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

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

Issue: 6

Prepared by
BAE Systems (Operations) Limited and Dassault Aviation

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

Contents

Figures

Tables

0 Introduction

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

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

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

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

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

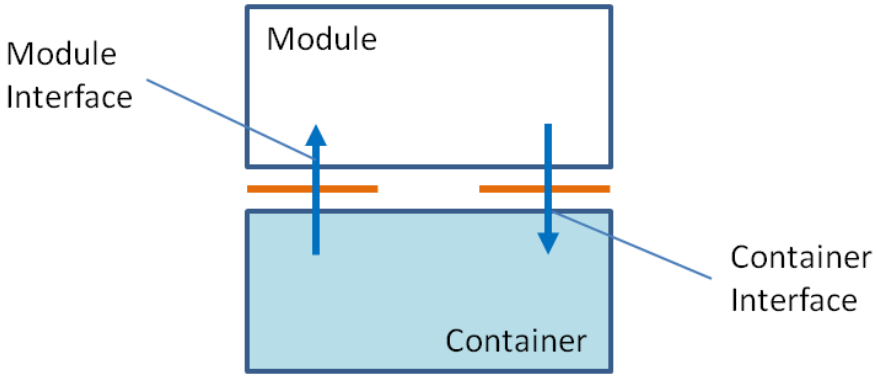

Figure 1 Module and Container Interface

Different bindings provide mappings for particular programming languages. Currently four language bindings are available: for C [Architecture Specification Part 8], C++ [Architecture Specification Part 9], Ada [Architecture Specification Part 10] and High Integrity Ada [Architecture Specification Part 11].

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

The information in this document is based on v2.0.0 of the ECOA® meta-model.

This document is structured as follows:

- Section 6 describes the Module to Language Mapping
- Section 7 describes the Parameters for operations
- Section 8 describes the Module Context
- Section 9 describes the Type libraries
- Section 10 describes the Module Interface
- Section 11 describes the Container Interface
- Section 12 describes the Container Types
- Section 13 describes the External Interface
- Section 14 describes the Default Values

- Section 15 describes Trigger Instances
- Section 16 describes Dynamic Trigger Instances
- Section 17 describes Reference Language Headers

1 Scope

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

2 Warning

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

3 Normative References

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

Architecture Specification Part 10	IAWG-ECOA-TR-003 / DGT 144476 Issue 6 Architecture Specification Part 10 – Ada Language Binding
Architecture Specification Part 11	IAWG-ECOA-TR-031 / DGT 154934 Issue 6 Architecture Specification Part 11 – High Integrity Ada Language Binding

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

4 Definitions

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

5 Abbreviations

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CPU	Central Processing Unit
ECOA	European Component Oriented Architecture. ECOA® is a registered trademark.
ELI	ECOA® Logical Interface
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 Module to Language Mapping

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

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

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

- **#component_impl_name#** is the name of the component implementation – the name is used for API generation.
- **#module_impl_name#** is the name of the module implementation – the name is used for API generation.
- **#module_instance_name#** is the name of a Module Instance – this name is used for deployment purposes,
- **#operation_name#** is the name of the module 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],
- **#context#** will be used to represent the reference to the context associated with a Module Instance.
- **#list_of_event_operations_specifications#** correspond to operations defined in section 10.1.3
- **#list_of_request_response_operations_specifications#** correspond to operations defined in section 10.1.1
- **#list_of_versioned_data_notifying_operations_specifications#** correspond to operations defined in section 10.1.2
- **#error_notification_operation_specification#** correspond to operations defined in section 10.3
- **#list_of_event_operations#** correspond to operations defined in section 10.1.3
- **#list_of_request_response_operations#** correspond to operations defined in section 10.1.1
- **#list_of_versioned_data_notifying_operations#** correspond to operations defined in section 10.1.2
- **#list_of_lifecycle_operations#** correspond to operations defined in section 10.2

---

[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#`*.

- **#event_operation_call_specifications#** correspond to operations defined in section 11.1.3
- **#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
- **#recovery_action_call_specification#** correspond to operations defined in section 11.6
- **#PINFO_read_call_specifications#** correspond to operations defined in section 11.5.1
- **#PINFO_seek_call_specifications#** correspond to operations defined in section 11.5.2
- **#Save_Warm_Start_Context_operation#** correspond to operations defined in section 11.7

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

Table 1 Module and Container Interfaces

Category	Abstract API Name	Container Operation	Module Operation	Section
Events API	Event_Send	Yes	No	11.1.3
	Event_Received	No	Yes	10.1.3
Request Response API	Request_Sync	Yes	No	11.1.1.2
	Request_Async	Yes	No	11.1.1.3
	Request_Received	No	Yes	10.1.1.1
	Response_Received	No	Yes	10.1.1.2
	Response_Send	Yes	No	11.1.1.1
Versioned Data API	Get_Read_Access	Yes	No	11.1.2.1
	Release_Read_Access	Yes	No	11.1.2.2
	Updated	No	Yes	10.1.2
	Get_Write_Access	Yes	No	11.1.2.3
	Cancel_Write_Access	Yes	No	11.1.2.4
	Publish_Write_Access	Yes	No	11.1.2.5
Properties API	Get_Value	Yes	No	11.2.1
Runtime Lifecycle API	Initialize_Received	No	Yes	10.2
	Start_Received	No	Yes	
	Stop_Received	No	Yes	
	Shutdown_Received	No	Yes	
Logging and Fault Management Services API	Log_Debug	Yes	No	11.3
	Log_Trace	Yes	No	
	Log_Info	Yes	No	
	Log_Warning	Yes	No	
	Raise_Error	Yes	No	
	Raise_Fatal_Error	Yes	No	
Time Services API	Get_Relative_Local_Time	Yes	No	11.4
	Get_UTC_Time	Yes	No	
	Get_Absolute_System_Time	Yes	No	

Category	Abstract API Name	Container Operation	Module Operation	Section
Fault Handling API (Fault Handlers Only)	Error_Notification	No	Yes	10.3
	Recovery_Action	Yes	No	11.6
Persistent Information (PINFO) Management	Read	Yes	No	11.5.1
	Seek	Yes	No	11.5.2
Context Management	Save_Warm_Start_Context	Yes	No	11.7

7 Parameters

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

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

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

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

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

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 (10.1.1.2) functions.

8 Module Context

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

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

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

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

The part of the Module Context related to user-defined local private data is optional. It will be made available for a given Module 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 Module Context which holds the infrastructure-level technical and specific data is not optional.

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

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

The language bindings specify the exact syntax required for the Module User Context and Warm Start Context, as well as the syntax for declaring any infrastructure-level technical and specific data in the Module 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 Module and Container Interfaces. Namespaces are used to organise the types into separate libraries.

9.1 Namespaces

Namespaces are used to organise the types used by an ECOA system into disjoint sets, or libraries. The namespaces are organised in a hierarchical manner, and all of the ECOA namespaces are subordinate to the ECOA base namespace as shown in Figure 2. Application based namespaces that are not subordinate to the ECOA namespace are also allowed.

