OpenCPI Application Control Interface Specification

(ACI)

Authors: Jim Kulp

Revision History

Revision	Description of Change	By	Date
1.01	Creation	jkulp	2010-07-01

Table of Contents

1 Introduction

1.1 References

This document depends on several others.

Table 1 - Table of Reference Documents

Title	Published By	Link
OpenCPI Technical Summary	OpenCPI	Public URL: http://www.opencpi.org/doc

1.2 Purpose

The purpose of this document is to specify a C++ interface for launching and controlling OpenCPI applications. Having prebuilt ready-to-run component implementations (workers) is a prerequisite for running applications based on these prebuilt binary component implementations (workers).

This C++ interface creates and/or uses the following objects (and classes) to run and control applications:

Containers: execution environments/processors where workers might execute

Applications: collection of workers together performing the work of the application

Artifacts: binary files like DLL/.so files or bitstream files for FPGAs from which workers are instantiated

Workers: instantiated implementations of components from binary code in Artifacts

Ports: attachment points on workers that produce or consume data with other workers or IO to and from the application as a whole

ExternalPorts: data sources and sinks in the control application itself

Properties: configuration properties of workers that may be read or written at runtime

The purpose of this API is to programmatically assemble an application by:

- instantiating and initializing **Workers** (specific component implementations) from **Artifacts** loaded into **Containers**
- connect the Workers' **Ports** to each other or to the application control code (caller of this interface)
- set initial configuration **Properties** of workers
- start the execution of the **Workers** in the **Application**
- during execution possibly pause, resume, read and write dynamic **property** values of the **workers** in the application
- tear down the **application** (all the **workers** on all the **containers** used).

Each of these object classes and their methods (called "member functions" in C++) will be defined below. This interface is for cases where the controlling application (the

conventional application that runs and controls the component-based OpenCPI application) wants direct and fine-grained control over the application and the workers and ports in the application. Other mechanisms exist for running OpenCPI applications that are more "fully scripted" so that no such code is required. In particular other component frameworks like the DoD SCA or the ISO/OMG CCM can be used on top of OpenCPI applications using adaptation layers to these frameworks.

1.3 Requirements for all classes in this API

- Clear lifecycle
- Use references for things not under the callers lifecycle control
- Use the OCPI namespace to avoid any namespace collisions with user code

2 Overview

The OpenCPI execution framework for component-based applications is based on OCPI::Worker*s* executing in OCPI::Container*s*, communicating using their OCPI::Port*s*, configured using their OCPI::Property*s*. The OCPI::Worker*s* are runtime instances of component implementations that are based on compiled code in files represented by OCPI::Artifact objects. The term "artifact" is used as a technology neutral term representing a compiled binary file that is the resulting artifact of compiling (or for FPGAs, "synthesizing") and linking a batch of source code that implements some number of workers. That build process results in artifacts that can be loaded as needed and used to instantiate the OCPI::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 type of worker, it is also common to build artifact files that contain the implementations of multiple workers (hence the term "shared library").

The OCPI::Application class is used as the lifecycle object that owns all the OCPI::Workers instantiated for the execution of the (component-based) application. Thus the interface described below only has lifecycle (create and destroy) control over the OCPI::Application classes and the OCPI::Container classes.

OCPI::Container objects represent an execution environment for workers. Containers for software workers (RCC etc.) can be within the process of the application using this control interface or can represent some other execution environment (other process or other processing device) that is available to the OpenCPI implementation and installation. At the level of abstraction of this application control interface, the caller knows which OCPI::Artifacts are appropriate and acceptable to which type of OCPI::Container. Other higher-level frameworks may be smart enough to find the right Artifacts (binary files) that contain Worker implementations appropriate to the available Containers.

Thus the typical sequence for using this Application Control Interface is:

- Find containers suitable for executing the workers based on knowing which implementations are available in which artifacts.
- Establish an application context on each container in which workers can be created incrementally, and removed as a group based on that context.
- Load the appropriate artifacts into these application contexts on each container
- Create the worker runtime instances based on these artifacts
- Connect the ports of the workers within the containers, between the containers, and to the control application itself (via "ExternalPorts").
- Start the workers using the OpenCPI "start" control operation.

At this point the *component application* is running and the *control application* can interact with the running *component application* by a combination of (1) sending and receiving messages on external ports attached to worker ports, and (2) reading runtime property values of workers to obtain scalar results and status, and (3) setting runtime property values to control/parameterize/modify the execution of the workers, and (4) start and stop individual workers for various purposes including debugging.

When the *control application* is finished using the *component application*, it can simply destroy all the application contexts on all containers.

3 Classes in the Application Control Interface

The classes in the ACI are listed in the order you would use them in a typical program.

