European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 8: Software Interface Reference Manual

BAE Ref No: IAWG-ECOA-TR-010
Dassault Ref No: DGT 144485-B

Issue: 2

Prepared by
BAE Systems (Operations) Limited and Dassault Aviation

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

1 Table of Contents

2 List of Figures

3 List of Tables

4 Abbreviations

API	Application Programming Interface
CPU	Central Processing Unit
ECOA	European Component Oriented Architecture
HR	High Resolution
POSIX	Portable Operating System Interface
RT	Real Time
SW	Software
XML	eXtensible Markup Language
UTC	Coordinated Universal Time

5 Introduction

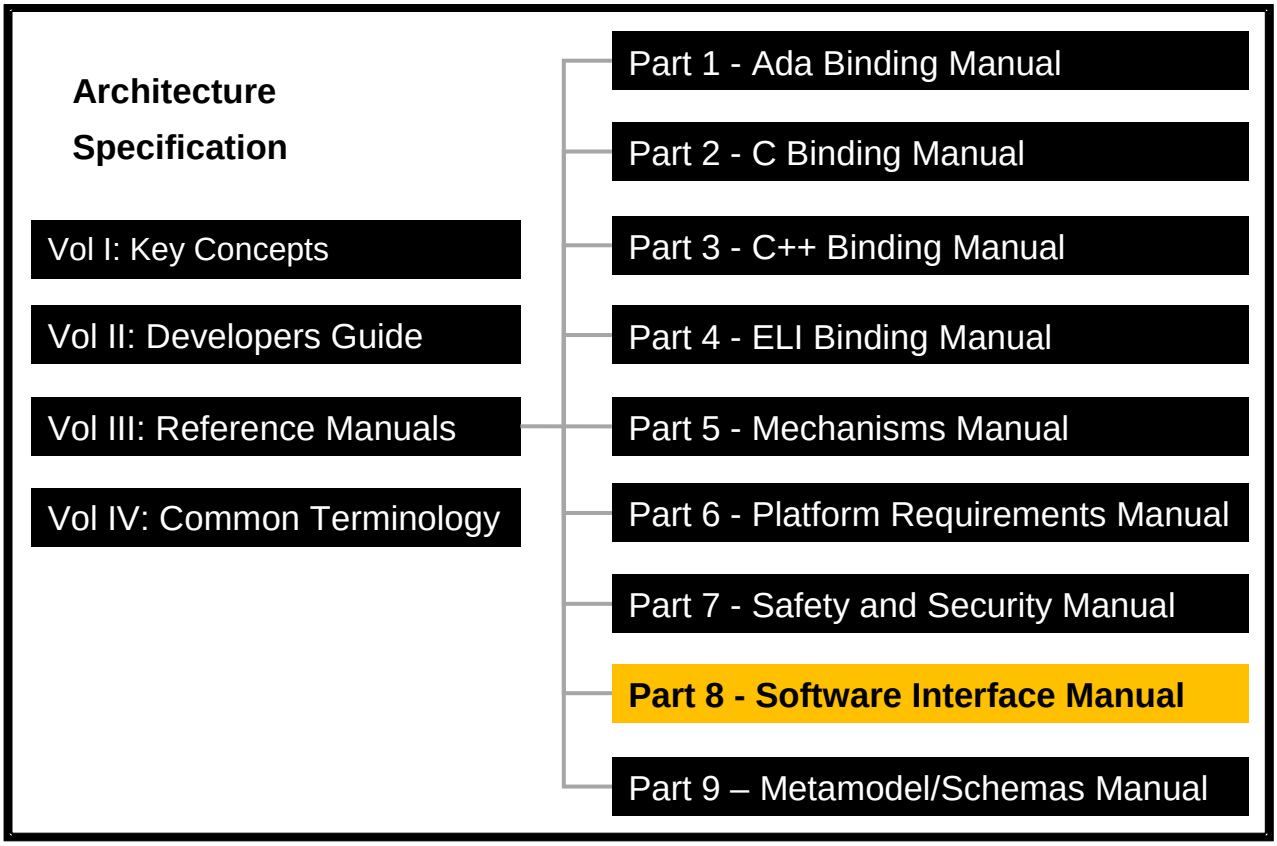

Figure 1 – ECOA Documentation

The Architecture Specification provides the definitive specification for creating ECOA-based systems. It describes the standardised programming interfaces and data-model that allow a developer to construct an ECOA-based system. It is introduced in Key Concepts (Reference 1) and uses terms defined in the Common Terminology (Reference 11). For this reason, the reader should read these documents, prior to this document. The details of the other documents comprising the rest of the Architecture Specification can be found in Section 12.

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

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

This document comprises Volume III Part 8 of the ECOA Architecture Specification, and describes the software interfaces used.

In an ECOA system, all interactions between Modules that implement Application Software Components rely on three mechanisms: event, versioned data, and request-response. In addition calls and handlers exist for infrastructure services to allow the management of the runtime lifecycle, logging, faults and time.

This document describes Application Software Component Interface (API) between modules and the containers that host them. The API, shown in Figure 2, comprises the Module Interface and the Container Interface and is referred to as the Application Software Component Interface:

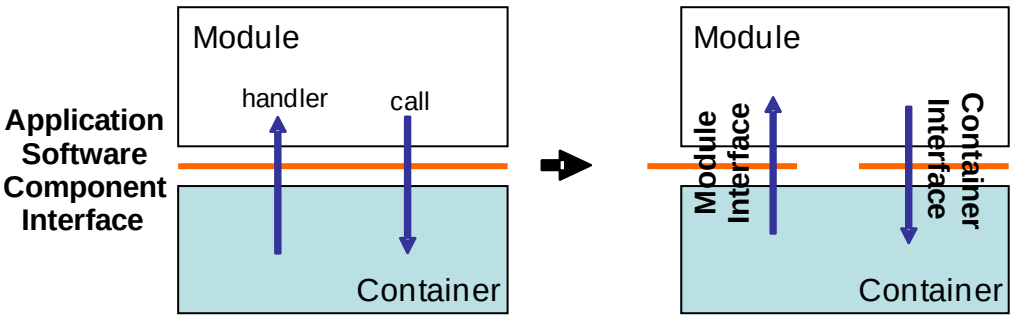

Figure 2 – Module and Container Interface

- The Module Interface specifies the interface to a module, which is used by the container to call module operations.
- The Container Interface specifies the functions that the container provides for a module.

Different bindings provide mappings for particular programming languages. Currently three bindings are available: for C [Ref 4], C++ [Ref 5] and Ada [Ref 3].

This document also describes the Parameters for the operations in the API and the types that the API relies on.

The information in this document is based on v1.9.0 of the ECOA meta-model.

This document is structured as follows:

- Section 6 describes the Module to Language Mapping
- Section 7 describes the Parameters for operations;
- Section 8 describes the Module Context
- Section 9 describes the Type libraries
- Section 10 describes the Module Interface and
- Section 11 describes the Container Interface.

6 Module to Language Mapping

This section gives an overview of the Module Interface and Container Interface APIs, in terms of the filenames and overall structure of the files. Refer to this section in the required language binding for details relevant to that specific language

Sections 10 and 11 contain prototype definitions of the Application Framework operations using C like syntax: the correct syntax is given by the appropriate language binding.

The name of each operation shall include the Module Implementation name for those languages that do not support namespacing. The following symbolic names are used in the prototypes:

- **#module_impl_name#** is the name of the module implementation – the name is used for API generation.
- **#module_instance_name#** is the name of a module instance – this name is used for deployment or for lifecycle purposes,
- **#operation_name#** is the name of the module operation (event, request-response or versioned data),
- **#parameters_in#** and **#parameters_out#** respectively correspond to the ordered list of input and output parameters specified for a Request_Received, Response_Received, Request_Sync, Request_Async, Request_Received_Deferred, or a Reply_Deferred operation,
- **#parameters#** corresponds to the ordered list of parameters specified for an event Send or event Received operation,
- **#type_name#** is the name of a data-type[1],
- **#context#** will be used to represent the reference to the context associated with a module instance.

Table 1 details the Module and Container Interface APIs. The actual API will include the name of the operation and module. How this is done is specified in the language independent section referenced in the table. The reader must refer to the appropriate language binding document to determine the actual syntax for a specific language.

[1] #type_name# may be extended by the addition of a qualifying prefix where a specific kind of type is indicated, as in *#record_type_name#*.

Category	Abstract API Name	Container Operation	Module Operation	Section
Events API	Event_Send	Yes	No	11.1.3.1
	Event_Received	No	Yes	10.1.3.1
Request Response API	Request_Sync	Yes	No	11.1.1.1
	Request_Async	Yes	No	11.1.1.2
	Request_Received	No	Yes	10.1.1.1
	Response_Received	No	Yes	10.1.1.3
	Request_Received_Deferred	No	Yes	10.1.1.2
	Reply_Deferred	Yes	No	11.1.1.3
Versioned Data API	Get_Read_Access	Yes	No	11.1.2.1
	Release_Read_Access	Yes	No	11.1.2.2
	Updated	No	Yes	10.1.2.1
	Get_Write_Access	Yes	No	11.1.2.3
	Cancel_Write_Access	Yes	No	11.1.2.4
	Publish_Write_Access	Yes	No	11.1.2.5
Properties API	Get_Value	Yes	No	11.2.1
Runtime Lifecycle API	Initialize_Received	No	Yes	10.3.1
	Start_Received	No	Yes	
	Stop_Received	No	Yes	
	Shutdown_Received	No	Yes	
	Reinitialize_Received	No	Yes	
Logging and Fault Management Services API	Log_Debug	Yes	No	11.6
	Log_Trace	Yes	No	
	Log_Info	Yes	No	
	Log_Warning	Yes	No	
	Raise_Error	Yes	No	
	Raise_Fatal_Error	Yes	No	
Time Services API	Get_Relative_Local_Time	Yes	No	11.7
	Get_UTC_Time	Yes	No	
	Get_Absolute_System_Time	Yes	No	
Lifecycle Management API (Supervision Modules Only)	Lifecycle_Notification	No	Yes	10.3.2
	Get_Lifecycle_State	Yes	No	11.4.2
	Stop_Module	Yes	No	
	Start_Module	Yes	No	
	Initialize_Module	Yes	No	
	Shutdown_Module	Yes	No	
Error Handler API (Supervision Modules Only)	Exception_Notification_Handler	No	Yes	10.5

Category	Abstract API Name	Container Operation	Module Operation	Section
Service Availability API (Supervision Modules Only)	Set_Service_Availability	Yes	No	11.5.1
	Get_Service_Availability	Yes	No	11.5.2

Table 1 - Module and Container Interfaces

7 Parameters

Request-response and event operations may have parameters associated with them:

- Request-response operations: may have inputs and outputs.
- Events: may have inputs.

