European Component Oriented Architecture (ECOA®) Collaboration Programme: Architecture Specification Part 9: C++ Language Binding

BAE Ref No: IAWG-ECOA-TR-005
Dassault Ref No: DGT 144478-F

Issue: 6

Prepared by
BAE Systems (Operations) Limited and Dassault Aviation

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.*

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.

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 9 of the Architecture Specification, and describes the C++ (ref ISO/IEC 14882:2003(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

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

4 Definitions

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

5 Abbreviations

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

6 Module to Language Mapping

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

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

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

The first class is the Module class:

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

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

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

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

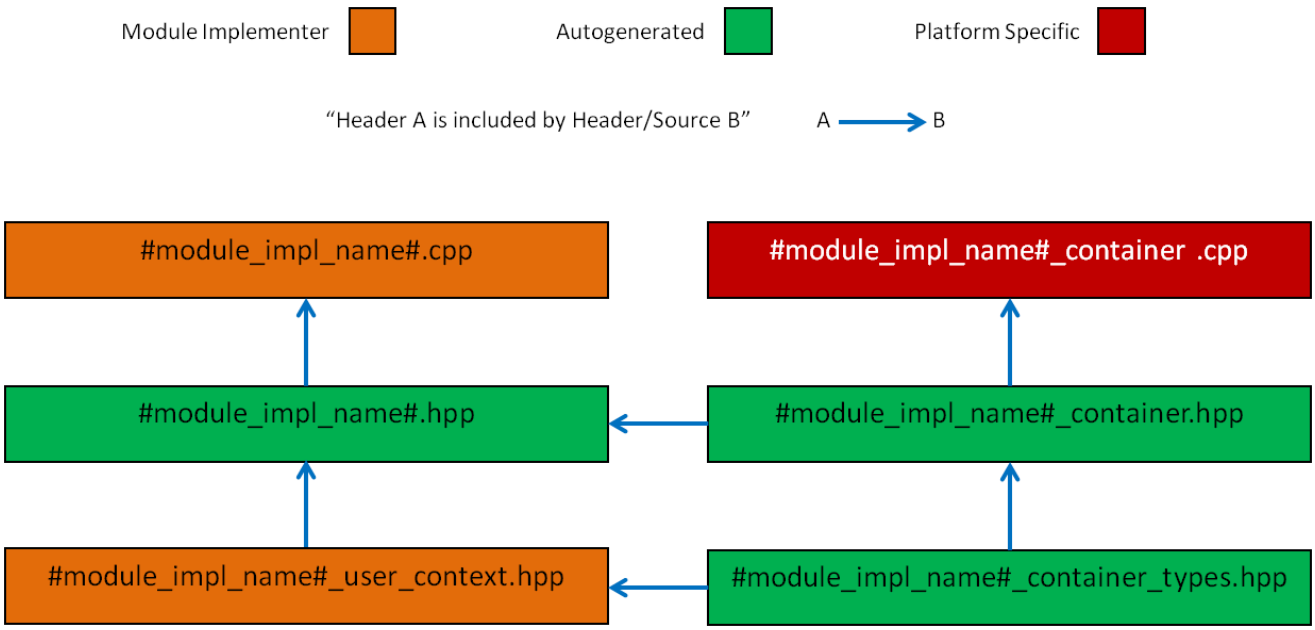

Figure 1 C++ Files Organization

Table 1 Filename Mapping

Filename	Use
#module_impl_name#.hpp	**Module Interface**: This header (.hpp) contains the declaration of the entry points provided by the module and callable by the container.
#module_impl_name#.cpp	**Module Implementation:** This source (.cpp) implements the Module Interface.
#module_impl_name#_container.hpp	**Container Interface**: This header (.hpp) contains the declaration of the functions provided by the container and callable by the module.
#module_impl_name#_container_types.hpp	**Container Types**: This header (.hpp) contains the declaration of container-level data types usable by the module. These types are related to the Container for a Module Implementation.
#module_impl_name#_container.cpp	**Container Implementation**: This source (.cpp) implements the Container Interface. It is out of scope of this document, as it is platform dependent. The Container may actually be a collection of source files depending upon the platform implementation

Filename	Use
#module_impl_name#_user_context.hpp	**Module User Context** User extensions to Module Context. These types are related to the Module Implementation.

