OpenCPI
Application Development Guide

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

Table of Contents

1 References

This document depends on the OpenCPI Overview. 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 Overview	OpenCPI	Public URL: https://github.com/opencpi/opencpi/raw/2017.Q2/doc/pdf/OpenCPI_Overview.pdf
OpenCPI Component Development Guide	OpenCPI	Public URL: https://github.com/opencpi/opencpi/raw/2017.Q2/doc/pdf/OpenCPI_Component_Development.pdf
OpenCPI RCC Development Guide	OpenCPI	Public URL: https://github.com/opencpi/opencpi/raw/2017.Q2/doc/pdf/OpenCPI_RCC_Development.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 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***). Workers are developed based on an OCS, and when built, are available for applications that are specified 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. How such component implementations are developed is 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++ API for manipulating and executing XML-based applications*

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

Component: a specific function with which 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 *assembly*).

Assembly: the composition of instances that define an application

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 communicate 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 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.

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

Package: a name scope for OpenCPI assets, mostly used for *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 ***workers*** are runtime 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***. In fact, we use the term ***worker*** both for a specific (coded) implementation of a ***component***, as well as the runtime instances of that implementation.

The 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 implementation), how the resulting ***workers*** should be connected, and how they should be configured via their properties. Specifying an instance is based on the name of the *component specification*.

The runtime software uses this name 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 name of a component specification can be prefixed with a ***package*** name (followed by a period). 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.

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* (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.

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

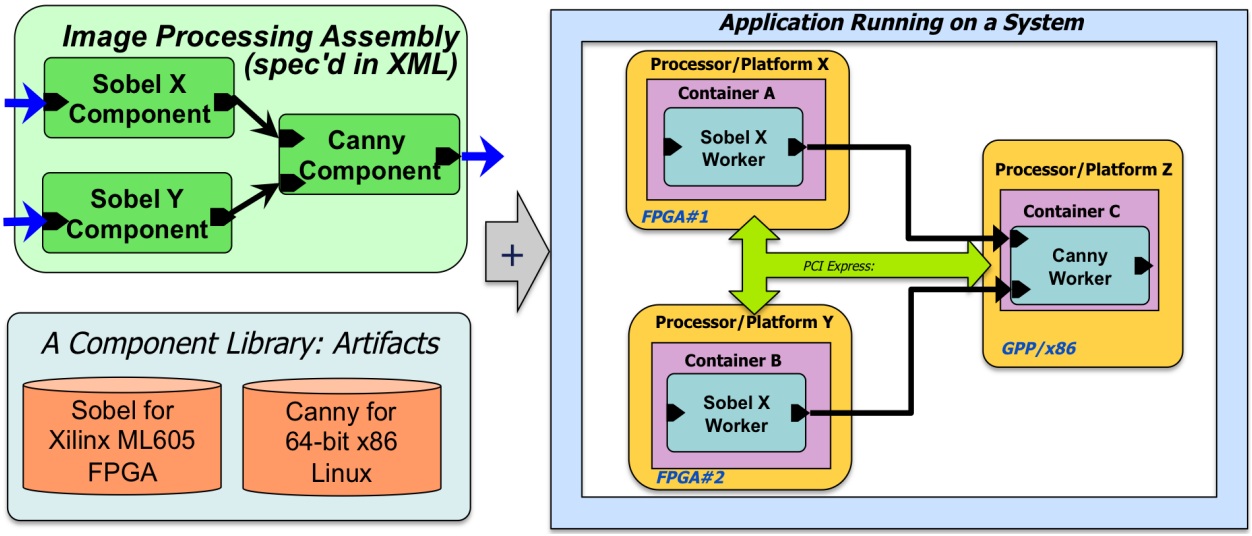

Figure 1: An Application of Components Deployed on a System

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 an assembly (collection) of components, 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 very 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.

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.

An element can be entered directly (as above) or entered 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">
  <xi: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.

