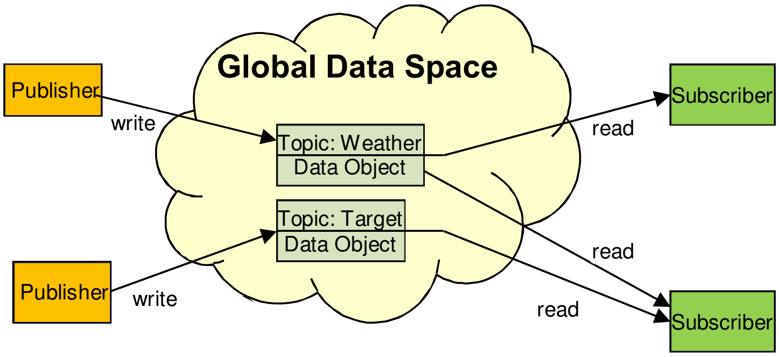
PubSub

Introduction

This document describes an ECOA® publish-subscribe example, named “PubSub”.

The publish-subscribe pattern (ref.[2]) is a data exchange pattern modelled on the newspaper or magazine distribution pattern. Information sources (publishers) do not send data (messages) directly to receivers (subscribers), but rather make the data available in classified topics to which an interested party subscribes and withdraws the information. Thus the publishers have no knowledge of which (if any) subscribers there are, and the subscribers have no knowledge of which publisher has produced the data. Figure 1 illustrates the conventional abstract view of the publish-subscribe pattern. In practise Subscribers to one data set might well be publishers of another…

Figure 1 Publish-Subscribe Data Distribution Pattern



The “Global Data Space” of Figure 1, may, or may not, have a physical existence depending on the implementation. Some implementations use a broker or agent known to both the publisher and subscriber and to which the data is published and temporarily held (fulfilling the “newsagent” role in the magazine analogy). The subscribers then obtain the data from that broker or agent. Other implementations have no middleman, and the data is exchanged directly between the publisher and the subscribers according to configuration meta-data discovered and shared by either at build, initialization, or runtime.

It is quite normal for subscribers to subscribe to partial data sets, using either:

- a topic-based mechanism (all subscribers get all updates to the data structured under a topic), and/or

- a content-based mechanism (a subscriber gets updates to only the data matching attributes or criteria predefined by the user, e.g. by using a query language to distinguish the content required).

This document presents the principal user generated artefacts required to create a "PubSub" publish-subscribe example using the ECOA. It is assumed that the reader is conversant with the ECOA Architecture Specification (ref.[1]) 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. An ability to follow UML diagrams is also assumed.

Aims

This ECOA "*PubSub*" example is intended to experiment with, and demonstrate, how the ECOA Versioned Data concept (see ref.[1]), facilitates the implementation of applications using the publish-subscribe data distribution pattern.

ECOA Features Exhibited

- Composition of an ECOA Assembly of multiple ECOA ASCs (components).
- Multiple cooperating ECOA Protection Domains.
- Publish-subscribe data distribution across a network..
- Experimental ECOA File Access API
- ECOA Logical Interface (ELI) between ECOA Protection Domains.
- ECOA Service Availability.
- ECOA Module Lifecycle Management.
- ECOA Runtime logging API.
- ECOA Component and Module Properties.

The example is also used as a vehicle for demonstrating that a Component can provide Services that are not used in a given ECOA Assembly, as may naturally occur when Components are reused.

Design and Definition

ECOA Publish-Subscribe Data Distribution Design

The ECOA "*PubSub*" publish-subscribe example will demonstrate a basic data distribution mechanism using ECOA Versioned Data, using two ECOA Protection Domains, possibly distributed across networked computing hosts.

In order to provide a measure of data scoping, individual data 'topics' will be mapped to ECOA Services, whilst Versioned Data items of a Service will provide access to subsets of the whole data set. The publisher will periodically update the Versioned Data items within the provided Services, and the subscriber will periodically read them. Figure 2 illustrates the publish-subscribe pattern (of Figure 1) interpreted, in UML, for this ECOA "*PubSub*" example, and identifies the topics and data items used.

Note that in Figure 2 the "read" arrow is reversed when compared with Figure 1, implying that the subscriber *gets* (requests) the data item rather than being *presented* with it. ECOA provides a Notifying Versioned Data mechanism where a subscriber is notified (by an ECOA Event operation) when the Versioned Data item is updated[1], but that mechanism is not used in this example. The subscriber ("*Sub*") must actively request (access to) the data[2].

Figure 2 ECOA "*PubSub*" Publish-Subscribe Pattern

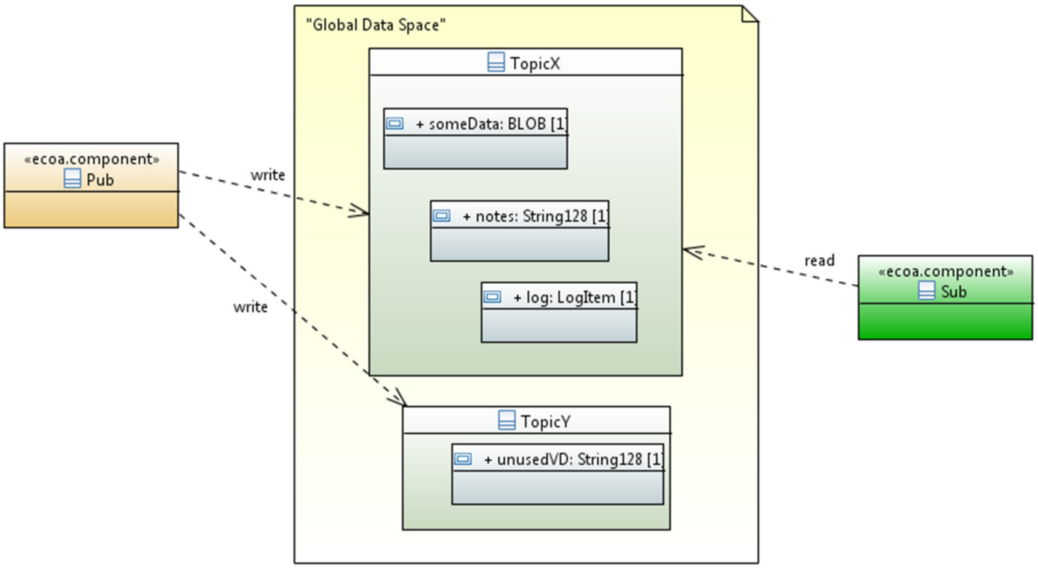


Figure 2 also illustrates the publisher ("*Pub*") publishing "*TopicY*" which, in this example, is never subscribed to; maybe because "*Pub*" has been re-used from a different system, or perhaps because the Component that once subscribed to "*TopicY*" has been removed...

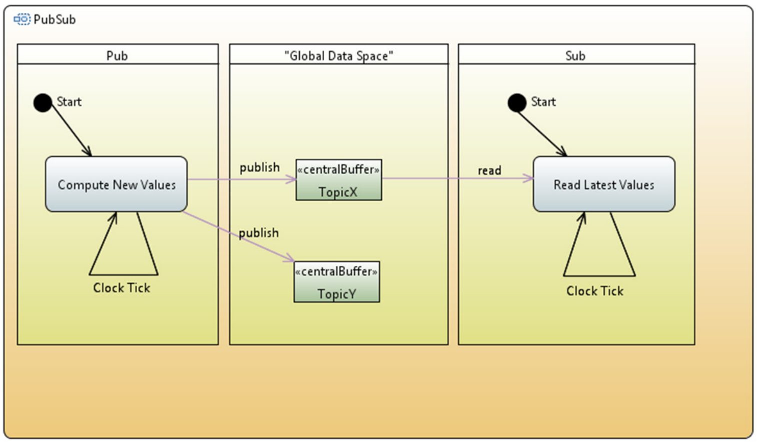
Behaviourally, the publisher and subscriber will be free-running software applications that interact only in as much as required to exchange data across the "Global Data Space" interface. That is, the publisher will continuously and periodically publish (update) the "TopicX" and "TopicY" data items. The subscriber will, independently, periodically read one or more of the "TopicX" data items.

---

[1] The subscriber, having been notified, must still *get* (request) the updated data item.

[2] Details of the data delivery mechanism for ECOA Versioned Data are described in ref.[1] (Part 3).

Figure 3 ECOA "*PubSub*" Example Behaviour



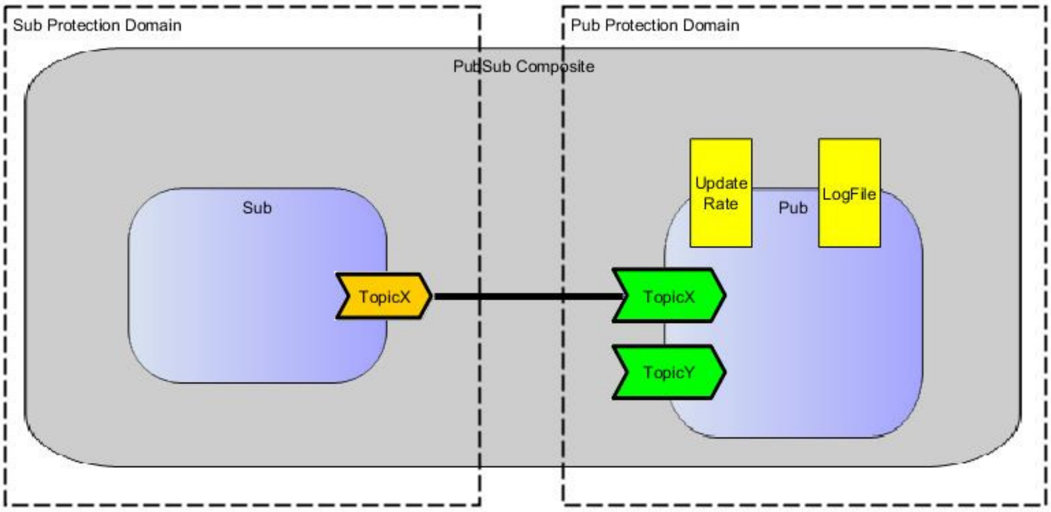
Note that in an ECOA implementation there is no runtime "subscribe" activity. The "subscription" is a design-time action when an ECOA Service *provides-references* association (Service Link) is declared.

