European Component Oriented Architecture (ECOA®) Collaboration Programme: Architecture Specification Part 8: C Language Binding

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

Issue: 6

Prepared by
BAE Systems (Operations) Limited and Dassault Aviation

This specification is developed by BAE SYSTEMS, Dassault Aviation, Bull SAS, Thales Systèmes Aéroportés, GE Aviation Systems Limited, General Dynamics United Kingdom Limited and Leonardo MW Ltd and the copyright is owned by BAE SYSTEMS, Dassault Aviation, Bull SAS, Thales Systèmes Aéroportés, GE Aviation Systems Limited, General Dynamics United Kingdom Limited and Leonardo MW Ltd. 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.

Note: *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 should rely on the DefStan or BNAE publications of the ECOA standard.*

Contents

Figures

Tables

0 Introduction

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

This document is Part 8 of the Architecture Specification, and describes the C (ref ISO/IEC 9899:1999(E)) language binding for the module and container APIs that facilitate communication between the module instances and their container in an ECOA® system.

This 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 basic 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 describes the Container Types;
- Section 13 describes the External Interface;
- Section 14 describes the Default Values;
- Section 15 describes Trigger Instances;
- Section 16 describes Dynamic Trigger Instances;
- Section 17 provides a reference C header for the ECOA® namespace, 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 should rely on the DefStan or BNAE publications of the ECOA standard.

3 Normative References

Architecture Specification Part 1	IAWG-ECOA-TR-001 / DGT 144474 Issue 6 Architecture Specification Part 1 – Concepts
Architecture Specification Part 2	IAWG-ECOA-TR-012 / DGT 144487 Issue 6 Architecture Specification Part 2 – Definitions
Architecture Specification Part 3	IAWG-ECOA-TR-007 / DGT 144482 Issue 6 Architecture Specification Part 3 – Mechanisms
Architecture Specification Part 4	IAWG-ECOA-TR-010 / DGT 144485 Issue 6 Architecture Specification Part 4 – Software Interface
Architecture Specification Part 5	IAWG-ECOA-TR-008 / DGT 144483 Issue 6 Architecture Specification Part 5 – High Level Platform Requirements
Architecture Specification Part 6	IAWG-ECOA-TR-006 / DGT 144481 Issue 6 Architecture Specification Part 6 – ECOA® Logical Interface
Architecture Specification Part 7	IAWG-ECOA-TR-011 / DGT 144486 Issue 6 Architecture Specification Part 7 – Metamodel
Architecture Specification Part 8	IAWG-ECOA-TR-004 / DGT 144477 Issue 6 Architecture Specification Part 8 – C Language Binding
Architecture Specification Part 9	IAWG-ECOA-TR-005 / DGT 144478 Issue 6 Architecture Specification Part 9 – C++ Language Binding
Architecture Specification Part 10	IAWG-ECOA-TR-003 / DGT 144476 Issue 6 Architecture Specification Part 10 – Ada Language Binding

Architecture Specification Part 11	IAWG-ECOA-TR-031 / DGT 154934 Issue 6 Architecture Specification Part 11 – High Integrity Ada Language Binding
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

This specification is developed by BAE Systems (Operations) Limited, Dassault Aviation, Bull SAS, Thales Systèmes Aéroportés, GE Aviation Systems Limited, General Dynamics United Kingdom Limited and Leonardo MW Ltd and the copyright is owned by BAE Systems (Operations) Limited, Dassault Aviation, Bull SAS, Thales Systèmes Aéroportés, GE Aviation Systems Limited, General Dynamics United Kingdom Limited and Leonardo MW Ltd. 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.

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__”.

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

It is important to ensure that the names of these functions and types 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 requests or received asynchronous responses).