In OpenCPI all attributes are defined with specific data types and/or 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. All 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`**, **`done`**, **`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 Done attribute (optional)

The **`done`** attribute identifies the instance within the OAS that is used to determine when the application is "done" executing. When the indicated instance is "done", then the whole application is considered "done". If this attribute is not supplied, the application is considered "done" when *all* its instances are "done". 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 "done". 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 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 **`local`**. The prefix of the core OpenCPI component library is **`ocpi`**. If you are using mostly components in that library, you might include **`package='ocpi'`** as an attribute. If you are using only components specified in your own library of components (which has a default prefix of **`local`**), you could ignore this attribute and use no prefixes at all. See the **`component`** attribute of the **`instance`** element below.

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** child elements, and may have **component**, **name**, **connect**, **selection**, **from**, **to** 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 name of 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.

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, 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.

*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.

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.

*3.3.3 **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. **centos7** or **ml605**
  - **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**.

*3.3.4 **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.

*3.3.5 **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.

*3.3.6 **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.

*3.3.7 **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.

*3.3.8 **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, without any indication of authoring model or package name prefix.

*3.3.9 **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. 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 with access as *initial* or *writable*.

3.3.9.1 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.

*3.3.9.2 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. I.e. 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.

*3.3.9.3 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.

3.3.9.4 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 using the **`ocpirun`** utility program described below). 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.

3.4 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).

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. See the OpenCPI Installation Guide.

3.5.3 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. However, port elements can be used to indicate more than two ports on the same connection, when there are multiple consumers for the connection (currently not supported).

3.5.3.1 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.

3.5.3.2 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.

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 “done”. As mentioned above, an application is done either when all the workers indicate they are “done” or when a single worker, identified using the **`done`** attribute in the OAS, says it is done. 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.

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.

4.1 General Options for `ocpirun`

The general options are:

Table 3: General Options to `ocpirun`

Name	Letter	Datatype	Description
`dump`	**`d`**	Bool	Dump all readable properties after initialization, and again after execution, to stderr.
`verbose`	**`v`**	Bool	Be verbose in describing what is happening.
`hex`	**`x`**	Bool	Print numeric property values in hex, not decimal.
`uncached`	U	Bool	When dumping property values do not use cached values that are "remembered" by **`ocpirun`** when they are written. Actually query the worker in its execution environment (which is much more expensive).
`processors`	**`n`**	ULong	Create this many RCC containers (default is 1).
`log-level`	**`l`**	ULong	Set the OpenCPI log level to the given level. Overrides any value of the OCPI_LOG_LEVEL environment variable.
`seconds`	**`t`**	Long	Stop execution after this many seconds. Used when there is no definition of "done" for the application, thus it would otherwise run indefinitely or until **`ocpirun`** was interrupted (e.g. control-C). If negative, it is the time to wait for "done". I.e. wait until done or until this many seconds have passed.
`server`	**`S`**	String	An server (name or IP address) to explicitly contact for remote containers whether or not the remote option is not specified. This option may be specified multiple times.
`remote`	**`R`**	Bool	Automatically discover servers offering remote containers using multicast UDP.
`deploy-out`	*none*	String	The name of a file in which to record deployment decisions for this execution, in XML format. Such an output file can be supplied in the **`deployment`** option, to specify the exact deployment recorded from a previous run (or from a **`no-execute`** function as defined below).
`library-path`	*none*	String	Override the **`OCPI_LIBRARY_PATH`** environment variable.
`dump-file`	*none*	String	The name of a file in which to record the final property values in machine-parseable form.

4.2 Function Options for `ocpirun`

The function options tell the ocpirun command to perform certain functions *other than executing the application*. Options only relating to these non-execution functions are also listed here in the following table.

Table 4: Function Options to `ocpirun`

Name	Letter	Datatype	Description
`list`	**`C`**	Bool	List all available containers, including those discovered on the network (if the **`-R`** or **`-S`** options are specified). Assign each a number for easy assignment with the **`-c`** instance option described below. The application is still executed if an application filename argument is specified after the options.
`only-platforms`	*none*	Bool	Modifies the **`list`** command to only print available platforms, listing any available platform only once even if there are more than one.
`list-artifacts`	**`A`**	Bool	This function searches for all artifacts for any of the listed targets or platforms (using the **`–target`**, or **`-r`** option), based on **`OCPI_LIBRARY_PATH`**, and prints the list to stdout. Used to collect artifacts for a specific system.
`list-specs`	*none*	Bool	This function searches for all artifacts for any of the listed targets or platforms (using the **`–target`**, or **`-r`** option), based on **`OCPI_LIBRARY_PATH`**, and prints the list to stdout. Used to collect available specs for a specific system.
`target`	**`r`**	String	A target (<*os*>-<*os-version*>-<*arch*>), or platform for the **`list-artifacts`** and **`list-specs`** commands. May be specified more than once.
`no-execute`	*none*	Bool	After the normal process of deciding, for each instance, what container it will run on, and what artifact will be used to run it, stop short of actually allocating any resources or performing execution. This option can be used as a “dry run” to see what would happen, and, with the **`deploy-out`** option, record the deployment decisions in an XML file.

