OpenCPI RCC Authoring Model Reference

a.k.a.
How to write an OpenCPI C Language worker

Authors: Jim Kulp, John Miller

Revision History

Revision	Description of Change	By	Date
1.01	Creation from previous “CP289” and other documents to be consistent with other OpenCPI authoring model specifications and share a base document.	jkulp	2010-04-01
1.02	Clarifications on port and buffer states, additional dynamic error string function. Add correct description of “top level sequence of fixed elements”. Add protocol operation enumerations and message structure packing.		2012-10-01

Table of Contents

1 References

This document depends on several others. Primarily, it depends on the "OpenCPI Generic Authoring Model Reference Manual", which describes concepts and definitions common to all OpenCPI authoring models. Since this RCC authoring model is based on the C language (specifically C90: ISO/IEC 9899:1990), it also depends on the ISO-C language reference manual and associated libraries. The exceptions to the C90 basis are the use of <stdint.h> from C99.

Table 1 - Table of Reference Documents

Title	Published By	Link
OpenCPI Generic Authoring Model 1.01	OpenCPI	http://www.opencpi.org/doc
ISO C Language Specification	C Language	Public URL: http://www.iso.org

2 Introduction

2.1 Purpose

The purpose of this document is to define the OpenCPI RCC Authoring Model. The term "RCC" is defined as "Resource-constrained C Language". This model is based on the C language and makes most design choices to minimize resources and thus to be appropriate for resource-constrained embedded systems. DSP processors with on-chip memories, microcontrollers, and some multi-core processors are natural targets for this authoring model. Of course the RCC model is a perfectly appropriate model for any general-purpose processor with a C compiler, when the developer is comfortable with the constraints of the C language. Workers written to this model tend to be smaller and faster than in any other model suitable for general-purpose processors. The basis of this specification is the OpenCPI Generic Authoring Model Reference (AMR) that defines concepts and metadata common to all OpenCPI authoring models. That document is a prerequisite for this one.

This document will introduce and describe the interfaces between the worker and its environment, and the semantics of those interfaces. It also defines the RCC OWD – the XML metadata that describes an RCC worker. It should enable the author of an RCC worker to write the code and to write the OWD for the RCC worker.

2.2 Requirements

- Independent from Specific Environments (operating systems or compilers)
- C90 compliant (plus stdint.h from C99)
- Must support compliance with the OpenCPI Generic Authoring Model Reference Manual (AMR)

2.3 Goals

This document defines, for OpenCPI RCC developers, *which* choices can be made by the RCC worker designer about Control, Data, and Local Service interfaces, *how* those choices are described, and how the code must be written to interface the worker to its environment. The intent is to be precise and complete.

2.3.1 Non-Goals

This document is not intended as a usage tutorial for RCC developers. It is a reference specification. It is recognized that separate documents for application developers are needed to reduce the learning and knowledge required to use this model.

2.4 Overview

RCC workers are hosted in a *container*, which is responsible for (1) loading, executing, controlling, and configuring the worker, (2) effecting data movement to and from the data ports of the worker, and (3) providing interfaces for the local services available to RCC workers

In a multi-processor system, there is one or more container(s) per processor, each with the same common API but with its implementation specialized to the type of device and the fabrics or networks to which it is connected.

When RCC workers are collocated, the container can make use of zero-copy to move data between them. For connections between workers in different containers, the containers move data between each other using some common messaging protocol. Containers can make use of the optimum data transport between the two devices.

There are two sub-profiles for RCC workers to service two design tradeoffs. Most RCC requirements are common to both profiles. There is a *base* profile, and an optional *multithreaded* profile that is a superset of the *base* profile. When a requirement is specific to the *multithreaded* profile, it will be identified as such. Thus RCC workers are written to require (from the container) and use only the *base* profile, or the multithreaded profile. The RCC worker's OpenCPI Worker Description (OWD) will indicate which profile is used.

- The *base* profile requires a minimal run-time infrastructure (and no threads) and places certain restrictions on worker implementations. It enforces a Finite State Machine (FSM) or fully event-driven coding style that does not always allow easy reengineering from legacy code in the more conventional sequential, blocking style.

  This profile is appropriate for simple worker implementations that can be easily coded this way, and for highly resource-constrained processing devices with little operating system and middleware support. It generally provides minimum footprint and maximum performance.

- A *multithreaded* profile, with a richer run-time infrastructure with fewer restrictions placed on the worker. A more common multithreaded and blocking I/O coding style is supported. Workers may create threads and make blocking calls to wait for I/O buffers.

  This profile is appropriate when porting code written in a multi-threaded/sequential style, or when the worker's behavior requires more capable operating system and middleware support.

An RCC container implementation may choose to support only the base profile, or the larger multithreaded profile. Any RCC container implementation that supports the multithreaded profile will also support the simpler base profile.