ECOA Assembly Design and Definition

This ECOA "*PubSub*" example ECOA Assembly comprises two ECOA ASCs named "*Pub*" and "*Sub*" (of ASC types of the same name). The "*Pub*" ASC provides two ECOA Services, named "*TopicX*" and "*TopicY*". The "*TopicX*" Service is referenced by the "*Sub*" ASC.

The ECOA "*PubSub*" Assembly is depicted in Figure 4.

Figure 4 ECOA "*PubSub*" Assembly Diagram



The "*Pub*" ASC defines two ECOA Properties, "*UpdateRate*" and "*LogFile*":

- "*UpdateRate*" sets the period at which the ASC publishes data (in "*TopicX*"), and

- "*LogFile*" set the name and access permissions/attributes of a file that is used to log data within the ASC.

This ECOA Assembly is defined in an Initial Assembly XML file, and declared in a Final Assembly (or Implementation) XML file (which is practically identical). The Final Assembly XML for the ECOA "*PubSub*" Assembly is as follows (file *PubSub_impl.composite*), reflecting the Assembly diagram above:

```
<csa:composite xmlns:csa="http://docs.oasis-open.org/ns/opencsa/sca/200912"
     xmlns:ecoa-sca="http://www.ecoa.technology/sca" name="PubSub_impl"
     targetNamespace="http://www.ecoa.technology/sca">
     <csa:component name="Pub">
          <ecoa-sca:instance componentType="Pub">
               <ecoa-sca:implementation name="Pub"/>
          </ecoa-sca:instance>
          <csa:reference name="TopicX"/>
          <csa:property name="Update_Rate">
               <csa:value>4.0</csa:value>
          </csa:property>
          <csa:property name="LogFile">
               <csa:value>
                    (ECOA__File__FileSpec)
                    {"Pub_LogFile.txt",ECOA__File__FileFlag_WRITE|
                    ECOA__File__FileFlag_APPEND|ECOA__File__FileFlag_CREATE,
                    {ECOA__File__Permission_ALL, ECOA__File__Permission_READ,
                    ECOA__File__Permission_READ}}
               </csa:value>
          </csa:property>
     </csa:component>
     <csa:component name="Sub">
          <ecoa-sca:instance componentType="Sub">
               <ecoa-sca:implementation name="Sub"/>
          </ecoa-sca:instance>
          <csa:service name="TopicX"/>
     </csa:component>
     <!-- System Wiring... -->
     <csa:wire source="Sub/TopicX" target="Pub/TopicX" ecoa-sca:rank="1"/>
</csa:composite>
```

The *Sub* ASC type is defined in XML as follows (file *Sub.componentType*):

```
<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">
     <reference name="TopicX">
          <ecoa-sca:interface syntax="TopicX"/>
     </reference>
</componentType>
```

that is, the ASC *references* a single Service (*TopicX*).

The *Pub* ASC type is defined in XML as follows (file *Pub.componentType*):

```
<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">
     <use library="ECOA.File"/>
     <service name="TopicX">
          <ecoa-sca:interface syntax="TopicX"/>
     </service>
     <service name="TopicY">
          <ecoa-sca:interface syntax="TopicY"/>
     </service>
     <property name="LogFile" type="xs:string"
                                   ecoa-sca:type="ECOA.File:FileSpec">
     </property>
     <property name="Update_Rate" type="xs:string" ecoa-sca:type="float32"/>
</componentType>
```

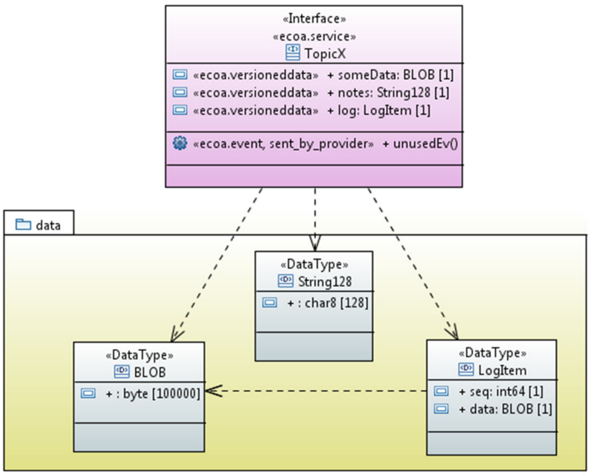
that is, in addition to *providing* the `TopicX` and `TopicY` Services, the ASC defines the ECOA Component Properties (`Update_Rate` and `LogFile`). Note that the `Update_Rate` and `LogFile` Property *values* are given (uniquely) for each instance of the ASC in the *PubSub* Assembly declaration (above), not here in the ASC type definition. Note that the `LogFile` Property value is presently expressed in C code, not the Property Value expression language defined in ref.[1] (Part 4).

The definition of the data type of the `LogFile` Property (`FileSpec`) is to be found in the `ECOA.File` types library.

ECOA Service and Types Definition

The `TopicX` Service, which is provided by the *Pub* ASC and referenced by the *Sub* ASC, is depicted in UML in Figure 5, and defined in a XML file (`TopicX.interface.xml`):

Figure 5 ECOA Service "*TopicX*" Definition (Depicted in UML)

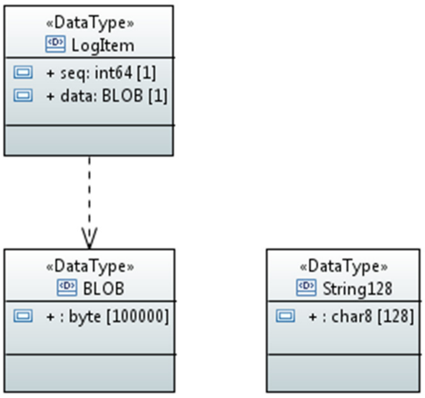

«Interface»
«ecoa.service»
TopicX
«ecoa.versioneddata» + someData: BLOB [1]
«ecoa.versioneddata» + notes: String128 [1]
«ecoa.versioneddata» + log: LogItem [1]
«ecoa.event, sent_by_provider» + unusedEv()
data
«DataType»
String128
+ : char8 [128]
«DataType»
BLOB
+ : byte [100000]
«DataType»
LogItem
+ seq: int64 [1]
+ data: BLOB [1]

```
<serviceDefinition xmlns="http://www.ecoa.technology/interface-1.0">
     <use library="data"/>
     <operations>
          <data name="someData" type="data:BLOB"/>
          <data name="notes" type="data:String128"/>
          <data name="log" type="data:LogItem"/>
          <!-- Experimental additions -->
          <event name="unusedEv" direction="SENT_BY_PROVIDER"/>
     </operations>
</serviceDefinition>
```

The Service comprises three ECOA Versioned Data items (`someData`, `notes`, and `log`) and an ECOA Event operation (`unusedEv`). This latter is only present to demonstrate that the development process, tooling, and particularly the runtime ECOA Software Platform code, allow for ECOA operations that in any given ECOA Assembly are not used.

The data types of the three Versioned Data items are declared in the (*PubSub*) "*data*" types library, depicted in UML in Figure 6, and defined in XML (file `data.types.xml`) as:

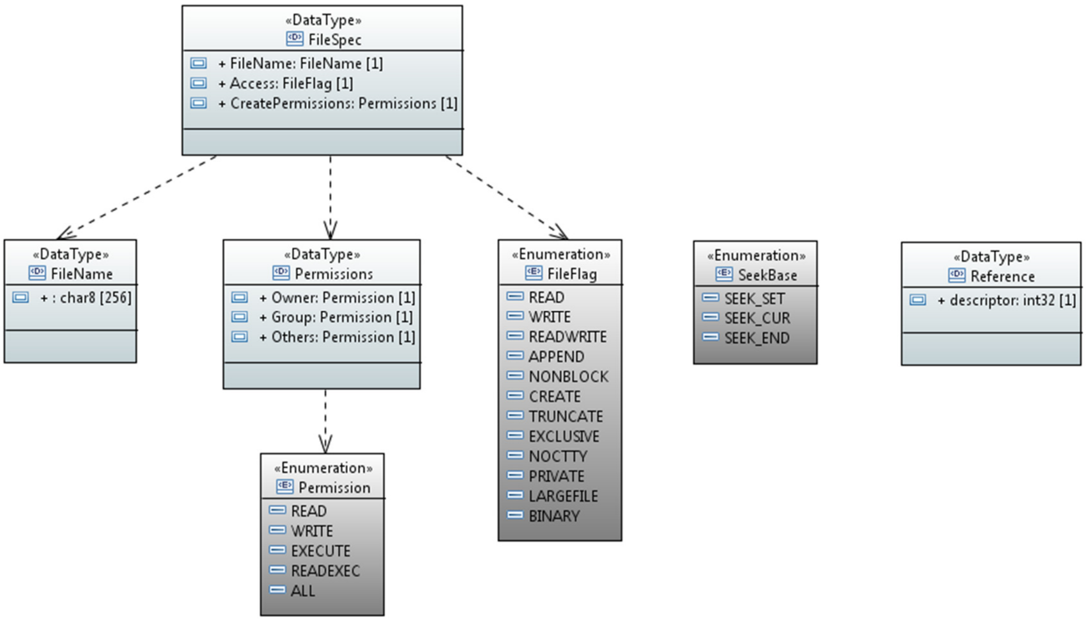
Figure 6 ECOA Types Library (PubSub) "*data*" Definition (Depicted in UML)



```
<library xmlns="http://www.ecoa.technology/types-1.0">
     <types>
          <array name="BLOB" maxNumber="100000" itemType="byte"/>
          <fixedArray name="String128" maxNumber="128" itemType="char8"/>
          <record name="LogItem">
               <field name="seq" type="int64"/>
               <field name="data" type="BLOB"/>
          </record>
     </types>
</library>
```

The experimental File Access API also requires certain data types, depicted in UML in Figure 7, and defined in XML in a types library (file `ECOA.File.data.types`):

Figure 7 ECOA Types Library "*ECOA.FIle*" Definition (Depicted in UML)



