WInnForum Facilities PSMs Mapping Rules

Document WINNF-TR-2008

Version V1.0.1

18 January 2022
TERMS, CONDITIONS & NOTICES

This document has been prepared by the Software Defined Systems (SDS) Harmonized Timing Service Task Group to assist The Software Defined Radio Forum Inc. (or its successors or assigns, hereafter “the Forum”). It may be amended or withdrawn at a later time and it is not binding on any member of the Forum or of the Harmonized Timing Service Task Group.

Contributors to this document that have submitted copyrighted materials (the Submission) to the Forum for use in this document retain copyright ownership of their original work, while at the same time granting the Forum a non-exclusive, irrevocable, worldwide, perpetual, royalty-free license under the Submitter’s copyrights in the Submission to reproduce, distribute, publish, display, perform, and create derivative works of the Submission based on that original work for the purpose of developing this document under the Forum's own copyright.

Permission is granted to the Forum’s participants to copy any portion of this document for legitimate purposes of the Forum. Copying for monetary gain or for other non-Forum related purposes is prohibited.

THIS DOCUMENT IS BEING OFFERED WITHOUT ANY WARRANTY WHATSOEVER, AND IN PARTICULAR, ANY WARRANTY OF NON-INFRINGEMENT IS EXPRESSLY DISCLAIMED. ANY USE OF THIS SPECIFICATION SHALL BE MADE ENTIRELY AT THE IMPLEMENTER'S OWN RISK, AND NEITHER THE FORUM, NOR ANY OF ITS MEMBERS OR SUBMITTERS, SHALL HAVE ANY LIABILITY WHATSOEVER TO ANY IMPLEMENTER OR THIRD PARTY FOR ANY DAMAGES OF ANY NATURE WHATSOEVER, DIRECTLY OR INDIRECTLY, ARISING FROM THE USE OF THIS DOCUMENT.

Recipients of this document are requested to submit, with their comments, notification of any relevant patent claims or other intellectual property rights of which they may be aware that might be infringed by any implementation of the specification set forth in this document, and to provide supporting documentation.

This document was developed following the Forum's policy on restricted or controlled information (Policy 009) to ensure that that the document can be shared openly with other member organizations around the world. Additional Information on this policy can be found here: http://www.wirelessinnovation.org/page/Policies_and_Procedures

Although this document contains no restricted or controlled information, the specific implementation of concepts contain herein may be controlled under the laws of the country of origin for that implementation. Readers are encouraged, therefore, to consult with a cognizant authority prior to any further development.

Wireless Innovation Forum™ and SDR Forum™ are trademarks of the Software Defined Radio Forum Inc.
# Table of Contents

TERM, CONDITIONS & NOTICES........................................................................................................ i

Table of Contents ................................................................................................................................. ii

List of Figures ........................................................................................................................................ v

Contributors .......................................................................................................................................... vi

WINnForum Facilities PSMs Mapping Rules ...................................................................................... 1

1 Introduction ......................................................................................................................................... 1

1.1 Document purpose ........................................................................................................................ 1

1.2 Reference architectural pattern .................................................................................................. 1

1.3 Reference definitions ................................................................................................................... 2

1.4 Conventions .................................................................................................................................. 2

1.4.1 Case conventions .................................................................................................................... 2

1.4.2 Functional support capability identification ......................................................................... 2

1.5 Conformance .................................................................................................................................. 3

1.5.1 Radio platform items ............................................................................................................ 3

1.5.2 Radio application items ......................................................................................................... 3

2 Native C++ ...................................................................................................................................... 4

2.1 General assumptions ..................................................................................................................... 4

2.1.1 Specified interfaces .............................................................................................................. 4

2.1.2 Architecture assumptions .................................................................................................... 4

2.2 Conformance .................................................................................................................................. 5

2.2.1 Radio platform items ............................................................................................................ 5

2.2.2 Radio application items ......................................................................................................... 5

2.3 PIM API mapping .......................................................................................................................... 5

2.3.1 Root namespace ...................................................................................................................... 5

2.3.2 Modules .................................................................................................................................. 5

2.3.3 Types ....................................................................................................................................... 6

2.3.4 Interface declaration properties .......................................................................................... 6

2.3.5 API types ............................................................................................................................... 7

2.3.6 Exceptions ................................................................................................................................ 7

2.3.7 Services interfaces ............................................................................................................... 7

2.3.8 Primitives .................................................................................................................................. 7

2.4 Specialization of PIM unspecified concepts ............................................................................... 7

2.4.1 Access capabilities ............................................................................................................... 7

2.4.2 Entry in CONFIGURED state ............................................................................................... 8

2.5 Referenced PIM version ............................................................................................................... 8

2.6 Facade class ................................................................................................................................... 8

2.6.1 activeServicesInitialized method .................................................................................... 9

2.6.2 activeServicesReleased method ........................................................................................ 10

2.6.3 Services access principles ................................................................................................... 11

2.6.4 Explicit services access ....................................................................................................... 12

2.6.5 Generic services access ....................................................................................................... 15

2.7 Header files ................................................................................................................................. 19

2.7.1 Code formatting style .......................................................................................................... 19

2.7.2 Reference configuration file for code formatting .............................................................. 20
3 SCA...........................................................................................................................................23
  3.1 General assumptions ...........................................................................................................23
    3.1.1 Specified interfaces....................................................................................................23
    3.1.2 Architecture assumptions.........................................................................................23
  3.2 Conformance...........................................................................................................................24
    3.2.1 Radio platform items...................................................................................................24
    3.2.2 Radio application items...............................................................................................24
  3.3 PIM API mapping ...................................................................................................................24
    3.3.1 Root namespace .........................................................................................................24
    3.3.2 Modules.......................................................................................................................24
    3.3.3 Types..........................................................................................................................25
    3.3.4 Interface declaration properties ...............................................................................25
    3.3.5 API types....................................................................................................................25
    3.3.6 Exceptions..................................................................................................................25
    3.3.7 Services interfaces .....................................................................................................25
    3.3.8 Services primitives .....................................................................................................25
  3.4 PIM attributes mapping ........................................................................................................26
    3.4.1 Mapping approach .......................................................................................................26
    3.4.2 Types..........................................................................................................................26
  3.5 SCA PSM management interfaces .......................................................................................26
    3.5.1 SCA management façade ...........................................................................................27
    3.5.2 SCA functional ports ..................................................................................................27
    3.5.3 SCA 2.2.2 management interfaces ..........................................................................27
    3.5.4 SCA 4.1 management interfaces ..............................................................................29
    3.5.5 PSM specific behaviors .............................................................................................31
  3.6 Specialization of PIM unspecified concepts .........................................................................31
    3.6.1 Access capabilities .......................................................................................................31
    3.6.2 CONFIGURED state ....................................................................................................31
  3.7 SCA functional ports ............................................................................................................31
    3.7.1 Service-wise assignment ............................................................................................32
    3.7.2 Services group-wise assignment .................................................................................32
    3.7.3 Ports implementation ..................................................................................................32
    3.7.4 Assignment mismatch ...............................................................................................32
  3.8 SCA PSM properties .............................................................................................................33
    3.8.1 Versions properties .....................................................................................................33
  3.9 IDL files ................................................................................................................................33
    3.9.1 Service interface declarations ....................................................................................33
    3.9.2 Types declarations .......................................................................................................33
    3.9.3 Files naming ................................................................................................................34
  4 FPGA .......................................................................................................................................35
  4.1 General assumptions ............................................................................................................35
  4.2 Conformance..........................................................................................................................36
    4.2.1 Radio platform items...................................................................................................36
    4.2.2 Radio application items................................................................................................36
  4.3 PIM API mapping ..................................................................................................................36
    4.3.1 Interface declaration properties ..................................................................................36
4.3.2 Modules..................................................................................................................36
4.3.3 Types......................................................................................................................36
4.3.4 API types...............................................................................................................37
4.3.5 Exceptions............................................................................................................37
4.3.6 Services interfaces .............................................................................................37
4.3.7 Services primitives.............................................................................................37
4.4 Specialization of PIM unspecified concepts.........................................................41
  4.4.1 Access capabilities.............................................................................................41
4.5 Referenced PIM version.........................................................................................41
4.6 VHDL packages.....................................................................................................41
5 References................................................................................................................42
  5.1 Referenced documents.........................................................................................42
END OF THE DOCUMENT............................................................................................43
List of Figures

Figure 1  Reference architectural pattern ................................................................. 1
Figure 2  Positioning of native C++ functional interfaces .......................................... 4
Figure 3  Class diagram of a native C++ façade .......................................................... 9
Figure 4  Class diagram of explicit services access ...................................................... 12
Figure 5  Class diagram of generic services access ..................................................... 16
Figure 6  Architecture concepts for SCA PSMs ......................................................... 24
Figure 7  SCA 2.2.2 PSM management interfaces .................................................... 28
Figure 8  SCA 4.1 PSM management interfaces ........................................................ 30
Figure 9  Positioning of FPGA functional interfaces ................................................ 35
Figure 10  Typical RST signal synchronisms ............................................................... 38
Figure 11  Exceptions notification mechanism .......................................................... 40
Contributors

The following individuals and their organization of affiliation are credited as Contributors to development of the specification, for having been involved in the work group that developed the draft then approved by WInnForum member organizations:

- Marc Adrat, FKIE,
- Guillaume Delbarre, DGA,
- David Hagood, Cynosure,
- Olivier Kirsch, KEREVAL,
- Frederic Le Roy, ENSTA Bretagne,
- Francois Levesque, NordiaSoft,
- Chuck Linn, L3Harris,
- David Murotake, HiKE,
- Eric Nicollet, Thales Communications & Security,
- Kevin Richardson, MITRE,
- Sarvpreet Singh, Viavi Solutions.
WInnForum Facilities PSMs Mapping Rules

1 Introduction

1.1 Document purpose

This document specifies the mapping rules for development of PSM specifications derived from PIM specifications of facilities of the Wireless Innovation Forum, based on the general approach described by Principles for WInnForum Facility Standards [Ref1].