A worker may be written to either profile. There may be separate worker implementations for both based on the same OCS.

Worker examples for both profiles, illustrating both conventional and FSM coding styles, are presented in section XYZ below. All RCC worker code, irrespective of profile, must be re-entrant since the same code may be run in multiple threads for different worker instances.

The following sections define these aspects of the RCC authoring model:

- the execution model: how the container executes the RCC worker
- the local services: how the worker uses local services and which ones are available
- container-to-worker interfaces: how the container calls the worker's entry points
- worker-to-container interfaces: how the worker calls the container's entry points

3 RCC Execution Model

In the base profile for RCC workers, the container executes workers such that::

- All execution threads are supplied by the container so there is no need for workers to create threads.

  Rationale:

  - *reduces complexity of component code, eliminates requirement to support a thread capability*
  - *allows container to decide whether multithreading is even needed - frequently it is not.*
  - *worker code is reusable between threaded and non-threaded containers.*
  - *container can be interrupt driven, single stack multi-tasking, or multithreaded*
- Execution of a worker is enabled when the worker is in the “operating” state, as defined above, and some combination of its ports are “ready”, or time has passed.
  A “run” method is called when ports are ready or time has passed.
  Worker execution is a series of “runs” initiated by container logic.

  Rationale:

  - *simplify component code, eliminate any "wait for next thing" logic or loops*
  - *make run decisions centralized in the container, fast, simple and small*
  - *make worker multiplexing within a container able to be more efficient than multithreading*
- Workers declare a “run condition” as a simple logical “or” of a small number of masks.
  Each mask indicates a logical “and” of “port readiness bits” (1 << port ordinal).
  An input port is ready when there is an input message for the worker to look at.
  An output port is ready when there is an output buffer that can be filled by the worker.
  The default run condition is one mask of all ones, meaning "run me if all my ports are ready". If a port being ready is not required for the run condition, the worker can still test its readiness. Otherwise it can assume the readiness of ports indicated in its run condition (except for optionally connected ports). A timeout is also provided in the run condition.

  Rationale:

  - *simplest model for specifying when to run components with multiple ports; common cases are trivial, minimal code, container can optimize synchronization, and minimize worker code.*
- Workers indicate which ports should “advance” at the end of each run, before returning. Advancing a port means releasing (input) or sending (output) the current buffer and implicitly requesting another one.

  Rationale:

  - *most efficient way to indicate this since it piggybacks on the “return”.*

The RCC “worker interface” is called by the container, and implemented by the worker. The worker provides its methods (entry point functions) to the container . A “worker context object” is provided to the worker by the container as the first argument to each worker method, analogous to the implied “this” argument to object methods in C++. This context object is a data structure with some advertised structure members (rather than an explicit structure definition: this is similar to the way POSIX defines public structure members in C APIs, also analogous to public member data in C++). This structure also provides the worker with access to port-specific state, any worker-requested memory resources, and the data structure containing its configuration properties.

The profile (base vs. multithreaded) affects the Worker interface in two major ways (in addition to the expanded AEP in the multithreaded profile).

3.1 Execution based on events

In the base profile, the container calls the worker's “run” method when the worker’s “run condition” is true, based on the availability of input buffers (with data) or output buffers (with space) or the passage of time. The worker’s run method executes, processes any available inputs and outputs, indicates when messages are completely consumed as input or produced as output, makes any changes to the “run condition”, and returns. Workers never block. The container conveys the messages in buffers between collocated workers as well as into and out of the container as required by the application assembly’s connections.

In the multithreaded profile, in addition to calling the “run” method if the “run condition” is true, the worker may also execute in threads it creates, and use a blocking API to wait for input or output buffers.

3.2 Sending or receiving messages via ports

In the base profile, the worker can only indicate that an input buffer has been processed and/or an output buffer has been filled in one execution of the run method. It must return from the run method in order to wait for more buffers to process. The worker never blocks.

In the multithreaded profile, the worker may explicitly wait for buffers to become available on individual ports, via calls to the container that block if no buffers are available.

3.3 Buffer management

The Worker Interface is designed to have the container provide and manage all buffers. Thus input ports operate by the container providing buffers to the worker filled with incoming messages, and the output buffers operate by the container providing buffers for the worker to fill with messages before being sent. Output buffers are either:

- obtained for a specific output port (since they may be in a special memory or pool specific to an particular output hardware path), *or*
- originally obtained from an input port and passed to output ports, possibly with no copying.

Logically, there are three operations performed with buffers, which is the basis for the specific APIs defined later. They are all non-blocking functions:

- Request that a new buffer be made available. For an input port, it is filled by the container with a new input message. For an output port, it is to be filled by the worker with a new output message. In both cases the ownership of the buffer passes from container to worker when it becomes available. The new buffer may or may not be immediately available.
- Release a buffer to be reused, with its contents discarded. The ownership passes from worker to container. Input buffers must be released (or sent) in the order received, i.e. ownership of input buffers must be passed from worker to container in the order that ownership was given from container to worker.
- Send (enqueue) a buffer on an output port, to be automatically released (later) after the data is sent. The ownership passes from worker to container. If the buffer was originally obtained from an input port, it must be sent or released in the order received.