4.3 Instance Options for `ocpirun`

The instance options allow a value to be specified that applies to either *all* instances or just *one* instance. All these options take string values and the values are of the form:

`[<instance-name>]=<value>`.

So, the option:

```
-m=rcc
```

would set the **`m`** option (the authoring model) for *all* instances to **`rcc`**, while the option:

```
-mctl=rcc
```

would set the **`m`** option for the **`ctl`** instance to be **`rcc`**. These options can appear more than once to indicate options for different instances. Most of these options override the previously described automatic deployment algorithm.

Specifying the option usually provides a constraint on the algorithm to only consider certain workers or certain containers. The following table lists all instance options:

Table 5: Instance Options to ***`ocpirun`***

Name	Letter	Description
container	**`c`**	Assign the instance to a specific container, using the name or number from the listing of the **`-C`** command. Examples: **`-c fft=1 -c fir=rcc2`**
model	**`m`**	Specify the authoring model of the named instance. This creates a constraint that the worker used for this instance must have this model. Examples: **`-m=hdl -m fft=rcc`**
platform	**`P`**	Assign instance only to containers for this platform type (see output from **`-C`**). Examples: **`-Pfft=ml605 -P=centos7`**
property	**`p`**	Set the value of a property. The value of the option is either: **`<property-name>=<value>`** for application-level properties, or: **`<instance-name>=<property-name>=<value>`** for per-instance property value settings. See below for more details.
selection	**`s`**	Set the selection expression for the instance. See below for more details. Example, to request that for the **`ctl`** instance, only workers with the **`snr`** property less than **`5`** are used: **`-s 'ctl=snr<5'`**. See the ***Instance Selection Attribute*** section.
worker	**`w`**	Specify the name of the worker (specific implementation) to be used for the instance. Note this does not include model suffix or package prefix.

4.3.1 Setting Properties in the Application

The property setting option (**`-p`**) can set an application-level property rather than an instance property. Application-level properties are those specified using the **`property`** element at the top level of the OAS, as a child element of the **`application`** element. Application-level properties have an application-level name that is mapped in the OAS to the underlying instance

When setting a top level application property value, the form of the option is:

```
-p control=5
```

which sets the application level **`control`** property to the value **`5`**.

When specifying the property value of an instance, the form of the option is:

```
-p file_read=filename=myinput.data
```

which sets the **`filename`** property value of the **`file_read`** instance to **`myinput.data`**.

Property values must be consistent with the data types defined in the component specification. The syntax for all data types is described in the ***Property Value Syntax and Ranges*** section below.

4.4 Simulation Options for `ocpirun`

The simulation options are used to control containers that are running HDL/FPGA simulators. Any available simulators are used as normal containers during the deployment algorithm which decides what artifacts are used and where instances will run. Available simulators are included in the output of the **`--list`** (or **`-C`**) function of **`ocpirun`**.

These options are normally used during HDL/FPGA component development and testing, but are listed here for completeness. They are described in more detail in the OpenCPI HDL Development document.

Table 6: Simulations Options to `ocpirun`

Name	Letter	Description
`sim_dir`	*none*	The name of a directory where simulation outputs will be placed. The default is **`simulations`**, relative to where **`ocpirun`** itself is running.
`sim-ticks`	*none*	The number of simulation clock cycles to execute or until the application is done.

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 three places:

- the **`value`** attribute of property elements in the 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 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 syntax is on the shell command line and 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. These 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.

4.5 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.

4.6 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.

4.7 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, the backslash can also escape commas and braces, i.e.:

\, \{ \}

4.8 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.