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

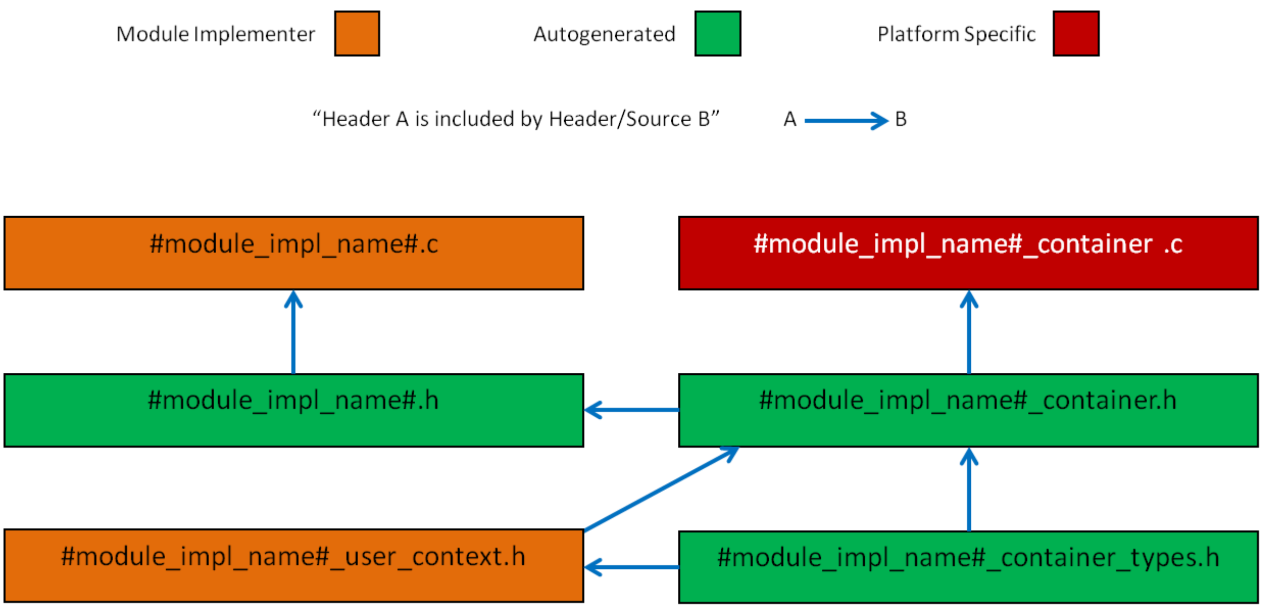
Module Implementer
Autogenerated
Platform Specific
“Header A is included by Header/Source B”
A
B
#module_impl_name#.c
#module_impl_name#_container .c
#module_impl_name#.h
#module_impl_name#_container.h
#module_impl_name#_user_context.h
#module_impl_name#_container_types.h

Figure 1 C Files Organization

Table 1 Filename Mapping

Filename	Use
#module_impl_name#.h	Module Interface declaration (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#_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.
#module_impl_name#_container_types.h	Container Types declaration (container-level data types usable by the module) These types are related to the Container for a Module Implementation and are declared in the **#module_impl_name#_container** namespace.

Filename	Use
#module_impl_name#_user_context.h	User extensions to Module Context. These types are related to the Module Implementation and are declared within the **#module_impl_name#** namespace.

Templates for the files in Table 1are provided in the following sections:

6.1 Module Interface Template

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

#if !defined(#MODULE_IMPL_NAME#_H)
#define #MODULE_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 “#module_impl_name#_container.h”
/* Include container types */
#include “#module_impl_name#_container_types.h”

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

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

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

void #module_impl_name#__SHUTDOWN__received
   (#module_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#

/* Error notification handler specification if this module is a */
/* Fault Handler */
#error_notification_operation_specification#

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

#endif  /* #MODULE_IMPL_NAME#_H */
```

```
/*
 * @file #module_impl_name#.c
 * Module Interface for Module #module_impl_name#
 * This file can be considered a template with the operation stubs
 * auto generated 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#