The concept of “current buffer” of a port exists to support a convenient model for workers that need no special buffer management. A port is *ready* if it has a current buffer. A current buffer on an input port is available to read data from. A current buffer on an output port is available to write data into. The concept of “advancing” a port, is simply a combination of releasing (input) or sending (output) the current buffer of the port, and requesting a new buffer to be made available on that port, to become the current buffer when it becomes available in the future. So simple workers wait for ports to be ready (to have buffers), process input buffers into output buffers, advance input and output ports, and return. The interface is designed to make this common model as simple as possible. Workers wait for some ports to be ready, and advance ports after processing buffers.

Several more advanced buffer management requirements are supported by adding extra capabilities for certain situations as needed:

- To support sliding window algorithms, allow the worker to own previous buffers (not releasing them), while new ones are requested; i.e. allow selective (in order) explicit input buffer release, not implicitly just the most recent buffer obtained. The worker must still release the buffers in the order received.
- To support zero copy from input ports to output ports, allow the worker to send a buffer obtained from an input port to an output port, and not require an empty current buffer to fill on the output port. Such buffers must be sent (or released) in the order received.
- To support port-specific callbacks without any worker dispatching overhead, allow the worker to associate callback functions with each port that convey new buffers from container to worker.
- To support blocking semantics in the multithreaded profile, allow for blocking calls to wait for an input buffer with new data on an input port, or wait for a new output buffer to fill on an output port.

The above features are only needed in certain cases, and can be ignored entirely for most simple workers. Supporting them results in the following interface features:

- Provide separate non-blocking interfaces for releasing, sending, requesting buffers.
- Provide an interface to wait for a buffer on a port.
- Provide for per-port worker-supplied callback functions.

4 The RCC Worker Interface

The methods in the interface described below are as if it was a C++ or IDL interface, but it is a C interface, with an initial explicit argument to each method that provides a worker context structure, analogous to the implicit "this" argument in C++.

Containers shall use, and RCC Workers shall implement, the following Worker C-language interface, defined in a header file named “RCC_Worker.h”, and reserve the prefix “RCC” for compile-time constants and types defined in this file:

The Worker interface consists of control operation methods whose behavior is defined in the AMR, plus an additional ***run*** method supports the event-driven execution model defined above. Only ***run*** is required in the base profile. Only ***start*** is required in the multithreaded profile. All the rest are optional. All processing of the worker occurs in the context of these operations unless, in the multithreaded profile, the worker creates its own threads. All operations take a pointer to an ***RCCWorker*** structure to provide a reentrant context to the worker (a C++ "this"), and to enable multiple instances of the same worker to coexist in a container (in the same linkage address space).

All defined data types use the common prefix “RCC”. *There are no defined external symbols* except those in the RCC AEP defined above.

The interface uses several basic integer types consistent with their CORBA C/C++ language mapping, to provide some compiler independence. The integer types are defined using the ISO C 99 <stdint.h> types. These basic types are: uint16_t and uint32_t. The type RCCBoolean is aliased to uint8_t, to be consistent with the defined size in property space and message content layouts. The RCCOrdinal type is an alias for uint16_t, and is used when ordinals are required (ports, operations, exceptions, properties).

4.1 RCC Interface Data Types

4.1.1 RCCResult

The RCCResult type is an enumeration type used as the return value for all worker methods. It indicates to the container what to do when the worker method returns, as follows:

- *RCC_OK: worker operation succeeded without error*
- *RCC_ERROR: operation did not succeed, but error is not fatal to worker or container, thus the operation may be retried.*
- *RCC_FATAL: worker should never run again; it is non-functional; the container or other workers may be damaged; the worker is in an **Unusable** control state as defined in the AMR. Note that the container may know that it, or other workers are protected from damage, but the portable worker indicates this condition in case there is no such protection.*
- *RCC_DONE: worker needs no more execution; a normal completion. See usage below.*

- *RCC_ADVANCE: worker is requesting that all ports be advanced (run method only).*
- *RCC_ADVANCE_DONE: worker is requesting that all ports be advanced (run method only) and declaring that it is also "done".*

These result values apply to each method as defined in their specific behavior. Some values are not valid results for all methods. When the result is RCC_ERROR or RCC_FATAL, the worker may set the *errorString* member of the RCCWorker structure to non-NULL to add a string description to the error. See the RCCWorker type below.

4.1.2 RCCPortMask

The RCCPortMask type is a bit mask used to indicate a boolean value for each port. It is an integer type where a given bit being == 1 (1L << port_ordinal) means the value for that port is TRUE.