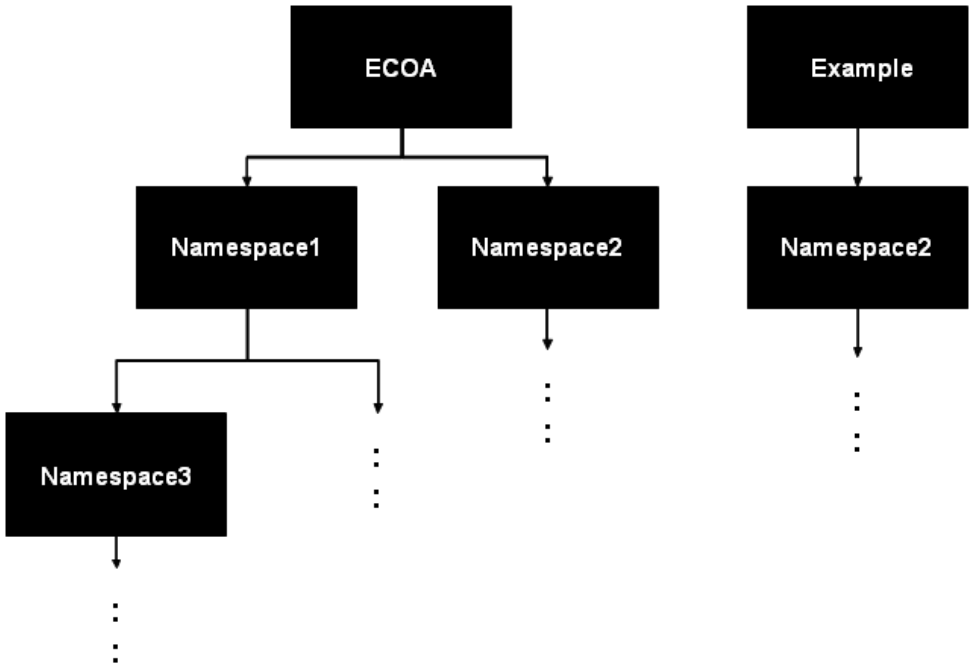

Figure 2 Namespaces

Type names within the same namespace shall be unique. All types declared in the same namespace 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:

```
#namespace1#[__#namespacen#].types.xml
```

When applied to the examples of Figure 2, it would be:

- ECOA__Namespace1__Namespace3.types.xml
- ECOA__Namespace2.types.xml
- Example__Namespace2.types.xml

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

9.2 Basic Types

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

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

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` and `ECOA:uint64` are only available on platforms which support 64bit integer arithmetic. A dedicated flag `ECOA_64BIT_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 Basic Type	Constant Name	Constant Value
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
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 these will be implemented as macros. These are also defined in the base namespace.

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#" />
```

[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.

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 Constants

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

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

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

EXAMPLE 1 defining a constant of type `ECOA:uint32`:

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

EXAMPLE 2 defining a constant of type `ECOA:double64`:

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

EXAMPLE 3 defining a constant of type `ECOA:char8`:

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

EXAMPLE 4 defining a constant of type `ECOA:char8 (hexadecimal)`:

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

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

%constant_name%.

EXAMPLE 3 using a constant to bound an array:

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

9.3.3 Enumerations

An enumeration type is the definition of a set of labels, derived from a pre-defined type, with optional values or integer-based constant definitions. 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#`.

9.3.4 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>
```

9.3.5 Variant Records

Variant Record types

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

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

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>
```

9.3.6 Fixed Arrays

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

EXAMPLE defining a fixed array:

```
<fixedarray name="#array_type_name#" itemType="#type_name#"
maxNumber="#uint32_constant#" />
```

9.3.7 Variable Arrays

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

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

EXAMPLE defining a variable array:

```
<array name="#array_type_name#" itemType="#type_name#"
maxNumber="#uint32_constant#" />
```

9.4 Predefined Types

9.4.1 ECOA:return_status

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

ECOA:OK = 0	No error has occurred
ECOA:INVALID_HANDLE = 1	An invalid handle has been used
ECOA:DATA_NOT_INITIALIZED = 2	The data has never been written
ECOA:NO_DATA = 3	The call is not able to provide any data
ECOA:INVALID_IDENTIFIER = 4	An invalid ID has been used.
ECOA:NO_RESPONSE = 5	No response was received for a request
ECOA:OPERATION_ALREADY_PENDING = 6	The requested operation is already being processed
ECOA:CLOCK_UNSYNCHRONIZED = 7	The clock is not synchronised
ECOA:RESOURCE_NOT_AVAILABLE = 8	Insufficient resource is available to perform the operation.
ECOA:OPERATION_NOT_AVAILABLE = 9	The requested operation is not available.
ECOA:INVALID_PARAMETER = 10	An invalid parameter has been used

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

```
<enum name="return_status" type="uint32">
   <value name="OK" valnum="0"/>
   <value name="INVALID_HANDLE" valnum="1" />
   <value name="DATA_NOT_INITIALIZED" valnum="2" />
   <value name="NO_DATA" valnum="3" />
   <value name="INVALID_IDENTIFIER" valnum="4" />
   <value name="NO_RESPONSE" valnum="5" />
   <value name="OPERATION_ALREADY_PENDING" valnum="6" />
   <value name="CLOCK_UNSYNCHRONIZED" valnum="7" />
   <value name="RESOURCE_NOT_AVAILABLE" valnum="8" />
   <value name="OPERATION_NOT_AVAILABLE" valnum="9" />
      <value name="INVALID_PARAMETER" valnum="10" />
</enum>
```

9.4.2 ECOA:hr_time

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

- **`ECOA:uint32 seconds`.** Seconds elapsed since some reference point in time. The value shall be positive.
- **`ECOA:uint32 nanoseconds`**. Nanoseconds measured within the current second. The value shall be between 0 and 999999999.

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

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

9.4.3 ECOA:global_time

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

- **`ECOA:uint32 seconds`**. Seconds elapsed since the POSIX Epoch (1st of January, 1970). The value shall be positive.
- **`ECOA:uint32 nanoseconds`**. Nanoseconds measured within the current second. The value shall be between 0 and 999999999.

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

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

9.4.4 ECOA:duration

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

- **`ECOA:uint32 seconds`**. The value shall be positive.
- **`ECOA:uint32 nanoseconds`**. Nanoseconds measured within the current second. The value shall be between 0 and 999999999.

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

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

9.4.5 ECOA:log

`ECOA:log` is a variable array of 256 **`ECOA:char8`** elements, that defines how a fault or information report is stored. The type is constrained to enable portability, because some implementations may not be able to support unconstrained logging. See Section 11.3 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 section9.3.7) defined in the ECOA namespace as follows:

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

9.4.6 ECOA:error_id

The `ECOA:error_id` is a 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.

9.4.7 ECOA:error_code

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

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

Error codes may be provided to the ECOA Fault Handler by Module 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 ECOA Fault Handler.

The functional meaning of error code values is not standardised by ECOA. Each ASC supplier is responsible for specifying the meaning of error codes raised by their Module Instances. Each ECOA Platform supplier is responsible for specifying the meaning of error codes raised by their ECOA Platform.

9.4.8 ECOA:asset_id

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

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

Asset IDs uniquely identify entities involved in the fault management, i.e.:

- Entities for which it is possible to detect an error: component instance, protection domain, computing node, computing platform and service operation.
- Entities on which it is possible to require a recovery action: component instance, protection domain, computing node, computing platform or deployment.

The Fault Handler and the Infrastructure need to share the same IDs since they are used to describe at Fault Handler level errors raised by the Infrastructure and to carry on by the Infrastructure recovery actions decided by the Fault Handler.

The platform tooling will 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 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` namespace 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_#protection_domain_name1#" type="ECOA:asset_id"
value="#PD_ID1#" />
   <constant name="PD_#protection_domain_name2#" type="ECOA:asset_id"
value="#PD_ID2#" />
   <!-- ... -->
   <constant name="PD_#protection_domain_nameN#" type="ECOA:asset_id"
value="#PD_IDN#" />

   <constant name="NOD_#computing_node_name1#" type="ECOA:asset_id"
value="#NOD_ID1#" />
   <constant name="NOD_#computing_node_name2#" type="ECOA:asset_id"
value="#NOD_ID2#" />
   <!-- ... -->
   <constant name="NOD_#computing_node_nameN#" type="ECOA:asset_id"
value="#NOD_IDN#" />

   <constant name="PF_#computing_platform_name1#" type="ECOA:asset_id"
value="#PF_ID1#" />
   <constant name="PF_#computing_platform_name2#" type="ECOA:asset_id"
value="#PF_ID2#" />
   <!-- ... -->
   <constant name="PF_#computing_platform_nameN#" type="ECOA:asset_id"
value="#PF_IDN#" />

   <constant name="SOP_#service_operation_name1#" type="ECOA:asset_id"
value="#SOP_ID1#" />
   <constant name="SOP_#service_operation_name2#" type="ECOA:asset_id"
value="#SOP_ID2#" />
```

```
   <!-- ... -->
   <constant name="SOP_#service_operation_nameN#" type="ECOA:asset_id"
value="#SOP_IDN#" />

   <constant name="DEP_#deployment_name1#" type="ECOA:asset_id"
value="#DEP_ID1#" />
   <constant name="DEP_#deployment_name2#" type="ECOA:asset_id"
value="#DEP_ID2#" />
   <!-- ... -->
   <constant name="DEP_#deployment_nameN#" type="ECOA:asset_id"
value="#DEP_IDN#" />
```