In light of the principles established [Ref1] the document addresses the following access paradigms:

- Native C++ (see section 2),
- SCA (see section 3),
- FPGA (see section 4).

1.2 Reference architectural pattern

The reference architectural pattern introduced by section 3.1 of [Ref1] is applied for all access paradigms, and can be illustrated as follows:

A radio application is possibly composed of a number of application components distributed across a composition of processing nodes.

The radio platform implements a number of functional support capabilities, accessible on a number of processing nodes through software interfaces presented by façades.

The façades are the software parts of a functional support capability implementation that present a number of service interfaces for employment by application components.

The set of service interfaces supported by a façade belongs to the API specified by the PIM specification of the considered functional support capability, and is derived by the PSM specification according to the applied access paradigm.

Nothing prevents a given processing node to support more than one access paradigm.
1.3 Reference definitions

The following definitions from *Principles for WINnForum Facility Standards* [Ref1] are applied:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Used definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base concepts</td>
<td>radio application, application component, radio platform, functional</td>
</tr>
<tr>
<td></td>
<td>support capability</td>
</tr>
<tr>
<td>Architecture concepts</td>
<td>façade, access paradigm, processing node</td>
</tr>
<tr>
<td>WINnForum facilities</td>
<td>facility, PIM specification, PSM specification</td>
</tr>
<tr>
<td>Services</td>
<td>service, service interface, provide service, use service, services group</td>
</tr>
<tr>
<td>Primitives</td>
<td>primitive, type, exception</td>
</tr>
<tr>
<td>Attributes</td>
<td>attribute, property</td>
</tr>
</tbody>
</table>

Table 1 Definitions from *Principles for WINnForum Facility Standards*

1.4 Conventions

1.4.1 Case conventions

The following case conventions are applicable:

- **Pascal case:** ThisIsMyExample, ScalsAnAcronym,
- **Snake case:** this_is_my_example, sca_is_an_acronym,
- **Screaming snake case:** THIS_IS_MY_EXAMPLE, SCA_IS_AN_ACRONYM,
- **Camel case:** thisIsMyExample, scaIsAnAcronym.

1.4.2 Functional support capability identification

A *functional support capability* can be identified using a tag (typically an acronym or an abbreviation of the full name of the *functional support capability*).

In order for *functional support capabilities* tags to have at least three characters, the letter “F”, standing for “facility”, may be added. This is for instance the case for “time service”, which tag is “TSF”.

The possible formal identifiers for *functional support capabilities* are specified as follows:

- `<FSC_TAG>`: screaming snake case of the *functional support capability* tag (e.g., used for prefixes identifiers),
- `<FscTag>`: Pascal case of the *functional support capability* tag (e.g., used for version identification properties),
- `<fsc_tag>`: Snake case of the *functional support capability* tag (e.g., used for VHDL constructs of the FGPA PSM),
- `<FscFull>`: Pascal case of the *functional support capability* name (e.g., used for main namespaces).
The following table gives examples of application of the previous rule:

<table>
<thead>
<tr>
<th>Functional support capability</th>
<th>Possible formal identifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>&lt;FSC_TAG&gt;</code></td>
</tr>
<tr>
<td>Time service</td>
<td>TSF</td>
</tr>
<tr>
<td>Transceiver</td>
<td>XCVR</td>
</tr>
</tbody>
</table>

Table 2  Examples of functional support capability formal identifiers

1.5 Conformance

1.5.1 Radio platform items

A façade is conformant with a PSM specification if it implements the service interfaces and complies with any PSM-specific additional requirements.

1.5.2 Radio application items

An application component is conformant with the PSM specification if it can use façades conformant with the PSM specification, without using any non-standard service interface for the functional support capability.
2 Native C++

The native C++ access paradigm is defined as an access paradigm based on direct native C++ connection between application components and façades.

It is based on two C++ versions: C++ 11 (see [Ref2]) and C++ 2003 (see [Ref3]).

A native C++ PSM specification is defined as a standard specifying, according to the native C++ access paradigm, interfaces between instances of radio applications and instances of the addressed functional support capability.

2.1 General assumptions

2.1.1 Specified interfaces

The C++ interfaces specified by a native C++ PSM specification are native C++ functional interfaces.

The native C++ functional interfaces are defined as C++ interfaces derived from the service interfaces of a PIM specification.

2.1.2 Architecture assumptions

A native C++ node is defined as a processing node supporting the Native C++ access paradigm.

A native C++ façade is defined as a façade of a functional support capability instance that runs within a native C++ node.

A native C++ instance is defined as a functional support capability instance with at least one of its façades being a native C++ façade.

A native C++ application component is defined as a module of a radio application implemented in a native C++ node that employs at least one native C++ façade.

The following figure illustrates the positioning of native C++ functional interfaces:

![Figure 2 Positioning of native C++ functional interfaces](image)

A native C++ application component can:

- Be a component of the radio application running in the same native C++ node,
- A proxy of a component of the radio application running in a remote processing node.
In the proxy case, the remote component complies with a *PSM specification* that may be:

- The *native C++ PSM specification*, if the remote *processing node* is another *native C++ node*,
- Another *PSM specification*, if the remote *processing node* is not a *native C++ node*.

The proxy uses a connectivity mechanism between the *native C++ node* and the remote *processing node* that can typically be standard compliant (e.g. MHAL Communication Service, MOCB, CORBA), or be a proprietary solution.

### 2.2 Conformance

#### 2.2.1 Radio platform items

A *native C++ façade* is **conformant with** the *native C++ PSM specification* of a *functional support capability* if it provides an implementation of the **Facade** class and its related service interfaces.

#### 2.2.2 Radio application items

A *native C++ application component* is **conformant with** the *native C++ PSM specification* if it can use *native C++ façades* conformant with the *native C++ PSM specification*, without using any non-standard service interface for the *functional support capability*.

### 2.3 PIM API mapping

The mapping generally complies, for C++11 [Ref2] or later, and for C++03 [Ref3] when applicable, with the IDL to C++11 language mapping standardized by the OMG [Ref4]. Exceptions are identified and justified, typically for implementation efficiency or specification simplicity.

For C++03 features not covered by [Ref4] specific mapping decisions have been taken.

#### 2.3.1 Root namespace

The root namespace for the *functional support capability* of interest is **specified as** `WInnF_Cpp::<FscFull>`.

The `WInnF_Cpp::` namespace is common to all native C++ PSMs root namespaces.

Example: `WInnF_Cpp::TimeService`.

#### 2.3.2 Modules

The *services groups* modules specified by *PIM specifications* map to C++ namespaces defined in `WInnF_Cpp::<FscFull>` namespace, keeping the modules names.

Example: `WInnF_Cpp::TimeService::SystemTimeAccess`.
2.3.3 Types

This section focuses on the Ultra-Lightweight (ULw) profile specified by *IDL Profiles for Platform-Independent Modeling of SDR Applications* [Ref5] extended with the `long long` and `unsigned long long` types.

The corresponding IDL keywords for Basic Types **map to** C++ types as follows.

The `boolean` type **maps to** the C++ native type `bool`.

The `boolean` constants `TRUE` and `FALSE` **map to** the C++ native `bool` constants `true` and `false`.

Each other basic type **maps to** the corresponding type of `<cstdint>` *(stdint.h)*:

<table>
<thead>
<tr>
<th>IDL Basic Type</th>
<th>C++ type of <code>&lt;cstdint&gt;</code> <em>(stdint.h)</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>octet</td>
<td><code>uint8_t</code></td>
</tr>
<tr>
<td>short</td>
<td><code>int16_t</code></td>
</tr>
<tr>
<td>unsigned short</td>
<td><code>uint16_t</code></td>
</tr>
<tr>
<td>long</td>
<td><code>int32_t</code></td>
</tr>
<tr>
<td>unsigned long</td>
<td><code>uint32_t</code></td>
</tr>
<tr>
<td>long long</td>
<td><code>int64_t</code></td>
</tr>
<tr>
<td>unsigned long long</td>
<td><code>uint64_t</code></td>
</tr>
</tbody>
</table>

Table 3 Basic types mapping to `<cstdint>` types

The IDL keywords for Constructed Types **map to** C++ concepts as follows:

- `typedef` maps to the C++ `typedef`,
- `struct` maps to the C++ `struct`,
- `enum` maps to:
  - For C++03, the C++ `enum`,
  - For C++11, the C++ `enum class`.