4.1.3 RCCRunCondition

The RCCRunCondition structure type holds the information used by the container to determine when it is appropriate to invoke the *run* operation of a worker. The defined members are always written by the worker, and never by the container.

RCCRunCondition Member Name	**Member Data Type**	**Member Description**
portMasks	RCCPortMask *	A pointer to a zero-terminated array of port masks, each of which indicates a bit-mask of port readiness. The run condition is considered "true" when any of the masks is "true" (logical OR of masks). A mask is "true" when all indicated ports are ready (logical AND of port readiness). A port is indicated by its bit being set (1 << port_ordinal). If the pointer is NULL, it indicates no masks and always ready. A bit set for an unconnected port is ignored, thus the default run condition can be used with unconnected ports.
timeout	RCCBoolean	Indicates that the *usecs* member is used to determine when enough time has passed to make the run condition true. Thus this value can used to enable or disable the timeout, without changing *usecs*.
usecs	uint32_t	If this amount of time has passed (in microseconds) since the run operation was last entered, the run condition is true.

Thus the overall run condition is the logical OR of the portMasks and the timeout. If the portMasks member is NULL (as opposed to there being no masks in the list), it indicates that no port readiness check is performed and thus the run condition is always TRUE.

Here are typical combinations:

Port mask	Timeout	RunCondtion Description
portMasks == NULL	ignored *	The NULL portMasks indicates that the worker is always ready to run, and the timeout is ignored.
portMasks != NULL	False	Run condition is TRUE when any mask is true. If there are no masks (portMasks[0] == 0), then the run condition can never be true.
portMasks != NULL and portMasks[0] != 0	True	Run condition is TRUE when any mask is true OR if the timeout expires. Timeout will take effect if time passes without the masks being satisfied
portMasks != NULL and portMasks[0] == 0	True	Since portMasks[0] == 0, port masks can never be true, thus this establishes period execution independent of port readiness.

The structure is completely defined with ordered members defined to enable static initialization:

```
typedef struct {
    RCCPortMask *portMasks;
    RCCBoolean  timeout;
    uint32_t    usecs;
} RCCRunCondition;
```

4.1.4 RCCMethod

The RCCMethod type is simply a convenience type declaration to declare worker methods (in C) that do not have any arguments other than the RCCWorker structure. It declares a function type that takes a worker's context structure as the single input argument, and returns an RCCResult value. It is defined as follows:

```
typedef RCCResult RCCMethod(RCCWorker *this);
```

4.1.5 RCCRunMethod

The *RCCRunMethod* type specifies the signature of the worker's *run* method. It is used in the definition of the *RCCDispatch* type, and can be used as a convenient way to forward-declare the method function for initializing the *RCCDispatch* structure. The method is described below. The type is defined as follows:

```
typedef RCCResult RCCRunMethod(RCCWorker *this,
                               RCCBoolean timedout,
                               RCCBoolean *newRunCondition);
```

4.1.6 RCCPortMethod

The *RCCPortMethod* type specifies the signature of functions that the worker can supply for any port, to request that the container call that individual function when a buffer is available for a particular port, or other port conditions occur. This is an alternative to having the container call the *run* method based on run conditions involving *all* ports. It would normally be used to process the current buffer on a port (to process data in an input message, or to produce data for an output message).

The arguments indicate which port, and why the callback was called. The *reason* argument is set by the container as follows

- RCC_OK: buffers are now available
- RCC_ERROR: data or messages were lost
- RCC_FATAL: connectivity was lost.

The *port* argument allows one callback function to be used for multiple ports. The return value indicates RCC_OK, RCC_ERROR, or RCC_FATAL.

The type is defined as follows:

```
typedef RCCResult RCCPortMethod(RCCWorker *this,
                                RCCPort *port,
                                RCCResult reason);
```

4.1.7 RCCPortInfo

The RCCPortInfo structure type holds the static port-related information needed by the container. It is supplied statically by the worker if the port has non-default port attributes. An array of these structures, terminated by the "port" member being RCC_NO_ORDINAL, is referenced by the "portInfo" member of the RCCDispatch structure defined below. . If all the attributes have default values, there is no need for a structure to be included for a given port.

RCCPortInfo Member Name	Member Data Type	Member Description
port	RCCOrdinal	The port that this structure describes, using the port enumeration described in 5.4.1.
maxLength	uint32_t	The maximum size in bytes of messages for this port. Default is 0, meaning there is no maximum size. Supplying messages larger than this at an input port is not allowed. For output ports, the worker will not need to produce any messages exceeding this size.
minBuffers	uint32_t	Indicates the minimum number of buffers that the worker requires to be available at this port. The default is 1. Zero means 1.