NOTE "deployment_name" is the filename of the deployment XML file.

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

Names for service operations are generated by concatenation as follows:

- [ComponentInstanceName]_[ServiceOrReferenceInstanceName]_[ServiceOperationName]

9.4.9 ECOA:asset_type

The data type `ECOA:asset_type` is an enumeration declared in the ECOA namespace, which is used to identify the type of asset either linked to an error or targeted by a recovery action. The enumeration values are:

COMPONENT	0	Component instance
PROTECTION_DOMAIN	1	Protection Domain
NODE	2	Computing Node
PLATFORM	3	Computing Platform
SERVICE	4	Service – used for service operation errors
DEPLOYMENT	5	Deployment – used for reloading the platform

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

```
<enum name="asset_type" type="uint32">
   <value name="COMPONENT" valnum="0" />
   <value name="PROTECTION_DOMAIN" valnum="1" />
   <value name="NODE" valnum="2" />
   <value name="PLATFORM" valnum="3" />
   <value name="SERVICE" valnum="4" />
   <value name="DEPLOYMENT" valnum="5" />
</enum>
```

9.4.10 ECOA:error_type

The data type `ECOA:error_type` is an enumeration declared in the ECOA namespace, which is used to specify the type of the error. The enumeration values are given by the table below. For each error, a short description is given as well as the potential origin of the error, i.e. the asset linked to the error.

Table 4 Table of Errors

Error	Value	Description	ECOA asset linked to the error				
			Component	Protection Domain	Computing Node	Computing Platform	Service Operation
RESOURCE_NOT_AVAILABLE	0	No more resources to carry on the element activities	X	X	X	X	
UNAVAILABLE	1	The element (potentially a remote platform) has disappeared for an unknown reason	X	X	X	X	
MEMORY_VIOLATION	2	Memory violation	X	X			
NUMERICAL_ERROR	3	Divide by zero or floating-point error	X	X			
ILLEGAL_INSTRUCTION	4	Illegal instruction in the binary code	X	X			
STACK_OVERFLOW	5	Module or protection domain stack corruption	X	X			
DEADLINE_VIOLATION	6	Module deadline violation	X				
OVERFLOW	7	The module queue is full or if the container has not enough resources to track concurrent requests.	X				
UNDERFLOW	8	The module queue is not enough fulfilled	X				
ILLEGAL_INPUT_ARGS	9	Illegal input arguments	X				
ILLEGAL_OUTPUT_ARGS	10	Illegal output arguments	X				
ERROR	11	Raise_error called by a Module	X				
FATAL_ERROR	12	Raise_fatal_error called by a Module	X				
HARDWARE_FAULT	13	Hardware fault			X	X	
POWER_FAIL	14	Power failure			X	X	
COMMUNICATION_ERROR	15	Communication error			X	X	
INVALID_CONFIG	16	Invalid configuration. The node is not able to load the configuration			X	X	
INITIALISATION_PROBLEM	17	Initialisation problem. The node is not able to allocate resources or to start components			X	X	
CLOCK_UNSYNCHRONIZED	18	The node clock is not synchronized with the other parts of the system.			X	X	
UNKNOWN_OPERATION	19	The client receives the ELI UNKNOWN_OPERATION return code when invoking a remote service operation					X
OPERATION_OVERRATED	20	The operation is called more than expected by the QoS.					X
OPERATION_UNDERRATED	21	The operation is called less than expected by the QoS.					X

The ECOA:error_type is an enumeration (see section 9.3.3) 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" />
   <value name="UNKNOWN_OPERATION" valnum="19" />
   <value name="OPERATION_OVERRATED" valnum="20" />
   <value name="OPERATION_UNDERRATED" valnum="21" />
</enum>
```

9.4.11 ECOA:recovery_action_type

The data type ECOA:recovery_action_type is an enumeration declared in the ECOA namespace, which is used to specify the type of action recovery the Infrastructure will carry on the target asset (i.e. on the target component, protection domain, node or platform). The enumeration values are:

SHUTDOWN	0	Shutdown the target asset
COLD_RESTART	1	Restart the target asset in cold mode
WARM_RESTART	2	Restart the target asset in warm mode
CHANGE_DEPLOYMENT	3	Reload the platform with a new deployment

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

```
<enum name="recovery_action_type" type="uint32">
   <value name="SHUTDOWN" valnum="0" />
   <value name="COLD_RESTART" valnum="1" />
   <value name="WARM_RESTART" valnum="2" />
   <value name="CHANGE_DEPLOYMENT" valnum="3" />
```

```
</enum>
```

9.4.12 ECOA:pinfo_filename

The data type `ECOA:pinfo_filename` is a variable array declared in the ECOA namespace. This type is used to specify where the file containing public PINFO data is stored before the actual deployment on the target ECOA Platform.

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

9.4.13 ECOA:seek_whence_type

The data type `ECOA:seek_whence_type` is an enumeration declared in the ECOA namespace, which is used to define the position to consider in a PINFO, when invoking the Seek operation. The enumeration values are:

SEEK_SET	0	Position is the beginning of the PINFO
SEEK_CUR	1	Position is the current PINFO'index
SEEK_END	2	Position is the end of the PINFO

The `ECOA:seek_whence_type` is an enumeration (see section 9.3.3) defined in the ECOA namespace 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>
```

10 Module Interface

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

10.1 Operations

The Module Interface provides a number of entry points that allow the Container to invoke Module Operations that cause a Module Instance to execute a block of functionality. The block of functionality may call any container operation API allowed by its type (see section 11).

10.1.1 Request-Response

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

10.1.1.1 Request Received

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

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

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

10.1.1.2 Response Received

For a Module declared as client of an asynchronous request-response operation, a function is implemented by the Module to handle the response generated by the server Module. The `ID` parameter is provided by the infrastructure to allow the Module Instance to associate the response with the request (see 11.1.1.3). This is required because the module 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. The name of the function shall be generated to include the name of the operation.