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

Templates for the files in Table 1 are provided below:

6.1 Module Interface Template

The following is a minimal Module Interface example:

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

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

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

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

namespace #module_impl_name#
{

class Module
{
   public:
```

```
      void INITIALIZE__received();

      void START__received();

      void STOP__received();

      void SHUTDOWN__received();

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

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

      // Optional Warm Start data for this module implementation may be declared
      // here as a single attribute named 'warm_start' which may be of a user
      // defined type.
      warm_start_context warm_start;

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

      // The following describes the API generated:
      // * For any Event: event_received operations
      // * For any Request-Response: request_received operations
      // * For any Asynchronous Request-Response: response_received operation
      // * For any Notifying Versioned Data Read: updated operation

      // * Fault handler API:
      // * * error_notification (if this is a fault handler module)

}; /* Module */

extern "C" {

   Module* #module_impl_name#__new_instance();

}

} /* #module_impl_name# */
```

```
#endif  /* #MODULE_IMPL_NAME#_HPP */
```

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

The following is an outline of a Module Implementation:

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

namespace #module_impl_name#
{

extern "C" {

   Module* #module_impl_name#__new_instance()
   {
       return new Module();
   }
}

void Module::#operation_name#__received()
{
   /* To be implemented by the module */

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

} /* #module_impl_name# */
```

6.2 Container Interface Template

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

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

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

namespace #module_impl_name#
{

class Container
{
   public:

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

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

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

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

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

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

```
/* Time services API */
void get_relative_local_time
   (ECOA::hr_time &relative_local_time);

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

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

/* Time resolution services API */
void get_relative_local_time_resolution
   (ECOA::duration &relative_local_time_resolution);

void get_UTC_time_resolution
   (ECOA::duration &utc_time_resolution);

void get_absolute_system_time_resolution
   (ECOA::duration &absolute_system_time_resolution);

/* Optional API for saving the warm start context */
void save_warm_start_context();

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

// The following describes the APIs generated:
// * For any Event: send
// * For any Get_Properties: get_#property_name#_value
// * For any Synchronous Request-Response: request_sync operation
// * For any Asynchronous Request-Response: request_async operation
// * For any Request-Response: response operation
// * For any Versioned Data Read Access: get_read_access,
//   release_read_access
// * For any Versioned Data Write Access: get_write_access,
//   cancel_write_access, publish_write_access

// PINFO APIs
// * For any PINFO Read Access: read, seek

// If this is a Fault Handler module then an additional API is declared:
// * Recovery Action API:
ECOA::return_status recovery_action
```

```
          (ECOA::recovery_action_type recovery_action,
           ECOA::asset_id asset_id,
           ECOA::asset_type asset_type);

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

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

   private:

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

}; /* Container */

} /* #module_impl_name# */

#endif  /* #MODULE_IMPL_NAME#_CONTAINER_HPP */
```

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

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

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

6.3 Container Types Template

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

The specification for these data types is provided in Section 12.

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

#if !defined(#MODULE_IMPL_NAME#_CONTAINER_TYPES_HPP)
```

```
#define #MODULE_IMPL_NAME#_CONTAINER_TYPES_HPP

#include <ECOA.hpp>

namespace #module_impl_name# {

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

} /* #module_impl_name# */

#endif  /* #MODULE_IMPL_NAME#_CONTAINER_TYPES_HPP */
```

6.4 User Module Context Template

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

The Module User Context header file is needed only if the user context and/or warm start context are used.

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

The following shows the C++ syntax for defining the 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.

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

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

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

namespace #module_impl_name#
```

```
{

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

} user_context;

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

} /* #module_impl_name# */

#endif /* #MODULE_IMPL_NAME#_USER_CONTEXT_HPP */
```

6.5 Guards

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

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

/* all the declarations shall come here */

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

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

7 Parameters

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

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

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

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

8 Module Context

8.1 User Module Context

In C++, the Module Context is a structure which holds the user local data (called “User Module Context” and “Warm Start Context”). 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. These structures may be declared in the Module Interface as member variables within the Module Implementation class (one public variable for the user context and one public variable for the warm start context).

