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-C

Issue: 3

Prepared by
BAE Systems (Operations) Limited and Dassault Aviation

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

Contents

Figures

Tables

0 Introduction

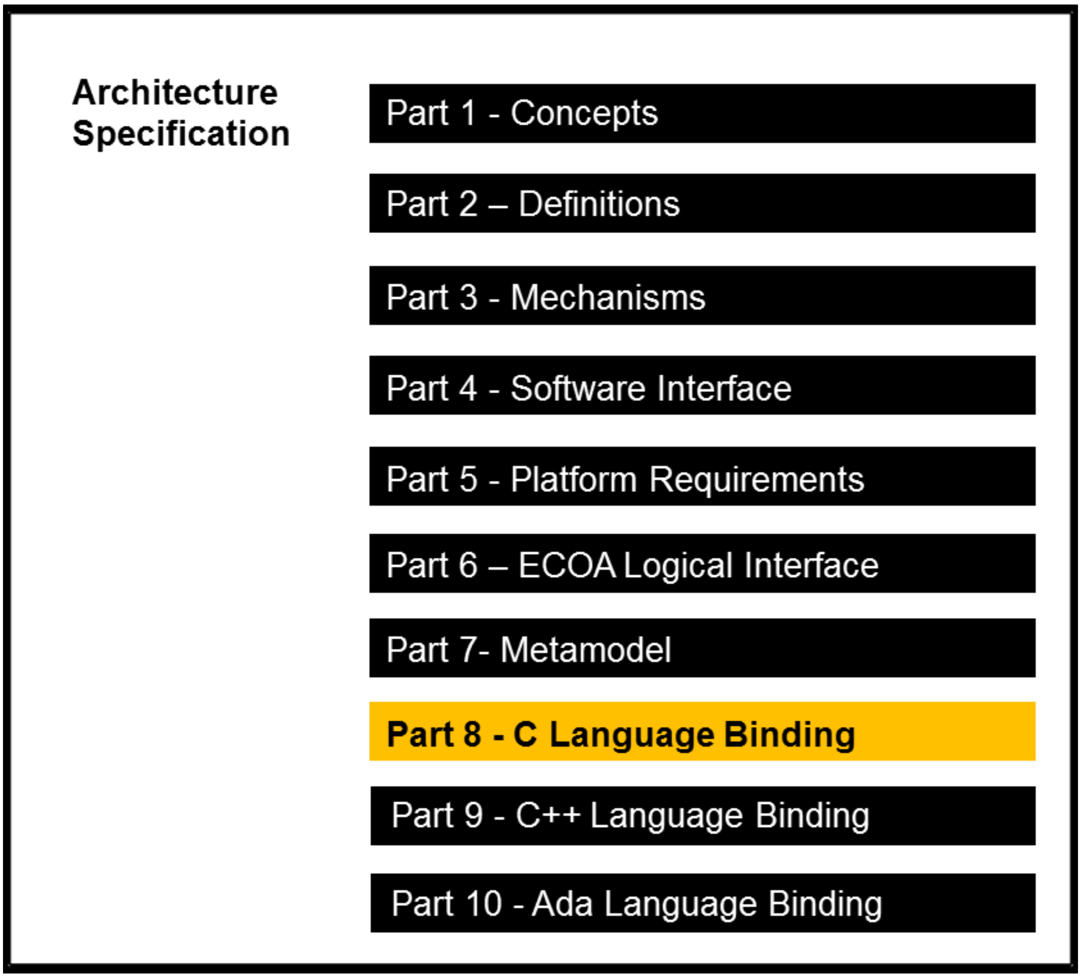

Figure 1 ECOA Documentation

This Architecture Specification provides the definitive specification for creating ECOA-based systems. It describes the standardised programming interfaces and data-model that allow a developer to construct an ECOA-based system. 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++ (C++ standard 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. The document is structured as follows:

- Section 6 describes the Module to Language Mapping;
- Section 7 describes the method of passing parameters;
- Section 8 describes the Module Context;
- Section 9 describes the pre-defined types that are provided and the types that can be derived from them;
- Section 10 describes the Module Interface;
- Section 11 describes the Container Interface;
- Section 12 describes the Fault Handler Interface;

- Section 13 provides a reference C++ header for the ECOA namespace, usable in any C++ binding implementation;

1 Scope

This purpose of this Architecture Specification is to establish 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 and contains mature high-level concepts, though low-level mechanisms and interfaces remain under development and are subject to change. This standard of documentation is recommended as appropriate for limited lab-based evaluation only. Product development based on this standard of documentation is not recommended.

