European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 2: C Binding Reference Manual

BAE Ref No: IAWG-ECOA-TR-004
Dassault Ref No: DGT 144477-B

Issue: 2

Prepared by
BAE Systems (Operations) Limited and Dassault Aviation

Note: *This specification represents the output of a research programme and contains mature high-level concepts, though low-level mechanisms and interfaces remain under development and are subject to change. This standard of documentation is recommended as appropriate for limited lab-based evaluation only. Product development based on this standard of documentation is not recommended.*

1 Table of Contents

2 List of Figures

3 List of Tables

4 Abbreviations

API	Application Programming Interface
ECOA	European Component Oriented Architecture
ELI	ECOA Logical Interface
UTC	Coordinated Universal Time
XML	eXtensible Markup Language
XSD	XML Schema Definition

5 Introduction

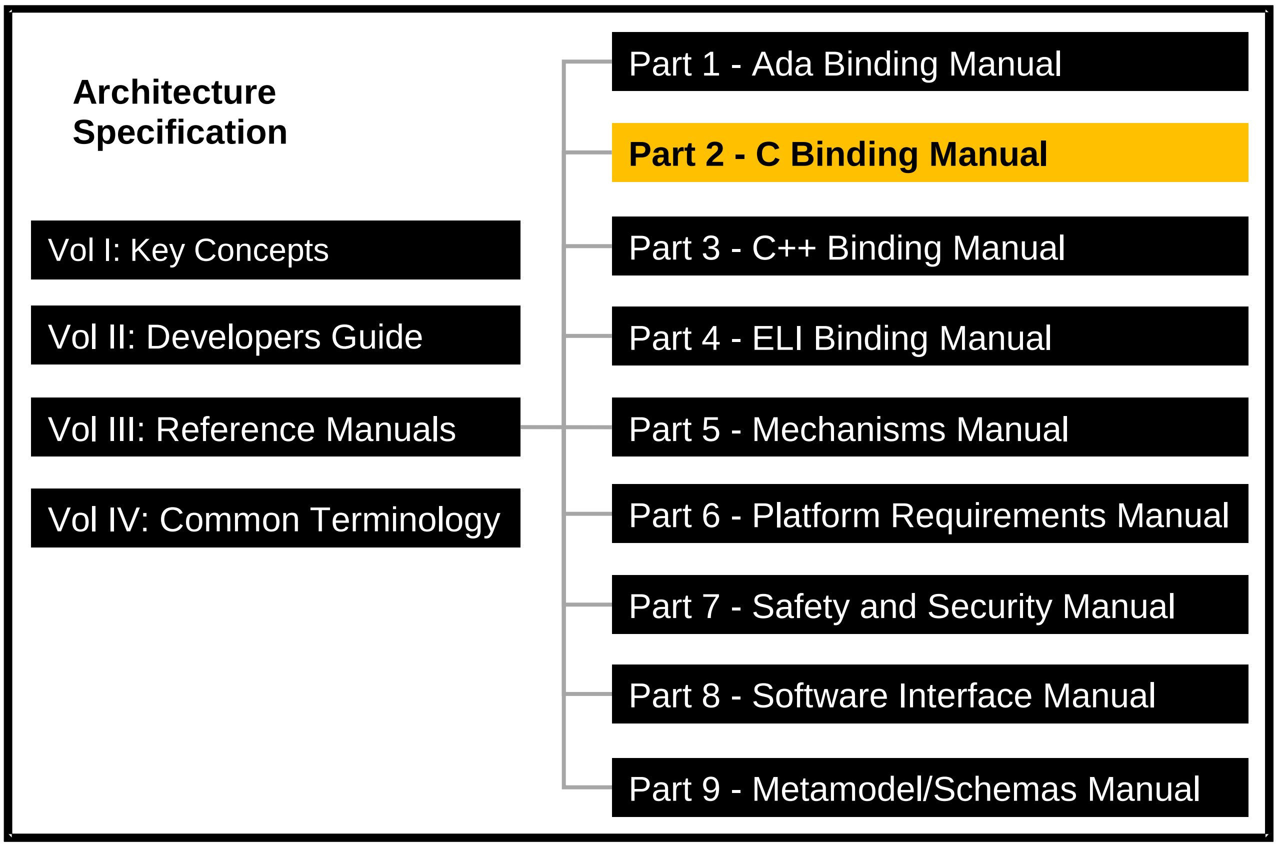
Architecture Specification
Vol I: Key Concepts
Vol II: Developers Guide
Vol III: Reference Manuals
Vol IV: Common Terminology
Part 1 - Ada Binding Manual
Part 2 - C Binding Manual
Part 3 - C++ Binding Manual
Part 4 - ELI Binding Manual
Part 5 - Mechanisms Manual
Part 6 - Platform Requirements Manual
Part 7 - Safety and Security Manual
Part 8 - Software Interface Manual
Part 9 - Metamodel/Schemas Manual

Figure 1 – ECOA Documentation

The Architecture Specification provides the definitive 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 is introduced in Key Concepts (Reference 1) and uses terms defined in the Common Terminology (Reference 11). For this reason, the reader should read these documents, prior to this document. The details of the other documents comprising the rest of the Architecture Specification can be found in Section 12.