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

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

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

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

/* The “#module_impl_name#_user_context.hpp” header inclusion is optional (depends
if user and/or warm start context are being used
 */
#include “#module_impl_name#_user_context.hpp”

#include “#module_impl_name#_container.hpp”

namespace #module_impl_name#
{

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

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

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

```
      // warm_start shall be
      // defined by the user in the #module_impl_name#_user_context.hpp file
      // to carry the module non-volatile data
      warm_start_context warm_start;

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

}; /* Module */

extern "C" {

   Module* #module_impl_name#__new_instance();

}

} /* #module_impl_name# */
```

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

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#].hpp`

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

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

namespace #namespace1# {
//…
   namespace #namespacen# {

      typedef #basic_type_name# #simple_type_name#;

   } /* #namespacen#  */
//…
} /* #namespace1#  */
```

9.2 Basic Types

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

Table 2 C++ Basic Type Mapping

ECOA Basic Type	C++ type
ECOA:boolean8	ECOA::boolean8
ECOA:int8	ECOA::int8
ECOA:char8	ECOA::char8
ECOA:int16	ECOA::int16
ECOA:int32	ECOA::int32
ECOA:int64	ECOA::int64
ECOA:uint8	ECOA::uint8
ECOA:byte	ECOA::byte
ECOA:uint16	ECOA::uint16

ECOA Basic Type	C++ type
ECOA:uint32	ECOA::uint32
ECOA:uint64	ECOA::uint64
ECOA:float32	ECOA::float32
ECOA:double64	ECOA::double64

The data-types in Table 2 are fully defined using the predefined constants shown in Table 3:

Table 3 C++ Predefined Constants

C++ Basic Type	C++ constant
ECOA::boolean8	ECOA::TRUE ECOA::FALSE
ECOA::int8	ECOA::INT8_MIN ECOA::INT8_MAX
ECOA::char8	ECOA::CHAR8_MIN ECOA::CHAR8_MAX
ECOA::byte	ECOA::BYTE_MIN ECOA::BYTE_MAX
ECOA::int16	ECOA::INT16_MIN ECOA::INT16_MAX
ECOA::int32	ECOA::INT32_MIN ECOA::INT32_MAX
ECOA::int64	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.

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

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

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

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

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

`minRange` or `maxRange` can also reference existing constants (see Section 9.3.2) as follows:

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

9.3.2 Constants

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

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

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

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

9.3.3 Enumerations

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

```
struct #enum_type_name#
{
   #basic_type_name# value;
   enum  EnumValues {
      #enum_value_name_1# = #enum_value_value_1#,
      #enum_value_name_2# = #enum_value_value_2#,
```

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

Where:

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

9.3.4 Records

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

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

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

9.3.5 Variant Records

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

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

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

```
typedef struct{

   #selector_type_name# #selector_name#;

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

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

} # variant_record_type_name#;
```

9.3.6 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.

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

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

9.3.7 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.

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

9.4 Predefined Types

9.4.1 ECOA:return_status

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

```
namespace ECOA {

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

} /* ECOA */
```

9.4.2 ECOA:hr_time

The binding for hr_time is:

```
namespace ECOA {

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

```
} /* ECOA */
```

9.4.3 ECOA:global_time

Global time is represented as:

```
namespace ECOA {

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

} /* ECOA */
```

9.4.4 ECOA:duration

Duration is represented as:

```
namespace ECOA {

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

} /* ECOA */
```

9.4.5 ECOA:log

The syntax for a log in C++ is:

```
namespace ECOA {

   const ECOA::uint32 LOG_MAXSIZE = 256;

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

} /* ECOA */
```