3.1 Class OCPI::ContainerManager

This class represents a singleton object in the process address space that is mostly used via static methods. It is a "bootstrapping" class in that it allows the control application to "get started" in using the ACI's objects. It provides access to containers that will be used as places to run workers. Since Container objects represent places to run workers, and they are commonly one-per-processing-node in the system, the ContainerManager can be thought of as the owner and manager of all containers accessible from the control application.

The ContainerManager has a number of ways to find out about containers in the system. Some of them may be internally configured, while others may be discoverable by looking at available hardware and available drivers for processing hardware in the system. The ContainerManager may be also explicitly informed of the existence of a container.

3.1.1 OCPI::ContainerManager::find

This method is used to find a container, usually of a particular type, but sometimes a specific container for a specific processor. From the point of view of the caller, all containers act the same, but control applications need to find certain types of container to run certain types of workers. The arguments of FindContainer allow the caller to narrow down the type of container it is looking for, in order to run certain workers. There are two common scenarios:

- Find a container to run a particular worker.
- Find a container and then find a worker that will run on that container.

There are a number of ways of identifying a container, and theoretically a container can magically execute a wide range of implementation types. However, the three primary aspects are authoring model (RCC, HDL, etc.), processor type/ISA (x86_64, virtex5), and a specific piece of hardware. Secondarily, there might be specific support for tool chains and operating environments. The ***find*** method has three overloaded versions. One simply has three arguments for the three most common container qualifiers: authoring model, processor, and specific machine, and a fourth argument to distinguish which container among those that match the previous criteria. All can be NULL for "don't care". The second method has an argument that is a property list using the PValue class.

```
namespace OCPI { class ContainerManager {
   static Container *
      find(const char *model,
           const char *processor = NULL,
           const char *machine = NULL,
           const char *which = NULL);
      find(const PValue *list);
};}
```

The return value will be NULL if no container can be found. Containers come into existence by being automatically discovered during the first call to the find method. There are many more complex ways that containers might be created but they are out of scope here. The default is that all container drivers that are linked with the OpenCPI library are used to discover and create available containers.

3.1.2 OCPI::ContainerManager::shutdown()

This method simply shuts down all containers and thus removes any resources associated with all containers. It is only used to ensure that no resources associated with containers are retained. It returns the OpenCPI subsystem to its initial state (after static construction).

```
namespace OCPI { class ContainerManager {
   void shutdown();
};}
```

3.2 Class OCPI::Container

This is the class of objects that represent execution environments for workers. It manages application contexts .

3.2.1 OCPI::Container::createApplication()

This method creates an application context object in which workers will be created. It can be deleted when all of the workers created with it are no longer needed. Note that the class of the created object is OCPI::Container::Application. Thus this object is scoped to the container that created it and represents the parts of the overall application that reside in this container. The OCPI::Application class (not in the Container namespace) is reserved for the application object that spans multiple containers (which is not yet supported).

```
namespace OCPI { class Container {
   class Application;
   Application *createApplication();
};}
```

The return value of this method is a newly created object that must be deleted by the caller when it is no longer needed (or by using, e.g. std::auto_ptr<>).

3.3 Class OCPI::Container::Application

This is the class of objects that represent application contexts on a container. It is the owner of the workers it creates. If multiple applications are using a container, different objects of this class will exist in the container.

3.3.1 OCPI::Container::Application::createWorker method

This method creates a worker in an application context in a container. It is given the name of the artifact (usually a file), a PValue list associated with the use of that artifact (which is usually empty), the worker implementation name to instantiate out of that artifact, and an optional “instance name” if in fact the artifact has fixed, pre-created instances rather than dynamically creatable instances. Typically, in a software artifact like a shared object/dynamic library, the worker implementation will support dynamic

creation of as many worker instances as are required in the applications running in the container. However, some artifacts are statically configured with a specific set of named instances and thus the instance must be identified. This is common in FPGA bitstreams and in some statically linked DSP images.

```
namespace OCPI {
   class Worker;
   class Container { class Application {
      Worker &createWorker(const char *artifact,
                           PValue *artifactProperties,
                           const *implementation,
                           const *instance = NULL);
   };};
}
```

Note that the return value is a reference to a worker. The caller does not own this object and cannot delete it.

3.3.2 OCPI::Container::Application::~Application method

When the workers created in this application context are no longer needed, this context object can be deleted, which will in turn destroy all workers within it.

```
namespace OCPI { class Container { class Application {
   ~Application();
};};};}
```

3.4 Class OCPI::Worker

This is the class of objects that represent worker instances (objects instantiated based on worker implementations in artifacts). They are owned by the OCPI::Container::Application objects. They are not deleted directly, but are only destroyed when the OCPI::Container::Application is destroyed.

