European Component Oriented Architecture (ECOA®) Collaboration Programme: Preliminary version of the ECOA Architecture Specification Part 4: Software Interface

Dassault Ref No: DGT 2041083-A
Thales DMS Ref No: 69398918-035 --

Issue: 7

Prepared by
Dassault Aviation and Thales DMS

Note: *This specification is preliminary and is subject to further adjustments. Consequently, users are advised to exercise caution when relying on the information herein. No warranties are provided regarding the completeness or accuracy of the information in this preliminary version. The final version of the document will be released to reflect further improvements.*

Without prejudice to the property rights relating to the ECOA AS6 Standard, the information in this document relating to the changes envisaged for the transition from the ECOA AS6 Standard to the ECOA AS7 Standard is the intellectual property of Dassault Aviation and Thales DMS France SAS. The information set out in this document is provided solely on an 'as is' basis and co-developers of this specification make no warranties expressed or implied, including no warranties as to completeness, accuracy or fitness for purpose, with respect to any of the information.

This Page Intentionally Left Blank

This Page Intentionally Left Blank

Contents

Figures

Tables

0 Introduction

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

This document is Part 4 of the Architecture Specification, and describes the software interfaces used.

In an ECOA® SW system, all interactions between 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, time, persistent information and context management.

This document describes the APIs between components and the containers that host them. The APIs, shown in Figure 1, are the Component Interface and the Container Interface:

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

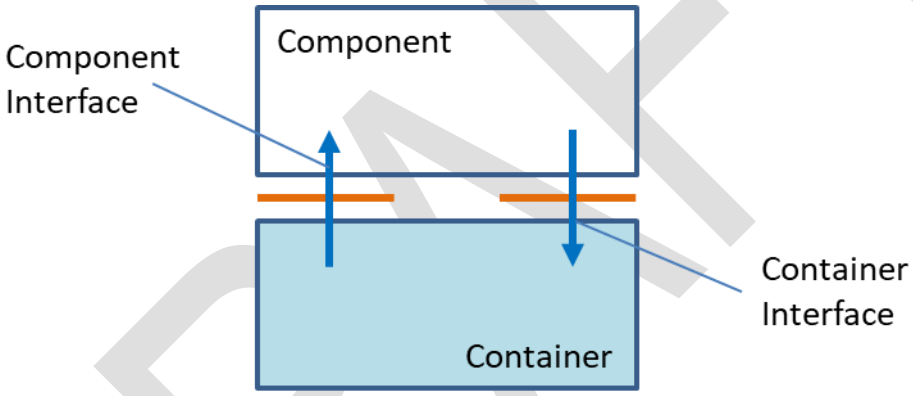

Figure 1 Component and Container Interface

Different language bindings provide mappings for particular programming languages or technologies mapping.

This document describes all the possible mandatory and optional operations a Component Interface and a Container Interface may rely on to be ECOA compliant.

The information in this document is based on v3.0.0 of the ECOA® meta-model.

Convention: optional elements of the SW Interface will be written in *italic* and highlighted.

This document is structured as follows:

- Section 6 describes the Component to Language Mapping
- Section 7 describes the General Rules applicable to SW Interface derived bindings
- Section 8 describes the Component Context
- Section 9 describes the Type libraries
- Section 9.6 describes the Component Interface
- Section 11 describes the Container Interface

- Section 12 describes the Container Types
- Section 13 describes Default Values
- Section 14 describes the External Interface
- Section 15 describes External Components
- Section 16 describes Periodic Trigger Manager Components
- Section 16.2 describes Reference Language Headers

Note: The structure of this document is also the template for the structure of all language bindings documents.

1 Scope

This Architecture Specification specifies a uniform method for design, development and integration of software systems using a component oriented approach.

2 Warning

This specification represents the output of a research programme. Compliance with this specification shall not in itself relieve any person from any legal obligations imposed upon them. Product development shall rely on the BNAE publications of the ECOA standard.

3 Normative References

Architecture Specification Part 1	Dassault Ref No: DGT 2041078-A Thales DMS Ref No: 69398915-035 -- Issue 7 Architecture Specification Part 1 – Concepts
Architecture Specification Part 2	Dassault Ref No: DGT 2041081-A Thales DMS Ref No: 69398916-035 -- Issue 7 Architecture Specification Part 2 – Definitions
Architecture Specification Part 3	Dassault Ref No: DGT 2041082-A Thales DMS Ref No: 69398917-035 -- Issue 7 Architecture Specification Part 3 – Mechanisms
Architecture Specification Part 4	Dassault Ref No: DGT 2041083-A Thales DMS Ref No: 69398918-035 -- Issue 7 Architecture Specification Part 4 – Software Interface
Architecture Specification Part 5	Dassault Ref No: DGT 2041084-A Thales DMS Ref No: 69398919-035 -- Issue 7 Architecture Specification Part 5 – High Level Platform Requirements
Architecture Specification Part 6	Dassault Ref No: DGT 2041491-A Thales DMS Ref No: 69398920-035 -- Issue 7 Architecture Specification Part 6 – Options
Architecture Specification Part 7	Dassault Ref No: DGT 2041086-A Thales DMS Ref No: 69398925-035 -- Issue 7 Architecture Specification Part 7 – Metamodel

4 Definitions

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

5 Abbreviations

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CPU	Central Processing Unit
ECOA	European Component Oriented Architecture. ECOA® is a registered trademark.
HR	High Resolution
ID	Identifier
OO	Object Oriented
PINFO	Persistent Information
POSIX	Portable Operating System Interface
QoS	Quality of Service
UTC	Coordinated Universal Time
XML	eXtensible Markup Language

6 Component to Language Mapping

This section gives an overview of the Component Interface and Container Interface APIs, in terms of symbolic names that refer to instanciated ECOA concepts. Refer to this section in the required language binding for details relevant to that specific language.

Sections 9.6 and 11 contain prototype definitions of Component/Container Interface operations, in table form specifying operation characteristics.

Each prototype is illustrated by an example using C like syntax. **The correct syntax is given by the appropriate language binding**.

Note : The name of each operation shall include the Component Implementation name for those languages that do not support namespacing.

The following symbolic names are used in the prototypes:

- **#component_impl_name#** is the name of the component implementation – the name is used for API generation.
- **#component_instance_name#** is the name of a Component Instance – this name is used for deployment purposes,
- **#operation_name#** is the name of the component operation (event, request-response or versioned data),
- **#external_operation_name#** is the name of the external event operation (used when generating the External Interface API for a Driver Component),
- **#request_parameters#** correspond to the ordered list of input parameters specified for a Request_Received, Request_Sync or a Request_Async operation,
- **#response_parameters#** correspond to the ordered list of output parameters specified for a Response_Received, Request_Sync or a Response_Send operation,
- **#event_parameters#** corresponds to the ordered list of input parameters specified for an event Send or event Received operation,
- **#type_name#** is the name of a data-type[1],
- **#max_concurrent_requests#** corresponds to component model attribute 'maxConcurrentRequests'
- **#context#** will be used to represent the reference to the context associated with a Component Instance defined in section 8,
- **#error_notification_operation_specification#** correspond to operations defined in section 10.4
- **#event_operation_call_specifications#** correspond to operations defined in section 11.1.2.7
- **#request_response_call_specifications#** correspond to operations defined in section 11.1.1
- **#versioned_data_call_specifications#** correspond to operations defined in section 11.1.2
- **#properties_call_specifications#** correspond to operations defined in section 11.2
- **#PINFO_read_call_specifications#** correspond to operations defined in section 11.6.1

[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#`*.

- **#PINFO_write_call_specifications#** correspond to operations defined in section 11.6.2
- **#PINFO_seek_call_specifications#** correspond to operations defined in section 11.6.3
- **#Save_Warm_Start_Context_operation#** correspond to operations defined in section 11.7
- **#trigger_name#** is the name of a trigger declared in a component.
- **#PINFO_name#** is the PINFO name declared at Component Type level within PINFO usage attributes.
- **#variable_name#** is the name of a variable defined in a supervisor component.
- **#variable_type#** is the type of a variable defined in a supervisor component.
- **#external_operation_name#** is the name of a component Event operation whose sender is declared as external in the assembly schema.

Table 1 details the Component and Container Interface APIs. The “level” column specifies whether an interface is mandatory (i.e. required in any language binding), or optional.

The actual API will include the name of the operation and component. 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.

Category	Abstract API Name	Container Operation	Component Operation	Level	Section
Events API	Event_Send	Yes	No	**MANDATORY**	11.1.3
	Event_Received	No	Yes	**MANDATORY**	10.1.3
Request Response API	Request_Sync	Yes	No	**MANDATORY**	11.1.1.1
	Request_Async	Yes	No	**MANDATORY**	11.1.1.2
	Request_Received	No	Yes	**MANDATORY**	10.1.1.1
	Response_Received	No	Yes	**MANDATORY***	10.1.1.2
	Response_Actually_Received Response_Not_Received	No No	Yes Yes		
	Response_Send	Yes	No	**MANDATORY**	11.1.1.3
	Request_Cancel	Yes	No	**OPTIONAL**	11.1.1.4
Versioned Data API	Get_Read_Access	Yes	No	**MANDATORY**	11.1.2.1
	Release_Read_Access	Yes	No	**MANDATORY**	11.1.2.2
	Updated	No	Yes	**MANDATORY**	10.1.2
	Get_Write_Access	Yes	No	**MANDATORY****	11.1.2.3
	Get_Selected_Write_Access	Yes	No		11.1.2.4
	Cancel_Write_Access	Yes	No	**MANDATORY**	11.1.2.5 11.1.2.6
	Publish_Write_Access	Yes	No	**MANDATORY**	11.1.2.6
	Is_Initialized	Yes	No	**OPTIONAL**	11.1.2.7
	Release_All_Data_Handles	Yes	No	**OPTIONAL**	11.1.2.8
Properties API	Get_Value	Yes	No	**MANDATORY**	11.2.1
Runtime Lifecycle API	Initialize_Received	No	Yes	**MANDATORY**	10.2
	Start_Received	No	Yes	**MANDATORY**	
	Stop_Received	No	Yes	**MANDATORY**	
	Shutdown_Received	No	Yes	**MANDATORY**	
	Reset_Received	No	Yes	**MANDATORY**	
Supervisor Components	On_State_Change	No	Yes	**OPTIONAL** [OPTION SUPERVISION]	10.3
	Get_Executable_Status	Yes	No		11.8.1
	Executable_Command	Yes	No		11.8.1
	Component_State_Command	Yes	No		11.8.2
	Get_Component_Status	Yes	No		11.8.2

Category	Abstract API Name	Container Operation	Component Operation	Level	Section
	Get_Variable	Yes	No		11.8.3
	Set_Variable	Yes	No		11.8.3
Logging and Fault Management Services API	Log_Debug Log_Trace Log_Info Log_Warning Raise_Error Raise_Fatal_Error	Yes Yes Yes Yes Yes Yes	No No No No No No	**MANDATORY****	11.3
	Flex_Log Flex_Raise_Fatal_Error	Yes Yes	No No		
	Error_Notification	No	Yes	**OPTIONAL** [OPTION FAULT HANDLER]	10.4
Time Services API	Get_Relative_Local_Time	Yes	No	**MANDATORY**	11.4
	Get_UTC_Time	Yes	No	**OPTIONAL** [OPTION UTC TIME]	
	Get_Absolute_System_Time	Yes	No	**MANDATORY**	

Category	Abstract API Name	Container Operation	Component Operation	Level	Section
	Get_Relative_Local_Time_ Resolution	Yes	No	**OPTIONAL**	11.4
	Get_UTC_Time_Resolution	Yes	No	**OPTIONAL**	
	Get_Absolute_System_ Time_Resolution	Yes	No	**OPTIONAL**	
Triggers	Trigger_Set	Yes	No	**MANDATORY**	11.5
	Trigger_Cancel	Yes	No	**MANDATORY**	
Persistent Information (PINFO) Management	Read	Yes	No	**MANDATORY**	11.6.1
	Write	Yes	No	**OPTIONAL** [OPTION PINFO WRITE]	11.6.2
	Seek	Yes	No	**MANDATORY**	11.6.3
Context Management	Save_Warm_Start_Context	Yes	No	**OPTIONAL** [OPTION WARM START CONTEXT]	11.7
External Interface	External_Event_Received	No	Yes	**OPTIONAL** [OPTION EXTERNAL INTERFACE]	14
External Components	External_Routine	No	Yes	**MANDATORY**	15
	Start_External_Task	Yes	No	**MANDATORY**	
	Stop_External_Task	Yes	No	**MANDATORY**	

Table 1 Component and Container Interfaces

* it is mandatory to define exactly one of the API alternatives in a language binding.
** it is mandatory to define at least one of the API alternatives in a language binding.

7 General Rules on Bindings

The present Architecture Specification Part aims to frame the definition of language bindings in order to:

- Ensure interoperability between ECOA components,
- Enhance portability and reuse of ECOA components.