4.9 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.

4.10 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 the 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.

4.11 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.

4.12 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.

4.13 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, recursively.

4.14 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.

4.15 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 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
```

4.15.1 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.
- 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, 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.

4.15.2 String Values

String constants (using double quotes) can be used in expressions using string-typed parameters. 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 or not). The **+** operator concatenates strings. There is no implicit or explicit conversion between string values and numeric values. So 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 Utility Components for Applications

There are several built-in components that are always available that application developers use frequently. These are listed in this section, along with the available implementations. 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.

All of these utility components are in the **`ocpi`** package, so using them usually involved specifying the instance's **`component`** attribute to **`ocpi.<component>`**.

5.1 File_Read Component that Reads Data or Messages from a File

The File_Read component 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 a 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, so that it is agnostic as to the protocol of the file data and the connected input port.

This component has two modes of operation: data streaming and messaging.

5.1.1 Data Streaming Mode

In data streaming mode, the contents of the file becomes the payloads of a stream of messages, each carrying a fixed number of bytes of file data (until the last) and all with the same opcode. The length and opcode of all output messages are specified as properties.

If the number of bytes in the file is not an even multiple of the message size the remaining bytes are sent in a final shorter message. The ***granularity*** of messages can also be specified. This forces the message size to be a multiple of this value, and forces truncation of the final message to be a multiple of this value. The default granularity is 1.

5.1.2 Messaging Mode

In messaging mode, the contents of the file are interpreted as a sequence of defined messages, with an 8-byte header in the file itself preceding the data for each message. This header contains the length and opcode of the message, with the data contents of the message following the header. The length can be zero, meaning that a message will be sent with the indicated opcode, and the length of the message will be zero.

The first 32-bit word of the header is interpreted as the message length in bytes, little-endian. The next 8-bit byte is the opcode of the message, followed by 3 padding bytes. E.g. in the C language (on a little-endian processor):

```
struct {
  uint32_t messageLength;
  uint8_t  opcode;
  uint8_t  padding[3];
};
```

This format of messages in a file is the format produced by the File_Write component when in messaging mode, described next.

If the end of the file is encountered while reading a message header, or while reading the header-specified length of the message payload, an error will be reported and the component will terminate.

5.1.3 End of File Handling

When the File_Read component reaches the end of its input file, it will do one of three things:

- send a zero-length message with an opcode set by the **`opcode`** property (default)
- declare itself to be "done" with no further action, when the **`suppressEOF`** property is **`true`**
- restart reading at the beginning of the file, when the **`repeat`** property is **`true`**

5.1.4 Properties

*Table 7: Properties of the **`File_Read`** Component*

Name	Read Access	Write Access	Default	Type	Description
`fileName`	Impl	Initial	*none*	String	Name of file to be read
`messagesInFile`	Readable	Initial	false	Bool	Indicates messaging mode
`opcode`	Readable	Initial	0	UChar	Opcode for all outgoing messages in data streaming mode, and EOF ZLM in messaging mode.
`messageSize`	Readable	Initial	4096	ULong	Size of outgoing messages, subject to granularity
granularity	Readable	Initial	1	ULong	Granularity of outgoing messages
repeat	Impl	Initial	false	Bool	Whether to repeat the file at EOF
suppressEOF	Impl	Initial	false	Bool	Do not send final EOF/ZLM
bytesRead	Volatile	none		ULong Long	How many bytes were read?
messagesWritten	Volatile	none		ULong Long	How many messages were written to the output port?
badMessage	Volatile	none		Bool	Was a bad messages encountered in the file?

Note all writable property values are cached and their values are readable even when the component does not specify this. When **`impl`** appears for read access, it indicates the (uncached) readability is implementation dependent.

5.2 File_Write Component that Writes Data or Messages to a File

The File_Write component 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 produces the data. The name of the file to be written is specified in a 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.

This component has two modes of operation: data streaming and messaging. These are similar, but not identical to the modes described in the **`File_Read`** component above.

5.2.1 Data Streaming Mode

In data streaming mode, the contents of the file becomes the payloads of the stream of messages arriving at the input port. No message lengths or opcodes are recorded in the output file.

5.2.2 Messaging Mode

In messaging mode, the contents of the output file is written as a sequence of defined messages, with an 8-byte header in the file itself preceding the data for each message written to the file. This header contains the length and opcode of the message, with the data contents of the message following the header. The length can be zero, meaning that a header will be written but no data will follow the header in the file.

The first 32-bit word of the header is written as the message length in bytes, little-endian. The next 8-bit byte is the opcode of the message, followed by 3 padding bytes. E.g. in the C language (on a little-endian processor):