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

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

Response Received operations may return the following status codes:

- `[ECOA:return_status:OK]`
  - No error
- `[ECOA:return_status:NO_RESPONSE]`
  - No response received within the expected time
  - The server module queue is full
  - Server module is IDLE/STOPPED
  - Server has called `raise_fatal_error()`
  - The operation is not connected to a RequestLink

The NO_RESPONSE status code hides several kinds of problem; depending on the problem, the infrastructure may implement several mechanisms to quicken the call of this function.

10.1.2 Versioned Data Updated

The Updated module operation is a callback used by the Container to notify a module when a new value of Versioned data is available. Once notified, the Module 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.

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

The following is a prototype definition for the operation:

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

10.1.3 Event Received

For a Module declared as a handler of an event, a function, method or procedure shall be implemented by the Module to handle the reception of the event from all possible senders. 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 appropriate language binding will define the correct syntax for this module operation, but the abstract format is given below:

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

10.2 Module Lifecycle

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

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

Operations are provided by the Module Interface to support the following Module Lifecycle functionality. These operations are applicable to all a Module Instances (“normal” modules, trigger and dynamic trigger).

- **INITIALIZE_Received**: this is the initialisation entry-point of the module used to perform its local initialisation; the Initialise entry-point of a Module is the function in which the Module is supposed to initialise all its local variables (user context).
- **START_Received**: this is the entry point which is used for starting a Module Instance from a technical point of view. Once this entry point has been completed, the Module 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 Module as part of a graceful shutdown procedure in order to change the Module state from RUNNING to READY
- **SHUTDOWN_Received**: this event may be sent to the Module as part of a graceful shutdown procedure in order to change the Module state from READY or RUNNING to IDLE

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

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

Within these four operations the module 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 modules. The module may still call any other container operation API allowed by its type (see section 11).

10.3 Error notification at fault handler level

The Fault Handler Module Interface provides error handling functionality that may be used by the platform to provide information to a Fault Handler when an error occurs.

The following is an abstract description of the operation:

```
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 Module Instances.

The parameter `#context#` represents a reference to the context associated with a Module Instance where the Fault Handler is implemented as an ECOA Component. Where the Fault Handler is implemented as part of the Platform Infrastructure it represents a reference to a structure which follows the same template as that used for a Module Instance context.

The parameter `error_id` identifies uniquely the error in the scope of the notified Fault Handler.

The parameter `timestamp` is the time at which the error has been initially detected.

The parameter `asset_id` identifies the asset linked to the error. The ID is unique for a given asset type.

The parameter `asset_type` identifies the type of asset linked to the error: component instance, protection domain, computing node, computing platform or service operation.

The parameter `error_type` provides the type of error raised.

The parameter `error_code` is information provided by the asset that raised the error. This may be used to pass more detailed information to the Fault Handler.

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 recovery action (11.6) or any log function (11.3).

11 Container Interface

11.1 Operations

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

11.1.1 Request Response

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

- Synchronous Request
- Asynchronous Request

A further operation, Response Send is provided to return the result to the requesting module.

11.1.1.1 Response Send

An operation provided by the Container, used by the Module to send a Response. The `ID` parameter is was provided by the infrastructure in the associated Request Received operation and enables the Module 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.

The following is a prototype definition for the operation:

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

Response operations may return the following status codes:

- `[ECOA:return_status:OK]`
  - No error
- `[ECOA:return_status:INVALID_IDENTIFIER]`
  - API called with an invalid request-response identifier

11.1.1.2 Synchronous Request

An operation provided by the Container, used by a Module to invoke an operation provided by a server Module. The `#request_parameters#` correspond to the "in" parameters of the request-response. The `#response_parameters#` correspond to the "out" parameters of the request-response. The calling Module is blocked until the response is received.

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

The following is a prototype definition for the operation:

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

Synchronous Request operations may return the following status codes:

- `[ECOA:return_status:OK]`
  - No error
- `[ECOA:return_status:NO_RESPONSE]`
  - No response received within the expected time
  - The server module queue is full
  - Server module is IDLE/STOPPED
  - Server has called `raise_fatal_error()`
  - Container unable to send request (including if the operation is not connected to a RequestLink)

The NO_RESPONSE status code covers several issues; and, depending on the problem, the infrastructure may implement several mechanisms to accelerate the response of this function.

11.1.1.3 Asynchronous Request

An operation provided by the Container, used by a Module to invoke an operation provided by a server Module. The `ID` parameter is provided by the infrastructure to allow the Module Instance to associate the response with the request (see 10.1.1.2). This `ID` is unique for each Module Instance and for each call of the operation (because the module could initiate multiple requests prior to receiving any responses). The `#request_parameters#` correspond to the "in" parameters of the request-response.

The operation returns immediately so the calling Module 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 module is authorized to perform has been reached and the module invokes another asynchronous request, the container shall not proceed the request and return RESOURCE_NOT_AVAILABLE to the module.

Asynchronous Request operations may return the following status codes:

- `[ECOA:return_status:OK]`
  - No error
- `[ECOA:return_status:RESOURCE_NOT_AVAILABLE]`
  - maxConcurrentRequests has been exceeded

The following is a prototype definition for the operation:

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

11.1.2 Versioned Data

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

- Get (request) Read Access
- Release Read Access
- Get (request) Write Access
- Cancel Write Access (without writing new data)
- Publish (write) new data (automatically releases write access)

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

A Data Handle structure contains the following fields:

- An attribute used to provide access to a local copy of the data
- 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)
- A platform hook, which is opaque to the user, and used by the ECOA infrastructure to handle that data

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

The following is a prototype definition for a Data Handle

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

11.1.2.1 Get_Read_Access

For a Module declared as a reader of a Versioned Data, the container shall provide a function to get read access to the Versioned Data. This operation shall output the Data Handle parameter that allows the subsequent code to access the data space:

- 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 Module 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 Module 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. The status code ECOA:NO_DATA is returned if the provider has never published an initial value. In this case the Data Handle will contain a null pointer and the stamp will be equal to zero.

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.

The following is a prototype definition for the operation:

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

Get Read Access operations may return the following status codes:

- `[ECOA:return_status:OK]`
    - No error
- `[ECOA:return_status:NO_DATA]`
    - The data has never been written (including if the operation is not connected to a DataLink)
- `[ECOA:return_status:INVALID_HANDLE]`
    - API called with an invalid versioned data handle

- [ECOA:return_status:RESOURCE_NOT_AVAILABLE]
  - Maximum number of versioned data reached
  - Container unable to provide a versioned data

11.1.2.2 Release_Read_Access

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

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.

The following is a prototype definition for the operation:

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

Release Read Access operations may return the following status codes:

- [ECOA:return_status:OK]
  - No error
- [ECOA:return_status:INVALID_HANDLE]
  - API called with an invalid versioned data handle

11.1.2.3 Get_Write_Access

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

- 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 data has never been written, Get_Write_Access returns the status code ECOA:DATA_NOT_INITIALIZED but returns a valid data handle towards a valid memory area.

If there is an infrastructure problem that prevents the API from succeeding, another error indication (RESOURCE_NOT_AVAILABLE, etc) is returned to the caller and the infrastructure handles the fault via the fault management infrastructure.

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

The following is a prototype definition for the operation:

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

Get Write Access operations may return the following status codes:

- [ECOA:return_status:OK]
  - No error
- [ECOA:return_status:DATA_NOT_INITIALIZED]
  - No error – the data has never been written
- [ECOA:return_status:INVALID_HANDLE]
  - API called with an invalid versioned data handle
- [ECOA:return_status:RESOURCE_NOT_AVAILABLE]
  - Maximum number of versioned data reached
  - Container unable to provide versioned data

11.1.2.4 Cancel_Write_Access

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

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.

The following is a prototype definition for the operation:

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

Cancel Write Access operations may return the following status codes:

- [ECOA:return_status:OK]
  - No error
- [ECOA:return_status:INVALID_HANDLE]
  - API called with an invalid versioned data handle

11.1.2.5 Publish_Write_Access

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

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 module 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 module is reusing a handle already released). Any other fault is handled by the infrastructure.

The following is an abstract definition of the operation:

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

Publish Write Access operations may return the following status codes:

- `[ECOA:return_status:OK]`
  - No error
- `[ECOA:return_status:INVALID_HANDLE]`
  - API called with an invalid versioned data handle

11.1.3 Event Send

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

The following is a prototype definition for the operation:

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

11.2 Properties

The Container Interface may include operations which allow a Module to access properties at runtime. Module properties are defined for a Module Type; the value is assigned for each Module Instance. The Module 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 Module Instance Property Value as detailed in section 11.2.2.

This mechanism allows different instances of Modules to have access to property values specified at the Module 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 Module Instances to get read only access to the properties. The abstract format of the interface is:

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

Where:

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

11.2.2 Expressing Property Values

Values given to properties are set in Component 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 Module Instance Property Value
  - To reference a Component Property from a Module Instance Property Value, the ‘$’ sign shall be used to prefix the name of the Component Property as follows:
    - $#component_property_name#
- Syntax to refer to an Assembly Property from an Component Instance Property Value
  - To reference an Assembly Property from a Component Instance Property value, the ‘$’ sign shall be used to prefix the name of the Assembly Property in the SCA source attribute as follows:
    - <csa:property name="#component_property_name#" source="$#assembly_property_name#" />
  - Reminder: by definition, artefacts within the Component Implementation scope, such as Module Instance Property Values cannot reference Assembly 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>
   <service .../>
   <reference .../>
   <property name="Update_Rate" ecoa-sca:type="float32"/>
   <property name="Component_Property1" ecoa-sca:type="float32"/>
</componentType>
```

