logocn1



|  |  |  |
| --- | --- | --- |
| REFERENCE | INDEX | DATE |
| LDP\_ASCTG\_EXVT\_UM | A | 15/03/2023 |
|  | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | | | |
| LDP/ASCTG/EXVT  USER MANUAL | | | | | | |
| SCOPE | | | |
| LDP/ASCTG/EXVT | | | |
|  | | | |
|  | | | |
|  | | | |
|  | | | | | |
| Direction Générale Technique | | | | | | | | |
| Issue | Date | Index | Author (name) | Visa Reviewer | | Visa Approbator  Capgemini:  Dassault Aviation: | | |
| Origin | 15/03/2023 | A | Cagemini |  | |  | | |
| Last Edition |  |  |  |  | |  | | |

Table of Contents

[1 Preamble 5](#_Toc130808340)

[1.1 Document issues 5](#_Toc130808341)

[1.2 Summary 5](#_Toc130808342)

[1.3 List of modified pages 6](#_Toc130808343)

[2 Introduction 6](#_Toc130808344)

[2.1 Identification 6](#_Toc130808345)

[2.2 Scope 6](#_Toc130808346)

[2.3 Limitations 6](#_Toc130808347)

[2.3.1 Missing features from ECOA Architecture Specification (Issue 6) 6](#_Toc130808348)

[2.3.2 Limits 6](#_Toc130808349)

[2.4 Definitions, Abbreviations & Acronyms 6](#_Toc130808350)

[2.5 Applicable and reference documents 8](#_Toc130808351)

[3 ECOA Architecture Overview 9](#_Toc130808352)

[4 Reminder of main ECOA Mechanisms 10](#_Toc130808353)

[*4.1* Module Interactions 10](#_Toc130808354)

[*4.2* Trigger 10](#_Toc130808355)

[*4.3* Dynamic Trigger 10](#_Toc130808356)

[4.4 Persistent Information 11](#_Toc130808357)

[4.4.1 Overview 11](#_Toc130808358)

[*4.4.2* PINFO organization 11](#_Toc130808359)

[5 How to create an ECOA application 12](#_Toc130808360)

[5.1 Design 12](#_Toc130808361)

[5.2 0-Types 15](#_Toc130808362)

[5.3 1-Services 15](#_Toc130808363)

[5.4 2-Component Definitions 16](#_Toc130808364)

[5.5 3-Initial Assembly 18](#_Toc130808365)

[5.6 4-Component Implementation 18](#_Toc130808366)

[5.6.1 Component implementation folders 18](#_Toc130808367)

[5.6.2 Launching script (Optional) 21](#_Toc130808368)

[5.6.3 C/C++ Tips 25](#_Toc130808369)

[5.7 5-Integration 26](#_Toc130808370)

[5.8 Project file 28](#_Toc130808371)

[6 EXVT Tool 30](#_Toc130808372)

[6.1 Principle 30](#_Toc130808373)

[6.2 Partial validation 30](#_Toc130808374)

[7 ASCTG Tool 31](#_Toc130808375)

[7.1 Principle 31](#_Toc130808376)

[7.2 Example 31](#_Toc130808377)

[7.2.1 Overview 31](#_Toc130808378)

[7.2.2 Schema 31](#_Toc130808379)

[7.2.3 Configuration file 32](#_Toc130808380)

[7.2.4 Output files 32](#_Toc130808381)

[8 LDP Tool 33](#_Toc130808382)

[8.1 Principle 33](#_Toc130808383)

[8.2 LDP Mechanism 33](#_Toc130808384)

[8.2.1 Fault Handler 33](#_Toc130808385)

[8.2.2 Multi-nodes 41](#_Toc130808386)

[8.2.3 local.cmake Mechanism 42](#_Toc130808387)

TABLE OF FIGURES

[**Figure 1: Launcher schema** 22](#_Toc130808314)

TABLE OF TABLES

[Table 1:Model Data Organisation 14](#_Toc130808323)

# Preamble

## Document issues

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Index | Author | Updating Purpose |
| 15/03/2023 | A | Capgemini | LDP V1.0.0 |
|  |  |  |  |
|  |  |  |  |

## Summary

This document is the User Manual for the EXVT, ASCTG and LDP tools.

## List of modified pages

|  |
| --- |
| All the pages of this document are at the latest issue. |

Not applicable to this issue

# Introduction

## Identification

This document applies to the products named EXVT, ASCTG and LDP.

## Scope

The present document explains how to use and develop the ECOA application to comply with ECOA specifications. It is assumed that the reader is familiar with the ECOA Architecture Specification and the process of defining and declaring ECOA Assemblies, ASCs (components), Modules, and deployments in XML, and then using code generation to produce Module framework (stub) code units and ECOA Container and Platform code.

This document covers the use of several elements such as:

* The middleware source code generator which is qualified on a perimeter corresponding to ECOA Architecture Specification (Issue 6).
* The ECOA XML Validation Tool (EXVT).
* The ASC Test Generator (ASCTG).
* The Lightweight Development Platform tool (LDP).

## Limitations

### Missing features from ECOA Architecture Specification (Issue 6)

As of now, the following features described in the ECOA Architecture Specification (Issue 6) have not yet been implemented:

* Module deployment is not allowed everywhere. Modules of the same component should be deployed on the same Protection Domain.
* ECOA model with recursive assembly.
* Quality of Service (QoS).

### Limits

* Communication between nodes with different endianness setting is not supported.

## Definitions, Abbreviations & Acronyms

|  |  |
| --- | --- |
| **Abbreviations/ Acronyms** | **Description** |
| ASC | Application Software Component |
| ECOA | European Component Oriented Architecture |
| ELI | ECOA® Logical Interface |
| HW | Hardware |
| N/A | Not Applicable |
| OS | Operating System |
| SRS | Software Requirement Specifications |
| TCP | Transmission Control Protocol |
| UDP | User Datagram Protocol |
| PD | Protection Domain |

## Applicable and reference documents

| **A/R** | **Document No.** | **Version** | **Dated** | **Document Title** |
| --- | --- | --- | --- | --- |
| [R-1] | RG AERO 000 973 | Ed. B |  | RG AERO 000 973 Edition B - Recommandations générales -  European Component Oriented Architecture – ECOA |
| [R-2] | IAWG-ECOA-TR-001  DGT 144474 | Issue 6 |  | European Component Oriented Architecture (ECOA®) Collaboration Programme:  Architecture Specification  Part 1: Concepts |
| [R-3] | IAWG-ECOA-TR-012  DGT 144487 | Issue 6 |  | European Component Oriented Architecture (ECOA®) Collaboration Programme:  Architecture Specification  Part 2: Definitions |
| [R-4] | IAWG-ECOA-TR-007  DGT 144482-F | Issue 6 |  | European Component Oriented Architecture (ECOA®) Collaboration Programme:  Architecture Specification  Part 3: Mechanisms |
| [R-5] | IAWG-ECOA-TR-010  DGT 144485-F | Issue 6 |  | European Component Oriented Architecture (ECOA®) Collaboration Programme:  Architecture Specification  Part 4: Software Interface |
| [R-6] | IAWG-ECOA-TR-008  DGT 144483-F | Issue 6 |  | European Component Oriented Architecture (ECOA®) Collaboration Programme:  Architecture Specification  Part 5: High Level Platform Requirements |
| [R-7] | IAWG-ECOA-TR-006  DGT 144481-F | Issue 6 |  | European Component Oriented Architecture (ECOA®) Collaboration Programme:  Architecture Specification  Part 6: ECOA® Logical Interface |
| [R-8] | IAWG-ECOA-TR-011  DGT 144486-F | Issue 6 |  | European Component Oriented Architecture (ECOA®) Collaboration Programme:  Architecture Specification  Part 7: Metamodel |
| [R-9] | IAWG-ECOA-TR-004  DGT 144477-F | Issue 6 |  | European Component Oriented Architecture (ECOA®) Collaboration Programme:  Architecture Specification  Part 8: C Language Binding |
| [R-10] | IAWG-ECOA-TR-005  DGT 144478-F | Issue 6 |  | European Component Oriented Architecture (ECOA®) Collaboration Programme:  Architecture Specification  Part 9: C++ Language Binding |