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

/* Lifecycle operation handlers */
#list_of_lifecycle_operations#

/* Error notification handler if this module is a Fault Handler */
#error_notification_operation#
```

6.2 Container Interface Template

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

#if !defined(#MODULE_IMPL_NAME#_CONTAINER_H)
#define #MODULE_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 “#module_impl_name#_container_types.h”
/* Optional User Context: the “#module_impl_name#_user_context.h” header
 * inclusion is optional (depends if user and/or warm start context
 * are being used
*/
#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
{
   /*
    * Other container technical data will be accessible through the pointer
    * defined here
    */
   struct #module_impl_name#__platform_hook *platform_hook;

   /* When the optional user context is used, 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;
   /* When the optional warm start context is used, the type
    * #module_impl_name#_warm_start_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#_warm_start_context warm_start;

} #module_impl_name#__context;

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

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

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

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

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

void #module_impl_name#_container__raise_fatal_error
   (#module_impl_name#__context* context,
    const ECOA__log log);

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

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

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

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

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

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

/* 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 */
#properties_call_specifications#

/* Recovery action service API call specification if this is a */
/* Fault Handler module */
#recovery_action_call_specification#

/* Persistent Information management operations */
#PINFO_read_call_specifications#
#PINFO_seek_call_specifications#

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

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

#endif  /* #MODULE_IMPL_NAME#_CONTAINER_H */
```

6.3 Container Types Template

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

6.4 User Module Context Template

```
/* @file #module_impl_name#_user_context.h
 * This is an example of a user defined User Module context
 */
#if !defined(#MODULE_IMPL_NAME#_USER_CONTEXT_H)
#define #MODULE_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 “#module_impl_name#_container_types.h”
/* User Module Context structure example */
typedef struct
{
   /* declare the User Module Context “local” data here */

} #module_impl_name#_user_context;

/* Warm Start Module Context structure example */
typedef struct
{
   /* declare the Warm Start Module Context data here */

} #module_impl_name#_warm_start_context;

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

#endif  /* #MODULE_IMPL_NAME#_USER_CONTEXT_H */
```

6.5 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 2 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 Module Context

In the C language, the Module Context is a structure which holds both the user local data (called “User Module Context” and “Warm Start Context”) and infrastructure-level technical data (which is implementation dependant). User context and warm start context features may be optionally selected in Module Type declarations using metamodel attributes. The presence or absence of declarations of corresponding fields in Module code must be in accordance with selections made in the Module Type declaration. 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 Module Context:

```
/* @file “#module_impl_name#_container.h”
 * Container Interface header for Module #module_impl_name#
 * Generated automatically from specification; do not modify here
 */

/* Container Types */
#include “#module_impl_name#_container_types.h”

/* Optional User Context: the “#module_impl_name#_user_context.h” header
inclusion is optional (depends if user and/or warm start context are being used
*/
#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
{
   /*
    * Other container technical data will accessible through the pointer */
    * defined here
    */
   struct #module_impl_name#__platform_hook *platform_hook;

   /* When the optional user context is used, 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 and the attribute
    * #module_impl_name#_user_context user shall be declared as follows:
    */
   #module_impl_name#_user_context user;

   /*
```

```
    * When the optional warm start context is used, the type
    * #module_impl_name#_warm_start_context shall be defined by the
    * user in the #module_impl_name#_user_context.h file to carry the module
    * implementation warm start private data and the attribute
    * #module_impl_name#_warm_start_context user shall be declared as follows:
    */
    #module_impl_name#_warm_start_context warm_start;

} #module_impl_name#__context;
```

8.1 User Module Context

The following shows the C syntax for the optional Module User Context (including an example data item; myCounter) and the Module Warm Start Context (including an example data item myData and validity flag warm_start_valid). The Module User Context header file is needed only if the user context and/or warm start context are used:

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

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

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

} #module_impl_name#_warm_start_context;
```

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

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++;
}
```

The optional 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 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 Module User Context and Warm Start Context may be declared in the bin-desc file related to the Component.

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

[1] The current ECOA **Error! Reference source not found.** 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 basic types and derived types are represented in C

9.1 Filenames and Namespace

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

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

```
/*
 * @file #namespace1#[__#namespacen#].h
 * Data-type declaration file
 * Generated automatically from specification; do not modify here
 */

typedef #basic_type_name#
  #namespace1#[__#namespacen#]__#simple_type_name#;
```

9.2 Basic Types

The basic 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 basic Boolean type, that are shown in Table 4.

Table 3 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 3 are fully defined using the predefined constants shown in Table 4:

Table 4 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 namespace.

9.3 Derived Types

9.3.1 Simple Types

The syntax for defining a Simple Type #simple_type_name# refined from a Basic Type #basic_type_name# in C is defined below. 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_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 syntax 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 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, 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).

