OpenCPI
Application Development Guide

OpenCPI Release: v2.4.8

Revision History

Revision	Description of Change	Date
1.01	Creation, in part from previous Application Control Interface document	2012-12-10
1.1	Add package naming and ocpirun flags that apply to all, not one, instance. Also the DumpFile attribute. Also some clarifications about property value formats (struct etc.).	2013-02-11
1.2	Add container ordinal and platform options for ocpirun, clarify array value format. Add a few missing Worker methods.	2013-02-28
1.3	Major update.	2015-01-15
1.4	Change to ODT, update for current capabilities and template	2016-02-23
1.5	Improve ocpirun issues, property value syntax, add project and HDL assembly sections	2016-04-28
1.6	Remove draft notation, add some improvements from CDG	2016-05-20
1.7	Update for 2017Q1	2017-02-12
1.8	Update for 2017.Q2, utilities, typed get/setPropertyValue, improved externalPort text	2017-06-20
1.9	Update for 2018.Q1 file I/O options, ocpirun/ACI/OAS consistency table,python,projects Remote containers, more ocpidev	2018-02-21
1.9.1	Initial/partial update for 2018.Q3	2018-08-29
2.0	Update for 1.4.0, string expressions, remote container env vars, ACI Makefile options	2018-08-29
2.1	Update for 1.5, slave attributes/elements, EOF update	2019-04-29
2.2	Update for 1.6 and running XML applications with ocpidev	2020-01-31
2.3	Update for 2.1 for DRC etc.	2021-01-08
2.4	Add shared glossary of terms	2021-03-15
2.5	Refer to ocpiserve, ocpiremote man pages, add dumpProperties, unreadable exceptions	2021-04-18
2.6	General edit and change to “makeless” instructions for applications	2021-06-24
2.6.1	Add missing angle bracket, gain_db -25 line to example p. 68	2022-03-05
2.6.2	Clarify development and runtime attributes in XML files in application directories	2022-03-14
2.6.3	Clarify use of default property values	2022-07-24
2.6.4	Add link to OpenCPI User Guide in kernel module and system.xml descriptions	2022-08-31
2.6.5	Remove references to CentOS or other deprecated platforms	2025-02-13

Table of Contents

1 References

This document depends on the introductory material in the OpenCPI User's Guide. For information on component development, which is *not* a prerequisite of this document, see the ***OpenCPI Component Development Guide (CDG)***.

Table 1: Table of Reference Documents

Title	Published By	Link
OpenCPI User Guide	OpenCPI	https://opencpi.gitlab.io/releases/v2.4.8/docs/OpenCPI_User_Guide.pdf
OpenCPI Component Development Guide	OpenCPI	https://opencpi.gitlab.io/releases/v2.4.8/docs/OpenCPI_Component_Development_Guide.pdf
OpenCPI RCC Development Guide	OpenCPI	https://opencpi.gitlab.io/releases/v2.4.8/docs/OpenCPI_RCC_Development_Guide.pdf

2 Overview

The purpose of this document is to specify how applications can be created and executed in OpenCPI. The OpenCPI framework supports Component-Based Development, where applications are composed of pre-existing components that may exist in ready-to-run binary form before the application is even defined.

An OpenCPI ***component*** is a functional abstraction with a specifically defined control and configuration interface based on ***configuration properties***, and zero or more ***data ports***, each with a defined messaging ***protocol***. An ***OpenCPI Component Specification (OCS)*** describes both of these aspects of a component, establishing interface requirements for multiple implementations (***workers***) of the component. Workers are developed and coded based on an OCS, and when built, are available for applications that are defined in terms of components that meet a spec. An application identifies the components it uses by the name of their spec.

Having one or more libraries of prebuilt, ready-to-run component implementations (a.k.a. ***workers***) is a prerequisite for running OpenCPI applications. The syntax and semantics of component specifications (***OCS***), and how component implementations are developed, are defined in the ***OpenCPI Component Development Guide (CDG)***. Creating and running applications for OpenCPI does ***not*** require the knowledge of how components are developed, but it does require some knowledge of how they are ***specified***. Component specifications are discussed briefly in this document, and described in detail in the CDG.

OpenCPI applications are defined as assemblies of component instances with connections among them. They can be specified two different ways:

1. A standalone XML document (text file).
2. An XML document embedded in, and manipulated by, a C++ program.

This document specifies:

- *The format and contents of the XML documents that define applications*
- *A utility program that directly executes the applications defined in XML files*
- *A C++ and Python API for manipulating and executing XML-based applications*

There are also related sections describing:

- *How to use other network-attached systems when executing applications, a.k.a.* ***remote containers****.*
- *How to develop and run applications in OpenCPI* ***projects****, which are development work areas for groups of assets like components and applications.*
- *How create FPGA artifacts which are* ***HDL assemblies*** *of workers that can implement a subset of an application for execution on an FPGA platform.*

OpenCPI uses several terms when describing component-based applications. In particular:

Component: a specific function used to compose applications. Components are described by an XML document called a ***component specification***.

Instance: the use of a component in an application (a part of an *application*).

Application: the composition of instances

Worker: a concrete implementation of a component, in three contexts: source code, compiled code for some target platform, runtime object executing the compiled code.

Container: the OpenCPI execution environment on some ***platform*** that will execute workers (i.e. where they execute).

Port: a communication interface of a component or worker, with which they send and receive messages with other components/workers.

Property: a configuration value applied to a component to control its function. Components have defined properties with defined data types, and workers implement those properties. Workers (specific implementations) may have additional properties beyond those defined by the component (spec) being implemented.

Platform: a particular type of processing hardware and/or software that can host a container for executing OpenCPI workers.

Artifact: a file containing binary code for one or more workers, built for a platform.

Artifact Library: a collection of artifacts in a hierarchical file system directory structure.

PackageID: a globally unique identifier for OpenCPI assets, in this case used to identify ***component specifications***.

System: a collection of ***platforms*** usually in a box or on a system bus or fabric.

The OpenCPI execution framework for component-based applications is based on workers executing in containers (on platforms), communicating via their ports, and configured via their properties. The runtime workers are instances of component implementations realized in artifact files. The term ***artifact*** is used as a technology neutral term which represents a compiled binary file that is the resulting ***artifact*** of compiling and linking (or for FPGAs, synthesizing etc.) some source code that implements some components. We use the term worker both for a specific (coded) implementation of a component, as well as the runtime instances of that implementation.

The component development and build process results in artifacts that can be loaded as needed and used to instantiate the runtime workers. Typical artifacts are “shared object” or “dynamic library” files on UNIX systems for software workers, and “bitstreams” for FPGA workers. While it is typical for artifacts to hold the implementation code for one worker, it is also common to build artifact files that contain multiple worker implementations.

OpenCPI applications are created by specifying which components should be instantiated (using some ***artifacts***), how the resulting workers should be connected, and how they should be configured via their properties. Specifying an instance is based on the ***package ID*** of the ***component specification***.

The runtime software uses a ***package ID*** to search the available artifact libraries for available implementations in artifacts, and matches those (binary) implementations to the available containers (processors of various types), running on ***platforms*** in the ***system***. The result of the search is a set of potential candidate workers for each instance. To be a candidate, an implementation must be able to execute in some available container.

The package ID of a component specification generally has a prefix followed by package name (followed by a period). This prefix is the name scope in which the component is defined. This allows components to be specified and implemented by different organizations, while still allowing any implementation found in a library to satisfy any (other) organization's component specifications. E.g., my project can have an additional, alternative implementation of a component specified in another library, or can define its own specification for a component with the same name in a different ***package ID*** name scope. OpenCPI package naming follows the Java package naming conventions, with the addition of two special top-level packages: **`local`** and **`ocpi`**.

To actually run the application, the ***deployment decision*** is made for each instance in the application:

- *which implementation/artifact* should be used, and
- *which container* (running on some platform) should it run in.

A set of mutually feasible deployment decisions results in the overall deployment of the application.

Unless a specific implementation is indicated, the **`OCPI_LIBRARY_PATH`** environment variable is used to indicate a list of colon-separated directories or files, which are searched to locate artifact files containing component implementations. The directories are searched recursively. **During this search, when the deployment decisions are made, there may be multiple possible deployments. Each possible deployment is scored and the first deployment among those with the best score is used. If two artifacts are considered equivalent, the one found earlier in `OCPI_LIBRARY_PATH` will be used. `OCPI_LIBRARY_PATH` is used similar to the `LD_LIBRARY_PATH` on Linux systems except that it descends recursively into directories when doing the search.**

The relationships between applications, artifacts, containers, workers, platforms etc. is shown in the following diagram:

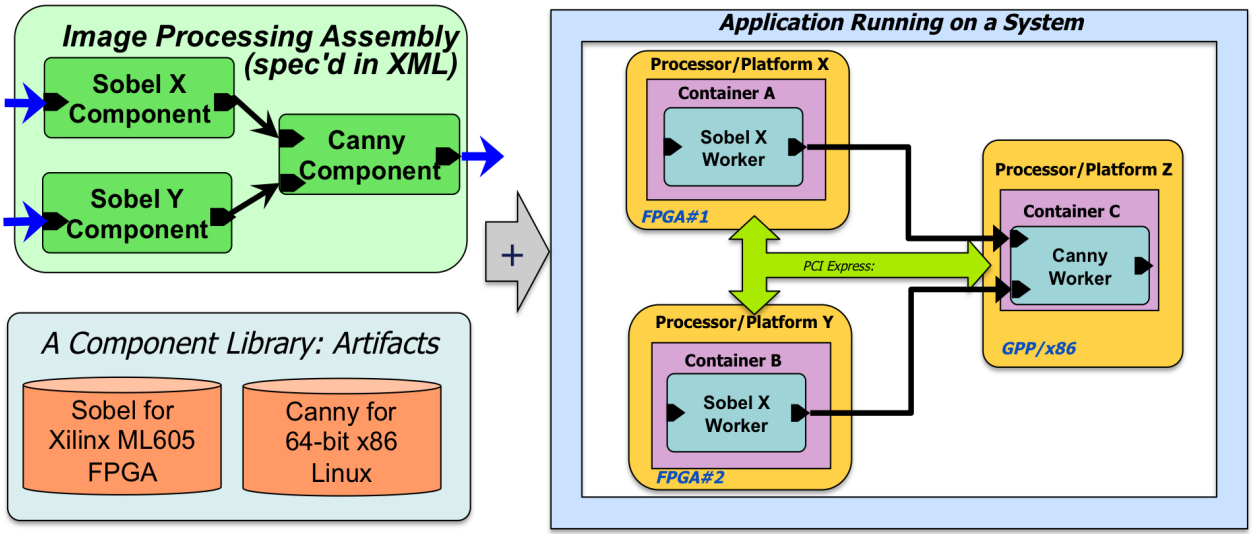

Figure 1: An Application of Components Deployed on a System

3 OpenCPI Application Specification (OAS) XML Documents

This section defines the XML document format for describing OpenCPI applications. Such XML documents may be held in files or in text strings within a program. They describe a collection of component instances, along with their interconnections and configuration properties. An OAS may be directly executed using the **`ocpirun`** utility program described below. An OAS may also be constructed and/or manipulated programmatically and dynamically, and executed using an API described in the later section: Application Control Interface (ACI)

The primary contents of the OAS are *component instances*. When the author of an OAS specifies a component *instance*, they are referring to a component *specification*. They are saying: I need a component implementation that meets this *specification*. Normally, the OAS says only that, and does *not* say which particular implementation of that component spec (i.e. which worker) should be used. This allows the OAS to be used in a variety of different configurations of hardware and different libraries of component implementations.

A simple example of an OAS is below, showing an application that reads data from a file, adds 1 to each data value, and writes the result.

```
<application>
  <instance component='file_read' connect='add1'>
    <property name='filename' value='test.input'/>
  </instance>
  <instance component='add1' connect='file_write'/>
  <instance component='file_write'>
    <property name='filename' value='test.outputwrong'/>
  </instance>
</application>
```

Each instance specifies the component, some properties, and a connection.

When used in a development environment and when the application is in a project, this XML file has other attributes unrelated to the actual execution of an application. See the Application XML Files in Projects section in the Developing Applications in OpenCPI Projects chapter for more details.

3.1 Quick XML Introduction

XML documents are text files that contain information formatted according to XML (E**X**tensible **M**arkup **L**anguage) syntax and structured according to a particular application-specific ***schema***. The textual XML information itself is formatted into ***elements***, ***attributes***, and ***textual content***. The OAS XML schema does not use or allow ***textual content*** at this time. XML ***elements*** have ***attributes*** and ***child elements*** (forming a hierarchy of elements). XML ***elements*** take two forms. The simpler one is when an element has no child (embedded) elements and no ***textual content***. It looks like this (for element of type **`xyz`**, with attribute **`abc`**):

```
<xyz abc='123'/>
```

Thus the element begins with the **<** character and the element type, and is terminated with the **/>** characters. Attributes have values in single or double-quotes. Any white space, indentation, or new lines can be inserted for readability between the element name and attributes or between attributes. Thus the above example could also be:

```
<xyz
  abc="123"
 />
```

When the element has child elements (in this case a child element of type **`ccc`** with attribute **`cat`**), it looks like:

```
<xyz abc="123">
  <ccc cat="345"/>
</xyz>
```

In this case the start of the **`xyz`** element (and its attributes), is surrounded by **<>**, and the end of the **`xyz`** element is indicated by **`</xyz>`**. An XML schema defines which elements, attributes, and child elements the document may contain. Every XML document has a single top-level element that must be structured (attributes and sub-elements) according to the schema. A **`<extension>`** child element is always legal and always ignored. It can be used when tools want private embedded elements that are unknown to OpenCPI.

An element can be entered directly (as above) or by referring to a separate file that contains that element. So the example above might have a file **`ccc1.xml`** containing:

```
<ccc cat="345"/>
```

And then a top-level file called "xyz1.xml" containing:

```
<xyz abc="123">
  <include href="ccc1.xml"/>
</xyz>
```

However, the schema specifies which elements are allowed to be top-level elements in any file. All element and attribute names used in OpenCPI are case ***in***sensitive.

All attributes are defined with specific data types and formats. When an attribute is defined as the boolean type, the default value (used when the attribute is not specified)

is “false” unless otherwise noted. In OpenCPI, element and attribute names are *case **in**sensitive*.

3.2 Top Level Element in an OAS: `application`

The top-level element in every OAS document (file or string) is the **`application`** element. Application elements contain child elements that are either **`instance`** or **`connection`**, and have attributes that are **`name`**, **`finished`**, **`package`**, and **`maxprocessors`**. These are described below.

3.2.1 Name attribute (optional)

The **`name`** attribute of an application is simply used in various error messages and other debug log messages. It has no functional purpose, only documentation and labeling.

3.2.2 Finished attribute (optional)

The **`finished`** attribute identifies the instance within the OAS that is used to determine when the application is "finished" executing. When the indicated instance is "finished", then the whole application is considered "finished". If this attribute is not supplied, the application is considered "finished" when *all* its instances are "finished". The value of this attribute must match the **`name`** attribute of one of the instance elements, described below.

For some applications, and some components, there is no definition or functionality of being "finished". In this case whatever mechanism started the application must decide when to stop it and shut it down.

3.2.3 Package attribute (optional)

The **`package`** attribute of an application is used as a default package prefix for all instances in the assembly. Any instance's **`component`** attribute that does not have a package prefix (has no periods) is assumed to be in the package indicated by this attribute. When not specified, the default package prefix for all components mentioned in the assembly is the package ID of the project the application is in. If the application is not in an OpenCPI project, the default prefix is: **`local`**.

The prefix of the core OpenCPI component library is **`ocpi.core`**. If you are using mostly components in that library, you might include **`package='ocpi.core'`** as an attribute. If you are using only components specified in your own library of components, you could ignore this attribute and use no prefixes at all. See the **`component attribute`** of the **`instance`** element below.

If the value of this attribute starts with a period, then the project's package ID is used as a prefix (or **`local`**, if the application is not in a project). Thus if the project's package ID is **`myorg.myproj`**, and this package attribute is **`.myutilslib`**, then the default prefix for all component attributes is **`myorg.myproj.myutilslib`**.

3.2.4 MaxProcessors attribute (optional)

The **`MaxProcessors`** attribute indicates the maximum number of processors (containers) that should be used to run the application. When instances are allocated to

processors, an algorithm decides which processor runs each instance. If this attribute is not set, the default behavior is to spread the instances across available processors, and use a “round-robin” assignment policy when there are more instances than processors.

If this numeric parameter is set, it limits the number of processors used, if possible. If more are necessary to host the necessary workers, more will indeed be used in any case. An example of when this attribute is *not* effective is when the availability of implementations of each instance dictate that more processors are needed, such as when the only implementation available for an instance is for a particular processor, which must then be used.

3.3 Instance Elements within the Application Element

The **`instance`** element is used as a child of the **`application`** element to specify a component instance in the application. It may have **`property`** or **`slave`** child elements, and may have **`component`**, **`name`**, **`connect`**, **`selection`**, **`from`**, **`to`**, **`slave`** and **`external`** attributes. For example:

```
<instance component='file_read' connect='add1'>
  <property name='filename' value='test.input'/>
</instance>
```

The order of instance elements is significant for the purpose of assigning names to each instance when the instance element does not specify a name. When applications are started, the instances in the application are started using ordering rules that do ***not*** depend on the ordering of instance elements in the OAS. The detailed ordering rules are outside the scope of this document, but in general instances with no input ports (typically sources of data) are started last to avoid startup overrun conditions.