9.4.6 ECOA:error_id

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

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

9.4.7 ECOA:error_code

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

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

9.4.8 ECOA:asset_id

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

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

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

```
/* File ECOA_Assets.hpp */

#include <ECOA.hpp>

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

      static const ECOA::asset_id CMP_#component_instance_name1# = #CMP_ID1#;
      static const ECOA::asset_id CMP_#component_instance_name2# = #CMP_ID2#;
      static const ECOA::asset_id CMP_#component_instance_nameN# = #CMP_IDN#;

      static const ECOA::asset_id PD_#protection_domain_name1# = #PD_ID1#;
      static const ECOA::asset_id PD_#protection_domain_name2# = #PD_ID2#;
      static const ECOA::asset_id PD_#protection_domain_nameN# = #PD_IDN#;

      static const ECOA::asset_id NOD_#computing_node_name1# = #NOD_ID1#;
      static const ECOA::asset_id NOD_#computing_node_name2# = #NOD_ID2#;
      static const ECOA::asset_id NOD_#computing_node_nameN# = #NOD_IDN#;

      static const ECOA::asset_id PF_#computing_platform_name1# = #PF_ID1#;
      static const ECOA::asset_id PF_#computing_platform_name2# = #PF_ID2#;
      static const ECOA::asset_id PF_#computing_platform_nameN# = #PF_IDN#;

      static const ECOA::asset_id SOP_#service_operation_name1# = #ELI_UID#;
```

```
        static const ECOA::asset_id SOP_#service_operation_name2# = #ELI_UID#;
        static const ECOA::asset_id SOP_#service_operation_nameN# = #ELI_UID#;

        static const ECOA::asset_id DEP_#deployment_name1# = #DEP_ID1#;
        static const ECOA::asset_id DEP_#deployment_name2# = #DEP_ID2#;
        static const ECOA::asset_id DEP_#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:

```
struct asset_type
{
   ECOA::uint32 value;
   enum EnumValues
   {
      COMPONENT         = 0,
      PROTECTION_DOMAIN = 1,
      NODE              = 2,
      PLATFORM          = 3,
      SERVICE           = 4,
      DEPLOYMENT        = 5
   };
   inline void operator = (ECOA::uint32 i) { value = i; }
   inline operator ECOA::uint32() const { return value; }
   inline asset_type (EnumValues v):value(v) {}
   inline asset_type ():value(COMPONENT) {}
};
```

9.4.10 ECOA:error_type

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

```
struct error_type
{
   ECOA::uint32 value;
   enum EnumValues
   {
      RESOURCE_NOT_AVAILABLE = 0,
      UNAVAILABLE            = 1,
      MEMORY_VIOLATION       = 2,
      NUMERICAL_ERROR        = 3,
      ILLEGAL_INSTRUCTION    = 4,
```

```
      STACK_OVERFLOW          = 5,
      DEADLINE_VIOLATION      = 6,
      OVERFLOW                = 7,
      UNDERFLOW               = 8,
      ILLEGAL_INPUT_ARGS      = 9,
      ILLEGAL_OUTPUT_ARGS     = 10,
      ERROR                   = 11,
      FATAL_ERROR             = 12,
      HARDWARE_FAULT          = 13,
      POWER_FAIL              = 14,
      COMMUNICATION_ERROR     = 15,
      INVALID_CONFIG          = 16,
      INITIALISATION_PROBLEM  = 17,
      CLOCK_UNSYNCHRONIZED    = 18,
      UNKNOWN_OPERATION       = 19,
      OPERATION_OVERRATED     = 20,
      OPERATION_UNDERRATED    = 21
   };
   inline void operator = (ECOA::uint32 i) { value = i; }
   inline operator ECOA::uint32() const { return value; }
   inline error_type (EnumValues v):value(v) {}
   inline error_type ():value(RESOURCE_NOT_AVAILABLE) {}
};
```

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:

```
struct recovery_action_type
{
   ECOA::uint32 value;
   enum EnumValues
   {
      SHUTDOWN          = 0,
      COLD_RESTART      = 1,
      WARM_RESTART      = 2,
      CHANGE_DEPLOYMENT = 3
   };
   inline void operator = (ECOA::uint32 i) { value = i; }
   inline operator ECOA::uint32() const { return value; }
   inline recovery_action_type (EnumValues v):value(v) {}
   inline recovery_action_type ():value(SHUTDOWN) {}
};
```

9.4.12 ECOA:pinfo_filename

The syntax for a pinfo_filename in C++ is:

```
namespace ECOA {

   const ECOA::uint32 PINFO_FILENAME_MAXSIZE = 256;