4.1.8 RCCDispatch

This type is the "dispatch table" for the operations of the worker interface. It represents functionality a worker provides to a container when it is loaded. The container must gain access to this structure when the worker is loaded and executed. All members are initialized by the worker source code. This structure also contains other descriptive information required by the container to use the worker.

The members of this structure are defined as follows:

RCCDispatch Member	Member Data Type	Member Description
Members below enable the container to check the metadata vs. the actual compiled description.		
version	uint32_t	Initialized from the RCC_VERSION macro to tell the container the version of RCC API (header files) used, to check mismatches.
numInputs	uint16_t	The number of input ports.
numOutputs	uint16_t	The number of output ports.
propertySize	uint32_t	The size in bytes (sizeof) of the property structure.
memSizes	uint32_t*	This zero-terminated array of memory sizes indicates a set of memory allocations required by the worker. Multiple allocations are allowed so that the worker need not aggregate all its requirements in a single allocation when it doesn't need to. This pointer may be NULL, indicating no such allocations are required.
threadProfile	RCCBoolean	Initialized to allow the container to error check
Members below supply the container with function pointers to methods.		
initialize	RCCMethod*	A pointer to the initialize method function. May be NULL if there is no functionality.
stop	RCCMethod*	A pointer to the stop method function. May be NULL if there is no functionality.
start	RCCMethod*	A pointer to the start method function. May be NULL if there is no functionality in the base profile. Required in the multi-threaded profile.
release	RCCMethod*	A pointer to the release method function. May be NULL if there is no functionality.
afterConfigure	RCCMethod*	A pointer to the test method function. May be NULL if there are no notify properties.
beforeQuery	RCCMethod*	A pointer to the test method function. May be NULL if there are no notify properties.
test	RCCMethod*	A pointer to the test method function. May be NULL if there are no tests.
run	RCCRunMethod*	A pointer to the run method function. May be NULL.
Members below supply the container with behavioral attributes		
runCondition	RCCRunCondition*	The initial run condition used. If this pointer is NULL it implies a run condition of all connected ports being ready and no timeout. If there are no ports, the default is no port masks to check, indicating always ready to run.
portInfo	RCCPortInfo *	A pointer to an array of RCCPortInfo structures, which describe static non-default port information to the container. Is NULL if no such non-default information is needed.

optionalPorts	RCCPortMask	A mask indicating ports that may be unconnected. The default, 0, means that all ports must be connected before the worker is started. The container enforces this.
portInfo	RCCPortInfo*	Points to an array of port descriptions for non-default port attributes. The pointer may be NULL, and if not, is terminated by an RCCPortInfo structure with the *port* member == RCC_NO_ORDINAL..

The portable worker source code shall provide an external symbol for its statically initialized *RCCDispatch* structure, and the development flow for the device will arrange for the container to find it.

There are several implementation-dependent approaches for the container to obtain the location of the *RCCDispatch* structure (which approach is used is determined by the development flow specified by the container for its workers), these include:

- Static linking of the worker with the container, using external symbols for worker implementation dispatch tables.
- Using the default (unnamed) entry point in the loadable worker code as the address of a function that returns the *RCCDispatch* structure or the address of the structure itself.
- Specifying the name of a symbol identifying the dispatch table when the worker is created.
- Providing a standard C “main” entry point which the container calls and which returns a pointer to the dispatch table.

Since the technique is determined by the platform, the developer of a portable implementation must only supply the dispatch table external symbol to the platform-specific build/compile tools/scripts.

In order to statically initialize this structure, the tagged member initialization syntax from the C99 standard should be used, e.g.:

```
RCCDispatch xyz = {
   .version = RCC_VERSION,
   .numInputs = XYZ_N_INPUT_PORTS,
   .numOutputs = XYZ_N_OUTPUT_PORTS,
   .threadedProfile = TRUE,
   .start = startXyz,
} RCCDispatch;
```

4.1.9 RCCBuffer

The RCCBuffer type is a structure that holds information about a buffer. It has the members described in the table below, all of which are written by container functions and not by the worker. There may be other undocumented members.

RCCBuffer Member Name	Member Data Type	Member Description
data	void * const	A const pointer to the data in the buffer. When this member of the structure is NULL, there is no buffer.
maxLength	const uint32_t	The maximum number of bytes that may be placed in the buffer (maximum message length). This is *not* the actual length of the valid data in the buffer.

4.1.10 RCCPort

The *RCCPort* type is a structure that contains the current state of a worker's port. The container is free to define this structure with any content and any ordering as long as the documented members are supported. (This style of structure standardization is from POSIX). The members are written by either the container or the worker, but never both.

Members written by the container are declared "const" to enhance error checking when compiling worker implementations. For code readability and reliability, there is a union in the structure that aliases two substructures with members used differently for input ports and output ports. Members used for input ports only are prefixed with "input.". Members used for output ports only are prefixed with "output." Members written by the worker maintain their value so if the worker only writes one value during execution, it can be set once (e.g. at "start" or "initialize"). All members written by the worker are initialized to zero/NULL unless otherwise noted.