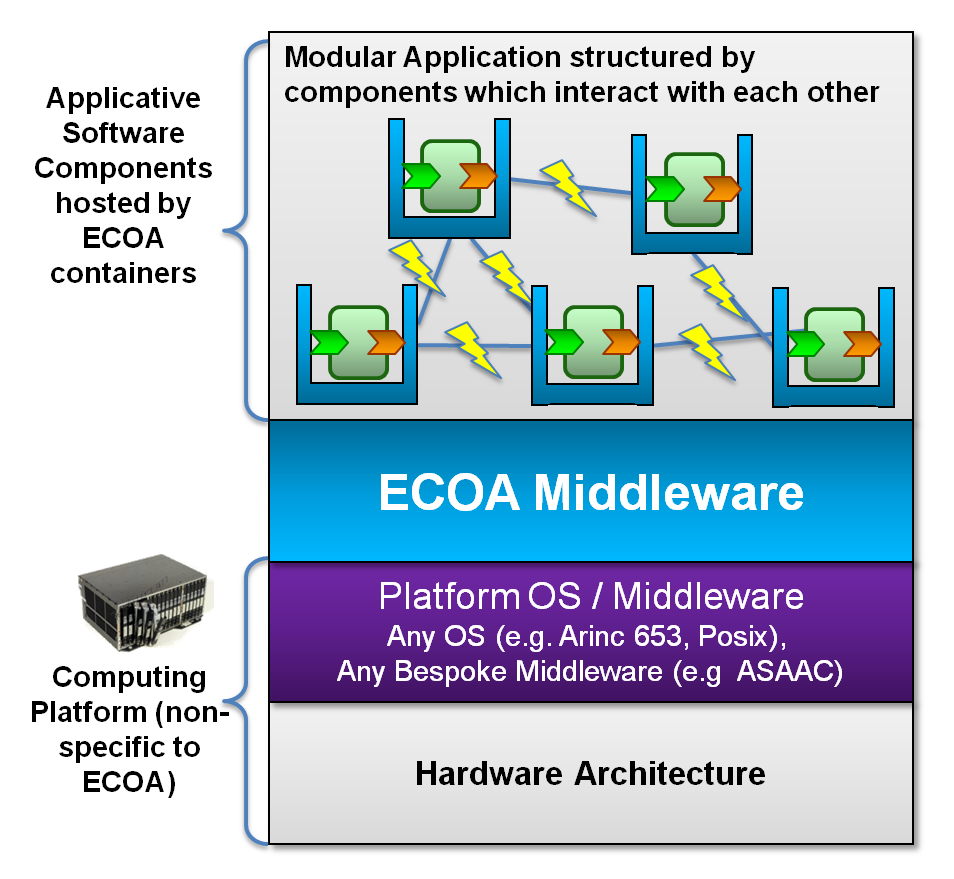
# ECOA Architecture Overview

The ECOA standard enables the construction of a service-oriented architecture using a model-based approach, which provides the capability to formally specify an assembly of Application Software Components, as well as their interfaces and deployment onto computing platforms.

Application Software Components are independent of the underlying computing platform thanks to the concept of “ECOA Containers”. These provide all the technical facilities needed by the functional code, using three standardized communication mechanisms (Request Response, Events and Versioned Data).

The key features provided by the ECOA technical approach are:

* Standardized service-oriented interface mechanisms, including quality of service.
* High level of functional interactions between Application Software Components.
* Portability of Application Software Components to any ECOA compliant computing platform.
* Interoperability between any ECOA compliant computing platforms.
* Support for many real-time scheduling policies.



**Figure 1: ECOA software architecture**

Building an ECOA system consists in formally specifying the interfaces and the internal architecture of Application Software Components, assembling these components together using service links and specifying the deployment of the components over the computing platform nodes.

# Reminder of main ECOA Mechanisms

The main mechanisms of the ECOA Architecture Specification (Issue 6) are described here, whereas others are only discussed at a high level, as they are covered in greater depth in ECOA specification documents.

## Module Interactions

Interactions between Module Instances in an ECOA system rely on three primary mechanisms:

* Events: The Event mechanism is used for one-way asynchronous “push-style” communication between Module Instances and may optionally carry typed data. When Events are used to implement a Service Operation, a Module Instance may be either the sender or receiver of an Event irrespective of whether it is designated as the Provider or Requester of the Service.
* Request-Response: The “Request-Response” mechanism is a two-way communication between Module Instances. The calling Module Instance requests an operation and the called Module Instance provides a Response. The Requesting Module Instance (sender of the Request) is named the “Client”, and the providing Module Instance (sender of the Response) is named the “Server”.
  + There are two mechanisms for Request operations which provide synchronous and asynchronous behavior at the Client and one mechanism for Response operations at the Server. The details of these are described in the following sections.
* Versioned Data (with or without access control): The Versioned Data mechanism allows Module Instances to share typed data. A reading Module Instance is named a “Reader”, and a writing Module Instance is named the “Writer”.
  + With access control, the ECOA Infrastructure ensures a concurrency-safe read-write paradigm between Module Instances by making a local copy of the data when a reader or a writer accesses it.
  + Without access control, the local data repository is accessed directly by all Module Instances. Therefore, concurrency must be managed at application level by the Component Supplier.

## Trigger

Triggers generate periodic Events which can be used to invoke some functionalities provided by a Module Instance or set of Module Instances. The Trigger can generate Events which are queued to Module Instances within the same Application Software Component and/or generate events which are queued to Module Instances in different Application Software Components via a Service e.g., where a single central Trigger is used to coordinate the execution of functionality of multiple Components, in the manner of a “central clock”.

## Dynamic Trigger

A Dynamic Trigger sends an Event after a given delay (known as the out Event) from the receipt of an input Event (known as the in Event). The in Event specifies the delay in the form of the expiry time at which the out Event shall be sent. A Dynamic Trigger may also receive a reset Event, which will purge all unexpired delays.

It is possible for multiple Module Instances to:

* Send in and reset Events to the same Dynamic Trigger.
* Receive the same out Event.

## Persistent Information

### Overview

Persistent information is broken down into two main categories related to the extent of its accessibility as follows:

* Private PINFO – is data accessible by Module Instances within the same ASC Instance.
* Public PINFO – is data accessible by any Module Instance in the Assembly Schema.

PINFO is Read-Only.

PINFO can be referenced (i.e., accessed) by multiple Module Instances.

### PINFO organization

Prior to deployment on the target ECOA Platform(s), PINFO is organized as follows:

* Public PINFO are stored in directories specified in figure 2. Public PINFO are stored in ‘5-Integration/Pinfo’ directory, and their Filename Association is declared as a relative path to that directory (sub-directories are allowed).
* Private PINFO is common to all ASC Instances of a given ASC Implementation (as it is stored at ASC Implementation level). Private PINFO are stored in ‘4-ComponentImplementations/<Component Implementation Name>/Pinfo’ directory, and their Filename Association is declared as a relative path to that directory (sub-directories are allowed).



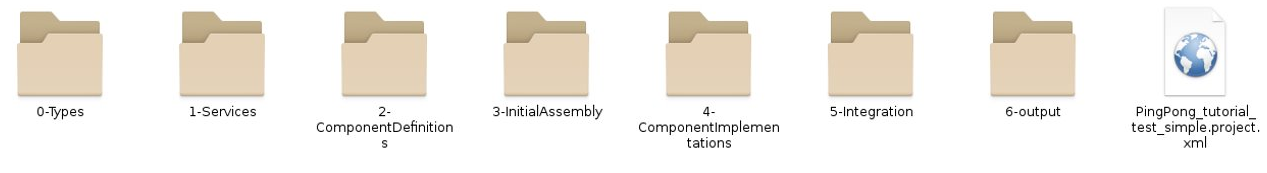
**Figure 2: PINFO organization prior to deployment**

# How to create an ECOA application

## Design

Even if they do not fully cover all ECOA metamodel aspects, the following chapters will provide some needed information to start.