All parameters must be ECOA pre-defined types or be defined in a type library.

The order of parameters of an operation is described in the Service Definitions, Component Definition and Component Implementation, and must be the same in all cases.

The manner in which parameters are passed is language dependent and is described in the individual language bindings.

8 Module Context

It is required that the same implementation of a module can be instantiated several times, possibly within the same protection domain, without causing any symbol collision. To achieve this requirement, it is expected, for example, that the implementer of a C or C++ Module would not use any static (either global or local) variables within the module (except for constants). To this end, modules are coded with instance specific data blocks referred to as the “module context".

The purpose of this “module context” is to hold all the private data that will be used:

- by the Container and the ECOA infrastructure to handle the Module Instance (infrastructure-level technical data),
- by the Module Instance itself to support its functions (user-defined local private data).

The use and the declaration of the “Module Context” structure may be adapted for each language binding.

For non-OO languages, the “Module Context” will be represented as a structure that shall hold both the user local data (called “User Module Context”) and all the infrastructure-level technical and specific part of “Module Context” (such technical data won’t be specified in this document as they are implementation dependant). For this reason, the Module Context may be generated by the ECOA infrastructure within the Container Interface Header, and be extended by a user defined "User Module Context" structure.

With OO languages, the Module Instance will be instantiated as an object of a Module Implementation class declared by the user; its associated Container will be associated to an instance of an ECOA-generated Module Container class. All the "User Module Context" shall be declared within the user Module Implementation class as its private attributes and accessed through public helper methods. The infrastructure-level technical data shall be declared by the ECOA-infrastructure within the corresponding (generated) Module Container class. In addition, the entry-points declared in the Container Interface are represented as methods of the Container object, so the Module Instance object must have access to its corresponding Container object to enable it to call these methods. This can be achieved by passing a pointer to the Container object as a parameter of the constructor of the Module Implementation class. The Module Instance object will use a private attribute to store this pointer to the Container object for future use.

The language bindings specify the exact syntax required for the Module User Context.

8.1 Timestamping

Freshness of data is an important consideration in a mission system and for this reason timestamping of operations (i.e. communication) is supported.

A timestamp point is related to the origin of the operation (Sender). The timestamp allows the user of the timestamp to rebuild a chronogram based on the same reference, the sender’s clock. The ECOA infrastructure (i.e. container) will be able to record timestamps for operations as shown in Table 2.

Operation API	Timestamp point	Description
`Event_Send`	When Module calls the Container API	When an event is sent by the requiring or providing Container
`Request_Sync`	When Module calls the Container API	When a request is sent by the requiring Container
`Request_Async`	When Module calls the Container API	

Operation API	Timestamp point	Description
`Request_Received`	When the called Module returns from the request operation or it calls the deferred response API	When a response is sent by the provider Container
`Publish_Write_Access`	When Module calls container API (publish)	When data is published by the provider Container
`Updated`	When Module calls container API (publish)	When data is published by the provider Container

Table 2 – Timestamp Points

At present operation timestamps will always be provided by the infrastructure. In future timestamps may be made optional (for performance reasons) and a component may be able to specify whether it provides or requires a timestamp.

8.1.1 Request Response and Events

The timestamp is associated with the sent event, sent request or sent response. These timestamps available to the receiving module in the module instance context.

8.1.2 Versioned Data

The timestamp is associated with publication of the version of data that is being read.

9 Types

The API relies on a set of pre-defined types, which can be used to construct user defined complex types. These types are used by operations on the Module and Container Interfaces. Namespaces are used to organise the types into separate libraries.

9.1 Namespaces

Namespaces are used to organise the types used by an ECOA system into disjoint sets, or libraries. The namespaces are organised in a hierarchical manner, and all of the ECOA namespaces are subordinate to the ECOA base namespace as shown in Figure 3. Application based namespaces that are not subordinate to the ECOA namespace are also allowed.

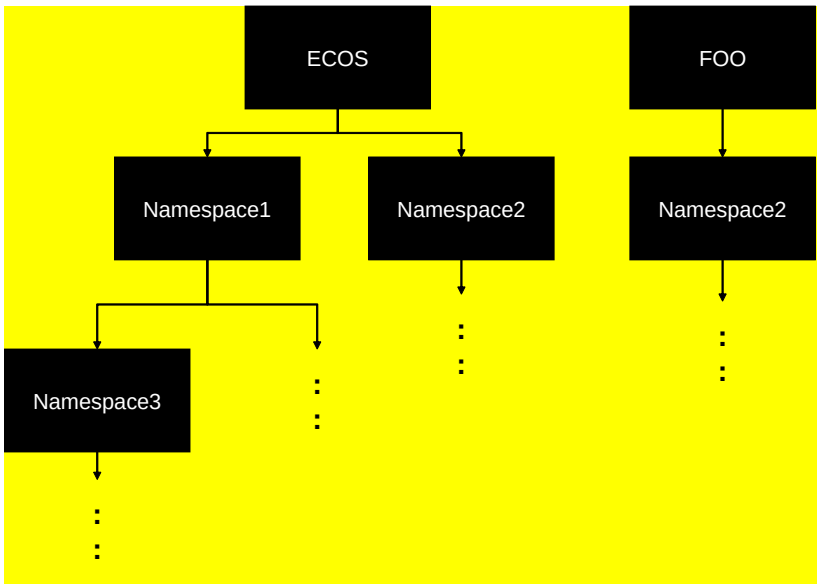

Figure 3 – Namespaces

Type names within the same namespace shall be unique. All types declared in the same namespace are located in the same file in the model: this file will usually be automatically generated by the ECOA toolset from the XML descriptions. The header file name complies with the following pattern:

```
#namespace1#__#namespace2#__[…]__#namespacen#
```

The file extension is language specific.

9.2 Predefined Types

A number of portable pre-defined basic types are provided within the ECOA namespace that should be used to write portable code. They are used for all data interchange between modules in an implementation. These portable types do not preclude the use of pre-existing language types, error handling or exception mechanisms. Mappings for specific languages are described by the bindings.

All of the ECOA pre-defined types, which are listed in Table 2, may be used directly in the XML descriptions without using the ECOA namespace.

ECOA Predefined Type	Description	XML Representation
ECOA:boolean8	8-bit boolean	boolean8 or ECOA:boolean8
ECOA:int8	8-bit signed integer	int8 or ECOA:int8
ECOA:char8	8-bit ASCII character	char8 or ECOA:char8
ECOA:byte	byte	byte or ECOA:byte
ECOA:int16	16 bits signed integer	int16 or ECOA:int16
ECOA:int32	32-bits signed integer	int32 or ECOA:int32
ECOA:int64	64 bits signed integer	int64 or ECOA:int64
ECOA:uint8	8 bit unsigned integer	uint8 or ECOA:uint8
ECOA:uint16	16-bit unsigned integer	uint16 or ECOA:uint16
ECOA:uint32	32-bit unsigned integer	uint32 or ECOA:uint32
ECOA:uint64	64-bit unsigned integer	uint64 or ECOA:uint64
ECOA:float32	Single precision IEEE 754 floating-point	float32 or ECOA:float32
ECOA:double64	Double precision IEEE 754 floating-point	double64 or ECOA:double64

Table 3 – ECOA Predefined Types

ECOA Predefined Type	Constant	Constant Value
ECOA:boolean8	TRUE	1
	FALSE	0
ECOA:int8	INT8_MIN	-128
	INT8_MAX	127
ECOA:char8	CHAR8_MIN	0
	CHAR8_MAX	127[2]
ECOA:byte	BYTE_MIN	0
	BYTE_MAX	255
ECOA:int16	INT16_MIN	-32768
	INT16_MAX	32767
ECOA:int32	INT32_MIN	-2147483648
	INT32_MAX	2147483647
ECOA:int64	INT64_MIN	-9223372036854775808
	INT64_MAX	9223372036854775807

[2] ECOA:char8 is an ASCII character, and as such its range is 0 to 127, however the 7 bit ASCII code uses 8 bits of storage, with the upper bit set to zero, because of this values in the range 128 to 255 are invalid.

ECOA Predefined Type	Constant	Constant Value
ECOA:uint8	UINT8_MIN	0
	UINT8_MAX	255
ECOA:uint16	UINT16_MIN	0
	UINT16_MAX	65535
ECOA:uint32	UINT32_MIN	0
	UINT32_MAX	4294967295
ECOA:uint64	UINT64_MIN	0
	UINT64_MAX	18446744073709551615
ECOA:float32	FLOAT32_MIN	-3.402823466e+38F
	FLOAT32_MAX	3.402823466e+38F
ECOA:double64	DOUBLE64_MIN	-1.7976931348623158e+308
	DOUBLE64_MAX	1.7976931348623158e+308

Table 4 – ECOA Predefined Constants

For all the pre-defined data types it shall be possible to determine the minimum and maximum values. In C/C++, for example these will be implemented as macros. These are also defined in the base namespace.