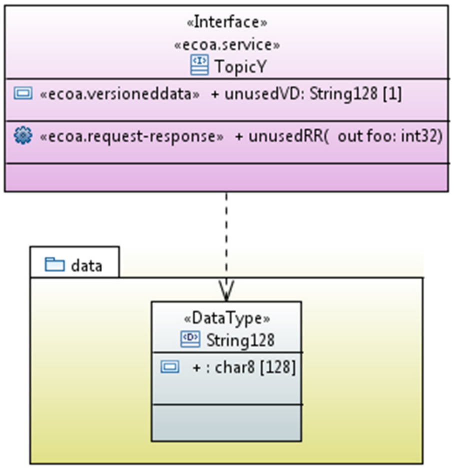
```
<library xmlns="http://www.ecoa.technology/types-1.0">
      <types>
            <record name="Reference">
                  <field name="descriptor" type="int32"/>
            </record>
            <fixedArray name="FileName" maxNumber="256" itemType="char8"/>
            <enum name="FileFlag" type="uint16">
                  <value name="READ"      valnum=    "0"/><!-- "0x0000" -->
                  <value name="WRITE"     valnum=    "1"/><!-- "0x0001" -->
                  <value name="READWRITE" valnum=    "2"/><!-- "0x0002" -->
                  <value name="APPEND"    valnum=    "8"/><!-- "0x0008" -->
                  <value name="NONBLOCK"  valnum=  "128"/><!-- "0x0080" -->
                  <value name="CREATE"    valnum=  "256"/><!-- "0x0100" -->
                  <value name="TRUNCATE"  valnum=  "512"/><!-- "0x0200" -->
                  <value name="EXCLUSIVE" valnum= "1024"/><!-- "0x0400" -->
                  <value name="NOCTTY"    valnum= "2048"/><!-- "0x0800" -->
                  <value name="PRIVATE"   valnum= "4096"/><!-- "0x1000" -->
                  <value name="LARGEFILE" valnum= "8192"/><!-- "0x2000" -->
                  <value name="BINARY"    valnum="16384"/><!-- "0x4000" -->
            </enum>
            <enum name="SeekBase" type="uint8">
                  <value name="SEEK_SET"/>
                  <value name="SEEK_CUR"/>
                  <value name="SEEK_END"/>
            </enum>
```

```
             <enum name="Permission" type="uint8">
                   <value name="READ"       valnum=    "4"/>
                   <value name="WRITE"      valnum=    "2"/>
                   <value name="EXECUTE"    valnum=    "1"/>
                   <value name="READEXEC"   valnum=    "5"/>
                   <value name="ALL"        valnum=    "7"/>
             </enum>
             <record name="Permissions">
                   <field name="Owner" type="Permission"/>
                   <field name="Group" type="Permission"/>
                   <field name="Others" type="Permission"/>
             </record>
             <record name="FileSpec">
                   <field name="FileName" type ="FileName"/>
                   <field name="Access" type="FileFlag"/>
                   <field name="CreatePermissions" type="Permissions"/>
             </record>
      </types>
</library>
```

that is, a number of enumeration and record types are defined that are used to define and control access to data file objects.

The `TopicY` ECOA Service is defined only to demonstrate that ECOA Services can be defined, declared as being *provided* by an ASC, and then not used (*referenced*) in any given Assembly. It is depicted in UML in Figure 8, and defined, as always, in a XML file (`TopicY.interface.xml`) and comprises an ECOA Versioned Data item (`unusedVD`) and an ECOA Request-Response operation (`unusedRR`):

Figure 8 ECOA Service "*TopicY*" Definition (Depicted in UML)

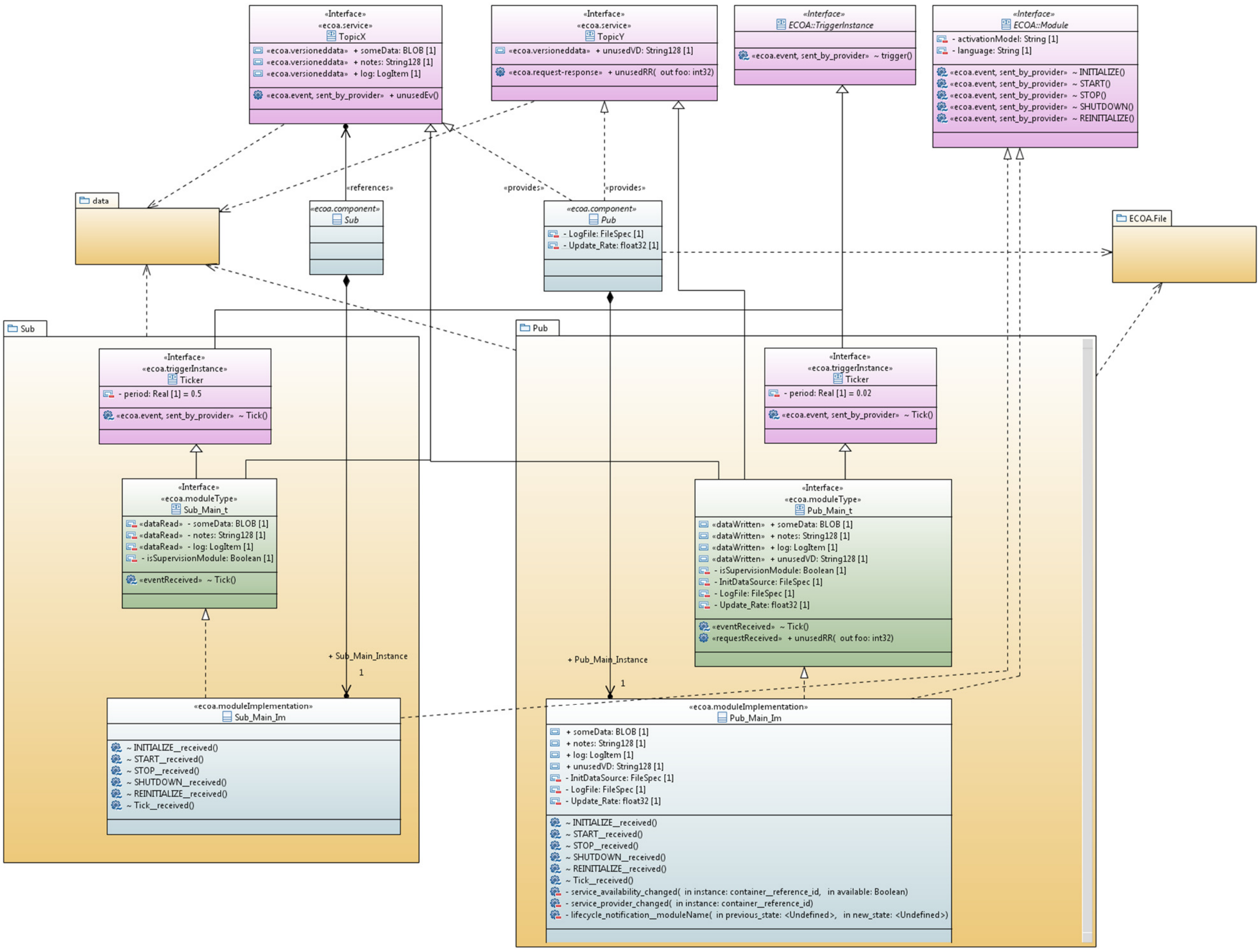

«Interface»
«ecoa.service»
TopicY
«ecoa.versioneddata» + unusedVD: String128 [1]
«ecoa.request-response» + unusedRR(out foo: int32)
data
«DataType»
String128
+ : char8 [128]

```
<serviceDefinition xmlns="http://www.ecoa.technology/interface-1.0">
     <use library="data"/>
     <operations>
          <data name="unusedVD" type="data:String128"/>
          <requestresponse name="unusedRR">
               <output name="foo" type="int32"/>
          </requestresponse>
     </operations>
</serviceDefinition>
```

ECOA Module Design and Definition

The `Pub` and `Sub` ASC (component) types are composed of a single ECOA Module each (Module Implementations `Pub_Main_Im` and `Sub_Main_Im` of Module Types `Pub_Main_t` and `Sub_Main_t` respectively) as illustrated in UML in Figure 9. Here is depicted in UML the `Pub` ASC (component) ***providing*** the `TopicX` ECOA Service, whilst the `Sub` ASC ***references*** the Service. As always in the ECOA, the Module Implementations implement the Module Lifecycle operations defined by the ECOA (as represented in UML by the interface class `ECOA::Module`).

Figure 9 ECOA "*PubSub*" Module Design (as UML Class Diagram)


«Interface»
«ecoa.service»
TopicX
«ecoa.versioneddata» + someData: BLOB [1]
«ecoa.versioneddata» + notes: String128 [1]
«ecoa.versioneddata» + log: LogItem [1]
«ecoa.event, sent_by_provider» + unusedEv()
«Interface»
«ecoa.service»
TopicY
«ecoa.versioneddata» + unusedVD: String128 [1]
«ecoa.request-response» + unusedRR(out foo: int32)
«Interface»
ECOA::TriggerInstance
«ecoa.event, sent_by_provider» ~ trigger()
«Interface»
ECOA::Module
- activationModel: String [1]
- language: String [1]
«ecoa.event, sent_by_provider» ~ INITIALIZE()
«ecoa.event, sent_by_provider» ~ START()
«ecoa.event, sent_by_provider» ~ STOP()
«ecoa.event, sent_by_provider» ~ SHUTDOWN()
«ecoa.event, sent_by_provider» ~ REINITIALIZE()
data
«references»
«provides»
«provides»
«ecoa.component»
Sub
«ecoa.component»
Pub
- LogFile: FileSpec [1]
- Update_Rate: float32 [1]
ECOA.File
Sub
«Interface»
«ecoa.triggerInstance»
Ticker
- period: Real [1] = 0.5
«ecoa.event, sent_by_provider» ~ Tick()
«Interface»
«ecoa.moduleType»
Sub_Main_t
«dataRead» - someData: BLOB [1]
«dataRead» - notes: String128 [1]
«dataRead» - log: LogItem [1]
- isSupervisionModule: Boolean [1]
«eventReceived» ~ Tick()
+ Sub_Main_Instance
1
«ecoa.moduleImplementation»
Sub_Main_Im
~ INITIALIZE__received()
~ START__received()
~ STOP__received()
~ SHUTDOWN__received()
~ REINITIALIZE__received()
~ Tick__received()
Pub
«Interface»
«ecoa.triggerInstance»
Ticker
- period: Real [1] = 0.02
«ecoa.event, sent_by_provider» ~ Tick()
«Interface»
«ecoa.moduleType»
Pub_Main_t
«dataWritten» + someData: BLOB [1]
«dataWritten» + notes: String128 [1]
«dataWritten» + log: LogItem [1]
«dataWritten» + unusedVD: String128 [1]
- isSupervisionModule: Boolean [1]
- InitDataSource: FileSpec [1]
- LogFile: FileSpec [1]
- Update_Rate: float32 [1]
«eventReceived» ~ Tick()
«requestReceived» + unusedRR(out foo: int32)
+ Pub_Main_Instance
1
«ecoa.moduleImplementation»
Pub_Main_Im
+ someData: BLOB [1]
+ notes: String128 [1]
+ log: LogItem [1]
+ unusedVD: String128 [1]
- InitDataSource: FileSpec [1]
- LogFile: FileSpec [1]
- Update_Rate: float32 [1]
~ INITIALIZE__received()
~ START__received()
~ STOP__received()
~ SHUTDOWN__received()
~ REINITIALIZE__received()
~ Tick__received()
- service_availability_changed(in instance: container__reference_id, in available: Boolean)
- service_provider_changed(in instance: container__reference_id)
- lifecycle_notification__moduleName(in previous_state: <Undefined>, in new_state: <Undefined>)