For more details, you can refer to the document ECOA\_AS\_Part\_7\_Metamodel.

ECOA systems are described in the folder **examples** of the repository **ecoa-genplatform** with the following hierarchy of 6 sub folders.

* 0-Types
* 1-services
* 2-ComponentDefinitions
* 3-InitialAssembly
* 4-ComponentImplementations
* 5-Integration
* 6-output (🡸 Generated)

| Directory | Sub-directory | Sub-directory 2 | Sub-directory N | Files |
| --- | --- | --- | --- | --- |
| 0-Types | N/A | N/A | N/A | * \*\*\*.types.xml   Example: pingpong.types.xml |
| 1-Services | N/A | N/A | N/A | \*\*\*.interface.xml |
| 2-ComponentDefintions | <component\_type\_name>  Examples: Ping and Pong | N/A | N/A | * <component\_type\_name>.componentType   Example: Ping.componentType and Pong.componentType |
| 3-InitialAssembly | N/A | N/A | N/A | * \*\*\*.composite   Example: demo.composite |
| 4-ComponentImplementations |  |  |  | * launcher.txt (Optional) |
| <component\_implementation\_name>  By example:myDemoPing and myDemoPong |  |  | * <component\_implementation\_name>.impl.xml   Example: myDemoPing.impl.xml and myDemoPing.impl.xml   * Bin-desc.xml (optional) * Binary files (e.g. \*.o or .dll)(optional) |
| <module\_implementation\_name>  By example: myDemoPing\_AM and myDemoPong\_AM | * Inc * src | * <module\_implementation\_name>\_user\_context.h   Example: myDemoPing\_AM\_user\_context.h and myDemoPong\_AM\_user\_context.h   * <module\_implementation\_name>.c |
| Pinfo (Optional) |  | Sub-directory for storing Private PINFO |
| <name\_of\_subdirectory> | Optional sub-directories for organizing Private PINFO |
| 5-Integration | N/A |  |  | * \*\*\*.impl.composite * \*\*\*.logical-system.xml * sca-contribution.xml * \*\*\*.cross\_platform\_view.xml * \*\*\*.ids.xml * ip\_address\_dployment.xml (Optional) * fine\_grain\_deployment.xml (Optional) |
| * src (Optional) |  |  | * myPlatform\_fault\_handler.c |

Table 1:Model Data Organisation

## 0-Types

In this folder you can declare all your data types used by operations using the file **types.xml**. This file allows declaring functional datatypes that are being exchanged through service operations. Each xml file is a <<library>> of datatypes that each ECOA xml file may reference to use these datatypes when declaring service operations. The datatypes according to ECOA standards that can be defined are: constant, basic, simple, enum, array, fixed array, record, variant record.

enum, record, basic, simple, fixed array, array, constant/basic/variant record.

* **pingpong.types.xml File Example:**

The datatypes like enum, simple, and record are defined here

|  |
| --- |
| <?xml version=*"1.0" encoding="UTF-8"?>*  <library xmlns=*"http://www.ecoa.technology/types-2.0">*  <types>  <enum name=*"T\_Side" type="uint8">*  <value name=*"PING"/>*  <value name=*"PONG"/>*  </enum>  <simple name=*"T\_Tactical\_Item\_ID" type="uint32"/>*  <simple name=*"T\_Angle" type="float32" unit="radian"/>*  <record name=*"T\_2D\_Position">*  <field name=*"Latitude" type="T\_Angle"/>*  <field name=*"Longitude" type="T\_Angle"/>*  </record>  <simple name=*"T\_Time" type="int64" unit="nanoseconds"/>*  <record name=*"T\_Target\_Position">*  <field name=*"Tactical\_Item\_ID" type="T\_Tactical\_Item\_ID"/>*  <field name=*"Location" type="T\_2D\_Position"/>*  <field name=*"Is\_Valid" type="boolean8"/>*  </record>  </types>  </library> |

## 1-Services

In this folder you can declare one or more XML services files.

These XML files allow declaring ECOA services. There is one XML file per ECOA service, the name of the files should be as follows: ***<service\_name>.interface.xml***.

Defining a service consists in defining the prototype of operations provided by this service.

The operations that can be defined are request response, event, versioned data.

The type or direction of operations must be defined in operations.

The direction, exclusively used for event is either “***RECEIVED\_BY\_PROVIDER***” or “***SENT\_BY\_PROVIDER***”.

To use data types declared in “0-Types” you can use the closing tag: ***<use library=<prefix\_of\_types\_file>/>***, once it is done the user would be able to use the declared data types from the libraries by using the following attributes*:* ***type=<library\_name>:<declared\_type>****.*

To declare parameter depending on the operations you can use the closing tags **input** and **output** for respectively input parameter and output parameters.

Note: The parameters order is important for the declared operations in every XML files.

For version data, there are no input or output tags. You should directly declare the version data type: ***<data name=<version\_data\_name> type=<data\_type>>***

* **svc\_PingPong.interface.xml file Example:**

Service operations like request response, event along with input and output types are defined here:

|  |
| --- |
| <?xml version=*"1.0"?>*  <serviceDefinition xmlns=*"http://www.ecoa.technology/interface-2.0">*  <use library=*"pingpong"/>*  <operations>  <requestresponse name=*"PingPong">*  <input name=*"Ping\_Target" type="pingpong:T\_Target\_Position"/>*  <output name=*"Pong\_Target" type="pingpong:T\_Target\_Position"/>*  </requestresponse>  <event direction=*"RECEIVED\_BY\_PROVIDER"  name="Ping"/>*  <event direction=*"SENT\_BY\_PROVIDER"  name="Pong"/>*  <data name=*"Counter" type="uint32"/>*  </operations>  </serviceDefinition> |

## 2-Component Definitions

User needs to create below said folder structure:

<component\_name\_1>

* + - * <component\_name\_1>.componentType

<component\_name\_N>

* + - * Pong.componentType

Example:

Ping

* + - * Ping.componentType

Pong

* + - * Pong.componentType

These XML files allow declaring ECOA component types that can be instantiated in the ECOA assembly. There is one such XML file per component type. Declaring a component type consists in declaring which services it provides and requires, among services declared in 1-Services.

When a component requires a service, the tag used in the XML is “***reference***”. Inside that opening tag you should declare the name of the service required with the closing tag “***ecoa-sca:interface”*** the syntax is:

***<reference name=<name\_of\_the\_service\_reference>>***

***<ecoa-sca:interface syntax=<name\_of\_service> qos=<optional\_prefix\_name\_of\_qos\_file>>***

***</reference>***

When a component provides a service, the tag used in the XML is “***service***”. Inside that opening tag you should declare the name of the service provided with the closing tag “***ecoa-sca:interface”*** the syntax is:

***<reference name=<name\_of\_the\_service\_provided>>***

***<ecoa-sca:interface syntax=<service\_name> qos=<optional\_prefix\_name\_of\_qos\_file>>***

***</reference>***

Note: the name\_of\_the\_service\_reference and name\_of\_the\_service\_provided could be different from the service\_name.

Note: QoS is not supported (see § 2.4.1).

* **Ping.componentType file Example:**

|  |
| --- |
| <?xml version*="1.0"*encoding*="UTF-8"*?>  <componentType xmlns*="http://docs.oasis-open.org/ns/opencsa/sca/200912"*  xmlns:xs*="http://www.w3.org/2001/XMLSchema"*  xmlns:ecoa-sca*="http://www.ecoa.technology/sca-extension-2.0"*>  <reference name*="svc\_PingPong"*>  <ecoa-sca:interface syntax*="svc\_PingPong"*/>  </reference>  </componentType> |

* **Pong.componentType file Example:**

Provided service name and Quality of Service name are defined here

|  |
| --- |
| <?xml version*="1.0"*encoding*="UTF-8"*?>  <componentType xmlns*="http://docs.oasis-open.org/ns/opencsa/sca/200912"*  xmlns:xs*="http://www.w3.org/2001/XMLSchema"*  xmlns:ecoa-sca*="http://www.ecoa.technology/sca"*>  <service name*="svc\_PingPong"*>  <ecoa-sca:interface syntax*="svc\_PingPong"*/>  </service>  </componentType> |

## 3-Initial Assembly

In this folder you declare a file(\*\*\*.composite) that allows building a logical system architecture by declaring instances of component types. It is here that the required/Provided services belonging to respective component are defined.