It is recommended to map boolean8, char8 and byte onto unsigned types of programming languages.

9.2.1 ECOA:error

The data type `ECOA:error` is an enumeration (using ECOA:uint32 as its base type) declared in the ECOA namespace, which is used to specify the return status of applicable application-software component interface API functions. This data type is also used by the error handler. The enumeration values are:

ECOA:OK = 0	No error has occurred
ECOA:INVALID_HANDLE = 1	An invalid handle has been used
ECOA:DATA_NOT_INITIALIZED = 2	The data has never been written
ECOA:NO_DATA = 3	The call is not able to provide any data
ECOA:INVALID_IDENTIFIER = 4	An invalid ID has been used.
ECOA:NO_RESPONSE = 5	No response was received for a request
ECOA:OPERATION_ABORTED = 6	The requested operation was aborted
ECOA:UNKNOWN_SERVICE_ID = 7	The service ID is not known
ECOA:CLOCK_UNSYNCHRONIZED = 8	The clock is not synchronised
ECOA:INVALID_STATE = 9	An invalid state has been used.
ECOA:INVALID_TRANSITION = 10	An invalid transition has been used.
ECOA:RESOURCE_NOT_AVAILABLE = 11	Insufficient resource is available to perform the operation.
ECOA:OPERATION_NOT_AVAILABLE = 12	The requested operation is not available.

The ECOA:error is an enumeration (see section 9.3.3) defined in the ECOA namespace as follows:

```
<enum name="error" type="uint32">
            <value name="OK" valnum="0"/>
            <value name="INVALID_HANDLE" valnum="1" />
            <value name="DATA_NOT_INITIALIZED" valnum="2" />
```

```
            <value name="NO_DATA" valnum="3" />
            <value name="INVALID_IDENTIFIER" valnum="4" />
            <value name="NO_RESPONSE" valnum="5" />
            <value name="OPERATION_ABORTED" valnum="6" />
            <value name="UNKNOWN_SERVICE_ID" valnum="7" />
            <value name="CLOCK_UNSYNCHRONIZED" valnum="8" />
            <value name="INVALID_STATE" valnum="9" />
            <value name="INVALID_TRANSITION" valnum="10" />
            <value name="RESOURCE_NOT_AVAILABLE" valnum="11" />
            <value name="OPERATION_NOT_AVAILABLE" valnum="12" />
</enum>
```

9.2.2 ECOA:hr_time

A type used as a local (high-resolution) time source. The `ECOA:hr_time` data-type is a record composed of the following fields:

- **`ECOA:uint32 seconds.`** Seconds elapsed since some reference point in time. The value shall be positive.
- **`ECOA:uint32 nanoseconds`**. Nanoseconds measured within the current second. The value shall be between 0 and 1.10^9.

The ECOA:hr_time is a record (see section 9.3.4) defined in the ECOA namespace as follows:

```
<record name="hr_time">
   <field type="uint32" name=”seconds" />
   <field type="uint32" name="nanoseconds" />
</record>
```

9.2.3 ECOA:global_time

A type used for global time source (*e.g.* UTC time). `ECOA:global_time` is a record composed of the following fields:

- **`ECOA:uint32 seconds`**. Seconds elapsed since the POSIX Epoch (1st of January, 1970). The value shall be positive.
- **`ECOA:uint32 nanoseconds`**. Nanoseconds measured within the current second. The value shall be between 0 and 1.10^9.

The ECOA:global_time is a record (see section 9.3.4) defined in the ECOA namespace as follows:

```
<record name="global_time">
   <field type="uint32" name=”seconds" />
   <field type="uint32" name="nanoseconds" />
</record>
```

9.2.4 ECOA:duration

A type used for operations that result in communications of delay or duration from one module to another. ECOA:duration is a record composed of the following fields:

- **`ECOA:uint32 seconds. The value shall be positive.`**
- **`ECOA:uint32 nanoseconds. Nanoseconds measured within the current second. The value shall be between 0 and 1.10^9.`**

The ECOA:duration is a record (see section 9.3.4) defined in the ECOA namespace as follows:

```
<record name="duration">
   <field type="uint32" name=”seconds" />
```

```
   <field type="uint32" name="nanoseconds" />
</record>
```

9.2.5 ECOA:timestamp

A type, set using `ECOA:global_time` , used for timestamping operations that result in communications from one Module Instance to another. `ECOA:timestamp` is a record composed of the following fields:

- **`ECOA:uint32 seconds`**. Seconds elapsed since the POSIX Epoch (1st of January, 1970). The value shall be positive.
- **`ECOA:uint32 nanoseconds`**. Nanoseconds measured within the current second. The value shall be between 0 and 1.10^9.

The ECOA:duration is a record (see section 9.3.4) defined in the ECOA namespace as follows:

```
<record name="timestamp">
   <field type="uint32" name=”seconds" />
   <field type="uint32" name="nanoseconds" />
</record>
```

9.2.6 ECOA:log

`ECOA:log` is a variable array of 256 **`ECOA:char8`** elements, that defines how a fault or information report is stored. The type is constrained to enable portability, because some implementations may not be able to support unconstrained logging. See Section 11.6 for information about logging and fault management.

Using a variable array potentially improves performance, because the size of the log can be efficiently managed.

The ECOA:log is a record (see section 9.3.4) defined in the ECOA namespace as follows:

```
<array name="log" itemType="char8" maxNumber="256" />
```

9.2.7 ECOA:component_states_type

The data type `ECOA:component_states_type` is an enumeration (using ECOA:uint32 as its base type) declared in the ECOA namespace, which is used to specify the status of an application-software component. The enumeration values are:

ECOA:IDLE = 0	The component is idle
ECOA:INITIALIZING = 1	The component is initializing
ECOA:STOPPED = 2	The component has stopped
ECOA:STOPPING = 3	The component is stopping
ECOA:RUNNING = 4	The component is running
ECOA:STARTING = 5	The component is starting
ECOA:FINISHING = 6	The component is shutting down
ECOA:FAILURE = 7	The component has failed

The ECOA:component_states_type is an enumeration (see section 9.3.3) defined in the ECOA namespace as follows:

```
<enum name="component_states_type" type="uint32">
             <value name="IDLE" valnum="0"/>
             <value name="INITIALIZING" valnum="1" />
             <value name="DATA_NOT_INITIALIZED" valnum="2" />
             <value name="STOPPED" valnum="3" />
             <value name="RUNNING" valnum="4" />
             <value name="STARTING" valnum="5" />
             <value name="FINISHING" valnum="6" />
```

```
            <value name="FAILURE" valnum="7" />
</enum>
```

9.2.8 ECOA:module_states_type

The data type `ECOA:module_states_type` is an enumeration (using ECOA:uint32 as its base type) declared in the ECOA namespace, which is used to specify the status of modules. The enumeration values are:

ECOA:IDLE = 0	The module is idle
ECOA:READY = 1	The module is ready
ECOA:RUNNING = 2	The module is running

The ECOA:module_states_type is an enumeration (see section 9.3.3) defined in the ECOA namespace as follows:

```
<enum name="module_states_type" type="uint32">
            <value name="IDLE" valnum="0"/>
            <value name="READY" valnum="1" />
            <value name="RUNNING" valnum="2" />
</enum>
```

9.2.9 ECOA:exception

The data type `ECOA:exception` is declared in the ECOA namespace, which is used by the error handler to provide details of the error. `ECOA:exception` is a record composed of the following fields:

- **`ECOA:timestamp timestamp`**. Time the error was raised.
- **`ECOA:service_id service_id`**. The ID of a service instance which raised the error.
- **`ECOA:operation_id operation_id`**. The ID of an operation which raised the error.
- **`ECOA:module_id module_id`**. The ID of an operation which raised the error.
- **`ECOA:exception_id exception_id`**. An enumeration, used to identify the type of error.

The ECOA:exception is a record (see section 9.3.4) defined in the ECOA namespace as follows:

```
<record name="exception">
   <field type="timestamp" name=”timestamp" />
   <field type="service_id" name="service_id" />
   <field type="operation_id" name="operation_id" />
   <field type="module_id" name="module_id" />
   <field type="exception_id" name="exception_id" />
</record>
```

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

9.2.9.1 ECOA:service_id

An `ECOA:uint32` used to identify the service instance.

The ECOA:service_id is a simple type (see section 9.3.1) defined in the ECOA namespace as follows:

```
<simple type="uint32" name="service_id" />
```

9.2.9.2 ECOA:operation_id

An `ECOA:uint32` used to identify the operation.

The ECOA:operation_id is a simple type (see section 9.3.1) defined in the ECOA namespace as follows:

```
<simple type="uint32" name="operation_id" />
```

9.2.9.3 ECOA:module_id

An `ECOA:uint32` used to identify the module instance.