The Sub ASC

The *Sub* ASC is declared in XML as follows (file `Sub.impl.xml`):

```
<componentImplementation
     xmlns="http://www.ecoa.technology/implementation-1.0"
     componentDefinition="Sub">
     <use library="ECOA" />
     <use library="data" />
     <moduleType name="Sub_Main_t" isSupervisionModule="true">
          <operations>
               <dataRead name="someData" type="data:BLOB" />
               <dataRead name="notes" type="data:String128" />
               <dataRead name="log" type="data:LogItem" />
               <eventReceived name="Tick" />
          </operations>
     </moduleType>
     <moduleImplementation     name="Sub_Main_Im"
                               moduleType="Sub_Main_t"
                               activationModel="reactive"
                               language="C" />
     <moduleInstance name="Sub_Main_Instance"
                               implementationName="Sub_Main_Im"
                               relativePriority="1">
     </moduleInstance>
     <triggerInstance name="Ticker" relativePriority="2"/>
     <dataLink>
          <writers>
               <reference operationName="someData"
                               instanceName="TopicX"/>
          </writers>
          <readers>
               <moduleInstance operationName="someData"
                               instanceName="Sub_Main_Instance"/>
          </readers>
     </dataLink>
     <dataLink>
          <writers>
               <reference operationName="notes"
                               instanceName="TopicX"/>
          </writers>
          <readers>
               <moduleInstance operationName="notes"
                               instanceName="Sub_Main_Instance"/>
          </readers>
     </dataLink>
     <dataLink>
          <writers>
               <reference operationName="log" instanceName="TopicX"/>
          </writers>
          <readers>
               <moduleInstance operationName="log"
                               instanceName="Sub_Main_Instance"/>
          </readers>
     </dataLink>
```

```
		<eventLink>
			<senders>
				<trigger instanceName="Ticker" period="0.5" />
			</senders>
			<receivers>
				<moduleInstance instanceName="Sub_Main_Instance"
							   operationName="Tick" />
			</receivers>
		</eventLink>
	</componentImplementation>
```

That is, a Module Type (*Sub_Main_t*) is declared which has three *dataRead* operations, one for each ECOA Versioned Data item that is read, "*someData*", "*notes*" and "*Log*", inherited from the *TopicX* ECOA Service (depicted by the UML *generalization* association). An *«eventReceived»* operation is also declared, inherited from the ECOA Trigger Instance (*Ticker*). This Module Type is implemented (realized) by a concrete Module Implementation *Sub_Main_Im* (depicted in the UML expanded in the form of the code class produced by the code generation process), which in turn is instantiated at runtime as the Module Instance *Sub_Main_Instance*.

The *Ticker* Trigger Instance is introduced so that a periodic polling behaviour of *TopicX* can be implemented. Once every period (0.5 seconds as set in the *<eventLink>* XML[3]) the Trigger will fire and the Module Operation *Tick* will be invoked.

The Service Link (*<dataLink>* and *<eventLink>*) XML segments logically associate the specific concrete operations of the runtime Module Instance with the abstract Service operations, or in the case of the "Tick" operation, associates the concrete Module Operation to the Trigger Instance operation.

A single functional code unit will be produced by the code generation process, implementing in code the concrete *Sub_Main_Im* class, and named "*Sub_Main_Im.c*" (assuming the Module Implementation declaration has set the *Language* property to "C").

The Pub ASC

The *Pub* ASC is declared in XML as follows (file *Pub.impl.xml*):

```
	<componentImplementation xmlns=http://www.ecoa.technology/implementation-1.0
		componentDefinition="Pub">
		<use library="ECOA" />
		<use library="data" />
		<use library="ECOA.File"/>
		<moduleType name="Pub_Main_t" isSupervisionModule="true">
			<properties>
				<property name="InitDataSource" type="ECOA.File:FileSpec"/>
				<property name="LogFile" type="ECOA.File:FileSpec"/>
				<property name="Update_Rate" type="float32"/>
			</properties>
```

---

[3] The UML does not explicitly depict Service Links. The period attribute is therefore depicted as a UML property of the *«ecoa.triggerInstance»* UML interface class.

```
     <operations>
          <dataWritten name="someData" type="data:BLOB" />
          <dataWritten name="notes" type="data:String128" />
          <dataWritten name="log" type="data:LogItem" />
          <eventReceived name="Tick" />
          <!-- Experimental additions -->
          <dataWritten name="unusedVD" type="data:String128" />
          <eventSent name="unusedEv"/>
          <requestReceived name="unusedRR">
          <output name="foo" type="int32"/>
          </requestReceived>
     </operations>
</moduleType>
<moduleImplementation     name="Pub_Main_Im"
                          moduleType="Pub_Main_t"
                          activationModel="reactive"
                          language="C" />
<moduleInstance name="Pub_Main_Instance"
                          implementationName="Pub_Main_Im"
                          relativePriority="1">
     <propertyValues>
         <propertyValue name="InitDataSource">
                          (ECOA__File__FileSpec)
                          {"Pub_Init.dat",ECOA__File__FileFlag_READ|
                          ECOA__File__FileFlag_BINARY,
                          {ECOA__File__Permission_ALL,
                          ECOA__File__Permission_READEXEC,
                          ECOA__File__Permission_READ}}
         </propertyValue>
         <propertyValue name="LogFile">$LogFile</propertyValue>
         <propertyValue name="Update_Rate">$Update_Rate</propertyValue>
     </propertyValues>
</moduleInstance>
<triggerInstance name="Ticker" />
<dataLink>
     <writers>
          <moduleInstance operationName="someData"
                          instanceName="Pub_Main_Instance"/>
     </writers>
     <readers>
          <service operationName="someData" instanceName="TopicX"/>
     </readers>
</dataLink>
<dataLink>
     <writers>
          <moduleInstance operationName="notes"
                          instanceName="Pub_Main_Instance"/>
     </writers>
     <readers>
          <service operationName="notes" instanceName="TopicX"/>
     </readers>
</dataLink>
```

```
        <dataLink>
            <writers>
                <moduleInstance operationName="Log"
                                instanceName="Pub_Main_Instance"/>
            </writers>
            <readers>
                <service operationName="Log" instanceName="TopicX"/>
            </readers>
        </dataLink>
        <eventLink>
            <senders>
                <trigger instanceName="Ticker" period="0.02" />
            </senders>
            <receivers>
                <moduleInstance instanceName="Pub_Main_Instance"
                                operationName="Tick" />
            </receivers>
        </eventLink>
        <dataLink>
            <writers>
                <moduleInstance operationName="unusedVD"
                                instanceName="Pub_Main_Instance"/>
            </writers>
            <readers>
                <service operationName="unusedVD"
                                instanceName="TopicY"/>
            </readers>
        </dataLink>
        <eventLink>
            <senders>
                <moduleInstance operationName="unusedEv"
                                instanceName="Pub_Main_Instance"/>
            </senders>
            <receivers>
                <service operationName="unusedEv"
                                instanceName="TopicX"/>
            </receivers>
        </eventLink>
        <requestLink>
            <clients>
                <service operationName="unusedRR"
                                instanceName="TopicY"/>
            </clients>
            <server>
                <moduleInstance operationName="unusedRR"
                                instanceName="Pub_Main_Instance"/>
            </server>
        </requestLink>
    </componentImplementation>
```

That is, a Module Type (*Pub_Main_t*) is declared which has four *dataWritten* operations, one for each ECOA Versioned Data item that is written (published), "*someData*", "*notes*" and "*Log*", inherited from the *TopicX* ECOA Service, and "*unusedVD*" inherited from the *TopicY* ECOA Service (depicted by the UML *generalization* associations). An *«eventReceived»* operation is also declared, inherited from the ECOA Trigger Instance (*Ticker*). An *«eventSent»* and a *«requestReceived»* operation are also declared, inherited from the *TopicY* ECOA Service.

The two ASC-level ECOA Properties (*LogFile* and *Update_Rate*) are also declared, mapping the ASC level definitions into the Module Type. In addition, a Module(-level) Property is defined, *InitDataSource*, which is also of type *FileSpec* from the *ECOA.File* types library

This Module Type is implemented (realized) by a concrete Module Implementation *Pub_Main_Im* (depicted in the UML expanded in the form of the code class produced by the code generation process), which in turn is instantiated at runtime as the Module Instance *Pub_Main_Instance*.

The *Ticker* Trigger Instance is introduced so that a periodic publish behaviour of *TopicX* can be implemented. Once every period (20 milli-seconds as set in the *<eventLink>* XML[4]) the Trigger will fire and the Module Operation *Tick* will be invoked.

The Service Link (*<dataLink>, <eventLink>* and *<requestLink>*) XML segments logically associate the specific concrete operations of the runtime Module Instance with the abstract Service operations, or in the case of the “Tick” operation, associates the concrete Module Operation to the Trigger Instance operation.

A single functional code unit will be produced by the code generation process, implementing in code the concrete *Pub_Main_Im* class, and named “*Pub_Main_Im.c*” (assuming the Module Implementation declaration has set the *Language* property to “C”).