Note that for portability and reuse, it is strongly recommended to use ECOA Reference Bindings, as ECOA platforms are required to implement one of them (see Architecture Specification Part 5).

The following rules must be strictly applied to any ECOA language binding:

1. Any language binding must specify source code files organization with their dependencies, explain which ones may be modified by component developers
2. Any SW Interface mandatory abstract data type must be clearly defined with an equivalent predefined type in the binding.
   In particular, for enumerations, each SW Interface mandatory label must be covered. The addition of new labels in an enumeration is authorized only if the abstract type is clearly tagged as extensible.
3. Any SW Interface mandatory constant must be clearly defined as a predefined constant in the binding.
4. A language binding shall not define types or constants that are neither used in the component interface nor in the container interface.
5. Fields order in “Record” or “Variant Record” types shall be the same as the one defined in the ECOA model.
6. A language binding shall describe the content of interface files corresponding to the definition of predefined types and constants.
7. A language binding must describe templates for component/container interface files.
8. Any SW Interface mandatory operation must be clearly associated to at least one function in the binding. In case the association is not bijective, the mapping of binding functions with SW Interface operation use cases must be explained.
9. If a language binding offers an optional SW Interface operation, the association between the binding function and the SW Interface operation must be clearly defined.
10. If a single function of the binding is associated to several SW Interface operations, the association must be clearly explained.
11. Any binding function that is not associated to a SW Interface operation shall not affect the functional behaviour of components (e.g. observability service).
12. Any binding function that is associated to a SW Interface operation ensures the same behaviour as specified in the SW Interface.
13. Variability patterns applied to API element names shall only rely on symbolic names defined in section 6 (e.g. #operation_name#).
14. Any binding function associated to a SW Interface operation shall offer at least equivalent mandatory return status.
15. In case a binding function offers complementary return status, the binding specification shall specify which SW Interface mandatory return status can be used instead.
16. Any binding function associated to a SW Interface operation shall offer at least equivalent mandatory parameters (equivalent type and equivalent in/out orientation) with an explicit association.
17. Mandatory parameters, and optional parameters when applicable to the binding, shall be defined in the binding function prototype, except for “context” parameter for which rules defined in section 8 apply.

18. All function parameter types must be ECOA pre-defined types or equivalent as defined in section 9.4, excluding parameters related to an event or a request-response (which are defined in the ECOA model).
19. When two possible abstract types are specified for an operation parameter, it means that binding definition shall fix which one is chosen in the operation prototype.
20. Binding functions parameters order is free, except for Event or Request-Response operation parameters which shall strictly follow the order defined in the ECOA model.
21. Binding functions parameter order shall be specified.
22. The manner in which parameters are passed is language dependent and is described in the individual language bindings.
23. For a request-response operation, any output parameters are treated as inputs when passed to the Response_Send (11.1.1.1) and Response_Received (10.1.1.1.2) functions.
24. The language binding shall specify which ECOA options it complies with (according to list of options defined in ECOA AS7 Part 5).

8 Component Context

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

The purpose of this “Component Context” is to hold all the private data that will be used by:

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

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

The part of the Component Context related to user-defined local private data is optional. It will be made available for a given Component Type depending on Metamodel attributes declared by the ASC supplier. The purpose is to allow ASC suppliers to have a simple stateless component if required. State is only generated and managed where it is required.

The part of the Component Context which holds the infrastructure-level technical and specific data is not optional.

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

With OO languages, the Component Instance will be instantiated as an object of a Component Implementation class declared by the user; its associated Container will be associated to an instance of an ECOA-generated Component Container class. All the "User Component Context" and “Warm Start Context”, where being used, shall be declared within the user Component Implementation class as public attributes. The infrastructure-level technical data shall be declared by the ECOA-infrastructure within the corresponding (generated) Component Container class. In addition, the entry-points declared in the Container Interface are represented as methods of the Container object, so the Component Instance object must have access to its corresponding Container object to enable it to call these methods. This is achieved by the Component Implementation having a pointer or reference to the Container object as a public attribute of the Component Implementation class. The Container would have access to enable it to set it to the appropriate Container object, whilst the Component Instance object will be able to access it for use within the Component Implementation.

The language bindings specify the exact syntax required for the Component User Context and Warm Start Context, as well as the syntax for declaring any infrastructure-level technical and specific data in the Component Context for non-OO languages.

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 Component and Container Interfaces. Libraries are used to organise the types into separate "packages" of types.

9.1 Libraries

Libraries are used to organise the types used by an ECOA system into disjoint sets.

NOTE: Libraries are not hierarchical, i.e. a library does not "contain" any other library.

Type names within the same library shall be unique. All types declared in the same library are located in the same header file. This file will usually be automatically generated by the ECOA toolset from the XML descriptions contained within files of the form:

```
# library#.types.xml
```

The generated header file name and file extension are language specific.

A library named "ECOA" is predefined. No library with this name shall be defined in an ECOA workspace. The predefined library cannot be modified or extended. It contains basic types and other predefined types.

9.2 Basic Types

A number of portable basic types are provided within the ECOA predefined library that should be used to write portable code. They are used for all data interchange between components 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 basic types, which are listed in Table 2, may be used directly in the XML descriptions without using the ECOA predefined library.

Table 2 ECOA Basic Types

ECOA Basic 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

(*) `ECOA:int64` is only available on platforms which support [OPTION INT64]. A dedicated flag `ECOA_INT64_SUPPORT` can be used to select their use – see reference headers in the various bindings.

(**) `ECOA:uint64` is only available on platforms which support [OPTION UINT64]. A dedicated flag `ECOA_INT64_SUPPORT` can be used to select their use – see reference headers in the various bindings.

Table 3 ECOA Predefined Constants

ECOA Basic Type	Constant Name	Constant Value
ECOA:boolean8	TRUE FALSE	1 0
ECOA:int8	*INT8_MIN* *INT8_MAX*	*-127* *127*
ECOA:char8	*CHAR8_MIN* *CHAR8_MAX*	*0 (NUL)* *127*[2] *(DEL)*
ECOA:byte	*BYTE_MIN* *BYTE_MAX*	*0* *255*
ECOA:int16	*INT16_MIN* *INT16_MAX*	*-32767* *32767*
ECOA:int32	*INT32_MIN* *INT32_MAX*	*-2147483647* *2147483647*
ECOA:int64	*INT64_MIN* *INT64_MAX*	*-9223372036854775807* *9223372036854775807*
ECOA:uint8	*UINT8_MIN* *UINT8_MAX*	*0* *255*
ECOA:uint16	*UINT16_MIN* *UINT16_MAX*	*0* *65535*
ECOA:uint32	*UINT32_MIN* *UINT32_MAX*	*0* *4294967295*
ECOA:uint64	*UINT64_MIN* *UINT64_MAX*	*0* *18446744073709551615*
ECOA:float32	*FLOAT32_MIN* *FLOAT32_MAX*	*-3.402823466e+38* *3.402823466e+38*

[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 Basic Type	Constant Name	Constant Value
ECOA:double64	*DOUBLE64_MIN* *DOUBLE64_MAX*	*-1.7976931348623157e+308* *1.7976931348623157e+308*

For all the basic types it shall be possible to determine the minimum and maximum values. In C/C++, for example mandatory constant for basic types will be implemented as macros. These are also defined in the base namespace, i.e. usable without the prefix "ECOA:".

Every specific binding shall define the basic types mapping on concrete language specific types.

9.3 Derived Types

9.3.1 Simple Types

A simple type is a refinement of a basic 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 basic ECOA basic type:

```
<simple type="uint32" name="#simple_type_name#" />
```

EXAMPLE 2 defining a simple type based on a basic ECOA basic type with a restricted range:

```
<simple type="uint32" name="#simple_type_name#" minRange="4" maxRange="10" />
```

EXAMPLE 3 defining a type based upon a previously defined simple type:

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

9.3.2 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. If the optional value of the label is 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). Value entries in the type definition shall be ordered in the numerical order of the associated values (from the lowest value to the highest one).

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#`.

9.3.3 Records

Record 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>
```

9.3.4 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.

It is forbidden to declare multiple 'when' entries with the same selector value.

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#" /> ]
</variantrecord>
```

9.3.5 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="#uint32_constant#" />
```

9.3.6 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="#uint32_constant#" />
```

9.4 Predefined Abstract Types

9.4.1 Function execution return status

The data type `ECOA:return_status` is mandatory and extensible. It is an enumeration (using `uint32` as its base type) declared in the ECOA predefined library, which is used to specify the return status of applicable API operations. The enumeration values are:

ECOA:OK	No error has occurred
ECOA:FAILURE	Generic default return status code for non-nominal execution
ECOA:INVALID_HANDLE	*An invalid handle has been used*
ECOA:DATA_NOT_INITIALIZED	*The data has never been written*
ECOA:NO_DATA	*The call is not able to provide any data*
ECOA:INVALID_IDENTIFIER	*An invalid ID has been used*
ECOA:NO_RESPONSE	*No response received for a request (for example: timeout reached or unavailable server)*
ECOA:TIMEOUT	*No response received for a request when timeout is reached*
ECOA:OPERATION_ALREADY_PENDING	*The requested operation is already being processed*
ECOA:CLOCK_UNSYNCHRONIZED	*The clock is not synchronised*
ECOA:RESOURCE_NOT_AVAILABLE	*Insufficient resource is available to perform the operation.*
ECOA:OPERATION_NOT_AVAILABLE	*The requested operation is not available.*
ECOA:INVALID_PARAMETER	*An invalid parameter has been used*
ECOA:INVALID_IN_PARAMETER	*An invalid IN parameter has been used*
ECOA:INVALID_OUT_PARAMETER	*An invalid OUT parameter has been used*

The `ECOA:return_status` may be defined in the ECOA library as follows:

```
<enum name="return_status" type="uint32">
   <value name="OK" valnum="0"/>
   <value name="FAILURE" valnum="1" />
   <value name="NO_RESPONSE" valnum="2" />
</enum>
```

Note that when a binding defines optional return status codes, it allows interfaces using them to clarify the cause of non-nominal executions. Then, the scope of possibilities covered by FAILURE default code is reduced. FAILURE code remains mandatory anyway.

9.4.2 Component and executable identifiers

The *ECOA:asset_id* is an optional simple type based on an ECOA positive integer type (see section 9.3.1) .

Asset IDs uniquely identify component instances and executables, for supervising or fault management purpose.

The platform tooling may generate a header file for target language bindings. This header file maps assets described above with Ids according to the deployment. This header file may then be used by the developer of the Fault Handler to implement supervisor components behaviour, or recovery actions as a result of errors raised by specific asset IDs and/or to target specific asset IDs.

Note: if the header file is included “as is” in the ECOA Fault Handler source code, the ECOA Fault Handler will need to be recompiled when the deployment changes. Another possible design choice is for the user to convert the header file into a user-defined configuration file (such as a binary PINFO) which would be read by the ECOA Fault Handler.

This header file contains declaration of constants defined in the `ECOA_Assets` library as follows:

```
   <constant name="CMP_#component_instance_name1#" type="ECOA:asset_id"
value="#CMP_ID1#" />
   <constant name="CMP_#component_instance_name2#" type="ECOA:asset_id"
value="#CMP_ID2#" />
   <!-- ... -->
   <constant name="CMP_#component_instance_nameN#" type="ECOA:asset_id"
value="#CMP_IDN#" />

   <constant name="PD_#executable_name1#" type="ECOA:asset_id"
value="#EXE_ID1#" />
   <constant name="PD_# executable_name2#" type="ECOA:asset_id"
value="#EXE_ID2#" />
   <!-- ... -->
   <constant name="PD_# executable_nameN#" type="ECOA:asset_id"
value="#EXE_IDN#" />
```

Names used in the constants are names used in the deployment.

Since it depends on a specific deployment, the `ECOA_Assets` library cannot be used from other ECOA models.

9.4.3 Write access mode

The data type *ECOA:write_access_mode* is an optional enumeration declared in the ECOA namespace, which is used to identify the write access mode required to update a versioned date. The enumeration values are:

READ_AND_UPDATE
WRITE_ONLY

The data type `ECOA:write_access_mode` is not extensible.

It may be defined in the ECOA namespace as follows:

```
<enum name="write_access_mode" type="uint32">
   <value name="READ_AND_UPDATE" valnum="0" />
```

```
    <value name="WRITE_ONLY" valnum="1" />
</enum>
```

9.4.4 Time management

It is mandatory to define one or several types for time management among the following ones.

9.4.4.1 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 999999999.

The `ECOA:hr_time` is a record (see section 9.3.3) defined in the ECOA library as follows:

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

9.4.4.2 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 999999999.

The `ECOA:global_time` is a record (see section 9.3.3) defined in the ECOA library as follows:

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

9.4.4.3 ECOA:duration

A type used for operations that result in communications of delay or duration from one component 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 999999999.

The ECOA:duration is a record (see section 9.3.3) defined in the ECOA library as follows:

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

9.4.4.4 ECOA:nano_time

Possible alternative to define time data based on a simple type:

```
<simple type="int64" name="nano_time" />
```

ECOA:nano_time is expressed in nanoseconds.