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

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

```
</moduleType>

...

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

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

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

```
<csa:component name="example">
   <ecoa-sca:instance componentType="example_instance">
   <ecoa-sca:implementation name="example_component"/>
   </ecoa-sca:instance>
   <csa:property name="Update_Rate"><csa:value>10.0</csa:value></csa:property>
   <csa:property name="Component_Property1" source="$Assembly_Property1"/>
</csa:component>
```

Assembly Property Values are defined in the system assembly schema (.impl.composite):

```
<csa:property name="Assembly_Property1" ecoa-sca:type="ECOA:uint32">
   <csa:value>10</csa:value>
</csa:property>
```

The above example would generate two Get_Value APIs:

```
void [example_mod_impl_container:]get_Update_Rate_value([#context#,] ECOA:float32* value);
void [example_mod_impl_container:]get_Module_Inst_Prop_value([#context#,] ECOA:uint32* value);
```

For the component instance “example_instance” the get_Update_Rate_value API would return 10.0 for both the “example_mod_inst1” and “example_mod_inst2” Module Instances. However the

get_Module_Inst_Prop_value API would return 20 for the “example_mod_inst1” Module Instance, but 2 for the “example_mod_inst2” Module Instance.

11.3 Logging and Fault Management

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

- **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.	Module shall be shutdown by the infrastructure and fault is reported to the fault management infrastructure. Information is logged. If a fatal error is reported by a Module, the infrastructure will shutdown all modules of the component. If a fatal error is reported by the ECOA Fault Handler, then: If there is only one ECOA Fault Handler instance on the platform (i.e. the one which raised the fatal error), the infrastructure will shutdown all protection domains associated with that Fault Handler. Else, another ECOA Fault Handler instance may handle the error.	No
ERROR	Used by the application to raise errors from which the application may be able to recover, with assistance.	The fault management infrastructure shall filter these errors to determine whether the Module is to be shutdown or not. Information is logged.	No
WARNING	Used by the application to log runtime issues which are undesirable or unexpected, but not necessarily "wrong". Useful for non-intrusive analysis. The Module Instance performing the log is not stopped (i.e. continues execution).	Information is logged.	Yes

Category	Definition	Infrastructure Response	Maskable Within the Deployment Schema
INFO	Used by the application to log runtime events (e.g. startup/shutdown). Useful for non-intrusive analysis. The current Module 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

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`.

The log levels are configured in the Deployment where certain categories may be masked, except for ERROR and FATAL.

The following shows how the logPolicy is captured in the Deployment XML.

```
  <logPolicy>
    <componentLog instanceName="component_instance_name"
enabledLevels="enabled_levels_mask">
      <moduleLog instanceName="module_instance_name"
enabledLevels="enabled_levels_mask"/>
    </componentLog>
  </logPolicy>
```

Where enabled_levels_mask is a list of logging levels (either WARN, DEBUG, INFO or TRACE) separated with "|".

Log level configuration information defined at Component Instance level is applied for any Module Instances where no configuration information is present in the Deployment. All log levels are enabled where no configuration information is given at the Component Instance level.

The actual log output configuration is platform dependant, and may be assigned to text files; flash disk etc. depending on the platform capabilities. However, the platform solution shall provide means to retrieve log output of Module Instances in the following format:

- No header line
- Each log message shall have the following format :

  `<SYSTEM_TIME>:<IS_UTC>:<MESSAGE_LEVEL>:<COMPONENT_INSTANCE_NAME>:<MODULE_INSTANCE_NAME>:<MESSAGE_TEXT><EOL>`
- Where:
  - <SYSTEM_TIME> is a string-typed field containing the log message writing date (t), according to a common ECOA system time reference.

    It is defined by a two values: the first value (s) is an integer expressing a number of seconds, while the second value (n) corresponds to time residue expressed in nanoseconds, so that t=s + (n/10E9) in seconds.

<SYSTEM_TIME> values are separated by a comma; the two values are written in quotation marks.

- <IS_UTC> is an integer representing a Boolean value :
  - Value is 0 when the time is local time, but expressed relative to Unix Epoch
  - Value is 1 if time is expressed using UTC timescale.
- <MESSAGE_LEVEL> is a string-typed field indicating the criticality level of the log message, either FATAL, ERROR, WARNING, DEBUG, INFO or TRACE. This field is written in quotation marks.
- <COMPONENT_INSTANCE_NAME> is a string-typed field containing the name of the Component Instance which produced the log (in accordance with Component Instance names defined in the final assembly and deployment XML files). This field is written in quotation marks.
- <MODULE_INSTANCE_NAME> is a string-typed field containing the name of the Module Instance which produced the log (in accordance with Module Instance names defined in the component implementation and deployment XML files). This field is written in quotation marks.
- <MESSAGE_TEXT> is a string-typed field containing the message defined by the user when calling the Logging and Fault API. This field is written in quotation marks. The maximum size of the user message is defined by the maximum size of `ECOA::log`.
- <EOL> is the UNIX end-of-line character.

EXAMPLE of a log message: `"1336048735,618033988":0:"INFO":"node5":"Exe3":"My comments"<EOL>`

The following sections detail the prototype definitions for the logging and fault operations.

11.3.1 Log_Trace

The following is a prototype definition for the operation:

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

11.3.2 Log_Debug

The following is a prototype definition for the operation:

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

11.3.3 Log_Info

The following is a prototype definition for the operation:

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

11.3.4 Log_Warning

The following is a prototype definition for the operation:

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

11.3.5 Raise_Error

The following is a prototype definition for the operation:

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

11.3.6 Raise_Fatal_Error

The following is a prototype definition for the operation:

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

11.4 Time Services

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

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

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

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

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

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

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

In addition, it is possible to retrieve the time resolution for each of the time sources. 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

The following is a prototype definition for the operation:

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

11.4.2 Get_UTC_Time

The following is a prototype definition for the operation:

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

Get UTC Time operations may return the following status codes:

- `[ECOA:return_status:OK]`
  - No error
- `[ECOA:return_status:CLOCK_UNSYNCHRONIZED]`
  - No error – clock is unsynchronized; a valid value is still returned
- `[ECOA:return_status:OPERATION_NOT_AVAILABLE]`
  - UTC is not available – zero is returned

11.4.3 Get_Absolute_System_Time

The following is a prototype definition for the operation:

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

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

- `[ECOA:return_status:OK]`
  - No error
- `[ECOA:return_status:CLOCK_UNSYNCHRONIZED]`
  - No error – clock is unsynchronized; a valid value is still returned

11.4.4 Get_Relative_Local_Time_Resolution

The following is a prototype definition for the operation:

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

11.4.5 Get_UTC_Time_Resolution

The following is a prototype definition for the operation:

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

If UTC Time is not available Get_UTC_Time_Resolution will return a zero `utc_time_resolution`.

11.4.6 Get_Absolute_System_Time_Resolution

The following is a prototype definition for the operation:

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

11.5 Persistent Information Management (PINFO)

Persistent Information can be in the form of private PINFO or public PINFO. PINFO is only available in read-only mode and makes use of the `read_#PINFOname#` and `seek_#PINFOname#` APIs.

The following sections define the PINFO prototype definitions:

11.5.1 PINFO read

Read operation parameters are as follows:

- #PINFOname# is the PINFO name declared at Module Type level within PINFO usage attributes.
- 'memory_address' is a pointer to a memory area where read data is copied to.
- 'in_size' is the number of bytes to read from the current PINFO'index position.
- 'out_size' is the actual number of bytes read, equal to the minimum of 'in_size' 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 following is a prototype definition for the operation:

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

Read operations may return the following status codes:

- `[ECOA:return_status:OK]`
  - No error. 'out_size' bytes have been read
- `[ECOA:return_status:RESOURCE_NOT_AVAILABLE]`
  - An infrastructure error occurred. Example: Mass memory failure.
- `[ECOA:return_status:INVALID_PARAMETER]`
  - 'memory_address' is a NULL pointer or inaccessible

11.5.2 PINFO seek

Seek operations parameters are as follows:

- #PINFOname# is the PINFO name declared at Module Type level within PINFO usage attributes
- 'whence' 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)
- 'offset' is the number of bytes to be added to the selected position. "offset" is an int32 so as to allow to seek backwards from the selected position in the persistent data.
- '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 following is a prototype definition for the operation:

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

Seek operations may return the following status codes:

- `[ECOA:return_status:OK]`
  - PINFO'index position has been set according to the parameters.
- `[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)

11.5.3 Example of defining Private PINFO

Private PINFO attributes are implemented in the XML Metamodel as follows:

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

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