ECOA Deployment Definition

The ECOA “*PubSub*” Assembly is deployed (that is, the declared Module and Trigger Instances are allocated to ECOA Protection Domains, which are themselves allocated to computing nodes) by the following XML (file *deployment.xml*):

```
<deployment xmlns="http://www.ecoa.technology/deployment-1.0"
     finalAssembly="PubSub_impl" logicalSystem="hostbased_logical_system">
     <protectionDomain name="pub">
          <executeOn computingNode="cpu1" computingPlatform="host"/>
          <deployedModuleInstance componentName="Pub"
                                   moduleInstanceName="Pub_Main_Instance"
                                   modulePriority="50"/>
          <deployedTriggerInstance componentName="Pub"
                                   triggerInstanceName="Ticker"
                                   triggerPriority="51"/>
     </protectionDomain>
```

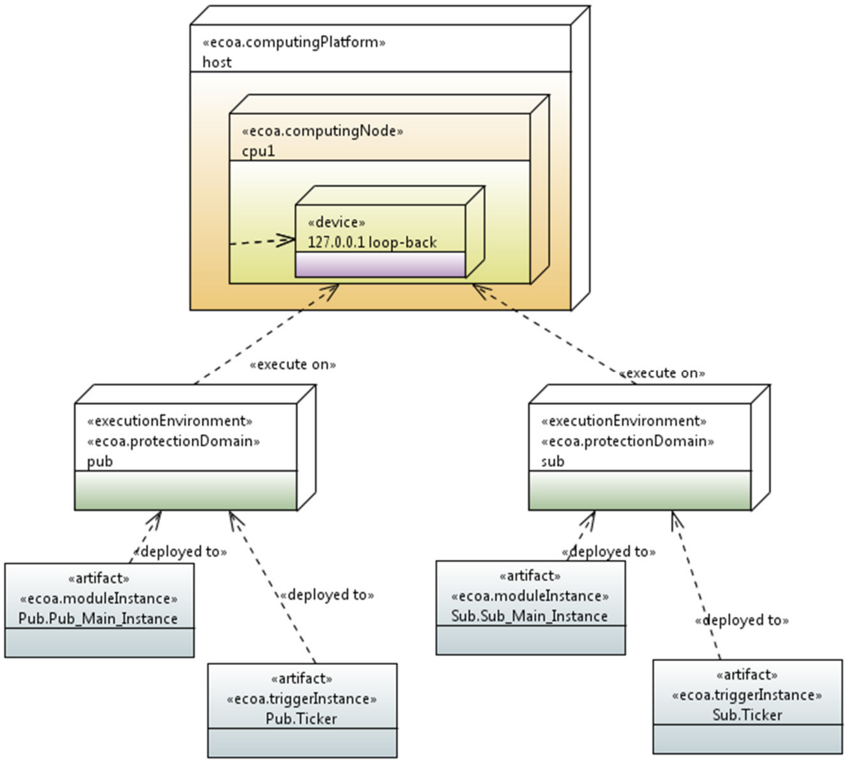
---

[4] The UML does not explicitly depict Service Links. The period attribute is therefore depicted as a UML property of the *«ecoa.triggerInstance»* UML interface class.

```
     <protectionDomain name="sub">
          <executeOn computingNode="cpu2" computingPlatform="host"/>
          <deployedModuleInstance componentName="Sub"
                                    moduleInstanceName="Sub_Main_Instance"
                                    modulePriority="50"/>
          <deployedTriggerInstance componentName="Sub"
                                    triggerInstanceName="Ticker"
                                    triggerPriority="51"/>
 </protectionDomain>
     <platformConfiguration notificationMaxNumber="8"
                                    computingPlatform="host" />
</deployment>
```

Thus two ECOA Protection Domains are declared (*pub* and *sub*) executing on ECOA Computing Node *cpu1*, in ECOA Computing Platform *host*. Though conceived as able to run in separate Computing Nodes on separate Computing Platforms, the more normal case represented here is for the two Protection Domains to be running on the same host (e.g. a Windows® or Linux desk or laptop PC, or a demonstration VxWorks SBC). This single-host deployment is represented as a UML Deployment Diagram in Figure 10:

Figure 10 ECOA "*PubSub*" Assembly Deployment



The (UDP) Transport Binding (file *udpbinding.xml*) for this single Computing Node, single Computing Platform, deployment therefore invokes the local loop-back address "127.0.0.1", which means that communicated data passes directly between the ECOA Protection Domains:

```
<ecoa:UDPBinding xmlns:ecoa="http://www.ecoa.technology/udpbinding-1.0">
    <ecoa:platform name="host" receivingMulticastAddress="127.0.0.1"
                               receivingPort="60421"
                               platformId="1"/>
</ecoa:UDPBinding>
```

ECOA Lifecycle and Service Availability Considerations

Since we have introduced a Trigger Instance into both the `Pub` and `Sub` ASC, it is necessary that the respective Supervision Modules (see ref.[1]) (i.e. the Module Implementations `Pub_Main_Im` and `Sub_Main_Im`) manage their lifecycle.

The management is generic and comprises:

- invoking an **INITIALIZE** ECOA Event Operation to the Trigger Instance to bring its Module Lifecycle state from **IDLE** to **READY**;
- invoking a **START** ECOA Event Operation to the Trigger Instance once its Module Lifecycle state is **READY.**

No other actions (such as invoking **STOP** or **SHUTDOWN**) are required in the present simple example.

Also, since the `Pub` ASC *provides* ECOA Services (`TopicX` and `TopicY`) it is necessary that the Services be declared (at runtime) as "available" (or not, as the case may be). For the present simple example, this will be done when the `Pub` ASC's Supervision Module (namely the Module Implementation `Pub_Main_Im`) receives a **START** Event Operation. No error conditions are defined, so once set, the `TopicX` and `TopicY` Services will always be "available".

Experimental File Access API Design

As part of the "*PubSub*" publish-subscribe example it was decided to experiment with File Access within an ECOA context. Two possible methods of providing file I/O were identified:

- a Service and Provider-ASC mechanism; or
- an ECOA Platform provided API[5] mechanism.

The Service and Provider-ASC mechanism is explored elsewhere (ref.[3]). The "*PubSub*" publish-subscribe example explores only the File Access API mechanism.

The experimental File Access API defines a minimal set of six operations:

i. **`open(…)`** takes a `FileName` and attempts to open the named file according to a set of `FileFlags` and `Permissions`, and returns a `FileReference` handle.
ii. **`close(…)`** closes the file referenced by a `FileReference` handle.

---

[5] Application Programming Interface

iii. ***read(…)*** (attempts to) read *byteCount* bytes from the file referenced by a *FileReference* handle, and stores them in a *buffer*. Returns the actual number of *bytesRead*.

iv. ***write(…)*** (attempts to) write *byteCount* bytes taken from a *buffer* to the file referenced by a *FileReference* handle. Returns the actual number of *bytesWritten* .

v. ***seek(…)*** (attempts to) move the current read or write point (within the file referenced by a *FileReference* handle) to the position specified by a *SeekBase* reference and an offset value (relative to the *SeekBase*).

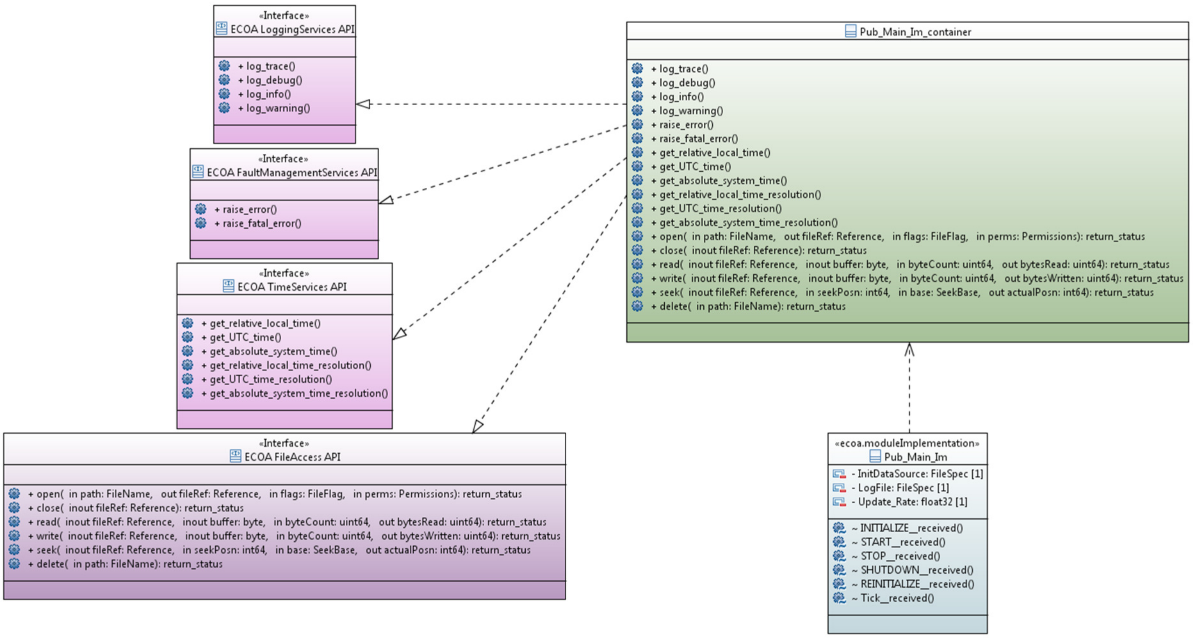
vi. ***delete(…)*** deletes the file named by a *FileName* parameter.

These operations would be made available to ECOA Module source code as ECOA Container Operations, prefixed with the name of the using Module Implementation, in the same manner as the Service APIs defined in ref.[1] (Part 4). The rationale behind this mechanism, rather than simply using the underlying Operating System's file I/O operations, is that the using Module Implementation becomes fully portable, and isn't dependent on a specific (for instance) *open()* API operation such as that defined by the POSIX standard. The open operation will ALWAYS be provided by the Module code calling (for instance) the *#module_implementation_-name#_container__open()* API operation.

In the specific case here, the source code that will be using File Access API is that of the *Pub* ASC's *Pub_Main_Im* Module Implementation, so the experiment calls for six operations named *Pub_Main_Im_container__open()* to *Pub_Main_Im_container__delete()*.