The ECOA:module_id is a simple type (see section 9.3.1) defined in the ECOA namespace as follows:

```
<simple type="uint32" name="module_id" />
```

9.2.9.4 ECOA:exception_id

The data type `ECOA:exception_id` is an enumeration declared in the ECOA namespace, which is used to specify a type of error. The enumeration values are:

ECOA:NO_RESPONSE = 1
ECOA:OPERATION_ABORTED = 2
ECOA:RESOURCE_NOT_AVAILABLE = 3
ECOA:OPERATION_NOT_INVOKED = 4
ECOA:ILLEGAL_INPUT_ARGS = 5
ECOA:ILLEGAL_OUTPUT_ARGS = 6
ECOA:MEMORY_VIOLATION = 7
ECOA:DIVISION_BY_ZERO = 8
ECOA:FLOATING_POINT_EXCEPTION = 9
ECOA:ILLEGAL_INSTRUCTION = 10
ECOA:STACK_OVERFLOW = 11
ECOA:HARDWARE_FAULT = 12
ECOA:POWER_FAIL = 13
ECOA:COMMUNICATION_ERROR = 14
ECOA:DEADLINE_VIOLATION = 15
ECOA:OVERFLOW_EXCEPTION = 16
ECOA:UNDERFLOW_EXCEPTION = 17
ECOA:OPERATION_OVERRATED = 18
ECOA:OPERATION_UNDERRATED = 19

The ECOA:exception_id is an enumeration (see section 9.3.3) defined in the ECOA namespace as follows:

```
<enum name="exception_id" type="uint32">
             <value name="NO_RESPONSE" valnum="1"/>
             <value name="OPERATION_ABORTED" valnum="2" />
             <value name="RESOURCE_NOT_AVAILABLE" valnum="3" />
             <value name="OPERATION_NOT_INVOKED" valnum="4" />
             <value name="ILLEGAL_INPUT_ARGS" valnum="5" />
             <value name="NO_RESPONSE" valnum="5" />
             <value name="OPERATION_ABORTED" valnum="6" />
             <value name="ILLEGAL_OUTPUT_ARGS" valnum="6" />
             <value name="MEMORY_VIOLATION" valnum="7" />
             <value name="DIVISION_BY_ZERO" valnum="8" />
             <value name="FLOATING_POINT_EXCEPTION" valnum="9" />
             <value name="ILLEGAL_INSTRUCTION" valnum="10" />
             <value name="STACK_OVERFLOW" valnum="11" />
             <value name="HARDWARE_FAULT" valnum="12" />
             <value name="POWER_FAIL" valnum="13" />
             <value name="COMMUNICATION_ERROR" valnum="14" />
             <value name="DEADLINE_VIOLATION" valnum="15" />
             <value name="OVERFLOW_EXCEPTION" valnum="16" />
```

```
            <value name="UNDERFLOW_EXCEPTION" valnum="17" />
            <value name="OPERATION_OVERRATED" valnum="18" />
            <value name="OPERATION_UNDERRATED" valnum="19" />
</enum>
```

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

A simple type is a refinement of a predefined type with a new name and optional additional restrictions (e.g. a more restrictive range). These restrictions can be expressed directly in strongly typed languages such as Ada, however in less strongly typed languages such a C/C++ they are expressed indirectly using min and max constants. A simple type can also be defined based upon another user defined simple type.

Example 1 – defining a simple type based on a predefined ECOA simple type:

```
<simple type="#ECOA_predefined_type_name#" name="#simple_type_name#" />
```

Example 2 – defining a type based upon a previously defined simple type:

```
<simple type="#simple_type_name#" name="#simple_type_name#" />
```

9.3.2 Constants

A constant is a defined constant value of a given, previously defined, type. A constant may be an integer or floating point. Constants can be referenced when defining other types; allowing a type to be sized or constrained.

```
<constant name="#constant_name#" type="#type_name#" value="#constant_value#" />
```

The #constant_value# may be an integer or floating-point value.

Example 1 - defining a constant of type ECOA:uint32:

```
<constant name="my_message_max_size" type="ECOA:uint32" value="1024" />
```

Example 2 – defining a constant of type ECOA:double64:

```
<constant name="Pi" type="ECOA:double64" value="3.141592654" />
```

Constants can be used with the XML notation by using the following syntax:

%constant_name%.

Example 3 using a constant to bound an array:

```
<array name="my_message" itemType="ECOA:char8" maxNumber="%my_message_max_size%" />
```

9.3.3 Enumerations

An enumeration type is the definition of a set of labels, derived from a pre-defined type, with optional values or integer-based constant definitions. Where the optional values are not defined they default to incrementing from zero.

All labels used in an enum shall be unique within the enum scope. The enum type shall be a pre-defined integer type, or a simple type derived from a pre-defined integer type.

Example – defining an enumeration type:

```
<enum name="#enum_type_name#" type="#type_name#">
```

```
    <value name="#enumeration_constant_name1#" valnum="#optional_enum_value_value1#"/>
    <value name="#enumeration_constant_name2#" valnum="#optional_enum_value_value2#"/>
    <value name="#enumeration_constant_name3#" valnum="%#optional_enum_constant_name#%"/>
</enum>;
```

Where: `#optional_enum_value_valueX#` is of type `#type_name#`.

9.3.4 Records

Records types are types containing a fixed set of fields of given types. All types used in a record shall be previously defined or ECOA pre-defined types.

All fields used in a record shall be unique within the record scope.

Example – defining a record type:

```
<record name="#record_type_name#">
    <field type="#type_name#" name="#record_field_name#" />
     <!— a record may consist of multiple <fields... /> -->
    [<field type="#type_name#" name="#record_field_name#" />]
</record>
```

9.3.5 Variant Records

Variant Record types

- may contain a fixed set of fields of given type
- shall contain a set of optional fields and a selector. The selector chooses the format of the record by controlling which optional fields are actually included in the record at runtime.

Variant records allow the definition of flexible data types: at runtime an instance of the variant record will contain any specified fixed fields plus a subset of the optional fields specified.

Example – defining a variant record:

```
<variantrecord name="#record_type_name#" selectName="#selector_name#" selectType="#type_name#">
    <field type="#type_name#" name="#record_field_name#" />
    <union type="#type_name#" name="#union_name#" when="#selector_value_constant#" />
     <!— a variantrecord may consist of multiple {<field... /><union... />} pairs... -->
    [<field type="#type_name#" name="#record_field_name#" />
     <union type="#type_name#" name="#union_name#" when="#selector_value_constant#" /> ]
</record>
```

9.3.6 Fixed Arrays

A fixed array is an ordered collection of a defined maximum number of elements of the same type. The value of maximum number shall be a positive constant of an integer type, and the array shall always contain this number of elements.

Example – defining a fixed array:

```
<fixedarray name="#array_type_name#" itemType="#type_name#" maxNumber="#int64_constant#" />
```

9.3.7 Variable Arrays

A variable array is an ordered collection of elements of the same type. The variable array has a “current size” and a “maximum size”. The “current size” enables the amount of data that needs to be copied to be minimised. The "maximum size" bounds the memory and data transfer requirements. Variable arrays of char8 shall be used to store character strings.

The values of “maximum size” and “current size” shall be positive and “current size” shall be less than or equal to “maximum size”.

Example – defining a variable array:

```
<array name="#array_type_name#" itemType="#type_name#" maxNumber="#int64_constant#" />
```

10 Module Interface

The Module Interface specifies the interface to a module, which is used by the container to call module operations.

10.1 Operations

The Module Interface provides a number of entry points that allow the Container to invoke Module Operations that cause a Module Instance to execute a block of functionality.

10.1.1 Request-Response

For modules which are declared as a server of a request response operation, two operation types exist:

- Immediate Response, which is analogous to a Synchronous Request
- Deferred Response, which is analogous to an Asynchronous Request

Immediate Response is the default behaviour.

For modules which are declared as a client of an asynchronous request response operation, Response_Received is provided to return the result of an asynchronous request.

10.1.1.1 Request Received Immediate Response

For a Module declared as server of a request-response operation with immediate response, a function is implemented by the Module to handle the request generated by the client Module. The declaration carries the input and output parameters. The name of the function shall be generated to include the name of the operation.

The appropriate language binding will define the correct syntax for this module operation, but the abstract format is given below:

```
void
[#module_implementation_name#:]#operation_name#__Request_Received([#context#,]#parameters_in#,
#parameters_out#);
```

10.1.1.2 Request Received Deferred Response

For a Module declared as server of a request-response operation with deferred response, a function is implemented by the Module to handle the request generated by the client Module. The declaration carries the input parameters. The name of the function shall be generated to include the name of the operation.

Request_Received_Deferred provides an ID parameter which is required by the infrastructure to associate the reply with the initiating request.

The appropriate language binding will define the correct syntax for this module operation, but the abstract format is given below:

```
void [#module_implementation_name#:]#operation_name#__Request_Received_Deferred([#context#,]
ECOA:uint32 ID, #parameters_in#);
```

10.1.1.3 Response Received

For a Module declared as client of an asynchronous request-response operation, a function is implemented by the Module to handle the response generated by the server Module. The declaration carries the input and output parameters. The name of the function shall be generated to include the name of the operation.

Response_Received provides an ID parameter which is used by the module instance to associate the response with the initiating request. This is required because the module could initiate multiple requests prior to receiving any responses.

The appropriate language binding will define the correct syntax for this module operation, but the abstract format is given below:

```
void    [#module_implementation_name#:]#operation_name#__Response_Received([#context#,],ECOA:uint32
ID, ECOA:error status, #parameters_out#);
```

Response Received operations may return the following error codes:

[ECOA:error:OK]

- No error

[ECOA:error:NO_RESPONSE]

- No response received within the expected time

[ECOA:error:OPERATION_NOT_AVAILABLE]

- Required service is not available
- The server module queue is full
- Server module is IDLE/STOPPED

[ECOA:error:OPERATION_ABORTED]

- Server has called raise_*_error()

10.1.2 Versioned Data

The Module Interface provides an optional entry point that is used to notify a Module when a new value of Versioned data is available.

10.1.2.1 Updated

The Updated module operation is a callback used by the Container to notify a module when a new value of Versioned data is available. The Module is provided with direct access to the data; the Container automatically calls Get_Read_Access and Release_Read_Access at the start and end of the operation respectively. This entry point is used to avoid the use of polling to identify when new values are available.

Note: The default behaviour of versioned data read operations is no notification callback.

The following is a prototype definition for the operation:

```
[#module_impl_name#:]#operation_name#__Updated([#context#,]
[#module_impl_name#:]#operation_name#_handle* );
```

10.1.3 Events

The Module Interface provides an entry point for the receipt of Events.

10.1.3.1 Received

For a Module declared as a handler of an event, a function, method or procedure shall be implemented by the Module to handle the reception of the event from all possible senders. Its input parameters shall correspond to the typed data carried by the event. The name of the function shall be generated to include the name of the operation.