The IDL keyword `sequence` (a Template Type) **maps to** C++ concepts as follows:

- For C++03, classes defined according to the OMG C++ language mapping rules of [Ref4] section 5.15,
- For C++11, the class `std::vector`.

2.3.4 Interface declaration properties

The *interface declaration properties* specified by PIM specifications **map to** C++ preprocessor identifiers or integer constants, which names are the *interface declaration properties* names prefixed by `<FSC_TAG>`.
2.3.5 **API types**

The API types specified by PIM specifications map to C++ types, whose names are identical to PIM names.

2.3.6 **Exceptions**

The exceptions specified by PIM specifications map to C++ exception classes inheriting from the exceptions defined in `<stdexcept>`.  

2.3.7 **Services interfaces**

The service interfaces specified by PIM specifications map to C++ interface classes, keeping the service interfaces names of the PIM.

2.3.8 **Primitives**

The primitives specified by PIM specifications map to C++ member methods of the service interfaces of the native C++ PSM, which has signatures created by applying the C++ language mapping rules [Ref4] of the Object Management Group (OMG).

2.4 **Specialization of PIM unspecified concepts**

This section specifies, for Native C++, concepts of the PIM specification whose complete specifications are deferred to PSM specifications.

2.4.1 **Access capabilities**

2.4.1.1 Functional support capability access

The Facade class (see section 2.6) provides functional support capability access.

One instance of the Facade class is implemented by each Native C++ processing node involved in implementation of a functional support capability.

Those instances enable native C++ application components to access to the services available on each individual native C++ façade.

2.4.1.2 Services access

The Facade class (see section 2.6) provides services access to the services implemented by a native C++ façade.

The services access can be explicit or generic.
2.4.2 Entry in CONFIGURED state

The activeServicesInitialized() method of the Facade class (see section 2.5) enables a native C++ application component to indicate, before an instance of the functional support capability enters in the CONFIGURED state, that all calls on the Facade for services access are completed.

2.5 Referenced PIM version

<FSC_TAG>_PIM_VERSION is specified as a preprocessing-defined value indicating the version of the PIM specification from which a native C++ PSM specification is derived.

For versions specified in the 3-digits VX.Y.Z form, <FSC_TAG>_PIM_VERSION is the hexadecimal value resulting from the concatenation of one octet for each of the X, Y, and Z digits.

Example values of <FSC_TAG>_PIM_VERSION are:

<table>
<thead>
<tr>
<th>PIM specification version</th>
<th>&lt;FSC_TAG&gt;_PIM_VERSION value</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1.0.0</td>
<td>0x010000</td>
</tr>
<tr>
<td>V2.0.0</td>
<td>0x020000</td>
</tr>
<tr>
<td>V3.4.10</td>
<td>0x03040A</td>
</tr>
</tbody>
</table>

Table 4 Examples of <FSC_TAG>_PIM_VERSION values

2.6 Facade class

The Facade class is specified as a class providing native C++ application components with access to native C++ façades.

For functional support capabilities featuring the CONFIGURED state, the Facade class owns activeServicesInitialized() and activeServicesReleased() methods.

The Facade class also owns at least one of the following interfaces for services access: ExplicitServicesAccess (see section 2.6.4) or GenericServiceAccess (see section 2.6.5).

The preprocessing variables EXPLICIT_SERVICES_ACCESS and GENERIC_SERVICES_ACCESS specify if explicit or generic services access is implemented.
The **Facade** class is represented by the following class diagram:

![Class diagram of a native C++ façade](image)

**Figure 3** Class diagram of a native C++ façade

A *native C++ node* needs to implement one instance of **Facade** for each *native C++ façade* it implements.

The lifecycle of the **Facade** instances is *unspecified*.

A *native C++ node* needs to provide the *native C++ application components* with visibility to the available **Facade** instances. How such visibility is given is *unspecified*.

### 2.6.1 activeServicesInitialized method

#### 2.6.1.1 Overview

*activeServicesInitialized()* commands the *native C++ façade* to proceed to the **CONFIGURED** state.

A precondition to call *activeServicesInitialized()* is that the *native C++ application component* has access to the required *services* implemented by the *native C++ façade*.

#### 2.6.1.2 Associated properties

Not applicable.

#### 2.6.1.3 Declaration

The C++ primitive of the operation **is specified as**:

```cpp
void activeServicesInitialized();
```

#### 2.6.1.4 Parameters

None.

#### 2.6.1.5 Returned value

None.
2.6.1.6 Originator

*Native C++ application component.*

2.6.1.7 Exceptions

The *exceptions* of the *operation* are specified by the following table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>UninitializedServiceException</td>
<td>The <em>native C++ application component</em> did not access to at least one <em>service</em> instantiated by the <em>native C++ façade</em>.</td>
</tr>
<tr>
<td>AlreadyInConfiguredStateException</td>
<td>The <em>native C++ façade</em> is already in <em>CONFIGURED</em> state.</td>
</tr>
</tbody>
</table>

*Table 5* *activeServicesInitialized()* exceptions

### 2.6.2 activeServicesReleased method

2.6.2.1 Overview

*activeServicesReleased()* commands the *native C++ façade* to exit from the *CONFIGURED* state.

A precondition to call *activeServicesReleased()* is that the *native C++ application component* will no longer use the *services* implemented by the *native C++ façade*.

2.6.2.2 Associated properties

Not applicable.

2.6.2.3 Declaration

The *C++ primitive* of the operation is specified as:

```c++
void activeServicesReleased();
```

2.6.2.4 Parameters

None.

2.6.2.5 Returned value

None.

2.6.2.6 Originator

*Native C++ application component.*
2.6.2.7 Exceptions

The exceptions of the operation are specified by the following table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>NotInConfiguredException</td>
<td>The façade is not in CONFIGURED state.</td>
</tr>
</tbody>
</table>

Table 6 activeServicesReleased() exceptions

2.6.3 Services access principles

2.6.3.1 Provide services access

The native C++ application components access to the employed provide services thanks to a “get” method returning a pointer to an instance of the service interface implemented by the native C++ façade.

The native C++ façade ensures that the returned pointer stays valid:
- If the functional support capability features the CONFIGURED state, until it has exited the CONFIGURED state,
- Otherwise, until the functional support capability is not to be used any further.

The native C++ façade keeps exclusive ownership of the provide service referenced by the returned pointer.

The native C++ application component is not responsible for releasing the provide service, and may not attempt to modify or delete the pointer.

2.6.3.2 Use services access

The employed use services of native C++ façades access to native C++ application component thanks to a “set” method taking as input a pointer to an instance of the service interface implemented by the native C++ application component.

The native C++ application component ensures that the passed pointer stays valid:
- If the functional support capability features the CONFIGURED state, until it has exited the CONFIGURED state,
- Otherwise, until the functional support capability is not to be used any further.

The native C++ application component keeps exclusive ownership of the use service referenced by the passed pointer.

The native C++ application component is not responsible for releasing the use service, and may not modify or delete the pointer until the use service is released.

2.6.3.3 Standard services access

Two standard solutions for services access: explicit and generic.

An implementation of a functional support capability needs to comply with at least one of those, in order to discourage use of proprietary approaches that hampers interoperability.
2.6.4 Explicit services access

2.6.4.1 Overview

An explicit services access is specified as a solution for services access with primitives names explicitly reflecting the names of available services.

It is suitable for constrained native C++ nodes where avoidance of program memory overheads is critical or where dynamic cast is not available.

The solution corresponds to value Explicit of ActiveServicesAccess property.

The pattern for explicit services access is represented by the following class diagram:

![Class diagram of explicit services access](image)

2.6.4.2 ExplicitServicesAccess interface

The ExplicitServicesAccess class owns one member method of the form get<ProvideService>() per provide service and one member method of the form set<UseService>() per use service.

2.6.4.2.1 get<ProvideService> methods

2.6.4.2.1.1 Overview

Each get<ProvideService>() returns a pointer referencing the instance of ProvideService interface implemented by the native C++ façade.

2.6.4.2.1.2 Associated properties

Not applicable.

2.6.4.2.1.3 Declaration

The C++ signature of the methods are specified as:

```cpp
<ProvideService1>* get<ProvideService1>();
<ProvideService2>* get<ProvideService2>();
...
```

2.6.4.2.1.4 Parameters

None.
2.6.4.2.1.5 Returned value

If an instance of a provide service is available, the method returns a `<ProvideService> *` pointer to its service interface.

Otherwise, the function returns `NULL`.

2.6.4.2.1.6 Originator

Native C++ application component.

2.6.4.2.1.7 Exceptions

The exceptions of the methods are specified by the following table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>UnavailableServiceException</td>
<td>The service is not available.</td>
</tr>
</tbody>
</table>

Table 7 get<ProvideService>() exceptions

2.6.4.2.1.8 Behavior requirements

A native C++ façade needs to, on a call to a `get<ProvideService>()`:

- If no instance of the provide service is available:
  - If exceptions are supported, throw `UnavailableServiceException`,
  - If exceptions are not supported, set the return value to `NULL`,
- If the provide service is available, set the return value with a reference to its service interface,
- Return the call,
- Keep the provide service active at least until the instance of the functional support capability has exited the `CONFIGURED` state, if applicable.