The Architecture Specification consists of four volumes, as shown in Figure 1:

- Volume I: Key Concepts
- Volume II: Developer's Guide
- Volume III: Reference Manuals
- Volume IV: Common Terminology

This document comprises Volume III Part 2 of the ECOA Architecture Specification, and describes the C (C standard ISO/IEC 9899:1999 – Reference 12) binding for the module and container APIs that facilitate communication between the module instances and their container in an ECOA system.

The document is structured as follows:

- Section 6 describes the Module to Language Mapping;
- Section 7 describes the method of passing parameters;

- Section 8 describes the Module Context;
- Section 9 describes the pre-defined types that are provided and the types that can be derived from them;
- Section 10 describes the Module Interface;
- Section 11 describes the Container Interface;
- Section 12 provides details of documents referenced from this one.

6 Module to Language Mapping

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

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

It is important to ensure that the names of these functions do not clash within a single protection domain. One way to achieve this is for each component supplier to define the module 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 a protection domain is possible.

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

Filename	Use
#module_impl_name#.h	Module Interface declaration (handlers entry points provided by the module and callable by the container)
#module_impl_name#.c	Module Implementation (implements the module interface)
#module_impl_name#_container.h	Container Interface declaration (functions provided by the container and callable by the module) Module Context type declaration
#module_impl_name#_user_context.h	User extensions to Module Context.

Table 1 – Filename Mapping

Templates for the files in Table 1 are provided below:

6.1 Module Interface Template

```
/*
 * @file #module_impl_name#.h
 * This is the Module Interface header for Module #module_impl_name#
 * This file is generated by the ECOA tools and shall not be modified
 */
```

```
/* Standard Types */
#include “ECOA.h”
/* Additionally created types */
#include #additionally_created_types#
/* Include container header */
#include “#module_impl_name#_Container.h”

/* Event operation handlers specifications */
#list_of_event_operations_specifications#

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

/* Lifecycle operation handlers specifications */
#list_of_lifecycle_operations_specifications#
```

```
/*
 * @file #module_impl_name#.c
 * This is the Module Interface for Module #module_impl_name#
 * This file can be considered a template with the operation stubs
 * autogenerated by the ECOA toolset and filled in by the module
 * developer.
 */

/* Include module interface header */
#include “#module_impl_name#.h”

/* Event operation handlers */
#list_of_event_operations#

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

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

6.2 Container Interface Template

```
/* @file “#module_impl_name#_container.h”
 * This is the Module Container header for Module #module_impl_name#
 * This file is generated by the ECOA tools and shall not be modified
 */

#include “#module_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 #module_impl_name#__platform_hook;

/* Module Context structure declaration */
typedef struct
{
        /*
         * the date of the calling operation
         */
        ECOA__timestamp operation_timestamp;

        /*
         * Other container technical data will accessible through the pointer defined here
         */
        struct #module_impl_name#__platform_hook *platform_hook;

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

} #module_impl_name#__context;
```

```
/* Event operation call specifications */
#event_operation_call_specifications#

/* Request-response call specifications */
#request_response_call_specifications#

/* Versioned data call specifications */
#versioned_data_call_specifications#

/* Functional parameters call specifications */
#propertys_call_specifications#

/* Logging services API call specifications */
#logging_services_call_specifications#

/* Time Services API call specifications */
#time_services_call_specifications#
```

6.3 User Module Context Template

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

/* User Module Context structure example */
typedef struct
{
   /* declare the User Module Context “local” data here */

} #module_impl_name#_user_context;
```

6.4 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. pre-defined, 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 constant pointers, output parameters defined with a complex type will be passed as pointers.

	Input parameter	**Output parameter**
Simple type	By value	Pointer
Complex type	Constant Pointer	Pointer

Table 2 – Method of Passing Parameters

NOTE: 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 Module Context

In the C language, the Module Context is a structure which holds both the user local data (called "User Module Context") and instracture-level technical data (which is implementation dependant). The structure is defined in the Container Interface.

The following shows the C syntax for the Module Context:

```
/* @file "#module_impl_name#_container.h"
 * This is the Module Container header for Module #module_impl_name#
 * This file is generated by the ECOA tools and shall not be modified
 */

#include "#module_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 #module_impl_name#__platform_hook;

/* Module Context structure declaration */
typedef struct
{
        /*
         * the date of the calling operation
         */
        ECOA__timestamp operation_timestamp;

        /*
         * Other container technical data will accessible through the pointer defined here
         */
        struct #module_impl_name#__platform_hook *platform_hook;

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

} #module_impl_name#__context;
```

8.1 User Module Context

The following shows the C syntax for the Module User Context (including an example data item; myCounter):

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

/* User Module Context structure example */
typedef struct
{
   /* declare the User Module Context "local" data here */
   ECOA__int8 myCounter;
} #module_impl_name#_user_context;
```

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

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

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

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

NB: currently, the user extensions to Module Context need to be known by the container in order to allocate the required memory area. This means that the component supplier is requested 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].

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

The Module Context structure will be passed by the Container to the Module as the first parameter for each operation that will activate the Module instance (i.e. received events, received request-response and asynchronous request-response sent call-back). This structure shall be passed by the Module to all Container Interface API functions it can call.

The Module Context will also be used by the Container to automatically timestamp operations on the emitter/requester side using an ECOA-provided attribute called operation_timestamp. The Container also provides a utility function to retrieve this from the Module Context. The way this structure is populated by the ECOA infrastructure is detailed in reference 2.

[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 namespaces, and how the ECOA pre-defined types and derived types are represented in C.

9.1 Filenames and Namespace

The type definitons are contained within one or more namespaces: all types for specific namespace #namespacen# shall be placed in a file called `#namespace1#__#namespace2#__[…]__#namespacen#.h`

The complete name of the declaration of a variable name and type name will be computed by prefixing these names with the names of all the namespaces from the first level to the last level, separated with underscores as illustrated below. In the C language, this naming rule will be used for each variable or type declaration to create the complete variable name, reflecting the namespaces onto which it is defined.