```
struct {
  uint32_t messageLength;
  uint8_t  opcode;
  uint8_t  padding[3];
};
```

This format of messages in a file is the format consumed by the File_Read component when in messaging mode, described earlier.

5.2.3 End of File Handling

When the File_Write component receives a zero length message, it will interpret it as the end of data if the **`stopOnEOF`** property is **`true`** (the default). In this case it will declare itself “done”, and not write any further messages to the file (and not write the zero-length message to the file either). If **`stopOnEOF`** is **`false`**, and it is in messaging mode, it will write the zero-length message to the file like any other message (with a header, but with no data).

5.2.4 Properties

*Table 8: Properties of the **`File_Write`** Component*

Name	Read Access	Write Access	Default	Type	Description
`fileName`	Impl	Initial	*none*	String	Name of file to be written
`messagesInFile`	Readable	Initial	false	Bool	Indicates messaging mode
stopOnEOF	Impl	Initial	true	Bool	Be “done” upon receipt of EOF/ZLM
bytesWritten	Volatile	none		ULong Long	How many bytes were written to file?
messagesWritten	Volatile	none		ULong Long	How many messages were written to the file?

Note all writable property values are cached and their values are readable even when the component does not specify this. When **`impl`** appears for read access, it indicates the (uncached) readability is implementation dependent.

6 API for Executing XML-based Applications in C++ Programs: 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.

1. The contents of the application XML (OAS) need to be constructed programmatically.
2. The C++ (main) program needs to directly connect to the ports of the running application (see ***external ports*** below).
3. The XML-based application needs to be run repeatedly (perhaps with configuration changes) in the same process.
4. Some of the attributes of the XML application need to be dynamically overridden by the C++ application.
5. 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++ application using this interface. In all examples below, the 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 referring to 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 "done"
}
```

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"
} catch (std::string &e) {
  std::cerr << "app failed: " << e << std::endl;
}
```

6.1 Class OA::Application

This class represents a running application, with a simple lifecycle. It has constructors and destructors suitable for automatic storage, 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 chooses an implementation from those available in the libraries, for each instance in the OAS. 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 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 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`** flags 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"
}
```

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 ordering rules described in the CDG.

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 in the CDG.

6.1.5 OA:: Application::wait Method

This method blocks the caller until the application is done: when all the workers are done or when the worker indicated by the “done” attribute in the OAS, is done. 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 “done”.

```
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 “done”. 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::getProperty Method

This method gets a property value by name or ordinal, returning the value in string 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`**.

Identifying properties by ordinal is useful when enumerating all properties. When not identified by ordinal, application-level properties are simply identified by name and

Instance-level properties are identified by a combination of *instance-name* and *property-name*.

When the property is identified by name, there are two arguments: the instance name and the property name. For top level (application-level) properties, the instance name may be NULL. As a convenience, when the instance name argument is NULL, the property name may still specify an instance property using the syntax: ***<instance>.<property_name>***

If there is no property with the given name, or the property is not readable, or some other error occurs reading the property value, an exception is thrown.

```
class Application {
  void getProperty(const char *instance_name, const char *pname,
                   std::string &value, bool hex = false);
  bool getProperty(unsigned ordinal, std::string &name,
                   std::string &value, bool hex = false,
                   bool *parameterp = NULL, bool *cachedp = NULL,
                   bool uncached = false);
};
```

When accessing a property by ordinal, its name is also returned in a **`std::string`** provided by reference. This is useful to retrieve all property values (and names) without knowing their names. The return value is **`true`** if the ordinal is valid. Thus a simple loop can retrieve all properties:

```
std::string name, value;
for (unsigned n = 0; app.getProperty(n, name, value); n++)
  std::cout << name << ":" << value << std::endl;
```

Other optional arguments are:

`hex` — an input indicating that numeric integer values should be in hexadecimal rather than decimal

`parameterp` — an output boolean indicating that the property is a parameter

`cachedp` — an output boolean indicating whether the value is cached (since the system knows what was last written)

`uncached` — an input requesting that caching should be ignored (values always read from workers, even when otherwise cached).

6.1.8 OA::Application::setProperty Method

This method sets a property value by name, taking the value in string 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 arguments are the same as in the **`getProperty`** method.

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 *instance_name,
                   const char *property_name,
                   const char *value);
};
```

6.1.9 OA::Application::getPropertyValue Method

This method gets property values in their native data type, without converting to a string form (as done in the **`getProperty`** methods). The properties are identified by name as is done in the **`getProperty`** methods (one or two names). 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, and has different variants for one or two names, and whether the value is returned as the return value or as an output argument:

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

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>(NULL, "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>(NULL, ¨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>(NULL, ¨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>(NULL, ¨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>(NULL, ¨prop¨, A(2));
```

6.1.10 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 *instance_name,
                                    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(NULL, ¨prop¨, 1.2, {2, ¨b¨});
```

6.1.11 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 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 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 unspecified.

The following diagram shows the relationships between Application objects and the ExternalPort objects and ExternalBuffer objects described next.

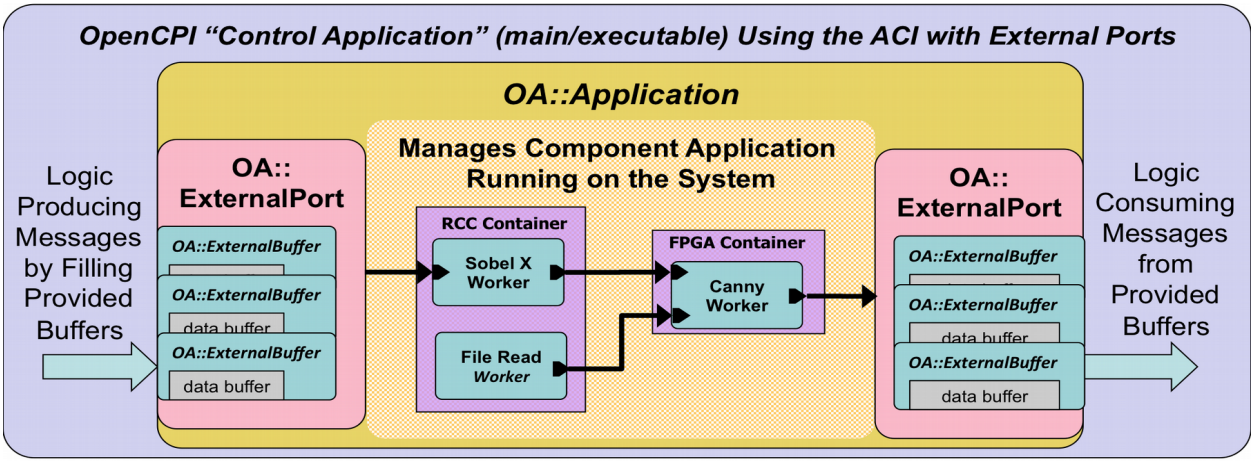

Figure 8: External Ports and Applications

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::getBuffer Method

This method is used to retrieve the next available buffer of the port. It returns 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 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 returned buffer object manages a data buffer containing the next message to be received by the control-application. When the control-application is done with the buffer, it calls the **put** method (when giving data) or the **release** method (when taking data).

In addition to returning a pointer to the buffer object, the **getBuffer** method also returns (as output arguments by reference), a pointer to its raw data buffer and the length of the message (when taking) or the length of the buffer (when giving). These are convenience values that are attributes of the returned buffer object. The data pointer returned by reference points to memory owned by the buffer object.

There are two overloaded **getBuffer** 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 (**opCode** and **endOfData**) in separate by-reference output arguments.