The appropriate language binding will define the correct syntax for this module operation, but the abstract format is given below:

```
void [#module_implementation_name#:]#operation_name#__Received([#context#,]#parameters#);
```

10.2 Component Lifecycle

The Component Lifecycle provides functionality to allow a supervision module within a component to manage its lifecycle state. It is the responsibility of the supervision module to

determine the state of the component based upon the states of its internal non-supervision modules and any lifecycle operations invoked by manager components.

The component and module lifecycles are discussed more fully in reference 7.

The service to manage the lifecycle of a component from another component is composed of:

- The current state of the component (versioned data)
- A command to change the state (event):
  - **INITIALIZE** - order the supervision module(s) to change the state of the appropriate modules to READY in order to reach the component-level STOPPED state,
  - **STOP** - order the supervision module(s) to change the state of the appropriate modules to READY in order to reach the component-level STOPPED state,
  - **START** - order the supervision module(s) to change the state of the appropriate modules to RUNNING in order to reach the component-level RUNNING state,
  - **RESTART** - order the supervision module(s) to change the state of the appropriate modules from RUNNING through READY and back to RUNNING in order to reach the component-level RUNNING state,
  - **RESET -** order the supervision module(s) to change the state of the appropriate modules from RUNNING through IDLE, READY and back to RUNNING in order to reach the component-level RUNNING state,
  - **SHUTDOWN** - order the supervision module(s) to change the state of all modules to IDLE in order to reach the component-level IDLE state.
- A notification (event) on state change (new component-level lifecycle state reached):
  - **initialized** – STOPPED reached from INITIALIZING
  - **started** – RUNNING reached
  - **stopped** – STOPPED reached from STOPPING
  - **idle** – IDLE reached
  - **failed** –FAILURE reached

The XML definition of supervision module operations that are connected to the lifecycle service is shown below:

```
<moduleType name="SupervisionModule_type">
  <operations>
    <eventReceived name="start_component"/>
    <eventReceived name="stop_component "/>
    <eventReceived name="restart_component "/>
    <eventReceived name="reset_component "/>
    <eventReceived name="shutdown_component "/>
    <eventReceived name="initialize_component "/>
    <eventSent name=" component_initialized"/>
    <eventSent name=" component_started"/>
    <eventSent name=" component_stopped"/>
    <eventSent name=" component_idle"/>
    <eventSent name=" component_failed"/>
    <dataWritten name="component_state" type="component_states_type" />
  </operations>
</moduleType>
```

10.2.1 Supervision Module Component Lifecycle API

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

10.2.1.1 Initialize Component

This is an event received to instruct the component to initialize. The event carries no parameters.

The appropriate language binding will define the correct syntax for this module operation, but the abstract format is given below:

```
void [#supervision_module_implementation_name#:]initialize_component__Received([#context#]);
```

10.2.1.2 Stop Component

This is an event received to instruct the component to stop. The event carries no parameters.

The appropriate language binding will define the correct syntax for this module operation, but the abstract format is given below:

```
void [#supervision_module_implementation_name#:]stop_component__Received([#context#]);
```

10.2.1.3 Restart Component

This is an event received to instruct the component to restart. The event carries no parameters.

The appropriate language binding will define the correct syntax for this module operation, but the abstract format is given below:

```
void [#supervision_module_implementation_name#:]restart_component__Received([#context#]);
```

10.2.1.4 Reset Component

This is an event received to instruct the component to reset. The event carries no parameters.

The appropriate language binding will define the correct syntax for this module operation, but the abstract format is given below:

```
void [#supervision_module_implementation_name#:]reset_component__Received([#context#]);
```

10.2.1.5 Shutdown Component

This is an event received to instruct the component to shut down. The event carries no parameters.

The appropriate language binding will define the correct syntax for this module operation, but the abstract format is given below:

```
void [#supervision_module_implementation_name#:]shutdown_component__Received([#context#]);
```

10.2.1.6 Start Component

This is an event received to instruct the component to start. The event carries no parameters.

The appropriate language binding will define the correct syntax for this module operation, but the abstract format is given below:

```
void [#supervision_module_implementation_name#:]start_component__Received([#context#]);
```

10.3 Module Lifecycle

The Module Interface provides functionality to allow the container to command changes to the lifecycle state of the Module Instances it hosts under the direction of a Supervision Module. The lifecycle of Modules is controlled by one or more Supervision Modules. Each Component must contain a Supervision Module, which may be the only Module hosted by the container i.e. it may also provide the functionality required to provide the Component's Services. Any Supervision Module is initialised and started automatically by the container.

The component and module lifecycles are discussed more fully in reference 7.

10.3.1 Generic Module API

Functionality is provided by the Module Interface to support the following Module Lifecycle functionality. These operations are applicable to all supervision, non-supervision, trigger and dynamic trigger module instances.

- **INITIALIZE_Received**: this is the initialisation entry-point of the module used to perform its local initialisation; the Initialise entry-point of a Module is the function in which the Module is supposed to initialise all its local variables to be functionally initialised. This event is sent to the Module when it has changed state from IDLE to READY.
- **REINITIALIZE_Received**: this is the reinitialisation entry-point of the module used to perform a subset of what is done by **INITIALIZE_Received**, mainly to initialise its local variables. This event is sent to the Module when it has changed state from RUNNING or READY back to READY.
- **START_Received**: this event is sent to the Module when it has changed state from READY to RUNNING
- **STOP_Received**: this event is sent to the Module when it has changed state from RUNNING to READY
- **SHUTDOWN_Received**: this event is sent to the Module when it has changed state from READY or RUNNING to IDLE

At API level, the following abstract functions will be invoked by the container and shall be implemented by the Module.

```
void [#module_implementation_name#:]INTIALIZE__Received([#context#]);
void [#module_implementation_name#:]START__Received([#context#]);
void [#module_implementation_name#:]STOP__Received([#context#]);
void [#module_implementation_name#:]SHUTDOWN__Received([#context#]);
void [#module_implementation_name#:]REINTIALIZE__Received([#context#]);
```

Within these five functions the module is restricted such that it may not call any Request Response container operation API (i.e. Request_Sync, Request_Async or Reply_Deferred). This is to prevent race conditions and deadlock due to the start-up order of modules.

10.3.2 Supervision Module Lifecycle API

The Supervision Module API provides functionality to allow the container to notify the supervision module that a module/trigger/dynamic trigger has changed state. The notification informs the Supervision Module of both the previous and new states of the Module.

The following is a prototype definition for the operation:

```
void                [#supervision_module_impl_name#:]lifecycle_notification__#module_instance_name#
([#context#,]ECOA:module_states_type previous_state, ECOA:module_states_type new_state);
```

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

10.4 Service Availability

The Module Interface provides functionality to allow the container to notify the supervision module of a client component about changes to the availability of required services.

10.4.1 Service Availability Changed

Supervision modules shall provide an entry point for the receipt of a notification that a required service has changed its availability state. The operation will only be available if the component has one or more

required services. The service instance is identified by the enumeration type reference_id defined in the Container Interface (Section 11.5.4)

The appropriate language binding will define the correct syntax for this module operation, but the abstract format is given below:

```
void          [#supervision_module_implementation_name#:]service_availability_changed([#context#,]
[#supervision_module_implementation_name#_container:]reference_id      instance,      ECOA:boolean8
available);
```

10.4.2 Service Provider Changed

Supervision modules shall provide an entry point for the receipt of a notification that a required service has changed provider. The operation will only be available if the component has one or more required services. The service instance is identified by the enumeration type reference_id defined in the Container Interface (Section 11.5.4)

The appropriate language binding will define the correct syntax for this module operation, but the abstract format is given below:

```
void              [#supervision_module_implementation_name#:]service_provider_changed([#context#,]
[#supervision_module_implementation_name#_container:]reference_id instance);
```

10.5 Error handling

The Supervision Module Interface provides error handling functionality that may be used by the container to provide information to a Supervision Module Instance when an error occurs.

The following is an abstract description of the operation:

```
void [#supervision_module_implementation_name#:]exception_notification_handler([#context#], const
ECOA:exception* exception);
```

This exception notification handler can be called when an asynchronous error occurs at container level (e.g. the container internal buffers are full) or at hardware level (e.g. a divide by zero exception) based on predefined actions specified at configuration time.

Note: the error handling functionality that is provided by an ECOA system has not been fully defined: the above is a preliminary outline of the functionality that may be provided and is not complete.

11 Container Interface

11.1 Operations

The Container Interface provides a number of operations that allow a module to invoke Container Operations to request Services from other Modules in the system.

11.1.1 Request Response

Two operations are provided to allow Modules to issue requests to other modules:

- Synchronous Request, which is analogous to an Immediate Response
- Asynchronous Request, which is analogous to a Deferred Response

A further operation, Reply_Deferred is provided to return the result of a deferred response to the requesting module.

11.1.1.1 Synchronous Request

An operation provided by the Container, used by a Module to invoke an operation provided by a server Module. Its parameters correspond to the “in” and “out” parameters of the request-response. The calling Module is blocked until the response is received.

An error indication is returned to caller if the call fails and the fault is then handled via the fault management infrastructure.

The following is a prototype definition for the operation:

```
ECOA:error
[#module_implementation_name#_container:]#operation_name#__Request_Sync([#context#,]#parameters_in
#, #parameters_out#);
```

Synchronous Request operations may return the following error codes:

[ECOA:error:OK]

- No error

[ECOA:error:NO_RESPONSE]

- No response received within the expected time

[ECOA:error:OPERATION_NOT_AVAILABLE]

- Required service is not available
- The server module queue is full
- Server module is IDLE/STOPPED

[ECOA:error:OPERATION_ABORTED]

- Server has called raise_*_error()

[ECOA:error:RESOURCE_NOT_AVAILABLE]

- Container unable to send request

11.1.1.2 Asynchronous Request

An operation provided by the Container, used by a Module to invoke an operation provided by a server Module. The first parameter is an ID, which is provided by the infrastructure to allow the module instance to associate the response with the request. This ID is unique for each module

instance and for each call of the operation (because the module could initiate multiple requests prior to receiving any responses). The remaining parameters correspond to the “in” parameters of the request-response.