```
/*
 * @file #namespace1#__#namespace2#__[…]__#namespacen#.h
 * This is data-type declaration file
 * This file is generated by the ECOA tools and shall not be modified
 */

#complete_data_type_name# #namespace1#__#namespace2#__[…]__#namespacen#__#variable_name#;
```

9.2 Predefined Types

The predefined types, shown in Table 3, shall be located in the “ECOA” namespace 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 pre-defined Boolean type, that are shown in Table 4.

ECOA Predefined 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:float32	**ECOA__float32**
ECOA:double64	**ECOA__double64**

Table 3 – C Predefined Type Mapping

The data-types in Table 1 are fully defined using the following set of 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**

Table 4 – C Predefined Constants

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

9.2.1 ECOA:error

In C `ECOA:error` translates to `ECOA__error`, with the enumerated values shown below:

```
typedef ECOA__uint32 ECOA__error;
#define ECOA__error_OK (0)
#define ECOA__error_INVALID_HANDLE (1)
#define ECOA__error_DATA_NOT_INITIALIZED (2)
#define ECOA__error_NO_DATA (3)
#define ECOA__error_INVALID_IDENTIFIER (4)
#define ECOA__error_NO_RESPONSE (5)
#define ECOA__error_OPERATION_ABORTED (6)
#define ECOA__error_UNKNOWN_SERVICE_ID (7)
#define ECOA__error_CLOCK_UNSYNCHRONIZED (8)
#define ECOA__error_INVALID_STATE (9)
#define ECOA__error_INVALID_TRANSITION (10)
#define ECOA__error_RESOURCE_NOT_AVAILABLE (11)
#define ECOA__error_OPERATION_NOT_AVAILABLE (12)
```

9.2.2 ECOA:hr_time

The binding for time is:

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

9.2.3 ECOA:global_time

Global time is represented as:

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

9.2.4 ECOA:duration

Duration is represented as:

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

9.2.5 ECOA:timestamp

The following binding shows how the timestamp, for operations etc, is represented in C:

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

9.2.6 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;
```

9.2.7 ECOA:component_states_type

In C `ECOA:component_states_type` translates to `ECOA__component_state_type`, with the enumerated values shown below:

```
typedef ECOA__uint32 ECOA__component_states_type;
#define ECOA__component_states_type_IDLE (0)
#define ECOA__component_states_type_INITIALIZING (1)
#define ECOA__component_states_type_STOPPED (2)
#define ECOA__component_states_type_STOPPING (3)
#define ECOA__component_states_type_RUNNING (4)
#define ECOA__component_states_type_STARTING (5)
#define ECOA__component_states_type_FINISHING (6)
#define ECOA__component_states_type_FAILURE (7)
```

9.2.8 ECOA:module_states_type

In C `ECOA:module_states_type` translates to `ECOA__module_states_type`, with the enumerated values shown below:

```
typedef ECOA__uint32 ECOA__module_states_type;
#define ECOA__module_states_type_IDLE (0)
#define ECOA__module_states_type_READY (1)
#define ECOA__module_states_type_RUNNING (2)
```

9.2.9 ECOA:exception

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

```
typedef struct
{
   ECOA__timestamp timestamp;
   ECOA__service_id service_id;
   ECOA__operation_id operation_id;
   ECOA__module_id module_id;
   ECOA__exception_id exception_id;
} ECOA__exception;
```

The types used in the `ECOA:exception` record are defined below:

9.2.9.1 ECOA:service_id

In C the syntax for a `ECOA:service_id` is:

```
typedef ECOA__uint32 ECOA__service_id;
```

9.2.9.2 ECOA:operation_id

In C the syntax for a `ECOA:operation_id` is:

```
typedef ECOA__uint32 ECOA__operation_id;
```

9.2.9.3 ECOA:module_id

In C the syntax for a `ECOA:module_id` is:

```
typedef ECOA__uint32 ECOA__module_id;
```

9.2.9.4 ECOA:exception_id

In C `ECOA:exception_id` translates to `ECOA__exception_id`, with the enumerated values shown below:

```
typedef ECOA__uint32 ECOA__exception_id;
#define ECOA__exception_id_NO_RESPONSE (1)
#define ECOA__exception_id_OPERATION_ABORTED (2)
#define ECOA__exception_id_RESOURCE_NOT_AVAILABLE (3)
#define ECOA__exception_id_OPERATION_NOT_INVOKED (4)
#define ECOA__exception_id_ILLEGAL_INPUT_ARGS (5)
#define ECOA__exception_id_ILLEGAL_OUTPUT_ARGS (6)
#define ECOA__exception_id_MEMORY_VIOLATION (7)
#define ECOA__exception_id_DIVISION_BY_ZERO (8)
#define ECOA__exception_id_FLOATING_POINT_EXCEPTION (9)
#define ECOA__exception_id_ILLEGAL_INSTRUCTION (10)
#define ECOA__exception_id_STACK_OVERFLOW (11)
#define ECOA__exception_id_HARDWARE_FAULT (12)
#define ECOA__exception_id_POWER_FAIL (13)
#define ECOA__exception_id_COMMUNICATION_ERROR (14)
#define ECOA__exception_id_DEADLINE_VIOLATION (15)
#define ECOA__exception_id_OVERFLOW_EXCEPTION (16)
#define ECOA__exception_id_UNDERFLOW_EXCEPTION (17)
#define ECOA__exception_id_OPERATION_OVERRATED (18)
#define ECOA__exception_id_OPERATION_UNDERRATED (19)
```

9.3 Derived Types

This Section describes the derived types that can be constructed from the ECOA pre-defined types.