The experimental File Access API is therefore an extension to the *Pub_Main_Im* container, illustrated in UML in Figure 11. This depicts the Module Container as a (concrete) UML class (*Pub_Main_Im_container*), *implementing* (dashed line with a closed arrowhead) the ECOA standard service API *interfaces* (*FaultManagement*, *Logging*, and *Time* are depicted), EXTENDED by the experimental *FileAccess* service API interface, and *used by* (dashed line with an open arrowhead) the Module Implementation class (*Pub_Main_Im*). Omitted for clarity from the diagram are some additional service API interfaces (see ref.[1] Part 4) that complete the Module-Container Interface, as well as the parameters of the standard ECOA service API operations.

Figure 11 Experimental File Access API as an Extension to the Module Container


«Interface»
ECOA LoggingServices API
+ log_trace()
+ log_debug()
+ log_info()
+ log_warning()
«Interface»
ECOA FaultManagementServices API
+ raise_error()
+ raise_fatal_error()
«Interface»
ECOA TimeServices API
+ get_relative_local_time()
+ get_UTC_time()
+ get_absolute_system_time()
+ get_relative_local_time_resolution()
+ get_UTC_time_resolution()
+ get_absolute_system_time_resolution()
«Interface»
ECOA FileAccess API
+ open(in path: FileName, out fileRef: Reference, in flags: FileFlag, in perms: Permissions): return_status
+ close(inout fileRef: Reference): return_status
+ read(inout fileRef: Reference, inout buffer: byte, in byteCount: uint64, out bytesRead: uint64): return_status
+ write(inout fileRef: Reference, inout buffer: byte, in byteCount: uint64, out bytesWritten: uint64): return_status
+ seek(inout fileRef: Reference, in seekPosn: int64, in base: SeekBase, out actualPosn: int64): return_status
+ delete(in path: FileName): return_status
Pub_Main_Im_container
+ log_trace()
+ log_debug()
+ log_info()
+ log_warning()
+ raise_error()
+ raise_fatal_error()
+ get_relative_local_time()
+ get_UTC_time()
+ get_absolute_system_time()
+ get_relative_local_time_resolution()
+ get_UTC_time_resolution()
+ get_absolute_system_time_resolution()
+ open(in path: FileName, out fileRef: Reference, in flags: FileFlag, in perms: Permissions): return_status
+ close(inout fileRef: Reference): return_status
+ read(inout fileRef: Reference, inout buffer: byte, in byteCount: uint64, out bytesRead: uint64): return_status
+ write(inout fileRef: Reference, inout buffer: byte, in byteCount: uint64, out bytesWritten: uint64): return_status
+ seek(inout fileRef: Reference, in seekPosn: int64, in base: SeekBase, out actualPosn: int64): return_status
+ delete(in path: FileName): return_status
«ecoa.moduleImplementation»
Pub_Main_Im
- InitDataSource: FileSpec [1]
- LogFile: FileSpec [1]
- Update_Rate: float32 [1]
~ INITIALIZE__received()
~ START__received()
~ STOP__received()
~ SHUTDOWN__received()
~ REINITIALIZE__received()
~ Tick__received()

This experimental File Access API requires the data types previously defined in the *`ECOA.File`* Types Library, and illustrated in Figure 7, for its operation parameters. Each operation of the experimental File Access API returns an ECOA *`return_status`* code.

Implementation

The Sub ASC

On invocation of the *`START`* operation, the *`Sub`* ASC will initialize its *`Ticker`* Trigger Instance, as required by the ECOA (ref.[1]). That *`START`* operation is implemented by the code function *`Sub_Main_Im__START__received`*:

```
void Sub_Main_Im__START__received(Sub_Main_Im__context *context)
{
        Sub_Main_Im_container__INITIALIZE__Ticker( context );
}
```

When a Module or Trigger Instance has changed Module Lifecycle state, the ECOA Software Platform issues a Lifecycle Notification to its parent Supervision Module. In the present case, the notification for the *`Sub`* ASC's *`Ticker`* Trigger Instance is implemented by the *`Sub_Main_Im__lifecycle_notification__Ticker`* code function:

```
void Sub_Main_Im__lifecycle_notification__Ticker(
     Sub_Main_Im__context* context,
     ECOA__module_states_type previous_state,
     ECOA__module_states_type new_state)
{
     if(( previous_state == ECOA__module_states_type_IDLE )&&
          ( new_state == ECOA__module_states_type_READY ))
          Sub_Main_Im_container__START__Ticker( context );
}
```

That is, if the *`Ticker`* Trigger Instance has transitioned from the **IDLE** to the **READY** state (i.e. it has been initialized), then it is started. Other Module Lifecycle state transitions, such as transitioning from the **READY** to the **RUNNING** state (as a result of a **START** operation) are ignored in this example.

There being no other Module Lifecycle operation we need to handle (**STOP**, **RESTART** etc.), all we need to do now then is to program the subscriber behaviour, which is to periodically read the *`TopicX`* data, and is therefore implemented in the periodically invoked code function *`Sub_Main_Im__Tick__received`*:

```
void Sub_Main_Im__Tick__received(Sub_Main_Im__context *context)
{
    Sub_Main_Im_container__someData_handle data_hndl;
    Sub_Main_Im_container__notes_handle notes_hndl;
    Sub_Main_Im_container__log_handle log_hndl;
    data__LogItem itm;
    //
    Sub_Main_Im_container__someData__get_read_access( context, &data_hndl );
    Sub_Main_Im_container__someData__release_read_access( context, &data_hndl );
    //
    Sub_Main_Im_container__notes__get_read_access( context, &notes_hndl );
    printf( "Accessed \"%s\"\n", *(notes_hndl.data) );
    Sub_Main_Im_container__notes__release_read_access( context, &notes_hndl );
    //
    Sub_Main_Im_container__log__get_read_access( context, &log_hndl );
    itm = *(log_hndl.data);
    Sub_Main_Im_container__log__release_read_access( context, &log_hndl );
}
```

Each period, each of the three *`TopicX`* data items is accessed. In the case of the *`notes`* item, the value is printed to the console. In the case of the *`Log`* item, the current value is simply copied (to the local variable *`itm`*). In the presented implementation, the *`data`* item is read, but ignored.

The Pub ASC

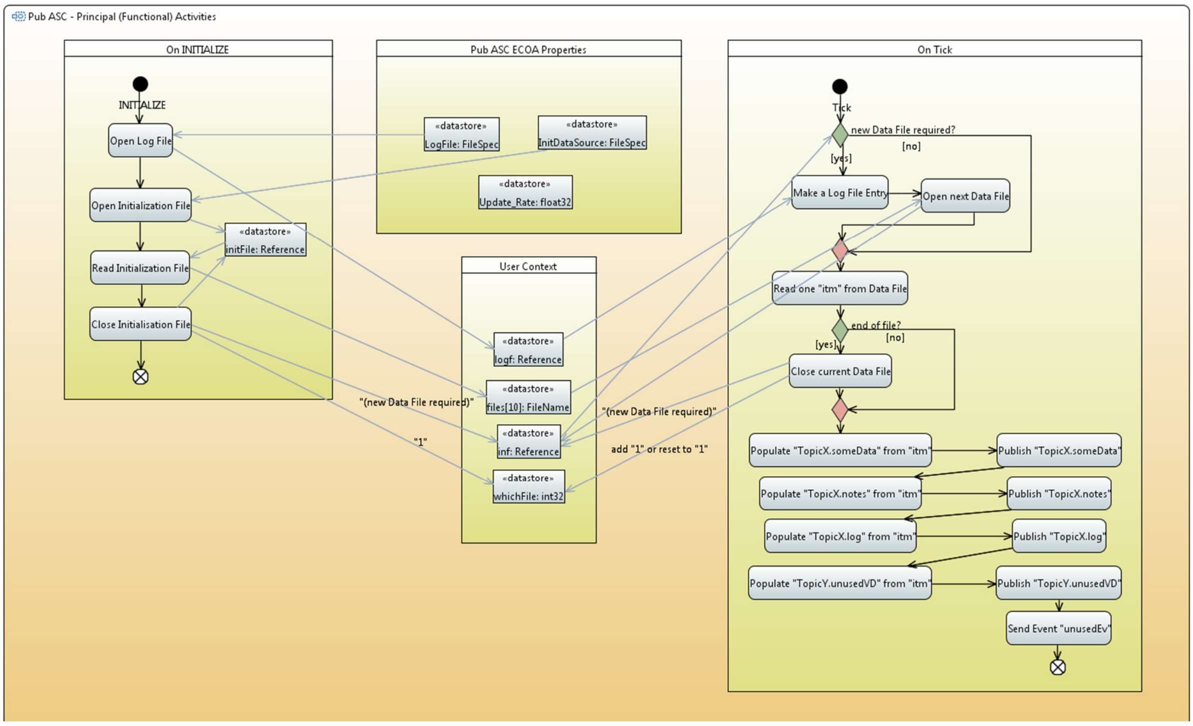
The *`Pub`* ASC provides data to the *`TopicX`* Service simply by reading it from a file – in fact one of number of files opened in sequence as the end of each is read. The names of the data files, and the order in which they are opened, are themselves read from the contents of an *Initialization File*. A *Log File* is opened and progress messages are written to it. The name of the *Initialization File* and the *Log File* are given by the ASC's ECOA Component Properties (*`InitDataSource`* and *`LogFile`* respectively). In this presented implementation of the `Pub` ASC, the `Update_Rate` ECOA Module Property is not used, only declared in the XML Module definition.

Following initialization, each time the ASC's `Ticker` trigger fires, invoking the `Tick` operation, the ASC reads an item of data ("`itm`") from the current data file, populates the `TopicX` and `TopicY` data items (i.e. the Versioned Data items of the `TopicX` and `TopicY` Services) from that data item, and publishes the updated topics (Versioned Data items).

Each time the end of a data file is reached, the file is closed. The next data file is opened on the next `Tick`, and a message is written to the *Log File* to that effect.

These principal functional activities of the Pub ASC are depicted in the UML Activity diagram Figure 12.

