECOA__FLOAT32_MAX ( 3.402823466e+38F)

/* ECOA:double64 */
typedef double ECOA__double64;
#define ECOA__DOUBLE64_MIN (-1.7976931348623157e+308)
#define ECOA__DOUBLE64_MAX ( 1.7976931348623157e+308)

#if defined(OPTION_INT64)

/* ECOA:int64 */
typedef long long int ECOA__int64;
#define ECOA__INT64_MIN (-9223372036854775807LL)
#define ECOA__INT64_MAX ( 9223372036854775807LL)

#endif /* ECOA_INT64_SUPPORT */

#if defined(OPTION_UINT64)

/* ECOA:uint64 */
```

```
typedef unsigned long long int ECOA__uint64;
#define ECOA__UINT64_MIN (0LLU)
#define ECOA__UINT64_MAX (18446744073709551615LLU)

#endif /* OPTION_UINT64 */

/* ECOA:return_status */
typedef ECOA__uint32 ECOA__return_status;
#define ECOA__return_status_OK                          (0)
#define ECOA__return_status_FAILURE                     (1)
#define ECOA__return_status_INVALID_HANDLE              (2)
#define ECOA__return_status_DATA_NOT_INITIALIZED        (3)
#define ECOA__return_status_NO_DATA                     (4)
#define ECOA__return_status_INVALID_IDENTIFIER          (5)
#define ECOA__return_status_NO_RESPONSE                 (6)
#define ECOA__return_status_OPERATION_ALREADY_PENDING   (7)
#define ECOA__return_status_CLOCK_UNSYNCHRONIZED        (8)
#define ECOA__return_status_RESOURCE_NOT_AVAILABLE      (9)
#define ECOA__return_status_OPERATION_NOT_AVAILABLE     (10)
#define ECOA__return_status_INVALID_PARAMETER           (11)

/* Component and executable identifiers ECOA:asset_id */
typedef ECOA__uint32 ECOA__asset_id;

/* ECOA:write_access_mode */
typedef ECOA__uint32 ECOA__write_access_mode;
#define ECOA__write_access_mode_READ_AND_UPDATE       (0)
#define ECOA__write_access_mode_WRITE_ONLY            (1)

/* ECOA:hr_time */
typedef struct {
   ECOA__uint32 seconds;     /* Seconds */
   ECOA__uint32 nanoseconds; /* Nanoseconds*/
} ECOA__hr_time;

/* ECOA:global_time */
typedef struct {
   ECOA__uint32 seconds;     /* Seconds */
   ECOA__uint32 nanoseconds; /* Nanoseconds*/
} ECOA__global_time;

/* ECOA:duration */
typedef struct {
   ECOA__uint32 seconds;     /* Seconds */
```

```
   ECOA__uint32 nanoseconds; /* Nanoseconds*/
} ECOA__duration;

/* ECOA:log */
#define ECOA__LOG_MAXSIZE (256)
typedef struct {
   ECOA__uint32 current_size;
   ECOA__char8  data[ECOA__LOG_MAXSIZE];
} ECOA__log;

/* ECOA:information_category */
typedef ECOA__uint32 ECOA__information_category;
#define ECOA__information_category_NONE              (0)
#define ECOA__information_category_CRITICAL          (1)
#define ECOA__information_category_ERROR             (2)
#define ECOA__information_category_WARNING           (3)
#define ECOA__information_category_INFO              (4)
#define ECOA__information_category_DEBUG             (5)
#define ECOA__information_category_TRACE             (6)

/* ECOA:error_id */
typedef ECOA__uint32 ECOA__error_id;

/* ECOA:error_code */
typedef ECOA__uint32 ECOA__error_code;

/* ECOA:asset_type */
typedef ECOA__uint32 ECOA__asset_type;
#define ECOA__asset_type_COMPONENT              (0)
#define ECOA__asset_type_EXECUTABLE             (1)

/* ECOA:error_type */
typedef ECOA__uint32 ECOA__error_type;
#define ECOA__error_type_RESOURCE_NOT_AVAILABLE         (0)
#define ECOA__error_type_UNAVAILABLE                    (1)
#define ECOA__error_type_MEMORY_VIOLATION               (2)
#define ECOA__error_type_NUMERICAL_ERROR                (3)
#define ECOA__error_type_ILLEGAL_INSTRUCTION            (4)
#define ECOA__error_type_STACK_OVERFLOW                 (5)
#define ECOA__error_type_DEADLINE_VIOLATION             (6)
#define ECOA__error_type_OVERFLOW                       (7)
#define ECOA__error_type_UNDERFLOW                      (8)
#define ECOA__error_type_ILLEGAL_INPUT_ARGS             (9)
#define ECOA__error_type_ILLEGAL_OUTPUT_ARGS            (10)
```

```
#define ECOA__error_type_ERROR                          (11)
#define ECOA__error_type_FATAL_ERROR                    (12)
#define ECOA__error_type_HARDWARE_FAULT                 (13)
#define ECOA__error_type_POWER_FAIL                     (14)
#define ECOA__error_type_COMMUNICATION_ERROR            (15)
#define ECOA__error_type_INVALID_CONFIG                 (16)
#define ECOA__error_type_INITIALISATION_PROBLEM         (17)
#define ECOA__error_type_CLOCK_UNSYNCHRONIZED           (18)

/* ECOA:pinfo_filename */
#define ECOA__PINFO_FILENAME_MAXSIZE 256

typedef struct {
   ECOA__uint32 current_size;
   ECOA__char8  data[ECOA__PINFO_FILENAME_MAXSIZE];
} ECOA__pinfo_filename;

/* ECOA:seek_whence_type */
typedef ECOA__uint32 ECOA__seek_whence_type;
#define ECOA__seek_whence_type_SEEK_SET (0)
#define ECOA__seek_whence_type_SEEK_CUR (1)
#define ECOA__seek_whence_type_SEEK_END (2)

/* ECOA:component_state */
typedef ECOA__uint32 ECOA__component_state;
#define ECOA__component_state_UNAVAILABLE               (0)
#define ECOA__component_state_IDLE                      (1)
#define ECOA__component_state_READY                     (2)
#define ECOA__component_state_RUNNING                   (3)

/* ECOA:component_command */
typedef ECOA__uint32 ECOA__component_command;
#define ECOA__component_command_INIT                      (O)
#define ECOA__component_command_START                     (1)
#define ECOA__component_command_STOP                      (2)
#define ECOA__component_command_RESET                     (3)
#define ECOA__component_command_SHUTDOWN                  (4)
#define ECOA__component_command_KILL                      (5)

/* ECOA:executable_state */
typedef ECOA__uint32 ECOA__executable_state;
#define ECOA__executable_state_NOT_LAUNCHED             (0)
#define ECOA__executable_state_LAUNCHED                 (1)
```

```
/* ECOA:executable_command */
typedef ECOA__uint32 ECOA__executable_command;
#define ECOA__executable_command_START                (0)
#define ECOA__executable_command_STOP                 (1)

#if defined(__cplusplus)
}
#endif /* __cplusplus */

#endif /* ECOA_H */
```

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