9.4.5 Logs

9.4.5.1 ECOA:log

ECOA:log is an optional type. It is a variable array of 256 **ECOA:char8** elements, that defines how an information report is stored. The type is constrained to enable portability, because some implementations may not be able to support unconstrained logging. That is why this type is used in reference bindings. See Section 11.3 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 variable array (see section9.3.6) defined in the ECOA library as follows:

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

9.4.5.2 ECOA:information_category

The data type *ECOA:information_category* is an optional enumeration declared in the ECOA namespace, which is used to specify the category of an information to be transmitted to logging and fault management infrastructure..

The data type ECOA:information_category is not extensible.

It may be defined in the ECOA namespace as follows:

```
<enum name="information_category" type="uint32">
   <value name="NONE" valnum="0" />
   <value name="CRITICAL" valnum="1" />
   <value name="ERROR" valnum="2" />
   <value name="WARNING" valnum="3" />
   <value name="INFO" valnum="4" />
   <value name="DEBUG" valnum="5" />
   <value name="TRACE" valnum="6" />
```

```
</enum>
```

9.4.6 Error management

All the following error management types are optional. They are recommended in language bindings to associate specific types to parameters and then reduce risks of mistake.

9.4.6.1 ECOA:error_id

The *ECOA:error_id* is an optional simple type (see section 9.3.1) defined in the ECOA namespace as follows:

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

Error IDs uniquely identify error occurrences. They are generated by the Infrastructure.

9.4.6.2 ECOA:error_code

The *ECOA:error_code* is an optional simple type (see section 9.3.1) defined in the ECOA library as follows:

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

Error codes may be provided by Component Instances when they raise errors or fatal errors, as well as by the ECOA Infrastructure when it detects an error, in order to provide contextual information about errors to the fault management infrastructure and to Fault Handler components.

The functional meaning of error code values is not standardised by ECOA. Each Component is responsible for specifying the meaning of the error codes it raises. Each Platform supplier is responsible for specifying the meaning of error codes it raises.

Nevertheless, the value 0 shall be considered as a default value that might mean “undefined error” (see usage in §11.3).

9.4.6.3 ECOA:asset_type

The data type *ECOA:asset_type* is an optional enumeration declared in the ECOA namespace, which is used to identify the type of asset either linked to an error. The enumeration values are:

COMPONENT	0	Component instance
EXECUTABLE	1	Executable

The data type `ECOA:asset_type` is not extensible.

It may be defined in the ECOA namespace as follows:

```
<enum name="asset_type" type="uint32">
   <value name="COMPONENT" valnum="0" />
   <value name="EXECUTABLE" valnum="1" />
</enum>
```

9.4.6.4 ECOA:error_type

The data type *ECOA:error_type* is an optional extensible enumeration declared in the ECOA namespace, which is used to specify the type of the error reported to the error handler. The enumeration values are given by the table below. For each error, a short description is given.

Table 4 Table of Errors

Error		Description
RESOURCE_NOT_AVAILABLE		No more resources to carry on the element activities
UNAVAILABLE		The element (potentially a remote platform) has disappeared for an unknown reason
MEMORY_VIOLATION		Memory violation
NUMERICAL_ERROR		Divide by zero or floating-point error
ILLEGAL_INSTRUCTION		Illegal instruction in the binary code
STACK_OVERFLOW		Stack overflow or corruption
DEADLINE_VIOLATION		*Component deadline violation*
OVERFLOW		The component's queue is full or if the container has not enough resources to track concurrent requests.
UNDERFLOW		*The component's queue is not enough fulfilled*
ILLEGAL_INPUT_ARGS		*Illegal input arguments*
ILLEGAL_OUTPUT_ARGS		*Illegal output arguments*
ERROR		Raise_error called by a Component
FATAL_ERROR		Raise_fatal_error called by a Component
HARDWARE_FAULT		*Hardware fault*
POWER_FAIL		*Power failure*
COMMUNICATION_ERROR		Communication error
INVALID_CONFIG		*Invalid configuration. The node is not able to load the configuration*
INITIALISATION_PROBLEM		Initialisation problem. The node is not able to allocate resources or to start components
CLOCK_UNSYNCHRONIZED		*The node clock is not synchronized with the other parts of the system.*

The `ECOA:error_type` is an enumeration (see section 9.3.2) that may be defined in the ECOA namespace as follows:

```
<enum name="error_type" type="uint32">
   <value name="RESOURCE_NOT_AVAILABLE" valnum="0" />
   <value name="UNAVAILABLE" valnum="1" />
   <value name="MEMORY_VIOLATION" valnum="2" />
   <value name="NUMERICAL_ERROR" valnum="3" />
   <value name="ILLEGAL_INSTRUCTION" valnum="4" />
   <value name="STACK_OVERFLOW" valnum="5" />
   <value name="DEADLINE_VIOLATION" valnum="6" />
   <value name="OVERFLOW" valnum="7" />
   <value name="UNDERFLOW" valnum="8" />
   <value name="ILLEGAL_INPUT_ARGS" valnum="9" />
   <value name="ILLEGAL_OUTPUT_ARGS" valnum="10" />
   <value name="ERROR" valnum="11" />
   <value name="FATAL_ERROR" valnum="12" />
   <value name="HARDWARE_FAULT" valnum="13" />
   <value name="POWER_FAIL" valnum="14" />
   <value name="COMMUNICATION_ERROR" valnum="15" />
   <value name="INVALID_CONFIG" valnum="16" />
   <value name="INITIALISATION_PROBLEM" valnum="17" />
   <value name="CLOCK_UNSYNCHRONIZED" valnum="18" />
</enum>
```

9.4.7 Pinfo management

The data type `ECOA:seek_whence_type` is mandatory and not extensible. It is an enumeration declared in the ECOA predefined library, which is used to define the position to consider in a PINFO, when invoking the Seek operation. The enumeration values are:

SEEK_SET	Position is the beginning of the PINFO
SEEK_CUR	Position is the current PINFO'index
SEEK_END	Position is the end of the PINFO

The `ECOA:seek_whence_type` is an enumeration (see section 9.3.2) that may be defined in the ECOA library as follows:

```
<enum name="seek_whence_type" type="uint32">
   <value name="SEEK_SET" valnum="0" />
   <value name="SEEK_CUR" valnum="1" />
   <value name="SEEK_END" valnum="2" />
</enum>
```

9.4.8 Lifecycle management

All Lifecycle management types are optional, but recommended in language bindings to associate specific types to parameters and then reduce risks of mistake.

Nevertheless, they seem only useful if [OPTION SUPERVISOR COMPONENTS] is available in the ECOA Platform.

9.4.8.1 ECOA:component_state

The data type *ECOA:component_state* is an enumeration declared in the ECOA predefined library, which is used to define the lifecycle state. It is not extensible.

The enumeration values are:

- UNAVAILABLE
- IDLE
- READY
- RUNNING

9.4.8.2 ECOA:component_command

The data type *ECOA:component_command* is an enumeration declared in the ECOA predefined library, which is used to list all possible commands to change the lifecycle state. It is not extensible.

The enumeration values are:

- INIT
- START
- STOP
- RESET
- SHUTDOWN
- KILL

9.4.8.3 ECOA:executable_state

The data type *ECOA:executable_state* is an enumeration declared in the ECOA predefined library, which is used to define the lifecycle state. It is not extensible.

The enumeration values are:

- NOT_LAUNCHED
- LAUNCHED

9.4.8.4 ECOA:executable_command

The data type *ECOA:executable_command* is an enumeration declared in the ECOA predefined library, which is used to list all possible commands to change the lifecycle state. It is not extensible.

The enumeration values are:

- START
- STOP

9.5 Constants

A constant is a defined constant value of a given, previously defined, type. It is defined and used in an ECOA model according to ECOA AS7 Part7 specifications.

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

Some examples to illustrate constants definition and use:

EXAMPLE 1 defining a constant of type `ECOA:uint32` with an integer #constant_value#:

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

EXAMPLE 2 defining a constant of type `ECOA:double64` with a floating-point #constant_value#:

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

EXAMPLE 3 defining a constant of type `ECOA:char8`: with an character #constant_value#:

```
<constant name="my_char_constant" type="ECOA:char8" value="e" />
```

EXAMPLE 4 defining a constant of type `ECOA:char8 (hexadecimal)`:with an character #constant_value#:

```
<constant name="my_hexadecimal_constant" type="ECOA:char8" value="0x4B" />
```

EXAMPLE 5 using a constant to bound an array:

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

9.6 Predefined constants

The constant *#component_impl_name#_#operation_name#_MAX_CONCURRENT_REQUESTS* is optional. It allows the component to be aware of the maximum number of requests defined in the component's model by attribute 'maxConcurrentRequests'. This information is useful to allocate memory for storage of the internal data associated to the handling ot these pending requests.

```
<constant name="#component_impl_name#_#operation_name#_MAX_CONCURRENT_REQUESTS"
type="ECOA:uint16" value="#max_concurrent_requests#" />
```

10 Component Interface

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

10.1 Operations

The Component Interface provides a number of entry points that allow the Container to invoke Component Operations that cause a Component Instance to execute a block of functionality. The block of functionality may call any container operation API allowed by its type (see section 10.3).

10.1.1 Request-Response

For components which are declared as a server of a request response operation, Request_Received is provided to initiate the entry point associated to that request.

For components 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

For a Component declared as server of a request-response operation, a function is implemented by the Component to handle the request generated by the client Component. The `ID` parameter is provided by the infrastructure to allow the Component Instance to associate the response with the request (see 11.1.1.1). The name of the function shall be generated to include the name of the operation.

10.1.1.1.1 Request Received when immediate=false

When the attribute "immediate" is set to false in the Component Type model, the Component can send the response at any time, by calling a reply function; not necessarily during the execution of the function handling the request.

Interface specifying elements:

Abstract API Name	Request_Received			
Use Case	immediate = false			
Level	MANDATORY			
Minimal variability patterns	#component_impl_name#; #operation_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	ECOA unsigned integer	IN	request identifier
	P3	#request_parameters#	IN	"in" parameters of the request-response

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

```
void [#component_impl_name#:]#operation_name#__Request_Received([#context#,] ECOA:uint32 ID,
#request_parameters#);
```

10.1.1.1.2 Request Received when immediate=true

When the attribute "immediate" is set to true in the Component Type model, the Component must send the response at the end of the execution of the function handling the request. In this case, in addition to input parameters, this function defines the output parameters of the Operation as output parameters of the function.

Interface specifying elements:

Abstract API Name	Request_Received			
Use Case	immediate = true			
Level	MANDATORY			
Minimal variability patterns	#component_impl_name#; #operation_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	ECOA unsigned integer	IN	request identifier
	P3	#request_parameters#	IN	"in" parameters of the request-response
	P4	#response_parameters#	OUT	"out" parameters of the request-response

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

```
void [#component_impl_name#:]#operation_name#__Request_Received([#context#,] uint32 ID,
#request_parameters#, #response_parameters#);
```

10.1.1.2 End of an asynchronous Request

For a Component declared as client of an asynchronous request-response operation, its interface must allow to handle the response generated by the server Component or to deal with a lack of response. In both cases, the `ID` parameter is provided by the infrastructure to allow the Component Instance to associate the response with the request (see 11.1.1.3**Erreur ! Source du renvoi introuvable.**). This is required because the component could initiate multiple requests prior to receiving any responses. The `#response_parameters#` correspond to the "out" parameters of the request-response, however they are treated as inputs to the function in line with section 7.

10.1.1.2.1 Alternative 1: generic interface

Interface specifying elements:

Abstract API Name	Response_Received			
Use Case	Gives the final result of the request : either the associated response, or the problem that occurred			
Level	MANDATORY			
Minimal variability patterns	#component_impl_name#; #operation_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	ECOA unsigned integer	IN	link to request identifier
	P3	#response_parameters#	IN	"out" parameters of the request-response if response is received
	P4	ECOA:return_status	IN	status on operation execution

The appropriate language binding will define the correct syntax for this required interfaces.

The following illustration provides an abstract definition of the interface to help binding elaboration which merges the two use cases in a single operation thanks to the optional return_status parameter:

```
void [#component_impl_name#:]#operation_name#__Response_Received([#context#,] uint32 ID,
ECOA:return_status status, #response_parameters#);
```

Note that several kinds of problems may explain why the expected response is not received:

- No response received within the expected time
- The server component queue is full
- Server component is IDLE/STOPPED

- Server has called `raise_fatal_error()`
- The operation is not connected to a RequestLink

Some bindings may offer return status codes to better identify which problem has occurred. Depending on the problem, the infrastructure may implement several mechanisms to quicken the call of this function.