2.6.4.2.2 set<UseService> method

2.6.4.2.2.1 Overview

`set<UseService>()` passes a pointer used by the façade to reference the instance of a `UseService` implemented by the native C++ application component.

2.6.4.2.2.2 Associated properties

Not applicable.

2.6.4.2.2.3 Declaration

The C++ signature of the methods are specified as:

```c++
void set<UseService1>( <UseService1>* reference);
void set<UseService2>( <UseService2>* reference);
...```
### 2.6.4.2.2.4 Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference</td>
<td>&lt;UseService&gt;*</td>
<td>Reference of the <em>use service</em> implemented by the <em>native C++ application component.</em></td>
</tr>
</tbody>
</table>

**Table 8** Specification of *set<UseService>()* parameters

### 2.6.4.2.2.5 Returned value

None.

### 2.6.4.2.2.6 Originator

*Native C++ application component.*

### 2.6.4.2.2.7 Exceptions

The *exceptions* of the methods are specified by the following table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>UnavailableServiceException</td>
<td>The service is not available.</td>
</tr>
<tr>
<td>InvalidReferenceException</td>
<td>The pointer type is not compatible with the service interface class.</td>
</tr>
</tbody>
</table>

**Table 9** *set<UseService>()* exceptions

### 2.6.4.2.2.8 Behavior requirements

A *native C++ façade* needs to, on a call to a *set<UseService>():*

- If no instance of the *use service* is available:
  - If *exceptions* are supported, throw *UnavailableServiceException*.
  - If the *use service* is available, use *reference* to set a pointer of the *native C++ façade* to reference the *service interface* implemented by the *native C++ application component*.
    - Throw *InvalidReferenceException* exception in case of a type mismatch,
  - Return the call,
  - Use the referenced *service interface* until the instance of the *functional support capability* implementation exits the *CONFIGURED* state, if applicable.

**Note:** this implies that the *native C++ application component* remains valid until the *functional support capability* exits the *CONFIGURED* state, if applicable.
2.6.4.3 `<FSC_TAG>_ExplicitServicesAccess.hpp`

The content of `<FSC_TAG>_ExplicitServicesAccess.hpp` is specified as:

```cpp
#ifndef <FSC_TAG>_EXPLICIT_SERVICES_ACCESS
#define <FSC_TAG>_EXPLICIT_SERVICES_ACCESS

namespace WinnF_Cpp
{
    namespace <FscFull>
    {
        namespace ActiveServicesAccess
        {
            // Access to specific active services
            class ExplicitServicesAccess
            {
                public:
                    // Provide services instances
                    // <services group 1>
                    virtual <Interface 1> *get<Interface 1>()
                        = 0;
                    virtual <Interface 2> *get<Interface 2>()
                        = 0;

                    // <services group 2>
                    virtual <Interface 3> *get<Interface 3>()
                        = 0;

                    // Use services
                    // <services group 3>
                    virtual void set<Interface 4>(
                        <Interface 4> *reference) = 0;
                    virtual void set<Interface 5>(
                        <Interface 5> *reference) = 0;

                    virtual ~ExplicitServicesAccess() NOEXCEPT {}
                }
            } // namespace ActiveServicesAccess
        } // namespace <FscFull>
    } // namespace WinnF_Cpp
} // ifndef <FSC_TAG>_EXPLICIT_SERVICES_ACCESS
```

2.6.5 **Generic services access**

2.6.5.1 **Overview**

A **generic services access** is specified as a solution for services access with a generic syntax, which facilitates insertion of additional services and takes advantage of C++ RTTI features such as `dynamic_cast<>`.

It is suitable for high-end **native C++ nodes** where the processing resources overhead of RTTI features are not an issue and where compilers fully support RTTI.

The solution corresponds to value **Generic** of **ActiveServicesAccess** property.
Generic services access is represented by the following class diagram:

![Class diagram of generic services access](image)

Figure 5  Class diagram of generic services access

2.6.5.2 Object class

Each service implemented by a native C++ façade needs to extend the Object class:

```cpp
class Object {
    public:
        virtual ~Object() {};
}
```

2.6.5.3 GenericServicesAccess interface

The GenericServicesAccess class enables native C++ application components to access to all implemented services.

The GenericServicesAccess class contains getProvideService() and setUseService().

2.6.5.3.1 getProvideService Operation

2.6.5.3.1.1 Overview

getProvideService() returns to the native C++ application component a reference to a provide service instantiated by the native C++ façade.

2.6.5.3.1.2 Associated Properties

None.

2.6.5.3.1.3 Declaration

The C++ signature of the operation is specified as:

```cpp
Object *getProvideService( const char *serviceName);
```

2.6.5.4 Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>serviceName</td>
<td>const char *</td>
<td>Name of the provide service to return a reference on.</td>
</tr>
</tbody>
</table>

Table 10 Specification of getProvideService() parameters
2.6.5.4.1.1 Return value
Pointer to \textit{Object}, to be dynamically cast (using \textit{dynamic\_cast<>}) to the appropriate interface.

2.6.5.4.1.2 Originator
\textit{Native C++ application component}.

2.6.5.4.1.3 Exceptions
The \textit{exceptions} of the \textit{primitive} are \textbf{specified by} the following table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{UnavailableServiceException}</td>
<td>The service is not available.</td>
</tr>
</tbody>
</table>

\textbf{Table 11 \textit{getProvideService()} exceptions}

2.6.5.4.1.4 Behavior requirements

A \textit{native C++ façade} needs to, on a call to \textit{getProvideService()}:  
\begin{itemize}
  \item Search for \texttt{serviceName} among the \textit{service interfaces} names of implemented \textit{provide services},
  \item If no instance of \textit{provide service} corresponding to \texttt{serviceName} is available:
    \begin{itemize}
      \item If \textit{exceptions} are supported, throw \texttt{UnavailableServiceException},
      \item If \textit{exceptions} are not supported, set the return value to \texttt{NULL},
    \end{itemize}
  \item If a \textit{provide service} corresponding to \texttt{serviceName} is available, set the return value with a reference to its \textit{service interface},
  \item Return the call,
  \item Keep the \textit{provide service} active at least until the instance of the \textit{functional support capability} has exited the \texttt{CONFIGURED} state, if applicable.
\end{itemize}

2.6.5.4.2 \textit{setUseService} Operation

2.6.5.4.2.1 Overview

\textit{setUseService()} specifies to the \textit{native C++ façade} a valid reference to a \textit{use service} implemented by the \textit{native C++ application component}.

2.6.5.4.2.2 Associated Properties
None.

2.6.5.4.2.3 Declaration

The C++ signature of the \textit{primitive} \textbf{is specified as}:

\begin{verbatim}
void setUseService( const char *serviceName, Object *useService);
\end{verbatim}
2.6.5.4.2.4 Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>serviceName</td>
<td>const char *</td>
<td>Name of the use service to which a reference is specified.</td>
</tr>
<tr>
<td>useService</td>
<td>Object *</td>
<td>Object reference to the native C++ application component implemented the specified use service.</td>
</tr>
</tbody>
</table>

Table 12 Specification of setUseService() parameters

2.6.5.4.2.5 Return value
None.

2.6.5.4.2.6 Originator
Native C++ application component.

2.6.5.4.2.7 Exceptions

The exceptions of the primitive are specified by the following table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>UnavailableServiceException</td>
<td>The service is not available.</td>
</tr>
<tr>
<td>InvalidReferenceException</td>
<td>The pointer / reference type is not compatible with the service class.</td>
</tr>
</tbody>
</table>

Table 13 setUseService() exceptions

2.6.5.4.2.8 Behavior requirements

A native C++ façade needs to, on a call to setUseService():

- Search for serviceName among the service interfaces names of implemented use services,
- If no instance of use service corresponding to serviceName is available:
  - If exceptions are supported, throw UnavailableServiceException.
- If a use service corresponding to serviceName is available, use useService to make a dynamic cast (using dynamic_cast<>) to reference the service interface implemented by the native C++ application component,
  - Throw InvalidReferenceException exception in case of types mismatch,
- Return the call,
- Use the referenced service interface until the instance of the functional support capability implementation exits the CONFIGURED state, if applicable.

Note: this implies that the native C++ application component remains valid until the functional support capability exits the CONFIGURED state, if applicable.
2.6.5.5 `<FSC_TAG>_GenericServicesAccess.hpp`

The `<FSC_TAG>_GenericServicesAccess.hpp` file shall be used with no user adaptation.

The content of `<FSC_TAG>_GenericServicesAccess.hpp` is specified as:

```cpp
#ifndef <FSC_TAG>_GENERIC_SERVICES_ACCESS
#define <FSC_TAG>_GENERIC_SERVICES_ACCESS

#include "<FSC_TAG>_Types.hpp"

namespace WINNF_Cpp
{
    namespace <FscFull>
    {
        namespace ActiveServicesAccess
        {
            class Object
            {
                public:
                    virtual ~Object() NOEXCEPT {};
            }

            class GenericServicesAccess
            {
                public:
                    // Access to active provide services
                    virtual Object *getProvideService(
                        const char *serviceName) = 0;