3 Normative References

Ref	Description
Architecture Specification Part 1	IAWG-ECOA-TR-001 / DGT 144474 Issue 3 Architecture Specification Part 1 – Concepts
Architecture Specification Part 2	IAWG-ECOA-TR-012 / DGT 144487 Issue 3 Architecture Specification Part 2 – Definitions
Architecture Specification Part 3	IAWG-ECOA-TR-007 / DGT 144482 Issue 3 Architecture Specification Part 3 – Mechanisms
Architecture Specification Part 4	IAWG-ECOA-TR-010 / DGT 144485 Issue 3 Architecture Specification Part 4 – Software Interface
Architecture Specification Part 5	IAWG-ECOA-TR-008 / DGT 144483 Issue 3 Architecture Specification Part 5 – Platform Requirements
Architecture Specification Part 6	IAWG-ECOA-TR-006 / DGT 144481 Issue 3 Architecture Specification Part 6 – ECOA Logical Interface
Architecture Specification Part 7	IAWG-ECOA-TR-011 / DGT 144486 Issue 3

Architecture Specification Part 7 – Metamodel

Architecture Specification Part 8

IAWG-ECOA-TR-004 / DGT 144477

Issue 3

Architecture Specification Part 8 – C Language Binding

Architecture Specification Part 9

IAWG-ECOA-TR-005 / DGT 144478

Issue 3

Architecture Specification Part 9 – C++ Language Binding

Architecture Specification Part 10

IAWG-ECOA-TR-003 / DGT 144476

Issue 3

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++

4 Definitions

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

5 Abbreviations

API	Application Programming Interface
ARINC	Aeronautical Radio, Incorporated
ASAAC	Allied Standards Avionics Architecture Council
ASC	Application Software Component
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
DDS	Data Distribution Service
ECOA	European Component Oriented Architecture
ELI	ECOA Logical Interface
EUID	ECOA Unique Identifier (ID)
FIFO	First In, First Out
HR	High Resolution

ID	Identifier
IMA	Integrated Modular Avionics
IoC	Inversion-of-Control
IP	Internet Protocol
LRU	Line Replaceable Unit
NaN	Not a Number
OS	Operating System
PC	Personal Computer
POSIX	Portable Operating System Interface
QoS	Quality of Service
RFC	Request For Comments
RT	Real Time
RTOS	Real-Time Operating System
SOA	Service-oriented Architecture
SW	Software
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UML	Unified Modeling Language
UTC	Coordinated Universal Time
VME	Versa Module Europa (bus)
XML	eXtensible Markup Language
XSD	XML Schema Definition

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.

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

The first two of these classes are abstract classes:

- A pure virtual class corresponding to the Module Interface (called Module Interface class in the rest of the document), which defines all of the methods that the (user-provided) Module Implementation shall implement (see below). This class has no attributes and cannot be instantiated. A container will use this interface to interact with the module operations without depending on the underlying implementation.
- A pure virtual class (called the Container Interface class in the rest of the document), which corresponds to the Container Interface (i.e. the operations that the Container API for the Module). This class has no attribute and cannot be instantiated,

The third class is an implementation of the abstract Module Interface class, which the Module Implementer will create. This shall contain the user functional code to implement the required operations:

- A concrete class (called the Module Implementation in the rest of the document), derived from the Module Interface, which implements all of the methods that the module type is required to provide. The instance objects of this class, corresponding to each declared Module Instance, will be allocated by the container. All the user private data of the Module Instance must be declared as attributes (public, private or protected) of this class. The constructor of these calls will be used by the ECOA infrastructure to pass to the Module Implementation Instance object a pointer to its corresponding Container object.

In addition, a concrete implementation of the Container Interface class, containing the functional code to implement this interface is required. This would usually be generated by an ECOA platform provider/integrator and shall not be covered in this document.

Figure 2 – C++ Class Hierarchy shows the relationship between classes mentioned above, whilst

Table 1 shows the filename mappings.