RCCPort Member Name	Member Data Type	Written by	Member Description
current	const RCCBuffer	container	The current buffer to use for the port. The actual format of the message in the buffer depends on which request, response, or exception message is represented. The value of the *data* member is NULL if there is no current buffer. Port readiness is equivalent to the current.data != NULL.
input.length	const uint32_t	container	For input ports, the actual number of bytes in the current buffer (message length).
input.u.operation	const RCCOperation	container	For input ports associated with provider component ports, the operation ordinal of the request message in the current buffer.
input.u.exception	const RCCException	container	For input ports associated with "uses" component ports, the exception ordinal of the response message in the current buffer. Zero indicates no exception.
output.length	uint32_t	worker	For output ports, the number of bytes in the message the worker has placed in the current output buffer. Initialized to maxLength.

output.u.operation	RCCOperation	worker	For output ports associated with “uses” ports, the operation ordinal of the request message in the current buffer. Initialized to zero.
output.u.exception	RCCException	worker	For output ports associated with “provides” ports, the exception ordinal of the response message in the current buffer. Zero indicates no exception. Initialized to zero.
callback	RCCPortMethod*	worker	Per port callback function optionally set by worker, initialized to NULL by container. If not NULL, called by container when port becomes ready or when an error on the port occurred. Initialized to NULL.
maxLength	const uint32_t	container	The maximum length of buffers at this port. For input ports with protocols that are bounded (all operations are bounded), this will be the maximum message length that can arrive.

The worker must set the non-const "output" members prior to indicating to the container that the output port should be advanced. The worker writes the "callback" member (to non-NULL) in the *initialize* or *start* operation. The callback member is examined by the container after the start *operation* returns, and after subsequent callback functions on the port return.

4.1.11 RCCContainer

This structure provides the worker with a set of container functions. It is a dispatch table in the RCC*Worker* structure rather than a set of external symbol definitions, for the following reasons:

- Preserves the entire RCC worker interface with no external symbols other than the standard AEP. This enables optimal incremental linking and loading scenarios.
- Enables container functions to be varied per worker by the container to enable certain optimizations based on worker properties or behavior or port configurations.
- Preserves an object-oriented style by treating container functions as a "base class".
- Simplifies symbol namespace used by RCC: no non-AEP link-time symbols.

The entire structure is const, and initialized by the container. The use of these container methods is optional, and entirely unneeded for simple workers that only use run conditions and the *run* method.

RCC-Container Member	Member Data Type	Member (function) Description
release	void (*) (RCCBuffer *buffer)	Release a buffer for reuse. If the buffer is a current buffer for a port, it will no longer be the current buffer. Buffer ownership passes back to the container. Non-blocking. Must be done in the order obtained, per port. Releasing the current buffer does *not* imply that more buffers are requested. The "request" or "advance" functions do that.
send	void (*) (RCCPort *port, RCCBuffer *buffer, RCCOrdinal op, uint32_t length)	Send a buffer on an *output* port. If the buffer is a current buffer for a port, this buffer will no longer be the port's current buffer, and a "request" on that port is implied. The "op" argument is an operation or exception ordinal. Buffer ownership passes back to the container. Non-blocking.
request	RCCBoolean (*) (RCCPort *port, uint32_t max)	Request a new buffer be made available as the current buffer on a port. An optional (non-zero) length may be supplied. If the port already has a current buffer, the request is considered satisfied. The return value indicates whether a new current buffer is available. Non blocking.
advance	RCCBoolean (*) (RCCPort *port, uint32_t max)	Release the current buffer *and* request that a new buffer be made available as the current buffer on a port. An optional (non-zero) length may be supplied. This is a convenience/efficiency combination of release-current-buffer and request-more-buffers. The return value indicates whether a new current buffer is available. Non blocking.
wait	RCCBoolean (*) (RCCPort *port, uint32_t max, uint32_t usecs)	Block the caller until there is a current buffer for a port, implying a request for a new buffer. It will timeout after *usecs* microseconds, returning TRUE on timeout, FALSE otherwise. Only available in the *multithreaded* profile.
take	void(*) (RCCPort *port, RCCBuffer *releaseBuffer, RCCBuffer * takenBuffer)	Take the current buffer from a port, placing it at *takenBuffer. If releaseBuffer != NULL, first release that buffer. Non-blocking. Ownership is retained by the worker. The current buffer taken is no longer the current buffer. Used when the worker needs access to more than one buffer at a time from an input port. A *request* for more buffers is implied.
setError	void (*) (const char *fmt, …)	A printf-style function that sets an error string for the worker when it returns RCC_ERROR or RCC_FATAL. Similar to, but different than setting the errorString member of RCCWorker.
time	RCCTime (*)()	Return current time of day in RCCTime format.