3.3.1 Component attribute (required)

The **`component`** attribute of an instance specifies the component being instantiated. The value is a string used to find implementations for this instance, by searching in the available artifact libraries. It is the name assigned by the component developer to the component *specification* used as the basis for implementations. Component specifications are themselves XML documents/elements called OCS (OpenCPI Component Specification). They have names (used to match this attribute's value), and describe the ports and properties that apply to all implementations of that component.

This attribute is required, and answers the question: *what function should this instance perform?* The process by which OpenCPI searches for implementations based on this attribute is described above in the package attribute section.

This attribute may have a package prefix (ending in a period) to indicate which package contains the component specification indicated. If there is no prefix (and no periods), the package prefix is taken from the default for the whole assembly, which is specified using the **`package`** attribute of the top-level **`application`** element. The component attribute value can have a leading period, which indicates a component name relative to the prevailing package prefix.

This attribute provides the ***PackageID*** of the component by combining any prefix from the application's **`package`** attribute with its value if the value contains no periods. Package IDs are meant to be global identifiers constructed similar to Java package names. The three cases of this attribute are:

1. No periods, simply appended to the prevailing package prefix.
2. Leading period, also appended to the prevailing package prefix.
3. Periods but not leading periods, used by itself with no prefix applied,

The second case is useful to indicate components in component libraries in the same project, e.g. **`component=".mylib.mycomp"`**.

3.3.2 Name attribute (optional)

The **name** attribute of an instance is optional, and provides a unique identifier for the instance within the application. If it is not supplied, one is assigned to the instance. If there is only one instance in the application for a type of component (i.e. the component is used only once), the assigned instance name is the same as the component name (without package prefix). If more than one such instance (of the same component) exists in the application, the assigned name is the component name (without package prefix) followed by the decimal ordinal of that instance among all those for the same component. Such ordinals are assigned starting with 0.

For example, in the example application above, there is one instance of the **file_read** component, and thus its instance name would be **file_read**. If the application used **file_read** twice (e.g. two different components were taking data from different files), the two instances would be named **file_read0** and **file_read1**, in the order they occurred in the OAS.

3.3.3 Connect attribute (optional)

Connecting instances together in an assembly can be done one of two ways. Using the ***connect*** attribute of an instance is the simplest, but cannot express all connections. The **connection** child element of the **application** element can be used to express all types of connections. It is described later.

The ***connect*** attribute defines exactly one connection from an output port of this instance to an input port of another instance. Its value is the name of the other instance. If this instance only has one output port and the other instance only has only one input port, then these are implied. The optional **from** attribute specifies the name of the output port of this instance if needed (if there are more than one), and the optional **to** attribute specifies the name of the input port of the other instance (if there are more than one). An example using all three attributes is:

```
<application>
  <instance component="psd"
            connect='demod' from='myout' to='demod_in'/>
  <instance component="demod" selection='model=="rcc"'/>
</application>
```

This simple connection method is useful for the many components that have only one output port.

3.3.4 Selection attribute (optional)

This attribute optionally specifies how to choose among alternative implementations when more than one is available. This capability also provides a way for the application to specify minimum conditions on the candidate implementations found in the library.

The attribute value is an expression in the syntax of the C language, with all the normal operators, including the **?:** ternary operator. Logical expressions (e.g. “a == 1”) return 1 on true and 0 when false. The variables that may appear in the expression are either:

- Property names that have fixed (not runtime variable) values

- Built-in identifiers that indicate well-known attributes of the implementation. The built-in identifiers are:
  - **`model`**: the name of the authoring model of the implementation, e.g. **`rcc`**.
  - **`platform`**: the name of the platform the implementation is built for, e.g. **`rocky9`** or **`zed`**
  - **`os`**: the name of the operating system the implementation is built for, e.g. **`linux`**

The value of the selection expression is considered an unsigned number, where a higher number is better than a lower number, and zero is considered unacceptable. I.e. if the expression when evaluated for an implementation has a zero value, that implementation is not considered a candidate. A simple example might be:

```
model=="rcc"
```

This indicates that the model must be **`rcc`** since otherwise the expression's value will be zero. The example:

```
error_rate < 5 ? 2 : 1
```

indicates that the **`error_rate`** property is relevant, and if less than 5, it is better than when greater than or equal to 5, but the latter is still acceptable.

If there is no selection expression, the "score" of the implementation is 1, unless it has hard-wired connections to other collocated workers (e.g. on an FPGA). In this case its value is 2.

An example of using the selection attribute is:

```
<application>
  <instance component="psd" selection='latency < 5'/>
  <instance component="demod" selection='model=="rcc"'/>
</application>
```

It indicates that the **`psd`** instance needs an implementation with latency less than 5, and the **`demod`** instance must have an implementation with an authoring model of **`rcc`**.

3.3.5 From attribute (optional)

This attribute is used to specify the name of the output port of this component instance in conjunction with using the **`connect`** attribute described above.

3.3.6 To attribute (optional)

This attribute is used to specify the name of the input port of the other component instance in conjunction with using the **`connect`** attribute described above.

3.3.7 External attribute (optional)

This string attribute is used to specify a port of the instance that is to be considered an external port *of the entire application*. Its value is the name of this instance's port that should be externalized. The external application-level name of the port is the same as its own name on this instance. To specify a different name, use the **`connection`**

element described below. Note that the name of this attribute is singular and distinct from the **`externals`** attribute described next.

3.3.8 Externals attribute (optional)

This *boolean* attribute is used to specify that all unconnected ports of the instance are to be considered external ports *of the entire application*. The external application-level names of the ports are the same as their own name on this instance. To specify different names, use the **`connection`** element described below. Note that the name of this attribute is plural and distinct from the **`external`** attribute described previously.

3.3.9 Worker attribute (optional)

This string attribute is used to specify a particular worker to use for this instance. Usage of this attribute is rare and generally not recommended in the OAS since it bypasses the automatic selection algorithm for choosing the worker based on available implementations and available containers. The string value is the name of the worker, with or without package prefix and with or without any suffix for "authoring model".

3.3.10 Slave attribute (optional, deprecated)

This string attribute specifies the name of another instance in this application that acts as a slave to this instance. This attribute implies that any worker used for this component must be a proxy and that the worker used for the identified slave instance must be an appropriate slave worker for the worker chosen for this proxy. This attribute is used when there is a single slave for this instance. If there are multiple slaves for the instance, the **`slave`** element described in Slave Elements must be used. More information about slaves and proxies is in that section. This attribute is deprecated in favor of the default automatic inclusion of slave workers in the application when a proxy instance is specified in the application.

3.3.11 Buffersize attribute (optional)

This numeric attribute specifies the size of message buffers for the connection on the output port in conjunction with using the **`connect`** attribute described above. For more details on message buffers, see the description of the **`connection`** element.

3.3.12 Property Elements within the Instance Element (optional)

The **`<property>`** element is used as a child of the **`<instance>`** element to specify configuration property values that should be configured in the worker when the application is run, prior to the application being started. If no **`<property>`** element is specified for a property for the instance, the default value for the property (from the component's OCS) is *implicitly* being requested by the application. Within an **`instance`** element, some examples of property (child) elements are:

```
<instance component="psd">
  <property name="size" value="17"/>
  <property name="symmetric" value="true"/>
</instance>
```

Properties can only be set if they are specified in the spec (***OCS***) with access as ***initial*** or ***writable***. Properties labeled in the OCS as ***initial*** can only be assigned values before the application runs, while those labeled as ***writable*** can be specified before the application runs (like ***initial*** properties) but can *also* be changed during execution, dynamically.

Property values may also be set outside the OAS (on the command line or as a runtime-parameter). Using these mechanisms outside the OAS makes the OAS more reusable since it can be used with different property values settings.

3.3.12.1 Name attribute (required)

The **`name`** attribute of a property element must match the name of a property of the specified component. I.e., it must be one of the defined configuration properties of the component. Component specifications define properties that are common to all implementations of a component. Component implementations (workers) can also define additional properties that are specific to that implementation, but mentioning such properties will only be accepted if the selected implementation has them. Otherwise an error results.

*3.3.12.2 Value attribute (one of **`value`** or **`valueFile`** is required)*

The **`value`** attribute is the value to be assigned to the configuration property of the worker just before being started. The attribute's value must be consistent with the data type of the property in the component specification. E.g. if the type of the property is **`ulong`**, then the attribute's value must be numeric and not negative.

The complete syntax of property values is described in the ***Property Value Syntax*** section.

*3.3.12.3 ValueFile attribute (one of **`value`** or **`valueFile`** is required)*

The **`ValueFile`** attribute is the name of a file containing the value to assign to the property. Using this attribute, rather then the **`value`** attribute, is convenient when the value is large, such as when the property's value is an arrays of values. When **`valueFile`** is used, all new lines in the file are interpreted as commas.

The complete syntax of property values is described in the ***Property Value Syntax*** section.

3.3.12.4 DumpFile attribute (optional)

The **`DumpFile`** attribute is the name of a file into which the value of the property will be written after execution. When **`DumpFile`** is used, all commas in the value are replaced by new lines in the written file. The complete syntax of property values is described in the ***Property Value Syntax*** section.

3.3.12.5 Delay attribute (optional)

Normally property values are set before the application starts, but it is possible for a property value to be set ***after*** the application starts. This **`delay`** attribute specifies

when the property value should be set ***after*** the application starts. A value of zero means as soon as possible after the application starts. Any value greater than zero indicates the amount of time to wait (in seconds) after the application starts, before setting the value. The delay values are in units of seconds, and may be floating point. Thus **`5e-6`** would indicate 5 microseconds.

Multiple delays and values may be specified for a property to indicate a timed sequence of property settings. This is done by inserting multiple **`set`** elements as child elements of the **`property`** element. Each **`set`** element may have its own delay and indicate the value to be set using the **`value`** or **`valuefile`** attributes. Here is an example that sets a property's value 5 seconds after the application starts, and then sets a different value 10 seconds after the application starts.

```
<application>
  <instance component='mycomp'>
    <property name='control'>
      <set value='123' delay='5'/>
      <set value='345' delay='10'/>
    </property>
    <property name='other' delay='20' value='789'/>
  </instance>
</application>
```

3.3.13 Slave Elements within the Instance Element (optional, deprecated)

This element is used to specify one or more slave instances that act as slaves to this instance. This element implies that any worker used for this instance must be a proxy and that the worker used for the identified slave instance must be an appropriate slave worker for the worker chosen for this proxy instance. Each **`slave`** element must have a **`name`** attribute whose value is the name of the other instance in the assembly that will act as a slave to this instance.

Workers can have a proxy/slave relationship where a one worker (a proxy) is allowed to control other workers (slaves). A proxy worker is designed and coded to control specific other workers as slaves, and this worker-to-worker relationship is declared by the proxy worker. Workers identified as slaves by proxies have no special declaration themselves and they do *not* need a proxy worker to function properly. This relationship is used by the code in the proxy worker

An example of using the elements is:

```
<instance component="device1"/>
<instance component="device2"/>
<instance component="device_controller">
  <slave name="device1"/>
  <slave name="device2"/>
</instance>
```

This slave element exists to allow slaves to be parameterized and connected within the application. This element is now deprecated. Now, the slaves will be included in the application automatically if a worker used for an instance is a proxy.

3.4 Property Elements within the Application Element (optional)

Property elements at the top level of an application (rather than under an **instance** element), represent properties of the application as a whole. They are essentially a mapping from a top-level property name to a property of some instance in the assembly.

This provides a convenient way to expose properties to the user of an application without requiring them to know the internal structure of the application. For example:

```
<application>
  <property name='infile' instance='file_read' property='filename'/>
  <instance component='file_read' connect='add1'>
    <property name='filename' value='test.input'/>
  </instance>
  <instance component='add1' connect='file_write'/>
  <instance component='file_write'>
    <property name='filename' value='test.outputwrong'/>
  </instance>
</application>
```

The above example provides an application-level property named **infile**, which is mapped to the **filename** property of the **file_read** component instance. In addition to the attributes listed here, the top-level property elements also accept the **value**, **valueFile**, and **dumpFile** attributes described in the previous section.

3.4.1 Name attribute (required)

The **name** attribute of an application-level property is the name that users of the application will use to read, write or display the value. If the **property** attribute just below is not present, then this name is also the name of the instance's property.

3.4.2 Instance attribute (required)

This attribute specifies the name of the instance that actually implements this property for the application. It is the instance that the application-level property is "mapped to".

3.4.3 Property attribute (optional)

If the application-level name of this property is not the same as the instance's property to which is it mapped, this attribute is used to specify the actual property of the instance. It is a string property that must match a property of the instance.

3.5 Connection Elements within the Application Element (optional)

Connecting instances together in an assembly can be done one of two ways. Using the ***connect*** attribute of an instance is the simplest (described above), but cannot express all connections. The ***connection*** child element of the **`application`** element can be used to express *all* types of connections. It describes connections among ports and also with "the outside world", i.e. external to the application. The **`connection`** element has optional **`name`** and **`transport`** attributes, and **`port`** and **`external`** child elements.

An example of an application with some connections is:

```
<application done='file_write'>
  <instance component='file_read'/>
  <instance component='bias'/>
  <instance component='file_write'/>
  <connection transport="socket">
    <port instance='file_read' name='out'/>
    <port instance='bias' name='in'/>
  </connection>
  <connection>
    <port instance='bias' name='out'/>
    <port instance='file_write' name='in'/>
  </connection>
</application>
```

The first connection connects the **`out`** port of the **`file_read`** instance to the **`in`** port of the **`bias`** instance, and specifies that the connection should use the **`socket`** transport mechanism. The second simply connects the **`out`** port of the **`bias`** instance to the **`in`** port of the **`file_write`** instance. This second connection could have been more simply accomplished by using the **`connect`** attribute on the **`bias`** instance.

3.5.1 Name attribute (optional)

This attribute specifies the name of the connection. It is only used for documentation and display purposes and has no specific other function. If it is not present, a name is assigned according to the **`conn`*`<n>`*** pattern, where ***`<n>`*** is the number of the connection in the application (0 origin). If the connection is thought of as a "wire", this is the name of the wire that is attached to various other things (instance ports and external ports). For an example of external ports, see Application getPort Method.

3.5.2 Transport attribute (optional)

This attribute specifies what transport mechanism should be used for this connection. OpenCPI supports a variety of transport technologies and middlewares that convey data/messages from one instance's port to another. Normally the transport mechanism is chosen automatically based on which ones are available and optimal. This attribute allows the application to override the default transport mechanism and force the usage of a particular one. The ones supported at the time of this document update are:

Table 2: Transport Options for Connections

Name	Description
`pio`	Programmed I/O using shared memory buffers between processes
`pci`	DMA or PIO over the PCI Express bus/fabric
`ofed`	RDMA using the OFED software stack, usually for Infiniband
`socket`	RDMA using TCP/IP sockets
`ether`	RDMA using Ethernet (link layer) frames
`udp`	RDMA using UDP/IP

Some of these transport mechanisms are only available if specifically installed in a system.

3.5.3 Buffersize attribute (optional)

This attribute specifies the message buffer size for the connection. Connections convey messages from input ports to output ports, using message buffers. The buffers must be large enough to fit the messages. The exact mechanism for buffering is determined by the low-level mechanisms and technologies used to convey the messages. Normally the OpenCPI runtime framework determines buffer sizes based on a combination of the protocol used for the connection and other information supplied by workers.

When there is no protocol associated with any of the ports of the connection, and no worker-based information to determine the buffer size, a system-wide default buffer size is used, which is 2K bytes.

If the application wants to override the above determination of buffer size, perhaps to reduce latency (by reducing buffer size) or increase throughput (by increasing buffer size), it can use this attribute to do so. Even when this attribute is specified, this value can be overridden by command line arguments or runtime ACI values. In some cases buffer sizes are constrained by hardware limitations.

3.5.4 Port Elements within the Connection Element (optional)

This element is used to specify a port that this connection should be attached to. The most common use of this element is to specify the consumer and producer of the connection, using a ***port*** element for each, within the same ***connection*** element.

3.5.4.1 Instance attribute (required)

This attribute specifies the name of the instance of the port to be attached to this connection. This instance name is used along with the name attribute to specify the port.

3.5.4.2 Name attribute (required)

This attribute specifies the name of the port that should be attached to this connection. This port name is scoped to the instance defined in the instance attribute of the connection element.

3.5.4.3 BufferCount attribute (optional)

This attribute specifies the number of message buffers to use at this port. Increasing this value can sometimes improve throughput at the expense of using more memory.

4 The `ocpirun` Utility Program for Executing XML-based Applications

The simplest way to run an OpenCPI application is to describe it in an XML file (an OAS as described above), and run it using the command-line utility **`ocpirun`**. This command reads the OAS file and runs the application. E.g., if the OAS was in a file named **`myapp.xml`**, the following command would run it:

```
ocpirun myapp
```

With some typical options, the command would be:

```
ocpirun -v -d -t 10 myapp
```

This would be verbose during execution, dump property values after initialization and after execution, and limit execution to 10 seconds.