The operation returns immediately so the calling Module is not blocked. An error message is returned to the caller if an infrastructure problem prevents the call from succeeding. The fault is then handled via the fault management infrastructure.

The following is a prototype definition for the operation:

```
ECOA:error
[#module_implementation_name#_container:]#operation_name#__Request_Async([#context#,],ECOA:uint32*
ID, #parameters_in#);
```

Asynchronous Request operations may return the following error codes:

[ECOA:error:OK]

- No error

[ECOA:error:RESOURCE_NOT_AVAILABLE]

- Container unable to send request

11.1.1.3 Reply Deferred

An operation provided by the Container, used by the Module to send a Deferred Response. The first parameter is an ID, which is provided by the infrastructure to identify the calling Module Instance and to allow that module to associate the response with the request. The remaining parameters correspond to the “out” parameters of the request-response.

An error indication is returned if an infrastructure problem prevents the API from succeeding, and the fault is handled via the fault management infrastructure.

The following is a prototype definition for the operation:

```
ECOA:error [#module_implementation_name#_container:]#operation_name#__Reply_Deferred([#context#,]
ECOA:uint32 ID, #parameters_out#);
```

Deferred Reply operations may return the following error codes:

[ECOA:error:OK]

- No error

[ECOA:error:INVALID_IDENTIFIER]

- API called with an invalid request-response identifier

11.1.2 Versioned Data

The container provides operations that allow Modules to read from or write to Versioned Data. The operations provided 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)

A Data Handle is provided by the container for each instance of Versioned data to allow Module Instances to access that Versioned Data.

A Data Handle structure contains the following fields:

- An attribute used to provide access to a local copy of the data
- A timestamp structure, which reflects the last 'commit' time for that version of the data
- A platform hook, which is opaque to the user, and used by the ECOA infrastructure to handle that data

The platform hook is typed as an array of bytes, to enable portability, to allow the infrastructure to allocate memory areas in order to store data handles. It is assumed that a size of 32 bytes is sufficient to cover any platform implementation.

The following is a prototype definition for a Data Handle

```
typedef struct {
   #type_name#* data;
   ECOA:timestamp timestamp;
   ECOA:byte platform_hook[32];
} [#module_impl_name#_container:]#operation_name#_handle;
```

11.1.2.1 Get_Read_Access

For a Module declared as a reader of a Versioned Data, the container shall provide a function to get read access to the Versioned Data. This operation shall output the Data Handle parameter that allows the subsequent code to access the data space containing a local, read-only copy of the data. The name of the function shall be generated to include the name of the operation.

The operation does not block and returns immediately with the latest available copy of data. The timestamp in the data handle enables the caller to determine the currency of the data. The error code ECOA:NO_DATA is returned if no data is available and the Data Handle contains a null pointer.

If there is an infrastructure problem that prevents the API from succeeding, an error indication is returned to the caller and the fault is handled via the fault management infrastructure. If an error is returned from Get_Read_Access, the call to Release_Read_Access is not required.

The following is a prototype definition for the operation:

```
ECOA:error [#module_impl_name#_container:]#operation_name#__Get_Read_Access([#context#,]
[#module_impl_name#_container:]#operation_name#_handle* data_handle);
```

Get Read Access operations may return the following error codes:

[ECOA:error:OK]

- No error

[ECOA:error:NO_DATA]

- No error – the data has never been written

[ECOA:error:INVALID_HANDLE]

- API called with an invalid versioned data handle

[ECOA:error:RESOURCE_NOT_AVAILABLE]

- Maximum number of versioned data reached
- Container unable to provide a versioned data

11.1.2.2 Release_Read_Access

This operation signals to the container that the calling module has finished working with the local copy of the Versioned Data, and that the data handle is no longer required. The module should not access the local copy of the data after calling this operation as it cannot be guaranteed to be consistent.

The following is a prototype definition for the operation:

```
ECOA:error [#module_impl_name#_container:]#operation_name#__Release_Read_Access([#context#,]
[#module_impl_name#_container:]#operation_name#_handle* data_handle);
```

Release Read Access operations may return the following error codes:

[ECOA:error:OK]

- No error

[ECOA:error:INVALID_HANDLE]

- API called with an invalid versioned data handle

11.1.2.3 Get_Write_Access

For a Module declared as a writer of a Versioned Data, the container shall provide a function to get write access to the versioned data. This operation shall output the Data Handle parameter that allows the subsequent code to access the data space containing a local, read-write copy of the data.

The operation does not block and returns immediately with the latest copy of the data. Each call to Get_Write_Access will use a new dedicated platform resource represented by the returned data handle and pointing to a new memory area with the most updated value. Hence, each call to Get_Write_Access will require a call to either Cancel_Write_Access or Publish_Write_Access to free that corresponding platform resources, and commit (publish) the modified data is required.

If data has never been written, Get_Write_Access returns the error code ECOA:DATA_NOT_INITIALISED but returns a valid data handle towards a valid memory area.

If there is an infrastructure problem that prevents the API from succeeding, another error indication (RESOURCE_NOT_AVAILABLE, etc) is returned to the caller and the infrastructure handles the fault via the fault management infrastructure. In the present issue the behaviour of the fault management infrastructure is not defined.

Obtains a handle that allows access to a copy of the data. Get_Write_Access does not block and returns immediately. If there is an infrastructure problem that prevents the API from succeeding, an error indication is returned to caller and the fault is handled via the fault management infrastructure. If an error is returned from Get_Write_Access, the call to Cancel_Write_Access is not required.

The following is a prototype definition for the operation:

```
ECOA:error [#module_impl_name#_container:]#operation_name#__Get_Write_Access([#context#,]
[#module_impl_name#_container:]#operation_name#_handle* data_handle);
```

Get Write Access operations may return the following error codes:

[ECOA:error:OK]

- No error

[ECOA:error:DATA_NOT_INITIALIZED]

- No error – the data has never been written

[ECOA:error:INVALID_HANDLE]

- API called with an invalid versioned data handle

[ECOA:error:RESOURCE_NOT_AVAILABLE]

- Maximum number of versioned data reached
- Container unable to provide versioned data

11.1.2.4 Cancel_Write_Access

This operation signals to the container that the calling module has finished working with the local copy of the Versioned Data, that no updates are required, and that the data handle is no longer required. Any local updates which may have been made should not be published to any readers of that versioned data. The module should not access the local copy of the data after calling this operation as it cannot be guaranteed to be consistent.

The following is a prototype definition for the operation:

```
ECOA:error [#module_impl_name#_container:]#operation_name#__Cancel_Write_Access([#context#,]
[#module_impl_name#_container:]#operation_name#_handle* data_handle);
```

Cancel Write Access operations may return the following error codes:

[ECOA:error:OK]

- No error

[ECOA:error:INVALID_HANDLE]

- API called with an invalid versioned data handle

11.1.2.5 Publish_Write_Access

This operation signals to the container that the calling module has finished working with the local copy of the Versioned Data and that the container is authorised to broadcast the revised data to all readers of the Versioned Data. The module should not access the local copy of the data after calling this operation as it cannot be guaranteed to be consistent.

The operation does not block. An error message is returned to the caller if the call is unsuccessful (e.g. a queue within the container is full), and the fault is handled by the infrastructure.

The following is an abstract definition of the operation:

```
ECOA:error [#module_impl_name#_container:]#operation_name#__Publish_Write_Access([#context#,]
[#module_impl_name#_container:]#operation_name#_handle* data_handle);
```

Publish Write Access operations may return the following error codes:

[ECOA:error:OK]

- No error

[ECOA:error:INVALID_HANDLE]

- API called with an invalid versioned data handle

[ECOA:error:RESOURCE_NOT_AVAILABLE]

- Container unable to write the versioned data
- Container unable to “push” the versioned data

11.1.3 Events

The Container Interface provides an operation that allows a Module to send Events.

11.1.3.1 Send

For a Module declared as a sender of an event, a function, method or procedure shall be implemented by the Container to send that event with typed parameters to all receivers. The name of the function shall be generated to include the name of the operation.

The operation returns immediately so the calling Module is not blocked. An error message is returned to the caller if an infrastructure problem prevents the call from succeeding (e.g. if erroneous parameters are given). The fault is then handled via the fault management infrastructure.

The following is a prototype definition for the operation:

```
ECOA:error
[#module_implementation_name#_container:]#operation_name#__Send([#context#,]#parameters#);
```

Send operations may return the following error codes:

`[ECOA:error:OK]`

- `No error`

`[ECOA:error:RESOURCE_NOT_AVAILABLE]`

- `Container unable to send` ***`any`*** `event`

11.2 Properties

The Container Interface API may include operations that can be used by the Modules to access component properties defined at the component level. These properties are defined within the component definition, assigned a value within the system assembly, and may then be mapped into module instances within the component implementation. It is also possible to provide module properties within the component implementation that are not specified at the component level. This allows for different instances of modules to have access to specific properties defined at both the module and component instance level.

11.2.1 Get_Value

Used by Module Instances to get read only access to the properties The abstract format of the message is:

```
void [#module_impl_name#_container:]get_#property_name#_value([#context#,] #property_type_name#*
value);
```

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.

11.2.2 Expressing Property Values

Values given to properties are set in component definitions, component implementations or in assembly schemas through the writing of character strings. This section describes the way to write these strings. It is based on a syntax allowing simple or complex data to be represented.