```
<moduleType name="M1_t">
<pinfo>
   <privatePinfo name="myprivate_R_PinfoOne" />
   <privatePinfo name="myprivate_R_PinfoTwo" />
</pinfo>
    <operations>
    <!-- … -->
    </operations>
</moduleType>

<!-- … -->

<moduleType name="M2_t">
<pinfo>
   <privatePinfo name="myprivate_R_PinfoOne" />
   <privatePinfo name="myprivate_R_PinfoTwo" />
   <privatePinfo name="myprivate_R_PinfoThree" />
</pinfo>
    <operations>
```

```
    <!-- … -->
    </operations>
</moduleType>
```

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

```
<moduleImplementation name="M1_Im" language="C" moduleType="M1_t"/>
<moduleImplementation name="M2_Im" language="C" moduleType="M2_t"/>

<!-- … -->

<moduleInstance name="M11" implementationName="M1_Im">
<pinfo>
   <privatePinfo name="myprivate_R_PinfoOne">"example_private_R_PINFO_11.txt"
   </privatePinfo>
   <privatePinfo name="myprivate_R_PinfoTwo">"example_private_R_PINFO_1.txt"
   </privatePinfo>
</pinfo>
</moduleInstance>

<moduleInstance name="M21" implementationName="M1_Im">
<pinfo>
   <privatePinfo name="myprivate_R_PinfoOne">"example_private_R_PINFO_21.txt"
   </privatePinfo>
   <privatePinfo name="myprivate_R_PinfoTwo">"example_private_R_PINFO_1.txt"
   </privatePinfo>
</pinfo>
</moduleInstance>

<moduleInstance name="M12" implementationName="M2_Im">
<pinfo>
   <privatePinfo name="myprivate_R_PinfoOne">"example_private_R_PINFO_12.txt"
   </privatePinfo>
   <privatePinfo name="myprivate_R_PinfoTwo">"example_private_R_PINFO_1.txt"
   </privatePinfo>
   <privatePinfo name="myprivate_R_PinfoThree">"example_private_R_PINFO_2.txt"
   </privatePinfo>
</pinfo>
</moduleInstance>
```

11.5.4 Example of defining Public PINFO

Public PINFO attributes are implemented in the XML Metamodel as follows:

- The XML Metamodel provides a way for defining Public PINFO at Module Type level:
  - Public PINFO usage attributes, which contain the following fields:
    - PINFO Name.
- The XML Metamodel provides a way for defining a Component Level Property for handling Public PINFO:
  - At Component Definition level, a Component Property is declared for handling Public PINFO.
- The XML Metamodel provides a way for referencing a Component Level Property at Module Instance level:
  - This is done via an association between PINFO name (defined in the Module Type) and a Component level Property (defined in the Component Definition). NOTE: it is forbidden to declare a Filename Association at this level for Public PINFO.
- The XML Metamodel provides a way for either:
  - referencing an Assembly Level Property for handling Public PINFO:
    - At Assembly Schema level, a PINFO Property and Filename Association is defined to associate a Property value with a filename. This file will provide the Public PINFO data for components which reference this Assembly level Property.
    - At Component Instance level, the Component Property is assigned a reference to an Assembly Schema property.
  - Or for declaring PINFO Filename Association at Component Instance level:
    - At Component Instance level a PINFO Filename Association is defined to associate a Property value with a filename. This file will provide the Public PINFO data for the Component Instance

The following example XML declares public PINFO, at assembly level (note that the “ecoa-sca:type” of the property is “ECOA:pinfo_filename”):

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<csa:composite xmlns:csa="http://docs.oasis-open.org/ns/opencsa/sca/200912"
               xmlns:ecoa-sca="http://www.ecoa.technology/sca"
               name="demo"  targetNamespace="http://www.ecoa.technology/sca">

<csa:property name="assembly_public_PINFO_1" ecoa-
sca:type="ECOA:pinfo_filename">
      <csa:value>"example_public_PINFO_1.txt"</csa:value> <!-- PINFO in "5-
Integration/Pinfo" folder -->
</csa:property>
<csa:property name="assembly_public_PINFO_2" ecoa-
sca:type="ECOA:pinfo_filename">
      <csa:value>"subfold/example_public_PINFO_2.txt"</csa:value> <!-- PINFO in
"5-Integration/Pinfo/subfold" folder -->
</csa:property>
```

The following example XML declares properties for public PINFO filename association, at Component Definition level:

```
<componentType>
   <!-- … -->
   <property name="component_public_PINFO_1"
      ecoa-sca:type="ECOA:pinfo_filename"/>
   <property name="component_public_PINFO_2"
      ecoa-sca:type="ECOA:pinfo_filename"/>
   <property name="component_public_PINFO_3"
      ecoa-sca:type="ECOA:pinfo_filename"/>
</componentType>
```