9.3.1 Simple Types

The syntax for defining a Simple Type #simple_type_name# refined from a Predefined Type #predef_type_name# in C is defined below. Note that as namespaces are not supported in the C language, the actual name of the type (known as the complete type (see para. 9.1) and referred to here by prefixing `complete_`) will be computed by prefixing the namespaces in which it is included as described previously.

```
typedef #predef_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 Constants

The sytax for the declaration of a Constant called “`#contant_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 prefixing the namespaces in which it is included as described previously.

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

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

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

```
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` or `ECOA__uint32`.
- `#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).

9.3.4 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. Note that as namespaces are not supported in the C language, the actual name of the type (known as the complete type (see para. 9.1) and referred to here by prefixing `complete_`) will be computed by prefixing the namespaces in which it is included as described previously. The order of fieds 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#;
```

9.3.5 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.
Note that as namespaces are not supported in the C language, the actual name of the type (known as the complete type (see para. 9.1) and referred to here by prefixing `complete_`) will be computed by prefixing the namespaces in which it is included as described previously.

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 predefined 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#;
```

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

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

```
#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#;
```

10 Module Interface

10.1 Operations

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

10.1.1 Request-response

10.1.1.1 Request Received Immediate Response

The following is the C syntax for invoking a request received by a module instance when an immediate response is required, where #module_impl_name# is the name of the module implementation providing the service and #operation_name# is the operation name. The same syntax is applicable for both synchronous and asynchronous request-response operations.

```
/*
 * @file #module_impl_name#.h
 * This is the Module Interface header for Module #module_impl_name#
 * This file is generated by the ECOA tools and shall not be modified
 */
void #module_impl_name#__#operation_name#__request_received(#module_impl_name#__context* context,
const #parameters_in#, #parameters_out#);
```

10.1.1.2 Request Received Deferred Response

The following is the C syntax for invoking a request received by a module instance when a deferred response is required, where #module_impl_name# is the name of the module implementation providing the service and #operation_name# is the operation name. The same syntax is applicable for both synchronous and asynchronous request-response operations.

```
/*
 * @file #module_impl_name#.h
 * This is the Module Interface header for Module #module_impl_name#
 * This file is generated by the ECOA tools and shall not be modified
 */
void #module_impl_name#__#operation_name#__request_received_deferred(#module_impl_name#__context*
context, const ECOA__uint32 ID, const #parameters_in#);
```

10.1.1.3 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 module instance that originally issued the request, where #module_impl_name# is the name of the module implementation providing the service and #operation_name# is the operation name. (The reply to a synchronous request response is provided by the return of the original request).

```
* @file #module_impl_name#.h
 * This is the Module Interface header for Module #module_impl_name#
 * This file is generated by the ECOA tools and shall not be modified
 */
void #module_impl_name#__#operation_name#__response_received(#module_impl_name#__context* context,
const ECOA__uint32 ID, const ECOA__error status, const #parameters_out#);
```

NOTE: the “#parameters_out# are the ‘out’ parameters of the original procedure and are passed as “const” parameters, so they are not modified by the container.

10.1.2 Versioned Data

10.1.2.1 Updated

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

```
void #module_impl_name#__#operation_name#__updated(#module_impl_name#__context* context,
#module_impl_name#_container__#operation_name#_handle* data_handle);
```

10.1.3 Events

10.1.3.1 Received

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

```
/*
 * @file #module_impl_name#.h
 * This is the Module Interface header for Module #module_impl_name#
 * This file is generated by the ECOA tools and shall not be modified
 */
void #module_impl_name#__#operation_name#__received(#module_impl_name#__context* context, const
#parameters# );
```

10.2 Component Lifecycle

This section describes the module operations that are used to perform the required component lifecycle activities.

10.2.1 Supervision Module Component Lifecycle API

The Component Lifecycle Service is provided by the supervision module of a component, and requires the supervision module to provide the functionality for the following operations.