Then provided/required services are connected to each other from source component to target component. This is called **“wiring”** the syntax is as follows**:**

***<csa:wire source=<component\_instance\_name\_1>/< name\_of\_the\_service\_reference” target==<component\_instance\_name\_2/ name\_of\_the\_service\_reference >>***

This is useful at high level system design time. This mapping is then used to perform communication between components via service operations.

Component name, its type, its respective required/provided service are defined here.

Wiring is done to connect two components through service.

* **Demo.composite file example**

|  |
| --- |
| <?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-extension-2.0"*  name=*"demo"*  targetNamespace=*"http://www.ecoa.technology/sca\_extension-2.0">*  <csa:component name=*"demoPing">*  <ecoa-sca:instance componentType=*"Ping"/>*  <csa:reference name=*"svc\_PingPong"/>*  </csa:component>  <csa:component name=*"demoPong">*  <ecoa-sca:instance componentType=*"Pong"/>*  <csa:service name=*"svc\_PingPong"/>*  </csa:component>  <csa:wire source=*"demoPing/svc\_PingPong" target="demoPong/svc\_PingPong" />*  </csa:composite> |

## 4-Component Implementation

This folder contains organized folders for every component implementation.

Names of the folders are the component implementation names.

### Component implementation folders

Inside each of these folders you can find the following XML file:

#### Component implementation XML file

The naming of this file is <component\_implementation\_name>.impl.xml.

This file allows defining a possible software implementation of an ECOA component instance, i.e., its internal breakdown into mono-threaded modules. Therefore, a component implementation XML file always references a component type.

Defining a component implementation XML is done by following these steps:

* + First, declare module types in the same fashion as component types: module types are being defined by their interface (input/output operations at the boundary of each module.).  
    At this stage, operations declared at module type level are not yet related to provided/required services declared at component level.
  + Secondly, this XML file allows defining several possible software implementations of module types (possibly using different programming languages: C, C++).
  + Thirdly, this XML file allows declaring module instances. A module instance is defined by the module type that it instantiates and the choice of a module implementation. It is also possible to declare modules called « trigger » and « dynamic trigger » that can be used for implementing periodic activations of other modules (« Heart\_Beat » in the below example) or for waking up a module after timeout.  
    Module Operations like request sent, request received, event sent, event received can be declared and defined using operation links along with their input and output parameters.

**myDemoPing.impl.xml file example:**

|  |
| --- |
| <componentImplementation  xmlns=*"http://www.ecoa.technology/implementation-2.0" componentDefinition="Ping">*  <!-- list of used libraries -->  <use library=*"pingpong"/>*  <!-- module AM to implement provided operations -->  <moduleType name=*"myDemoPing\_AM\_t">*  <operations>  <requestSent name=*"PingPong" isSynchronous="true“ timeout="30.0">*  <input name=*"Ping\_Target" type="pingpong:T\_Target\_Position"/>*  <output name=*"Pong\_Target" type="pingpong:T\_Target\_Position"/>*  </requestSent>  <eventSent name=*"Ping"/>*  <eventReceived name=*"Pong"/>*  <dataRead type=*"uint32" name="Counter" maxVersions="8"/>*  <eventReceived name=*"TriggerPingRequest"/>*  <eventReceived name=*"TriggerPingEvent"/>*  <eventReceived name=*"TriggerPingCounter"/>*  </operations>  </moduleType>  <moduleImplementation name=*"myDemoPing\_AM" language="C" moduleType="myDemoPing\_AM\_t"/>*  <moduleInstance name=*"myDemoPing\_AM\_I" implementationName="myDemoPing\_AM" relativePriority=“20"/>*  <triggerInstance name=*"Heart\_Beat" relativePriority=“10"/>*  <!-- Definition of module operation links -->  <requestLink>  <clients>  <moduleInstance instanceName=*"myDemoPing\_AM\_I" operationName="PingPong"/>*  </clients>  <server>  <reference instanceName=*"svc\_PingPong" operationName="PingPong"/>*  </server>  </requestLink>  <eventLink>  <senders>  <trigger instanceName=*"Heart\_Beat" period="2.000"/>*  </senders>  <receivers>  <moduleInstance instanceName=*"myDemoPing\_AM\_I" operationName="TriggerPingRequest"/>*  </receivers>  </eventLink>  <eventLink>  <senders>  <trigger instanceName=*"Heart\_Beat" period="3.000"/>*  </senders>  <receivers>  <moduleInstance instanceName=*"myDemoPing\_AM\_I" operationName="TriggerPingEvent"/>*  <moduleInstance instanceName=*"myDemoPing\_AM\_I" operationName="TriggerPingCounter"/>*  </receivers>  </eventLink>  <eventLink>  <senders>  <moduleInstance instanceName=*"myDemoPing\_AM\_I" operationName="Ping"/>*  </senders>  <receivers>  <reference instanceName=*"svc\_PingPong" operationName="Ping"/>*  </receivers>  </eventLink>  <eventLink>  <senders>  <reference instanceName=*"svc\_PingPong" operationName="Pong"/>*  </senders>  <receivers>  <moduleInstance instanceName=*"myDemoPing\_AM\_I" operationName="Pong"/>*  </receivers>  </eventLink>  <dataLink>  <writers>  <reference instanceName=*"svc\_PingPong" operationName="Counter"/>*  </writers>  <readers>  <moduleInstance instanceName=*"myDemoPing\_AM\_I" operationName="Counter"/>*  </readers>  </dataLink>  </componentImplementation> |

#### Module implementation folders

For every module implementation, the following folders are used to build the corresponding module:

* + **<module\_implementation\_name>/inc :**

This folder contains the user\_context header files used in application. These header files contain the declarations user\_context structure.

* + **<module\_implementation\_name>/src :**

This folder contains application source file which contains all the logic to execute the module operations, module entry points.

* + **<module\_implementation\_name>/inc-gen :**

These header files contain the declarations of various structures like module context, function prototypes of Logging APIs, time related services, module operations.

* + **<module\_implementation\_name>/src-gen :**

This folder contains application source file which contains function definitions of module operations, container Logging APIs, time related APIs.

#### PINFO

In <component\_implementation\_name> folder you can have a PINFO folder to store Private PINFO files.

### Launching script (Optional)

By default, when all components are connected, all modules are initialized. Once all of them have been connected, they are started.

By adding a file `launcher.txt` in `4-ComponentImplementations`, it is possible to manually schedule the lifecycle of modules. It is also possible to request for some breaks between different operations are sent.

To use that functionality, you must specify instructions like the following:

* <Lifecycle instruction> <Name of the component or Protection domain> <Name of the module>

As <Lifecycle instruction> you have: Initialize, Start, Shutdown and Stop

**Example:**

Initialize demoPong myDemoPong\_AM\_I

Initialize demoPing Heart\_Beat

Start demoPing myDemoPing\_AM\_I

Start demoPing Heart\_Beat

You can also specify a latency by using the following instruction:

* Wait <duration> <duration unit>

<duration> is an integer

As <duration unit> you can have ms(milliseconds), s(seconds), m(minutes)

**Example:**

Wait 1 s



**Figure 1: Launcher schema**

Note: Once you decide to use a “launcher.txt” file, you must mention all modules of your application in it. Otherwise, you will get an error at the execution of your application such as: one module is not in the right state.