                    // Access to active use services
                    virtual void setUseService(
                        const char *serviceName,
                        Object *useService) = 0;
                    virtual ~GenericServicesAccess() NOEXCEPT {};
                } // namespace ActiveServicesAccess
            } // namespace <FscFull>
        } // namespace WINNF_Cpp
    } // namespace <FscFull>
} // namespace WINNF_Cpp
#endif // ifndef <FSC_TAG>_GENERIC_SERVICES_ACCESS
```

2.7 Header files

2.7.1 Code formatting style

The code formatting convention used for native C++ header files (*.hpp) comply with Allman style formatting [Ref6], complemented with C++ specific additions.

The following code snippet gives an example of classes declarations using Allman style:

```cpp
class SomeClass
{
public:
    virtual ~SomeClass() {};
    virtual void MemberOne() = 0;
};
```
2.7.2 Reference configuration file for code formatting

In order to avoid detailed specification of the Allman style convention, the clang-format tool
(http://clang.llvm.org/) is used.

Usage of the following configuration file, created with a couple of edits on top of the default
configuration file provided by the tool for Allman style, is recommended:

```---
Language:        Cpp
AccessModifierOffset: -2
AlignAfterOpenBracket: Align
AlignConsecutiveMacros: true
AlignConsecutiveAssignments: true
AlignConsecutiveBitFields: true
AlignConsecutiveDeclarations: true
AlignEscapedNewlines: Right
AlignOperands:   Align
AlignTrailingComments: true
AllowAllArgumentsOnNextLine: true
AllowAllConstructorsOnNextLine: false
AllowAllParametersOfDeclarationOnNextLine: true
AllowShortEnumsOnASingleLine: true
AllowShortBlocksOnASingleLine: Never
AllowShortCaseLabelsOnASingleLine: false
AllowShortFunctionsOnASingleLine: Empty
AllowShortLambdasOnASingleLine: All
AllowShortIfStatementsOnASingleLine: Never
AllowShortLoopsOnASingleLine: false
AlwaysBreakAfterDefinitionReturnType: None
AlwaysBreakAfterReturnType: None
AlwaysBreakBeforeMultilineStrings: false
AlwaysBreakTemplateDeclarations: MultiLine
BinPackArguments: false
BinPackParameters: false
BreakBeforeBinaryOperators: None
BreakBeforeBraces: Allman
BreakBeforeInheritanceComma: false
BreakInheritanceList: BeforeColon
BreakBeforeTernaryOperators: true
BreakConstructorInitializersBeforeComma: false
BreakConstructorInitializers: BeforeColon
BreakAfterJavaFieldAnnotations: false
BreakStringLiterals: true
ColumnLimit:     80
CommentPragmas:  '^ IWYU pragma:'
CompactNamespaces: false
ConstructorInitializerAllOnOneLineOrOnePerLine: false
ConstructorInitializerIndentWidth: 4
ContinuationIndentWidth: 4
Cpp11BracedListStyle: true
DeriveLineEnding: true
DerivePointerAlignment: false
DisableFormat:   false
ExperimentalAutoDetectBinPacking: false
FixNamespaceComments: true
ForEachMacros:   
  - foreach
  - Q_FOREACH
  - BOOST_FOREACH
IncludeBlocks:   Preserve
```
| IncludeCategories: | - Regex: | `'^"(llvm|llvm-c|clang|clang-c)/'` |
| Priority: | 2 |
| SortPriority: | 0 |
| - Regex: | `'^<|"(gtest|gmock|isl|json)/'` |
| Priority: | 3 |
| SortPriority: | 0 |
| - Regex: | `'.*'` |
| Priority: | 1 |
| SortPriority: | 0 |
| IncludeIsMainRegex: | `(Test)?$` |
| IncludeIsMainSourceRegex: | `''` |
| IndentCaseLabels: | false |
| IndentCaseBlocks: | true |
| IndentGotoLabels: | true |
| IndentPPDirectives: | BeforeHash |
| IndentExternBlock: | AfterExternBlock |
| IndentWidth: | 3 |
| IndentWrappedFunctionNames: | false |
| InsertTrailingCommas: | None |
| JavaScriptQuotes: | Leave |
| JavaScriptWrapImports: | true |
| KeepEmptyLinesAtTheStartOfBlocks: | true |
| MacroBlockBegin: | `''` |
| MacroBlockEnd: | `''` |
| MaxEmptyLinesToKeep: | 1 |
| NamespaceIndentation: | All |
| ObjCBinPackProtocolList: | Auto |
| ObjCBreakIndentWidth: | 2 |
| ObjCBreakBeforeNestedBlockParam: | true |
| ObjCSpaceAfterProperty: | false |
| ObjCSpaceBeforeProtocolList: | true |
| PenaltyBreakAssignment: | 2 |
| PenaltyBreakBeforeFirstCallParameter: | 19 |
| PenaltyBreakComment: | 300 |
| PenaltyBreakFirstLessLess: | 120 |
| PenaltyBreakString: | 1000 |
| PenaltyBreakTemplateDeclaration: | 10 |
| PenaltyExcessCharacter: | 1000000 |
| PenaltyReturnTypeOnItsOwnLine: | 1000 |
| PointerAlignment: | Right |
| ReflowComments: | true |
| SortIncludes: | true |
| SortUsingDeclarations: | true |
| SpaceAfterCStyleCast: | false |
| SpaceAfterLogicalNot: | false |
| SpaceAfterTemplateKeyword: | false |
| SpaceBeforeAssignmentOperators: | true |
| SpaceBeforeCpp11BracedList: | false |
| SpaceBeforeCtorInitializerColon: | true |
| SpaceBeforeInheritanceColon: | true |
| SpaceBeforeParens: | ControlStatements |
| SpaceBeforeRangeBasedForLoopColon: | true |
| SpaceInEmptyBlock: | false |
| SpaceInEmptyParentheses: | false |
| SpacesBeforeTrailingComments: | 1 |
| SpacesInAngles: | false |
| SpacesInConditionalStatement: | false |
| SpacesInContainerLiterals: | true |
| SpacesInCStyleCastParentheses: | false |
| SpacesInParentheses: | false |
| SpacesInSquareBrackets: | false |
| SpaceBeforeSquareBrackets: | false |
Standard: Latest
StatementMacros:
  - Q_UNUSED
  - QT_REQUIRE_VERSION
TabWidth: 8
UseCRLF: false
UseTab: Never
WhitespaceSensitiveMacros:
  - STRINGIZE
  - PP_STRINGIZE
  - BOOST_PP_STRINGIZE
...
3 SCA

The SCA access paradigm is defined as an access paradigm based on SCA connections between application components and façades.

It is based on two SCA versions: SCA 2.2.2 (see [Ref7]) and SCA 4.1 (see [Ref8]).

An SCA PSM specification is defined as a standard specifying, according to the SCA access paradigm, interfaces between instances of radio applications and instances of the addressed functional support capability.

3.1 General assumptions

3.1.1 Specified interfaces

The SCA interfaces specified by an SCA PSM specification are functional interfaces and management interfaces.

The SCA PSM functional interfaces are defined as SCA interfaces used between radio applications and instances of functional support capabilities.

The SCA PSM functional interfaces are mapped from the PIM specification according to the rules specified in section 3.3.

The SCA PSM management interfaces are defined as the subset of the SCA standard interfaces used for configuration and management of SCA instances of a functional support capability.

The SCA PSM management interfaces are selected from the standard SCA 2.2.2 or SCA 4.1 interfaces as specified in section 3.5.

3.1.2 Architecture assumptions

An SCA node is defined as a processing node that supports the SCA access paradigm.

An SCA façade is defined as a façade of a functional support capability instance that runs within an SCA node and conforms to the associated SCA PSM specification.

An SCA façade is composed of one or several SCA components.

An SCA instance is defined as functional support capability instance with at least one of its façades being an SCA façade.

An SCA application component is defined as an SCA application component running in an SCA node.

An SCA application is defined as a radio application with at least one of its components being an SCA application component.
The following figure illustrates the previous concepts:

![Architecture concepts for SCA PSMs](image)

**Figure 6  Architecture concepts for SCA PSMs**

### 3.2 Conformance

#### 3.2.1 Radio platform items

An *SCA façade is conformant with* the SCA PSM specification of a functional support capability if it provides an SCA implementation of *service interfaces*.

#### 3.2.2 Radio application items

An *SCA application component is conformant with* the SCA PSM specification of a functional support capability if it can use *SCA façades conformant with the SCA PSM specification*, without using any non-standard *service interface* for the functional support capability.

### 3.3 PIM API mapping

#### 3.3.1 Root namespace

The SCA PSM root namespace for a *functional support capability is specified as* `WINnF_Sca::<FscFull>`.

The `WINnF_Sca::` namespace is common to all SCA PSMs root namespaces.

Example: `WINnF_Sca::TimeService`.

#### 3.3.2 Modules

The *services groups modules* specified by a *PIM specification map to* SCA namespaces defined in `WINnF_Sca::<FscFull>`, keeping the modules names.

Example: `WINnF_Sca::TimeService::SystemTimeAccess`. 
3.3.3 Types

The IDL keywords for Basic Types, Constructed Types and Template Types used by the *PIM specification* map to same keywords in *SCA PSM specifications*.

3.3.4 Interface declaration properties

Using the concept of interface declaration properties, a *PIM specification* can specify, if needed, options in the signature of *primitives of service interfaces*.

Support of such options in IDL implies to specify different IDL files, distinguishing in the file name the selected options, in order to avoid, for interoperability across ORBs, multiple definitions in the same IDL interface.

It is therefore desirable that *SCA PSM specifications* limit the number of *service interfaces* options to the greatest extent, while keeping the options that enable significant savings in execution resources usage.

3.3.5 API types

The *API types* specified by a *PIM specification* map to same IDL types in the derived *SCA PSM specification*.

3.3.6 Exceptions

The *exceptions* specified by a *PIM specification* map to same IDL exceptions in the derived *SCA PSM specification*.

*EXCEPTIONS_SUPPORT* is always equal to *true*, meaning the *exceptions* mechanism of IDL is always used.