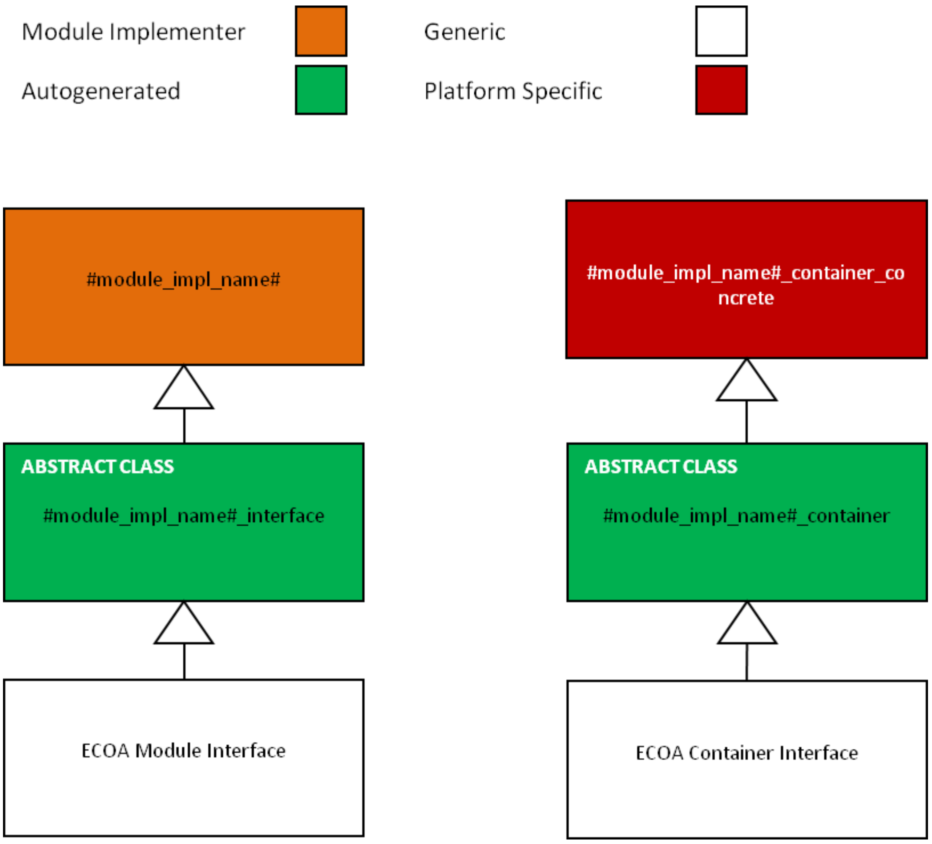

Figure 2 – C++ Class Hierarchy

Table 1 – Filename Mapping

• Filename	**Use**
#module_impl_name#_interface.hpp	Pure Virtual Module Interface class containing the declarations of the handlers entry points provided by the module and callable by the container
#module_impl_name#.hpp, **#module_impl_name#**.cpp	Module Implementation concrete class derived from module interface class
#module_impl_name#_container.hpp	Pure Virtual Container Interface class containing the declaration of functions provided by the container and callable by the module. The Module software shall only use this Container Interface to call the Container operations, without knowing the container concrete class which is platform dependant.

The ECOA infrastructure is responsible for allocating the appropriate Containers and Module objects; a pointer to the Container object shall be passed to its corresponding Module Implementation object as a parameter of the constructor of the Module Implementation object. The Module Implementation constructor shall have the signature specified below. The Module Implementation class shall contain a pointer to the Container object (#module_impl_name#_container*). This pointer to the Container shall remain valid while the Module Implementation object is active.

The Container shall automatically date operations on the emitter/requester side using an ECOA-provided structure called ECOA::timestamp. The Container also provides a utility method (called **`get_last_operation_timestamp`**) to retrieve this data when necessary.

Finally, Module Interface and Container Interface classes shall provide implementations for the pure virtual functions (from ECOA::Module_interface and ECOA::Container_interface respectivley) that they shall extend.

Templates for the files in Table 1 are provided below:

6.1 Module Interface Template

The following abstract class definition inherits from the ECOA:Module_Interface class (see section 6.4) and will define all operations available to be invoked on a module.

Note. In order to ensure binary compatability in C++, the order in which virtual methods are defined is of importanance. As such, the following order must be maintained.

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


class #module_impl_name#_interface : public virtual ECOA::Module_interface
{
    public:

       virtual void INITIALIZE__received() = 0;

       virtual void START__received() = 0;

       virtual void STOP__received() = 0;

       virtual void SHUTDOWN__received() = 0;

       virtual void REINITIALIZE__received() = 0;

       // All the operations for this Module implementation interface will be
       // declared as public pure virtual methods here in the order that the module
       // operations are defined in the XML

       // 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

       // If this is a Superervision module then additional APIs are declared in the
       // following order:

       // * Service Availability API:
       // * * service_availability_changed (if component has any required services)
       // * * service_provider_changed (if component has any required services)
```

```
    // * Supervision Module API for lifecycle operations (one set per non-supervision
    //   module instance, following the order that the module instances are defined
    //   in the XML, then trigger instance, then dynamic trigger instance)
    // * * lifecycle_notification__#module_instance_name#
    // * * error_notification__#module_instance_name#

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


}; /* #module_impl_name#_interface */
```

The following is a minimal Module Implementation class example (which inherits from the #module_impl_name#_interface.hpp):

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

extern "C" {

    #module_impl_name#_interface*
    #module_impl_name#__new_instance(#module_impl_name#_container* container);

}

class #module_impl_name# : public virtual #module_impl_name#_interface
{
    public:

       void INITIALIZE__received();

       void START__received();

       void STOP__received();

       void SHUTDOWN__received();

       void REINITIALIZE__received();

       // The constructor of the Component shall have the following
       // signature:
       #module_impl_name#(#module_impl_name#_container* container);

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


    private:
       // the Module Implementation shall hold a Container pointer
       // which is passed within the constructor
       #module_impl_name#_container* container;

       // user data for this module implementation must be declared here as
       // public, protected or private attributes
       int myUserCounter;

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

Note the inclusion of “extern C” at the beginning of the header file above. This 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
 * The following code illustrates an example of a constructor method
 * and a Received Event entry-point
 */

extern "C" {

    #module_impl_name#_interface*
    #module_impl_name#__new_instance(#module_impl_name#_container* container) {
        return new #module_impl_name#(container);
    }

}

#module_impl_name#::#module_impl_name#(#module_impl_name#_container* container)
{
   /* uses the logging functionality to trace */
   container->Log_Trace("Constructor entered.");
   /* initializes the container pointer */
   this->container = container;
   /* Initialises the other private attributes */
   myUserCounter = -1;
}

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

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

6.2 Container Class Template

The following abstract class definition inherits from the ECOA:Container_Interface class (see section 6.5) and will define all container operations which a module can invoke.

Note. In order to ensure binary compatability in C++, the order in which virtual methods are defined is of importanance. As such, the following order must be maintained.

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

```
class #module_impl_name#_container : public virtual ECOA::Container_interface
{
    public:

      /* get_last_operation_timestamp API */
      virtual void get_last_operation_timestamp(ECOA::timestamp &timestamp) = 0;

      /* Logging and fault management services API */
      virtual void log_trace
         (const ECOA::log &log) = 0;

      virtual void log_debug
         (const ECOA::log &log) = 0;

           virtual void log_info
         (const ECOA::log &log) = 0;

     virtual void log_warning
         (const ECOA::log &log) = 0;

     virtual void raise_error
         (const ECOA::log &log) = 0;

     virtual void raise_fatal_error
         (const ECOA::log &log) = 0;