The execution ends when the application described in the OAS is "finished" or the provided time duration is exceeded. As mentioned above, an application is finished either when all its workers indicate they are "finished" or when a single worker, identified using the **`finished`** attribute in the OAS, says it is finished. The **`ocpirun`** utility also has an option to stop execution after a fixed period of time.

There are a number of options to **`ocpirun`**, which are all printed in the help message when it is executed with no arguments. Options that are "Bool", have no value: their presence indicates true. When an option has a value, the value can immediately follow the option letter, or be in the next argument. There are general options, function options and options that refer to a specific instance in the application. These are all described below. Some options can also be expressed as XML attributes in the OAS and some may also be used in the Application Control Interface described below in ACI. A complete alphabetical option summary, for all three ways of indicating options, is in the Options Summary section.

When ocpirun executes the application, it must make *deployment decisions*, which decide, for each instance:

- ***which*** *worker in which compiled* ***artifact*** should be used, and
- ***which container*** should it run in.

There is an automatic built-in algorithm to make these decisions, as well as a number of options described below that override or guide the automatic deployment algorithm.

The complete interface and options for **`ocpirun`** are described in the ocpirun manual page

5 Property Value Syntax and Ranges

This section describes how property values are formatted to be appropriate for their data types. Property values for applications occur in four places:

- the **`value`** attribute of property elements in the OAS XML
- in the file indicated by the **`valueFile`** attribute of property elements in OAS XML
- on the **`ocpirun`** command line when options are used to set property values
- in C++ when the ACI is used to apply property values to applications (the ACI is described below in ACI)

The syntax accepted depends on the type of the property whose value is being set, and certain quoting requirements depend on the context where the value is specified.

In XML attributes: Attribute values in XML syntax are in single or double quotes. The property value syntax described below is used inside these quotes (in the OAS). To have quotes *inside* XML attribute values, the other type of quotes is used to delimit the attribute value. In either case, inside the quoted attribute value, the **`&`** and **<** characters must be escaped using the official XML notations: **`&`** for **`&`** and **`<`** for **<**. If *both* types of quotes must be in an attribute value, then the official XML escape sequences for the quotes can be used: **`&quot;`** for double quote, and **`&apos;`** for single quote.[1]

On the shell command line: Similarly, when used on the command line for **`ocpirun`**, the shell quoting rules are different than in XML. If the property value has no single quotes at all, then using single quotes for the shell command line argument is the most convenient when any of the shell's *metacharacters* are in the property value. The shell metacharacters are these:

`|  &  ;  (  )  <  >  *  ?  [  ]  {  }` ***`space  tab`***

If single quotes are in the value, or if shell variable or history expansion is required, the ***QUOTING*** section of the shell/bash manual page defines how to escape them.

In a C++ program: In C++, the values will be defined in double-quoted string literals, where only double-quote characters and backslash characters must be escaped by preceding them with a backslash.

These XML/shell/C++ rules are applied after the value is constructed according to the general property value syntax defined below.

Property values are also used when creating component specifications and workers. That usage is described in the ***OpenCPI Component Development Guide***, but the format is as described here.

5.1 Values of Unsigned Integer Types: uchar, ushort, ulong, ulonglong

These numeric values can be entered in decimal, octal with leading zero, or hexadecimal with leading 0x. The limits are the typical ranges for unsigned 8, 16, 32, or 64 bits respectively.

1 The details of XML attribute encoding can be found at Wikipedia XML Character Entities

The **uchar** type can also be entered as a value in single quotes, which indicates that the value is an ASCII character, with backslash escaping as defined in the C language. The syntax inside the single quotes is as described for the **char** type below.

5.2 Values of Signed Integer Types: short, long, longlong

These numeric values can be entered in decimal, octal with a leading zero, or hexadecimal with a leading 0x, with an optional leading minus sign to indicate negative values. The limits are the typical ranges for signed 16, 32, or 64 bits respectively.

5.3 Values of the Type: char

This type is meant to represent a character, i.e. a unit of a string. In software it is represented as a **signed char** type, with the typical numeric range for a signed 8-bit value. The format of a value of this type is simply the character itself, with the typical set of escapes for non-printing characters, as specified in the C programming language and IDL:

\n \t \v \b \r \f \a \\ \? \′ \″

A series of 1-3 octal digits can follow the backslash, and a series of 1-2 hex digits can follow **\x**.

OpenCPI adds two additional escape sequences as a convenience for entering signed and unsigned decimal values of type char. The sequence **\d** may be followed by an optional minus sign (**–**) and one to three decimal digits, limited to the range of -128 to 127. The sequence **\u** can be followed by one to three decimal digits, limited to the range of 0 to 255.

These escapes can also be used in a string value. Due to the requirements of the arrays and sequence values (see below), the backslash can also escape commas and braces, i.e.:

\, \{ \}

5.4 Values of the Types: float and double

These values represent the IEEE floating point types with their defined ranges and precision. The values are those acceptable to the ISO C99 **strtof** and **strtod** functions respectively.

5.5 Values of the Type: bool

These values represent the Boolean type, which is logical true or false. The values can be case insensitive: **true** or **1** for a true value, and **false** or **0** for a false value.

5.6 Values of the Type: string

These values are simply character strings, but also can include all the escape sequences defined for the char type above. Due to the requirements of arrays and sequence values, the backslash can also escape commas and braces (**\,** and **\{** and **\}**). Double quotes may be used to surround strings, which protects commas, braces,

and leading white space. To be interpreted this way, the first character must be a double quote. Two double quotes can represent an empty string.

5.7 Values in a Sequence Type

Values in a sequence type are comma-separated values. When the type of a sequence is **`char`** or **`string`**, backslash escapes are used when the data values include commas.

5.8 Values in an Array Type

When a value is a one-dimensional array, the format is the same as the sequence, with the number of values limited by the size of the array. If the number of comma-separated values is less than the size of the array, the remaining values are filled with the *null* value appropriate for the type. Null values are zero for all numeric types and the type char. Null values for string types are empty strings.

5.9 Values in Multidimensional Types

For multidimensional arrays or sequences of arrays, the curly brace characters (**`{`** and **`}`**) are used to define a sub-value. For example, a sequence of 3 elements, each consisting of arrays of length 3 of type char, would be:

```
{a,b,c},{x,y,z},{p,q,r}
```

This would also work for a 3 x 3 array of type char. Braces are used when an item is itself an array, sequence,or structure recursively.

5.10 Values in Struct Types

Struct values are a comma-separated sequence of members, where each member is a member name followed by white space, followed by the member value. A struct value can be “sparse”, i.e. only have values for some members. If the struct type was:

```
struct { long el[2][3]; string m2; char c; };  // C pseudo code
```

A valid value would be:

```
el {{1,3,2},{4,5,6}}, c x
```

This struct value would not have a value for the **`m2`** member. Unmentioned members have null values.

5.11 Expressions in Property Values

Both numeric and string typed scalar values can be specified using an expression syntax and operator precedence from the C language, where any parameter property with a value can be accessed as a variable. All C expression operators can be used except the comma operator, assignments or self-increments/decrements. The conditional operator using **`?`** and **`:`** *is* supported. Expressions can be used as elements of arrays or sequences, or as structure member values.

For example, if the **`nbranches`** property was a parameter, a valid expression might be:

```
nbranches == 0x123 ? 2k-1 : 0177
```

5.11.1 Numeric Values

The numeric constant syntax is typical C language syntax (integer and floating point), with the following additions:

- Integers with explicit radix after a leading **0** can use **0t** for base 10 and **0b** for base 2, in addition to the normally used **0x** for base 16 and no letter for base 8. All these prefixes can be applied to the fraction and exponent for floating-point syntax.
- Integers can use a letter suffix of **K**, **M**, or **G**, upper or lower case, indicating 2^10, 2^20 or 2^30 respectively. E.g. **2k-1** is 2047.
- All arithmetic is done using a numeric data type exceeding the range and precision of **uint64_t**, **int64_t** and **double**, and then assigned to the actual data type of the property whose value is being specified.
- The ** binary operator (pow) from the python and FORTRAN languages is also supported.

When the value of the expression is assigned to the property value or numeric property attribute (e.g. **ArrayLength**), it is range checked for validity. Boolean properties are set to true if the value is non-zero. Fractions are discarded when assigning values to integer types.

5.11.2 String Values

Within expressions, string constants (using double quotes as in C) and string-valued parameter properties can be used. All comparison operators are case sensitive and result in boolean numeric values (0 or 1). All operators requiring boolean values (**!**, **||**, **&&**, **?:**) use the length of the string (zero being false, otherwise true). The **+** operator concatenates strings. There is no implicit or explicit conversion between string values and numeric values. E.g. if **sparam** is a string-typed parameter with the value **abc**, then this expression has the numeric value of 1:

sparam == “abc”

This expression would have the string value **xyz_abc**:

“xyz_” + sparam

5.11.3 Expressions used for string values

To distinguish when a string value is an expression, as opposed to a string that might look like an expression, the string value must have a special prefix of **\:** (backslash followed by colon). So if a property (or sequence/array element or structure member) is a string type, any value assigned to it is considered not to be an expression by default. To make the string value interpreted as an expression, use the **\:** prefix.

So if **sparam** is a parameter with value **abc**, then specifying the property's string value

sparam+”xyz”

would simply define that exact string (with plus sign and double quotes included) but if the string value was:

```
\:sparam+"xyz"
```

then it would be interpreted as an expression, and the actual string value would be:

```
abcxyz
```

6 API for Executing XML-based Applications in C++/Python: ACI

Although XML applications are easily executed using the **ocpirun** command, there are cases where more programmatic and/or dynamic creation or control of the XML-based application is required. This section describes an API that supports these scenarios, called the ***OpenCPI Application Control Interface (ACI)***. Here are examples of when **ocpirun** may not be sufficient, and may require using the ACI.

4. The contents of the application XML (OAS) need to be constructed programmatically or some of its attributes need to be dynamically set.
5. The C++ main program needs to directly connect to the ports of the running application (see ***external ports*** below), and send or receive data to/from it.
6. The XML-based application needs to be run repeatedly (perhaps with configuration changes) in the same process.
7. Component property values need to be read or written dynamically during the execution of the application.

We use the term ***control-application*** to describe the C++ or Python application using this interface. The ACI is described as a C++ API, with a section Using the ACI with Python describing how using it in Python differs from using it in C++. In all examples below, the C++ namespace prefix **OA** is used as an abbreviation of the actual namespace of the ACI: **OCPI::API**, i.e. assuming:

```
#include "OcpiApi.hh"
namespace OA = OCPI::API;
```

The ACI, for executing XML-based applications, is based primarily on one C++ class: **OCPI::API::Application**. It is constructed by passing it the OAS and has various lifecycle control member functions. It is well suited to being constructed with automatic storage (on the stack) and using the implicit destruction at the end of the block. A simple example using this API, assuming the OAS is in the file **myapp.xml**, is:

```
{
  OA::Application app("myapp.xml");
  app.initialize(); // all resources have been allocated
  app.start();      // execution is started
  app.wait();       // wait until app is "finished"
  app.finish();.....// do end-of-run functions like property dumping
}
```

All exceptions thrown inherit from the **std::string** class, so at a minimum, the value of the string can be used to print an error message to determine what went wrong, e.g.:

```
try {
  OA::Application app("myapp.xml");
  app.initialize(); // all resources have been allocated
  app.start();      // execution is started
  app.wait();       // wait until app is "done"
  app.finish();     // do end-of-run functions like property dumping
} catch (const std::string &e) {
  std::cerr << "app failed: " << e << std::endl;
}
```

When OpenCPI is executing an application, some of the containers may in fact be executing inside the same process, in their own threads. This may occur for software containers running software workers, or even for OpenCL or FPGA simulation containers. Thus using the ACI implies that there will be "background processing" overheads introduced into the process, running in background threads.

This means that callers of the ACI must generally avoid standard library APIs or system calls that are not thread safe. Also, no calls to the **`exit()`** library function should be made until any **`OA::Application`** objects are destroyed. Typically this means that an **`OA::Application`** object should either go out of scope or be explicitly deleted before exiting. E.g. for a simple block, the following example is ***bad*** coding practice::

```
{
  OA::Application app("myapp.xml");
  if (something_bad)
    exit(1);
  ....
}
```

Where as the following is ok:

```
int exitval = 0;
do {
  OA::Application app("myapp.xml");
  if (something_bad) {
    exitval = 1;
    break;
  }
  ....
} while(0);
if (exitval)
  exit(exitval);
```

6.1 Class OA::Application

This class represents a running application, with a simple lifecycle. It has constructors and destructors suitable for automatic storage, and methods for:

- controlling the lifecycle
- getting and setting configuration properties
- directly communicating with the external ports defined in the application.

6.1.1 OA:: Application::Application constructors

There are two constructors for this class that differ in the type of the first argument. The first argument is either a **`const char *`**, or a **`const std::string &`**. It is either a filename containing the OAS, or the OAS XML string itself. If the string starts with the **<** character after initial white space, it is considered the latter (XML). The second argument is a parameter array, of type **`const OA::PValue *`**. It defaults to **`NULL`** (no parameters).

The constructor searches the available artifact libraries as specified in the **`OCPI_LIBRARY_PATH`** environment variable, and, for each instance in the OAS, chooses an implementation from those available in the libraries. Resources are *not* allocated (no loading or instantiating or configuring or connecting is performed). When the constructor returns successfully (without exception), the OAS is valid and implementations (artifacts) have been found and selected for all instances in the OAS. Here are the two constructors:

```
class Application {
  Application(const char *file, OA::PValue *params = NULL);
  Application(std::string &oas, OA::PValue *params = NULL);
};
```

The **`params`** argument is used to provide additional options and constraints on the selection of implementations and the assignment to containers, in addition to providing more property values. All these values could be specified in the OAS, but this allows the OAS to remain constant while various aspects of the execution are overridden or augmented here in the ACI.

The **`property`**, **`selection`**, **`model`**, and **`container`** parameters perform the same function as the **`-p`**, **`-s`**, **`-m`**, and **`-c`** instance options to the ***ocpirun*** utility program. Their values are strings that specify a parameter relative to a particular instance. An example is:

```
{
  OA::PValue params[] = { PVString("model",      "psd1=rcc"),
                          PVString("selection",  "filter=snr<40"),
                          PVString("property",   "filter=mode=6"),
                          PVEnd
                        };
  OA::Application app("myfile.xml", params);
  app.initialize(); // all resources allocated
  app.start();      // start execution
  app.wait();       // wait until app is "done"
  app.finish();.....// do end-of-run processing like dump properties
}
```

The syntax of the OA::PValue class is described below in Class OA::PValue. Except for the **property** parameter, if there is no instance (followed by equal sign), the parameter applies to all instances. E.g.:

```
OA::PValue params[] = { PVString("model",     "rcc"),
                        PVString("selection", "filter=snr<40"),
                        PVString("property",  "filter=mode=6"),
                        PVEnd
                        };
}
```

would specify that all instances should use the **rcc** authoring model, and the **filter** instance should only use implementations (workers) whose **snr** property value was less than **40**. It would also set the **mode** property value for the **filter** instance to **6**.

Most **ocpirun** options may be set using this method, with the convention that option names with hyphens are replaced with camel-case names, e.g.:

```
ocpirun --log-level=8
```

would require:

```
PVUChar("logLevel", 8);
```

6.1.2 OA:: Application::initialize Method

This method initializes the application by allocating all necessary resources and loading, creating, initializing, configuring and connecting all workers necessary to run the application. When this method returns, the application is "ready to run". Any errors that might occur when allocating resources, loading code, instantiating/initializing workers, configuring workers or connecting workers, will have happened via exceptions before this method returns.

```
class Application {
  void initialize();
};
```

6.1.3 OA:: Application::start Method

This method starts the application by starting all the workers in the **OA::Application**. When this method returns the application is running.

```
class Application {
  void start();
};
```

Workers in the application are started using the following rough ordering rules. Note that the notion of a "source" worker is one with output ports but with no input ports that must be connected (i.e. no input ports that are *not* "optional" ports). Overall, workers that are proxies are started before workers that are not proxies (thus a proxy is always started before its slave, so that it might control how a slave is started). Also, workers that are sources are started after workers that are not sources (thus consumers are started before producers).

6.1.4 OA::Application::stop Method

This method suspends execution of the application. When the method returns, the application is no longer executing. Properties may be queried (and should not be changing) after the application is suspended. Some workers do not implement this operation (they are not suspendable), and if so an exception is thrown. When the application can be successfully stopped, it can be resumed by again using the **start** method described above.

```
class Application {
  void stop();
};
```

Workers in the application are stopped using the ordering rules described for the ***start*** method.