The IDL of an *SCA PSM specification* declares all the possibly supported *exceptions*, those mapped from the *PIM specification* and those specific to the *SCA PSM specification*.

*EXCEPTIONS* is left for implementer usage as specified in the *PIM specification*.

3.3.7 Services interfaces

The *service interfaces* specified by a *PIM specification* map to same IDL *service interfaces* in the derived *SCA PSM specification*.

3.3.8 Services primitives

The *primitives* specified by a *PIM specification* map to same IDL *primitives* in the derived *SCA PSM specification*.
3.4 PIM attributes mapping

3.4.1 Mapping approach

An attribute specified by a PIM specification maps to an SCA Property specified by an SCA PSM specification if a case for its implementation by an SCA instance is identified. The PIM attributes of categories capabilities and properties map to SCA Properties in accordance with the types mapping rules specified in section 3.4.2. Since PIM attributes of type variables are only accessed via access primitives, they don’t need to be mapped to PSM level SCA Properties.

3.4.2 Types

3.4.2.1 Simple types

PIM attributes with a simple type map to SCA Properties of the PSM specification as follows:
- Name: same as the mapped PIM attribute,
- Type: SCA type corresponding to the PIM attribute type.

3.4.2.2 Enumerated types

PIM attributes with an enumerated type map to SCA Properties of the PSM specification as follows:
- Name: same as the mapped PIM attribute,
- Type: SCA long type,
- Reserved values: specified for the possible enumeration values.

3.4.2.3 Structure types

PIM attributes with a structure type map to one simple SCA Property of the PSM specification for each field of the structure.

3.4.2.4 Constants

For constant values of signed types, the signed number representation is used in IDL declarations instead of the hexadecimal representation (e.g. -12 instead of 0xFFFFFFF4), in order to be supported by more IDL compilers.

For constant values of structured types, elementary scalar constants are specified for each field, in order to be supported by more IDL compilers.

3.5 SCA PSM management interfaces

The SCA PSM management interfaces enable control of:
The life cycle of an SCA instance,
The transitions with the CONFIGURED state of the SCA instance, if applicable,
The connection of SCA functional ports with implemented services.

SCA applications should avoid usage of SCA PSM management interfaces.
The SCA PSM management interfaces for SCA 2.2.2 and SCA 4.1 are different.
The property <FscTag>ScaVersion (see section 3.8.1.2) indicates the used SCA version.

3.5.1 SCA management façade

The SCA management façade of an SCA instance is defined as its unique SCA façade that supports SCA PSM management interfaces.
The SCA management façade needs to at least provide the SCA PSM management interfaces specified for the used version of SCA.
An SCA instance may have other SCA façades than its SCA management façade.
In such case, the SCA management façade hides away any implementation details related to the other SCA components, and can be an SCA Aggregate Device.

3.5.2 SCA functional ports

An SCA functional port is defined as an SCA port of an SCA façade that implements one or several service interfaces specified by the PIM specification.
A “provide” SCA functional port connects an SCA application component to a provide service of an SCA façade.
A “use” SCA functional port connects a use service of an SCA façade to an SCA application component.
The SCA PSM management interfaces enable SCA application components and SCA façades to be connected to each other using SCA functional ports.

3.5.3 SCA 2.2.2 management interfaces

This section specifies normative content applicable in case SCA 2.2.2 is used.

3.5.3.1 Software interfaces

The SCA 2.2.2 PSM management interfaces are specified as the SCA 2.2.2 interface CF::Device and the interfaces it inherits from (CF::LifeCycle, CF::PortSupplier, CF::PropertySet, CF::TestableObject and CF::Resource).
The following figure, extracted from [Ref7], identifies the SCA 2.2.2 PSM management interfaces and their inheritance relationships:

![Diagram showing SCA 2.2.2 PSM management interfaces]

SCA 2.2.2 main specification [Ref7] specifies the SCA 2.2.2 PSM management interfaces in the following sections:

- **CF::LifeCycle**: section 3.1.3.1.2,
- **CF::PortSupplier**: section 3.1.3.1.4,
- **CF::PropertySet**: section 3.1.3.1.5,
- **CF::TestableObject**: section 3.1.3.1.3,
- **CF::Resource**: section 3.1.3.1.6,
- **CF::Device**: section 3.1.3.3.1.

3.5.3.2 Behavior requirements

The specified behaviors are standard SCA 2.2.2 behaviors extended by PSM-specific behaviors. A number of aspects are explicitly left unspecified.

3.5.3.2.1 LifeCycle

An SCA management façade needs to implement the `initialize()` and `releaseObject()` operations of the SCA 2.2.2 **CF::LifeCycle** interface according to the standard behavior.

3.5.3.2.2 PortSupplier

An SCA management façade needs to implement the `getPort()` operation of the SCA 2.2.2 **CF::PortSupplier** interface according to its standard behavior.
An SCA management façade needs to implement the `getPort()` operation so that it can return the SCA functional ports of all provide services.

All SCA management façades need to implement the `connectPort()` and `disconnectPort()` operations of the SCA 2.2.2 `CF::PortSupplier` interface according to the standard behavior. All SCA management façades need to implement the `connectPort()` and `disconnectPort()` operations so that they can connect and disconnect the SCA functional ports of all use services.

### 3.5.3.2.3 PropertySet

An SCA management façade needs to implement the `configure()` and `query()` operations of the SCA 2.2.2 `CF::PropertySet` interface according to the standard behavior. The set of supported Configure properties is *unspecified*.

### 3.5.3.2.4 TestableObject

An SCA management façade needs to implement the `runTest()` operation of the SCA 2.2.2 `CF::TestableObject` interface according to the standard behavior. The set of supported tests is *unspecified*.

### 3.5.3.2.5 Resource

An SCA management façade needs to implement the `start()` and `stop()` operations of the SCA 2.2.2 `CF::Resource` interface according to the standard behavior. Section 3.5.5 specifies additional requirements applicable to `start()` and `stop()` implementations. The value of the `identifier` attribute is *unspecified*.

### 3.5.3.2.6 Device

An SCA management façade needs to implement the `allocateCapacity()` and `deallocateCapacity()` operations of the SCA 2.2.2 `CF::Device` interface according to the standard behavior. The set of supported Capacity properties is *unspecified*.

### 3.5.4 SCA 4.1 management interfaces

This section specifies normative content applicable in case SCA 4.1 is used.

#### 3.5.4.1 Software interfaces

The *SCA 4.1 PSM management interfaces are specified as* the SCA 4.1 interfaces `CF::LifeCycle`, `CF::PortAccessor` and `CF::ControllableInterface`. Usage of other SCA 4.1 standard interfaces is *unspecified*. 

---

Copyright © 2022 The Software Defined Radio Forum Inc.
All Rights Reserved
SCA 4.1 main specification [Ref8] specifies the *SCA 4.1 PSM management interfaces* in the following sections:

- **CF::LifeCycle**: section 3.1.3.2.1.3,
- **CF::PortAccessor**: section 3.1.3.2.1.2,
- **CF::ControllableInterface**: section 3.1.3.2.1.6.

The following figure, extracted from [Ref8], identifies the *SCA 4.1 PSM management interfaces*:

![Diagram](image)

**Figure 8 SCA 4.1 PSM management interfaces**

### 3.5.4.2 Behavior requirements

The specified behaviors are standard SCA 4.1 behaviors completed by PSM-specific behaviors.

#### 3.5.4.2.1 LifeCycle

An *SCA management façade* needs to implement the *initialize()* and *releaseObject()* operations of the SCA 4.1 **CF::LifeCycle** interface according to the standard behavior.

#### 3.5.4.2.2 PortAccessor

An *SCA management façade* needs to implement the *connectUsesPorts()*, *disconnectPorts()*, and *getProvidesPorts()* operations of the SCA 4.1 **CF::PortAccessor** interface according to their standard behavior.

An *SCA management façade* may implement *getProvidesPorts()* operation so that it can return the SCA functional ports of all provide services.

An *SCA management façade* needs to implement the *connectUsesPorts()* and *disconnectPorts()* operations so that they can connect and disconnect the SCA functional ports of all use services.

#### 3.5.4.2.3 ControllableInterface

An *SCA management façade* needs to implement the *start()* and *stop()* operations of the SCA 4.1 **CF::ControllableInterface** interface according to the standard behavior.

Section 3.5.5 specify additional requirements applicable to *start()* and *stop()* implementations.
3.5.5 **PSM specific behaviors**

3.5.5.1 `start()`

`start()` indicates to an *SCA instance* that any external configuration (including any ports connection) is completed prior to usage by a *radio application*.

The `start()` implementation needs to trigger an *SCA instance* to enter the `CONFIGURED` state of its state machine, if applicable.

The `start()` implementation returns once the transition to `CONFIGURED` state is completed, if applicable.

3.5.5.2 `stop()`

`stop()` indicates to an *SCA instance* that it is no longer used by a *radio application*.