     /* Time services API */
     virtual ECOA::return_status get_relative_local_time
         (ECOA::hr_time &relative_local_time) = 0;

           virtual ECOA::return_status get_UTC_time
         (ECOA::global_time &utc_time) = 0;

     virtual ECOA::return_status get_absolute_system_time
         (ECOA::global_time &absolute_system_time) = 0;

     /* Time resolution services API */
           virtual void get_relative_local_time_resolution
         (ECOA::duration &relative_local_time_resolution) = 0;

     virtual void get_UTC_time_resolution
         (ECOA::duration &utc_time_resolution) = 0;

           virtual void get_absolute_system_time_resolution
         (ECOA::duration &absolute_system_time_resolution) = 0;

       // All the operations for this Container interface will be declared as public
       // pure virtual 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: data_handle, get_read_access,
       //   release_read_access
       // * For any Versioned Data Write Access, data_handle, get_write_access,
       //   cancel_write_access, publish_write_access

       // If this is a Superervision module then additional APIs are declared in the
```

```
        // following order:

        // * Service Availability API:
        // * * get_service_availability (if component has any required services)
        // * * set_service_availability (if component has any provided services)

        // * Supervision Module API for lifecycle operations (one set per non-supervision
        //   module instance, following the order that the module instances are defined
        //   in the XML, then trigger instance, then dynamic trigger instance)
        // * * get_lifecycle_state_#module_instance_name#
        // * * STOP__#module_instance_name#
        // * * START__#module_instance_name#
        // * * INITIALIZE__#module_instance_name#
        // * * SHUTDOWN__#module_instance_name#

        // * Recovery Action API:
        // * * recovery_action (if the module is a fault handler)


}; /* #module_impl_name#_container */
```

In the rest of the document, the C++ bindings corresponding to the operations are presented as pure virtual functions i.e. as part of the Module Interface or the Container Interface.

6.3 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#_HH)
#define _#macro_protection_name#_HH

/* all the declarations shall come here */

#endif  /* _#macro_protection_name#_HH */
```

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

6.4 ECOA Module Interface Class

The following shows the outline for the Module Interface class, declared within the ECOA namespace:

Note. In order to ensure binary compatability in C++, the order in which virtual methods are defined is of importanance. As such, the following order must be maintained.

```
namespace ECOA {

class Module_interface
{
    public:

        // virtual destructor
        virtual ~Module_interface() {}

        virtual void INITIALIZE__received() = 0;

        virtual void START__received() = 0;
```

```
        virtual void STOP__received() = 0;

        virtual void SHUTDOWN__received() = 0;

        virtual void REINITIALIZE__received() = 0;

}; /* Module_interface */

} /* ECOA */
```

Note that the C++ fault_handler is not fully compatible with the expected binding of the ECOA fault_handler function: the above definition allows a generic container to be created without thorough understanding of the module contents.

6.5 ECOA Container Interface Class

The following shows the outline for the Container Interface class, declared within the ECOA namespace:

Note. In order to ensure binary compatability in C++, the order in which virtual methods are defined is of importanance. As such, the following order must be maintained.

```
namespace ECOA {

class Container_interface
{
    public:

        virtual void get_last_operation_timestamp(ECOA::timestamp& timestamp) = 0;

        virtual void log_trace(const ECOA::log &log) = 0;

        virtual void log_debug(const ECOA::log &log) = 0;

        virtual void log_info(const ECOA::log &log) = 0;

        virtual void log_warning(const ECOA::log &log) = 0;

        virtual void raise_error(const ECOA::log &log) = 0;

        virtual void raise_fatal_error(const ECOA::log &log) = 0;

        virtual ECOA::return_status get_relative_local_time(ECOA::hr_time
&relative_local_time) = 0;

        virtual ECOA::return_status get_UTC_time(ECOA::global_time &utc_time) = 0;

        virtual ECOA::return_status get_absolute_system_time(ECOA::global_time
&absolute_system_time) = 0;

        virtual void get_relative_local_time_resolution(ECOA::duration
&relative_local_time_resolution) = 0;

        virtual void get_UTC_time_resolution(ECOA::duration &utc_time_resolution) = 0;

        virtual void get_absolute_system_time_resolution(ECOA::duration
&absolute_system_time_resolution) = 0;

};  /* Container_interface */
```

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

The Container of a given Module, which shall implement all methods specific to that Module, is implemented by a concrete class that extends the Container Interface.

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

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

8 Module Context

Not applicable to C++ binding.

This section is however kept for coherency with other language bindings.

8.1 User Module Context

In C++, the User Module Context shall be declared as private member variables within the Module Implementation class. Additionally a pointer to the Container object is also stored as a private 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 passing a pointer to the Container object as a parameter of the Module Implementation class constructor.

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

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

extern "C" {

    #module_impl_name#_interface*
    #module_impl_name#__new_instance(#module_impl_name#_container* container);

}

class #module_impl_name# : public virtual #module_impl_name#_interface
{
    public:
       // The constructor of the Component shall have the following
       // signature:
       #module_impl_name#(#module_impl_name#_container* container);

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


    private:
       // the Module Implementation shall hold a Container pointer
       // which is passed within the constructor
       #module_impl_name#_container* container;

       // user data for this module implementation must be declared here as
       // public, protected or private attributes
       int myCounter;

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

9 Types

This section describes the convention for creating namespaces, and how the ECOA pre-defined types and derived types are represented in C++.