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

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::getBuffer`** 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::getBuffer`** 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.getBuffer(data, length, opcode, end); while (!b);
  printf("%u: .*s\n", opcode, (int)length, data);
  b->release();
}
```

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 (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.getBuffer(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 very 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 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 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 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. It 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” 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 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 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. When the ACI is used outside OpenCPI projects, presumably in the context of other software libraries or executables, the OpenCPI CDK must still be installed.

The make file fragment **ocpisetup.mk** (from the **include** subdirectory of the CDK installation) can be included or examined to determine the correct values of gnumake variables for use outside OpenCPI projects. In particular:

- The include file search path must include the directory defined in **OCPI_INC_DIR**.
- 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 subdirectory 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_PREREQUISITE_LIBS** must be included as *static* libraries using, e.g.:

```
$(foreach l,$(OCPI_PREREQUISITES_LIBS),\
  /opt/opencpi/prerequisites/$l/linux-c7-x86_64/lib/lib$l.a)
```

Building ACI programs for embedded systems outside of OpenCPI projects is not explicitly supported.

7 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 Altera Stratix4)
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.

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

8 Developing Applications in Projects

Applications are typically simple XML files (OASs) that rely on the existence of existing artifacts. They may also be created as part of an OpenCPI ***project*** containing other OpenCPI assets such as components, workers, primitive libraries, etc. In such cases it is advantageous to also create related applications within an OpenCPI project using the **ocpidev** command. Even if the project contains *only* applications, there are some advantages to putting the applications into projects, especially during development.

Projects are considered development work areas and exist on development systems with the OpenCPI CDK installed. They use scripts and executables found in the installed CDK, and use the **make** utility to build and execute applications. This is different from the more minimal requirements of an execute-only environment that has significantly fewer installation and execution dependencies.

The **ocpidev** command is fully described and documented in the CDG, but for pure application users (that are not developing components or workers), the small necessary subset of this command's functionality is described here. Projects are created with the command:

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

This creates a project in a directory ***<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 9: 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 run, analogous to the **-C** option in the POSIX **make** command.
-D	yes		Specify a colon-separated list of other projects that this project depends on.
-K	yes	**local**	Specify the package name 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.

8.1 Applications in Projects

Applications in projects live in the **applications/** subdirectory of the project and are either XML applications, based on an OAS 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 C++ 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>
```

*Table 10: Options for **ocpidev** when Creating Applications*

Option	Value?	Default	Description
-v	no		Be verbose, describing what is happening in more detail.
-X	no		The application will be a simple XML OAS file in the **applications/** directory of the project, named **<name>.xml**.
-x	no		The application will be an XML application file in its own directory, in **applications/<name>/<name>.xml**.

The **-X** or **-x** options specify an XML application, the latter in its own directory. In these cases an empty OAS file is created with the indicated name and can then be edited as necessary to create the application.

Without the **-X** or **-x** options, an ACI C++ application is created in its own directory under the **applications/** directory with the indicated name in a file named **<name>.cc**, containing the main program.

When an application is created in its own directory (either XML or ACI C++) a default **Makefile** is created in that directory. Placing an application in its own directory allows customizations in the associated **Makefile** for options, and the inclusions of test or data files or even other **make** targets.

The **Makefile** in the top level **applications/** directory will build (for ACI-based C++ applications), and execute all 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** variable may be set in this **Makefile** to contain a list of the applications to be used and the order in which they are built and/or used, e.g.:

```
Applications=myapp1 myapp3 # do not use myapp2 for now
```

To delete applications, the following command is used. The same options allowed for creating an application are valid for deleting one.

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

8.1.1 Building Applications in Projects

ACI-based applications are built in their own directories, simply using **make**. With no target platform specified, the executable is built to run on the development system itself.

To build for other software platforms, you can provide a list using the RccPlatforms variable, which specifies to build the application for all the platforms in this (space-separated) list. You can include the development system in this list by using the OCPI_TOOL_HOST variable.

E.g., for the ZedBoard embedded platform, the software platform name for the embedded linux is something like **xilinx13_4**. Thus to build an ACI program for that system would be:

```
% make RccPlatforms=xilinx13_4
```

To build for both the development system as well as this embedded platform would be:

```
% make RccPlatforms='$(OCPI_TOOL_HOST) xilinx13_4'
```

To build the application for a software (RCC) platform associated with a particular HDL platform, you can use the variable RccHdlPlatforms, which essentially adds the RCC platform associated with these HDL platforms to the RccPlatforms variable. 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.