3.4.1 OCPI::Worker::getPort method

This method retrieves a reference to a port object representing one of the ports of the worker. It takes the string name of the port and returns a reference to that port. An exception is thrown if there is no port with that name.

```
namespace OCPI {
   class Port;
   class Worker {
      Port &getPort(const char *name);
   };
}
```

3.4.2 OCPI::Worker::setProperty method

This method sets a worker's property by name, providing the value in string form, which is then parsed and error checked according to the data type of the property. It should only be used in preference to the OCPI::Property class below, when performance is not important, since it has much higher overhead internally. If the value cannot be parsed

for the appropriate type, or there is no property with the given name, an exception is thrown.

```
namespace OCPI {
   class Worker {
      void &setProperty(const char *name, const char *value);
   };
}
```

3.4.3 OCPI::Worker::start method

This method starts the execution of the worker. It will continue to run until the "stop" method is called or until the application is destroyed.

```
namespace OCPI {
   class Worker {
      void start();
   };
}
```

3.4.4 OCPI::Worker::stop method

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

```
namespace OCPI {
   class Worker {
      void stop();
   };
}
```

3.4.5 OCPI::Worker::setProperty method

This method sets a worker's property by name, providing the value in string form, which is then parsed and error checked according to the data type of the property. It should only be used in preference to the OCPI::Property class below, when performance is not important, since it has much higher overhead internally. If the value cannot be parsed for the appropriate type, or there is no property with the given name, an exception is thrown.

```
namespace OCPI {
   class Port;
   class Worker {
      void &setProperty(const char *name, const char *value);
   };
}
```

3.5 Class OCPI::Port

This is the class of objects that represent a worker's ports. They are owned by the OCPI::Worker objects. They are not deleted directly, but are only destroyed when the OCPI::Container::Application is destroyed.

3.5.1 OCPI::Port::connect method

This method makes a connection between two ports of different workers that may or may not be in different containers. It configures the connection and enables messages to flow from one to the other. If the two ports have the same role (both are producer/user/client or both are consumer/provider/server), an exception is thrown. The port on which this method is invoked can be either a producer/user/client or a consumer/provider/server. A PValue list is provided to each side of the connection in order to provide configuration information about the connection that is not the default. The possible PValue types for connections (for either side) are:

PValue Name	Data Type	Default	Description
bufferCount	ULong	2	The number of buffers to allocate for this end of the connection
bufferSize	ULong	From Worker	The number of bytes per buffer
xferRole	String	optimized for "push"	Specify DMA transfer roles when the connection is crossing a fabric or bus. Can be: "passive", "active", "flowcontrol", or "activeonly"

```
namespace OCPI { class Port {
   void connect(PValue *myProperties,
                Port &otherPort,
                PValue *otherProperties);
};}
```

3.5.2 OCPI::Port::connectExternal method

This method makes a connection between ports of a worker in some container, and the control application itself. It allows messages to flow between the worker port and the control application. A 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 ExternalPort object is opposite from the role of the worker port object on which the method is called. The possible PValue types for these external connections are the same as the connect method above. This method returns a reference to an ExternalPort object that is used by the control application to, itself, produce or consume data.

```
namespace OCPI {
   class ExternalPort;
   class Port {
   ExternalPort& connectExternal(PValue *myProperties,
                                 const char *externalName,
                                 PValue *externalProperties);
};}
```

If the connection cannot be made or the PValue lists are invalid an exception is thrown.

3.6 Class OCPI::ExternalPort

This is the class of objects that represent a communication endpoint in the control application used to communicate with workers' ports. They are owned by the OCPI::Worker objects. They are not deleted directly, but are only destroyed when the OCPI::Container::Application is destroyed.

3.6.1 OCPI::ExternalPort::getBuffer method

This method is used to retrieve the next available buffer on an external port. It returns a pointer ExternalBuffer object, or NULL if there is no buffer available. Thus it is a non-blocking I/O call used by application control application. For external ports acting in the producer/user/client role, the returned buffer is a buffer to fill with a message to send. For external ports acting in the consumer/provider/server role, the returned buffer is a buffer that contains the next message that can be received/processed by the application. When the control application is done with the buffer if calls the "put" method (for sending/producing) or the "release" method (for discarding input buffers). In addition to returning the buffer object, the getBuffer method also returns (as output arguments by reference), the data pointer into the buffer and the length of the message (for input) or buffer (for output).

There are actually two overloaded getBuffer methods, for the two directions. The first, for getting a buffer filled with an incoming message, also returns the metadata for message (opCode and endOfData) in separate by-reference output arguments.

```
namespace OCPI {
   class ExternalBuffer;
   class ExternalPort {
      // Input: get a buffer filled with the next incoming message
      ExternalBuffer *getBuffer(uint8_t &data,
                                uint32_t &length,
                                uint8_t &opCode,
                                bool &endOfData);
      // Output: get a buffer to fill with the next outgoing message
      ExternalBuffer *getBuffer(uint8_t &data,
                                uint32_t &length);
   };
}
```

3.6.2 OCPI::ExternalPort::endOfData() method

This method, used only when the role of the port is producer/user/client, is used to indicate that no more messages will be send on this connection. This propagates an

out-of-band indication across the connection to the worker port. Note that this indication can also be made in the 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.