The following example XML declares Public PINFO at Module Type level. The example shows two Module Type definitions, both of which can access public PINFO:

```
<moduleType name="M1_t">
<pinfo>
   <publicPinfo name="mypublicPinfoOne"/>
   <publicPinfo name="mypublicPinfoTwo"/>
   <publicPinfo name="mypublicPinfoThree"/>
</pinfo>
    <operations>
    <!-- … -->
    </operations>
</moduleType>

<!-- … -->

<moduleType name="M2_t">
<pinfo>
   <publicPinfo name="mypublicPinfoOne"/>
</pinfo>
    <operations>
    <!-- … -->
    </operations>
</moduleType>
```

The following example XML declares references to Public PINFO at Module Instance level. The example shows three Module Instances being defined and Public PINFO assignments being made to reference component level PINFO properties:

```
<moduleImplementation name="M1_Im" language="C" moduleType="M1_t"/>
<moduleImplementation name="M2_Im" language="C" moduleType="M2_t"/>
```

```
<!-- … -->

<moduleInstance name="M11" implementationName="M1_Im">
<pinfo>
   <publicPinfo name="mypublicPinfoOne">$component_public_PINFO_1</publicPinfo>
   <publicPinfo name="mypublicPinfoTwo">$component_public_PINFO_2</publicPinfo>
   <publicPinfo
name="mypublicPinfoThree">$component_public_PINFO_3</publicPinfo>

</pinfo>
</moduleInstance>

<moduleInstance name="M21" implementationName="M1_Im">
<pinfo>
   <publicPinfo name="mypublicPinfoOne">$component_public_PINFO_1</publicPinfo>
   <publicPinfo name="mypublicPinfoTwo">$component_public_PINFO_2</publicPinfo>
   <publicPinfo
name="mypublicPinfoThree">$component_public_PINFO_3</publicPinfo>

</pinfo>
</moduleInstance>

<moduleInstance name="M12" implementationName="M2_Im">
<pinfo>
   <publicPinfo name="mypublicPinfoOne">$component_public_PINFO_1</publicPinfo>
</pinfo>
</moduleInstance>
```

The following example XML declares Public PINFO assignments, at Component Instance level. The first two examples reference Public PINFO properties at assembly level; the third value is defined within the Component Instance:

```
<csa:component name="ComponentOne">
   <!-- … -->
   <csa:property name="component_public_PINFO_1"
source="$assembly_public_PINFO_1" />
   <csa:property name="component_public_PINFO_2"
source="$assembly_public_PINFO_2" />
   <csa:property name="component_public_PINFO_3">
          <csa:value>example_ComponentOne_public_PINFO_1.txt</csa:value>
     </csa:property>
</csa:component>
```

11.6 Recovery Action

After analysis of error notifications, the Fault Handler carries on recovery actions provided by the Container Interface API.

Recovery actions cover actions on Components, Protection Domains, Computing Nodes and Computing Platforms. For Components, Protection Domains and Computing Nodes there are two recovery actions – Shutdown and Restart. For Computing Platforms there are three recovery actions – Shutdown, Restart and Change Deployment. The following describes each recovery action for each asset.

Note: it is being assumed in the following that the ECOA Platform is able to maintain warm start context according to the Warm Restart recovery actions being specified below

- Shutdown Component:
  - The Infrastructure puts all Module Instances of the targeted Component Instance in idle state. The Infrastructure frees associated resources, without calling the shutdown entry point of these potentially faulty Module Instances.
- Restart Component in cold or warm mode:
  - The Infrastructure performs the “Shutdown Component” recovery action (see above).
  - In case of cold start, the Infrastructure zeroes the ‘saved’ value of the Warm Start Context for all Module Instances within the component. In case of warm start, the Infrastructure restores the ‘saved’ value of the Warm Start Context without modifying it.
  - The Infrastructure initializes all Module Instances within the Component (INITIALIZE entry-point).
  - The Infrastructure starts all Module Instances within the Component (START entry-point).
- Shutdown the Protection Domain:
  - The Infrastructure puts in idle state all Module Instances of all Component Instances of which at least one Module Instance is deployed in the targeted Protection Domain. The Infrastructure frees associated resources, without calling the shutdown entry point of these potentially faulty Module Instances
  - Depending on underlying capabilities, the infrastructure may unload the Protection Domain from memory.

  Note: : according to Architecture Specification Part 3, it is required to target Component Instances consistently (i.e. if one Module Instance of a Component Instance goes to Idle state, all other Module Instances of that Component Instance must go to Idle state too, regardless of whether they are being deployed into different Protection Domains or not).
- Restart the Protection Domain in cold or warm mode :
  - The Infrastructure performs the “Shutdown Protection Domain” recovery action (see above).
  - Depending on underlying capabilities, the Infrastructure may unload the protection domain from memory and reload it to restart from a clean technical context.
  - In case of cold start, the Infrastructure zeroes the ‘saved’ value of the Warm Start Context for all Module Instances of all Component Instances of which at least one Module Instance is deployed within the targeted Protection Domain. In case of warm start, the Infrastructure restores the ‘saved’ value of the Warm Start Context without modifying it.
  - The Infrastructure initializes all Module Instances of all Component Instances of which at least one Module Instance is deployed within the targeted Protection Domain (INITIALIZE entry-point).
  - The Infrastructure starts all Module Instances of all Component Instances of which at least one Module Instance is deployed within the targeted Protection Domain (START entry-point).

  Note: according to Architecture Specification Part 3, it is required to target Component Instances consistently (i.e. if one Module Instance of a Component Instance goes to Idle state, all other Module Instances of that Component Instance must go to Idle state too, regardless of whether they are being deployed into different Protection Domains or not).

- Shutdown the Computing Node:
  - The Infrastructure performs the “Shutdown Protection Domain” recovery action for all Protection Domains of the computing node. See bullet above.

  Note: Other operating environments may still continue to run on the Computing Node. The shutdown recovery action at Computing Node level is only related to ECOA assets.
- Restart the Computing Node in cold or warm mode:
  - The Infrastructure performs the “Restart Protection Domain in cold or warm mode” recovery action for all Protection Domains of the computing node. See bullet above.
- Shutdown the Computing Platform:
  - The Infrastructure performs the “Shutdown Computing Node” recovery action for all computing nodes of the Platform. See bullet above.

  NOTE Other operating environments may still continue to run on the Computing Platform. The shutdown recovery action at Computing Platform level is only related to ECOA assets.
- Restart the Computing Platform in cold or warm mode
  - The Infrastructure performs the “Restart Computing Node in cold or warm mode” recovery action for all computing nodes of the Platform. See bullet above. The deployment is unchanged.
- Change the deployment on a Computing Platform:
  - The Infrastructure performs the “Shutdown Protection Domain” recovery action for all Protection Domains of all computing nodes of the Platform. See bullet above.
  - The Infrastructure loads in memory Protection Domains associated to the new deployment.
  - The Infrastructure zeroes the Warm Start Context for all Module Instances within all components deployed on the Platform (i-e a cold start).
  - For all components deployed on the Platform:
    - The Infrastructure initializes all Module Instances (INITIALIZE entry-point).
    - The Infrastructure starts all Module Instances (START entry-point).

  NOTE Other operating environments may still continue to run on the Computing Platform. The reload recovery action at Computing Platform level is only related to ECOA assets.