- « Predef », « Simple » : direct value
  - Examples: 16, 0xFFFFFFFF, -10, 100.234, true (=1), false (=0)
- « Enum » : symbol
  - The case shall follow the one used in the XML type definition.
  - Examples: AIR, GROUND, etc.
- « Record » : list of field names and values, separated by ",", surrounded with curly braces
  - The case shall follow the one used in the XML type definition.
  - {isValid: true,x: 10,y: 100.0, mode:GROUND}
- « VariantRecord » : same as "record", except that first field is always the discriminant (name of this field = "select")

- Some fields may not exist, depending on the value of the discriminant.
- Examples:
  - {select: AIR, position3d: {x:2,y:3,z:4}}
  - {select: GROUND, position2d: {x:5,y:6}}

- "FixedArray » : list of « maxNumber » values, comma separated, surrounded by []
  - Example: [1,2,3,4,5]
  - Special case if element type is char8: string syntax with surrounded ""
    - Equivalent to an array of int with values of ASCII codes
    - The number of characters must be equal to maxNumber.
    - Escape character for "" is '\'.
    - Example (for a fixedarray with maxNumber=5): "ABCDE"
- « Array » : list of N values, comma separated, surrounded by [] (with 0<N<maxNumber):
  - Examples: [] (empty array), [100.5, 329.3, -456.99]
  - Special case if element type is char8: string syntax with ""
    - Example (for maxNumber=7): "ABCDE" = [0x41, 0x42, 0x43, 0x44, 0x45]
- Notion of « multiplier » to repeat an element in an array : #N:element
  - Examples:
    - [#10:0] (10 times the value 0)
    - a record repeated 10 times: [#20:{isValid:true, x:100.0, y:-10, mode:AIR}]
    - repeat until the end of the array: [1,2,3,#*:999]
    - 10*10 matrix with zeroes: [#10:[#10:0]]
- Support for constants
  - Suppose the following is defined in the library "mylib":
    - <constant name="MY_CONST" type="int32" value="32"/>
  - Then the expression "%mylib:MY_CONSTANT%" is allowed in properties values:
  - This is only valid for integer and floating-point types only.
- Character syntax
  - For type char8, the expression '0x' is allowed in properties values to represent characters by their ASCII codes. By example, 'A' can also be written as the ASCII code 0x41.

11.2.3 Example of Defining and Using Properties

The following XML defines a component with a simple property "Update_Rate" (example.componentType):

```
<componentType>
  <service .../>
  <reference .../>
  <property name="Update_Rate" type="xs:string" ECOA-sca:type="float32"/>
</componentType>
```

The following XML defines how the module type and instance defines how the property is mapped. Also a property local to the module instance is defined (Module_Inst_Prop), which allows the module instance to have different values:

```
<moduleType name="example_mod_type" isSupervisionModule="false">
      <properties>
```

```
            <property name="Update_Rate" type="float32"/>
            <property name="Module_Inst_Prop" type="uint32"/>
      </properties>

</moduleType>

...

<moduleInstance name="example_mod_inst1"
                    moduleDeadline="245"
                    implementationName="example_mod_impl">
                    <propertyValues>
                     <propertyValue name="Update_Rate">$Update_Rate</propertyValue>
                     <propertyValue name="Module_Inst_Prop"> 20 </propertyValue>
                    </propertyValues>
</moduleInstance>

<moduleInstance name="example_mod_inst2"
                    moduleDeadline="245"
                    implementationName="example_mod_impl">
                    <propertyValues>
                     <propertyValue name="Update_Rate">$Update_Rate</propertyValue>
                     <propertyValue name="Module_Inst_Prop"> 2 </propertyValue>
                    </propertyValues>
</moduleInstance>
```

Values are assigned to component properties in the system assembly schema (.impl.composite):

```
<csa:component name="example">
  <ECOA-sca:instance componentType="example_instance">
    <ECOA-sca:implementation name="example_component"/>
  </ECOA-sca:instance>
    <csa:property name="Update_Rate"><csa:value>10.0</csa:value></csa:property>
</csa:property>
</csa:component>
```

The above example would generate two Get_Value APIs:

```
void [example_mod_impl_container:]get_Update_Rate_value([#context#,] ECOA:float32* value);

void [example_mod_impl_container:]get_Module_Inst_Prop_value([#context#,] ECOA:uint32* value);
```

For the component instance “example_instance” the get_Update_Rate_value API would return 10.0 for both the “example_mod_inst1” and “example_mod_inst2” module instances. However the get_Module_Inst_Prop_value API would return 20 for the “example_mod_inst1” module instance, but 2 for the “example_mod_inst2” module instance.

11.3 Component Lifecycle

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

The component and module lifecycles are discussed more fully in reference 7.

11.3.1 Supervision Module Component Lifecycle API

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

11.3.1.1 Component Initialized

This is an event sent to indicate that the component has initialized. The event carries no parameters.

The appropriate language binding will define the correct syntax for this container operation, but the abstract format is given below:

```
ECOA:error
[#supervision_module_implementation_name#_container:]component_initialized__Send([#context#]);
```

11.3.1.2 Component Started

This is an event sent to indicate that the component has started. The event carries no parameters.

The appropriate language binding will define the correct syntax for this container operation, but the abstract format is given below:

```
ECOA:error
[#supervision_module_implementation_name#_container:]component_started__Send([#context#]);
```

11.3.1.3 Component Stopped

This is an event sent to indicate that the component has stopped. The event carries no parameters.

The appropriate language binding will define the correct syntax for this container operation, but the abstract format is given below:

```
ECOA:error
[#supervision_module_implementation_name#_container:]component_stopped__Send([#context#]);
```

11.3.1.4 Component Idle

This is an event sent to indicate that the component has gone idle. The event carries no parameters.

The appropriate language binding will define the correct syntax for this container operation, but the abstract format is given below:

```
ECOA:error [#supervision_module_implementation_name#_container:]component_idle__Send([#context#]);
```

11.3.1.5 Component Failed

This is an event sent to indicate that the component has failed. The event carries no parameters.

The appropriate language binding will define the correct syntax for this container operation, but the abstract format is given below:

```
ECOA:error
[#supervision_module_implementation_name#_container:]component_failed__Send([#context#]);
```

11.3.1.6 Component State

This is versioned data operation published to indicate the current component state.

The appropriate language binding will define the correct syntax for the component state, but the abstract format is given below:

```
#define ECOA_VERSIONED_DATA_HANDLE_PRIVATE_SIZE 32
typedef struct {
   ECOA:component_states_type* data;
   ECOA:timestamp timestamp;
   ECOA:byte platform_hook[ECOA_VERSIONED_DATA_HANDLE_PRIVATE_SIZE];
} [#supervision_module_impl_name#_container:]component_state_handle;
```

The appropriate language binding will define the correct syntax for the container operations, but the abstract formats are given below:

```
ECOA:error [#supervision_module_impl_name#_container:]state__Get_Read_Access([#context#,]
[#supervision_module_impl_name#_container:]component_state_handle* data_handle);

ECOA:error [#supervision_module_impl_name#_container:]state__Release_Read_Access([#context#,]
[#supervision_module_impl_name#_container:]component_state_handle* data_handle);

ECOA:error [#supervision_module_impl_name#_container:]state__Get_Write_Access([#context#,]
[#supervision_module_impl_name#_container:]component_state_handle* data_handle);

ECOA:error [#supervision_module_impl_name#_container:]state__Cancel_Write_Access([#context#,]
[#supervision_module_impl_name#_container:]component_state_handle* data_handle);
```

```
ECOA:error [#supervision_module_impl_name#_container:]state__Publish_Write_Access([#context#,]
[#supervision_module_impl_name#_container:]component_state_handle* data_handle);
```

The container will also provide an operation to allow the component to publish its state and send the appropriate event using one call.

The appropriate language binding will define the correct syntax, but the abstract format is given below:

```
ECOA:error [#supervision_module_impl_name#_container:]set_component_state([#context#,]
ECOA:component_states_type state);
```

The event sent by the set_component_state container operation is given in the table below (invalid transitions have been greyed out):

		New State							
		IDLE	INITIALIZING	STOPPED	STOPPING	RUNNING	STARTING	FINISHING	FAILURE
Previous State	-	Evt component_idle VD IDLE							Evt component_failed VD FAILURE
	IDLE	VD IDLE	*No event VD INITIALIZING*						Evt component_failed VD FAILURE
	INITIALIZING		VD INITIALIZING	Evt component_initialized VD STOPPED					Evt component_failed VD FAILURE
	STOPPED		*No event VD INITIALIZING*	VD STOPPED			*No event VD STARTING*	*No event VD FINISHING*	Evt component_failed VD FAILURE
	STOPPING			Evt component_stopped VD STOPPED	VD STOPPING				Evt component_failed VD FAILURE
	RUNNING		*No event VD INITIALIZING*		*No event VD STOPPING*	VD RUNNING		*No event VD FINISHING*	Evt component_failed VD FAILURE
	STARTING					Evt component_started VD RUNNING	VD STARTING		Evt component_failed VD FAILURE
	FINISHING	Evt component_idle VD IDLE						VD FINISHING	Evt component_failed VD FAILURE
	FAILURE								VD FAILURE

The Set Component State operation may return the following error codes:

[ECOA:error:OK]

- No error

[ECOA:error:INVALID_STATE]

- Operation called with invalid state
- State transition not allowed by component lifecycle state automata

11.4 Module Lifecycle

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

The component and module lifecycles are discussed more fully in reference 7.

11.4.1 Generic Module API

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

11.4.2 Supervision Module API

The Container Interface provides functionality to allow the supervision module to command changes to the lifecycle states of other module/trigger/dynamic trigger instances.