The ECOA platform shall be able to manage at least the following return status codes:

- `[ECOA:return_status:OK]`
  No error
- `[ECOA:return_status:FAILURE]`
  Any error that is not managed by a dedicated status code

10.1.1.2.2 Alternative 2: dedicated interfaces

Interface specifying elements:

Abstract API Name	Response_Actually_Received			
Use Case	Response is received before timeout			
Level	MANDATORY			
Minimal variability patterns	#component_impl_name#; #operation_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	ECOA unsigned integer	IN	link to request identifier
	P3	#response_parameters#	IN	"out" parameters of the request-response

Abstract API Name	Response_Not_Received			
Use Case	Warns component not to expect response any more			
Level	MANDATORY			
Minimal variability patterns	#component_impl_name#; #operation_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	ECOA unsigned integer	IN	link to request identifier
Optional Parameters				
	P3	ECOA:return_status	IN	status on interface execution

The optional return status of Response_Not_Received may be used to give information about the reason why the response is not received.

The appropriate language binding will define the correct syntax for these two required interfaces.

The following illustrations provide abstract definitions of theses interfaces to help binding elaboration:

```
void [#component_impl_name#:]#operation_name#__Response_Actually_Received([#context#,] uint32 ID,
#response_parameters#);

void [#component_impl_name#:]#operation_name#__Response_Not_Received([#context#,] uint32 ID,
ECOA:return_status status);
```

10.1.2 Versioned Data Updated

The Updated component operation is a callback used by the Container to notify a component when a new value of Versioned data is available. Once notified, the Component has to explicitly call Get_Read_Access

and Release_Read_Access to access to the updated data. This entry point is used to avoid the use of polling to identify when new values are available.

Interface specifying elements:

Abstract API Name	Versioned_Data_Updated			
Use Case	Versioned Data is refreshed			
Level	MANDATORY			
Minimal variability patterns	#component_impl_name#; #operation_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context

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

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

```
void [#component_impl_name#:]#operation_name#__Updated(#context#);
```

10.1.3 Event Received

For a Component declared as a handler of an event, a function, method or procedure shall be implemented by the Component to handle the reception of the event from all possible senders. The `#event_parameters#` correspond to the "input" parameters of the event. The name of the function shall be generated to include the name of the operation.

Interface specifying elements:

Abstract API Name	Event_Received			
Use Case	Event Received			
Level	MANDATORY			
Minimal variability patterns	#component_impl_name#; #operation_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	#event_parameters#	IN/OUT	values of event parameters

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

```
void [#component_impl_name#:]#operation_name#__Received([#context#,]#event_parameters#);
```

10.2 Component Lifecycle

The Component Interface provides functionality to allow the container to command changes to the lifecycle state of the Component Instances it hosts under the direction of the ECOA Infrastructure. Any Component is initialised and started automatically by the container.

The component lifecycle is discussed more fully in Architecture Specification Part 3.

Operations are provided by the Component Interface to support the following Component Lifecycle functionality. These operations are applicable to all Component Instances (including Periodc Trigger Manager and EXTERNAL components).

- **INITIALIZE_Received**: this is the initialisation entry-point of the component used to perform its local initialisation; the Initialize entry-point of a Component is the function in which the Component is supposed to initialise all its local variables (user context).
- **START_Received**: this is the entry point which is used for starting a Component Instance from a technical point of view. Once this entry point has been completed, the Component Instance is ready to process incoming Operations. The ECOA Infrastructure calls this entry point after the **INITIALIZE_Received** entry point has returned.
- **STOP_Received**: this event may be sent to the Component as part of a graceful shutdown procedure in order to change the Component state from RUNNING to READY
- **SHUTDOWN_Received**: this event may be sent to the Component as part of a graceful shutdown procedure in order to change the Component state from READY or RUNNING to IDLE
- **RESET_Received**: this event may be sent to the Component when it is in RUNNING state, in order to "reset" its state in a functional way. The exact meaning of "reset" is only dependent on the Component itself, and not constrained by the standard. Note that this entrypoint may or may not exist: it is an option defined in the implementation model.

At API level, the following abstract operation will be invoked by the container and shall be implemented by the Component (mandatory function) for each lifecycle transition among INITIALIZE, START, STOP, SHUTDOWN and RESET.

Interface specifying elements for a lifecycle operation:

Abstract API Name	XXXX_Received			
Use Case	Component XXXX transition is activated			
Level	MANDATORY			
Minimal variability patterns	#component_impl_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context

The appropriate language binding will define the correct syntax for these component operations.

The following format is given as an example of prototype definitions in a specific binding:

```
void [#component_impl_name#:]INTIALIZE__Received(#context#);

void [#component_impl_name#:]START__Received(#context#);

void [#component_impl_name#:]STOP__Received(#context#);

void [#component_impl_name#:]SHUTDOWN__Received(#context#);

void [#component_impl_name#:]RESET__Received(#context#);
```

Within these five operations the component is restricted such that it may not call any Request Response container operation API (i.e. Request_Sync, Request_Async or Response_Send). This is to prevent race conditions and deadlock due to the start-up order of components. The component may still call any other container operation API allowed by its type (see section 10.3).

10.3 Supervisor components

This section is specific to [OPTION SUPERVISION].

The Component Interface of SUPERVISOR components allows them to be informed of lifecycle state changes of component instances of the application.

A specific handler operation shall be called by the ECOA Infrastructure immediately after all lifecycle state changes of all component instances of the application, including itself.

This operation shall be called independently of the lifecycle state of the SUPERVISOR component itself. This property allows a SUPERVISOR to initialize and start all the components of an application, including itself.

Typically, a SUPERVISOR component will use this operation to manage components and executables lifecycles, by calling container interface to send commands to components and executables (see §11.8), whenever required.

Interface specifying elements:

Abstract API Name	On_State_Change			
Use Case	Supervisor component is informed of a supervised component state change			
Level	OPTIONAL			
Minimal variability patterns	#component_impl_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	ECOA:asset_id	IN	supervised component identifier
	P3	ECOA:component_state	IN	new supervised component state
	P4	ECOA:component_state	IN	previous supervised component state

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

```
void [#component_impl_name#:]On_State_Change([#context#,] ECOA:asset_id component_id,
  ECOA:component_state state, ECOA:component_state previous_state);
```

10.4 Error notification for fault handler components

This section is applicable only to components with the option 'isFaultHandler' set to true in the component implementation model (then #error_handler_implementation_name# = #component_impl_name#).

The following operation provides error handling functionality that may be used by the platform to provide information to Fault Handler components when an error occurs.

Interface specifying elements:

Abstract API Name	Error_Notification			
Use Case	Warns fault handler components that an error occurred			
Level	OPTIONAL			
Minimal variability patterns	#error_handler_implementation_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	ECOA:error_id	IN	error identifier (unique int hte scope of the notified Fault Handler)
	P3	*ECOA time predefined type*	IN	time at which the error has been initially detected (type to be chosen in section 9.4.4)
	P4	ECOA:asset_id	IN	identified the asset linked to the error (unique for a given asset_type)
	P5	ECOA:asset_type	IN	type of asset linked to the error (component or executable)
	P6	ECOA:error_type	IN	type of the error raised
	P7	ECOA:error_code	IN	information provided by the asset that raised the error. This may be used to pass more detailed information to the Fault Handler.

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

```
void [#error_handler_implementation_name#:]error_notification([#context#,] ECOA:error_id error_id,
ECOA:global_time timestamp, ECOA:asset_id asset_id, ECOA:asset_type asset_type, ECOA:error_type
error_type, ECOA:error_code error_code);
```

This error notification API 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 error), or when errors are raised by components.

The Infrastructure will not provide incompatible asset ID and error types (e.g. a module cannot be associated to an OVERRATED error).

Within the handler, the Fault Handler may at least call any log function (§11.3).

The availability of error notifications and their applicability to different asset types depend on the capabilities offered by the underlying platform. They may be specified in more detail by extensions of the ECOA Standard.

11 Container Interface

11.1 Operations

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

11.1.1 Request Response

Two operations are provided to allow Components to issue requests to other components:

- Synchronous Request (mandatory)
- Asynchronous Request (mandatory)

The operation Response Send (mandatory) is provided to return the result to the requesting component. At last, an optional operation Request Cancel allows to cancel the handling of a pending request.

11.1.1.1 Synchronous Request

An operation provided by the Container, used by a Component to invoke an operation provided by a server Component. The calling Component is blocked until the response is received.

NOTE: If the calling Component shares a same thread with other Components of equal priorities, and makes a synchronous request, all these components are blocked waiting for the response or the expiration of the timeout..

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

Interface specifying elements:

Abstract API Name	Request_Sync			
Use Case	A client component sends a synchronous request to a server component			
Level	MANDATORY			
Minimal variability pattern	#component_impl_name#; #operation_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	#request_parameters#	IN	"in" parameters of the request-reponse
	P3	#response_parameters#	OUT	"out" parameters of the request-response
	P4	ECOA:return_status	OUT	status on interface execution

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

```
ECOA:return_status
[#component_impl_name#_container:]#operation_name#__Request_Sync([#context#,]#request_parameters#,
#response_parameters#);
```

The ECOA platform shall be able to manage at least the following return status codes:

- `[ECOA:return_status:OK]`
  No error
- `[ECOA:return_status:NO_RESPONSE]`
  No response received within the expected time
- `[ECOA:return_status:FAILURE]`
  Any error that is not managed by a dedicated status code

Some ECOA platforms may indeed offer complementary return status codes to provide a more accurate analysis of failure cases. For example :

- `[ECOA:return_status:INVALID_PARAMETER]`

A Synchronous Request FAILURE status code can also cover the following issues:

- The server component queue is full
- Server component is IDLE/STOPPED
- Server has called `raise_fatal_error()`
- Container unable to send request (including if the operation is not connected to a RequestLink)

Depending on the problem, the infrastructure may implement several mechanisms to accelerate the response of this function.

11.1.1.2 Asynchronous Request

An operation provided by the Container, used by a Component to invoke an operation provided by a server Component. The `ID` parameter is provided by the infrastructure to allow the Component Instance to associate the response with the request (see 10.1.1.1.2). This `ID` is unique for each Component Instance and for each call of the operation (because the component could initiate multiple requests prior to receiving any responses). The `#request_parameters#` correspond to the "in" parameters of the request-response.

The operation returns immediately so the calling Component is not blocked. If an infrastructure problem prevents the call from succeeding, the fault is handled via the fault management infrastructure.

Furthermore, if the maximum number of concurrent asynchronous requests that the component is authorized to perform has been reached and the component invokes another asynchronous request, the container shall not proceed the request and return a failure status code.

Interface specifying elements:

Abstract API Name	Request_Async			
Use Case	A client component sends an asynchronous request to a server component			
Level	MANDATORY			
Minimal variability pattern	#component_impl_name#; #operation_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	ID	OUT	request identifier
	P3	#request_parameters#	IN	"in" parameters of the request-reponse
	P4	ECOA:return_status	OUT	status on interface execution

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

```
ECOA:return_status
[#component_impl_name#_container:]#operation_name#__Request_Async([#context#,]ECOA:uint32* ID,
#request_parameters#);
```

The ECOA platform shall be able to manage at least the following return status codes:

- `[ECOA:return_status:OK]`
  No error
- `[ECOA:return_status:FAILURE]`
  Any error that is not managed by a dedicated status code

Examples of other return status codes that the platform may implement to specify failure cases:

- `[ECOA:return_status:RESOURCE_NOT_AVAILABLE]`
- `[ECOA:return_status:INVALID_PARAMETER]`

11.1.1.3 Response Send

An operation provided by the Container, used by the Component to send a Response. The `ID` parameter is provided by the infrastructure in the associated Request Received operation and enables the Component Instance to associate the response with the request (see 10.1.1.1). The `#response_parameters#` correspond to the “out” parameters of the request-response, however they are treated as inputs to the function in line with section 7.

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

Interface specifying elements:

Abstract API Name	Response_Send			
Use Case	A server component sends a response to a client component asynchronous request			
Level	MANDATORY			
Minimal variability pattern	#component_impl_name#; #operation_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	ID	IN	client request identifier
	P3	#response_parameters#	IN	"out" parameters of the request-reponse
	P4	ECOA:return_status	OUT	status on interface execution

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

```
ECOA:return_status [#component_impl_name#_container:]#operation_name#__Response_Send([#context#,]
ECOA:uint32 ID,  #response_parameters#);
```

The ECOA platform shall be able to manage at least the following return status codes:

- `[ECOA:return_status:OK]`
  No error
- `[ECOA:return_status:FAILURE]`
  Any error that is not managed by a dedicated status code

Examples of other return status codes that the platform may implement to specify failure cases:

`[ECOA:return_status:INVALID_PARAMETER]`
`[ECOA:return_status:INVALID_IDENTIFIER]`

11.1.1.4 Request Cancel

As the number of pending requests for a server Component is limited (see §.9.6), an operation is defined to cancel the handling of a pending request, i.e. notify the infrastructure and the caller that no response will ever be sent for this request. If a timeout value is defined, the timeout expiration is anticipated.

