OpenCPI Application Control Interface Specification

(ACI)

Revision History

Revision	Description of Change	By	Date
1.01	Creation	jkulp	2010-07-01
1.1	Edits, corrections, bring into consistency with implementation, move opcode arguments	Jkulp	2011-05-08
1.2	Add create-time worker properties, get stringified values, OCPI_LIBRARY_PATH etc.	Jkulp	2011-08-27

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, communicating with other workers in the application, or as input/output 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::API namespace (generally abbreviated OA::) to avoid any namespace collisions with user code

2 Overview

The OpenCPI execution framework for component-based applications is based on ***workers*** executing in ***containers***, communicating using their ***ports***, configured using their ***properties***. The ***workers*** are runtime instances of ***component*** implementations that are based on compiled and linked code in files represented by ***artifact*** objects. The term "component" represents the abstract functionality. The term "*artifact*" is used as a technology neutral term which represents a compiled binary file that is the resulting *artifact* of compiling (or for FPGAs, "synthesizing" etc.) and linking 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 (hence the term "shared library").

The ***application*** class is used as the lifecycle object that owns all the ***workers*** instantiated for the execution of the (component-based) application. Thus the interface described below only has lifecycle (create and destroy) control over the ***application*** classes and the ***container*** classes. All these classes (Container, Application, Artifact, Worker, Port, Property) are in the OCPI::API C++ namespace.

Container objects represent an execution environment for ***workers***. ***Container*** objects for software workers (RCC etc.) can be within the process of the control application using this API, or can represent some other execution environment (other process or other processing device) that is available to the OpenCPI implementation and installation (system). At the level of abstraction of this *application control interface*, the caller knows the name of the component to be executed, and then the specific implementation in a specific artifact is found by OpenCPI by looking for ***artifacts*** in component ***libraries*** that incorporate implementations of the named ***component***.

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

- Find ***containers*** on which the ***components*** of the application should execute.
- Establish an ***application*** context on each ***container*** in which workers can be created incrementally, and removed as a group based on that context.
- Create the ***worker*** runtime instances based on ***artifacts*** that are found by OpenCPI in some component library in the OpenCPI component library path, based on the type of container.
- 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" operation on the ***application*** objects.

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 (or simply exit, since they will all be automatically destroyed anyway).

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. All are in the OCPI::API C++ namespace. It is common practice to use an abbreviation for this namespace via the C++ statement:

```
      namespace OA = OCPI::API;
```

All the descriptions below will use the OA abbreviation.

3.1 Class OA::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 **OA::ContainerManager::find**

This *static* 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 behave the same, but control applications may need to find certain types of container to run certain types of workers. The arguments of **find** 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 two primary aspects are authoring model (RCC, HDL, etc), and a specific piece of hardware. Secondarily, there might be specific support for tool chains and operating environments. The ***find*** method has two overloaded versions. One simply has three arguments: the authoring model, the specific name of the container, and a PValue list for more specific options. The last two can be NULL for "don't care". The second method has an argument that is a property list using the PValue class that simply enumerates the characteristics of the container being sought.

```
class ContainerManager {
  static Container *
    find(const char *model,
         const char *which = NULL,
         const PValue *list = 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. Software containers are created on demand by name. Thus they are always “found” since they will be created if no one with the given name exists.

3.1.2 **OA::ContainerManager::shutdown**

This *static* 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). This method is only needed when resources should be recovered *before* the program exits. Normally a program can simply exit and OpenCPI’s container resources will be automatically recovered as appropriate.

```
class ContainerManager {
  static void shutdown();
};
```

3.2 Class OA::Container

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

3.2.1 **OA::Container::createApplication** method

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).

```
class Container {
  ContainerApplication *
    createApplication(const char *name = NULL,
                      const PValue *props = NULL);
};
```

The return value of this method is a newly created object that may be deleted by the caller when it is no longer needed (or by using, e.g. std::auto_ptr<>). This is only necessary if the container’s resources must be recovered before program exit. The *name* argument is useful in certain debugging/tracing scenarios, and the *props* argument specified properties of the application (none are currently defined). Both are optional.

3.3 Class OA::ContainerApplication

This is the class of objects that represent application contexts on a specific 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 **OA::ContainerApplication::createWorker** methods (2 overloaded)

These two methods are used to create a worker (instance) in an application context in a container. The simpler form is given the name of this worker instance within this application (nameWithinApp), and the name of the component that the worker should implement, followed a list of initial configuration property values. This version of createWorker searches the artifact libraries indicated by the OCPI_LIBRARY_PATH environment variable (containing colon-separated directory names), looking for an artifact containing a worker implementation that will execute on the container.

The second more complicated version has two more arguments at the beginning of the argument list to specify an artifact file, and a parameter list for using the artifact. It also has an extra argument to identify a particular instance in the artifact – when the artifact has fixed, static instances (like FPGA bitstreams or some statically linked DSPs).

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.