An instance of the following operations is provided for each non-supervision module, trigger and dynamic trigger hosted by the container controlled by that Supervision Module:

- Get_lifecycle_state: request the current state of a module/trigger/dynamic trigger

- STOP: request the module/trigger/dynamic trigger to stop
- START: request the module/trigger/dynamic trigger to start
- INITIALISE: request the module/trigger/dynamic trigger to initialise
- SHUTDOWN: request the module/trigger/dynamic trigger to shutdown

The appropriate language binding will define the correct syntax for these container operations, but the abstract format is given below:

```
void
[#supervision_module_implementation_name#_container:]get_lifecycle_state__#module_instance_name#([
#context#,] ECOA:module_states_type* current_state);

ECOA:error
[#supervision_module_implementation_name#_container:]INTIALIZE__#module_instance_name#([#context#]
);

ECOA:error [#supervision_module_implementation_name#_container
:]START__#module_instance_name#([#context#]);

ECOA:error [#supervision_module_implementation_name#_container
:]STOP__#module_instance_name#([#context#]);

ECOA:error [#supervision_module_implementation_name#_container
:]SHUTDOWN__#module_instance_name#([#context#]);
```

The INITIALIZE, START, STOP and SHUTDOWN operations may return the following error codes:

[ECOA:error:OK]

- No error

[ECOA:error:INVALID_TRANSITION]

- State transition not allowed by module lifecycle state automata

11.5 Service Availability

The Container Interface provides functionality to allow a supervision module to set the availability of its provided services and get the availability of its required services.

11.5.1 Set Service Availability (Server Side)

An operation is provided to allow a supervision module to set the availability state of its provided services. The operation will only be available if the component has one or more provided services. The service instance is identified by the enumeration type service_id defined in the Container Interface (Section 11.5.3)

The following is a prototype definition for the operation:

```
ECOA:error
[#supervision_module_implementation_name#_container:]set_service_availability([#context#,]
[#supervision_module_implementation_name#_container:]service_id instance, ECOA:boolean8
available);
```

The Set Service Availability operation may return the following error codes:

[ECOA:error:OK]

- No error

[ECOA:error:INVALID_SERVICE_ID]

- Operation called with invalid service ID

11.5.2 Get Service Availability (Client Side)

An operation is provided to allow a supervision module to get the availability state of its required services. The operation will only be available if the component has one or more required services. The service instance is identified by the enumeration type reference_id defined in the Container Interface (Section 11.5.4)

The following is a prototype definition for the operation:

```
ECOA:error
[#supervision_module_implementation_name#_container:]get_service_availability([#context#,]
[#supervision_module_implementation_name#_container:]reference_id instance, ECOA:boolean8*
available);
```

The Get Service Availability operation may return the following error codes:

[ECOA:error:OK]

- No error

[ECOA:error:INVALID_SERVICE_ID]

- Operation called with invalid reference ID

11.5.3 Service ID Enumeration

service_id is an enumeration type which identifies one of the service instances provided by the component.

The abstract enumeration type name is the following:

```
#supervision_module_implementation_name#_container:]service_id.
```

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

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

11.5.4 Reference ID Enumeration

reference_id is an enumeration type which identifies one of the service instances required by the component.

The abstract enumeration type name is the following:

```
#supervision_module_implementation_name#_container:]reference_id.
```

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

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

11.6 Logging and Fault Management

The Container Interface provides dedicated functionality for each Module Instance to provide information to the infrastructure. This information may be logged and falls into two categories:

- **Faults** for which the infrastructure is able to provide run-time responses
- **Execution Information** that can aid offline analysis of problems for system development and integration

Six categories of information can be recorded: two categories for faults and four categories relating to execution information as shown in Table 5.

Category	Definition	Infrastructure Response	Maskable Within the Deployment Schema
FATAL	Used by the application to raise severe errors from which it knows it cannot recover. No filtering is useful or desirable.	Module shall be shutdown by the infrastructure and fault is reported to the fault management infrastructure. Information is logged.	No
ERROR	Used by the application to raise errors from which the application may be able to recover, with assistance.	The fault management infrastructure shall filter these errors to determine whether the Module is to be shutdown or not. Information is logged.	No
WARNING	Used by the application to log runtime situations that are undesirable or unexpected, but not necessarily "wrong". Useful for non-intrusive analysis. The current module instance is not stopped and continues execution.	Information is logged.	Yes
INFO	Used by the application to log interesting runtime events (eg. startup/shutdown). Useful for non-intrusive analysis. The current module instance is not stopped and continues execution.	Information is logged.	Yes
DEBUG	Detailed information on the flow through the system.	Information is logged.	Yes
TRACE	More detailed information.	Information is logged.	Yes

Table 5 – Logging Error Level

An entry-point in the Container Interface is associated with each of the categories in Table 5. If necessary the container shall truncate to the data to the maximum size of `ECOA::log`. An output log is associated with each Module Instance. The logs are configured in the Deployment where:

- The output destination file name for each module instance is given, (where applicable)
- Certain categories may be masked, except for ERROR and FATAL.

Logging configuration information defined at Component Instance level is applied for any Module Instances where no configuration information is present in the Deployment. The following defaults are applied where no configuration information is given at the Component Instance level:

- A dedicated logging directory is associated with the Component Instance within a specified Logging device (where applicable)
- All logging levels are enabled.

The actual log output configuration is platform dependant, and may be assigned to text files; flash disk etc. depending on the platform capabilities.

The following are prototype definitions for the logging and fault operations:

```
void [#module_impl_name#_container:]log_trace([#context#],const ECOA:log log);
void [#module_impl_name#_container:]log_debug([#context#],const ECOA:log log);
void [#module_impl_name#_container:]log_info ([#context#],const ECOA:log log);
void [#module_impl_name#_container:]log_warning([#context#],const ECOA:log log);
```

```
void [#module_impl_name#_container:]raise_error([#context#],const ECOA:log log);
void [#module_impl_name#_container:]raise_fatal_error([#context#],const ECOA:log log);
```

11.7 Time Services

The Container Interface API provides the Modules with a set of library functions used to access time services. Three, possibly distinct, time sources shall be provided:

- **Relative Local Time** - The high-resolution real-time clock local to the current computing node, representing the time elapsed since node start up.
- **Absolute System Time** – The synchronised time across an ECOA system, relative to a system clock reference defined by the system integrator. **Absolute System Time** may or may not coincide with **UTC Time**.
- **UTC Time** - The synchronised time across all systems (ECOA and non-ECOA). Defined in terms of UTC, and offset such that zero corresponds to 00:00 1 Jan 1970. **UTC Time** may not be available in all ECOA systems.

The first time source may generally be used to compute and express durations with a high resolution required for real-time precision services. The ECOA infrastructure provides the modules with a high resolution clock which may not be synchronized with other time sources.

As a consequence, the HR clock is considered as local to a Module, and should only be used to locally compute RT durations. The HR clock (called type `ECOA:hr_time`) expressed in seconds and nanoseconds and representing the time elapsed since system start up on that CPU. It is represented as two 32 bit unsigned integers expressed in seconds and nanoseconds. It may only be considered as local to the Module, as Modules may be deployed in different protection domains and hence on different computing nodes, which would mean that the HR time cannot be guaranteed to be synchronised between them.

The ECOA infrastructure may provide the SW modules with UTC time. The globally defined clock has a less precise clock, and should be used to date events. The ECOA infrastructure provides the SW modules with a function to return the currently most precise UTC clock accessible on the current computing node.

A non-UTC global time source is also useful because it may not be desirable to convert to UTC time (e.g. for performance reasons).

The `ECOA:global_time` is used for both UTC and non-UTC system times comprising two 32 bits unsigned integers , seconds and nanoseconds.

The following are prototype definitions for the get time service operations:

```
ECOA:error [#module_impl_name#_container:]get_relative_local_time([#context#],const ECOA:hr_time
*relative_local_time);

ECOA:error [#module_impl_name#_container:]get_UTC_time([#context#],const ECOA:global_time
*utc_time);

ECOA:error [#module_impl_name#_container:]get_absolute_system_time([#context#],const ECOA:global_time
*absolute_system_time);
```

Get Relative Local Time operations may return the following error codes:

`[ECOA:error:OK]`

- `No error`

Get UTC Time operations may return the following error codes:

[ECOA:error:OK]

- No error

[CLOCK_UNSYNCHRONIZED]

- No error – clock is unsynchronized; a valid value is still returned

Get Absolute System Time operations may return the following error codes:

[ECOA:error:OK]

- No error

[CLOCK_UNSYNCHRONIZED]

- No error – clock is unsynchronized; a valid value is still returned

In addition, it is possible to retrieve the time resolution through the following API:

```
void [#module_impl_name#_container:]get_relative_local_time_resolution([#context#],const ECOA:duration
*relative_local_time_resolution);

void [#module_impl_name#_container:]get_UTC_time_resolution ([#context#],const ECOA:duration
*utc_time_resolution);

void [#module_impl_name#_container:]get_absolute_system_time_resolution ([#context#],const ECOA:duration
*absolute_system_time_resolution);
```

The output resolution parameter contains the time resolution provided the underlying software environment. The time resolution is the shortest duration between two updates of the associated clock.

The get time resolution functions shall always return a valid value.

12 References

Ref.	Document Number	Version	Title
1.	IAWG-ECOA-TR-001	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume I Key Concepts
2.	IAWG-ECOA-TR-002	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume II Developers Guide
3.	IAWG-ECOA-TR-003	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 1: Ada Binding Reference Manual
4.	IAWG-ECOA-TR-004	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 2: C Binding Reference Manual
5.	IAWG-ECOA-TR-005	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 3: C++ Binding Reference Manual
6.	IAWG-ECOA-TR-006	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 4: ELI and Transport Binding Reference Manual
7.	IAWG-ECOA-TR-007	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 5: Mechanisms Reference Manual
8.	IAWG-ECOA-TR-008	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 6: Platform Requirements Reference Manual
9.	IAWG-ECOA-TR-009	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 7: Approach to Safety and Security Reference Manual
10.	IAWG-ECOA-TR-011	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume III Part 9: Metamodel and XSD Schemas Reference Manual
11.	IAWG-ECOA-TR-012	Issue 2	European Component Oriented Architecture (ECOA) Collaboration Programme: Volume lV Common Terminology

Table 6 - Table of ECOA references