The worker source code would call these container functions based on the "this" argument, like:

```
this->container.wait(port, usecs);
this->container.release(id);
// or
RCCContainer *c = &this->container;
c->wait(port, usecs);
c->release(id);
```

This approach was taken both for brevity and to avoid any external symbolic linkage for this interface.

4.1.12 RCCWorker

This structure type (typedef) represents the visible state of a worker. It is passed by reference to every operation, analogous to the implicit “this” argument in C++. The container is free to define this structure (in RCC_Worker.h) with any content and any member ordering as long as the documented members are supported. (This style of structure standardization – defined members but no explicit standardized structure definition or member ordering, is from POSIX). The members are written by either the container or the worker, but not both. Members written by the container are declared "const" to enhance error checking when compiling worker implementations.

RCCWorker Member Name	**Member Data Type**	**Written by**	**Member Description**
properties	void * const	container	A const pointer to the properties structure for the worker, whose layout is implied by the properties of the implementation. The value may be NULL if there are no such properties.
memories	void * const []	container	An array of const pointers to the memory resources requested by the worker in allocation properties in its implementation PRF. Any memories that are not read-only are initialized to zero before a worker executes any method.
container	const RCCContainer	container	A dispatch table of container functions.
runCondition	RCCRunCondition*	worker	Initialized from the RCCDispatch runCondition member. Checked by container after calling the start method.
errorString	char *	worker/ container	Initialized to NULL by the container. Optionally set by worker before returning RCC_ERROR or RCC_FATAL. Reset to NULL by container after worker returns. It is assumed to be static, not heap allocated. See the setError container function for dynamically created error strings.
connectedPorts	RCCPortMask	container	A mask indicating which ports are connected. A worker can check this to know whether an optional port is connected.
ports	RCCPort []	varies by member	An array of RCCPort structures defined above, indexed by port ordinals derived from the SCD. This array is the last member, and thus makes this structure variable length.

4.2 Methods called by container, implemented by worker

4.2.1 initialize

This method implements the OpenCPI *initialize* control operation. See AMR for rationale and behavior.

4.2.1.1 Synopsis.

```
RCCResult initialize(RCCWorker *this);
```

4.2.1.2 Returns.

This method shall return an RCCResult value.

4.2.1.3 Exceptions/Errors.

If the initialization cannot succeed, it shall return RCC_ERROR. If the worker detects an error that would disable the implementation or its environment, it shall return RCC_FATAL. Otherwise it shall return RCC_OK if normal worker execution should proceed.

4.2.2 start

This method implements the OpenCPI *start* control operation. See AMR for rationale and behavior.

4.2.2.1 Synopsis.

```
RCCResult start(RCCWorker *this);
```

4.2.2.2 Returns.

This method shall return an RCCResult value.

4.2.2.3 Exceptions/Errors.

If the *start* method cannot succeed, it shall return RCC_ERROR. If the worker detects an error that would disable the implementation or its environment, it shall return RCC_FATAL. Otherwise it shall return RCC_OK if normal worker execution should proceed. It shall return RCC_DONE, if no worker execution should proceed. Returning RCC_DONE indicates to the container that the worker will never require execution, and provides this advice to the container to allow the container to take advantage of this fact and possibly release resources (such as I/O buffers) prior to the worker being destroyed or reused.

4.2.3 stop

This method implements the OpenCPI *stop* control operation. See AMR for rationale and behavior.

4.2.3.1 Synopsis.

```
RCCResult stop(RCCWorker *this);
```

4.2.3.2 Returns.

This method shall return an RCCResult value.

4.2.3.3 Exceptions/Errors.

If the *stop* method cannot succeed, it shall return RCC_ERROR. If the worker detects an error that would disable the implementation or its environment, it shall return RCC_FATAL. Otherwise it shall return RCC_OK.

4.2.4 release

This method implements the OpenCPI *release* control operation. See AMR for rationale and behavior.

4.2.4.1 Synopsis.

```
RCCResult release(RCCWorker *this);
```

4.2.4.2 Returns.

This method shall return an RCCResult value.

4.2.4.3 Exceptions/Errors.

If the *release* method cannot succeed, it shall return RCC_ERROR. If the worker detects an error that would disable the implementation or its environment, or it is unable to return resources it allocated, it shall return RCC_FATAL. Otherwise it shall return RCC_OK.

4.2.5 afterConfigure

This method implements the OpenCPI *afterConfigure* control operation. See AMR for rationale and behavior.

4.2.5.1 Synopsis.

```
RCCResult afterConfigure(RCCWorker *this);
```

4.2.5.2 Returns.

This method shall return an *RCCResult* value.

4.2.5.3 Exceptions/Errors.

If the *afterConfigure* method cannot succeed, it shall return RCC_ERROR. If the worker detects an error that would disable the implementation or its environment, it shall return RCC_FATAL. Otherwise it shall return RCC_OK.