The `stop()` implementation may trigger an *SCA instance* to exit from the `CONFIGURED` state, if applicable.

3.6 **Specialization of PIM unspecified concepts**

This section specifies, for SCA, concepts of the *PIM specification* whose complete specifications are deferred to *PSM specifications*.

3.6.1 **Access capabilities**

3.6.1.1 Functional support capability access

The *SCA management façade* of an *SCA instance* provides *functional support capability access*.

3.6.1.2 Services access

The *SCA functional ports* provides *services* access to the *services* implemented by an *SCA instance*.

3.6.2 **CONFIGURED state**

The SCA `start()` and `stop()` operations of the *SCA PSM management interfaces* (see section 3.5.5) enable *SCA application components* to trigger the transitions to and from the `CONFIGURED` state, if applicable.

3.7 **SCA functional ports**

There are two possibilities to assign functional interfaces to *SCA functional port*.

The choice between those assignment possibilities is up to the implementer.
3.7.1 Service-wise assignment

A service-wise assignment is defined as the assignment approach where each service interface is assigned to a dedicated SCA port.

Benefits of service-wise assignment are support for finer-grained optionality (exhaustive support of all possible option) and ports of the model reflecting the used services.

3.7.2 Services group-wise assignment

A services group-wise assignment is defined as the assignment approach where all the service interfaces of a services group are assigned to a dedicated SCA port.

In case a services group has optional services not supported by an implementation, a services group-wise assignment would not present the un-implemented services.

Benefits of services group-wise assignment are easier use with fewer connections, possibly some minor savings in code size and XML manual writing effort, and possibly some minor runtime performance improvements, especially with SCA 2.2.2.

3.7.3 Ports implementation

The SCA functional ports need to support connections to all services of an SCA instance of a functional support capability with the radio application.

The PSM specifications may only specify indicative names for SCA functional ports.

The SCA functional ports connections need to be specified by the Software Assembly Descriptor (SAD) file (see [Ref7] and [Ref8]) of the radio application.

For services group-wise assignment, provide ports need to support multiple port connections to enable usage by an object applying service-wise assignment.

In an SCA 4.1 dynamic uses port creation context, the implementer needs to be concerned about the overhead for each created dynamic port.

3.7.4 Assignment mismatch

An assignment mismatch is a situation where a radio application and a radio platform do not follow the same assignment approach.

Since connecting an SCA port with services group-wise assignment to an SCA port with service-wise assignment is not possible, ports refactoring is needed. It is recommended to make the needed refactoring on the radio application side.

Since connecting an SCA port with service-wise assignment to an SCA port with services group-wise assignment is possible, mismatch situations can be handled by making multiple connections.
3.8 SCA PSM properties

An *SCA PSM property* is defined as an SCA property attached to an SCA instance of a *functional support capability*.

3.8.1 Versions properties

3.8.1.1 Referenced PIM version

The `<FscTag>PimVersion` SCA PSM property is specified as an integer indicating the version of the *PIM specification* of a *functional support capability* from which an *SCA PSM specification* is derived.

The defined values for `<FscTag>PimVersion` are specified as:

<table>
<thead>
<tr>
<th>PIM specification version</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1.0.0</td>
<td>0x010000</td>
</tr>
</tbody>
</table>

Table 14 `<FscTag>PimVersion` defined values

3.8.1.2 SCA versions

The `<FscTag>ScaVersion` SCA PSM property is specified as an integer indicating the used SCA version.

The defined values for `<FscTag>ScaVersion` are specified as:

<table>
<thead>
<tr>
<th>SCA version</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.2</td>
<td>0x020202</td>
</tr>
<tr>
<td>4.1</td>
<td>0x040100</td>
</tr>
</tbody>
</table>

Table 15 `<FscTag>ScaVersion` defined values

3.9 IDL files

3.9.1 Service interface declarations

Each *service interface* of the SCA PSM has one dedicated IDL file. This enables implementers to adjust the size of the created binaries to the set of interfaces actually implemented by each *façade*.

The names of the specified IDL files are the concatenation of the IDL modules names with the IDL interface name.

3.9.2 Types declarations

The types declarations required for the *service interfaces* are contained in dedicated IDL files specified on a per *services groups* basis.

The names of the specified IDL files are the concatenation of the IDL modules names with *Types* postfix.
3.9.3 Files naming

The IDL files are named using Pascal case, with a prefix $Sca<FscTag>$ followed by $<FileDescription>$ name and the .idl extension.

Examples: $ScaTsfTypes.idl$ and $ScaTsfSpecificTimeHandling.idl$. 
4 FPGA

4.1 General assumptions

FPGA functional interfaces are defined as the FPGA interfaces derived from the service interfaces of a PIM specification.

An FPGA PSM specification is defined as a specification that standardizes FPGA functional interfaces between instances of radio applications and functional support capabilities.

An FPGA node is defined as an FPGA of a radio platform providing radio applications with FPGA functional interfaces related to one or several functional support capabilities.

An FPGA façade is defined as a façade of a functional support capability instance that executes within an FPGA node.

An FPGA applicative module is defined as a module of a radio application implemented in an FPGA node that employs at least one FPGA façade.

The following figure illustrates the positioning of FPGA functional interfaces:

![Figure 9 Positioning of FPGA functional interfaces](image)

The FPGA applicative module can:
- Be a component of the radio application running in the same FPGA node,
- A proxy of a component of the radio application running in a remote processing node.

In the proxy case, the remote component conforms with a PSM specification that may be:
- The FPGA PSM specification, if the remote processing node is another FPGA node,
- Another PSM specification, if the remote processing node is not an FPGA node.

The proxy uses a connectivity mechanism between the FPGA node and the remote processing node that can typically be a standard (e.g. MHAL Communication Service, MOCB), an FPGA extension of CORBA, or a proprietary solution.
4.2 Conformance

4.2.1 Radio platform items

An FPGA façade is conformant with the FPGA PSM specification of a functional support capability if it provides an FPGA implementation of service interfaces.

4.2.2 Radio application items

An FPGA applicative module is conformant with the FPGA PSM specification of a functional support capability if it can use FPGA façades conformant with the FPGA PSM specification, without using any non-standard service interface for the functional support capability.

4.3 PIM API mapping

4.3.1 Interface declaration properties

The interface declaration properties specified by a PIM specification map to conditional declarations depending on value of constants.

4.3.2 Modules

The root module and services groups modules of a PIM specification map to no concept.

4.3.3 Types

The IDL keywords for Basic Types used by a PIM specification map to the following concepts:

- boolean maps to 1-bit signal,
- short and unsigned short map to a 16-bit vector,
- long and unsigned long map to a 32-bit vector,
- unsigned long long map to a 64-bit vector.

The default representation is unsigned.

For signed types, the mention “signed” reminds that a signed representation is applied.
float maps to no standard PSM concept.

The IDL keywords for Constructed Types used by a PIM specification map to the following concepts:

- typedef maps to types declarations,
- struct maps to structures declarations, when supported,
- enum maps to a vector which size is equal to the number of bits required to encode all enumerated values.
The IDL keywords for Template Types listed by a PIM specification is **sequence**, which maps to a set of RTL signals enabling transfer of a number of elements equal to the sequence length.

### 4.3.4 API types

Each API type specified by a PIM specification maps to a PSM API type.

For RTL, no formal identifier is used in column “Format”.

For VHDL, the name of a PSM API type is a snake-case identifier defined as follows:
- Snake-case transformation of the PIM API type name,
- Followed by postfix _type.

### 4.3.5 Exceptions

The exceptions specified by a PIM specification map to the exceptions RTL signals specified in section 4.3.7.3.5.

Usage of exceptions RTL signals is not mandatory.

### 4.3.6 Services interfaces

The service interfaces of by a PIM specification map to no formalized concept.

When a service interface has several primitives, a note introduced by “To be implemented with the other FPGA primitives (…)” indicates which other FPGA primitives are to be jointly implemented.

### 4.3.7 Services primitives

An FPGA primitive is defined as the FPGA interface derived from a primitive specified by a PIM specification.