6.1.5 OA:: Application::wait Method

This method blocks the caller until the application is finished: when all the workers are finished or when the worker indicated by the "finished" attribute in the OAS, is finished. The single argument indicates how long to wait in microseconds. If the value is zero, the wait will not timeout. The return value is true when the timeout expired, and false when the application was "finished".

```
class Application {
  bool wait(unsigned timeout_us);
};
```

6.1.6 OA:: Application::finish Method

This method performs various functional (not cleanup) actions when the application is "finished". It should be called after **`wait`** returns, whether timeout or not. Among other things, this is required to perform the "dumpfile" action for properties, as indicated in the OAS or **`ocpirun`** options.

```
class Application {
  void finish();
};
```

6.1.7 OA:: Application::dumpProperties Method

This method prints the ordinals, names and values of all properties. With no arguments it prints all properties. Setting the **printParameters** argument to false suppresses printing of ***parameter*** (compile time) properties. Setting the **printCached** argument to false suppresses printing of properties that are not changed during execution (such as ***initial*** properties). Setting the **context** argument to non-NULL puts that string in the heading printed before properties are printed. This method is automatically called inside the **start** and **finish** methods described above, if the **dump** option is used when the application is created.

```
class Application {
  void dumpProperties(bool printParameters = true,
                      bool printCached = true,
                      const char *context = NULL) const;
};
```

6.1.8 OA::Application::getProperty Method — by Property Name

This method gets a property value by name, returning the value in text form into the **std::string** whose reference is provided. It should be used in preference to the **OA::Property** class below, when performance is not important, since although it has higher overhead internally, it is simpler to use than using **OA::Property**.

Application-level properties are simply identified by name and Instance-level properties are identified as ***<instance>.<property_name>***. The name argument can be a literal string constant (const char *) or something that can be resolved to std::string, such as:

```
"myfilter0.myprop"
inst_name + ".myprop"
inst_name + "." + prop_name
"my_app_prop"
```

If there is no property with the given name, or some other error occurs reading the property value, an exception is thrown. An exception is also thrown if the property is not readable (unless the **OA::UNREADABLE_OK** option is used — see below).

```
class Application {
  const char *getProperty(const std::string &name, std::string &val,
                          OA::AccessList &list = <none>,
                          OA::PropertyOptionList &options = <none>,
                          OA::PropertyAttributes *attrs = NULL);
  const char *getProperty(const char *name, std::string &val,
                          OA::AccessList &list = <none>,
                          OA::PropertyOptionList &options = <none>,
                          OA::PropertyAttributes *attrs = NULL);
};
```

The **list** optional argument allows navigation to parts of a complex data types. See OA::AccessList Arguments for Accessing within Complex Types below. The **options** optional argument provides options for formatting and accessing the property value. See OA::PropertyOptionList Arguments when Getting Property Values below. The

attrs optional argument allows the retrieval of various attributes of the property into a structure provided by the caller. See OA::PropertyAttributes Arguments for Getting Property Attributes.

The return value is simply the **c_str()** value of the provided string, to allow the method call to be used directly in formatted output, e.g.:

```
std::string val;
cout << "myprop: " << app.getProperty("myprop", val) << std::endl;
printf("property myprop is: %$s\n", app.getProperty("myprop", val));
```

6.1.8.1 OA::AccessList Arguments for Accessing within Complex Types

The optional **AccessList** argument provides for navigation to values within properties with complex types. **AccessList** arguments are specified in the syntax of **std::initializer_list**, which is a brace-enclosed comma-separated list. The elements of the list are either indices (for arrays or sequences), or member names (for structures). For example, if the property was an array of structures with members **a** and **b**, then:

```
// XML:  <property name='propA' type='struct' arraylength='4'>
//         <member name='a' type='float'/>
//         <member name='b' type='bool'/>
//       </property>
getProperty("propA", val, {2, "a"});
```

would access member **a** of the structure that was element 2 of the array. If the property was a sequence of floats, we would just use:

```
// XML:  <property name='propB' type='float' sequencelength='10'/>
getProperty("propB", val, {2});
```

The **std::initializer_list** feature of C++ was first implemented in GCC 4.4 (the compiler used in CentOS6) but was not entirely compliant with the language standard in that compiler version. For such older compilers a type must be applied to the **AccessList** arguments, e.g.:

```
getProperty("prop", val, OA::AccessList({2}));
```

This can be somewhat mitigated by using a variadic macro, e.g.:

```
#ifdef __NEWER_COMPILER__
#define A(...) {__VA_ARGS__}
#else
#define A(...) OA::AccessList({__VA_ARGS__})
#endif
```

Using such a macro, the code becomes:

```
getProperty("prop", val, A(2));
```

6.1.8.2 OA::PropertyOptionList Arguments when Getting Property Values

This data type is also based on **std::initializer_list**, and is a list of options in the form of enumeration constants that apply to the retrieval of the property value. The options are:

`OA::HEX` — an indication that numeric integer values should be formatted in hexadecimal rather than decimal

`OA::UNCACHED` — an indication that any caching should be ignored (values read from workers if possible, even when otherwise cached)

`OA::APPEND` — an indication that the value should be *appended* to the provided **`std::string`** value rather than setting it.

`OA::UNREADABLE_OK` — an indication that allows an otherwise unreadable property to return an empty string, and not throw an exception in that case. An example of an unreadable property is a debug property (one defined with the **`debug`** attribute set) for a worker not build in debug mode (when the worker is built with the **`ocpi_debug`** parameter set).

An example that appends the value to the incoming string, and formatted the textual value in hexadecimal, with no AccessList is:

```
getProperty("prop", val, {}, {OA::APPEND, OA::HEX});
```

6.1.8.3 OA::PropertyAttributes Argument for Getting Property Attributes

This argument allows the caller to provide a structure that will be filled in when the property value is retrieved. The members of the structure are all boolean members except the **`name`** member which is a **`std::string`**. The attributes retrieved are:

`name` — the name of the property, as a **`std::string`**

`isParameter` — it is a ***parameter***

`isinitial` — it is ***initial*** and cannot be modified after the application is started

`isWritable` — it is ***writable*** and thus may be modified while application is running

`isVolatile` — it ***volatile*** thus worker may change it during execution

`isDebug` — it is only present if the worker was built in debug mode

`isHidden` — it ***hidden*** and thus will not normally be printed in some contexts

`isWorker` — it was defined by the worker, and is not in the OCS

`isBuiltin` — it is defined by OpenCPI, and not defined by the OCS or the worker

`isPadding` — it is defined by the worker to be ***padding*** and is not accessible

`isRaw` — it is defined by the worker as ***raw*** (defined in the HDL guide)

`isCached` — the value retrieved was based on a cache and not the actual worker

`isUnreadable` — the value was not readable, set to the empty string, and would throw an exception unless the **`OA::UNREADABLE_OK`** option was set

6.1.9 OA::Application::getProperty Method — by Property Ordinal

This method gets a property value by ordinal, returning the value in string form into the **`std::string`** whose reference is provided.

Identifying properties by ordinal is useful when enumerating all properties.

If the property is not readable, or some other error occurs reading the property value, an exception is thrown.

```
class Application {
  const char *getProperty(unsigned ordinal, std::string &val,
                          OA::AccessList &list = <none>,
                          OA::PropertyOptionList &options = <none>,
                          OA::PropertyAttributes *attrs = NULL);
};
```

When accessing a property by ordinal, its name may be returned in the optional **`OA::PropertyAttributes`** structure specified by the **`attrs`** argument.

The return value is **`NULL`** if the ordinal is out of range.

This is useful to retrieve all property values (and names) without knowing their names. Thus a simple loop can retrieve all properties:

```
std::string value;
OA::PropertyAttributes attrs;
for (unsigned n = 0;
     app.getProperty(n, value, {}, {OA::UNREADABLE_OK}, &attrs);
     n++)
  std::cout << attrs.name << ":" << value << std::endl;
```

Note that by supplying the **OA::UNREADABLE_OK** option, this loop will not fail on properties that are not readable (e.g. a debug property for a worker not build in debug mode).

6.1.10 OA::Application::setProperty Method

This method sets a property value by name, taking the value in text form, which is then parsed and error checked according to the data type of the property. It should be used in preference to the **OA::Property** class below, when performance is not important, since although it has higher overhead internally, it is simpler than using **OA::Property**.

The **name** and **list** arguments are the same as in the **getProperty** method. The **value** arguments are const, and the value may also be a string literal.

If the value cannot be parsed for the appropriate type, or there is no property with the given name, or the worker itself does not accept the property setting, an exception is thrown.

```
class Application {
  void setProperty(const char *property_name, const char *value,
                   OA::AccessList &list = <none>);
  void setProperty(const std::string &property_name,
                   const std::string &value,
                   OA::AccessList &list = <none>);
};
```

6.1.11 OA::Application::getPropertyValue Method

This method gets property values in their native data type, without converting to a string form (as is done by the **get/setProperty** methods). The properties are identified by name as is done in the **getProperty** methods. This method also allows for navigation within the property's value when it is an array, sequence, or structure type.

This method is templated based on the type of *scalar* value requested:

```
class Application {
  T getPropertyValue<typename T>(const char *property_name,
                                 AccessList &list = emptyList);
  void getPropertyValue<typename T>(const char *property_name,
                                    T &value,
                                    AccessList &list = emptyList);
};
```

When the property value is retrieved, it is error-checked for a valid conversion to the explicit type and if the value cannot be represented in the explicit type, an exception will be thrown. This method can only be used to retrieve scalar values from properties

whose type is scalar, or when the **list** argument is used to navigate to a scalar value in more complex types.

Since C++ overload resolution is not available based on return type, the variants that directly return values must include the data type template parameter at the call site, e.g.:

```
float f = app.getPropertyValue<float>("prop") + 1e9;
```

The optional **AccessList** argument provides for navigation to scalar values in properties with complex types. **AccessList** arguments are specified in the syntax of **std::initializer_list**, which is a brace-enclosed comma-separated list. The elements of the list are either indices (for arrays or sequences), or member names (for structures). For example, if the property was an array of structures with members **a** and **b**, then:

```
// XML:  <property name='prop' type='struct' arraylength='4'>
//         <member name='a' type='float'/>
//         <member name='b' type='bool'/>
//       </property>
float f = getPropertyValue<float>(¨prop¨, {2, ¨a¨});
```

would access member **a** of the structure that was element 2 of the array. If the property was a sequence of floats, we would just use:

```
// XML:  <property name='prop' type='float' sequencelength='10'/>
float f = getPropertyValue<float>(¨prop¨, {2});
```

The **std::initializer_list** feature of C++ was first implemented in GCC 4.4 (the compiler used in CentOS6) but was not entirely compliant with the language standard in that compiler version. For such older compilers a type must be applied to the AccessList arguments, e.g.:

```
float f = getPropertyValue<float>(¨prop¨, AccessList({2}));
```

This can be somewhat mitigated by using a variadic macro, e.g.:

```
#ifdef __NEWER_COMPILER__
#define A(...) {__VA_ARGS__}
#else
#define A(...) OA::AccessList({__VA_ARGS__})
#endif
```

Using such a macro, the code becomes:

```
float f = getPropertyValue<float>(¨prop¨, A(2));
```

6.1.12 OA::Application::setPropertyValue Method

This method works analogous to **getPropertyValue**, but since it takes the value to set as an argument, no explicit template type argument is required.

The value to be set is error-checked for a valid conversion to the property's type and if the value cannot be represented in that type, an exception will be thrown.

If the type of the value supplied is not valid for any OpenCPI property type, then a compiler error may result since the template methods are only implemented for those valid data types.

```
class Application {
  void setPropertyValue<typename T>(const char *property_name,
                                    const T value,
                                    AccessList &list = emptyList);
};
```

Using the example above, to set the b member of element 2 of the array to 1.2:

```
app.setPropertyValue(¨prop¨, 1.2, {2, ¨b¨});
```

6.1.13 OA::Application::getPort Method

This method is used when the C++ program wants to directly connect to an external port of the application. Such a connection is external to the application as defined in the OAS (via the **`external`** attribute of an **`instance`** element, or an **`external`** child element of a **`connection`** element). This allows the C++ program to directly send and receive messages to/from the application (actually to/from some port of some instance in the application).

An optional **`OA::PValue`** list is provided to each side of the connection in order to provide optional configuration information about the connection. The producer or consumer type of the created **`OA::ExternalPort`** object is implicitly opposite from the role of the external port. E.g. if the external port of the application is an output port, then the **`OA::ExternalPort`** object acts as an input port on which to receive messages. This method returns a reference to an **`OA::ExternalPort`** object (see below) that is used by the control-application to, itself, produce or consume messages.

```
class ExternalPort;
class Application {
  ExternalPort &getPort(const char *externalName,
                        const PValue *myProperties = NULL,
                        const PValue *extProperties = NULL);
};
```

If the connection cannot be made or the **`OA::PValue`** lists are invalid, an exception is thrown. The possible **`OA::PValue`** types for these external connections are currently the same as those for connections (and ports of connections) in the OAS. Connection-level Pvalues can be specified in either list (e.g. transport or buffer size).

The following diagram shows the relationships between Application objects and the **`OA::ExternalPort`** objects and **`OA::ExternalBuffer`** objects described next.

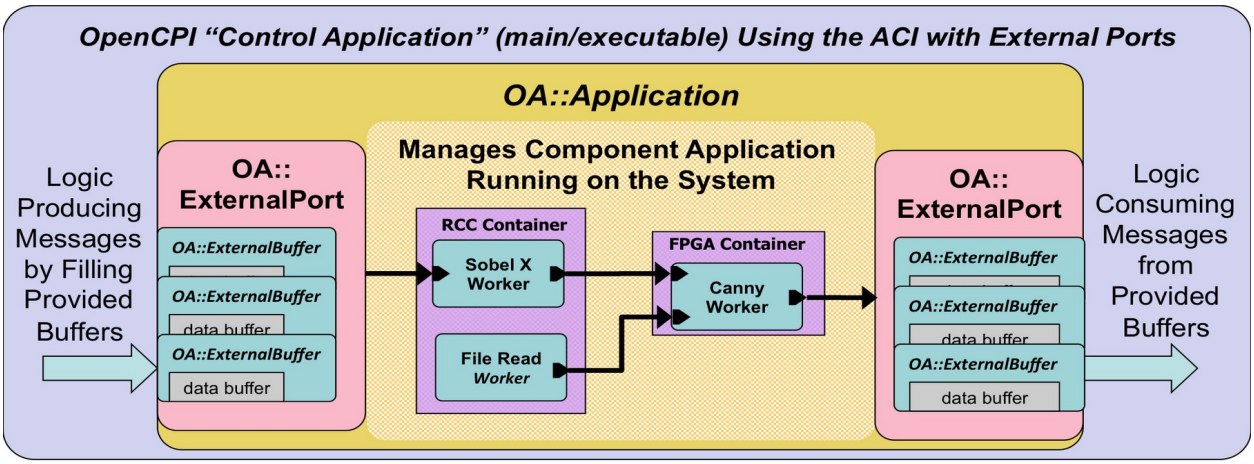

Figure 9: External Ports and Applications

6.2 Class OA::ExternalPort

This class represents a communication endpoint for the control-application itself, used to communicate with external ports of the application. These objects are owned by the **`OA::Application`** object and should *not* be deleted directly.

6.2.1 OA::ExternalPort::getInputBuffer and getOutputBuffer Methods

These methods are used to retrieve the next available buffer of the port. They return a pointer to an **`OA::ExternalBuffer`** object, or **`NULL`** if there is no buffer available. Thus it is a non-blocking I/O call. The buffer objects encapsulate the actual raw data buffers and are owned by the **`OA::ExternalPort`** objects.

For external ports giving data *to* the application, (connected to a worker *input* port inside the application), the **`getOutputBuffer`** method is used and the returned buffer object manages a data buffer to fill with a message to send into the application.

For external ports taking data *from* the application, the **`getInputBuffer`** method is used and the returned buffer object manages a data buffer containing the next message to be received by the control-application.

When the ACI application is done with the buffer, it calls the **`put`** method (for buffers obtained using **`getOutputBuffer`**) or the **`release`** method (for buffers obtained using **`getInputBuffer`**).

In addition to returning a pointer to the buffer object, these methods also return (as output arguments by reference), a pointer to its raw data buffer and the length of the message (input buffers) or the length of the buffer (output buffers). These values are attributes of the returned buffer object. The data pointer returned by reference points to memory owned by the buffer object.

There are two methods, for the two directions. The first, for taking data by getting a buffer filled with a message, also returns the metadata for the message (**`length`**, **`opCode`**, and **`endOfData`**) in separate by-reference output arguments.

```
class ExternalPort {
  // Take data from app: get buffer filled with next message
  ExternalBuffer *getInputBuffer(uint8_t *&data,
                                 uint32_t &length,
                                 uint8_t &opCode,
                                 bool &endOfData);
  // Give data to app: get buffer to fill with next message
  ExternalBuffer *getOutputBuffer(uint8_t *&data, uint32_t &length);
};
```

On input, when the **`endOfData`** is returned **`true`** (by reference), the **`data`** output argument, when not NULL, indicates that there is a message present, as well as the EOF indication. If **`data == NULL`**, there is no message, only EOF.

6.2.2 OA::ExternalPort::endOfData Method

This method indicates that no more messages will be sent to the application on this external port. It is only used when giving data. This propagates an out-of-band indication across the connection to the worker port. Note that this indication can also be made in the **`OA::ExternalBuffer::put()`** method below if the message being sent is the last message to be sent. This latter method may be more efficient, since the out-of- band indication can be carried with the message, rather than by itself.

```
class ExternalPort {
  void endOfData();
};
```

6.3 Class OA::ExternalBuffer

This class represents buffers attached to (and owned by) **`ExternalPort`** objects. They are returned (by pointer return value) from the **`OA::ExternalPort::get{Input,Output}Buffer`** methods, and recycled back to the external port via the **`put`** method (when giving data to the application) or the **`release`** method (when taking data from the application).