9.3.4 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 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#;
```

9.3.5 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 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#;
```

9.3.6 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#;
```

9.4 Predefined Types

9.4.1 ECOA: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_INVALID_HANDLE            (1)
#define ECOA__return_status_DATA_NOT_INITIALIZED      (2)
#define ECOA__return_status_NO_DATA                   (3)
#define ECOA__return_status_INVALID_IDENTIFIER        (4)
#define ECOA__return_status_NO_RESPONSE               (5)
#define ECOA__return_status_OPERATION_ALREADY_PENDING (6)
#define ECOA__return_status_CLOCK_UNSYNCHRONIZED      (7)
#define ECOA__return_status_RESOURCE_NOT_AVAILABLE    (8)
#define ECOA__return_status_OPERATION_NOT_AVAILABLE   (9)
```

```
#define ECOA__return_status_INVALID_PARAMETER         (10)
```

9.4.2 ECOA:hr_time

The binding for time is:

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

9.4.3 ECOA:global_time

Global time is represented as:

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

9.4.4 ECOA:duration

Duration is represented as:

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

9.4.5 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;
```

9.4.6 ECOA:error_id

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

```
typedef ECOA__uint32 ECOA__error_id;
```

9.4.7 ECOA:error_code

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

```
typedef ECOA__uint32 ECOA__error_code;
```

9.4.8 ECOA:asset_id

In C the syntax for a `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>

#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__PD_#protection_domain_name1# (#PD_ID1#)
#define ECOA_Assets__PD_#protection_domain_name2# (#PD_ID2#)
#define ECOA_Assets__PD_#protection_domain_nameN# (#PD_IDN#)

#define ECOA_Assets__NOD_#computing_node_name1# (#NOD_ID1#)
#define ECOA_Assets__NOD_#computing_node_name2# (#NOD_ID2#)
#define ECOA_Assets__NOD_#computing_node_nameN# (#NOD_IDN#)

#define ECOA_Assets__PF_#computing_platform_name1# (#PF_ID1#)
#define ECOA_Assets__PF_#computing_platform_name2# (#PF_ID2#)
#define ECOA_Assets__PF_#computing_platform_nameN# (#PF_IDN#)

#define ECOA_Assets__SOP_#service_operation_name1# (#ELI_UID#)
#define ECOA_Assets__SOP_#service_operation_name2# (#ELI_UID#)
```

```
#define ECOA_Assets__SOP_#service_operation_nameN# (#ELI_UID#)

#define ECOA_Assets__#deployment_name1# (#DEP_ID1#)
#define ECOA_Assets__#deployment_name2# (#DEP_ID2#)
#define ECOA_Assets__#deployment_nameN# (#DEP_IDN#)

#endif
```

9.4.9 ECOA:asset_type

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

```
typedef ECOA__uint32 ECOA__asset_type;
#define ECOA__asset_type_COMPONENT         (0)
#define ECOA__asset_type_PROTECTION_DOMAIN (1)
#define ECOA__asset_type_NODE              (2)
#define ECOA__asset_type_PLATFORM          (3)
#define ECOA__asset_type_SERVICE           (4)
#define ECOA__asset_type_DEPLOYMENT        (5)
```

9.4.10 ECOA:error_type

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

```
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)
#define ECOA__error_type_UNKNOWN_OPERATION      (19)
```

```
#define ECOA__error_type_OPERATION_OVERRATED    (20)
#define ECOA__error_type_OPERATION_UNDERRATED   (21)
```

9.4.11 ECOA:recovery_action_type

In C `ECOA:recovery_action_type` translates to `ECOA__recovery_action_type`, with the enumerated values shown below:

```
typedef ECOA__uint32 ECOA__recovery_action_type;
#define ECOA__recovery_action_type_SHUTDOWN          (0)
#define ECOA__recovery_action_type_COLD_RESTART      (1)
#define ECOA__recovery_action_type_WARM_RESTART      (2)
#define ECOA__recovery_action_type_CHANGE_DEPLOYMENT (3)
```

9.4.12 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;
```

9.4.13 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)
```

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

The following is the C syntax for invoking a request received by a module instance when a 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
 * Module Interface header for Module #module_impl_name#
 * Generated automatically from specification; do not modify here
 */

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

10.1.1.2 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
 * Module Interface header for Module #module_impl_name#
 * Generated automatically from specification; do not modify here
 */

void #module_impl_name#__#operation_name#__response_received
   (#module_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 module.