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

} /* ECOA */
```

9.4.13 ECOA:seek_whence_type

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

```
struct seek_whence_type
{
   ECOA::uint32 value;
   enum EnumValues
   {
      ECOA_SEEK_SET = 0,
      ECOA_SEEK_CUR = 1,
      ECOA_SEEK_END = 2
   };
   inline void operator = (ECOA::uint32 i) { value = i; }
   inline operator ECOA::uint32() const { return value; }
   inline seek_whence_type (EnumValues v):value(v) {}
   inline seek_whence_type ():value(ECOA_SEEK_SET) {}
};
```

10 Module Interface

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 Operations

10.1.1 Request-Response

10.1.1.1 Request Received

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

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

namespace #module_impl_name#
{

class Module
{
   public:

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

}; /* Module */

} /* #module_impl_name# */
```

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. (The reply to a synchronous request response is the provided by return of the original request).

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

namespace #module_impl_name#
{

class Module
```

```
{
   public:

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

}; /* Module */

} /* #module_impl_name# */
```

The “#response_parameters#” are the “out” parameters of the request-response operation, but are treated as inputs to the function and passed as “const” parameters, so they are not modified by the 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#.hpp
 * Module Interface class header for Module #module_impl_name#
 * Generated automatically from specification; do not modify here
 */

namespace #module_impl_name#
{

class Module
{
   public:

      void #operation_name#__updated();

}; /* Module */

} /* #module_impl_name# */
```

10.1.3 Event Received

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

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

```
 */

namespace #module_impl_name#
{

class Module
{
   public:

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

}; /* Module */

} /* #module_impl_name# */
```

10.2 Module Lifecycle

The methods that are used to command a module/trigger/dynamic trigger instance to change (lifecycle) state are defined as follows in C++:

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

namespace #module_impl_name#
{

class Module
{
   public:

      void INITIALIZE__received() ;

      void START__received();

      void STOP__received();

      void SHUTDOWN__received();

}; /* Module */

} /* #module_impl_name# */
```

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

10.3 Error_notification at Fault Handler level

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

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

namespace fault_handler_impl_name#
{

class Module
{
   public:

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

}; /* Module */

} /* #fault_handler_impl_name# */
```

11 Container Interface

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

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.hpp
 * Container Interface class header for Module #module_impl_name#
```

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

namespace #module_impl_name#
{

class Container
{
   public:

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

}; /* Container */

} /* #module_impl_name# */
```

The "#response_parameters#" are the "out" parameters of the request-response operation, but are treated as inputs to the function and passed as "const" parameters, so they are not modified by the 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.hpp
 * Container Interface class header for Module #module_impl_name#
 * Generated automatically from specification; do not modify here
 */

namespace #module_impl_name#
{

class Container
{
   public:

      ECOA::return_status #operation_name#__request_sync
         (const #request_parameters#,
          #response_parameters#);

}; /* Container */
```

```
} /* #module_impl_name# */
```

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

#include "#module_impl_name#_container_types.hpp"

namespace #module_impl_name#
{

class Container
{
   public:

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

}; /* Container */

} /* #module_impl_name# */
```

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.hpp
 * Container Interface class header for Module #module_impl_name#
```

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

#include "#module_impl_name#_container_types.hpp"

namespace #module_impl_name#
{

class Container
{
   public:

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

} /* #module_impl_name# */
```

11.1.2.2 Release Read Access

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

#include "#module_impl_name#_container_types.hpp"

namespace #module_impl_name#
{

class Container
{
   public:

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

}; /* Container */

} /* #module_impl_name# */
```

11.1.2.3 Get Write Access

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

#include "#module_impl_name#_container_types.hpp"

namespace #module_impl_name#
{

class Container
{
   public:

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

}; /* Container */

} /* #module_impl_name# */
```

11.1.2.4 Cancel Write Access

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

#include "#module_impl_name#_container_types.hpp"

namespace #module_impl_name#
{

class Container
{
   public:

      ECOA::return_status #operation_name#__cancel_write_access
         (#operation_name#_handle& data_handle);
```