Figure 12 *Pub* ASC Principal Functional Activities



In order to retain the list of data files, as well as the `ECOA__File__Reference` handle[6] for the current data file and the *Log File*, the `Pub_Main_Im` Module Implementation defines a user context structure in the source code (header) file `Pub_Main_Im_user_context.h`:

---

[6] In a C code implementation, the design data type name `ECOA.File.Reference` becomes the implementation data type name `ECOA__File__Reference`.

```
typedef struct
{

      ECOA__File__Reference inf, logf;
      ECOA__int32           whichFile;
      ECOA__File__FileName  files[10];

} Pub_Main_Im_user_context;
```

This specifies that a list of up to 10 data file names can be stored (“*files[10]*”). The current data file, once open, is referenced by the “*inf*” *ECOA__File__Reference* handle. The *Log File* is referenced by the *ECOA__File__Reference* “*logf*”.

The *INITIALIZE* Lifecycle operation (of the *Pub* ASC – i.e. Module Implementation *Pub_Main_Im*) is implemented by the code function *Pub_Main_Im__INITIALIZE__received* of the (C) code unit *Pub_Main_Im.c*, which opens the *Log File*, reads the *Initialization File*, and initializes the user context variables as discussed above:

```
void Pub_Main_Im__INITIALIZE__received(Pub_Main_Im__context *context)
{
      ECOA__File__FileSpec  initFileSpec, logFileSpec;
      ECOA__File__Reference initFile;
      ECOA__log    msg;
      ECOA__int32  indx = 0;
      ECOA__char8  bigbuf[32768], tmpbuf[1024], *p;
      ECOA__uint64 bc;
      //
      // Open the LogFile
      Pub_Main_Im_container__get_LogFile_value( context, &logFileSpec );
      if( Pub_Main_Im_container__open( context, logFileSpec.FileName,
                                 &(context->user.logf), logFileSpec.Access,
                                 logFileSpec.CreatePermissions ) !=
                                 ECOA__return_status_OK ){
            msg.current_size = sprintf( msg.data, "%s: %s",
                                 logFileSpec.FileName, "Open failed.");
            Pub_Main_Im_container__raise_error( context, msg );
      }
      // Open the initialisation data source
      Pub_Main_Im_container__get_InitDataSource_value( context, &initFileSpec
                                 );
      if( Pub_Main_Im_container__open( context, initFileSpec.FileName,
                                 &initFile, initFileSpec.Access,
                                 initFileSpec.CreatePermissions ) !=
                                 ECOA__return_status_OK ){
            msg.current_size = sprintf( msg.data, "%s: %s",
                                 initFileSpec.FileName, "Open failed.");
            Pub_Main_Im_container__raise_error( context, msg );
      }
```

```
        // Read the initialisation data
        if( Pub_Main_Im_container__read( context, &initFile, &bigbuf, 32767,
                                 &bc ) != ECOA__return_status_OK ){
             msg.current_size = sprintf( msg.data, "%s: %s",
                                 initFileSpec.FileName, "Read failed.");
             Pub_Main_Im_container__raise_error( context, msg );
        }
        // Parse the initialisation data
        p = strtok( bigbuf, "\n" );
        for(;;){
             p = strtok( NULL, "\n" );
             if( !p || sscanf( p, "%d, %s", &indx, tmpbuf ) < 2 )
                  break;
             else
                  strcpy( context->user.files[indx], tmpbuf );
        }
        Pub_Main_Im_container__close( context, &initFile );
        //
        // Other initialisations
        context->user.inf.descriptor = -1;
        context->user.whichFile = 1;
   }
```

On invocation of the *START* operation, the *Pub* ASC will initialize its *Ticker* Trigger Instance, as required by the ECOA (ref.[1]), and set the *TopicX* and *TopicY* Services as being “available”. That *START* operation is implemented by the code function *Pub_Main_Im__START__received*:

```
   void Pub_Main_Im__START__received(Pub_Main_Im__context *context)
   {
        Pub_Main_Im_container__INITIALIZE__Ticker( context );
        Pub_Main_Im_container__set_service_availability( context,
                                 Pub_Main_Im_container__service_id__TopicX,
                                 ECOA__TRUE );
        Pub_Main_Im_container__set_service_availability( context,
                                 Pub_Main_Im_container__service_id__TopicY,
                                 ECOA__TRUE );
   }
```

When a Module or Trigger Instance has changed Module Lifecycle state, the ECOA Software Platform issues a Lifecycle Notification to its parent Supervision Module. In the present case, the notification for the *Pub* ASC’s *Ticker* Trigger Instance is implemented by the *Pub_Main_Im__lifecycle_notification__Ticker* code function:

```
void Pub_Main_Im__lifecycle_notification__Ticker
   (Pub_Main_Im__context* context,
    ECOA__module_states_type previous_state,
    ECOA__module_states_type new_state)
{
     if(( previous_state == ECOA__module_states_type_IDLE )&&( new_state ==
                                ECOA__module_states_type_READY ))
          Pub_Main_Im_container__START__Ticker( context );
}
```

That is, if the *`Ticker`* Trigger Instance has transitioned from the **IDLE** to the **READY** state (i.e. it has been initialized), then it is started. Other Module Lifecycle state transitions, such as transitioning from the **READY** to the **RUNNING** state (as a result of a **START** operation) are ignored in this example.

The "unusedRR" Service Operation is handled by the code function *`Pub_modMain_Im__unusedRR__request_received`*. Whilst in the "*PubSub*" publish-subscribe example this request-response operation is unused, it is necessary to send a response, should the *`Pub`* ASC ever be reused in an application where the *`TopicY`* Service is invoked. The required return value (of type *`ECOA__int32`*) is, for the current purposes, simply a numerical copy of the request-response sequence number (*`ID`*).

```
void Pub_Main_Im__unusedRR__request_received
   (Pub_Main_Im__context* context,
    const ECOA__uint32 ID)
{
     ECOA__int32 foo = ID;
     Pub_Main_Im_container__unusedRR__response_send( context, ID, foo );
}
```

All we need do now is to program what to do when the Ticker Trigger fires, i.e. to populate the *`Pub_modMain_Im__Tick__received`* function stub (see over page) according to the required activities of Figure 12.

That is, the data file reference (*`inf`*[7]) is checked to see if a new data file needs to be opened (*`inf.descriptor`* has the "magic" value "-1"). If so, a message is logged to the ECOA software platform logging mechanism (using the *`Log_info`* API) and is written to the *`Pub`* ASC's own log file (referenced by file reference *`Logf`*). The data file whose name is in the list at position *`files[whichFile]`* is the opened, and *`inf`* is changed to reference this new file.

The next (or first if the file has just been opened) data item (of type *`data__LogItem`*, i.e. type *`LogItem`* defined in Types Library *`data`*) is read from the data file. For portability of the data file, numerical values read from the file are checked, and if necessary corrected, for byte and word endianness.

---

[7] Since the *`inf`* file reference is held in the Module Implementation's user context, it is referenced in the C code using "*`context->user.inf`*".

This document is developed for and on behalf of BAE Systems (Operations) Limited, Dassault Aviation, Bull SAS, Thales Systèmes Aéroportés, AgustaWestland Limited, GE Aviation Systems Limited, General Dynamics United Kingdom Limited and Selex ES Ltd, and the copyright is owned by BAE Systems (Operations) Limited, Dassault Aviation, Bull SAS, Thales Systèmes Aéroportés, AgustaWestland Limited, GE Aviation Systems Limited, General Dynamics United Kingdom Limited and Selex ES Ltd. This document is developed by BAE Systems (Operations) Limited, Electronic Systems, and is the Intellectual Property of BAE Systems (Operations) Limited, Electronic Systems. The information set out in this document is provided solely on an 'as is' basis and the co-developers of this software make no warranties expressed or implied, including no warranties as to completeness, accuracy or fitness for purpose, with respect to any of the information.
24

The four ECOA Versioned Data items are then populated using the read data item.

Finally the `unusedEv` ECOA Event operation is triggered, which, of course, in the present example has no receiver.

```c
void Pub_Main_Im__Tick__received(Pub_Main_Im__context *context)
{
    Pub_Main_Im_container__someData_handle data_hndl;
    Pub_Main_Im_container__notes_handle notes_hndl;
    Pub_Main_Im_container__log_handle log_hndl;
    Pub_Main_Im_container__unusedVD_handle unusedVD_hndl;
    ECOA__log      msg;
    data__LogItem itm;
    ECOA__uint64  bc;
    //
    if( context->user.inf.descriptor == -1 ){
        msg.current_size = sprintf( msg.data, "Opening data file '%s'", context->user.files[context->user.whichFile]);
        Pub_Main_Im_container__log_info( context, msg );
        Pub_Main_Im_container__write( context, &(context->user.logf), &msg.data, msg.current_size, &bc );
        if( Pub_Main_Im_container__open( context, context->user.files[context->user.whichFile], &(context->user.inf),
                ECOA__File__FileFlag_READ|ECOA__File__FileFlag_BINARY,
                (ECOA__File__Permissions){ ECOA__File__Permission_ALL, ECOA__File__Permission_READEXEC, ECOA__File__Permission_READ }
            ) != ECOA__return_status_OK ){
            msg.current_size = sprintf( msg.data, "%s: %s", context->user.files[context->user.whichFile], "Open failed.");
            Pub_Main_Im_container__raise_error( context, msg );
        }
    }
    //
    if( Pub_Main_Im_container__read( context, &(context->user.inf), (ECOA__byte*)&itm, sizeof(itm), &bc ) != ECOA__return_status_OK ){
        if( context->user.files[context->user.whichFile+1][0] == '\000' )
            context->user.whichFile = 1;
        else
            context->user.whichFile += 1;
        Pub_Main_Im_container__close( context, &(context->user.inf) );
        context->user.inf.descriptor = -1;
        return;
    }
    itm.seq = ntohll( itm.seq );
    itm.data.current_size = ntohl( itm.data.current_size );
    //
    Pub_Main_Im_container__someData__get_write_access( context, &data_hndl );
    memcpy( (data_hndl.data), &itm.data, sizeof(data__BLOB));
    Pub_Main_Im_container__someData__publish_write_access( context, &data_hndl );
    //
    Pub_Main_Im_container__notes__get_write_access( context, &notes_hndl );
    sprintf( *(notes_hndl.data), "Entry %lld", itm.seq );
    Pub_Main_Im_container__notes__publish_write_access( context, &notes_hndl );
    //
    Pub_Main_Im_container__log__get_write_access( context, &log_hndl );
    memcpy( (log_hndl.data), &itm, sizeof(data__LogItem));
    Pub_Main_Im_container__log__publish_write_access( context, &log_hndl );
    //
    Pub_Main_Im_container__unusedVD__get_write_access( context, &unusedVD_hndl );
    sprintf( *(unusedVD_hndl.data), "Entry %lld", itm.seq );
    Pub_Main_Im_container__unusedVD__publish_write_access( context, &unusedVD_hndl );
    //
    Pub_Main_Im_container__unusedEv__send( context );
}
```