10.1.2 Versioned Data 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.

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

void #module_impl_name#__#operation_name#__updated
```

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

10.1.3 Event Received

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

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

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

10.2 Module Lifecycle

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

10.2.1 Initialize_Received

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

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

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

10.2.2 Start_Received

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

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

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

10.2.3 Stop_Received

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

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

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

10.2.4 Shutdown_Received

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

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

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

10.3 Error_notification at Fault Handler level

The C syntax for the container to report an error to a Fault Handler is:

```c
/*
 * @file #fault_handler_module_impl_name#.h
 * Module Interface header for the Fault Handler Module
 * #fault_handler_module_impl_name#
 * Generated automatically from specification; do not modify here
 */

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);
```

11 Container Interface

11.1 Operations

11.1.1 Request Response

11.1.1.1 Response Send

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

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

ECOA__return_status
  #module_impl_name#_container__#operation_name#__response_send
   (#module_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.

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
 * Container Interface header for Module #module_impl_name#
 * Generated automatically from specification; do not modify here
 */

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

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
 * Container Interface header for Module #module_impl_name#
 * Generated automatically from specification; do not modify here
 */

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

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)

- Note: the definition of versioned data handles involved in all #operation_name# is done in the Container Types header file, as specified in Section 12.1.1.

11.1.2.1 Get Read Access

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

#include "#module_impl_name#_container_types.h"

ECOA__return_status
  #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

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

#include “#module_impl_name#_container_types.h”

ECOA__return_status
  #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
 * Container Interface header for Module #module_impl_name#
 * Generated automatically from specification; do not modify here
 */

#include “#module_impl_name#_container_types.h”

ECOA__return_status
  #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

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

#include “#module_impl_name#_container_types.h”

ECOA__return_status
  #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

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

#include “#module_impl_name#_container_types.h”

ECOA__return_status
   #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
 * Container Interface header for Module #module_impl_name#
 * Generated automatically from specification; do not modify here
 */

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

11.2 Properties

This section describes the syntax for the Get_Value operation to request the module 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 #module_impl_name#_container.h
 * Container Interface header for Module #module_impl_name#
 * Generated automatically from specification; do not modify here
 */

void #module_impl_name#_container__get_#property_name#_value
   (#module_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

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

11.3.1 Log_Trace

```
/* @file “#module_impl_name#_container.h”
 * Container Interface header for Module #module_impl_name#
 * Generated automatically from specification; do not modify here
 */

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

11.3.2 Log_Debug

```
/* @file “#module_impl_name#_container.h”
 * Container Interface header for Module #module_impl_name#
 * Generated automatically from specification; do not modify here
 */

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

11.3.3 Log_Info

```
/* @file “#module_impl_name#_container.h”
 * Container Interface header for Module #module_impl_name#
 * Generated automatically from specification; do not modify here
 */

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

11.3.4 Log_Warning

```
/* @file “#module_impl_name#_container.h”
 * Container Interface header for Module #module_impl_name#
 * Generated automatically from specification; do not modify here
 */

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

11.3.5 Raise_Error

```
/* @file “#module_impl_name#_container.h”
 * Container Interface header for Module #module_impl_name#
 * Generated automatically from specification; do not modify here
 */

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

11.3.6 Raise_Fatal_Error

```
/* @file “#module_impl_name#_container.h”
 * Container Interface header for Module #module_impl_name#
 * Generated automatically from specification; do not modify here
 */

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

11.4 Time Services

11.4.1 Get_Relative_Local_Time

```
/* @file “#module_impl_name#_container.h”
 * Container Interface header for Module #module_impl_name#
 * Generated automatically from specification; do not modify here
 */

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

11.4.2 Get_UTC_Time

```
/* @file “#module_impl_name#_container.h”
 * Container Interface header for Module #module_impl_name#
 * Generated automatically from specification; do not modify here
 */

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

11.4.3 Get_Absolute_System_Time

```
/* @file “#module_impl_name#_container.h”
 * Container Interface header for Module #module_impl_name#
 * Generated automatically from specification; do not modify here
 */