#### Syntax rules

Here are some syntax rules to apply to the launcher.text file:

* One line per operation (one line break per operation).
* Operations are either `Initialize`, `Start`, `Kill` or `Shutdown`. Case DOES matter.
* The syntax of these operations is the following:
  + `#NameOfTheOperation# #NameOfTheComponent# #NameOfTheModule#`
* It is possible to apply an operation to all Modules of a Component with the syntax:
  + `#NameOfTheOperation# #NameOfTheComponent# \*`
* It is possible to apply an operation to all Modules with the syntax:
  + `#NameOfTheOperation# \*`
* It is possible to apply an operation to all Modules of a given Protection Domain with the syntax:
  + `#NameOfTheOperation# \*PD #NameOfTheProtectionDomain#`
* To request for a break of a certain amount of time, the syntax is the following:
  + `Wait Duration DurationUnit`
  + Duration must be an unsigned integer.
  + DurationUnit can be either `ms` (milliseconds), `s` (seconds) or `m` or `min` (minutes). Negative values of duration will be considered as "0" and floats will make the line get discarded and raise warnings.
* It is possible to comment some lines: Just add `#` in front of the line. They won't be considered.
* It is possible to \*print some comment lines\*: Just add `###` in front of the line. They won't be considered but the whole line (including `###`) will be displayed as a log.
* You might add comments at the end of a line. Using `#` is highly recommended for readability.

Be careful, **syntax is case sensitive!**

In case of wrong syntax, lines that do not correspond to any standard line will be discarded (a warning should be raised but will not interrupt the launch).

##### Size

The maximum length of a line is 512 characters

The length of a module, component of protection domain cannot be bigger than 128 characters

In case of longer words, the supplementary words will be discarded. In case of a too long line, the supplementary words will be considered as part of another line: this may raise some issues.

#### Limits

The order and the timing of the requested operations are not guaranteed to be the same at the end.

For example, if two operations A and B are requested to be executed in this order (A then B), B might be executed A.

Because of this timing constraints, the user must pay attention as it may result in failed operations.

For example, if the initialization and the start of the same module are requested without any break, the module might be requested to start before being requested to initialize. In this case, the start request will fail.

Therefore, when the order does matter, it is highly recommended to put some breaks in `launcher.txt` to avoid facing those issues.

To be noted that the duration of breaks requested does not include any delay (execution duration, processing duration, etc.).

#### Example

Based on the following platform:

|  |  |  |
| --- | --- | --- |
| Protection Domain | Component | Modules |
| PD1 | Comp1 | Mod1 |
| PD1 | Comp1 | Mod2 |
| PD1 | Comp1 | Mod3 |
| PD1 | Comp2 | Mod4 |
| PD1 | Comp2 | Mod5 |
| PD1 | Comp2 | Mod6 |
| PD2 | Comp3 | Mod7 |
| PD2 | Comp3 | Mod8 |
| PD2 | Comp3 | Mod9 |
| PD2 | Comp1 | Mod10 |

One example of code with a correct syntax would be:

|  |
| --- |
| #Initialization  Initialize Comp1 Mod1  Initialize Comp1 Mod2  Initialize Comp1 Mod3  Initialize Comp1 Mod10  Initialize Comp2 \* #Initializes Mod4, Mod5 and Mod6  Initialize Comp3 \* #Initializes Mod7, Mod8 and Mod9  #Wait to make sure that all modules are initialized  Wait 2 s  Start \*PD PD1 #Starting all modules of Protection Domain PD1  Start \*PD PD2 #Starting all modules of Protection Domain PD2  #Let everything run 2 seconds  Wait 2 s  #Stop everything at the same time  Stop \*  #Wait 2 seconds again and start PD1  Start \*PD PD1  #Wait 10 s and Shutdown everything  Shutdown \* |

#### Tests

You can find several examples for tests in the folder examples/tests\_launcher/ of **ecoa-genplatform** Git repository.

The duration for these tests is 15 seconds since some Wait requests are added to avoid any irrelevant trouble.

### C/C++ Tips

Keep in mind that programming language rules apply.

E.g., reserved keywords are not available for redefinition.

## 5-Integration

In this folder you have different files:

* **logical-system.xml:**

Thisfile allows declaring high-level physical characteristics of target ECOA platforms resources like processor architecture, operating system, available memory, endianness, computing node. Several platforms can be defined to allow a multi-platforms deployment (supposing then platforms compliance with optional ECOA implementation)

**Logical-system.xml file Example:**

|  |
| --- |
| <ecoa:logicalSystem id=*"cs1"*  xmlns:ecoa=*"http://www.ecoa.technology/logicalsystem-2.0">*  <!--  Computing Node = « Alienware Aurora » desktop PC  HyperThreading disabled (BIOS config)  4 CPU cores  Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz  Bogomips :  6785.34  (dmesg | grep BogoMIPS)  stepDuration = 1/BogoMIPS = 1.47376e-4 s  -->  <logicalComputingPlatform id=*"myPlatform">*  <logicalComputingNode id=*"machine0">*  <endianess type=*"BIG" />*  <logicalProcessors number=*"4" type="x86\_64">*  <stepDuration nanoSeconds=*"147376" />*  </logicalProcessors>  <os name=*"linux" />*  <availableMemory gigaBytes=*"6" />*  <moduleSwitchTime microSeconds=*"10" />*  </logicalComputingNode>  </logicalComputingPlatform>  </ecoa:logicalSystem> |

* **\*\*\*.impl.composite:**

This XML file allows declaring the software level system architecture. It consists of declaring the assembly of component instances, considering chosen component implementation for each component instance. Consequently, this XML file is a software solution to the logical system architecture previously defined at component type level (in demo.composite file).

**Demo.impl.composite file Example :**

|  |
| --- |
| <?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-extension-2.0"*  name=*"demo"*  targetNamespace=*"http://www.ecoa.technology/sca-extension-2.0">*  <csa:component name=*"demoPing">*  <ecoa-sca:instance componentType=*"Ping">*  <ecoa-sca:implementation name=*"myDemoPing"/>*  </ecoa-sca:instance>  <csa:reference name=*"svc\_PingPong"/>*  </csa:component>  <csa:component name=*"demoPong">*  <ecoa-sca:instance componentType=*"Pong">*  <ecoa-sca:implementation name=*"myDemoPong"/>*  </ecoa-sca:instance>  <csa:service name=*"svc\_PingPong"/>*  </csa:component>  <csa:wire source=*"demoPing/svc\_PingPong" target="demoPong/svc\_PingPong"/>*  </csa:composite> |

* **\*\*\*.deployment.xml:**

This XML file allows mapping the ECOA SW architecture onto computing nodes of the target ECOA computing platforms. This file is used by each ECOA computing platform for configuring fault handler notifications, loading, and deploying its associated components conformly to component implementations stored in « 4-ComponentImplementations ».

**demo.deployment.xml file Example:**

|  |
| --- |
| <deployment finalAssembly=*"demo" logicalSystem="logical\_system"*  xmlns=*"http://www.ecoa.technology/deployment-2.0">*  <protectionDomain name=*"Ping\_PD">*  <executeOn computingNode=*"machine0"* computingPlatform=*"myPlatform"/>*  <deployedModuleInstance componentName=*"demoPing" moduleInstanceName="myDemoPing\_AM\_I"modulePriority="30"/>*  <deployedTriggerInstance componentName=*"demoPing" triggerInstanceName="Heart\_Beat" triggerPriority="10"/>*  </protectionDomain>  <protectionDomain name=*"Pong\_PD">*  <executeOn computingNode=*"machine0"*computingPlatform=*"myPlatform"/>*  <deployedModuleInstance componentName=*"demoPong" moduleInstanceName="myDemoPong\_AM\_I" modulePriority="30"/>*  </protectionDomain>  <platformConfiguration computingPlatform=*"myPlatform"* faultHandlerNotificationMaxNumber=*"8"* />  </deployment> |

* **Other integration files and directories:**
  + META-INF :

sca-contribution.xml

* + inc :

It contains the header file machine0\_fault\_handler\_user\_context.h and the user\_context structure declaration

* + src :
    - machine0\_fault\_handler.c

This file contains the error\_notification function definition.

inc/ and src/ directories allow defining files to configure the ECOA Fault Handler, for platforms on which the ECOA Fault Handler is implemented as a function within the infrastructure rather than an ASC. This example illustrates a platform on which the ECOA Fault Handler is implemented as a function of the ECOA infrastructure. In that case, platform documentation gives ECOA Fault Handler level (platform or node), which allows defining files and functions names. Files content has then to be filled conformly to expected behaviour in case of error. For more information about the fault handler go to § 8.4.