10.2.1.1 Initialize Component

The following is the C syntax for the intialize component event received by a supervision module instance.

```
void
#supervision_module_implementation_name#__initialize_component__received(#supervision_module_imple
mentation_name#__context);
```

10.2.1.2 Stop Component

The following is the C syntax for the stop component event received by a supervision module instance.

```
void
#supervision_module_implementation_name#__stop_component__received(#supervision_module_implementat
ion_name#__context);
```

10.2.1.3 Restart Component

The following is the C syntax for the restart component event received by a supervision module instance.

```
void
#supervision_module_implementation_name#__restart_component__received(#supervision_module_implemen
tation_name#__context);
```

10.2.1.4 Reset Component

The following is the C syntax for the reset component event received by a supervision module instance.

```
void
#supervision_module_implementation_name#__reset_component__received(#supervision_module_implementa
tion_name#__context);
```

10.2.1.5 Shutdown Component

The following is the C syntax for the shutdown component event received by a supervision module instance.

```
void
#supervision_module_implementation_name#__shutdown_component__received(#supervision_module_impleme
ntation_name#__context);
```

10.2.1.6 Start Component

The following is the C syntax for the start component event received by a supervision module instance.

```
void
#supervision_module_implementation_name#__start_component__received(#supervision_module_implementa
tion_name#__context);
```

10.3 Module Lifecycle

This section describes the module operations that are used to perform the required module lifecycle activities.

10.3.1 Generic Module API

The following operations are applicable to supervision, non-supervision, trigger and dynamic-trigger module instances.

10.3.1.1 Initialize_Received

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

```
/*
 * @file #module_impl_name#.h
 * This is the Module Interface header for Module #module_impl_name#
 * This file is generated by the ECOA tools and shall not be modified
 */

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

10.3.1.2 Start_Received

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

```
/*
 * @file #module_impl_name#.h
 * This is the Module Interface header for Module #module_impl_name#
 * This file is generated by the ECOA tools and shall not be modified
 */

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

10.3.1.3 Stop_Received

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

```
/*
 * @file #module_impl_name#.h
 * This is the Module Interface header for Module #module_impl_name#
 * This file is generated by the ECOA tools and shall not be modified
 */

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

10.3.1.4 Shutdown_Received

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

```
/*
 * @file #module_impl_name#.h
 * This is the Module Interface header for Module #module_impl_name#
 * This file is generated by the ECOA tools and shall not be modified
 */void #module_impl_name#__SHUTDOWN__received(#module_impl_name#__context* context);
```

10.3.1.5 Reinitialize_Received

The C syntax for an operation to reinitialise a module instance is:

```
/*
 * @file #module_impl_name#.h
 * This is the Module Interface header for Module #module_impl_name#
 * This file is generated by the ECOA tools and shall not be modified
 */
void #module_impl_name#__REINITIALIZE__received(#module_impl_name#__context* context);
```

10.3.2 Supervision Module API

The C syntax for an operation that is used by the container to notify the supervision module that a module/trigger/dynamic trigger has changed state is:

```
/*
 * @file #supervision_module_impl_name#.h
 * This is the Module Interface header for Supervision Module #supervision_module_impl_name#
 * This file is generated by the ECOA tools and shall not be modified
 */
void
#supervision_module_impl_name#__lifecycle_notification__#module_instance_name#(#module_impl_name#_
_context* context , ECOA__module_states_type previous_state , ECOA__module_states_type new_state);
```

Note: the supervision module API will contain a Lifecycle Notification procedure for every module/trigger/dynamic trigger in the Component i.e. the above API will be duplicated for every #module_instance_name# module/trigger/dynamic trigger in the Component. `ECOA.Module_States_Type` is an enumerated type that contains all of the possible lifecycle states of the module instance.

10.4 Service Availability

This section contains details of the operations which allow the container to notify the supervision module of a client component about changes to the availability of required services.

10.4.1 Service Availability Changed

The following is the C syntax for an operation used by the container to invoke a service availability changed operation to a supervision module instance. The operation will only be available if the component has one or more required services. The reference_id type is an enumeration type defined in the Container Interface (Section 11.5.4).

```
/*
 * @file #module_impl_name#.h
 * This is the Module Interface header for Module #module_impl_name#
 * This file is generated by the ECOA tools and shall not be modified
 */
void #supervision_module_impl_name#__service_availability_changed(#supervision_module_impl_name
#__context* context, #supervision_module_impl_name#_container__reference_id instance,
ECOA__boolean8 available);
```

10.4.2 Service Provider Changed

The following is the C syntax for an operation used by the container to invoke a service provider changed operation to a supervision module instance. The operation will only be available if the component has one or more required services. The reference_id type is an enumeration type defined in the Container Interface (Section 11.5.4).

```
/*
 * @file #module_impl_name#.h
 * This is the Module Interface header for Module #module_impl_name#
 * This file is generated by the ECOA tools and shall not be modified
 */
void #supervision_module_impl_name#__service_provider_changed(#supervision_module_impl_name
#__context* context, #supervision_module_impl_name#_container__reference_id instance);
```

10.5 Error Handling

The C syntax for the container to report an error to a supervision module instance is:

```
/*
 * @file #supervision_module_impl_name#.h
 * This is the Module Interface header for the Supervision Module #supervision_module_impl_name#
 * This file is generated by the ECOA tools and shall not be modified
 */
void #supervision_module_impl_name#__exception_notification_handler(#module_impl_name#__context*
context, const ECOA__exception* exception);
```

11 Container Interface

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

11.1 Operations

11.1.1 Request Response

11.1.1.1 Reply Deferred

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

```
 * @file #module_impl_name#_container.h
 * This is the Container Interface header for Module #module_impl_name#
 * This file is generated by the ECOA tools and shall not be modified */

ECOA__error #module_impl_name#_container
__#operation_name#__reply_deferred(#module_impl_name#__context* context, const ECOA__uint32 ID,
const #parameters_out#);
```

Note: the “#parameters_out# in the above code snippet are the out parameters of the original request, not of this operation: they are passed as ‘const’ values, as they should not be modified by the container. The ID parameter is that which is passed in during the invocation of the request received deferred operation.

11.1.1.2 Synchronous Request

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