These objects encapsulate raw data buffers which are provided to the caller of the **`OA::ExternalPort::get{Input,Output}Buffer`** methods via an output pointer argument, by reference. Thus all buffering is managed by these objects, and pointers to the objects as well as to the internal raw data buffers are provided to callers.

6.3.1 OA::ExternalBuffer::release Method

This is the method used to discard an input buffer (output from the application) after the message in it has been processed/consumed by the control-application.

```
class ExternalBuffer {
  void release();
};
```

A simple loop that prints 10 (text) messages (without blocking or yielding) might be:

```
for (unsigned n = 0; n < 10; ++n) {
  OA::ExternalBuffer *b;
  uint8_t *data, opcode;  size_t length;  bool end;
  do
    b = port.getInputBuffer(data, length, opcode, end);
  while (!b);
  printf("%u: %.*s\n", opcode, (int)length, data);
  b->release();
  if (end) break; // in case EOF happens before 10 messages
}
```

6.3.2 OCPI::ExternalBuffer::put Method

This method is used to send an output buffer after it has been filled by the control-application. The arguments specify the metadata associated with the message:

- the length in bytes of message data
- the opcode of the message
- whether it is the last message to be sent

If it is not known whether the message is the last to be sent at the time of the call, it can be sent without that indication, and the **`endOfData()`** method can be called on the **`ExternalPort`** object at a later time.

The declaration is:

```
class ExternalBuffer {
  void put(uint32_t length,
           uint8_t opCode = 0,
           bool endOfData = false);
};
```

A simple loop that fills/sends 10 (text) messages plus EOF, without blocking or yielding might be:

```
for (unsigned n = 0; n < 10; ++n) {
  OA::ExternalBuffer *b;
  uint8_t *data;  size_t length;
  do b = port.getOutputBuffer(data, length); while (!b);
  snprintf(data, length, "Message number %n", n);
  b->put(strlen(data), 0);
}
port.endOfData();
```

6.4 Class OA::Property

This class represents a runtime accessor for a property. They are normally created with automatic storage (on the stack) and simply cache the necessary information to efficiently read or write property values. The control application that uses this class is responsible for creating and deleting the objects, although typical usage is automatic instances that are automatically deleted.

*6.4.1 **OA::Property::Property** Constructor Method*

This constructor initializes the **`OA::Property`** object such that it is specific to the application and specific to a single named property of that application.

```
class Property {
  Property(Application &app, const char *name);
};
```

The **`name`** argument specifies the property the same as the **`getProperty`** method in the application class described above. Typical usage would be:

```
{
  OA::Application app("myapp.xml");
  app.initialize();
  OA::Property freq(w, "frequency"), peak(w, "peak");
  app.start();
  freq.setFloatValue(5.4);        // set this during execution
  float p = peak.getFloatValue(); // get this during execution
  app.wait();
}
```

The **set** and **get** methods are all strictly typed. They cannot be overloaded since overloading of integral types in C++ does not prevent truncation.

This same class is used in the more detailed ACI classes described below. In particular, there is another constructor for this class based on a Worker object:

```
class Property {
  Property(Worker &worker, const char *name);
};
```

Beyond the fact that it is based on a worker rather than an application, the constructed **`OA::Property`** object is used with all the same methods.

6.4.2 OA::Property::set{Type}Value Methods

There is a **set** method for each property scalar data type. The **set** methods are strongly typed and individually named to avoid the unintended consequences of numerical type conversions of the C++ language. If the wrong **set** method is used for a property (e.g. **`setULong`** for a property whose type of **`Float`**), an exception is thrown. If the string in **`setStringValue`** is longer than the worker property's maximum string length, an exception is thrown.

```
class Property {
  void setBoolValue(bool val);
  void setCharValue(int8_t val);
  void setDoubleValue(double val);
  void setFloatValue(float val);
  void setShortValue(int16_t val);
  void setLongValue(int32_t val);
  void setUCharValue(uint8_t val);
  void setULongValue(uint32_t val);
  void setUShortValue(uint16_t val);
  void setLongLongValue(int64_t val);
  void setULongLongValue(uint64_t val);
  void setStringValue(const char *string);
};
```

6.4.3 OA::Property::get{Type}Value Methods

There is a **get** method for each property data type. The **get** methods are strongly typed and individually named to avoid the unintended consequences of numerical type conversions of the C++ language. If the wrong **get** method is used for a property (e.g. **`getULong`** for a property whose type of **`Float`**), an exception is thrown. If the string buffer in **`getStringValue`** is not long enough to hold the worker property's current string value, an exception is thrown. If there is an error accessing the worker's property value, an exception is thrown.

```
class Property {
  bool getBoolValue();
  int8_t getCharValue();
  double getDoubleValue();
  float getFloatValue();
  int16_t getShortValue();
  int32_t getLongValue();
  uint8_t getUCharValue();
  uint32_t getULongValue();
  uint16_t getUShortValue();
  int64_t getLongLongValue();
  uint64_t getULongLongValue();
  void getStringValue(char *string, unsigned length);
};
```

6.4.4 OA::Property::set{Type}SequenceValue Methods

There is a ***set sequence value*** method for each property data. The set sequence methods are strongly typed and individually named. If the wrong set sequence method is used for a property (e.g. setULongSequence for a property whose type of Float), an exception is thrown. If any of the strings in **`setStringValueSequence`** is longer than the property's maximum string length, an exception is thrown. If the number of items in the provided sequence is greater than the maximum sequence or array length of the property, an exception is thrown. If there is an error accessing the property value, an exception is thrown.

```
class Property {
  void
    setBoolSequenceValue(bool *vals, unsigned n),
    setCharSequenceValue(int8_t *vals, unsigned n),
    setDoubleSequenceValue(double *vals, unsigned n),
    setFloatSequenceValue(float *vals, unsigned n),
    setShortSequenceValue(int16_t *vals, unsigned n),
    setLongSequenceValue(int32_t *vals, unsigned n),
    setUCharSequenceValue(uint8_t *vals, unsigned n),
    setULongSequenceValue(uint32_t *vals, unsigned n),
    setUShortSequenceValue(uint16_t *vals, unsigned n),
    setLongLongSequenceValue(int64_t *vals, unsigned n),
    setULongLongSequenceValue(uint64_t *vals, unsigned n),
    setStringSequenceValue(const char **string, unsigned n);
};
```

6.4.5 OA::Property::get{Type}SequenceValue Methods

There is a ***get sequence value*** method for each scalar data type. The get sequence methods are strongly typed and individually named. If the wrong get sequence method is used for a property (e.g. **`getULongSequenceValue`** for a property whose type of **`Float`**), an exception is thrown. If there is an error accessing the worker's property value, an exception is thrown.

The first argument, **`vals`**, points to an array where the values will be placed. The second argument, **`n`**, is the space available provided by the caller. If there is not enough room in the array, an exception is thrown. The return value is the number of elements returned in the array.

```
class Property {
  unsigned
    getBoolSequenceValue(bool *vals, unsigned n),
    getCharSequenceValue(int8_t *vals, unsigned n),
    getDoubleSequenceValue(double *vals, unsigned n),
    getFloatSequenceValue(float *vals, unsigned n),
    getShortSequenceValue(int16_t *vals, unsigned n),
    getLongSequenceValue(int32_t *vals, unsigned n),
    getUCharSequenceValue(uint8_t *vals, unsigned n),
    getULongSequenceValue(uint32_t *vals, unsigned n),
    getUShortSequenceValue(uint16_t *vals, unsigned n),
    getLongLongSequenceValue(int64_t *vals, unsigned n),
    getULongLongSequenceValue(uint64_t *vals, unsigned n),
    getStringSequenceValue(const char **string, unsigned n,
                           char *buf, unsigned maxStringSpace);
};
```

For **`getStringSequenceValue`**, the first argument is an array of pointers provided by the caller, whose length is **`n`**. These pointers will point to the returned strings. The **`buf`** argument is space for the returned strings to be stored, whose length is indicated by the **`maxStringSpace`** argument. If **`maxStringSpace`** is insufficient to store all the strings (each with null termination), an exception will be thrown.

6.5 Class OA::PValue: Named and Typed Parameters

This class represents a strongly typed name/value pair, and is always used as a member of a null-terminated array of such objects. Its usage is typically to provide a pointer to an array of **OA::PValue** structures, usually statically initialized. There are derived classes (of **OA::PValue**) for each supported data type, which is the same set of types supported for component properties in the OCS. For each supported scalar data type, the name of the derived class is **OA::P<*type*>**, where ***<type>*** can be any of:

Bool, **Char**, **Double**, **Float**, **Short**, **Long**, **UChar**, **ULong**, **UShort**, **LongLong**, **ULongLong**, or **String**.

The corresponding C++ data types are:

bool, **char**, **double**, **float**, **int16_t**, **int32_t**, **uint8_t**, **uint32_t**, **uint16_t**, **int64_t**, **uint64_t**, **char ***.

Common usage for static initialization is to declare a PValue array and initialize it with typed values and terminate the array with the symbol PVEnd, which is a value with no name, e.g.:

```
PValue pvlist[] = {
  PVULong("bufferCount", 7),
  PVString("xferRole", "active"),
  PVULong("bufferSize", 1024),
  PVEnd
};
```

Note that **OA::PValue** objects are used to provide named and typed parameters to the ACI, and are in fact unrelated to component or worker properties except they share data types.

6.6 Building ACI Programs

Programs using the ACI are normally built in the context of OpenCPI projects (see Applications in Projects), in which case the compilation and link commands are provided by OpenCPI and **ocpidev**. When the ACI is used *outside* of OpenCPI projects, presumably in the context of other software libraries, executables or build systems, the OpenCPI CDK must still be present, and the **OCPI_CDK_DIR** variable must be set.

The make file fragment **ocpisetup.mk** (from the **include** subdirectory of the CDK installation) must be included to set the correct values of gnumake variables for use outside OpenCPI projects. They all have the prefix **OCPI_**. In particular:

- The include file search path must include the directory defined in **OCPI_INC_DIR**.
- The compiler must be enabled at least for the subset of C++11 that was implemented in GCC4.4. Use **-std=c++0x** or **-std=c++11** as appropriate.
- External OpenCPI symbols in the executable must be available to dynamically loaded libraries using linker options defined in **OCPI_EXPORT_DYNAMIC**.

- The link-time library search path (usually using **-L** options) must include the directory in **OCPI_LIB_DIR**.
- The OpenCPI framework libraries found in the **OCPI_API_LIBS** variable must be included in the link command, typically using:

  $(OCPI_API_LIBS:%=-locpi_%)
- The OpenCPI prerequisite libraries found in the **OCPI_PREREQUISITES_LIBS** must be included as ***static*** libraries using, e.g.:

  $(OCPI_PREREQUISITES_LIBS:%=$(OCPI_LIB_DIR)/lib%.a)

Building ACI programs for embedded systems outside of OpenCPI projects is not explicitly supported yet. An example Makefile for a simple main program built outside of OpenCPI projects is:

```
include $(OCPI_CDK_DIR)/include/ocpisetup.mk

tapp: tapp.cc
   g++ -o $@ -std=c++0x $(OCPI_EXPORT_DYNAMIC) -I$(OCPI_INC_DIR)
$< \
     -L$(OCPI_LIB_DIR) $(OCPI_API_LIBS:%=-locpi_%) \
     $(OCPI_PREREQUISITES_LIBS:%=$(OCPI_LIB_DIR)/lib%.a) \
     $(OCPI_SYSTEM_LIBS:%=-l%)

run: tapp
   OCPI_LIBRARY_PATH=$(OCPI_ROOT_DIR)/projects/core/artifacts ./$<

clean:
   rm -f tapp
```

And the simple hello world application in the file **tapp.cc** is:

```
#include <iostream>
#include "OcpiApi.hh"
int main(int argc, char **argv) {
  try {
    OCPI::API::Application
      app("<application>"
          "  <instance component='ocpi.core.hello_world'/>"
          "</application>");
    app.initialize();
    app.start();
    app.wait();
    app.finish();
  } catch (const std::string &e) {
    std::cout << e << std::endl;
    return 1;
  }
  return 0;
}
```

6.7 Using the ACI with Python

ACI programs can be written in Python, with the ACI used directly by importing the **`opencpi.aci`** module. This is only supported on development hosts, and not on any cross-compiled or embedded systems.

This module serves as the namespace for the Python ACI, much like the **`OCPI::API`** C++ namespace. Thus a typical initial python code might be:

```
import opencpi.aci as OA
app = OA.Application('myapp.xml')
...
```

When the OpenCPI environment is set up (using **`opencpi-setup.sh`**, which is typically run at login or in a new terminal window) the **`PYTHONPATH`** environment is modified to include the location where the appropriate OpenCPI python module files are located.

The same object classes and methods described for C++ are available in Python, with the following rules and exceptions.

- Strings in C++ ACI methods usually use the **`const char *`** type, which requires the corresponding python arguments to be Python strings as in the **`OA.Application`** constructor example above. Methods that return strings (e.g. **`getProperty`**), return Python string objects.
- Data buffers that are available based on a **`uint8_t *`** pointer in C++ are of Python built-in type **`memoryview`**. Writing data to a buffer requires a Python slice assignment to the provided data buffers.
- Methods that have by-reference output arguments in C++ return a tuple in Python consisting of the normal return value followed by all the by-reference output arguments described for C++.
- Methods that take a (optional) parameter(s) argument of type: **`const OA::PValue *`** take a Python dictionary instead, with the keys being strings, e.g. {**`'verbose': True`**}.
- Methods that take an optional **`OA::AccessList`** argument, use a Python list instead, e.g. **`[ 1, 'member1', 2 ]`**

For example, the **`getInputBuffer`** method for receiving data from an application has these C++ arguments:

```
ExternalBuffer *
  getInputBuffer(uint8_t *&data, size_t &length, uint8_t &opCode,
                 bool &endOfData);
```

and a typical usage in C++ would be:

```
uint8_t *data;
size_t length;
uint8_t opcode;
bool eof;
ExternalBuffer *buffer = port.getInputBuffer(data, length, opcode,
                                             eof);
```

In python this would be:

```
buffer, data, length, opcode, eof = port.getBuffer()
```

with the buffer return value being a Python **None** if the C++ return value is **NULL**.

Following are two example programs, both in C++ and in Python. The first reads from a file named hello_in, containing “Hello, World\n”, and prints out the contents using the **file_read** component.

The application XML file **hello_in.xml** is:

```
<application>
  <instance component='ocpi.core.file_read' externals='true'>
    <property name='filename' value='hello_in'/>
  </instance>
</application>
```

Here is the C++ version, using the ACI:

```
#include <iostream>
#include "OcpiApi.hh"
namespace OA = OCPI::API;
int main(int argc, const char **argv) {
  OA::Application app("hello_in.xml");
  app.initialize();
  app.start();
  OA::ExternalPort port = app.getPort("out");
  bool eof;
  do {
    OA::ExternalBuffer *buf;
    uint8_t *data;
    size_t length;
    uint8_t opcode;
    do
      buf = port.getInputBuffer(data, length, opcode, eof);
    while(!buf);
    std::cout.write((const char *)data, length);
    buf.release();
  } while (!eof);
}
```

Here is the Python version using the ACI, using the rules mentioned above:

```
import opencpi.aci as OA
app = OA.Application("hello_in.xml")
app.initialize()
app.start()
port = app.getPort('out')
eof = False
while not eof:
    buf = None
    while not buf:
        buf, data, length, op, eof = port.getInputBuffer()
    print(data.tobytes().decode(),end='')
    buf.release()
```

Notice that the returned raw data buffer, must be converted to ASCII bytes, and then converted to a Python string (decoded). Another example is where data is produced for an external port that is a worker input port. The XML file **hello_out.xml** is:

```
<application finished="file_write">
  <instance component='ocpi.core.file_write' externals='true'>
    <property name='filename' value='hello_out'/>
  </instance>
</application>
```

And the Python is:

```
import opencpi.aci as OA
app = OA.Application("hello_out.xml",
                     {'verbose': True, 'dump':True})
app.initialize()
app.setProperty('file_write.fileName','hello_out1')
app.start()
port = app.getPort('in')
print("FILE:"+app.getProperty('file_write.fileName'))
buf = None
while not buf:
    buf, data, length = port.getOutputBuffer()
msg="Hello, World\n"
data[0:len(msg)] = msg.encode()
buf.put(len(msg), 0, True) # send the message with EOF
app.wait()
app.finish()
```

This Python ACI capability is preliminary and not all ACI methods are supported.

7 Using Remote Containers: Network-Connected Processors

When an application is run, containers are found and used where workers (based on artifacts) may execute. The set of containers considered are those that are part of, or directly attached to, the local system. This includes RCC (software) containers based on the local CPU, and FPGAs attached to the system's bus or fabric such as PCI Express or the local interconnect between CPU and FPGA on Zynq SoCs.

OpenCPI's ***remote containers*** feature adds containers available in other systems on the network to the set of containers considered for execution. Remote containers are containers offered for use by these other network-accessible systems. The local system where the application is launched acts as a network client and the systems offering remote containers act as network servers.