## Project file

The file <Name\_of\_the\_application>.project.xml contains paths to the files needed to generate an ECOA application. In addition of these files, you also have the output directory which is the directory where the generated binaries would be deployed.

Example: **PingPong.project.xml**

|  |
| --- |
| <ECOAProject xmlns=*"http://www.ecoa.technology/project-2.0"* name=*"PingPong"*>  <types>  <file>0-Types/pingpong.types.xml</file>  </types>  <serviceDefinitions>  <file>1-Services/svc\_PingPong.interface.xml</file>  </serviceDefinitions>  <componentDefinitions>  <file>2-ComponentDefinitions/Ping/Ping.componentType</file>  <file>2-ComponentDefinitions/Pong/Pong.componentType</file>  </componentDefinitions>  <initialAssembly>3-InitialAssembly/demo.composite</initialAssembly>  <componentImplementations>  <file>4-ComponentImplementations/myDemoPing/myDemoPing.impl.xml</file>  <file>4-ComponentImplementations/myDemoPong/myDemoPong.impl.xml</file>  </componentImplementations>  <implementationAssembly>5-Integration/demo.impl.composite</implementationAssembly>  <deploymentSchema>5-Integration/demo.deployment.xml</deploymentSchema>  <logicalSystem>5-Integration/cs1.logical-system.xml</logicalSystem>  <outputDirectory>..</outputDirectory>  </ECOAProject> |

Note: Operations, operation parameters and modules names must not clash with reserved words of the target language.

**Example:** You are not allowed to use a parameter called “NULL” if you want to generate a C/C++ code.

# EXVT Tool

**X**

## Principle

The aim of the ECOA XML Validation Tool is to check the collection of XML files provided by an ECOA application. Different types of verifications are performed syntax, semantic and consistency between all XML files.

The tool can be used separately to validate an ECOA application, but it can also be used by other tools to do the validation steps before executing their own treatments.

## Partial validation

The ECOA application validation by default is performed on the complete description of the ECOA application.

It is also possible to run the validation only on a part of the ECOA application’s description. This can be achieved by specifying one the six ECOA levels (see § 5.1), in this case only the XML files belonging to an inferior level or the specified level are considered.

# ASCTG Tool

## Principle

The aim of the ASC Test Generator tool is to generate a HARNESS component for testing the components specified in the ASCTG configuration file.

This HARNESS component replace the subset of components not present in the ASCTG configuration file.

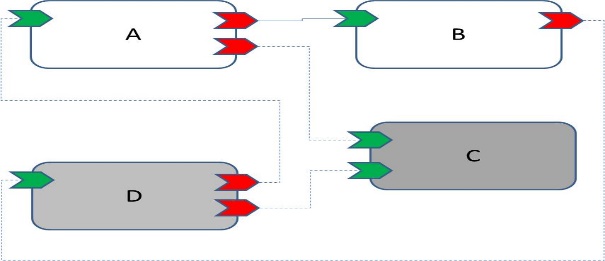
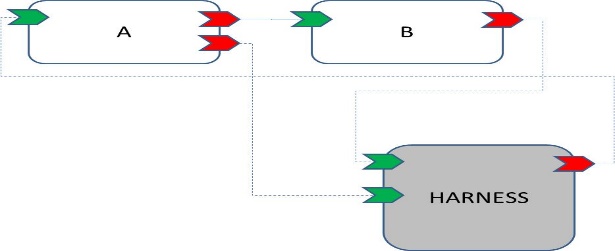
## Example

### Overview

* The provider F use the ASCTG configuration file to select A and B.
* The ASCTG tool then remove other components than A and B of the “HARNESS” assembly (associated wires are destroyed but wires between A and B are kept.
* The ASCTG tool create a HARNESS component presenting a “mirroring” service for each service of A and B not connected in the “HARNESS” assembly.
* The ASCTG tool add the new HARNESS component in the “HARNESS” assembly and add wires between each service not connected of A and B, and its mirror service in the HARNESS component.
* The ASCTG tool add a deployment of A, B and HARNESS components

### Schema

The following schema describes the ECOA application initial assembly vs the “HARNESS” assembly.



**X**

### Configuration file

The following configuration file allows the selection of A and B components.

|  |
| --- |
| <asctg*>*  <components*>*  <componentInstance>A</componentInstance>  <componentInstance>B</componentInstance>  </components>  </asctg> |

### Output files

The ASCTG tool will create:

* A new component through HARNESS\_type.componentType in 2-ComponentDefinitions/HARNESS\_type
* A new component implementation HARNESS.impl.xml in 4-ComponentImplementations/HARNESS composed of one module for each connected component, one trigger with a period of 160 ms for each module.

The ASCTG tool will also update:

* xx-harness.deployment.xml: By adding a protection domain HARNESS\_PD which is deployed on an existing platform regarding the other protection domains.
* xx-harness.impl.composite: By adding a harness component and new wires.
* xx-harness.project.xml: By adding new files to parse.

# LDP Tool

## Principle

The aim of the Lightweight Development Platform Tool is to generate the source code of an ECOA application.

The generated source code is composed of C/C++ files and cmake files.

## LDP Mechanism

### Fault Handler

Fault Management is performed at various levels within the infrastructure. The aim is to handle faults in such a way as to isolate and minimize fault propagation between Components.

* Error Detection is the general term for detecting an error wherever it is detected
* Fault Handling is the general term for handling a fault wherever it is handled

Fault management involves a pattern made of:

* Detection of errors
* Decision of recovery actions to be performed
* Execution of decided recovery actions.

### 

#### Table of errors managed by the fault handler

Here is a table listing all supported errors by the Fault Handler:

|  |  |  | ECOA Asset | |  |
| --- | --- | --- | --- | --- | --- |
| Error | Value | Description | Component | Protection Domain (\*) | Type |
| RESOURCE\_NOT\_AVAILABLE | 0 | No more resources to carry on the element activities | x | x | LDP Framework |
| UNAVAILABLE | 1 | The element (potentially a remote platform) has disappeared for an unknown reason | x | x | LDP Framework |
| MEMORY\_VIOLATION | 2 | Memory violation |  | x | Error signal SIGSEGV |
| NUMERICAL\_ERROR | 3 | Divide by zero or floating-point error |  | x | Error signal SIGFPE |
| ILLEGAL\_INSTRUCTION | 4 | Illegal instruction in the binary code |  | x | Error signal SIGILL |
| STACK\_OVERFLOW | 5 | Module or protection domain stack corruption |  | x | Error signal SIGBUS |
| OVERFLOW | 7 | The module queue is full or if the container has not enough resources to track concurrent requests. | x |  | LDP Framework |
| UNDERFLOW | 8 | The module queue is not enough fulfilled | x |  | LDP Framework |
| ERROR | 11 | Raise\_error called by a Module | x |  | ECOA Application |
| FATAL\_ERROR | 12 | Raise\_fatal\_error called by a Module | x |  | ECOA Application |
| COMMUNICATION\_ERROR | 15 | Communication error | x |  | LDP Framework |
| INITIALISATION\_PROBLEM | 17 | Initialisation problem. The node is not able to allocate resources or to start components | x |  | LDP Framework |

(\*) Only errors at the Protection Domain level are eligible to recovery action.

#### Error notification API

##### Types definition

The following types used by Error Notification API are defined in ECOA.h file.