```
}; /* Container */

} /* #module_impl_name# */
```

11.1.2.5 Publish Write Access

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

#include "#module_impl_name#_container_types.hpp"

namespace #module_impl_name#
{

class Container
{
   public:

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

}; /* Container */

} /* #module_impl_name# */
```

11.1.3 Event Send

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

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

namespace #module_impl_name#
{

class Container
{
   public:
```

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

}; /* Container */

} /* #module_impl_name# */
```

11.2 Properties

11.2.1 Get Value

The syntax for Get_Value is shown below where:

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

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

namespace #module_impl_name#
{

class Container
{
   public:

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

}; /* Container */

} /* #module_impl_name# */
```

11.2.2 Expressing Property Values

Not applicable to the C++ Binding.

11.2.3 Example of Defining and Using Properties

Not applicable to the C++ Binding.

11.3 Logging and Fault Management

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

namespace #module_impl_name#
{

class Container
{
   public:

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

}; /* Container */

} /* #module_impl_name# */
```

11.3.2 Log_Debug

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

namespace #module_impl_name#
{

class Container
```

```
{
   public:

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

}; /* Container */

} /* #module_impl_name# */
```

11.3.3 Log_Info

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

namespace #module_impl_name#
{

class Container
{
   public:

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

}; /* Container */

} /* #module_impl_name# */
```

11.3.4 Log_Warning

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

namespace #module_impl_name#
{

class Container
```

```
{
   public:

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

}; /* Container */

} /* #module_impl_name# */
```

11.3.5 Raise_Error

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

namespace #module_impl_name#
{

class Container
{
   public:

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

}; /* Container */

} /* #module_impl_name# */
```

11.3.6 Raise_Fatal_Error

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

namespace #module_impl_name#
{
```

```
class Container
{
   public:

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

}; /* Container */

} /* #module_impl_name# */
```

11.4 Time Services

11.4.1 Get_Relative_Local_Time

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

namespace #module_impl_name#
{

class Container
{
   public:

      void get_relative_local_time
         (ECOA::hr_time& relative_local_time);

}; /* Container */

} /* #module_impl_name# */
```

11.4.2 Get_UTC_Time

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

```
namespace #module_impl_name#
{

class Container
{
   public:

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

}; /* Container */

} /* #module_impl_name# */
```

11.4.3 Get_Absolute_System_Time

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

namespace #module_impl_name#
{

class Container
{
   public:

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

}; /* Container */

} /* #module_impl_name# */
```

11.4.4 Get_Relative_Local_Time_Resolution

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

```
namespace #module_impl_name#
{

class Container
{
   public:

      void get_relative_local_time_resolution
         (ECOA::duration& relative_local_time_resolution);

}; /* Container */

} /* #module_impl_name# */
```

11.4.5 Get_UTC_Time_Resolution

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

namespace #module_impl_name#
{

class Container
{
   public:

      void get_UTC_time_resolution
         (ECOA::duration& utc_time_resolution);

}; /* Container */

} /* #module_impl_name# */
```

11.4.6 Get_Absolute_System_Time_Resolution

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

```
namespace #module_impl_name#
{

class Container
{
   public:

      void get_absolute_system_time_resolution
         (ECOA::duration& absolute_system_time_resolution);

}; /* Container */

} /* #module_impl_name# */
```

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

namespace #module_impl_name#
{

class Container
{
   public:

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

}; /* Container */

} /* #module_impl_name# */
```

11.5.2 PINFO seek

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

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

namespace #module_impl_name#
{

class Container
{
   public:

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

}; /* Container */

} /* #module_impl_name# */
```

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

The code below describes the C++ syntax for the recovery action operation provided by the container to a fault handler instance.