Interface specifying elements:

Abstract API Name	Request_Cancel			
Use Case	A server component cancels the handling of a pending request			
Level	OPTIONAL			
Minimal variability pattern	#component_impl_name#; #operation_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	ID	IN	client request identifier
	P3	ECOA:return_status	OUT	status on interface execution

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

```
ECOA:return_status [#component_impl_name#_container:]#operation_name#__Request_Cancel([#context#,]
ECOA:uint32 request_ID);
```

Mandatory return status codes:

- `[ECOA:return_status:OK]`

  No error
- `[ECOA:return_status:FAILURE]`

  Any error that is not managed by a dedicated status code

11.1.2 Versioned Data

The container provides operations that allow Components to read from or write to Versioned Data. The operations provided allow a Component Instance to:

- Get (request) Read Access (Mandatory)
- Release Read Access (Mandatory)
- Get (request) Write Access (Mandatory alternative)
- Get (request) Selected rWrite Access (Mandatory alternative)
- Cancel Write Access (without writing new data) (Mandatory)
- Publish (write) new data (automatically releases write access) (Mandatory)
- Is Initialized (Optional)
- Release All Data Handles (Optional)

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

A Data Handle structure shall contain the following fields:

- An attribute used to provide access to the data version
- An attribute, called "stamp", which reflects if a writer has performed a publish action on the data (without access control), or if the data has been locally updated (with access control)

Additionally, the Data Handle structure may contain 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 appropriate language binding will define the correct syntax for the Data Handle structure.

The following illustration provides an abstract definition of Data Handle structure to help binding elaboration:

```
typedef struct {
   #type_name#* data;
   ECOA:uint32 stamp;
   ECOA:byte platform_hook[32];
} [#component_impl_name#_container:]#operation_name#_handle;
```

11.1.2.1 Get_Read_Access

For a Component 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:

- With access control this data space contains a local, read-only copy of the data.
    - NOTE: Although this is a read-only operation, it is possible for a Component to locally change the data. Any local changes made will be lost after a release operation however.
- Without access control this data space is the actual version data repository space as there are no local copies.
    - NOTE: Although this is a read-only operation, any change to the data made by a Reader Component will affect the repository. The ECOA Infrastructure cannot prevent a Reader from writing in the repository when access control is disabled.

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 (with access control), or with the handle to the actual data in the repository (without access control). The stamp attribute in the data handle enables the caller to determine whether the data has been locally updated (with access control) or if a writer has performed a publish action on the data (without access control). It does not reflect a global information shareable between all readers.

If the provider has never published an initial value, the Data Handle will contain a null pointer and the stamp will be equal to zero. Any language binding shall give a solution to detect this case (using a dedicated status code or Is_Initialized API).

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 should not be used. In an error condition, the stamp shall be set to the default for the type.

Architecture Specification Part 3 specifies how the ECOA Infrastructure shall manage the stamp value.

Interface specifying elements:

Abstract API Name	Get_Read_Access			
Use Case	Gets read access to a Versioned Data (provided that the component is declared as a reader for this VD)			
Level	MANDATORY			
Minimal variability pattern	#component_impl_name#; #operation_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	#component_impl_name# #operation_name# Data Handle structure	OUT	data handle which allows to access actual data or copy of the actual data (depending on access control parameter)
	P3	ECOA:return_status	OUT	status on interface execution

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

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

The ECOA platform shall be able to manage at least the following return status codes:

- `[ECOA:return_status:OK]`

No error and a data value is available for reading

- `[ECOA:return_status:FAILURE]`

Any other case that is not managed by a dedicated status code, for example:

- No initial value ever published.
- API called with an invalid versioned data handle
- Maximum number of versioned data reached
- Container unable to provide a versioned data

If the platform does not implement the optional "Is_Initialized" API, then it shall also manage the following return status:

- `[ECOA:return_status:NO_DATA]`

No initial value ever published. The data has never been written (including if the operation is not connected to a DataLink).

Examples of other return status codes that the platform may implement to specify failure cases:

`[ECOA:return_status:INVALID_HANDLE]`

`[ECOA:return_status:RESOURCE_NOT_AVAILABLE]`

11.1.2.2 Release_Read_Access

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

Without access control, this operation signals to the Infrastructure that the data handle is no longer required. There is no local copy to be released.

Interface specifying elements:

Abstract API Name	Release_Read_Access			
Use Case	Reader component has finished reading the data			
Level	MANDATORY			
Minimal variability pattern	#component_impl_name#; #operation_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	#component_impl_name# #operation_name# Data Handle structure	OUT	data handle which allows to access actual data or copy of the actual data (depending on access control parameter)
	P3	ECOA:return_status	OUT	status on interface execution

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

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

The ECOA platform shall be able to manage at least the following return status codes:

- `[ECOA:return_status:OK]`

No error

- `[ECOA:return_status:FAILURE]`

Any error that is not managed by a dedicated status code, for example:

  - API called with an invalid versioned data handle

Examples of another return status code that the platform may implement to better identify failure cases:

```
[ECOA:return_status:INVALID_HANDLE]
```

11.1.2.3 Get_Write_Access

For a Component 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:

- With access control:
  - In “Read+Write” mode, this data space contains a local, read-write copy of the data initialized with the latest value available locally.
  - In “Write only” mode, this data space contains a local uninitialized copy of the data.
- Without access control this data space is the actual version data repository space as there are no local copies.

The operation does not block and returns immediately with a handle as previously described. The stamp attribute in the data handle enables the caller to determine whether the data has been locally updated (with access control) or if a writer has performed a publish action on the data (without access control). With access control, 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 either the most updated value (in “Read+Write” mode) or an uninitialized value (in “Write only” mode). With access control, 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. Without access control, each call to Get_Write_Access will require a call to either Cancel_Write_Access or Publish_Write_Access to release the data handle and avoid reaching the maxVersion attribute value.

If the data has never been written, Get_Write_Access does not return ECOA:OK but returns a valid data handle towards a valid memory area. Any language binding shall give a solution to detect this case which is not an error (using a dedicated status code or Is_Initialized API).

If there is an infrastructure problem that prevents the API from succeeding, ECOA:FAILURE or a dedicated optional status code is returned to the caller, and the infrastructure handles the fault via the fault management infrastructure.

If an error is returned from Get_Write_Access, the call to Cancel_Write_Access is not required. In an error condition, the stamp shall be set to the default for the type.

NOTE: if the operation is not connected to a DataLink, the Writer will still be able to write in a local repository not accessible by any other component

Interface specifying elements:

Abstract API Name	Get_Write_Access			
Use Case	Writer component gets write access to a Versioned Data (for partial or full update, depending on specified mode and access control)			
Level	MANDATORY			
Minimal variability pattern	#component_impl_name#; #operation_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
		#component_impl_name# #operation_name#		data handle which allows to access actual
	P2	Data Handle structure	OUT	data or copy of the actual data
	P3	ECOA:return_status	OUT	status on interface execution

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

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

The ECOA platform shall be able to manage at least the following return status codes:

- `[ECOA:return_status:OK]`

No error and the data has been previously initialized

- `[ECOA:return_status:FAILURE]`

Any other case that is not managed by a dedicated status code, for example:

- No error – the data has never been written. Initialization requested.
- API called with an invalid versioned data handle
- Maximum number of versioned data reached
- Container unable to provide versioned data

Examples of other return status code that the platform may implement to specify failure cases:

[ECOA:return_status:DATA_NOT_INITIALIZED]

[ECOA:return_status:INVALID_HANDLE]

[ECOA:return_status:RESOURCE_NOT_AVAILABLE]

11.1.2.4 Get_Selected_Write_Access

Interface specifying elements:

Abstract API Name	Get_Selected_Write_Access			
Use Case	Writer component gets write access to a Versioned Data accordingly to the selected mode (full overwrite or partial update)			
Level	MANDATORY			
Minimal variability pattern	#component_impl_name#; #operation_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	#component_impl_name# #operation_name# Data Handle structure	OUT	data handle which allows to access actual data or copy of the actual data
	P3	ECOA:write_access_mode	IN	write access mode
	P4	ECOA:return_status	OUT	status on interface execution

P3 parameter specifies if P2 data handle should be initialized by the infrastructure or not:

- WRITE_ONLY: do not initialize the area. It may contain any value. Use this to optimize execution time when the application code will fully initialize the area anyway.
- READ_AND_UPDATE: initialize the area with the current version of the data. If it does not exist, or if the DataWritten operation has the 'writeOnly' atttribute set in the component's model, then a failure status is returned and data handle in P2 is not valid.

The ECOA platform shall be able to manage at least the following return status codes:

- [ECOA:return_status:OK]

No error and the data has been previously initialized

- [ECOA:return_status:FAILURE]

11.1.2.5 Cancel_Write_Access

With access control, this operation signals to the container that the calling component 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 component should not access the local copy of the data after calling this operation as it cannot be guaranteed to be consistent.

Without access control, although there is no local copy to be released, this operation signals to the Infrastructure that the data handle is no longer required.

Interface specifying elements:

Abstract API Name	Cancel_Write_Access			
Use Case	Releases writer component data handle			
Level	MANDATORY			
Minimal variability pattern	#component_impl_name#; #operation_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	#component_impl_name# #operation_name# Data Handle structure	OUT	data handle which allows to access actual data or copy of the actual data
	P3	ECOA:return_status	OUT	status on interface execution

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

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

The ECOA platform shall be able to manage at least the following return status codes:

- `[ECOA:return_status:OK]`

No error

- `[ECOA:return_status:FAILURE]`

Any error that is not managed by a dedicated status code, for example:

  - API called with an invalid versioned data handle

Example of another return status code that the platform may implement to specify failure cases:

`[ECOA:return_status:INVALID_HANDLE]`

11.1.2.6 Publish_Write_Access

With access control, this operation signals to the container that the calling component 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 component should not access the local copy of the data after calling this operation as it cannot be guaranteed to be consistent.

Without access control, this operation signals to the Infrastructure that the data handle is no longer required. There is no local copy to be released as all changes made to the data are directly written in the repository by the component without going through a commit phase.

The operation does not block. An error message is returned to the caller if the handle is invalid (e.g. the component is reusing a handle already released). Any other fault is handled by the infrastructure.

Interface specifying elements:

Abstract API Name	Publish_Write_Access			
Use Case	Makes writer component data update available for readers			
Level	MANDATORY			
Minimal variability pattern	#component_impl_name#; #operation_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	#component_impl_name# #operation_name# Data Handle structure	OUT	data handle which allows to access actual data or copy of the actual data
	P3	ECOA:return_status	OUT	status on interface execution

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

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

The ECOA platform shall be able to manage at least the following return status codes:

- `[ECOA:return_status:OK]`

No error

- `[ECOA:return_status:FAILURE]`

Any error that is not managed by a dedicated status code, for example:

  - API called with an invalid versioned data handle

Example of another return status code that the platform may implement to specify failure cases:

`[ECOA:return_status:INVALID_HANDLE]`

11.1.2.7 Is Initialized

This function allows the component to know if a data has a value or not, i.e. if it has been initialized, either by a default value in the assembly, or by a write operation.

It is used when functions “Get Access” on data do not offer dedicated return status code to detect when these data are not initialized.

Interface specifying elements:

Abstract API Name	Is_Initialized			
Use Case	Indicates whether a data is initialized (i.e. has a significant value) or not			
Level	OPTIONAL			
Minimal variability pattern	#component_impl_name#; #operation_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	ECOA:boolean8	OUT	true if the data is initialized

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

```
void [#component_impl_name#_container:]#operation_name#__Is_Initialized([#context#,] ECOA:boolean8*
result);
```

11.1.2.8 Release All Data Handles

This function allows to release all data handles (read and write) obtained by the component. It can be used to simplify data handles management, by ensuring at a given point in code that no handle is kept by the component.

Interface specifying elements:

Abstract API Name	Release All Handles			
Use Case	Releases all data handles owned by a component			
Level	OPTIONAL			
Minimal variability pattern	#component_impl_name#; #operation_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

```
void [#component_impl_name#_container:]#operation_name#__Release_All_Data_Handles(#context);
```

11.1.3 Event Send

For a Component 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 `#event_parameters#` correspond to the “input” parameters of the event. The name of the function shall be generated to include the name of the operation.

The operation returns immediately so the calling Component is not blocked. If an infrastructure problem prevents the call from succeeding (e.g. if erroneous parameters are given), an error indication may be returned to the caller. Anyway, the fault is handled via the fault management infrastructure.

Interface specifying elements:

Abstract API Name	Event Send			
Use Case	Sends an Event			
Level	MANDATORY			
Minimal variability pattern	#component_impl_name#; #operation_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	#event_parameters#	IN	parameters associated to the event
Optional parameters				
	P3	ECOA:return_status	OUT	status on interface execution

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

```
void [#component_impl_name#_container:]#operation_name#__Send([#context#,]#event_parameters#);
```

The ECOA platform may manage the following return status codes:

- `[ECOA:return_status:OK]`
  No error
- `[ECOA:return_status: INVALID_PARAMETER]`
  Erroneous parameter given as an input
- `[ECOA:return_status:FAILURE]`
  Any other error

11.2 Properties

The Container Interface may include operations which allow a Component to access properties at runtime. Component properties are defined for a Component Type; the value is assigned for each Component Instance. The Component Instance Property Value can be either:

- A literal value;
- A reference to a Component Property

A Component Property is defined within the Component Definition, whose value is assigned for each Component Instance within the Assembly. In addition it is also possible to define Assembly Property Values which may be referenced by a Component Instance Property Value. Note that it is not possible to directly reference an Assembly Property Value from a Component Instance Property Value as detailed in section 11.2.2.