```
/*
 * @file #module_impl_name#_container.h
 * This is the Container Interface header for Module #module_impl_name#
 * This file is generated by the ECOA tools and shall not be modified
 */
ECOA__error
#module_impl_name#_container__#operation_name#__request_sync(#module_impl_name#__context* context,
const #parameters_in#, #parameters_out#);
```

11.1.1.3 Asynchronous Request

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

```
/*
 * @file #module_impl_name#_container.h
 * This is the Container Interface header for Module #module_impl_name#
 * This file is generated by the ECOA tools and shall not be modified
 */
ECOA__error
#module_impl_name#_container__#operation_name#__request_async(#module_impl_name#__context*
context, ECOA__uint32* ID, const #parameters_in#);
```

11.1.2 Versioned Data

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

- Get (request) Read Access
- Release Read Access

- Get (request) Write Access
- Cancel Write Access (without writing new data)
- Publish (write) new data (automatically releases write access)

11.1.2.1 Get Read Access

```
/*
 * @file #module_impl_name#_container.h
 * This is the Container Interface header for Module #module_impl_name#
 * This file is generated by the ECOA tools and shall not be modified
 */

#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 {
   #type_name#* data;   /* pointer to the local copy of the data */
   ECOA__timestamp timestamp; /* date of the last update of that version of the data */
   ECOA__byte platform_hook[ECOA_VERSIONED_DATA_HANDLE_PRIVATE_SIZE]; /* technical info associated
with the data (opaque for the user, reserved for the infrastructure) */
} #module_impl_name#_container__#operation_name#_handle;

ECOA__error
#module_impl_name#_container__#operation_name#__get_read_access(#module_impl_name#__context*
context, #module_impl_name#_container__#operation_name#_handle* data_handle);
```

11.1.2.2 Release Read Access

```
ECOA__error
#module_impl_name#_container__#operation_name#__release_read_access(#module_impl_name#__context*
context, #module_impl_name#_container__#operation_name#_handle* data_handle);
```

11.1.2.3 Get Write Access

```
/*
 * @file #module_impl_name#_container.h
 * This is the Container Interface header for Module #module_impl_name#
 * This file is generated by the ECOA tools and shall not be modified
 */

#define ECOA_VERSIONED_DATA_HANDLE_PRIVATE_SIZE 32

typedef struct {
   #type_name#* data;
   ECOA__timestamp timestamp;
   ECOA__byte platform_hook[ECOA_VERSIONED_DATA_HANDLE_PRIVATE_SIZE];
} #module_impl_name#_container__#operation_name#_handle;


ECOA__error
#module_impl_name#_container__#operation_name#__get_write_access(#module_impl_name#__context*
context, #module_impl_name#_container__#operation_name#_handle* data_handle);
```

11.1.2.4 Cancel Write Access

```
ECOA__error
#module_impl_name#_container__#operation_name#__cancel_write_access(#module_impl_name#__context*
context, #module_impl_name#_container__#operation_name#_handle* data_handle);
```

11.1.2.5 Publish Write Access

```
ECOA__error
#module_impl_name#_container__#operation_name#__publish_write_access(#module_impl_name#__context*
context, #module_impl_name#_container__#operation_name#_handle* data_handle);
```

11.1.3 Events

11.1.3.1 Send

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

```
* @file #module_impl_name#_container.h
 * This is the Container Interface header for Module #module_impl_name#
 * This file is generated by the ECOA tools and shall not be modified
 */
ECOA__error #module_impl_name#_container__#operation_name#__send(#module_impl_name#__context*
context, const #parameters# );
```

11.2 Properties

This section describes the syntax for the Get_Value operation to request the module properties.

11.2.1 Get Value

The syntax for Get_Value is shown below, where

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

```
/*
 * @file #module_impl_name#_container.h
 * This is the Container Interface header for Module #module_impl_name#
 * This file is generated by the ECOA tools and shall not be modified
 */
void #module_impl_name#_container__get_#property_name#_value(#module_impl_name#__context* context,
#property_type_name#* value);
```

11.3 Component Lifecycle

This section describes the container operations that are used to perform the required component lifecycle activities.

11.3.1 Supervision Module Component Lifecycle API

The Container Interface provides functionality to allow the supervision module to manage the component lifecycle.

11.3.1.1 Component Initialized

The C syntax for a supervision module instance to perform an initialized event send operation is:

```
ECOA__error
#supervision_module_implementation_name#_container__component_initialized__send(#supervision_modul
e_implementation_name#__context);
```

11.3.1.2 Component Started

The C syntax for a supervision module instance to perform a started event send operation is:

```
ECOA__error
#supervision_module_implementation_name#_container__component_started__send(#supervision_module_im
plementation_name#__context);
```

11.3.1.3 Component Stopped

The C syntax for a supervision module instance to perform a stopped event send operation is:

```
ECOA__error
#supervision_module_implementation_name#_container__component_stopped__send(#supervision_module_im
plementation_name#__context);
```

11.3.1.4 Component Idle

The C syntax for a supervision module instance to perform an idle event send operation is:

```
ECOA__error
#supervision_module_implementation_name#_container__component_idle__send(#supervision_module_imple
mentation_name#__context);
```

11.3.1.5 Component Failed

The C syntax for a supervision module instance to perform a failed event send operation is:

```
ECOA__error
#supervision_module_implementation_name#_container__component_failed__send(#supervision_module_imp
lementation_name#__context);
```

11.3.1.6 Component State

The C syntax for the component state is:

```
#define ECOA_VERSIONED_DATA_HANDLE_PRIVATE_SIZE 32
typedef struct {
   ECOA__component_states_type* data;
   ECOA__timestamp timestamp;
   ECOA__byte platform_hook[ECOA_VERSIONED_DATA_HANDLE_PRIVATE_SIZE];
} #supervision_module_impl_name#_container__component_state_handle;
```

The C syntax for the operations that manage the component state is:

```
ECOA__error
#supervision_module_impl_name#_container__component_state__get_read_access(#supervision_module_imp
l_name#_container__context, #supervision_module_impl_name#_container__component_state_handle*
data_handle);

ECOA__error
#supervision_module_impl_name#_container__component_state__release_read_access(#supervision_module
_impl_name#_container__context, #supervision_module_impl_name#_container__component_state_handle*
data_handle);
```

```
ECOA__error
#supervision_module_impl_name#_container__component_state__get_write_access(#supervision_module_im
pl_name#_container__context, #supervision_module_impl_name#_container__component_state_handle*
data_handle);

ECOA__error
#supervision_module_impl_name#_container__component_state__cancel_write_access(#supervision_module
_impl_name#_container__context, #supervision_module_impl_name#_container__component_state_handle*
data_handle);

ECOA__error
#supervision_module_impl_name#_container__component_state__publish_write_access(#supervision_modul
e_impl_name#_container__context, #supervision_module_impl_name#_container__component_state_handle*
data_handle);
```

The C syntax for the operation that publishes the component state and sends the event is:

```
ECOA__error
#supervision_module_impl_name#_container__set_component_state(#supervision_module_impl_name#_conta
iner__context, ECOA__component_states_type state);
```

11.4 Module Lifecycle

This section describes the container operations that are used to perform the required module lifecycle activities.

11.4.1 Non-Supervision Container API

Container operations are only available to supervision modules to allow them to manage the module lifecycle of non-supervision modules.

11.4.2 Supervision Container API

The C Syntax for the operations that are called by the supervision module to request the container to command a module/trigger/dynamic trigger instance to change (lifecycle) state is:

```
/*
 * @file #supervision_module_impl_name#_container.h
 * This is the Container Interface header for Supervision Module #supervision_module_impl_name#
 * container
 * This file is generated by the ECOA tools and shall not be modified
 */

void #supervision_module_impl_name#_container__get_lifecycle_state__#module_instance_name#
(#module_impl_name#__context* context , ECOA__module_states_type* current_state);

ECOA__error
#supervision_module_impl_name#_container__STOP__#module_instance_name#(#module_impl_name#__context
* context);
ECOA__error
#supervision_module_impl_name#_container__START__#module_instance_name#(#module_impl_name#__contex
t* context);
ECOA__error
#supervision_module_impl_name#_container__INITIALIZE__#module_instance_name#(#module_impl_name#__c
ontext* context);
ECOA__error
#supervision_module_impl_name#_container__SHUTDOWN__#module_instance_name#(#module_impl_name#__con
text* context);
```

An instance of each of the above operations is created for each module/trigger/dynamic trigger instance in the component, where #module_instance_name# above represents the name of the module/trigger/dynamic trigger instance.

11.5 Service Availability

This section contains details of the operations which allow supervision modules to set the availability of provided services or get the availability of required services.

11.5.1 Set Service Availability (Server Side)

The following is the C syntax for invoking the set service availability operation by a supervision module instance. The operation will only be available if the component has one or more provided services. The service instance is identified by the enumeration type service_id defined in the Container Interface (Section 11.5.3).

```
* @file #module_impl_name#_container.h
 * This is the Container Interface header for Module #module_impl_name#
 * This file is generated by the ECOA tools and shall not be modified */

ECOA__error
#supervision_module_impl_name#_container__set_service_availability(#supervision_module_impl_name#_
_context* context, #supervision_module_impl_name#_container__service_id instance, ECOA__boolean8
available);
```

11.5.2 Get Service Availability (Client Side)

The following is the C syntax for invoking the get service availability operation by a supervision module instance. The operation will only be available if the component has one or more required services. The service instance is identified by the enumeration type reference_id defined in the Container Interface (Section 11.5.4).

```
* @file #module_impl_name#_container.h
 * This is the Container Interface header for Module #module_impl_name#
 * This file is generated by the ECOA tools and shall not be modified */

ECOA__error #supervision_module_impl_name#_container__get_service_availability
(#supervision_module_impl_name#__context* context,
#supervision_module_impl_name#_container__reference_id instance, ECOA__boolean8* available);
```

11.5.3 Service ID Enumeration

In C `service_id` translates to `#supervision_module_impl_name#_container__service_id`.

This enumeration has a value for each element <*service/*> defined in the file .componentType, whose name is given by its attribute *name* and the numeric value is the position (starting by 0). The service_id enumeration is only available if the component provides one or more services.