The Experimental ECOA File Access API

The implementation of the experimental File Access API will not be described in full. Think of it as an exercise for the reader. Suffice it to say that for the experiment, the various operations simply map to an equivalent POSIX operation API.

```
ECOA__return_status Pub_Main_Im_container__open(
                                  Pub_Main_Im__context    *context,
                                  ECOA__File__FileName     path,
                                  ECOA__File__Reference   *fileRef,
                                  ECOA__File__FileFlag     flags,
                                  ECOA__File__Permissions perms );
```

The experimental implementation maps the *flags* and *perms* onto POSIX values, and calls the POSIX *open()* API.

```
ECOA__return_status Pub_Main_Im_container__close(
                                  Pub_Main_Im__context   *context,
                                  ECOA__File__Reference *fileRef );
```

The experimental implementation calls the POSIX *close()* API on the file referenced by *fileRef*.

```
ECOA__return_status Pub_Main_Im_container__read(
                                  Pub_Main_Im__context   *context,
                                  ECOA__File__Reference *fileRef,
                                  ECOA__byte            *buffer,
                                  ECOA__uint64           byteCount,
                                  ECOA__uint64          *bytesRead );
```

The experimental implementation calls the POSIX *read()* API on the file referenced by *fileRef* to read (up to) *byteCount* bytes into *buffer*.

```
ECOA__return_status Pub_Main_Im_container__write(
                                  Pub_Main_Im__context   *context,
                                  ECOA__File__Reference *fileRef,
                                  ECOA__byte            *buffer,
                                  ECOA__uint64           byteCount,
                                  ECOA__uint64          *bytesWritten );
```

The experimental implementation calls the POSIX *write()* API on the file referenced by *fileRef* to write (up to) *byteCount* bytes from *buffer*.

```
ECOA__return_status Pub_Main_Im_container__seek(
                                  Pub_Main_Im__context   *context,
                                  ECOA__File__Reference *fileRef,
                                  ECOA__int64            seekPosn,
                                  ECOA__File__SeekBase   base,
                                  ECOA__int64           *actualPosn );
```

The experimental implementation calls the POSIX *lseek()* API on the file referenced by *fileRef* to position the current read/write point, mapping the *base* and *seekPosn* values to the POSIX equivalent.

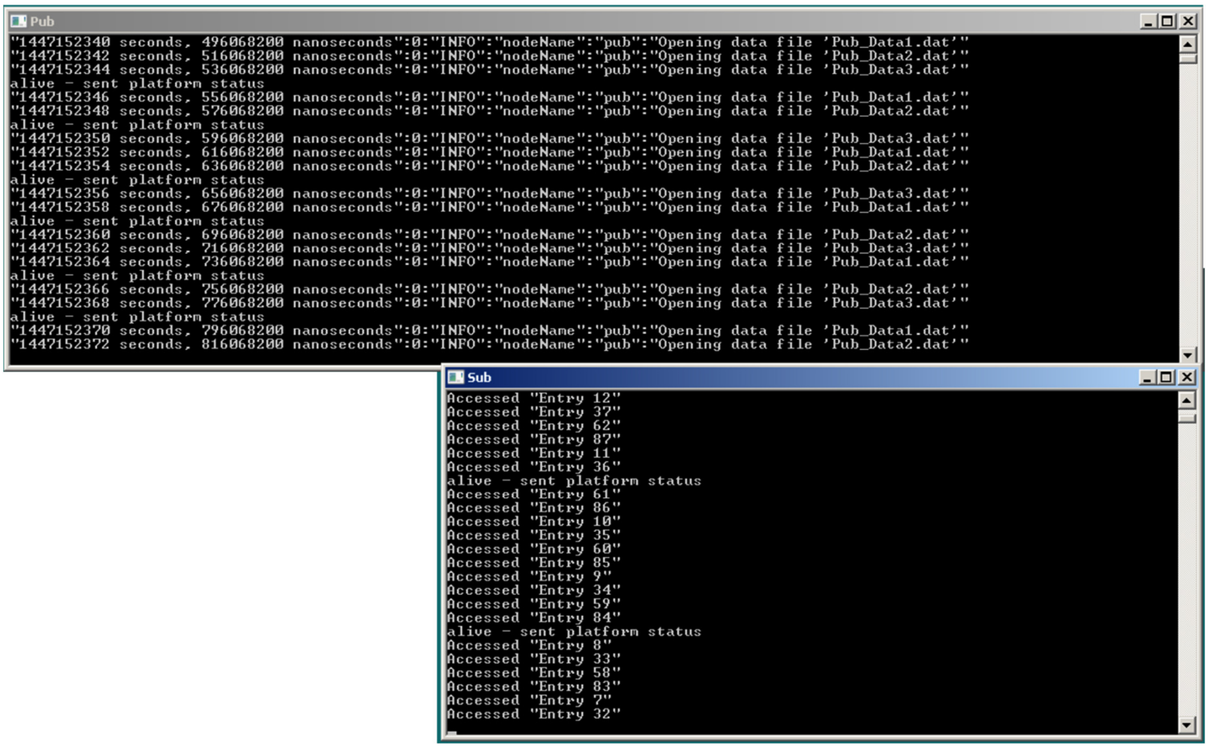
```
    ECOA__return_status Pub_Main_Im_container__delete(
                                    Pub_Main_Im__context *context,
                                    ECOA__File__FileName  path );
```

`Deletes the file named by` *`path`* `using the POSIX` *`unlink()`* `API.`

Program Output

When the ECOA "*PubSub*" Assembly is built and run, an output similar to Figure 13 should be achieved. This shows two Windows® Command Prompt panes, one running the "*Pub*" Protection Domain, and one running the "*Sub*" protection Domain. The *Pub* and *Sub* ASC trace messages are output to their respective console panes, in the case of the *Pub* ASC prefixed by miscellaneous logging data (time stamp, logging type, etc.) because the ECOA logging API is used. Each *Pub* ASC trace message reports a switch from one data file to another. Each *Sub* ASC message reports the ASC accessing a particular Versioned Data record, and shows how only some updates of the Versioned Data item are read by the *Sub* (since the *Pub* ASC publishes a continuous sequence of numbered "Entry *N*" item values). These outputs are interleaved with ECOA Platform logging messages (the 10 second periodic "alive" messages in the example shown):

Figure 13 ECOA "*PubSub*" Assembly in Execution



```
"1447152340 seconds, 496068200 nanoseconds":0:"INFO":"nodeName":"pub":"Opening data file 'Pub_Data1.dat'"
"1447152342 seconds, 516068200 nanoseconds":0:"INFO":"nodeName":"pub":"Opening data file 'Pub_Data2.dat'"
"1447152344 seconds, 536068200 nanoseconds":0:"INFO":"nodeName":"pub":"Opening data file 'Pub_Data3.dat'"
alive - sent platform status
"1447152346 seconds, 556068200 nanoseconds":0:"INFO":"nodeName":"pub":"Opening data file 'Pub_Data1.dat'"
"1447152348 seconds, 576068200 nanoseconds":0:"INFO":"nodeName":"pub":"Opening data file 'Pub_Data2.dat'"
alive - sent platform status
"1447152350 seconds, 596068200 nanoseconds":0:"INFO":"nodeName":"pub":"Opening data file 'Pub_Data3.dat'"
"1447152352 seconds, 616068200 nanoseconds":0:"INFO":"nodeName":"pub":"Opening data file 'Pub_Data1.dat'"
"1447152354 seconds, 636068200 nanoseconds":0:"INFO":"nodeName":"pub":"Opening data file 'Pub_Data2.dat'"
alive - sent platform status
"1447152356 seconds, 656068200 nanoseconds":0:"INFO":"nodeName":"pub":"Opening data file 'Pub_Data3.dat'"
"1447152358 seconds, 676068200 nanoseconds":0:"INFO":"nodeName":"pub":"Opening data file 'Pub_Data1.dat'"
alive - sent platform status
"1447152360 seconds, 696068200 nanoseconds":0:"INFO":"nodeName":"pub":"Opening data file 'Pub_Data2.dat'"
"1447152362 seconds, 716068200 nanoseconds":0:"INFO":"nodeName":"pub":"Opening data file 'Pub_Data3.dat'"
"1447152364 seconds, 736068200 nanoseconds":0:"INFO":"nodeName":"pub":"Opening data file 'Pub_Data1.dat'"
alive - sent platform status
"1447152366 seconds, 756068200 nanoseconds":0:"INFO":"nodeName":"pub":"Opening data file 'Pub_Data2.dat'"
"1447152368 seconds, 776068200 nanoseconds":0:"INFO":"nodeName":"pub":"Opening data file 'Pub_Data3.dat'"
alive - sent platform status
"1447152370 seconds, 796068200 nanoseconds":0:"INFO":"nodeName":"pub":"Opening data file 'Pub_Data1.dat'"
"1447152372 seconds, 816068200 nanoseconds":0:"INFO":"nodeName":"pub":"Opening data file 'Pub_Data2.dat'"
```

```
Accessed "Entry 12"
Accessed "Entry 37"
Accessed "Entry 62"
Accessed "Entry 87"
Accessed "Entry 11"
Accessed "Entry 36"
alive - sent platform status
Accessed "Entry 61"
Accessed "Entry 86"
Accessed "Entry 10"
Accessed "Entry 35"
Accessed "Entry 60"
Accessed "Entry 85"
Accessed "Entry 9"
Accessed "Entry 34"
Accessed "Entry 59"
Accessed "Entry 84"
alive - sent platform status
Accessed "Entry 8"
Accessed "Entry 33"
Accessed "Entry 58"
Accessed "Entry 83"
Accessed "Entry 7"
Accessed "Entry 32"
```

References

1	European Component Oriented Architecture (ECOA) Collaboration Programme: Architecture Specification (Parts 1 to 11) “ECOA” is a registered trade mark.
2	Publish-subscribe pattern *https://en.wikipedia.org/wiki/Publish/subscribe*
3	European Component Oriented Architecture (ECOA) Collaboration Programme: Guidance Document: Data Servers