This mechanism allows different instances of Components to have access to property values specified at the Component Instance, Component Instance or Assembly level.

A property may be a Basic Type, a Simple Type, an Enumeration, or a Fixed or Variable Array of these types.

11.2.1 Get_Value

Used by Component Instances to get read only access to the properties.

Interface specifying elements:

Abstract API Name	Get Value			
Use Case	Allows to read a property value			
Level	MANDATORY			
Minimal variability pattern	#component_impl_name#; #property_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	#property_type_name#	OUT	value of the property
Optional Parameter				
	P3	ECOA:return_status	OUT	status on interface execution

Where:

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

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

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

The ECOA platform may manage the following return status codes:

- `[ECOA:return_status:OK]`
  No error
- `[ECOA:return_status:FAILURE]`
  Any error not managed by a dedicated error code

11.2.2 Expressing Property Values

Values given to properties are set in Component Implementations or in assembly schemas through the writing of strings, as described below. It is a syntax that allows Basic, Simple, Enumerations and Fixed or Variable Arrays of these types to be represented.

NOTE: the character strings used to assign property values do not need to be enclosed in double quotes except for the special case of an array of char8 detailed below.

- « Basic », « Simple » : direct value
  EXAMPLES 16, 0xFFFFFFFF, -10, 100.234
- « Enum » : symbol

The case shall follow the one used in the XML type definition.

EXAMPLE: AIR, GROUND.

- « FixedArray » : list of « maxNumber » values of Basic, Simple or Enumerations, comma separated, surrounded by square braces '[ ]' or string syntax with ""
  EXAMPLE (for maxNumber=5): [1,2,3,4,5]
- « Array » : list of N values (where 0≤N≤maxNumber) of Basic, Simple or Enumerations, comma separated, surrounded by square braces '[ ]' or string syntax with ""
  EXAMPLE (for maxNumber=10, current_size = 3): [1, 2, 3]
- Character syntax for type char8

The expression ' ' is allowed in property values to represent a single character.

EXAMPLE 'K'.

EXAMPLE for an array "KEY" can be written as ['K', 'E', 'Y'].

A single character can be represented by its ASCII code using integer or hexadecimal.

EXAMPLE 'K' can be written as 75.

EXAMPLE for an array "KEY" can be written as [75, 69, 89].

EXAMPLE 'K' can be written as 0x4B.

EXAMPLE for an array "KEY" can be written as [0x4B, 0x45, 0x59].

It is possible to mix the different character syntaxes

EXAMPLE for an array "KEY" can be written as [0x4B, 'E', 89].

- String syntax for FixedArray or Array of type char8

Character list surrounded with ""

Equivalent to an array with values of char8

For FixedArray, the number of initializer elements must be equal to maxNumber.

EXAMPLE (for maxNumber=5): “HELLO”

For Array, the number of initializer elements must be less than or equal to maxNumber.

EXAMPLE (for Array with maxNumber=10, current_size = 2): “HI”

Escape character for ‘"’ is ‘\’.

EXAMPLE (for a FixedArray with maxNumber=7): "\"ABCDE\""

- Support for constants - valid for integer and floating-point types only

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 property values:

- Syntax to refer to an Component Property from a Component Instance Property Value

To reference a Component Property from a Component Instance Property Value, the ‘$’ sign shall be used to prefix the name of the Component Property as follows:

  - $#component_property_name#

- Syntax to refer to a Composite Property from an Component Instance Property Value

To reference Composite Property from a Component Instance Property value, the ‘$’ sign shall be used to prefix the name of the Composite Property in the "value" attribute as follows:

  - <propertyValue name="#component_property_name#" value="$#assembly_property_name#" />

Reminder: by definition, artefacts within the Component Implementation scope, such as Component Instance Property Values cannot reference Composite Properties since these are not visible from the scope of Component Implementation.

11.2.3 Example of Defining and Using Properties

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

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

The following XML shows how a property is defined for a Component Type and how a value is assigned to a Component Instance. Two properties are defined for the Component Type. One property is assigned a literal value, the other references a Component Property.

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

...

<instance name="inst1" type="component1" implementation="impl">
```

```
      <propertyValue name="Update_Rate" value="$Update_Rate" />
      <propertyValue name="Component_Inst_Prop" value="20" />
</instance>

<instance name="inst2" type="component1" implementation="impl">
      <propertyValue name="Update_Rate" value="$Update_Rate" />
      <propertyValue name="Component_Inst_Prop" value="2" />
</instance>
```

The composite's Property Values are defined in the upper-lever assembly:

```
<instance name="example_instance" type="..." implementation="...">
      <propertyValue name="Update_Rate" value="10.0" />
</instance>
```

According to the language binding example given in section 11.2.1, 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_Component_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 “inst1” and “inst2” Component Instances. However the get_Component_Inst_Prop_value API would return 20 for the “inst1” Component Instance, but 2 for the “inst2” Component Instance.

11.3 Logging and Fault Management

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

- **Application 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.

Table 5 Logging Error Level

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.	Component shall be shutdown by the infrastructure and fault is reported to the fault management infrastructure. The fault management infrastructure shall filter these errors to determine whether the Application is to be shutdown or not. 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 Component is to be shutdown or not. Information is logged.	No
WARNING	Used by the application to log runtime issues which are undesirable or unexpected, but not necessarily "wrong". Useful for non-intrusive analysis. The Component Instance performing the log is not stopped (i.e. continues execution).	Information is logged.	Yes
INFO	Used by the application to log runtime events (e.g. startup/shutdown). Useful for non-intrusive analysis. The current Component Instance is not stopped (i.e. 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

At runtime, logging levels are determined according to the alternative of fault management API available in the language binding. There are only two possible alternatives, and at least one of them shall be offered:

1. **Using fixed interfaces:** an entry-point in the Container Interface is associated with each of the categories listed in Table 5. A fixed list of parameters is defined for each operation prototype. This alternative may be recommended for development requiring strict coding rules.
2. **Using flex interfaces:** one entry-point for fatal errors, and a single generic entry-point allowing to manage information categories other than FATAL. All operation prototypes are flex.

Note that it is possible for a language binding to additionally offer all or part of the other alternative, taking care then to clearly make the difference between fixed and flex interfaces for raising a fatal error.

11.3.1 Alternative 1: using fixed interfaces

An entry-point in the Container Interface is associated with each of the categories in Table 5. If necessary the container shall truncate the data to the maximum size of `ECOA:log`.

11.3.1.1 Log_Trace

Interface specifying elements:

Abstract API Name	Log_Trace			
Use Case	Provides information to ECOA infrastructure according to error category TRACE			
Level	MANDATORY (if alternative 1)			
Minimal variability pattern	#component_impl_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	ECOA:log	IN	information given to the ECOA infrastructure

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

```
void [#component_impl_name#_container:]log_trace([#context#,]const ECOA:log log);
```

11.3.1.2 Log_Debug

Interface specifying elements:

Abstract API Name	Log_Debug			
Use Case	Provides information to ECOA infrastructure according to error category DEBUG			
Level	MANDATORY (if alternative 1)			
Minimal variability pattern	#component_impl_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	ECOA:log	IN	information given to the ECOA infrastructure

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

```
void [#component_impl_name#_container:]log_debug([#context#,]const ECOA:log log);
```

11.3.1.3 Log_Info

Interface specifying elements:

Abstract API Name	Log_Info			
Use Case	Provides information to ECOA infrastructure according to error category INFO			
Level	MANDATORY (if alternative 1)			
Minimal variability pattern	#component_impl_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	ECOA:log	IN	information given to the ECOA infrastructure

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

```
void [#component_impl_name#_container:]log_info ([#context#,]const ECOA:log log);
```

11.3.1.4 Log_Warning

Interface specifying elements:

Abstract API Name	Log_Warning			
Use Case	Provides information to ECOA infrastructure according to error category WARNING			
Level	MANDATORY (if alternative 1)			
Minimal variability pattern	#component_impl_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	ECOA:log	IN	information given to the ECOA infrastructure

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

```
void [#component_impl_name#_container:]log_warning([#context#,]const ECOA:log log);
```

11.3.1.5 Raise_Error

Interface specifying elements:

Abstract API Name	Raise_Error			
Use Case	Provides information to ECOA infrastructure according to error category ERROR			
Level	MANDATORY (if alternative 1)			
Minimal variability pattern	#component_impl_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	ECOA:log	IN	information given to the ECOA infrastructure
Optional parameters				
	P3	ECOA:error_code	IN	contextual information about error

The value 0 shall be used as default value for error_code. Thus, if the optional parameter is not present in a language binding, the function shall behave the same way than when forcing the error_code to 0.

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

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

11.3.1.6 Raise_Fatal_Error

Interface specifying elements:

Abstract API Name	Raise_Fatal_Error			
Use Case	Provides information to ECOA infrastructure according to error category FATAL			
Level	MANDATORY (if alternative 1)			
Minimal variability pattern	#component_impl_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	ECOA:log	IN	information given to the ECOA infrastructure
Optional parameters				
	P3	ECOA:error_code	IN	contextual information about error

The value 0 shall be used as default value for error_code. Thus, if the optional parameter P3 is not present in a language binding, the function shall behave the same way than when forcing the error_code to 0.

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

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

11.3.2 Alternative 2: using flex interfaces

11.3.2.1 Flex_Log

Flex_Log generic operation allows to define a formatted string to be logged with the same approach than `printf` in C language: this string contains both ordinary characters and format specifiers, each of these specifier refers an argument that is passed as an optional parameter of the function prototype.

The formatted string is truncated by the container to 1024 characters, including a terminating null byte.

Interface specifying elements:

Abstract API Name	Flex_Log			
Use Case	Provides information to ECOA infrastructure for logging and fault management			
Level	MANDATORY (if alternative 2)			
Minimal variability pattern	#component_impl_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	ECOA:information_category	IN	information category
	P3	formatted string	IN	information given to the ECOA infrastructure as a formatted string which may require arguments
Optional parameters				
	P3 arguments			

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

```
void [#component_impl_name#_container:]flex_log([#context#,] ECOA:information_category category,
ECOA:char8* log_string,…);
```

11.3.2.2 Flex_Raise_Fatal_Error

Flex_Raise_Fatal_Error input formatted string has the same characteristics than Flex_Log one.

Interface specifying elements:

Abstract API Name	Flex_Raise_Fatal_Error			
Use Case	Provides information to ECOA infrastructure according to error category FATAL			
Level	MANDATORY (if alternative 2)			
Minimal variability pattern	#component_impl_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	formatted string	IN	information given to the ECOA infrastructure as a formatted string which may require arguments
Optional parameters				
	P2 arguments			
	P3	ECOA:error_code	IN	contextual information about error

The value 0 shall be used as default value for error_code. Thus, if the optional parameter P3 is not present in a language binding, the function shall behave the same way than when forcing the error_code to 0.

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

```
void [#component_impl_name#_container:]raise_fatal_error([#context#,] ECOA:char8* log_string,…);
```

11.4 Time Services

The Container Interface API can provide the Components with a set of functions used to access time services. Three, possibly distinct, time sources are defined:

- **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 Platform, relative to a system clock reference defined by the system integrator. **Absolute System Time** may or may not coincide with **UTC Time**. **Absolute System Time** may or may not be synchronised with other ECOA Platforms, and with non-ECOA systems.
- **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.

Each time source is available for a Component only if the Component Implementation's model requires it, through the following implementation options (refer to Metamodel, [Architecture Specification Part 7]):

- needsLocalTime
- needsSystemTime

- needsUTCTime

Thus, it is possible to identify, only looking at the models, the Components that need each kind of time reference. A Component that does not use time is more easily reusable in different contexts (e.g. simulation) than a component that needs it.

The **Relative Local 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 components with a high resolution (HR) clock which may not be synchronized with other time sources.

As a consequence, the HR clock is considered as local to a Component, and should only be used to locally compute real time durations. The HR clock (expressed with type `ECOA:hr_time` or `ECOA:nano_time`) represents the time elapsed since system start up on that CPU. It may only be considered as local to the Component, as Components may be deployed in different executables 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 software components with **UTC time**. The globally defined clock has a less precise clock, and should be used to date events.

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.

In addition, if the Component Implementation's model requires it through the implementation option "needsTimeResolution", it is possible to retrieve the time resolution for each of the time sources available to the component. The output resolution parameter contains the time resolution provided by the underlying software environment. The time resolution is the shortest duration between two updates of the associated clock. The get time resolution operations shall always return a valid value.

The following sections provide prototype definitions for the time service operations.

11.4.1 Get_Relative_Local_Time

Interface specifying elements:

Abstract API Name	Get_Relative_Local_Time			
Use Case	Returns relative local time			
Level	MANDATORY			
Minimal variability pattern	#component_impl_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	ECOA:hr_time or ECOA:nano_time	OUT	local time
Optional parameters				
	P2	#context#	IN/OUT	component context

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