*/\* ECOA:error\_id \*/*

*typedef ECOA\_\_uint32 ECOA\_\_error\_id;*

*/\* ECOA:asset\_id \*/*

*typedef ECOA\_\_uint32 ECOA\_\_asset\_id;*

*/\* ECOA:asset\_type \*/*

*typedef ECOA\_\_uint32 ECOA\_\_asset\_type;*

*#define ECOA\_\_asset\_type\_COMPONENT (0)*

*#define ECOA\_\_asset\_type\_PROTECTION\_DOMAIN (1)*

*/\* ECOA:error\_type \*/*

*typedef ECOA\_\_uint32 ECOA\_\_error\_type;*

*#define ECOA\_\_error\_type\_RESOURCE\_NOT\_AVAILABLE (0)*

*#define ECOA\_\_error\_type\_UNAVAILABLE (1)*

*#define ECOA\_\_error\_type\_MEMORY\_VIOLATION (2)*

*#define ECOA\_\_error\_type\_NUMERICAL\_ERROR (3)*

*#define ECOA\_\_error\_type\_ILLEGAL\_INSTRUCTION (4)*

*#define ECOA\_\_error\_type\_STACK\_OVERFLOW (5)*

*#define ECOA\_\_error\_type\_DEADLINE\_VIOLATION (6)*

*#define ECOA\_\_error\_type\_OVERFLOW (7)*

*#define ECOA\_\_error\_type\_UNDERFLOW (8)*

*#define ECOA\_\_error\_type\_ERROR (11)*

*#define ECOA\_\_error\_type\_FATAL\_ERROR (12)*

*#define ECOA\_\_error\_type\_INITIALISATION\_PROBLEM (17)*

##### Source file

The error notification source file path is:

<app\_name>/5-Integration/src/<platform\_name>\_fault\_handler.c

It contains the implementation of the <platform\_name>\_\_error\_notification function following the prototypes given by ECOA standard (**[R-8] § 10.3**):

|  |
| --- |
| *void* <platform\_name>*\_\_error\_notification (*  *ldp\_fault\_handler\_context\* 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\_\_uint32 error\_code)*  *{*  */\* Implementation \*/*  *}* |

The implementation of this function provides the way to define the Fault Handler recovery policy dedicated to the application.

The explanation of parameters is provided by ECOA standard (**[R-8] § 10.3**).

##### Example

This example of an error notification implementation is extracted from the *PingPong* application

(03\_codeo\_application/src/parsec/PingPong/5-Integration/src/myPlatform\_fault\_handler.c).

In this example “myPlatform” refers to the name of the platform. It should be adapted regarding the name of the platform used in the current application.

void myPlatform\_\_error\_notification **(**

ldp\_fault\_handler\_context**\*** 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\_\_uint32 error\_code**)**

**{**

ldp\_Main\_ctx**\*** ctx**=(**ldp\_Main\_ctx**\*)** context**->**platform\_hook**;**

ldp\_log\_PF\_log\_var**(**ECOA\_LOG\_INFO\_PF**,** "INFO"**,** ctx**->**logger\_PF**,** "[MAIN] Fault Handler NOTIFICATION: [%d:%d] error\_id=%d asset\_id=%d asset\_type=%d error\_type=%d error\_code=%d"**,** timestamp**.**seconds**,** timestamp**.**nanoseconds**,**error\_id**,** asset\_id**,** asset\_type**,** error\_type**,** error\_code**);**

ECOA\_\_log log**;**

**if** **(**asset\_type **==** ECOA\_\_asset\_type\_PROTECTION\_DOMAIN**)** **{**

**if** **(**error\_type **==** ECOA\_\_error\_type\_RESOURCE\_NOT\_AVAILABLE**)** **{**

log**.**current\_size **=** snprintf **((**char**\*)**log**.**data**,** ECOA\_\_LOG\_MAXSIZE**,** "[MAIN] Fault Handler NOTIFICATION: RESOURCE\_NOT\_AVAILABLE error received for PD=%d ..."**,** asset\_id**);**

myPlatform\_container\_\_log\_info**(**context**,** log**);**

**if** **(**myPlatform\_container\_\_recovery\_action **(**context**,**

ECOA\_\_recovery\_action\_type\_COLD\_RESTART**,**

asset\_id**,**

asset\_type**)** **!=** ECOA\_\_return\_status\_OK**)** **{**

log**.**current\_size **=** snprintf **((**char**\*)**log**.**data**,** ECOA\_\_LOG\_MAXSIZE**,** "[MAIN] Fault Handler NOTIFICATION: failed to restart PD=%d ..."**,** asset\_id**);**

myPlatform\_container\_\_log\_error**(**context**,** log**);**

**}**

**}** **else** **if** **(**error\_type **==** ECOA\_\_error\_type\_UNAVAILABLE**)** **{**

log**.**current\_size **=** snprintf **((**char**\*)**log**.**data**,** ECOA\_\_LOG\_MAXSIZE**,** "[MAIN] Fault Handler NOTIFICATION: UNAVAILABLE error received for PD=%d ..."**,** asset\_id**);**

myPlatform\_container\_\_log\_info**(**context**,** log**);**

**if** **(**myPlatform\_container\_\_recovery\_action **(**context**,**

ECOA\_\_recovery\_action\_type\_SHUTDOWN**,**

asset\_id**,**

asset\_type**)** **!=** ECOA\_\_return\_status\_OK**)** **{**

log**.**current\_size **=** snprintf **((**char**\*)**log**.**data**,** ECOA\_\_LOG\_MAXSIZE**,** "[MAIN] Fault Handler NOTIFICATION: failed to shutdown PD=%d ..."**,** asset\_id**);**

myPlatform\_container\_\_log\_error**(**context**,** log**);**

**}**

**}** **else** **{**

log**.**current\_size **=** snprintf **((**char**\*)**log**.**data**,** ECOA\_\_LOG\_MAXSIZE**,** "[MAIN] Fault Handler NOTIFICATION: signal error received for PD=%d ..."**,** asset\_id**);**

myPlatform\_container\_\_log\_info**(**context**,** log**);**

**if** **(**myPlatform\_container\_\_recovery\_action **(**context**,**

ECOA\_\_recovery\_action\_type\_COLD\_RESTART**,**

asset\_id**,**

asset\_type**)** **!=** ECOA\_\_return\_status\_OK**)** **{**

log**.**current\_size **=** snprintf **((**char**\*)**log**.**data**,** ECOA\_\_LOG\_MAXSIZE**,** "[MAIN] Fault Handler NOTIFICATION: failed to restart PD=%d ..."**,** asset\_id**);**

myPlatform\_container\_\_log\_error**(**context**,** log**);**

**}**

**}**

**}else{**

log**.**current\_size **=** snprintf **((**char**\*)**log**.**data**,** ECOA\_\_LOG\_MAXSIZE**,** "[MAIN] Fault Handler NOTIFICATION: error '%d' received for MOD\_ID=%d ..."**,** error\_type**,** asset\_id**);**

myPlatform\_container\_\_log\_info**(**context**,** log**);**

**}**

**}**

##### Error\_code parameter

The error notification API have an additional parameter. This error\_code is used to provide the location in the code where the error has occurred.

For each occurrence of an error\_type in the code, a value of error\_code is provided, so every time an error\_type trace is logged, it allows to identify where it is coming from.

As an example, let’s look at the following log:

|  |
| --- |
| *"1627467299,677227060":1:"INFO":"main\_PD":"main\_node":"[MAIN] Fault Handler NOTIFICATION: [1627467299:677207363] error\_id=0 asset\_id=2 asset\_type=0 error\_type=0* ***error\_code=15****"* |

The *error\_type RESOURCE\_NOT\_AVAILABLE (id=0)* is displayed with an *error\_code* set to 15.