There are several reasons for using remote containers:

- Scaling the available resources by using all the containers on a collection of network-connected systems (e.g. clusters, multiple radios)
- Heterogeneous testing, control, and experimentation allowing the client to perform application control and testing, using many disparate platforms
- Support containers which are only controllable via networks, e.g.:
  – plug-in processor cards controllable via network over backplane connectors
  – virtual machines running other OSs or licensed simulators
- Support embedded server nodes which only require a small server software configuration to run and nothing else (minimal configuration and footprint)

An OpenCPI system may offer its containers to other systems for use as remote containers by running the **`ocpiserve`** command. This command says: *make my local containers available for use by other systems acting as network clients*. The **`ocpiserve`** command runs a "container server" serving up local containers as remotely accessible containers to network clients running **`ocpirun`** or ACI-based applications. These clients are where the application is initiated and controlled (launched).

Since **`ocpiserve`** does not run, initiate or control entire OpenCPI applications, it requires fewer resources than **`ocpirun`** or ACI executables. No OpenCPI artifacts are necessary on the server system. Instead, the artifacts are downloaded, then cached, from the client to the servers, *on demand*.

The **`ocpiremote`** command, run on a network *client* or development system, can be used to provision and control a system acting as a container server. This command eliminates all requirements to manually:

- copy any files to the server system
- run setup scripts on the server system
- have any file system network mounting (NFS) between client and server systems.
- install or run the **`ocpiserve`** command on the server system

See the man page for **`ocpiremote`** at Manual Page for ocpiremote. The **`ocpiserve`** command is fully described in its man page at: Manual Page for ocpiserve.

7.1 Enabling Remote/Embedded Systems to be Container Servers

7.1.1 Using `ocpiremote` to Enable and Manage Container Servers

This command is used separate from application execution or container server discovery to conveniently manage a remote container server from a client system. It loads the OpenCPI server software onto the remote system, and starts the server. Other than specifying the remote system's IP address (and optionally, the port, and SSH user name and password), no other action is necessary to use the remote system.

The **`ocpiremote`** command can use the same environment variables as **`ocpirun`** and the ACI to know the addresses of available servers. See the man page for **`ocpiremote`** at Manual Page for ocpiremote.

7.1.2 Using `ocpiserve` to Offer Containers to Remote Clients

When not using **`ocpiremote`**, the **`ocpiserve`** command is run manually on a server system to offer its local containers to remote clients. When it starts, **`ocpiserve`** discovers all local network interfaces and prepares to be contacted by clients for all of them. When given the **`-d`** (**`--discoverable`**) option, it prepares to receive multicast queries from clients on all network interfaces. It is a fully multi-homed server, using all network interfaces for normal usage (via TCP) and discovery (via multicast UDP).

Running the **`ocpiserve`** command manually on the server system is not necessary when you use the **`ocpiremote`** command.

The **`ocpiserve`** command does not need or use the **`OCPI_LIBRARY_PATH`** environment variable since the client-server protocol automatically downloads (from client to server) each artifact needed by the application using remote containers. This automatic artifact downloading operates as a cache. When any client requests execution using any of **`ocpiserve`**'s containers, they also indicate which artifacts should be used in each container. If these artifacts have been previously downloaded, they are reused. If not, they are downloaded from client to server.

The **`ocpiserve`** command maintains this cache in a directory called **`artifacts`** (unless overridden by an option). The artifact cache is normally maintained after **`ocpiserve`** exits (e.g. via control-C). An option indicates whether it should be removed when **`ocpiserve`** exits.

Complete information about **`ocpiserve`** is found at: Manual Page for ocpiserve.

7.1.3 Server Side Remote Container Setup Requirements

On the server side, the **`ocpiserve`** executable must be present and there should be enough disk space to hold the artifact cache. The **`ocpiremote`** command puts **`ocpiserve`** where it needed to be, or it can be part of a complete SD-card based OpenCPI installation.

If it is desirable to allow the server to be discovered via multicast, its local firewall must allow inbound UDP multicast packets from the local network or at least selected clients.

To open the firewall to local network traffic the following command can be used (on RockyLinux when it is a server):

```
sudo firewall-cmd --zone=trusted --change-interface=ens160
```

Of course more fine-grained permissions may be appropriate.

7.1.4 Example Setup Script for Enabling a Server

Here is a sequence of steps done on a client system to setup a server with the **ocpiremote** tool and environment variables. The IP/TCP port being used at the server is **12345**, and the explicitly specified network address of the server is **10.0.0.86**.

```
# set the server address and port for both ocpiremote and ocpirun
export OCPI_SERVER_ADDRESSES=10.0.0.86:12345
ocpiremote --password root\
           reload \
           --rcc-platform xilinx24_1_aarch32\
           --hdl-platform zed
ocpiremote --password root\
           start -b
```

At this point the server is ready to be used by a client. An example of running an application using a server set up this way is below at Example Script for Using a Container Server.

7.2 Using Remote Containers from Client/Development Systems

When applications run, if remote containers are available, some of the components in the application may be deployed onto those remote containers (provided by **ocpiserve** running on remote systems). The remote containers are added to the set of places where components may execute. Use of remote containers is configured, initiated and controlled from a client system (sometimes the development system).

Using remote containers is transparent: applications use the additional containers when considering where to run the component instances in the application.

Whether an application is started from within a C++ main program using the ACI, or using the **ocpirun** command, it uses the remote containers it knows about, and finds out about available remote containers by contacting systems running container servers, which run the **ocpiserve** command.

Container servers are found by two methods: locally specified and discovered.

7.2.1 Locally Specified Container Servers

Locally specifying servers avoids any necessity of discovery or multicast, but requires a manual method to obtain the IP address of the server system and provide that address locally on the client system.

When the addresses of server systems are known in advance, they are specified in one of three ways:

- ***environment variables*** containing addresses or specifying filenames where addresses are found
- ***command line options*** to the **ocpirun** command to specify server addresses.
- ***ACI functions*** that can specify server addresses

The **OCPI_SERVER_ADDRESSES** environment variable provides a list of the IP address and port number of available servers. The list is comma-separated, and each address is followed by a port number after a colon. For example:

```
export OCPI_SERVER_ADDRESSES=10.1.2.3:12345,192.168.1.2:54332
```

specifies two servers, with different IP addresses and port numbers. These servers will be contacted when an application is started, whether via **ocpirun** or the ACI. A third optional field (after IP address and port) can contain **ocpiremote** options that should be used for all **ocpiremote** commands. I.e. if the password of the remote system was **admin**, then specifying the server address as:

```
export OCPI_SERVER_ADDRESSES="10.1.2.3:12345:-p admin"
```

would mean the **-p admin** option would be inserted for all **ocpiremote** commands.

A similar environment variable, **OCPI_SERVER_ADDRESSES_FILE** can refer to a file containing the same type of server addresses, one per line.

The **-S** (**--server**) option to **ocpirun** may also specify the IP address (and port) of a server system, and that option can be supplied multiple times.

Finally there is an ACI function that can be used to specify a server to access remote containers. The function:

```
void OA::useServer(const char *server);
```

is the ACI equivalent to the **ocpirun -S** option.

The **OCPI_SERVER_ADDRESSES_FILE** variable can be used when the client and server both have the same file system mounted since the **ocpiserve** command can be told to put its address(es) into a named file. This is a tradeoff between the overhead and complexity of having a common mounted file system (e.g. when the server has an NFS mount to the client or vice versa), vs. the need to know the server's IP address, which could be dynamically set using DHCP.

When not using server discovery via multicast (see below), and there is a common NFS mounted directory, using **OCPI_SERVER_ADDRESSES_FILE** is a convenient way to avoid knowing or typing or copy-pasting IP addresses. If you cannot use discovery and do not want or have a common mounted file system, using **OCPI_SERVER_ADDRESSES** on the client side (or the **--server** option to **ocpirun**) may be best. The **ocpiserve -v** option causes **ocpiserve** to print out its IP addresses.

7.2.2 Discovered Container Servers

When not locally specified, remote containers may be discovered using a multicast technique to find the systems running **ocpiserve**, and retrieve from them the list of available containers on each such system.

The **OCPI_ENABLE_REMOTE_DISCOVERY** variable can be set to 1 to enable the multicast discovery of servers running **ocpiserve**.

The **ocpirun** command has an option for this same purpose. The **-R** (**--remote**) option tells **ocpirun** to "discover" and use remote containers via multicast discovery. The command:

```
ocpirun -R -C
```

runs **ocpirun** simply to list available containers, including any remote ones discovered via multicast.

Similarly, there is a **-d** (**--discoverable**) option to **ocpiserve** that says: make **ocpiserve** discoverable via multicast. This enables clients using the **-R** option to find the server. Without the **-d** (**--discoverable**) option, the **ocpiserve** command must be contacted using its explicit IP address (e.g. using the **--server** option to **ocpirun**).

The ACI function:

```
void OA::enableServerDiscovery();
```

is the ACI equivalent to the **ocpirun -R** option, and enables remote container discovery. The ACI function:

```
bool OA::isServerSupportAvailable();
```

is a function used to determine whether remote container support is enabled in the current environment. This function can be called early in the ACI program to decide whether the other two (**OA::useServer** or **OA::enableServerDiscovery**) should be called at all. It essentially indicates whether the remote container plug-in is loaded via being specified in the **system.xml** file on the client system.

7.2.3 Client Side Remote Container Setup Requirements

Client systems must be enabled for remote containers. The first requirement is that the remote container OpenCPI plug-in must be loaded, which is specified in the **system.xml** file for the client system. E.g. the **load** attribute of the remote container plug-in must be set to one:

```
<opencpi>
  <container>
   <rcc load='1'/>
   <remote load='1'/>
   ...
  </container>
  ...
</opencpi>
```

This setting is normally already set. The second requirement, which is optional, is to enable clients to automatically discover servers via multicast UDP. This requirement is to provide a proper multicast route to the local network interface on the client system. This is normally accomplished by a command such as:

```
route -n add -net 224.0.0.0 netmask 240.0.0.0 dev enp2s0
```

This assumes the normal default network interface for the client is **enp2s0**.

If multicast discovery is not being used (i.e. explicit server addresses are specified), this route setup requirement is unnecessary.

The client system also must be contactable (as a IP/TCP server) when data stream connections (from worker to worker) are made from the server back to the client. This may require that the firewall on the client system allow connections back from the server. On RockyLinux this means a command such as this is necessary:

```
sudo firewall-cmd --zone=trusted --change-interface=ens160
```

The **OCPI_SOCKET_INTERFACE** variable is needed when the host/client system running **ocpirun** or ACI or unit test, has multiple active network interfaces. When this is the case, it must be set to the network interface name that should be used for contacting container servers running **ocpiserve**. I.e. access to remote container servers is currently limited to using a single network interface. The names of network interfaces are obtained using the standard Linux or MacOS **ifconfig** or **ip addr** command. This variable should be set to the interface with the same network (IP address anded with the netmask) as the servers being used. Using this variable has no affect on firewall configurations. The **ocpihdl ethers** command shows this information in a different format.

In summary, when servers are set up (see previous section), they are made usable by:

- enabling the client systems for discovering servers or specifying their addresses
- enabling data connections from server back to the client system
- running ocpirun or an ACI program that then uses the remove containers

7.2.4 Example Script for Using a Container Server

Here is a sequence of steps done on a client system to run an application using remote containers setup previously. The network interface on the client system that is used to reach the server system is **`eth1`**, the IP/TCP port being used at the server is **`12345`**, and the explicitly specified network address of the server is **`10.0.0.86`**.

```
# set the server address for both ocpiremote and ocpirun
export OCPI_SERVER_ADDRESSES=10.0.0.86:12345
PDIR=$OCPI_ROOT_DIR/projects
export OCPI_LIBRARY_PATH=$PDIR/assets/artifacts:$PDIR/core/artifacts
# needed if there is more than one interface with an IP address
export OCPI_SOCKET_INTERFACE=eth1
cd $PDIR/assets/applications
ocpirun -C # output should confirm that server can be reached
ocpirun -v -m bias=hdl testbias.xml
```

8 Utility Components for Applications

There are several built-in components that application developers use frequently. These are listed in this section. Their availability on a given platform depends on whether they have been built for that platform and whether artifacts are available in the path specified by the **OCPI_LIBRARY_PATH** environment variable. That variable is set automatically when the **ocpidev** tool is used to run applications, but in a runtime-only environment without **ocpidev**, **OCPI_LIBRARY_PATH** must be set.

All of these utility components are in the **ocpi.core** package, so using them usually involves specifying the instance's **component** attribute as **ocpi.core.*<component>***. If most of the components in the application are in this package it may be more convenient to simply set the **package** attribute for the whole application to provide a default prefix.

8.1 file_read Component that Reads Data or Messages from a File

The **file_read** component (**ocpi.core.file_read**) injects file-based data into an application. It is normally used by specifying an instance of the **file_read** component, and connecting its output port to an input port of the component which will process the data first. The name of the file to be read is specified in the **fileName** property.

This component has one output port whose name is **out**, which carries the messages conveying data read from the file. There is no protocol associated with the port: it is agnostic as to the protocol of the file data and the connected input port.

For details about how to use the **file_read** component, refer to the data sheet in the **ocpi.core** project.

8.2 file_write Component that Writes Data or Messages to a File

The **`file_write`** component (**`ocpi.core.file_write`**) writes application data to a file. It is normally used by specifying an instance of the **`file_write`** component, and connecting its input port to an output port of the component producing the data. The name of the file to be written is specified in the **`fileName`** property.

This component has one input port whose name is **`in`**, which carries the messages to be written to the file. There is no protocol associated with the file, enabling it to be agnostic as to the protocol of the file data and the connected output port.

For details about how to use the **`file_write`** component, refer to the data sheet in the **`ocpi.core`** project.

8.3 The Digital Radio Controller (DRC) Component to Use Radio Hardware

The DRC component is used when an application needs to use radio hardware in the system, for the purpose of controlling it and streaming sample data to and from it. The "digital radio" functionality in a system usually has antennas for transmitting and receiving RF signals, and ***channels*** which convert the RF signals to and from base-band digital samples that are produced and consumed by the application.

Applications that use the DRC component are usually called SDR (software-defined radio) applications.

SDR application developers use the DRC component by instantiating it in their applications, and setting properties to control and configure radio channel hardware (and possible some extra hidden components behind the scenes). Applications also make connections to the DRC's RX (receive) and TX (transmit) data ports to produce or consume baseband digital samples, usually using the **`complex_short_timed_sample`** protocol.

The following diagram shows how a typical implementation (worker) of a DRC works. The DRC usually has "slave workers" behind the scenes, controlling radio hardware. These slaves are configured by the DRC to implement channel processing between RF ports and application ports. Channel processing is accomplished by some hardware-specific combination of actual hardware (e.g. RF filters, digital sampling) and some signal processing (e.g. digital down conversion).

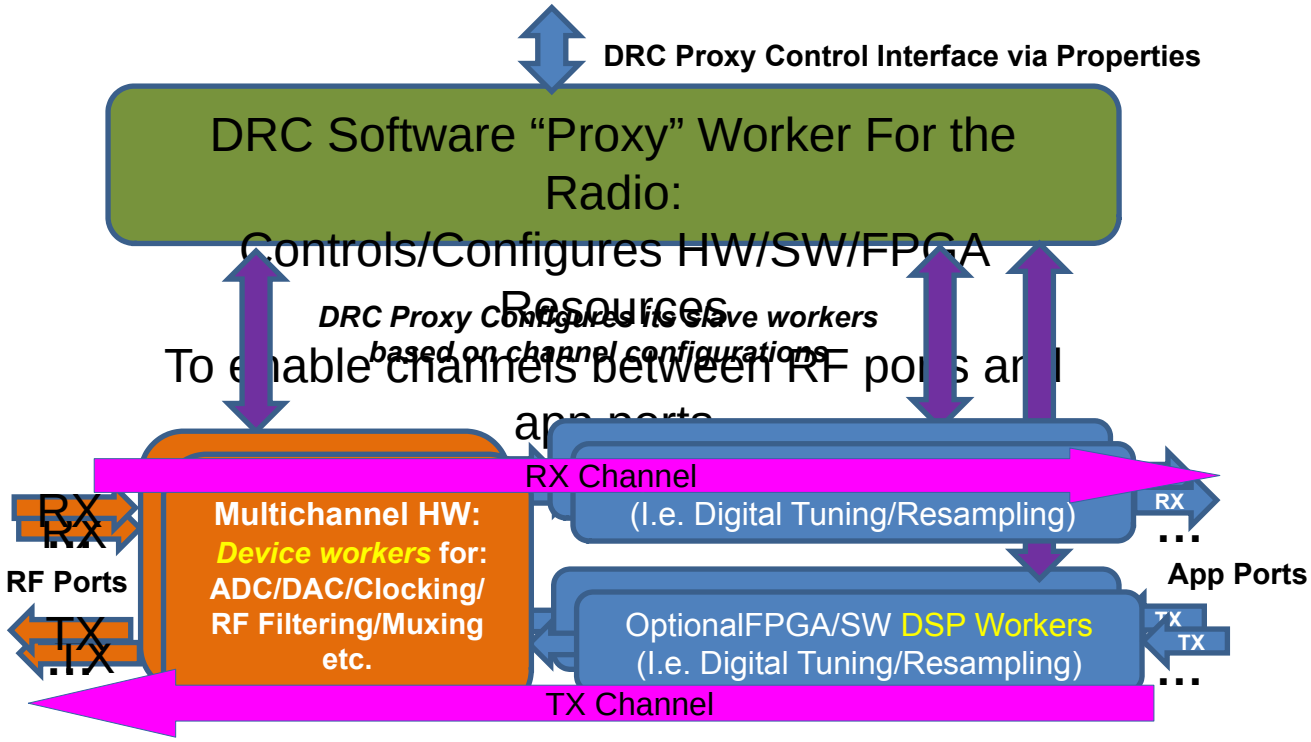

Figure 2: Digital Radio Controller Concept

8.3.1 DRC Properties

Applications may use DRC components *statically*, where the configuration properties are set only the application's XML, or *dynamically* where an ACI program changes and accesses them during execution. In either case, the primary usage of properties is to configure channels that perform processing between RF antennas or connectors on one side (RF ports) and baseband digital samples data ports (application ports) on the other. In the description below we use the term ***sequence*** consistent with how OpenCPI property data types are defined: it is a variable length set of typed values.