9.1 Filenames and Namespace

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

The syntax for declaring a data type #data_type_name# and variable of #variable_name# is:

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


namespace #namespace1# {
namespace #namespace2# {
[…]
namespace #namespacen# {

#data_type_name# #variable_name#;
// others definitions of this namespace will follow here:

} /* #namespacen#  */
[…]
} /* #namespace2#  */
} /* #namespace1#  */
```

9.2 Predefined Types

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

Table 2 – C++ Predefined Type Mapping

ECOA Predefined 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: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 following set of predefined constants:

Table 3 – 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.

9.2.1 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,
            INVALID_SERVICE_ID = 7,
            CLOCK_UNSYNCHRONIZED = 8,
            INVALID_TRANSITION = 9,
            RESOURCE_NOT_AVAILABLE = 10,
            OPERATION_NOT_AVAILABLE = 11,
            PENDING_STATE_TRANSITION = 12
    };
    inline void operator = (ECOA::uint32 i) { value = i; }
    inline operator ECOA::uint32 () const { return value; }
};

[…]

} /* ECOA */
```

9.2.2 ECOA:hr_time

The binding for time is:

```
namespace ECOA {

[…]

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

[…]

} /* ECOA */
```

9.2.3 ECOA:global_time

Global time is represented as:

```
namespace ECOA {

[…]

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

[…]

} /* ECOA */
```

9.2.4 ECOA:duration

Duration is represented as:

```
namespace ECOA {

[…]

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

[…]

} /* ECOA */
```

9.2.5 ECOA:timestamp

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

```
namespace ECOA {

[…]

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

[…]

} /* ECOA */
```

9.2.6 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 */
```

9.2.7 ECOA:module_states_type

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

```
namespace ECOA {

[…]

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

```
};
[…]

} /* ECOA */
```

9.2.8 ECOA:module_error_type

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

```
struct module_error_type {
        ECOA::uint32 value;
        enum EnumValues {
            ERROR = 0,
            FATAL_ERROR = 1
        };
        inline void operator = (ECOA::uint32 i) { value = i; }
        inline operator ECOA::uint32() const { return value; }
};
```

9.2.9 ECOA:error_id

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

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

9.2.10 ECOA:asset_id

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

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

9.2.11 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; }
};
```

9.2.12 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_INPUT_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; }
};
```

9.2.13 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_COMPONENT = 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; }
};
```

9.3 Derived Types

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

9.3.1 Simple Types

The syntax for defining a Simple Type #simple_type_name# refined from a Predefined Type #predef_type_name# in C++ is defined below.

```
typedef #predef_type_name# #simple_type_name#;
```

Optional minRange or maxRange constant definitions must be provided after the type definitions where required as follows:

```
static const #predef_type_name# #complete_simple_type_name#_minRange = #minrange_value#;
```

```
static const #predef_type_name# #complete_simple_type_name#_maxRange = #maxrange_value#;
```

9.3.2 Constants

The syntax for declaring a Constant called "`#contant_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.

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; }
};
```

Where:

- `#basic_type_name#` is ECOA::boolean8, ECOA::int8, ECOA::char8, ECOA::byte, ECOA::int16, ECOA::int32, ECOA::int64, ECOA::uint8, ECOA::uint16 or ECOA::uint32.
- `#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 fieds in the struct shall follow the order of fields used in the XML definition.

```
/*
 *  #selector_type_name# can be of any simple predefined 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#;
```

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.

Note. In order to ensure binary compatability in C++, the order in which virtual methods are defined is of importananance. As such, the order must be as identified in section 6.4.

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


class #module_impl_name#_interface: public virtual ECOA::Module_interface
{
   public:

      //…

        virtual void #operation_name#__request_received (const ECOA::uint32 ID, const #parameters_in#)
= 0;

        //…

}; /* #module_impl_name#_interface */
```

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#_interface.hpp
 * This is the Module Interface class for Module #module_impl_name#
 * This file is generated by the ECOA tools and shall not be modified
 */


class #module_impl_name#_interface: public virtual ECOA::Module_interface
{
   public:

      //…
```

```
        virtual void #operation_name#__response_received (const ECOA::uint32 ID, const
ECOA::return_status status, const #parameters_out#) = 0;

        //…
}; /* #module_impl_name#_interface */
```

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

10.1.2 Versioned Data

10.1.2.1 Updated

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

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


class #module_impl_name#_interface: public virtual ECOA::Module_interface
{
   public:

        //…

        virtual void #operation_name#__updated(const ECOA::return_status status,
#operation_name#_handle & data_handle) = 0;

        //…

}; /* #module_impl_name#_interface */
```

10.1.3 Events

10.1.3.1 Received

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

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

class #module_impl_name#_interface: public virtual ECOA::Module_interface
{
   public:

        //…

        virtual void #operation_name#__received(const #parameters#) = 0;

        //…

}; /* #module_impl_name#_interface */
```

10.2 Module Lifecycle

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

10.2.1 Generic Module API

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


class #module_impl_name#_interface : public virtual ECOA::Module_interface
{
   public:

      //…

      // the following methods are inherited from ECOA::Module_interface

        virtual void INITIALIZE__received() = 0;

        virtual void START__received()= 0;

        virtual void STOP__received()= 0;

        virtual void SHUTDOWN__received()= 0;

        virtual void REINITIALIZE__received()= 0;

      //…

}; /* #module_impl_name#_interface */
```

Note: The above operations are applicable to supervision, non-supervision, trigger and dynamic-trigger module instances.

10.2.2 Supervision Module API

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

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

class #supervision_module_impl_name#_interface: public virtual ECOA::Module_interface
{
  public:
  […]
    virtual void lifecycle_notification__#module_instance_name#(ECOA::module_states_type
previous_state , ECOA::module_states_type new_state) = 0;
[…]
};
```

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

10.3 Service Availability

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

10.3.1 Service Availability Changed

The following is the C++ syntax for an operation used by the container to invoke a service availability changed operation to a supervision module instance. The operation will only be available if the component has one or more required services. The reference_id type is an enumeration type defined in the Container Interface (Section 11.4.4).

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


class #supervision_module_impl_name#_interface: public virtual ECOA::Module_interface
{
   public:

      //…

        virtual void service_availability_changed(#supervision_module_impl_name#::reference_id
instance, ECOA::boolean8 available) = 0;

        //…

}; /* #module_impl_name#_interface */
```

10.3.2 Service Provider Changed

The following is the C++ syntax for an operation used by the container to invoke a service provider changed operation to a supervision module instance. The operation will only be available if the component has one or more required services. The reference_id type is an enumeration type defined in the Container Interface (Section 11.4.4).

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


class #supervision_module_impl_name#_interface: public virtual ECOA::Module_interface
{
   public:

      //…
```

```
        virtual void service_provider_changed(#supervision_module_impl_name#::reference_id instance) =
0;

        //…

}; /* #module_impl_name#_interface */
```

10.4 Error_notification binding at application level

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

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


class #supervision_module_impl_name#_interface : public virtual ECOA::Module_interface
{
   public:

      //…

        virtual void error_notification__#module_instance_name#(const ECOA::module_error_type&
module_error_type) = 0;

      //…
}; /* #supervision_module_impl_name#_interface */
```

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.

Note. In order to ensure binary compatability in C++, the order in which virtual methods are defined is of importanance. As such, the order must be as identified in section 6.5.

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
 * This is the Container Interface class for Module #module_impl_name#
 * This file is generated by the ECOA tools and shall not be modified
 */


class #module_impl_name#_container: public virtual ECOA::Container_interface
{
   public:

      //…

       virtual ECOA::return_status #operation_name#__response_send(const ECOA::uint32 ID, const
#parameters_out#) = 0;

       //…


}; /* #module_impl_name#_container */
```

Note: the “#parameters_out# in the above code snippet are the out parameters of the original request, not of this operation: they are passed as ‘const’ values, as they should not be modified by the container. The ID parameter is that which is passed in during the invocation of the request received 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
 * This is the Container Interface class for Module #module_impl_name#
 * This file is generated by the ECOA tools and shall not be modified
 */

class #module_impl_name#_container: public virtual ECOA::Container_interface
{
```

```
   public:

        // …

        virtual ECOA::return_status #operation_name#__request_sync(const #parameters_in#,
#parameters_out#) = 0;

        //…


}; /* #module_impl_name#_container */
```

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
 * This is the Container Interface class for Module #module_impl_name#
 * This file is generated by the ECOA tools and shall not be modified
 */


class #module_impl_name#_container: public virtual ECOA::Container_interface
{
   public:

      //…

        virtual void #operation_name#__request_async(ECOA::uint32& ID, const #parameters_in#) = 0;

        //…


}; /* #module_impl_name#_container */
```

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)

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

#define ECOA_VERSIONED_DATA_HANDLE_PRIVATE_SIZE 32

class #module_impl_name#_container: public virtual ECOA::Container_interface
{
    public:
        /*
         * The following is the data handle structure associated to the data
         * operation called #operation_name# of data-type #type_name#
         */
        typedef struct {
           #type_name#* data; /* pointer to the local copy of the data */
```

```
            ECOA::timestamp timestamp; /* date of the last update of that version of the data */
            ECOA::byte platform_hook[ECOA_VERSIONED_DATA_HANDLE_PRIVATE_SIZE]; /* technical info
associated with the data (opaque for the user, reserved for the infrastructure) */
        } #operation_name#_handle;

      // other operation methods may appear here …

      virtual ECOA::return_status #operation_name#__get_read_access(#operation_name#_handle &
data_handle) = 0;

       virtual ECOA::return_status #operation_name#__release_read_access(#operation_name#_handle &
data_handle) = 0 ;

      // other operation methods may appear here …

}; /* #module_impl_name#_container */
```

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


#define ECOA_VERSIONED_DATA_HANDLE_PRIVATE_SIZE 32

class #module_impl_name#_container: : public virtual ECOA::Container_interface
{
   public:


      //…

        typedef struct {
           #type_name#* data;
           ECOA::timestamp timestamp;
           ECOA::byte platform_hook[ECOA_VERSIONED_DATA_HANDLE_PRIVATE_SIZE];
        } #operation_name#_handle;

       virtual ECOA::return_status #operation_name#__get_write_access(#operation_name#_handle &
data_handle) = 0;

       virtual ECOA::return_status #operation_name#__cancel_write_access(#operation_name#_handle &
data_handle) = 0;

       virtual ECOA::return_status #operation_name#__publish_write_access(#operation_name#_handle &
data_handle) = 0;

       //…

}; /* #module_impl_name#_container */
```