An FPGA primitive is specified for each primitive of the PIM specification, as an RTL (Register-Transfer Level) [Ref9] digital interface specified in two steps:
- Specification of the RTL signals,
- Specification of the associated chronogram.

The specification is independent from the programming language used (e.g. VHDL or Verilog).

#### 4.3.7.1 RTL signals origin

The origin of an RTL signal is defined as the FPGA module controlling the signal.

The origin can either be an FPGA applicative module or an FPGA façade.

The caller is defined as the FPGA module making calls to an FPGA primitive.

The callee is defined as the FPGA module receiving calls made to an FPGA primitive.
For an FPGA primitive of a provide service, the caller is the FPGA applicative module and the callee is the FPGA façade.

For an FPGA primitive of a use service, the caller is the FPGA façade and the callee is the FPGA applicative module.

4.3.7.2 Primitive prefix

A **primitive prefix** is defined as the prefix used to name all the signals of the RTL signals set of an FPGA primitive.

A primitive prefix concatenates:

- The `<FSC_TAG>` field, identifying the functional support capability of interest,
- The `<instNum>` field, optionally identifying instances of functional support capability in case there are more than one (starting count from 1),
- The `<PRIM_NAME>` field, identifying the FPGA primitive using a screaming snake case transcription of the primitive name in the PIM specification.

4.3.7.3 Base RTL signals

4.3.7.3.1 Structural RTL signals

The structural RTL signals are defined as the RTL signals conveying clock and reset attached to all FPGA primitives.

The structural RTL signals are specified by the following table:

<table>
<thead>
<tr>
<th>RTL signal name</th>
<th>Origin</th>
<th>Format</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;FSC_TAG&gt;_instNum_PRIM_NAME</code></td>
<td><em>FPGA façade</em></td>
<td>1-bit signal</td>
<td>Clock attached to the FPGA primitive.</td>
</tr>
<tr>
<td>CLK</td>
<td><em>FPGA façade</em></td>
<td>1-bit signal</td>
<td>Hardware reset propagation to the FPGA primitive.</td>
</tr>
</tbody>
</table>

Table 16 Structural RTL signals

The synchronism of the RTL signal RST is unspecified.

The following figure illustrates two typical synchronisms for reset:

![Figure 10 Typical RST signal synchronisms](image)
4.3.7.3.2 Semantics RTL signals

The semantics RTL signals are defined as the RTL signals supporting semantics aspects of an FPGA primitive.

The semantics RTL signals are specified by the following table:

<table>
<thead>
<tr>
<th>RTL signal name</th>
<th>Origin</th>
<th>Format</th>
<th>Usage case</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>Caller</td>
<td>1-bit</td>
<td>No in parameter and no explicit return.</td>
<td>The FPGA primitive is called.</td>
</tr>
<tr>
<td>RDY</td>
<td>Callee</td>
<td>1-bit</td>
<td>Blocking behavior.</td>
<td>The callee is ready to receive a new call on the FPGA primitive.</td>
</tr>
</tbody>
</table>

Table 17 Semantics RTL signals

4.3.7.3.3 Parameters RTL signals

The parameters RTL signals are defined as the RTL signals supporting in parameters, out parameters, and explicit return situations.

The parameters RTL signals are specified by the following table:

<table>
<thead>
<tr>
<th>RTL signal name</th>
<th>Origin</th>
<th>Format</th>
<th>Usage case</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN_IN</td>
<td>Caller</td>
<td>1-bit</td>
<td>in param(s) or explicit return.</td>
<td>The FPGA primitive is called. Validates in param(s).</td>
</tr>
<tr>
<td>DATA_IN.&lt;param_n&gt;</td>
<td>Caller</td>
<td>param_n format</td>
<td>in param(s).</td>
<td>Value of n\textsuperscript{th} in param.</td>
</tr>
<tr>
<td>EN_OUT</td>
<td>Callee</td>
<td>1-bit</td>
<td>Explicit return.</td>
<td>The FPGA primitive returns. Validates out param(s).</td>
</tr>
<tr>
<td>DATA_OUT.&lt;param_n&gt;</td>
<td>Callee</td>
<td>param_n format</td>
<td>out param(s).</td>
<td>Value of n\textsuperscript{th} out param.</td>
</tr>
</tbody>
</table>

Table 18 Parameters RTL signals

<param_n> is the snake case transcription of the parameter name in the PIM specification.

4.3.7.3.4 Sequence RTL signals

The sequence RTL signals are defined as the RTL signals supporting one in parameter of type sequence< type_spec>.

A data item is defined as the elementary piece of information conveyed by a parameter of type sequence.
The sequence RTL signals are specified by the following table:

<table>
<thead>
<tr>
<th>RTL signal name</th>
<th>Origin</th>
<th>Format</th>
<th>Usage case</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;FSC_TAG&gt;_TAG_&lt;instNum&gt;_PRIM_NAME</code> +</td>
<td>Caller</td>
<td>1-bit signal</td>
<td>The data item is the first of the sequence.</td>
<td></td>
</tr>
<tr>
<td><code>&lt;TYPE_SPEC&gt;_FIRST</code></td>
<td>Caller</td>
<td>1-bit signal</td>
<td>The data item is the last of the sequence.</td>
<td></td>
</tr>
<tr>
<td><code>&lt;TYPE_SPEC&gt;_LAST</code></td>
<td>Caller</td>
<td>1-bit signal</td>
<td>Validation of the data item.</td>
<td></td>
</tr>
<tr>
<td><code>&lt;TYPE_SPEC&gt;_EN</code></td>
<td>Caller</td>
<td>1-bit signal</td>
<td>Value of the data item.</td>
<td></td>
</tr>
<tr>
<td><code>&lt;TYPE_SPEC&gt;_DATA</code></td>
<td>Caller</td>
<td><code>&lt;type_spec&gt;</code> format</td>
<td>The callee makes flow control.</td>
<td>The callee is ready to accept a new data item.</td>
</tr>
<tr>
<td><code>&lt;TYPE_SPEC&gt;_RDY</code></td>
<td>Callee</td>
<td>1-bit signal</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 19 Sequence RTL signals

4.3.7.3.5 Exceptions RTL signals

The exceptions RTL signals are defined as the RTL signals supporting notification of exceptions. Support of exceptions RTL signals is optional.

The exceptions RTL signals are specified by the following table:

<table>
<thead>
<tr>
<th>RTL signal name</th>
<th>Origin</th>
<th>Format</th>
<th>Usage case</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;FSC_TAG&gt;_instNum_&lt;PRIM_NAME&gt;</code> +</td>
<td>Callee</td>
<td>1-bit signal</td>
<td>Exception notification</td>
<td>Indicates the exception was detected.</td>
</tr>
<tr>
<td><code>IRQ_&lt;EXCEPTION_NAME&gt;</code></td>
<td>Callee</td>
<td>1-bit signal</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 20 Exceptions RTL signals

`<EXCEPTION_NAME>` is the screaming snake case transcription of the exception name in the PIM specification.

The following figure specifies the exceptions notification mechanism:

![Figure 11 Exceptions notification mechanism](image)

4.3.7.4 Stream-oriented primitives

Mapping of stream-oriented FPGA primitives will be addressed in a future version of the document.
4.4 Specialization of PIM unspecified concepts

This section specifies, for FPGA, concepts of the *PIM specification* whose complete specification are deferred to *PSM specifications*.

4.4.1 Access capabilities

4.4.1.1 Functional support capability access

Handling of *FPGA façades* during development of FPGA layouts enables *FPGA applicative modules* to access to *FPGA façades*.

4.4.1.2 Services access

Usage of the FPGA interfaces during development of FPGA layouts enables *FPGA applicative modules* to be connected to the *services* implemented by *FPGA façades*.

4.5 Referenced PIM version

*<FSC_TAG>_PIM_VERSION* is specified as an FPGA constant indicating the version of the *PIM specification* from which an FPGA *PSM specification* is derived.

For *PIM specifications* versions specified in the 3-digits VX.Y.Z form, *<FSC_TAG>_PIM_VERSION* is the hexadecimal value resulting from the concatenation of one hexadecimal octet for each digit.

Example values of *<FSC_TAG>_PIM_VERSION* are:

<table>
<thead>
<tr>
<th>PIM specification version</th>
<th>&lt;FSC_TAG&gt;_PIM_VERSION value</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1.0.0</td>
<td>0x010000</td>
</tr>
<tr>
<td>V2.0.0</td>
<td>0x020000</td>
</tr>
<tr>
<td>V3.4.10</td>
<td>0x03040A</td>
</tr>
</tbody>
</table>

Table 21 Examples of *<FSC_TAG>_PIM_VERSION* values

4.6 VHDL packages

Normative concepts specified for VHDL programming (see [Ref10]) are VHDL packages. The specified packages need to be compiled in a library named *<fsc_tag>_api*.

A VHDL package named *pkg_<fsc_tag>_api_types.vhd* is specified for declaration of types.

A VHDL package named *pkg_<fsc_tag>_operations_parameters.vhd* is specified for parameters of *FPGA primitives*.

Other VHDL packages can be specified as needed.
5 References

5.1 Referenced documents

https://sds.wirelessinnovation.org/specifications-and-recommendations

http://www.cplusplus.com/

http://www.cplusplus.com/

http://www.omg.org/spec/CPP11/1.2

https://sds.wirelessinnovation.org/specifications-and-recommendations
https://winnf.memberclicks.net/assets/work_products/Specifications/winnf-14-s-0016-v2.0.2.pdf

[Ref6] Allman style section of the Wikipedia article “Indentation style”
https://en.wikipedia.org/wiki/Indentation_style

https://www.jtnc.mil/Resources-Catalog/Category/16990/sca/

[Ref8] The Software Communications Architecture Specification, version 4.1, Joint Tactical Networking Center (JTNC), 20 August 2015
https://www.jtnc.mil/Resources-Catalog/Category/16990/sca/

https://en.wikipedia.org/wiki/Register-transfer_level


The URLs above were successfully accessed at release date.