```
void [#component_impl_name#_container:]get_relative_local_time([#context#,] ECOA:hr_time
*relative_local_time);
```

This operation is available only if option 'needsLocalTime' is set in the Component Implementation model.

11.4.2 Get_UTC_Time

This operation is available only if option 'needsUTCTime' is set in the Component Implementation.

Interface specifying elements:

Abstract API Name	Get_UTC_Time			
Use Case	Returns UTC Time			
Level	OPTIONAL			
Minimal variability pattern	#component_impl_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	ECOA:global_time or ECOA:nano_time	OUT	UTC time
	P2	ECOA:return_status	OUT	status on interface execution
Optional parameters				
	P3	#context#	IN/OUT	component context

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

```
ECOA:return_status [#component_impl_name#_container:]get_UTC_time([#context#], ECOA:global_time
*utc_time);
```

The ECOA platform shall be able to manage at least the following return status codes:

- `[ECOA:return_status:OK]`
  No error
- `[ECOA:return_status:CLOCK_UNSYNCHRONIZED]`
  No error – clock is unsynchronized; the local time (converted if necessary) is returned
- `[ECOA:return_status:FAILURE]`
  Any other status that is not managed by a dedicated status code, for example:
  - UTC is not available

  Zero is returned as UTC time in case of failure.

Example of other return status code that the platform may implement to specify failure cases:

```
[ECOA:return_status:OPERATION_NOT_AVAILABLE]
```

11.4.3 Get_Absolute_System_Time

This operation is available only if option 'needsSystemTime' is set in the Component Implementation model.

Interface specifying elements:

Abstract API Name	Get_Absolute_System_Time			
Use Case	Returns system time			
Level	MANDATORY			
Minimal variability pattern	#component_impl_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	ECOA:global_time or ECOA:nano_time	OUT	system time
Optional parameters				
	P2	#context#	IN/OUT	component context
	P3	ECOA:return_status	OUT	status on interface execution

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

```
ECOA:return_status [#component_impl_name#_container:]get_absolute_system_time([#context#],
ECOA:global_time *absolute_system_time);
```

When a return status is available for this operation, the ECOA platform shall at least manage the following status codes:

- `[ECOA:return_status:OK]`
  No error

- `[ECOA:return_status:CLOCK_UNSYNCHRONIZED]`
  No error – clock is unsynchronized; the local time (converted if necessary) is returned

- `[ECOA:return_status:FAILURE]`
  Any other error (such as: system time is not available); the local time (converted if necessary) is then returned.

11.4.4 Get_Relative_Local_Time_Resolution

This operation is available only if option 'needsTimeResolution' is set in the Component Implementation model.

Interface specifying elements:

Abstract API Name	Get_Relative_Local_Time_Resolution			
Use Case	Returns relative local time resolution			
Level	OPTIONAL			
Minimal variability pattern	#component_impl_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	ECOA:duration	OUT	local time resolution
Optional parameters				
	P2	#context#	IN/OUT	component context

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

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

11.4.5 Get_UTC_Time_Resolution

This operation is available only if option 'needsTimeResolution' and 'needsUTCTime' are set in the Component Implementation model.

Interface specifying elements:

Abstract API Name	Get_UTC_Time_Resolution			
Use Case	Returns UTC time resolution			
Level	OPTIONAL			
Minimal variability pattern	#component_impl_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	ECOA:duration	OUT	UTC time resolution
Optional parameters				
	P2	#context#	IN/OUT	component context

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

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

If UTC Time is not available when the operation is called, Get_UTC_Time_Resolution will return a zero `utc_time_resolution`.

11.4.6 Get_Absolute_System_Time_Resolution

This operation is available only if options ' needsTimeResolution' and 'needsSystemTime' are set in the Component Implementation model.

Interface specifying elements:

Abstract API Name	Get_Absolute_System_Time_Resolution			
Use Case	Returns system time resolution			
Level	OPTIONAL			
Minimal variability pattern	#component_impl_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	ECOA:duration	OUT	system time resolution
Optional parameters				
	P2	#context#	IN/OUT	component context

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

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

11.5 Triggers

For a Component whose ECOA model contains a 'trigger' object in its definition, two functions, methods or procedures shall be implemented by the Container. The name of the functions shall be generated to include the name of the operation.

The operation returns immediately so the calling Component is not blocked. If an infrastructure problem prevents the call from succeeding (e.g. if erroneous parameters are given), the fault is handled via the fault management infrastructure.

11.5.1 Trigger_Set

Interface specifying elements:

Abstract API Name	Trigger_Set			
Use Case	Sets a trigger from an input delay			
Level	MANDATORY			
Minimal variability pattern	#component_impl_name#; #trigger_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	ECOA:duration or ECOA:nanotime	IN	delay
	P3	ECOA:return_status	OUT	status on interface execution

Upon delay expiration, the corresponding declared event will be received by the Component.

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

```
ECOA:return_status       [#component_impl_name#_container:]#trigger_name#__Trigger_Set([#context#,]
ECOA:duration delay);
```

The ECOA platform shall be able to manage at least the following return status codes:

- `[ECOA:return_status:OK]`
  No error
- `ECOA:return_status:FAILURE]`
  Any error that is not managed by a dedicated status code

Some ECOA platforms may indeed offer complementary return status codes to provide a more accurate analysis of failure cases. For example : `[ECOA:return_status:OPERATION_ALREADY_PENDING]` in case the trigger is already set and has not expired yet.

11.5.2 Trigger_Cancel

Interface specifying elements:

Abstract API Name	Trigger_Cancel			
Use Case	Cancels a pending trigger			
Level	MANDATORY			
Minimal variability pattern	#component_impl_name#; #trigger_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	ECOA:return_status	OUT	status on interface execution

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

```
ECOA:return_status [#component_impl_name#_container:]#trigger_name#__Trigger_Cancel([#context#]);
```

This method has no effect if the trigger is no currently pending trigger.

The ECOA platform shall be able to manage at least the following return status codes:

- `[ECOA:return_status:OK]`
  No error.
- `ECOA:return_status:FAILURE]`
  No pending trigger or any error that is not managed by a dedicated status code.

11.6 Persistent Information Management (PINFO)

Persistent Information can be in the form of PINFO. PINFO makes use of the `read_#PINFOname#`, `write_#PINFOname#` , and `seek_#PINFOname#` APIs.

The following sections define the PINFO prototype definitions:

11.6.1 PINFO read

Interface specifying elements:

Abstract API Name	PINFO_Read			
Use Case	Reads a data buffer in a PINFO from an index position			
Level	MANDATORY			
Minimal variability pattern	#component_impl_name#; #PINFO_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	ECOA:byte	IN	memory address (pointer) of the memory area where read data are copied to
	P3	ECOA:uint32	IN	number of bytes to read from the current PINFO'index position
	P4	ECOA:uint32	OUT	actual number of bytes read
	P5	ECOA:return_status	OUT	status on interface execution

Some details on operation characteristics:

- P4 parameter is the actual number of bytes read, equal to the minimum of P3 and (PINFO'size – PINFO'index).

The read operation will adjust PINFO'index position to the minimum of (PINFO'index position + out_size) and PINFO'size.

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

```
ECOA:return_status [#component_impl_name#_container:]read_#PINFOname#([#context#], ECOA:byte
*memory_address, ECOA:uint32 in_size, ECOA:uint32 *out_size);
```

The ECOA platform shall be able to manage at least the following return status codes:

- `[ECOA:return_status:OK]`
  No error. PINFO data have been properly read.
- `[ECOA:return_status:FAILURE]`
  Any error that is not managed by a dedicated status code

Some ECOA platforms may indeed offer complementary return status codes to provide a more accurate analysis of failure cases. For example :

- `[ECOA:return_status:RESOURCE_NOT_AVAILABLE]`
  An infrastructure error occurred. Example: Mass memory failure.
- `[ECOA:return_status:INVALID_PARAMETER]`
  P2 is a NULL pointer or inaccessible

11.6.2 PINFO write

The **Write** operation is only available if the PINFO is declared as "writable" in the Component Type model. This declaration, in turn, is allowed only if [OPTION PINFO WRITE] si supported by the ECOA Platform).

Interface specifying elements:

Abstract API Name	PINFO_Write			
Use Case	Writes a data buffer into a PINFO from an index position			
Level	OPTIONAL			
Minimal variability pattern	#component_impl_name#; #PINFO_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	ECOA:uint32	IN	memory address (pointer) of the memory area where data to be written are taken from
	P3	ECOA:uint32	IN	number of bytes to write from the current PINFO'index position
	P4	ECOA:uint32	OUT	actual number of bytes written in PINFO
	P5	ECOA:return_status	OUT	status on interface execution

Some details on interface characteristics:

- P4 parameter is the actual number of bytes written from the current PINFO'index position, equal to the minimum of P3 and the maximal size of PINFO.

The write operation will adjust PINFO'index position to the end of the written data.

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

```
ECOA:return_status [#component_impl_name#_container:]write_#PINFOname#([#context#], ECOA:byte
*memory_address, ECOA:uint32 in_size, ECOA:uint32 *out_size);
```

The ECOA platform shall be able to manage at least the following return status codes:

- `[ECOA:return_status:OK]`
  No error. PINFO data have been properly read.
- `[ECOA:return_status:FAILURE]`
  Any error that is not managed by a dedicated status code

Some ECOA platforms may indeed offer complementary return status codes to provide a more accurate analysis of failure cases. For example :

- `[ECOA:return_status:RESOURCE_NOT_AVAILABLE]`
  An infrastructure error occurred. Example: Mass memory failure.
- `[ECOA:return_status:INVALID_PARAMETER]`
  P2 is a NULL pointer or inaccessible

11.6.3 PINFO seek

Interface specifying elements:

Abstract API Name	PINFO_Seek			
Use Case	Allows to move PINFO'index			
Level	MANDATORY			
Minimal variability pattern	#component_impl_name#; #PINFO_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	ECOA:int32	IN	offset, i.e number of bytes to be added to the selected position chosen in P3
	P3	ECOA:seek_whence_type	IN	defines the selected reference position
	P4	ECOA:uint32	OUT	new PINFO'index position
	P5	ECOA:return_status	OUT	status on interface execution

Some details on interface characteristics:

- P2 ‘offset parameter’ is an int32 so as to allow to seek backwards from the selected position in the persistent data.
- P3 ‘whence parameter’ defines how the offset is applied (SEEK_SET is relative to the start of the PINFO, SEEK_CUR is relative to PINFO’index position and SEEK_END is relative to the end of the PINFO)
- ‘new_position’ returns the value of the new PINFO’index position.

If the seek operation is successful, the PINFO’index position is adjusted to the minimum of (‘whence’ position + ‘offset’) and PINFO’size.

If the seek operation is unsuccessful, the PINFO’index position is not modified and the ‘new_position’ contains the value of the original PINFO’index position.

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

```
ECOA:return_status [#component_impl_name#_container:]seek_#PINFOname#([#context#,] ECOA:int32
offset, ECOA:seek_whence_type whence, ECOA:uint32 *new_position);
```

The ECOA platform shall be able to manage at least the following return status codes:

- `[ECOA:return_status:OK]`
  No error. PINFO'index position has been set according to the parameters.
- `[ECOA:return_status:FAILURE]`
  Any error that is not managed by a dedicated status code

Some ECOA platforms may indeed offer complementary return status codes to provide a more accurate analysis of failure cases. For example :

- `[ECOA:return_status:RESOURCE_NOT_AVAILABLE]`
  An infrastructure error occurred. Example: Mass memory failure.
- `[ECOA:return_status:INVALID_PARAMETER]`
  When SEEK_SET is chosen, the offset is outside the range of 0..PINFO'size
  When SEEK_CUR is chosen, the current PINFO'index position + 'offset' is outside the range of 0..PINFO'size
  When SEEK_END is chosen, the PINFO'size + 'offset' is outside the range of 0..PINFO'size (i.e. must be a negative offset)

11.6.4 Example of defining PINFO

PINFO attributes are implemented in the XML Metamodel as follows:

- The XML Metamodel provides a way for defining PINFO at Component Type level. The following attributes can be configured:
  - PINFO Name.
- The XML Metamodel provides a way for declaring PINFO Filename Association at Component Instance level:
  - This is done via an association between PINFO name and a filename. This file will provide the PINFO data.

The following example XML declares PINFO at Component Type level. The example shows two Component Type definitions, both of which can access a number of PINFO (both read-only and read-write PINFO):

```
<componentType>
  <pinfos>
   <pinfo name="PinfoOne" />
   <pinfo name="PinfoTwo" writable="true"/>
  </pinfos>
  <operations>
    <!-- … -->
  </operations>
</componentType>
```

The following example XML declares PINFO at Component Instance level. The example shows three Component Instances being defined and PINFO assignments being made:

```
<instance name="M11" type="M1" implementation="M1_Im">
  <pinfoValue name="PinfoOne" value="example_PINFO_1.txt"/>
  <pinfoValue name="PinfoTwo" value="example_PINFO_2.txt"/>
</instance>
```

11.7 Save Warm Start Context

This operation is available only if option 'hasWarmStartContext' is set in the Component Implementation model and if [OPTION WARM START CONTEXT] is available in the ECOA Platform.

Interface specifying elements:

Abstract API Name	Save_Warm_Context			
Use Case	Saves a context to be restored as part of a recovery action			
Level	OPTIONAL			
Minimal variability patterns	#component_impl_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context

Warm start context is a non-volatile context such that it will be restored by the ECOA Infrastructure upon a Warm Restart.

Only the latest saved version of the warm start context will be restored by the ECOA Infrastructure, according to the Platform ability to do so.

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

```
void [#component_impl_name#_container:]save_warm_start_context([#context#]);
```

11.8 Supervisor components

This section is specific to [OPTION SUPERVISION].

For SUPERVISOR components, the Container Interface API is extended with operations that can be used to control the executables and the components of the Application. It can also read and modify variables.

11.8.1 Supervision of executables

Interface specifying elements:

Abstract API Name	Get_Executable_State			
Use Case	Get the current status of a component executable			
Level	OPTIONAL			
Minimal variability pattern	#component_impl_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	ECOA:asset_id or ECOA:int32	IN	executable identifier
	P3	ECOA:executable_state	OUT	state of the executable
Optional parameters				
	P4	ECOA:return_status	OUT	status on the interface execution

Abstract API Name	Executable_Command			
Use Case	Allows to apply a command on an component executable			
Level	OPTIONAL			
Minimal variability pattern	#component_impl_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	ECOA:asset_id or ECOA:int32	IN	executable identifier
	P3	ECOA:executable_command	IN	command to be applied on the executable
Optional parameters				
	P4	ECOA:return_status	OUT	status on the interface execution

11.8.2 Supervision of components

Interface specifying elements:

Abstract API Name	Get_Component_State			
Use Case	Gets the current lifecycle state of a component instance			
Level	OPTIONAL			
Minimal variability pattern	#component_impl_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	ECOA:asset_id or ECOA:int32	IN	component instance identifier
	P3	ECOA:component_state	OUT	state of the component instance
Optional parameters				
	P4	ECOA:return_status	OUT	status on the interface execution

Abstract API Name	Component_State_Command			
Use Case	Allows to apply a lifecycle command on an component instance			
Level	OPTIONAL			
Minimal variability pattern	#component_impl_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	ECOA:asset_id or ECOA:int32	IN	executable identifier
	P3	ECOA:component_command	IN	command to be applied on the component
Optional parameters				
	P4	ECOA:return_status	OUT	status on the interface execution

11.8.3 Supervision variables

Interface specifying elements:

Abstract API Name	Get_Variable			
Use Case	Gets the current value of a supervisor component variable			
Level	OPTIONAL			
Minimal variability pattern	#component_impl_name#; #variable_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	#variable_type#	OUT	current value of the variable
	P3	ECOA:return_status	OUT	status on the interface execution

Abstract API Name	Set_Variable			
Use Case	Sets the new value of a supervisor component variable			
Level	OPTIONAL			
Minimal variability pattern	#component_impl_name#; #variable_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	#variable_type#	IN	variable value to be set
	P3	ECOA:return_status	OUT	status on the interface execution

For those two operations, the ECOA platform shall be able to manage at least the following return status codes:

- `[ECOA:return_status:OK]`
  No error.
- `[ECOA:return_status:FAILURE]`
  Any error that is not managed by a dedicated status code

Notes

- Variables are used to activate/deactivate conditional links defined in the assembly model.
- The name and type of each variable is defined in the Component Type model.
- The type of a variable #variable_type# must be based on an integer type of 32 bits or less.

12 Container Types

This section contains details of the data types that comprise the container API i.e. the data types that can be used by a component.

12.1 Versioned Data Handles

This section in the language binding will contain the language specific syntax in order to define data handles for versioned data operations defined in the Container Interface.

This definition shall respect requirements related to Data Handle Structure defined in section 11.1.2.

13 Default Values

Platforms shall initialize data to a deterministic, legal value regarding all languages when these values are initially allocated and provided by the container to the component.

It is applicable to:

- Request Response callback arguments in case of “ECOA:NO_RESPONSE” return status,
- Memory space pointed by a get_write_access for the initial access (“ECOA:DATA_NOT_INITIALIZED” return status),

The initialisation mechanism shall rely on the following rules:

- The initial value of a data shall be set according to its simple type, using the minimum boundary of the sub-range of that simple type.
- For an enumeration as well as for the select field of variant records, it shall correspond to the lowest numerical value.
- The default values of the union in a variant record shall be set according to the default value of the select field.

14 External Interface

This section only applies to ECOA Platform implementing [OPTION EXTERNAL INTEFACE] so it may not be available in some language bindings.

A Driver Component allows non-ECOA software to interact asynchronously with the ECOA System (see ECOA AS7 Part 3). The Container will provide interfaces which may be used by non-ECOA software to post an event to the Component Instance queue specified,

Interface specifying elements:

Abstract API Name	External_Event_Received			
Use Case	activates a component entry-point dedicated to the reception of non-ECOA external software data			
Level	OPTIONAL			
Minimal variability patterns	#component_impl_name#; #external_operation_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	#event_parameters#	IN	input external data

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

```
void [#component_impl_name#__]#external_operation_name#([#context#,]const #event_parameters#);
```

An external interface to a Component is specified by the use of the “external” sender notation within an associated eventLink.

This eventLink can connect the external interface to an eventReceived operation defined within the Component.

The example below shows an eventReceived “TheFeedback”, that may be used to asynchronously notify the Component of an event.

01-Components/cp_type2/cp_type2.comp.xml:

```
<componentType kind="STANDARD">
   <operations>

      <eventReceived name="TheFeedback">
         <input name="param1" type="uint32"/>
      </eventReceived>

   </operations>
</componentType>
```

01-Components/cp_type2/C/cp_type2.C.impl.xml:

```
<implementation>
   <language.c fullName=”cp_type2”>
</ implementation >
```

02-Assemblies

```
<instance name="MyComp" componentType="cp_type2" implementation="C"/>
```

An external interface may be linked to the eventReceived operation using the “external” sender, as shown below, where the external interface operation is “FeedbackLegacy”.

```
<eventLink>
   <sender><external operation="FeedbackLegacy" language="C"/></senders>
   <receiver instance="MyComp" operation="TheFeedback"/>
</eventLink>
```

This will result in an external API being provided by the Container with the following prototype definition:

```
void [#component_impl_name#__]FeedbackLegacy_External_Event_Received([#context#,] const ECOA:uint32
param1);
```

On the ECOA side, the external API permits all the normal ECOA connection flexibility for Component Event Operations.

Note: the choice of the language for generating external APIs is made separately from the choice of the language for generating ECOA modules APIs. The choice of supported languages is made depending on needs that are to be taken into account in platform procurement requirements.

15 External Components

External Components are a special kind of Components, which have an additional specific, private thread. This thread is called the "external thread".

Standard Components are entirely executed by the ECOA Infrastructure in a thread that is possibly shared with other Components of equal priority. Therefore, standard Components shall not make "blocking" calls, because it may block other Components. External Components are designed to overcome this limitation.

An external component actually behaves exactly the same as a standard component. The only difference is that each instance of external component will start an additional thread, which implementation is left to the Component's developer.

External components allow to perform specific treatments, without disturbing the execution of other Components, especially blocking calls (like reading/writing from/to a device, listening on a socket, …).

The properties of the external thread of an External Component are the following:

- It is private to the External Component Instance. Each instance will have its own thread.
- It only executes a single routine (function/procedure/method), called the "external routine", which is implemented by the External Component. The external routine has no argument except the context of the External Component Instance.
- It typically contains an infinite loop, and shall never return. If it returns, the external thread terminates.
- It can call any Container API of the External Component, with restrictions on request-response operations (detailed below).
- It can be started automatically by the ECOA infrastructure, if the option “autoStartExternalThread“ is set to 'true' in the External Component implementation model.
- It can be stopped and started by the External Component itself, using a specific container API detailed below (Start/Stop_External_Thread). Note that an External Component is not a Supervision component.
- It is independent of the External Component lifecycle, i.e. it is not stopped or destroyed when the External component is stopped or shut down.
- External routine can call any Container API of the External Component.

Interface specifying elements for external routine:

Abstract API Name	External_Routine			
Use Case	Treatments to be executed in an external thread (for external components only)			
Level	MANDATORY			
Minimal variability patterns	#component_impl_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context

The appropriate language binding will define the correct syntax for this component operation.

The following illustration provides an abstract definition of the interface to help binding elaboration:

```
void [#component_impl_name#:]external_routine ([#context#]);
```

Note: In the following, the "standard thread" denotes the thread onto which this component instance is deployed, like any other component. The "external thread" denotes the additional thread that is specially created for this component instance because it is an external component.

The communication of information from the external thread to the standard thread can be done through a sent event of the component, connected to a received event of the same component.

The synchronisation between the external thread and the standard thread, if needed, is under the responsibility of the component implementation. This synchronisation may be necessary to guarantee data consistency, for example if both threads use information from the component's user context, in a concurrent way. This synchronisation may be realised by lock-free structures, or by directly accessing the OS' mutexes (in which case the component becomes non portable on other OSes).

Since external components have the same container API as other components, they can call container functions from both the external thread and the standard thread. However:

- Calling a synchronous request-response from the external thread is not allowed, and causes a runtime error trace.
- Calling an asynchronous request-response from the external thread is allowed, but the callback corresponding to the response received by the external component will always be executed by the standard thread (like every other entry-point).

In addition to common methods that are generated for all standard ECOA components, the external component container API is enriched with dedicated functions to enable starting and stopping the external thread, directly from the standard thread.

Interface specifying elements for these functions:

Abstract API Name	Start_External_Task			
Use Case	Starts the external thread of an external component			
Level	MANDATORY			
Minimal variability pattern	#component_impl_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	ECOA:return_status	OUT	status on the interface execution

Abstract API Name	Stop_External_Task			
Use Case	Stops the external thread of an external component			
Level	MANDATORY			
Minimal variability pattern	#component_impl_name#			
Mandatory parameters	**Name**	**Abstract Type**	**IN/OUT**	**Role**
	P1	#context#	IN/OUT	component context
	P2	ECOA:return_status	OUT	status on the interface execution

For those two operations, the ECOA platform shall be able to manage at least the following return status codes:

- `[ECOA:return_status:OK]`
  No error.
- `[ECOA:return_status:FAILURE]`
  Any error that is not managed by a dedicated status code

For a given component instance, there is only one instance of the external thread, so any call to start_external_thread() when it is already started will fail. A dedicated status code may be defined for that case : `[ECOA:OPERATION_NOT_AVAILABLE]`.

16 TriggerManager components

16.1 PeriodicTriggerManager Components

PeriodicTriggerManager components require a Component Type to be specified, as any other component; however, the implementation is implicit and managed by the infrastructure.

A PeriodicTriggerManager Component Type defines a list of periodic events to be sent. These events will be sent, periodically, while the component instance is in RUNNING state.

A PeriodicTriggerManager Component Type has the following specificities:

- Every operation must be an eventSent operation, without any parameters.
- Every operation must define an attribute 'period', in order to define the period of the periodic event to be sent by the PeriodicTriggerManager component.
- Every operation can define an attribute 'delay', in order to define the initial delay of the periodic event to be sent by the PeriodicTriggerManager component. This delay is counted from the moment the component is started.
- The component shall define no property, and no PINFO.

The following XML shows an example of a PeriodicTriggerManager Component definition:

```
<componentType>
 <operations>
  <eventSent name="out1" period="100" />
  <eventSent name="out2" period="20" delay="100" />
 </operations>
</componentType>
```

In this example, the component sends 2 period events, one every 100ms, the other every 20ms.

Since the event 'out2' has a delay of 100ms, the first occurence of 'out2' will occur at the same time the second occurence of 'out1' is sent.

- times for out1: 0, 100, 200, 300,...
- times for out2: 100, 120, 140,...

Since the two events are defined with the same component, they are based on the same time reference, and will stay synchronised (i.e. they will occur simultaneoulsy for 1 out of 5 occurences of 'out2') forever, even if the PeriodicTriggerManager is stopped and restarted.

16.2 DynamicTriggerManager Components

When DYNAMIC_TRIGGER option is available, DynamicTriggerManager components can be defined using a dedicated Component Type. The implementation DynamicTriggerManager components is implicit and managed by the infrastructure. See ECOA AS7 Part7 for specification of such a component in an ECOA model.

17 Reference Language Header

This section in the language binding contains a reference header for the specific language containing the Basic and Predefined ECOA types defined within the ECOA predefined library.