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.hpp
 * This is the Container Interface class for Module #module_impl_name#
 * This file is generated by the ECOA tools and shall not be modified
 */
```

```
class #module_impl_name#_container: public virtual ECOA::Container_interface
{
   public:

      //…

        virtual void #operation_name#__send(const #parameters#) = 0;

        //…

}; /* #module_impl_name#_container */
```

11.2 Properties

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

11.2.1 Get Value

The syntax for Get_Value is shown below where:

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

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


class #module_impl_name#_container: public virtual ECOA::Container_interface
{

   public:

      //…

        virtual void get_#property_name#_value(#property_type_name#& value) = 0;

        //…

}; /* #module_impl_name##_container */
```

11.3 Module Lifecycle

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

11.3.1 Non-Supervision Container API

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

11.3.2 Supervision Container API

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

```
/*
 * @file #supervision_module_impl_name#_container.hpp
```

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

class #supervision_module_impl_name#_container
{
  public:
  […]
    virtual ECOA::return_status STOP__#module_instance_name#() = 0;
    virtual ECOA::return_status START__#module_instance_name#() = 0;
    virtual ECOA::return_status INITIALIZE__#module_instance_name#() = 0;
    virtual ECOA::return_status SHUTDOWN__#module_instance_name#() = 0;
    virtual void get_lifecycle_state__#module_instance_name#(ECOA::module_states_type& current_state)
= 0;
[…]
};
```

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

11.4 Service Availability

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

11.4.1 Set Service Availability (Server Side)

The following is the C++ syntax for invoking the set service availability operation by a supervision module instance. The operation will only be available if the component has one or more provided services. The service instance is identified by the enumeration type service_id defined in the Container Interface (Section 11.4.3).

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


class #supervision_module_impl_name#_container: public virtual ECOA::Container_interface
{
   public:

      //…

       virtual ECOA::return_status set_service_availability(service_id instance, ECOA::boolean8
available) = 0;

       //…


}; /* #module_impl_name#_container */
```

11.4.2 Get Service Availability (Client Side)

The following is the C++ syntax for invoking the get service availability operation by a supervision module instance. The operation will only be available if the component has one or more required services. The service instance is identified by the enumeration type reference_id defined in the Container Interface (Section 11.4.4).

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


class #supervision_module_impl_name#_container: public virtual ECOA::Container_interface
{
   public:

      //…

        virtual ECOA::return_status get_service_availability(reference_id instance, ECOA::boolean8
&available) = 0;

        //…


}; /* #module_impl_name#_container */
```

11.4.3 Service ID Enumeration

In C++ `service_id` translates to `service_id`.

This enumeration has a value for each element *<service/>* defined in the file .componentType, whose name is given by its attribute *name* and the numeric value is the position (starting by 0).

The service_id enumeration is only available if the component provides one or more services.

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


class #supervision_module_impl_name#_container: public virtual ECOA::Container_interface
{
   public:

struct service_id {
    ECOA::uint32 value;
    enum  EnumValues {
        #service_instance_name# = 0
    };
    inline void operator = (ECOA::uint32 i) { value = i; }
    inline operator ECOA::uint32 () const { return value; }
};

}; /* #module_impl_name#_container */
```

11.4.4 Reference ID Enumeration

In C++ `reference_id` translates to `reference_id`.

This enumeration has a value for each element *<reference/>* defined in the file .componentType, whose name is given by its attribute *name* and the numeric value is the position (starting by 0).

The reference_id enumeration is only available if the component requires one or more services.

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


class #supervision_module_impl_name#_container: public virtual ECOA::Container_interface
{
   public:

struct reference_id {
    ECOA::uint32 value;
    enum  EnumValues {
        #reference_instance_name# = 0
    };
    inline void operator = (ECOA::uint32 i) { value = i; }
    inline operator ECOA::uint32 () const { return value; }
};

}; /* #module_impl_name#_container */
```

11.5 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

```
/*
 * @file #module_impl_name#_container.hpp
 * This is the Container Interface class
 * This file is generated by the ECOA tools and shall not be modified
 */

class #module_impl_name#_container : public virtual ECOA::Container_interface
{
    public:

        // the following method shall be implemented by the Container to provide
        // the Module Implementation with a Logging functionnality.
        virtual void log_trace(const ECOA::log &);

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

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

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

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

        virtual void raise_fatal_error(const ECOA::log &log);
}; /* #module_impl_name#_container */
```

Definitions above are already described in section 6.5. This section is however kept for coherency with other language bindings.

11.6 Time Services

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

11.6.1 Get_Relative_Local_Time

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


class #module_impl_name#_container: public virtual ECOA::Container_interface
{
   public:

      //…

       virtual ECOA::return_status get_relative_local_time(ECOA::hr_time &relative_local_time) = 0;

        //…


}; /* #module_impl_name#_container */
```

11.6.2 Get_UTC_Time

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


class #module_impl_name#_container: public virtual ECOA::Container_interface
{
   public:

      //…

       virtual ECOA::return_status get_UTC_time(ECOA::global_time &utc_time) = 0;