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

3.6.3 OCPI::ExternalPort::tryFlush() method

This method, used only when the role of the external port is producer/user/client, is used to attempt to “move data out the door”, when messages are locally buffered in this single threaded non-blocking environment. The return value indicates whether there are still messages locally buffered that will require further calls to tryFlush.

```
namespace OCPI { class ExternalPort {
      bool tryFlush();
};}
```

3.7 Class OCPI::ExternalBuffer

This is the class of objects that represent buffers attached to (owned by) external ports. They are returned (by reference return value) from the ExternalPort::getBuffer methods, and given back to the external port via the “put” method (for output) or the “release” method for input.

3.7.1 OCPI::ExternalBuffer::release() method

This is the method used to discard an input buffer after it has been processed/consumed by the control application.

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

3.7.2 OCPI::ExternalBuffer::put() method

This the method used to send an output buffer after it has been filled by the control application. The arguments specify the metadata associated with the message: (1) the opCode, (2) the length in bytes of valid message data, and (3) whether it is the last message to be sent (if that fact is known at the time of the call).

```
namespace OCPI { class ExternalBuffer {
      void put(uint8_t opCode, uint32_t length, bool endOfData);
};}
```

3.8 Class OCPI::PValue

This is the class of objects that represent 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 PValue) for each supported data type, which is the same set of types supported for worker configuration properties and protocol

operation arguments in the metadata associated with workers. Thus for each supported scalar data type, the name of the derived class is 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.:

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

3.9 Class OCPI::Property

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

3.9.1 OCPI::Property::Property constructor method

This constructor initializes the Property object such that it is specific to a worker and specific to a single named property of that worker.

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

Typical usage would be:

```
{
   OCPI::Worker &w = app->createWorker(..);
   OCPI::Property
      freq(w, “frequency”),
      peak(w, “peak”);

   freq.setFloatValue(5.4);
   float p = peak.getFloatValue();
}
```

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

3.9.2 OCPI::Property::set{Type}Value methods

There is a *set* method for each data type that a worker's property is allowed to have. 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 (i.e. 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.

```
namespace OCPI { 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);
};}
```

3.9.3 OCPI::Property::get{Type}Value methods

There is a get method for each data type that a worker's property is allowed to have. 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 (i.e. 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.

```
namespace OCPI { 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);
};}
```

3.9.4 OCPI::Property::set{Type}SequenceValue methods

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

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

3.9.5 OCPI::Property::get{Type}SequenceValue methods

There is a get sequence method for each data type that a worker's property is allowed to have. The get sequence methods are strongly typed and individually named. If the wrong get sequence method is used for a property (i.e. getULongSequenceValue for a property whose type of Float), an exception is thrown. If the string buffers in getStringSequenceValue (specified by the maxStringLength argument) are not long enough to hold the worker property's current string values, an exception is thrown. If there is an error accessing the worker's property value, an exception is thrown.

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

4 An Example of Using the ACI

This example uses an RCC worker called “copy” that copies its input to its output, and is compiled and linked in the file “workers.so”. The control application creates the worker, creates an external port to talk to its input port, and another external port to talk to its output port, sends a “hello” message, and expects to receive a “hello” message back.

```
namespace OCPI {
   Container &c = ContainerManager::find(“RCC”)
   Application *app = c.createApplication();
   Worker &w = app->createworker(“workers.so”, NULL, “copy”);
   Port
     &win = w.getPort(“in”),
     &wout = w.getPort(“out”);
   ExternalPort
     &myIn = win.connectExternal(“aci_out”),
     &myOut = wout.connectExternal(“aci_in”);
   w.start();
   uint8_t opcode, *idata, *odata;
   uint32_t ilength, olength;
   bool end;
   ExternalBuffer *myOutput = win.getBuffer(odata, olength);
   assert(myOutput && olength >= strlen(“hello”) + 1);
   strcpy(odata, “hello”);
   myoutput->put(0, strlen(“hello”) + 1, false);
   ExternalBuffer *myInput =
     wout.getBuffer(opcode, idata, ilength, end);
   assert(myInput && opcode == 0 &&
          ilength == strlen(“hello”) + 1 && !end &&
          strcmp(idata, “hello”);
   delete app;
}
```

5 Glossary

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

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

Configuration Properties – Named values 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. Configuration properties with side effects can be used for custom worker control. Each worker may have its own, possibly unique, set of configuration properties. They may include hardware resource such registers, memory, and state.

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.

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

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