```
class Worker;
class Connection;
class ContainerApplication {
  Worker &createWorker(const char *nameWithinApp,
                       const char *component,
                       const PValue *properties = NULL,
                       const PValue *params = NULL,
                       const Connection *conns = NULL);
  Worker &createWorker(const char *nameWithinApp,
                       const char *component,
                       const PValue *properties = NULL,
                       const PValue *params = NULL,
                       const Connection *conns = NULL);
};
```

Note that the return value is a reference to a worker. The caller does not own this object and cannot delete it. The *nameWithinApp* argument simply identifies this worker within the application (i.e. when there are two workers in the application that perform the same function, they are both based on the same component, but will have different names within the application). The *component* argument identifies the component function that this worker will perform. This is used to find the implementation in some artifact in some library indicated in the OCPI_LIBRARY_PATH environment variable.

The *properties* argument contains instantiation properties of the worker as a whole (non are currently defined). The *conns* argument indicates an array of port connections that will be made later for this worker, to help the library search process choose the best component. These connections are simply defined by three strings: the port name on this worker, the component name of the “other” worker, and the port name of the “other” worker. The array of connection structures is terminated with a null port name.

3.3.2 **OA::ContainerApplication::start** method

This method starts all the workers in the OA::ContainerApplication. This is a convenience so that each worker doesn’t have to be individually started.

```
class ContainerApplication {
  start();
};
```

3.3.3 **OA::ContainerApplication::~ConnectionApplication** 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. This is only necessary if resources need to be recovered prior to program exit.

```
class ContainerApplication {
  ~ContainerApplication();
};
```

3.4 Class OA::Worker

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

3.4.1 **OA::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.

```
class Port;
class Worker {
  Port &getPort(const char *name
                const PValue *props = NULL);
};
```

The *props* argument specified port properties.

3.4.2 **OA::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, or the worker itself does not accept the property setting, an exception is thrown.

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

3.4.3 **OA::Worker::start** method

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

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

3.4.4 **OA::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.

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

3.4.5 **OA::Worker::getProperty** method

This method retrieves the name and stringified value of a property identified by its ordinal. This makes it very easy to introspect the properties of a worker without knowing their names or types. Property ordinals start at zero. The value retrieved into the referenced std::string is converted from its native type into a string. If the ordinal is invalid (greater or equal to the number of available properties), the return value is false..

```
class Worker {
  bool getProperty(unsigned ordinal,
                   std::string &name,
                   std::string &value);
};
```

Here is an example code that prints all a worker's properties using C printf:

```
OA::Worker &w;
std::string name, value;
for (unsigned n = 0; w.getProperty(n, name, value); n++)
  printf("Property %u has name \"%s\" and value \"%s\"\n",
         name.c_str, value.c_str());
```

3.4.6 **OA::Worker::setProperties** method

This method sets multiple configuration values at once, although this method is not as fast as the methods using the **Property** class below. It takes a PValue list that expressed the names, types, and values of the properties to be set.

```
class Worker {
  void setProperties(const PValue *props);
};
```

3.5 Class OA::Port

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

3.5.1 **OA::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"

```
class Port {
  void connect(Port &otherPort,
               const PValue *myProperties = NULL,
               const PValue *otherProperties = NULL);
};}
```

3.5.2 **OA: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.

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

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

3.6 Class OA::ExternalPort

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

3.6.1 **OA:ExternalPort::getBuffer** method

This method is used to retrieve the next available buffer on an external port. It returns a pointer to an ExternalBuffer object, or NULL if there is no buffer available. Thus it is a non-blocking I/O call used by the 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, it 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.

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

3.6.2 **OA::ExternalPort::endOfData** method

This method, used only when the role of the external 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.

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

3.6.3 **OA::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.

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

3.7 Class OA::ExternalBuffer

This is the class of objects that represent buffers attached to (owned by) external ports. They are returned (by pointer 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 **OA::ExternalBuffer::release** method

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

```
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 length in bytes of valid message data, (2) (optional) the opcode of the message, and (3) (optional) whether it is the last message to be sent (if that fact is known at the time of the call).

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

3.8 *Class OA::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.:

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

3.9 *Class OA::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 very 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 **OA::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.

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

Typical usage would be:

```
{
  OA::Worker &w = app->createWorker(..);
  OA::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 **OA::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.

```
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 **OA::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.

```
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 **OA::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.

```
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);
};
```

3.9.5 **OA::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.

```
class Property {
  void
    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,
                           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, linked and in a directory accessible via OCPI_LIBRARY_PATH. 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.

```
#include “OcpiApi.h”
namespace OA = OCPI::API;
int main(int, char**) {
  OA::Container &c = OA::ContainerManager::find(“rcc”);
  OA::ContainerApplication *app = c.createApplication();
  Worker &w = app->createworker(“mycopy”, “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(strlen(“hello”) + 1);
  ExternalBuffer *myInput =
    wout.getBuffer(idata, opcode, ilength, end);
  assert(myInput && opcode == 0 &&
         ilength == strlen(“hello”) + 1 && !end &&
         strcmp(idata, “hello”);
  return 0;
}
```

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 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 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.

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