This command can be used at the top level of a project, in the **applications** directory, or in a particular application's directory.

8.1.2 Executing Applications in Projects

Simple OAS XML applications may be run directly using the **ocpirun** command. All applications in a project may be run in sequence using the **make run** command from a project directory or its **applications/** subdirectory. This will run all applications, one after the other, with no arguments specified. To run a particular application, you can either run **make run** in an application's directory, or, from the project or **applications/** directory, you can provide the application's directory using the -C option to make, e.g.

```
% make -C applications/myapp run # from the project directory
% make -C myapp run # from the project's applications/ directory
```

Running all the applications in a project is normally used for test purposes.

To provide arguments to applications, these **Makefile** variables may be set, either in Makefiles or on the **make** command line.

Table 11: Make Variables for Running Applications

Makefile Variable	Description
OcpiRunBefore	Arguments to insert before the ACI executable or **ocpirun**, such as environment settings or prefix commands like **time** or **valgrind**.
OcpiRunArgs	Arguments to insert immediately after the ACI executable or **ocpirun**, such as ocpirun options like **-v** or **-m** -or **-p**
OcpiRunAfter	Arguments to insert at the end of the execution command line.
OcpiRunBefore_*<app>*	Like **OcpiRunBefore**, but only for the ***<app>*** application.
OcpiRunArgs_*<app>*	Like **OcpiRunBefore**, but only for the ***<app>*** application.
OcpiRunAfter_*<app>*	Like **OcpiRunBefore**, but only for the ***<app>*** application.

For applications other than simple OAS XML applications without their own directory, the **Makefile** may be customized for any control-application arguments.

8.2 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 **Makefile**, and it can contain a setting of the **Assemblies** variable 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 **make** command in that assembly's directory, or, for building all the assemblies in the project, the **make** command may be issued from the **hdl/assemblies** directory.

The resulting FPGA artifact files, with the suffix **.bitz**, are created in target-specific directories 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.

9 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 drivers, depending on which hardware resources are enabled for OpenCPI to use during execution. There is a Linux kernel module that is loaded using the **`ocpidriver`** command line tool. There are user-mode drivers that are loaded according to a system configuration file, at **`/opt/opencpi/system.xml`**, or a location indicated by the **`OCPI_SYSTEM_CONFIG`** environment variable.

All of these drivers are optional, and are used based on the hardware that is intended to be used by OpenCPI application execution. The following table lists the drivers and their use cases:

Table 12: 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 systems using the FPGA subsystem. Loading this driver requires root/sudo privileges. Loaded using the **`ocpidriver`** command.
RCC Container Driver	Execution of RCC/Software workers. Indicated in **`system.xml`**
HDL Container Driver	Execution of HDL/FPGA workers (in hardware or in simulators). Indicated in **`system.xml`**
DMA Transport Driver	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 Driver	Data plane transport for software containers using shared memory. Indicated in **`system.xml`**
Socket Transport Driver	Data plane transport for software containers using network sockets. Indicated in **`system.xml`**

In addition to the executables, artifacts, and optionally loaded drivers, there are a small number of utility scripts (such as **`ocpidriver`**), support files (such as **`/opt/opencpi/system.xml`**), and utility commands (which includes **`ocpirun`**) that are typically included in a runtime-installation. Installation packages are generally prepared for a particular runtime environment.

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 Altera) that must be installed.

10 Glossary

Authoring Model – A particular way to write the source code and XML for a worker, usually associated with a class of processors, and a set of related languages.

Component Application – A component application is a composition or assembly of components that as a whole perform some useful function.

Configuration Properties – Named value locations of a worker that may be read or written. Their values indicate or control aspects of the worker's operation. Reading and writing these property values may or may not have side effects on the operation of the worker. Each worker (component implementation) may have its own, possibly unique, set of configuration properties.

Control-Application – A control-application is the conventional application that constructs and runs component applications.

Control Operations – A fixed set of control operations that every worker has. The control aspect is a common control model that allows all workers to be managed without having to customize the management infrastructure software for each piece of IP, while the aforementioned configuration properties are used to specialize components. The most commonly used are "start" and "stop".

Worker – A concrete implementation (and possibly runtime instance) of a component, written according to an authoring model.