Looking for *error\_type “RESOURCE\_NOT\_AVAILABLE”* in the generated source code, we found:

|  |
| --- |
| *$ grep -ri ECOA\_\_error\_type\_RESOURCE\_NOT\_AVAILABLE app.rootfs*  *app.rootfs/PingPong/0-Types/inc-gen/ECOA.h:#define ECOA\_\_error\_type\_RESOURCE\_NOT\_AVAILABLE (0)*  *app.rootfs/PingPong/0-Types/inc-gen/ECOA.h:#define ECOA\_\_error\_type\_RESOURCE\_NOT\_AVAILABLE (0)*  *app.rootfs/PingPong/4-ComponentImplementations/myDemoPing/myDemoPing\_AM/src/myDemoPing\_AM.c: ldp\_send\_fault\_error\_to\_father(demoPing\_ctx, getpid(), ECOA\_\_asset\_type\_PROTECTION\_DOMAIN, ECOA\_\_error\_type\_RESOURCE\_NOT\_AVAILABLE, 0);*  ***app.rootfs/PingPong/4-ComponentImplementations/myDemoPing/myDemoPing\_AM/src-gen/myDemoPing\_AM\_container.c: ECOA\_\_error\_type\_RESOURCE\_NOT\_AVAILABLE, 15);***  *app.rootfs/PingPong/4-ComponentImplementations/myDemoPing/myDemoPing\_AM/src-gen/myDemoPing\_AM\_container.c: ECOA\_\_error\_type\_RESOURCE\_NOT\_AVAILABLE, 16);*  *app.rootfs/PingPong/4-ComponentImplementations/myDemoPong/myDemoPong\_AM/src/myDemoPong\_AM.c: ldp\_send\_fault\_error\_to\_father(demoPong\_ctx, getpid(), ECOA\_\_asset\_type\_PROTECTION\_DOMAIN, ECOA\_\_error\_type\_RESOURCE\_NOT\_AVAILABLE, 0);*  *app.rootfs/PingPong/4-ComponentImplementations/myDemoPong/myDemoPong\_AM/src-gen/myDemoPong\_AM\_container.c: ECOA\_\_error\_type\_RESOURCE\_NOT\_AVAILABLE, 12);*  *app.rootfs/PingPong/4-ComponentImplementations/myDemoPong/myDemoPong\_AM/src-gen/myDemoPong\_AM\_container.c: ECOA\_\_error\_type\_RESOURCE\_NOT\_AVAILABLE, 13);*  *app.rootfs/PingPong/4-ComponentImplementations/myDemoPong/myDemoPong\_AM/src-gen/myDemoPong\_AM\_container.c: ECOA\_\_error\_type\_RESOURCE\_NOT\_AVAILABLE, 14);*  *app.rootfs/PingPong/5-Integration/src/myPlatform\_fault\_handler.c: if (error\_type == ECOA\_\_error\_type\_RESOURCE\_NOT\_AVAILABLE) {*  *app.rootfs/platform/lib/ECOA.h:#define ECOA\_\_error\_type\_RESOURCE\_NOT\_AVAILABLE (0)*  *app.rootfs/platform/lib/ldp\_mod\_container\_util.c: ECOA\_\_error\_type\_RESOURCE\_NOT\_AVAILABLE, 1);*  *app.rootfs/platform/lib/ldp\_mod\_container\_util.c: ECOA\_\_error\_type\_RESOURCE\_NOT\_AVAILABLE, 2);*  *app.rootfs/platform/lib/ldp\_mod\_container\_util.c: ECOA\_\_error\_type\_RESOURCE\_NOT\_AVAILABLE, 3);*  *app.rootfs/platform/lib/ldp\_mod\_container\_util.c: ECOA\_\_error\_type\_RESOURCE\_NOT\_AVAILABLE, 4);*  *app.rootfs/platform/lib/ldp\_mod\_container\_util.c: ECOA\_\_error\_type\_RESOURCE\_NOT\_AVAILABLE, 5);*  *app.rootfs/platform/lib/ldp\_mod\_container\_util.c: ECOA\_\_error\_type\_RESOURCE\_NOT\_AVAILABLE, 6);*  *app.rootfs/platform/lib/ldp\_mod\_container\_util.c: ECOA\_\_error\_type\_RESOURCE\_NOT\_AVAILABLE, 7);*  *app.rootfs/platform/lib/ldp\_mod\_container\_util.c: ECOA\_\_error\_type\_RESOURCE\_NOT\_AVAILABLE, 8);*  *app.rootfs/platform/lib/ldp\_mod\_container\_util.c: ECOA\_\_error\_type\_RESOURCE\_NOT\_AVAILABLE, 9);*  *app.rootfs/platform/lib/ldp\_mod\_container\_util.c: ECOA\_\_error\_type\_RESOURCE\_NOT\_AVAILABLE, 10);*  *app.rootfs/platform/lib/ldp\_mod\_container\_util.c: ECOA\_\_error\_type\_RESOURCE\_NOT\_AVAILABLE, 11);* |

*error\_code ‘15’* is set in file *myDemoPing\_AM\_container.c*.

Looking in this source code file, we can then find the function raising the *error\_type* *RESOURCE\_NOT\_AVAILABLE* (based on *error\_code ‘15’*)*.*

#### Raise error and raise fatal error API

These API are described by ECOA standard (**[SPEC\_ECOA\_PART4] § 11.3**).

It allows to raise error to the fault handler from the application.

##### 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);

##### 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);

The “module\_impl\_name” is the name of the module’s implementation.

The context is the module’s context.

### Multi-nodes

#### Activating multi-nodes

The activation of multi-nodes is done by specifying different computing nodes for protection domains deployment. This is achieved by modifying the xx.deployment.xml in 5-Integration folder.

**demo.deployment.xml file example:**

|  |
| --- |
| <deployment finalAssembly=*"demo" logicalSystem="logical\_system"*  xmlns=*"http://www.ecoa.technology/deployment-2.0">*  <protectionDomain name=*"Ping\_PD">*  <executeOn computingNode=*"****machine0****"* computingPlatform=*"myPlatform"/>*  <deployedModuleInstance componentName=*"demoPing" moduleInstanceName="myDemoPing\_AM\_I"modulePriority="30"/>*  <deployedTriggerInstance componentName=*"demoPing" triggerInstanceName="Heart\_Beat" triggerPriority="10"/>*  </protectionDomain>  <protectionDomain name=*"Pong\_PD">*  <executeOn computingNode=*"****machine1****"*computingPlatform=*"myPlatform"/>*  <deployedModuleInstance componentName=*"demoPong" moduleInstanceName="myDemoPong\_AM\_I" modulePriority="30"/>*  </protectionDomain>  <platformConfiguration computingPlatform=*"myPlatform"* faultHandlerNotificationMaxNumber=*"8"* />  </deployment> |

#### The nodes\_deployment.xml

When activating the multi-nodes, a nodes\_deployment.xml file is required in 5-Integration folder. This file is used to associate the ip addresses with the computing nodes where the protection domains are running on.

The following nodes\_deployment.xml file allows to set ip addresses for machine0 and machine1.

|  |
| --- |
| <nodesDeployment*>*  <logicalComputingNode id=”main” ipAddress=”10.0.1.3”/>  <logicalComputingNode id=”machine0” ipAddress=”10.0.1.4”/>  <logicalComputingNode id=”machine1” ipAddress=”10.0.1.3”/>  </components>  </nodesDeployment> |

#### The multi-node.py file

When activating multi-nodes LDP generate a file named multi-nodes.py, this file is used to launch the ECOA platform in multi-nodes instead launching directly the “platform” binary.

|  |
| --- |
| $ ./multi-nodes.py |

The file multi-nodes.py optionally allows the deployment of binaries and dependencies on the computing nodes before running the ECOA platform (using “–d” switch).

|  |
| --- |
| $ ./multi-nodes.py –d $ECOA\_DEPS\_DIR |

### local.cmake Mechanism

A module could use an external library by adding a cmake file in the component implementation directory. This cmake file extension is used by the main cmake.

For every component implementation, a static library is created. It is name “lib\_#ComponentName#”.

To add an external library to a component implementation, create a file named “local.cmake” in the directory “4-ComponentImplementation/#ComponentName#”.

Then write the following lines in “local.cmake” file:

* include\_directories(${CMAKE\_CURRENT\_LIST\_DIR}/#relative\_lib\_path#/include/)
* target\_sources(lib\_#ComponentName# PRIVATE ${CMAKE\_CURRENT\_LIST\_DIR}/#relative\_lib\_path#/src/\*.c)