The availability of recovery actions depends on the capabilities offered by the underlying platform.

Recovery actions are implemented by a single container operation.

The following is the prototype definition for the container operation:

```
ECOA:return_status              [#error_handler_impl_name#_container:]recovery_action([#context#],
ECOA:recovery_action_type recovery_action, ECOA:asset_id asset_id, ECOA:asset_type asset_type);
```

The parameter recovery_action defines the recovery action to apply.

The parameter asset_id defines the asset ID on which the recovery action is applied.

The parameter asset_type identifies the type of asset targeted by the recovery action: component instance, protection domain, computing node or computing platform.

The operations may return the following status codes:

- `[ECOA:return_status:OK]`
  - No error
- `[ECOA:return_status:OPERATION_NOT_AVAILABLE]`
  - The recovery action is not implemented by the Infrastructure
  - The recovery action is not permitted for the targeted asset type

- [ECOA:return_status:INVALID_IDENTIFIER]
  - Operation called with an unknown asset ID
  - Operation called with an asset ID not consistent with the asset type
- [ECOA:return_status:OPERATION_ALREADY_PENDING]
  - Pending operation on the same asset already in progress

11.7 Save Warm Start Context

The Container Interface API allows a Module Instance to save its warm start (non-volatile) context such that it will be restored by the ECOA Infrastructure upon a Warm Restart.

This API is optional and is made available or not for a given Module Type depending on Metamodel attributes declared by the ASC supplier for the corresponding Module Type.

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 with regard to supported Fault Handler recovery actions.

The following is the prototype definition for the save_warm_start_context operation:

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

12 Container Types

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

12.1.1 Versioned Data Handles

This section 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.

13 External Interface

A Driver Component permits non-ECOA software to interact asynchronously with the ECOA System (see Architecture Specification Part 3). The Container will provide interfaces which may be used by non-ECOA software to post an event:

- To the Module Instance queue specified,
- Or to the Dynamic Trigger Instance queue specified.

The interface names and specifications shall adhere to the following prototype definitions:

```
void [#component_impl_name#__]#external_operation_name#(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 moduleType or a Dynamic Trigger Instance.

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

```
<moduleType name="mod2Type">
   <operations>

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

   </operations>
</moduleType>

<moduleImplementation name="mod2Impl" moduleType="mod2Type" language="C"/>

<moduleInstance name="module2" implementationName="mod2Impl"/>
```

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>
   <senders><external operationName="FeedbackLegacy" language="C"/></senders>
   <receivers><moduleInstance instanceName="module2"
operationName="TheFeedback"/></receivers>
</eventLink>
```

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

```
void [#component_impl_name#__]FeedbackLegacy(const ECOA:uint32 param1);
```

On the ECOA side, the external API permits all the normal ECOA connection flexibility for Module 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.

14 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 module.

It is applicable to:

- Request Response callback arguments in case of “NO_RESPONSE”,
- Memory space pointed by a get_write_access for the initial access (“DATA_NOT_INITIALIZED”),
- The arguments of the notification of a notifying versioned data in case of “RESOURCE_NOT_AVAILABLE”,

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.

15 Trigger Instances

Trigger Instances do not require a moduleType to be specified; its definition is implicit and managed by the infrastructure. The following XML shows what a Dynamic Trigger Module definition would look like:

```
<moduleType>
 <operations>
  <eventSent name="out" />
 </operations>
</moduleType>
```

15.1 XML definitions of Trigger Instance and associated links

The XML snippet below provides an example of a Trigger Instance connected to a Module Instance with a period set to 2 seconds (i.e. an out event will be sent to the Module Instance every 2 seconds when the Trigger Instance is running).

```
<triggerInstance name="aTriggerInstance"/>
...
 <eventLink>
    <senders>
         <trigger instanceName="aTriggerInstance" period="2" />  </senders>
    <receivers>
      <moduleInstance instanceName="consumerModule"
operationName="periodicOperation"/>
   </receivers>
 </eventLink>
 ...
```

16 Dynamic Trigger Instances

Dynamic Trigger Instances do not require a moduleType to be specified; its definition is implicit and managed by the infrastructure. The parameters associated with a given Dynamic Trigger are implied from the operations connected to it. The following XML shows what a Dynamic Trigger Module definition would look like:

```
<moduleType>
 <operations>
```

```
  <eventReceived name="in">
   <input name="expiryTime" type="ECOA:global_time"/>
   <input name="param1" type="T1"/>
   <input name="param2" type="T2"/>
    ...
  </eventReceived>
  <eventReceived name="reset"/>
  <eventSent name="out">
   <input name="param1" type="T1"/>
   <input name="param2" type="T2"/>
    ...
  </eventSent>
 </operations>
</moduleType>
```

16.1 XML definitions of Dynamic Trigger Instance and associated links

A Dynamic Trigger Instance is defined with the tag <dynamicTriggerInstance > within the Component implementation model (component.impl.xml); it is then used as any other ordinary Module.

Parameters of a Dynamic Trigger Instance are:

- Maximum number, named `size`, of waiting Events (outside possible queuing at network level)
  - By default: 1
  - Any Events that cause the maximum number of pending Events to be exceeded are discarded; an error is raised to the Fault Handler by the Infrastructure.

The XML snippet below provides an example of a Dynamic Trigger Instance with one integer parameter.

```
...
  <moduleType name="ProducerComputer">
    <operations>
      <eventSent name="set">
        <input name="expiryTime" type="ECOA:global_time"/>
        <input name="p1" type="int32"/>
      </eventSent>
    </operations>
  </moduleType>

  <moduleType name="ConsumerComputer">
    <operations>
      <eventReceived name="result">
        <input name="p1" type="int32"/>
      </eventReceived>
    </operations>
  </moduleType>
```

```
<moduleType name="ManagerComputer">
  <operations>
    <eventSent name="reset"/>
  </operations>
</moduleType>

<moduleImplementation name="ProducerComputerImpl"
  language="C" moduleType="ProducerComputer"/>
<moduleImplementation name="ConsumerComputerImpl"
  language="C" moduleType="ConsumerComputer"/>
<moduleImplementation name="ManagerComputerImpl"
  language="C" moduleType="ManagerComputer"/>

<moduleInstance name="producerComputer"
  implementationName="ProducerComputerImpl" relativePriority="3"/>
<moduleInstance name="consumerComputer"
  implementationName="ProducerComputerImpl" relativePriority="3"/>
<moduleInstance name="managerComputer"
  implementationName="ManagerComputerImpl" relativePriority="2"/>

<dynamicTriggerInstance name="delayResult"
  size="10" relativePriority="1">
  <parameter name="p1" type="int32"/>
</dynamicTriggerInstance>

<eventLink>
  <senders>
    <moduleInstance instanceName="producerComputer"
      operationName="set"/>
  </senders>
  <receivers>
    <dynamicTrigger instanceName="delayResult"
      operationName="in"/>
  </receivers>
</eventLink>

<eventLink>
  <senders>
    <dynamicTrigger instanceName="delayResult"
      operationName="out"/>
  </senders>
  <receivers>
    <moduleInstance instanceName="consumerComputer"
```

```
        operationName="result"/>
    </receivers>
  </eventLink>

  <eventLink>
    <senders>
      <moduleInstance instanceName="managerComputer"
        operationName="reset"/>
    </senders>
    <receivers>
      <dynamicTrigger instanceName="delayResult"
        operationName="reset"/>
    </receivers>
  </eventLink>

...
```

Module Instances interact with the Dynamic Trigger Instance through the usual Event API generated as defined in 10.1.3 and in 11.1.1.1.

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 namespace.