        //…


}; /* #module_impl_name#_container */
```

11.6.3 Get_Absolute_System_Time

```
/*
```

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


class #module_impl_name#_container: public virtual ECOA::Container_interface
{
   public:

      //…

      virtual ECOA::return_status get_absolute_system_time(ECOA::global_time &absolute_system_time) =
0;

        //…


}; /* #module_impl_name#_container */
```

11.6.4 Get_Relative_Local_Time_Resolution

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


class #module_impl_name#_container: public virtual ECOA::Container_interface
{
   public:

      //…

      virtual void get_relative_local_time_resolution(ECOA::duration &relative_local_time_resolution)
= 0;

        //…


}; /* #module_impl_name#_container */
```

11.6.5 Get_UTC_Time_Resolution

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


class #module_impl_name#_container: public virtual ECOA::Container_interface
{
   public:

      //…

      virtual void get_UTC_time_resolution(ECOA::duration &utc_time_resolution) = 0;

        //…
```

```
}; /* #module_impl_name#_container */
```

11.6.6 Get_Absolute_System_Time_Resolution

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


class #module_impl_name#_container: public virtual ECOA::Container_interface
{
   public:

      //…

      virtual void get_absolute_system_time_resolution(ECOA::duration
&absolute_system_time_resolution) = 0;

        //…


}; /* #module_impl_name#_container */
```

12 Fault Handler Interface

12.1 Error_notification binding 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#_interface.h
 * This is the Module Interface class for the Fault handler #fault_handler_impl_name#
 * This file is generated by the ECOA tools and shall not be modified
 */


class fault_handler_impl_name#_interface : public virtual ECOA::Module_interface
{
   public:

      //…

       virtual void error_notification(ECOA::error_id error_id,
                                        const ECOA:timestamp& timestamp,
                                        ECOA::asset_id asset_id,
                                        ECOA::asset_type asset_type,
                                        ECOA::error_type error_type
                                       ) = 0;

      //…
}; /* #fault_handler_impl_name#_interface */
```

12.2 Recovery_action binding

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
 * This is the Container Interface class
 * This file is generated by the ECOA tools and shall not be modified
 */

class #fault_handler_impl_name#_container : public virtual ECOA::Container_interface
{
    public:

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

}; /* #module_impl_name#_container */
```

13 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 = -1.7976931348623158e+308D;
  static const double64 DOUBLE64_MAX =  1.7976931348623158e+308D;

  /* 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,
      INVALID_SERVICE_ID =         7,
      CLOCK_UNSYNCHRONIZED =       8,
      INVALID_TRANSITION =         9,
      RESOURCE_NOT_AVAILABLE =    10,
      OPERATION_NOT_AVAILABLE =   11,
      PENDING_STATE_TRANSITION =  12
    };
    inline void operator = (ECOA::uint32 i) { value = i; }
    inline operator ECOA::uint32 () const { return value; }
  };

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

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

  /* ECOA:module_states_type */
```

```
struct module_states_type {
  uint32 value;
  enum EnumValues {
    IDLE = 0,
    READY = 1,
    RUNNING = 2
  };
  inline void operator = (uint32 i) { value = i; }
  inline operator uint32() const { return value; }
};

/* ECOA:module_error_type */
struct module_error_type {
  ECOA::uint32 value;
  enum EnumValues {
    ERROR = 0,
    FATAL_ERROR = 1
  };
  inline void operator = (ECOA::uint32 i) { value = i; }
  inline operator ECOA::uint32() const { return value; }
};

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

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

/* ECOA:asset_id */
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; }
};

/* 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_INPUT_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; }
```

```
  };

  /* ECOA:recovery_action_type */
  struct recovery_action_type {
    ECOA::uint32 value;
    enum EnumValues {
      SHUTDOWN_COMPONENT = 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; }
  };

  class Module_interface
  {
  public:
    // virtual destructor
    virtual ~Module_interface() {}
    virtual void INITIALIZE__received() = 0;
    virtual void START__received() = 0;
    virtual void STOP__received() = 0;
    virtual void SHUTDOWN__received() = 0;
    virtual void REINITIALIZE__received() = 0;
  }; /* Module_interface */

  class Container_interface
  {
  public:
    virtual void get_last_operation_timestamp(ECOA::timestamp& timestamp) = 0;
    virtual void log_trace(const ECOA::log &log) = 0;
    virtual void log_debug(const ECOA::log &log) = 0;
    virtual void log_info(const ECOA::log &log) = 0;
    virtual void log_warning(const ECOA::log &log) = 0;
    virtual void raise_error(const ECOA::log &log) = 0;
    virtual void raise_fatal_error(const ECOA::log &log) = 0;
    virtual ECOA::return_status get_relative_local_time(ECOA::hr_time &relative_local_time) = 0;
    virtual ECOA::return_status get_UTC_time(ECOA::global_time &utc_time) = 0;
    virtual ECOA::return_status get_absolute_system_time(ECOA::global_time &absolute_system_time) = 0;
    virtual void get_relative_local_time_resolution(ECOA::duration &relative_local_time_resolution) =
0;
    virtual void get_UTC_time_resolution(ECOA::duration &utc_time_resolution) = 0;
    virtual void get_absolute_system_time_resolution(ECOA::duration &absolute_system_time_resolution)
= 0;
  }; /* Container_interface */

} /* ECOA */

#endif /* __ECOA_HPP__ */
```