Three concepts are used when configuring channels:

An ***RF port*** is a source or destination of RF signals, usually with a physical connector or antenna with a label, on the outside of the radio hardware.

An ***Application port*** is a source or destination for digital samples of a baseband signal, which communicates to or from the rest of the application.

A ***Channel*** is a virtual processing resource that conveys and transforms information between RF ports and application ports. Channels are *configured* as to their endpoints (RF ports and application ports) as well as the typical radio-related parameters such as sampling rate, bandwidth, and tuning frequency. Channels are virtual in that a DRC may use a combination of various hardware and firmware resources to perform the necessary processing and thus *implement the channel*.

A ***Channel Configuration*** is a set of parameters and identified endpoints (RF and app ports) specifying an operating mode of a channel.

A ***DRC Configuration*** is a *sequence of channel configurations* that are specified, enabled (started) or disabled *as a group*.

The properties of a DRC component are:

- **`configurations`** — a sequence of DRC configurations that may be activated.
- **`prepare`**/**`start`**/**`stop`**/**`release`** — used to control individual configurations
- **`status`** — provides for reading back the current status of configurations
- **`constraints`** — provides the channel and tuning constraints of the radio.

There are additional properties that allow an application to query for radio attributes like the labels on connectors, or the OpenCPI platform it is running on. These can be used to select a particular radio in environments where there are more than one available.

*8.3.1.1 The **Configurations** Property Specifies a Set of DRC Configurations*

This property is writable both when an application starts and also during runtime. Its value is a ***sequence of** DRC **configurations***, each of which configures some *channels*. Writing this property has no side effect: activation is accomplished separately. Each configuration in the sequence is a structure with these members:

- **`Description`** — an optional string description of the configuration (perhaps describing the mode of the radio when operating under this configuration).

- **`Recoverable`** — a boolean indicating whether errors encountered when enabling the configuration should result in a recoverable error (i.e. ***not*** creating application exceptions). Useful in some ACI applications, but default is **`false`**. When recoverable, a configuration's status can be queried using the **`status`** property.
- **`Channels`** — a sequence of channel configurations that will be enabled and disabled ***together*** when this DRC configuration is enabled/disabled.

A typical static DRC application would have one configuration defined directly in its XML (in the OAS); it would not need a description or be recoverable. It would simply be enabled when the application is started, and the sample data would flow to and from the application ports. An example of such a static configuration is as follows:

```
<Application>
  <Instance component='drc'>
    <property name='configurations'
              value='{channels {
                       {rx true,
                        tuning_freq_mhz 2450,
                        bandwidth_3db_mhz 0.24,
                        sampling_rate_Msps 0.25,
                        samples_are_complex true,
                        gain_db -25,
                        tolerance_tuning_freq_mhz 0.01,
                        tolerance_sampling_rate_msps 0.01,
                        tolerance_gain_db 1},
                       {rx false,
                        tuning_freq_mhz 2450,
                        bandwidth_3db_mhz 0.24,
                        sampling_rate_Msps 0.25,
                        samples_are_complex true,
                        gain_db -25,
                        tolerance_tuning_freq_mhz 0.01,
                        tolerance_sampling_rate_msps 0.01,
                        tolerance_gain_db 1}}}'/>
    <property name='start' value='0'/>
  </Instance>
  . . . other component instances in the app. . .
</Application>
```

With the ACI, all configurations can be written at once, individual configurations can be written, individual channels within a configuration may be written, or even individual attributes of the channel may be written. Two examples of writing the **`gain_db`** to **`-23`** of channel 1 in configuration 2, would be:

```
OA::Application app(xml);
app.setProperty("drc", "configurations", "-23",
                {2, "channels", 1, "gain_db"});
OA::Property configs(app, "drc.configurations");
configs.setValue(-23, {2, "channels", 1, "gain_db"});
```

An example of setting one channel configuration (say #1) within DRC configuration 0, using a text value, would be:

```
configs.setProperty("rx false,"
                    "tuning_freq_mhz 2450,"
                    "bandwidth_3db_mhz 0.24,"
                    "sampling_rate_Msps 0.25,"
                    "samples_are_complex true,"
                    "gain_db -25,"
                    "tolerance_tuning_freq_mhz 0.01,"
                    "tolerance_sampling_rate_msps 0.01,"
                    "tolerance_gain_db 1",
                    { 0, "channels", 1});
```

In the above example, commas between members are inside the quotes (in red).

Configurations are not volatile. Their value remains whatever the application last set them to. The **`status`** property described below can be read to determine the current state of a configuration (and its channels). The application can change a configuration during execution, since it is only looked at (as a snapshot) when the application requests that a configuration should be ***prepared*** or ***started***.

8.3.1.2 The `Channels` Structure Member of DRC Configurations

A DRC configuration contains a sequence of channel configurations to be controlled as a group. Each channel in the group is specified and configured via structure members defined in the following table. When specifying a structure in text form, any member not mentioned takes on its default value. All members have a default value or zero, false, or the empty string, depending on their type. Any channel structure member can be accessed as a scalar binary value, using the **`setValue`** method (example above).

Table 3: Channel Configuration Structure Members

Name	Type	Description
`description`	String	String describing the channel, optional
`rx`	Bool	Whether channel is RX (RF port => App port) or TX (App port => RF port)
`tuning_freq_MHz`	Double	
`bandwidth_3db_MHz`	Double	
`sampling_rate_Msps`	Double	
`samples_are_complex`	Bool	Samples are complex (I/Q) or real.
`gain_db`	Double	
`gain_mode`	String	
`tolerance_tuning_freq_MHz`	Double	
`tolerance_bandwidth_3db_MHz`	Double	
`tolerance_sampling_rate_MSPS`	Double	
`tolerance_gain_db`	Double	
`rf_port_num`	UChar	Ordinal of RF port among RX or TX RF ports (depending on setting of **`rx`**)
`rf_port_name`	String	Radio-specific name of the RF port. Using this is not portable across radios.
`app_port_num`	UChar	Ordinal of app port among the possible RX or TX App ports (**`rx`** says which)

As a default, when **`rf_port_num`** and **`app_port_num`** are left at their default zero values, the first defined RF and app ports are used. For an RX channel (**`rx == true`**), this would be the first RF port that is defined to be capable of reception, and the app port whose name is **`rx`**, with an index of zero. For an TX channel (**`rx == false`**), this would be the first RF port that is defined to be capable of transmission, and the app port whose name is **`tx`**, with an index of zero. See the section on application ports below.

8.3.1.3 Properties for Controlling DRC Configurations

Each DRC configuration has a lifecycle similar (but not identical) to components in an OpenCPI application. The lifecycle state machine for each DRC configuration is shown in the following diagram:

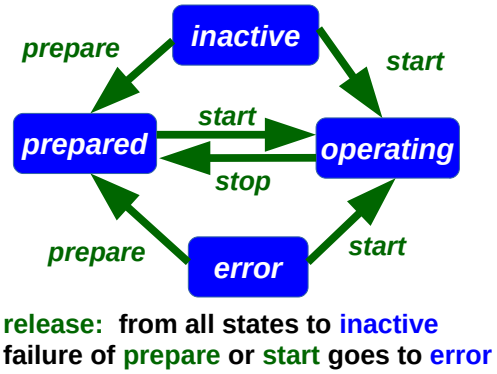

Figure 3: DRC Configuration Lifecycle

DRC configurations begin in the ***inactive*** state, where they can be modified by the application without any action being taken by the DRC. The ***prepare*** operation performs validation and setup of the channels, without making them operational, and without interfering with any currently operational configuration/channels. A successful ***prepare*** operation achieves the ***prepared*** state of the DRC configuration. From there, the ***start*** operation makes the configuration operational, in the ***operating*** state. Using the ***start*** operation in the ***inactive*** state automatically *prepares* the configuration.

As shown in the state machine diagram above, the state of a configuration is changed using the following four operations, which are performed by setting these 4 properties, whose name is the name of the operation.

- ***prepare*** — validate all the channel configurations, and do any additional work to be able to start them quickly as a group. Enter the ***prepared*** state.
- ***start*** — transition to the ***operating*** state, with all channels in the configuration configured and operating. If issued in the inactive state, this operation also implies the ***prepare*** operation.
- ***stop*** — revert to the ***prepared*** state, where the channel configurations are valid and ready to be (re)started.
- ***release*** — revert to the ***inactive*** state where no assumptions are made about the validity of the configuration, and any preparation resources are released.

All four properties are written with the value being the index of the DRC configuration to be acted upon. Thus starting the first DRC configuration involves writing a zero to the **start** property. In the XML example above, there is the setting:

```
<property name='start' value='0'/>
```

The above starts configuration 0, when the overall application is started. This operation, when defined in the XML, is deferred until the application is started. With an ACI program, dynamic controls are possible. For example, to disable configuration 1 and enable configuration 0, would be (assuming the **OA** namespace is **OCPI::API**):

```
OA::Application app(xml_filename_or_string_with_drc_instance);
app.initialize();
app.start();
OA::Property start(app, "drc", "start"), stop(app, "drc", "stop");
...
stop.setValue(1);
start.setValue(0);
```

8.3.1.4 The Status Property for Reading the Actual State of DRC Configurations

This volatile property is used to find out the exact state of the DRC configurations. Its value is similar to the **configurations** property in that it is a sequence of structures, one per DRC configuration, and within each each of those, is a sequence of structures for the status of each channel in the configuration.

While the **configurations** property is what is being requested, the **status** property is what has been achieved. For each DRC configuration (structure) in the **configurations** property (sequence), there is a structure in the **status** property (sequence), with

- a **state** structure member (an enumeration value for the state: **inactive**, **prepared**, **operating**, **error**)
- an **error** structure member (a string value describing why the ***error*** state was entered), and
- a **channels** structure member which is itself a sequence of structures with members describing the *actual* values of each channel when the DRC configuration is in the **prepared** or **operating** states.

Upon a successful **prepare** or **start** operation, these values should be within the requested tolerance specified in the **configurations** property for that channel.

Table 4: Channel Status Structure Members

Name	Type
tuning_freq_MHz	Double
bandwidth_3db_MHz	Double
sampling_rate_Msps	Double
samples_are_complex	Bool
gain_db	Double
rf_port_name	String

8.3.1.5 Properties for Describing a DRC's Static Attributes

The **constraints** property is a readable set of radio capabilities that the application may use to know what the radio is capable of, and based on that, the application can either produce useful error messages, or possibly adjust the configurations being requested based on those constraints. Also, in an environment with multiple radios, these attributes may be used to assign different tasks to different radios based on what they are capable of.

The value of this property is a sequence of constraint structures describing the radio's capabilities when using each of its RF ports for a particular direction. The members of each constraint structure are:

Table 5: Channel Status Structure Members

Name	Type
rf_port_name	String
rx	Boolean mode of using the RF port
tuning_freq_MHz	Sequence of min/max double values
bandwidth_3dB_MHz	Sequence of min/max double values
sampling_rate_Msps	Sequence of min/max double values
samples_type	Boolean “real” and “complex” members

The members that are sequences of min/max values are sequences of ranges representing the feasible ranges of values supported by the radio when using the given RF port in the given direction. The ranges can specify multiple, different, non-overlapping feasible ranges for the attribute in question.

8.3.1.6 The RX_RF_PORTS and TX_RF_PORTS Properties for RF Port Names

These two properties are both sequences of strings that are the names of the RF ports on the radio which are capable of RX or TX respectively. An RF port name can be in both lists if the radio can be configured to receive and/or transmit from that one RF port. The **rf_port_num** members of the channel configuration structure are used to index into these sequences. The first in each list is the default (index 0).

8.3.1.7 The MAX_RX_CHANNELS and MAX_TX_CHANNELS Properties

These allow the application to know the maximum number of simultaneously enabled (operating) channels that are possible in each direction.

9 Preparing HDL Assemblies for Use by Applications

Developing HDL component implementations (workers) for FPGAs is out of scope for this application development guide. That process is fully described in the ***OpenCPI HDL Development Guide***. Utilizing FPGAs in OpenCPI requires that component libraries, with built/compiled HDL/FPGA workers, be supplied for applications to use FPGAs.

HDL assemblies are the way compiled HDL workers (in built component libraries) are transformed into the artifacts necessary to execute applications that use workers executing on FPGAs. The steps to using FPGAs with OpenCPI are:

1. HDL workers are written in an HDL (hardware description language), typically VHDL.
2. HDL workers are built/compiled for a specific type of FPGA (e.g. Xilinx Zynq or Xilinx ZynqUltrascale)
3. HDL assemblies are defined in simple XML files as a set of connected HDL workers that can act as a proper subset of an application.
4. HDL assemblies are converted into ready-to-execute artifacts by a build process that incorporates the built/compiled HDL workers into a bitstream file targeting a particular FPGA platform.

Steps 1 and 2 are performed by HDL component developers who create libraries of HDL workers compiled for a variety of targeted FPGA devices.

Steps 3 and 4 ***do not*** *require* ***VHDL coding*** *or* ***specific knowledge of*** *or interaction with* ***FPGA tools,*** but the FPGA development tools (as well as the CDK) ***are*** required to be installed***.***

Step 4 is more complex when the application is accessing I/O devices directly attached to the FPGA, but still requires no VHDL coding nor vendor tools knowledge.

Thus HDL assemblies are in a middle ground between HDL worker development and application execution. Once artifacts are produced from HDL assemblies, neither the CDK nor the FPGA build tools are required. A set of artifacts based on HDL assemblies and built for some HDL platforms, acts as a runtime library for using FPGAs to support executing applications.

Except under unusual conditions (e.g. when the HDL assembly does not fit into the targeted FPGA device), building FPGA artifacts using HDL assemblies can be considered part of application development, and not component development.

Creating HDL assemblies in projects is described in the ***HDL Assemblies in Projects*** section below. Building HDL assemblies outside of projects is not explicitly supported.

The section ***HDL Assemblies for Creating Bitstreams***, in the ***OpenCPI HDL Development Guide***, describe steps 3 and 4 in detail.

10 Developing Applications in OpenCPI Projects

Applications are typically XML files. They may also be C++ or Python main programs using the ACI. All types of applications rely on the existence of existing artifacts found via **`OCPI_LIBRARY_PATH`**.

Applications may also be created as part of an OpenCPI ***project*** containing other OpenCPI ***assets*** such as components, workers, primitive libraries, etc.

In OpenCPI a project represents a work area in which a variety of assets are created and developed. Projects can contain all types of OpenCPI assets that are described in this document or in others. A project can contain:

- Component libraries with specs and workers (described in the ***Component Development Guide***)
- Applications whether simple XML or using the ACI (described here).
- HDL primitives and assemblies (described in the ***HDL Development Guide***)
- HDL devices, cards, slots (described in the ***HDL Development Guide***)
- Platform support assets (described in the ***Platform Development Guide***)

A project uses a standardized directory structure that holds the various OpenCPI assets in source code form, along with other files that describe how they are built. The project structure provides a means to co-develop a collection of assets which may have a logical relationship or be created for a specific overall application. Projects may depend on other projects when assets from other projects are needed.

All the assets in a project have a package ID and they all normally share a project-level package prefix.

When an OpenCPI application is developed in a project, an IDE or the **`ocpidev`** command line tool is used to manage the project and the assets inside it. Even if the project contains *only* applications, there are advantages to putting applications into OpenCPI projects, especially during development and especially for ACI-based applications. The **`ocpidev`** tool is described next. Discussing any IDE for this purpose is out of scope for this document.

Projects are for development. When an OpenCPI application is deployed on a runtime-only non-development system, projects are not used. Deploying applications and the various files they depend on (e.g. OAS files, ACI executables, artifacts, **`ocpirun`**), is described in the Deploying Applications in a Runtime Environment section below.

10.1 The `ocpidev` Tool as Used for OpenCPI Applications in Projects

The **`ocpidev`** command is fully described and documented in the ocpidev man page, but for pure application users (that are not developing other asset types), the small necessary subset of this tool's functionality is described here. Projects are created with the command:

```
% ocpidev [options] create project <name>
```

This creates a project in a new directory called ***`<name>`***, which must be a name without slashes. The project directory is created under the current working directory where **`ocpidev`** is executed. The **`-d <directory>`** option can be used to create the project's directory under a different directory. The options available during project creation are:

*Table 6: Options for **`ocpidev`** when Creating Projects*

Option	Value ?	Default	Description
`-v`	no		Be verbose, describing what is happening in more detail.
`-d`	yes	`.`	Specify the directory in which this command should be performed, analogous to the **`-C`** option in the POSIX **`make`** command.
`-D`	yes		Specify a project (using its package-ID) that this project depends on.
`-K`	yes	`local`	Specify the package ID when creating a project.