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

11.4.4 Get_Relative_Local_Time_Resolution

```
/* @file “#module_impl_name#_container.h”
 * Container Interface header for Module #module_impl_name#
 * Generated automatically from specification; do not modify here
 */

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

11.4.5 Get_UTC_Time_Resolution

```
/* @file “#module_impl_name#_container.h”
 * Container Interface header for Module #module_impl_name#
 * Generated automatically from specification; do not modify here
 */

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

11.4.6 Get_Absolute_System_Time_Resolution

```
/* @file “#module_impl_name#_container.h”
 * Container Interface header for Module #module_impl_name#
 * Generated automatically from specification; do not modify here
 */

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

11.5 Persistent Information management (PINFO)

11.5.1 PINFO read

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

```
/* @file “#module_impl_name#_container.h”
 * Container Interface header for Module #module_impl_name#
 * Generated automatically from specification; do not modify here
 */

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

11.5.2 PINFO seek

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

```
/* @file “#module_impl_name#_container.h”
 * Container Interface header for Module #module_impl_name#
 * Generated automatically from specification; do not modify here
 */

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

11.5.3 Example of Defining Private PINFO

Not applicable to the C Binding.

11.5.4 Example of Defining Public PINFO

Not applicable to the C Binding.

11.6 Recovery Action

This section contains the C syntax for the recovery action service provided to Fault Handlers by the container.

```
/* @file “#fault_handler_impl_name#_container.h”
 * Container Interface header for Fault Handler Module
 * #fault_handler_impl_name#
 * Generated automatically from specification; do not modify here
 */

ECOA__return_status #fault_handler_impl_name#_container__recovery_action
   (#fault_handler_impl_name#__context* context,
    ECOA__recovery_action_type recovery_action,
    ECOA__asset_id asset_id,
    ECOA__asset_type asset_type);
```

11.7 Save Warm Start Context

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

```
/* @file “#module_impl_name#_container.h”
 * Container Interface header for Module
 * #module_impl_name#
 * Generated automatically from specification; do not modify here
 */

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

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 module.

12.1.1 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 #module_impl_name#_container_types.h
 * Container Types header for Module #module_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];
} #module_impl_name#_container__#operation_name#_handle;
```

13 External Interface

This section contains the C syntax for the ECOA external interface provided to non-ECOA software by the container.

Note: the choice of the language for generating external APIs is made separately from the choice of the language for generating ECOA modules 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
 */
void #component_impl_name#__#external_operation_name#
   (const #event_parameters#);
```

14 Default Values

Not applicable to the C Binding.

15 Trigger Instances

Not applicable to the C Binding.

16 Dynamic Trigger Instances

Not applicable to the C Binding.

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(ECOA_64BIT_SUPPORT)

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

/* ECOA:uint64 */
typedef unsigned long long int ECOA__uint64;
#define ECOA__UINT64_MIN (0LLU)
#define ECOA__UINT64_MAX (18446744073709551615LLU)

#endif /* ECOA_64BIT_SUPPORT */


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

/* 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:error_id */
typedef ECOA__uint32 ECOA__error_id;

/* ECOA:asset_id */
typedef ECOA__uint32 ECOA__asset_id;

/* ECOA:asset_type */
typedef ECOA__uint32 ECOA__asset_type;
#define ECOA__asset_type_COMPONENT         (0)
#define ECOA__asset_type_PROTECTION_DOMAIN (1)
#define ECOA__asset_type_NODE              (2)
#define ECOA__asset_type_PLATFORM          (3)
#define ECOA__asset_type_SERVICE           (4)
#define ECOA__asset_type_DEPLOYMENT        (5)

/* 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)
#define ECOA__error_type_UNKNOWN_OPERATION      (19)
#define ECOA__error_type_OPERATION_OVERRATED    (20)
#define ECOA__error_type_OPERATION_UNDERRATED   (21)

/* ECOA:recovery_action_type */
typedef ECOA__uint32 ECOA__recovery_action_type;
#define ECOA__recovery_action_type_SHUTDOWN          (0)
#define ECOA__recovery_action_type_COLD_RESTART      (1)
#define ECOA__recovery_action_type_WARM_RESTART      (2)
#define ECOA__recovery_action_type_CHANGE_DEPLOYMENT (3)

#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)

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

#endif /* ECOA_H */
```