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

namespace #fault_handler_impl_name#
```

```
{

class Container
{
   public:

      // the following method shall be implemented by the Container to provide
      // the Fault Handler Implementation with a Recovery Action functionnality.
      ECOA::return_status recovery_action
         (ECOA::recovery_action_type recovery_action,
          ECOA::asset_id asset_id,
          ECOA::asset_type asset_type);

}; /* Container */

} /* #fault_handler_impl_name# */
```

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

namespace #module_impl_name#
{

class Container
{
   public:

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

}; /* Container */

} /* #module_impl_name# */
```

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.hpp
 * Container Types for Module #module_impl_name#
 * Generated automatically from specification; do not modify here
 */

#define ECOA_VERSIONED_DATA_HANDLE_PRIVATE_SIZE 32

namespace #module_impl_name# {

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

} /* #module_impl_name# */
```

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.hpp
 * External Interface header for Component Implementation
 * #component_impl_name#
```

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

namespace #component_impl_name#_External_Interface
{
   public:
      void #external_operation_name#(const #event_parameters#);

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

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.hpp
 */

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

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


#if !defined(ECOA_HPP)
#define ECOA_HPP

namespace ECOA {

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

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

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

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

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

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

    /* ECOA:uint8 */
   typedef unsigned char uint8;
   static const uint8 UINT8_MIN = 0;
   static const uint8 UINT8_MAX = 255;

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

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

#if defined (ECOA_64BIT_SUPPORT)
   /* ECOA:int64 */
   typedef long long int int64;
```

```
   static const int64 INT64_MIN = -9223372036854775807LL;
   static const int64 INT64_MAX = 9223372036854775807LL;

   /* ECOA:uint64 */
   typedef unsigned long long int uint64;
   static const uint64 UINT64_MIN = 0LLU;
   static const uint64 UINT64_MAX = 18446744073709551615LLU;
#endif /* ECOA_64BIT_SUPPORT */

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

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

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

   /* ECOA:hr_time */
```

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

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

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

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

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

/* ECOA:asset_id */
typedef ECOA::uint32 asset_id;

/* ECOA:asset_type */
struct asset_type {
   ECOA::uint32 value;
   enum EnumValues
   {
      COMPONENT         = 0,
      PROTECTION_DOMAIN = 1,
      NODE              = 2,
      PLATFORM          = 3,
      SERVICE           = 4,
      DEPLOYMENT        = 5
   };
   inline void operator = (ECOA::uint32 i) { value = i; }
   inline operator ECOA::uint32() const { return value; }
```

```
      inline asset_type (EnumValues v):value(v) {}
      inline asset_type ():value(COMPONENT) {}
   };

   /* ECOA:error_type */
   struct error_type {
      uint32 value;
      enum EnumValues
      {
         RESOURCE_NOT_AVAILABLE = 0,
         UNAVAILABLE            = 1,
         MEMORY_VIOLATION       = 2,
         NUMERICAL_ERROR        = 3,
         ILLEGAL_INSTRUCTION    = 4,
         STACK_OVERFLOW         = 5,
         DEADLINE_VIOLATION     = 6,
         OVERFLOW               = 7,
         UNDERFLOW              = 8,
         ILLEGAL_INPUT_ARGS     = 9,
         ILLEGAL_OUTPUT_ARGS    = 10,
         ERROR                  = 11,
         FATAL_ERROR            = 12,
         HARDWARE_FAULT         = 13,
         POWER_FAIL             = 14,
         COMMUNICATION_ERROR    = 15,
         INVALID_CONFIG         = 16,
         INITIALISATION_PROBLEM = 17,
         CLOCK_UNSYNCHRONIZED   = 18,
         UNKNOWN_OPERATION      = 19,
         OPERATION_OVERRATED    = 20,
         OPERATION_UNDERRATED   = 21
      };
      inline void operator = (uint32 i) { value = i; }
      inline operator uint32() const { return value; }
      inline error_type (EnumValues v):value(v) {}
      inline error_type ():value(RESOURCE_NOT_AVAILABLE) {}
   };

   /* ECOA:recovery_action_type */
   struct recovery_action_type {
      ECOA::uint32 value;
      enum EnumValues
      {
         SHUTDOWN          = 0,
```

```
         COLD_RESTART      = 1,
         WARM_RESTART      = 2,
         CHANGE_DEPLOYMENT = 3
      };
      inline void operator = (ECOA::uint32 i) { value = i; }
      inline operator ECOA::uint32() const { return value; }
      inline recovery_action_type (EnumValues v):value(v) {}
      inline recovery_action_type ():value(SHUTDOWN) {}
   };

   const ECOA::uint32 PINFO_FILENAME_MAXSIZE = 256;

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

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

} /* ECOA */

#endif /* ECOA_HPP */
```