```
typedef ECOA__uint32 #supervision_module_impl_name#_container__service_id;
#define #supervision_module_impl_name#_container__service_id__#service_instance_name# (0)
```

11.5.4 Reference ID Enumeration

In C `reference_id` translates to `#supervision_module_impl_name#_container__reference_id`.

This enumeration has a value for each element <*reference/*> defined in the file .componentType, whose name is given by its attribute *name* and the numeric value is the position (starting by 0). The reference_id enumeration is only available if the component requires one or more services.

```
typedef ECOA__uint32 #supervision_module_impl_name#_container__reference_id;
#define #supervision_module_impl_name#_container__reference_id__#reference_instance_name# (0)
```

11.6 Logging and Fault Management

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 module 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

11.6.1 Log_Trace Binding

```
/* @file “#module_impl_name#_container.h”
 * This file is generated by the ECOA tools and shall not be modified
 */
void #module_impl_name#_container__log_trace(#module_impl_name#__context* context, const ECOA__log
log);
```

11.6.2 Log_Debug Binding

```
/* @file “#module_impl_name#_container.h”
 * This file is generated by the ECOA tools and shall not be modified
 */
void #module_impl_name#_container__log_debug(#module_impl_name#__context* context, const ECOA__log
log);
```

11.6.3 Log_Info Binding

```
* @file “#module_impl_name#_container.h”
 * This file is generated by the ECOA tools and shall not be modified
 */
void #module_impl_name#_container__log_info(#module_impl_name#__context* context, const ECOA__log
log);
```

11.6.4 Log_Warning Binding

```
/* @file “#module_impl_name#_container.h”
 * This file is generated by the ECOA tools and shall not be modified
 */
void #module_impl_name#_container__log_warning(#module_impl_name#__context* context, const
ECOA__log log);
```

11.6.5 Raise_Error Binding

```
/* @file “#module_impl_name#_container.h”
 * This file is generated by the ECOA tools and shall not be modified
 */
void #module_impl_name#_container__raise_error(#module_impl_name#__context* context, const
ECOA__log log);
```

11.6.6 Raise_Fatal_Error Binding

```
/* @file “#module_impl_name#_container.h”
```

```
 * This file is generated by the ECOA tools and shall not be modified
 */
void #module_impl_name#_container__raise_fatal_error(#module_impl_name#__context* context, const
ECOA__log log);
```

11.7 Time Services

This section contains the C syntax for the time services provided to module instances by the container.

11.7.1 Get_Relative_Local_Time

```
/* @file “#module_impl_name#_container.h”
 * This file is generated by the ECOA tools and shall not be modified
 */
ECOA__error #module_impl_name#_container__get_relative_local_time(#module_impl_name#__context*
context, ECOA__hr_time *relative_local_time);
```

11.7.2 Get_UTC_Time

```
/* @file “#module_impl_name#_container.h”
 * This file is generated by the ECOA tools and shall not be modified
 */
ECOA__error #module_impl_name#_container__get_UTC_time(#module_impl_name#__context* context,
ECOA__global_time *utc_time);
```

11.7.3 Get_Absolute_System_Time

```
/* @file “#module_impl_name#_container.h”
 * This file is generated by the ECOA tools and shall not be modified
 */
ECOA__error #module_impl_name#_container__get_absolute_system_time(#module_impl_name#__context*
context, ECOA__global_time *absolute_system_time);
```

11.7.4 Get_Relative_Local_Time_Resolution

```
/* @file “#module_impl_name#_container.h”
 * This file is generated by the ECOA tools and shall not be modified
 */
void #module_impl_name#_container__get_relative_local_time_resolution
(#module_impl_name#__context* context, ECOA__duration *relative_local_time_resolution);
```

11.7.5 Get_UTC_Time_Resolution

```
/* @file “#module_impl_name#_container.h”
 * This file is generated by the ECOA tools and shall not be modified
 */
void #module_impl_name#_container__get_UTC_time_resolution(#module_impl_name#__context* context,
ECOA__duration *utc_time_resolution);
```

11.7.6 Get_Absolute_System_Time_Resolution

```
/* @file “#module_impl_name#_container.h”
 * This file is generated by the ECOA tools and shall not be modified
 */
void
#module_impl_name#_container__get_absolute_system_time_resolution(#module_impl_name#__context*
context, ECOA__duration *absolute_system_time_resolution);
```

12 References

Ref.	Document Number	Version	Title
1.	IAWG-ECOA-TR-001	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume I Key Concepts
2.	IAWG-ECOA-TR-002	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume II Developers Guide
3.	IAWG-ECOA-TR-003	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 1: Ada Binding Reference Manual
4.	IAWG-ECOA-TR-005	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 3: C++ Binding Reference Manual
5.	IAWG-ECOA-TR-006	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 4: ELI and Transport Binding Reference Manual
6.	IAWG-ECOA-TR-007	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 5: Mechanisms Reference Manual
7.	IAWG-ECOA-TR-008	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 6: Platform Requirements Reference Manual
8.	IAWG-ECOA-TR-009	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 7: Approach to Safety and Security Reference Manual
9.	IAWG-ECOA-TR-010	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 8: Software Interface Reference Manual
10.	IAWG-ECOA-TR-011	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 9: Metamodel and XSD Schemas Reference Manual
11.	IAWG-ECOA-TR-012	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume IV Common Terminology

Table 5 - Table of ECOA references

Ref.	Document Number	Version	Title
12.	ISO/IEC 9899	1999	ISO/IEC Standard, Programming Languages - C

Table 6 – Table of External References