The **`-D`** option is useful to specify other projects that the assets in this project depend on, such as projects that may contain component libraries. The **`-K`** option is only needed when the project will be globally published and requires a globally unique name.

Projects are deleted using the command:

```
% ocpidev [options] delete project <name>
```

When using projects only for applications, only two types of assets are created in the project: applications and HDL assemblies.

10.2 Applications in Projects

Applications in projects live in the **applications/** subdirectory of the project and are either XML applications, based on an OAS XML file, or ACI-based C++ programs. XML applications can simply be OAS files in the **applications/** subdirectory, or be in a directory of their own, also under **applications/**. ACI-based applications are always in their own directory. Applications are created in projects by executing this command in a project's directory or in its **applications/** subdirectory:

```
% ocpidev [options] create application <name>
```

This command is used to create any of 3 types of application:

- A single OAS XML file, **<name>.xml**, in the **applications/** directory (using the **--xml-app** option)
- An application in its own directory under **applications/**, with the OAS XML file created there (using the **--xml-dir-app** option)
- An ACI application in its own directory under **applications/**, with the OAS XML file created there, *and* a C++ source file **<name>.cc**, also created there.

*Table 7: Options for **ocpidev** when Creating Applications*

Option	Description
--xml-app, -X	The application will be a simple XML OAS file
--xml-dir-app, -x	The application will be an XML application file in its own directory

Placing an application in its own directory allows other related files (e.g. test data, or other source files) to be in the same place.

The **applications.xml** file in the top level **applications/** directory can specify some options that apply to all the applications. If only a subset of the applications should be built or executed, or if they must be built or executed in a particular order, the **Applications** attribute in that file may contain a list of the applications to be used (built or run) and the order in which they are built and/or used, e.g.:

```
<applications Applications='myapp1 myapp3'/>
```

This would only build and run **myapp1** and **myapp3**, even if **myapp2** was also present.

Even when an application is excluded from the **Applications** attribute, it is still possible to manually enter its directory and build or run there. To delete applications, the following command is used.

```
% ocpidev [options] delete application <name>
```

When the **ocpidev build** or **ocpidev run** command is issued in the **applications/** directory, all applications are built or run, subject to the **Applications** attribute if set in the **applications.xml** file.

10.2.1 Building Applications in Projects

Building XML-only applications (not ACI-based applications) does nothing. ACI-based applications are built in their own directories using the **ocpidev** command. From a project's top level directory or its **applications/** subdirectory, a specific application can be built using the command:

```
% ocpidev [options] build application <name>
```

To build all applications in a project, you can issue the following command either in the project's directory or in its **applications/** subdirectory:

```
% ocpidev build applications
```

From the application's directory itself, you can simply use:

```
% ocpidev build
```

With no target platform specified, the executable is built to run on the development system itself. To build for other software platforms, you can use the **--rcc-platform** option (multiple times if desired).

E.g., for the ZedBoard embedded platform, the software platform name for the embedded linux is something like **xilinx24_1_aarch32**. Thus to build an ACI program for that system would be:

```
% ocpidev build --rcc-platform xilinx24_1_aarch32
```

To build the application for a software (RCC) platform associated as the default for a particular HDL platform, you can use the option **--hdl-rcc-platform**, which adds the RCC platform associated with the specified HDL platform to the platforms to build. Not all HDL platforms have a specific associated RCC platforms, but SoC platforms like Zynq usually do, since there is an FPGA side and a GPP CPU side to one chip.

All executables are created in subdirectories named **target-<*platform*>**, so executables for multiple different platforms can coexist.

10.2.2 Application XML Files In Projects

At runtime, an OAS XML file described above in the section on OAS XML Files is used to execute the application, whether in a development environment or a pure runtime environment (with no projects and no **ocpidev**). The OAS in an application's directory *in a project* may have additional attributes that specify options for how the application is built, run, and cleaned in the project (development) environment using **ocpidev**.

These attributes are ignored if the application is executed outside the development/project environment (i.e. not using **ocpidev**). They specify:

- Additional source files to compile for an ACI application
- Additional collocated executables to build along with the application
- Additional secondary source files to compile with the ACI application
- Prerequisite libraries required by the ACI application.
- Options for executing the application in a project environment

For ACI applications, the actual name of the OAS XML file used at runtime is determined in the C++ source code. In fact, the OAS can be a literal or constructed string in the C++ source with no OAS file used at all for execution. The **ocpidev**-related attributes listed below may be in the OAS file if it exists (it must exist for an XML-based application created using the **--xml-dir-app** option). In some cases, it is preferable to put the **ocpidev**-related attributes in a separate file (leaving the OAS a purely runtime-environment application specification for use with **ocpirun**). In this latter case, the runtime and development attributes can be in a separate XML file named ***<appname>*-app.xml**, which is only used by the **ocpidev** command for the **build**, **run**, and **clean** functions.

The following table lists these **ocpidev**-related attributes in the OAS. There are no child elements for development purposes:

Table 8: Attributes in the Application Development XML File

Attribute Name	Description
Options for development and building	
FileName	The name of the primary source file for an ACI application. Defaults to the application's name: the name of its directory. This value is without any language suffix such as **.cc**, **.cxx**, **.c** etc.
OtherMains	The names of other main source files, without language suffixes. Used for related executables to be built in this same directory.
SourceFiles	Other source files to be compiled and linked with the executables built in this directory. These file names include suffixes, which may be **.c**, .**cc**, .**cxx**, **.cpp**.
PrereqLibs	Installed prerequisite library names that are required to build the executable(s) from the source files here. E.g. **liquid**, a DSP function library that is installed with OpenCPI. If the executables are built statically (the default), the statically-linked version of the library will be used; otherwise, the dynamically-linked version will be used.
PackageName	The package name for this application.
PackageID	The full packageID for this application.
ExcludePlatforms	Platforms that should not be built in this directory.
OnlyPlatforms	The only platforms for which this directory should be built.
*Options for runtime/execution in the development environment, using **ocpidev run***	
NoRun	If set, indicates that this application cannot be run automatically using **ocpidev run**. When the run is attempted, a message is printed and the exit status is 0 (no error indicated).
RunBefore	Command arguments on the command line before the executable name (e.g. environment variable settings).
RunArgs	Command arguments immediately following the executable name (e.g. command line options, to **ocpirun**).
RunAfter	Command arguments at the end of the command line.
*Options for cleaning **in the development environment,** using **ocpidev clean***	
CleanFiles	Files and directories to be removed when the application is cleaned.

10.2.3 Executing/Running Applications in Projects

All applications in a project may be run in sequence using the command:

```
% ocpidev run applications
```

from a project directory or its **applications/** subdirectory. In the **applications/** directory, only this command is necessary to run all applications:

```
% ocpidev run
```

This will run all applications, one after the other, with no arguments specified. To run a particular ACI application, you can either run **ocpidev run** in the application's directory, or, from the project or **applications/** directory, you can use:

```
% ocpidev run application <appname>
```

This also works with simple XML applications (those without their own directories), whether ***<appname>*** includes the **.xml** suffix or not.

When applications are run using **ocpidev**, the **OCPI_LIBRARY_PATH** environment variable, if not set already, will be automatically set to search the project's own **artifacts/** subdirectory (where all artifacts built in the project reside), as well as the exported artifacts from any of the projects this project depends on.

Running all the applications in a project with default arguments is normally used for test purposes. If an application cannot be run with default arguments or needs to interact with a user, it should not fail (with non-zero exit status), but simply avoid running and print a message that it is not being run because it cannot run in this simple way with default arguments. This allows for an easy “test run of all applications in the project”. Setting the **noRun** attribute to non-empty in the ***<application>*.xml** file avoids running the application, but prints a message and returns a zero exit status indicating success.

Several **ocpidev** options to change how the application is run in the project environment, are described in the following table. For each, there are also XML attributes as described in the above table. While the XML attributes are lists, these **ocpidev** options specify one value, but may be used more than once. The XML attributes act as defaults, with the **ocpidev** options overriding them.

*Table 9: Options for **ocpidev** when Running Applications*

Command Option	Description
--before	Arguments to insert before the ACI executable or **ocpirun**, such as environment settings or prefix commands like **time** or **valgrind**. The XML equivalent attribute is **runBefore**
--run-arg	Arguments inserted just after the ACI executable or **ocpirun**, such as **ocpirun** options like **-v** or **-m** -or **-p**. The XML equivalent attribute is **runArgs**.
--after	Arguments to insert at the end of the execution command line. The XML equivalent attribute is **runAfter**.

For XML applications, these options (or XML attributes) are applied to the underlying **ocpirun** command using the following pattern:

```
<befores> ocpirun <run-args> <xml-OAS-file> <afters>
```

For ACI C++ applications, the pattern is:

```
<befores> <executable> <run-args> <afters>
```

As a convenience, any non-option (positional) **ocpidev** arguments after the application name are added as **run-args**. When running an application in its own directory (simply using **ocpidev run**) a **--** (two hyphens) argument indicates that any further command arguments are *not* options to the **ocpidev** command itself, but are passed to the underlying executable (or **ocpirun**). For example, if you want to run an application with a number of **run-arg** arguments, you can do, in the project or **applications/** directory, any of these ways:

```
ocpidev run application myapp -- -v -d -Pxsim
ocpidev run application myapp --run-arg=-v --run-arg=-d --run-arg=-Pxsim
ocpidev run application myapp --run-arg="-v -d -Pxsim"
```

which will, for XML apps, perform:

```
ocpirun -v -d -Pxsim myapp
```

If the **runArgs** XML attribute was set as: **runArgs='-v -d -Pxsim'**, the same effect would result by just running:

```
ocpidev run application myapp
```

When in an application's own directory, the equivalent would simply be:

```
ocpidev run -- -v -d -Pxsim
```

10.3 HDL Assemblies in Projects

HDL assemblies may also be created in projects, using the command:

```
% ocpidev create hdl assembly <name>
```

Within a project, HDL assemblies are created in the **hdl/assemblies/** directory in the project. Similar to applications, that directory has a standard XML file named **assemblies.xml**, and it can contain a setting of the **Assemblies** attribute when the default behavior, of all assemblies being built in alphabetical order, is not desired.

Each assembly is created as an XML file in its own directory. Thus creating the HDL assembly whose name is **myassy**, would create the **myassy.xml** file in the **hdl/assemblies/myassy** directory. After editing this file to describe the required worker instances and connections, artifacts based on this assembly can be created using the **ocpidev build** command in that assembly's directory, or, for building all the assemblies in the project, the **ocpidev build hdl assemblies** command may be issued from the project directory, or the **ocpidev build** command may be issued in the **hdl/assemblies** directory.

The resulting FPGA artifact files, with the suffix **.bitz**, are created in target-specific directories with the prefix **container-**, created under the assembly's directory.

Details about this artifact building process are in the ***OpenCPI HDL Development Guide***.

HDL assemblies may be deleted using this command:

```
% ocpidev delete hdl assembly <name>
```

Once the HDL assemblies are built, resulting in the **.bitz** artifact files, applications can use them as long as they are accessible using the **OCPI_LIBRARY_PATH** environment variable. If applications are run using **ocpidev run**, the library path will automatically find the built assemblies in the same project as the application or in projects that the application's project depends on.

11 Deploying Applications in a Runtime Environment

OpenCPI applications require a small set of required dependencies during execution, which is considerably smaller than the requirements for OpenCPI development, which requires the installation of the OpenCPI Component Development Kit (CDK). While the installation requirements and procedures are described in full in the ***OpenCPI Installation Guide***, the essential requirements required to support application execution are described here.

The basic requirements are to have the executable (**`ocpirun`** or a custom ACI C++ program), as well as the artifacts for the workers used during execution. The executable can use whatever artifacts are available, as built for the available processing resources on the system. By limiting the artifacts available (accessed via the **`OCPI_LIBRARY_PATH`** environment variable setting), the system requirements are reduced to the minimum.

Beyond these two important elements (executables and artifacts), there are several dynamically loaded and used plug-ins and drivers, depending on which hardware resources are enabled for OpenCPI to use during execution. There is an OpenCPI Linux kernel device driver module that is loaded using the **`ocpidriver`** command line tool. There are user-mode plug-ins that are loaded according to a system configuration file, at **`$OCPI_ROOT_DIR/system.xml`**, or a location indicated by the **`OCPI_SYSTEM_CONFIG`** environment variable.

All of these drivers and plug-ins are optional, and used based on the hardware needed by OpenCPI application execution. The following table lists them and their use cases:

Table 10: Loadable Drivers for OpenCPI Execution

Driver	Required for:
Kernel Module	Access to DMA devices on the system bus. E.g. FPGA PCIe cards in slots on the system's motherboard, or Zynq/SoC systems using the FPGA subsystem. Loading this driver requires root/sudo privileges. It is loaded using the **`ocpidriver`** command. See the ***OpenCPI User Guide*** for details on this module.
RCC Container Plug-in	Execution of RCC/Software workers. Indicated in **`system.xml`**. See the ***OpenCPI User Guide*** for details on this file.
HDL Container Plug-in	Execution of HDL/FPGA workers (in hardware or in simulators). Indicated in **`system.xml`**.
DMA Transport Plug-in	Data plane transport between software containers and DMA devices, e.g. FPGA containers. Requires the kernel module to be loaded prior to execution. Indicated in **`system.xml`**.
PIO Transport Plug-in	Data plane transport for software containers using shared memory. Indicated in **`system.xml`**.
Socket Transport Plug-in	Data plane transport for containers using network sockets. Indicated in **`system.xml`**. Required by HDL simulators and remote containers.

In addition to the executables, artifacts, and optionally loaded drivers and plug-ins, there are a small number of utility scripts (such as **ocpidriver**), support files (such as **system.xml**), and utility commands (which includes **ocpirun**) that are typically included in a runtime-installation. Installation packages are prepared for a particular runtime environment. If not using remote containers, the **ocpiadmin** command:

```
ocpiadmin deploy platform <rcc-platform> <hdl-platform>
```

can be used to create the contents of an SD card for the embedded system.

For some hardware and software configurations, third-party software is required for execution. One example is that for execution on FPGA boards that require dynamic JTAG loading of FPGA configuration files (a.k.a. bitstreams), there are drivers and utilities required from the FPGA vendors (Xilinx or Intel/Altera) that must be installed.

12 Execution Options Summary — Alphabetical

Table 11: Options for ocpirun, OAS XML, and ACI PValue

ocpirun Option Name	ocpirun Option Letter	OAS XML Attribute	ACI PValue Name	Type	Description
buffer-count	**B**	**bufferCount**	**bufferCount**	String	Set the number of buffers for an instance port or an external port.
buffer-size	**Z**	**bufferSize**	**bufferSize**	String	Set the size of both ports of a connect for instance port or an external port.
component				Bool	Use the first command argument after options as the component for a single-component OAS.
container	**c**		**container**	String	Specify the container for an instance.
deploy-out				String	Specify the filename to write deployment decisions into, which can be used later using the **deployment** option.
deployment				String	Specify a deployment file, containing hardwired deployment decisions.
device	**D**		**device**	String	Reserved for future use.
dump	**d**		**dump**	Bool	Dump property values to stderr, before starting the application, and after it is done
dump-file			**dumpFile**	String	Dump property values after the application is done, in a machine parseable format.
dump-platforms	**M**		**dumpPlatforms**	Bool	Dump property values for non-application workers before and after execution.
duration	**t**			ULong	Run the application for a time duration in seconds if it doesn't finish earlier.
file	**f**		**file**	String	Connect an external port of the application to a file.
hex	**x**		**hex**	Bool	When dumping property values, use hexadecimal when possible.
library-path				String	Set the OCPI_LIBRARY_PATH.
list	**C**			Bool	List available containers
list-artifacts				Bool	List available artifacts for the target values provided using **--target**
list-specs				Bool	List available specs for the target values provided using **--target**.
log-level	**l**			ULong	Set the logging level.
no-execute				Bool	Figure out how to execute the application, but then don't actually

					execute. Usually used with **--deploy-out**
model	**m**		**model**	String	Specify the model for an instance
only-platforms				Bool	When listing containers, actually only list platforms.
platform	**P**		**platform**	String	Specify the platform for an instance
processors	**n**			ULong	Specify the number of RCC containers to create.
property	**p**	**property** (element)	**property**	String	Instance-specific or application-level property value.
remote	**R**			Bool	Discover remote containers using Multicast UDP.
scale	**N**		**scale**	Scale	Set the scale factor for an instance
selection	**s**	**selection**	**selection**	String	Set the selection expression for an instance
server	**S**			String	Contact this server for remote containers
sim-dir				String	Set the directory for simulation output
sim-ticks				ULong	Set the number of clock cycles for simulations.
target	**r**			String	Set a target to use with **list-artifacts** or **list-specs**
timeout	**O**			ULong	Set a timeout after which the application will be stopped and considered failed.
transfer-role			**transferRole**	String	Reserved
transport	**T**	**transport**	**transport**	String	Set the transport for a connection
uncached	**U**		**uncached**	Bool	Dump properties without caching.
url	**u**		**url**	String	Reserved
verbose	**v**		**verbose**	Bool	Be verbose with messages
worker	**w**	**worker**	**worker**	String	Specify the worker for an instance.