4.2.6 beforeQuery

This method implements the OpenCPI *beforeQuery* control operation. See AMR for rationale and behavior.

4.2.6.1 Synopsis.

```
RCCResult beforeQuery(RCCWorker *this);
```

4.2.6.2 Returns.

This method shall return an RCCResult value.

4.2.6.3 Exceptions/Errors.

If the *beforeQuery* method cannot succeed, it shall return RCC_ERROR. If the worker detects an error that would disable the implementation or its environment, it shall return RCC_FATAL. Otherwise it shall return RCC_OK.

4.2.7 run

4.2.7.1 Brief Rationale.

The *run* method requests that the worker perform its normal computation. The container only calls this method when the worker's *run condition* is true. This allows the container to fully manage all synchronization, minimizing the resources in use when the worker is not able to run, and minimizes the time to switch between running one worker and running another when the container uses a simple single threaded execution model.

4.2.7.2 Synopsis.

```
RCCResult run(RCCWorker *this,
              RCCBoolean timedout,
              RCCBoolean *newRunCondition);
```

4.2.7.3 Behavior.

The *run* method shall perform the worker's computational function and return a result. The *run* method may use information in its property structure, state of its ports, and its private memory to decide what to do. Normally this involves using messages in buffers at input ports to produce messages in buffers at output ports.

The *timedout* input parameter indicates whether the *run* method is being invoked due to time passing (the *usecs* value of the run condition). This indication is independent of port readiness.

Each port's *current.data* member may be tested to indicate port readiness.

The *run* method can indicate that all ports should be advanced by a special return value. It can also indicate disposition of buffers and ports by using the release, send, request, or advance container functions. Each function may only be called once per port per execution of the run method.

The *run* method may change the run condition by writing a TRUE value to the location indicated by the newRunCondition output argument, and setting a new runCondition in the runCondition member of RCCWorker. The worker maintains storage ownership of all run conditions. If the runCondition member of RCCWorker is set to null, the default runCondition is restored. The runCondition member of RCCWorker is initially set to the value in the RCCDispatch structure (including NULL). Since the worker code is managing this structure member, it can use it as a convenient "state variable" when its execution modes each have a different run condition.

4.2.7.4 Returns.

This method shall return an RCCResult value.

The value of RCC_ADVANCE shall indicate that all ports should be advanced that were ready on entry to the *run* function and were not subject to container functions called since then (e.g. advance, request etc.).

4.2.7.5 Exceptions/Errors.

If the *run* method cannot succeed, it shall return RCC_ERROR, indicating that it should not be called again (run condition always false). If the worker detects an error that would disable the implementation or its environment, it shall return RCC_FATAL. Otherwise it shall return RCC_OK or RCC_ADVANCE if normal worker execution should proceed (and the *run* method called again when the run condition is true). It shall return RCC_DONE, if no worker execution should proceed. Returning RCC_DONE indicates to the container that the worker will never require execution, and provides this advice to the container to allow the container to take advantage of this fact and possibly release resources (such as I/O buffers) prior to the worker instance being destroyed or reused.

4.2.8 *test*

This method implements the OpenCPI *test* control operation. See AMR for rationale and behavior.

4.2.8.1 Synopsis.

```
RCCResult test(RCCWorker *this);
```

4.2.8.2 Returns.

This method shall return an RCCResult value.

4.2.8.3 Exceptions/Errors.

If the *test* method cannot succeed, it shall return RCC_ERROR. If the worker detects an error that would disable the implementation or its environment, it shall return RCC_FATAL. Otherwise it shall return RCC_OK. Note that test “results” are provided in readable properties, so returning RCC_ERROR implies that the test could not be run at all, usually due to invalid test properties. It does not indicate that the result of running the test was not success.

5 Code generation from RCC OWD Metadata

5.1 Ports inferred from the RCC OWD DataInterfaceSpec element.

The port interfaces for a worker shall be consistent with the associated DataInterfaceSpec XML element in the RCC OWD for the worker. RCC worker ports are inferred from the RCC OWD information as follows:

- Ports are ordered according to their appearance in the OCS.
- Ports with protocols with no two-way operations imply one RCC worker port.
- Ports with some two-way operations imply two RCC worker ports; first read-only input then write-only output.
- Consumer/Server/Provider ports take as input request messages and write response messages as output.
- Producer/Client/User ports write request messages as output and read response messages as input.
- All RCC worker port interfaces are read or write, but not both.

Thus there is a single ordering of all worker ports, combining the required input and output ports implied by the OCS. For a worker named “xyz”, an enumeration of named worker port ordinals will be generated in the file Xyz_Worker.h, using the typedef name XyzPort (Where Xyz is the capitalized worker implementation name in the OWD). The enumerated constants will be all upper case, of the form XYZ_PPP_RRR_DIR. XYZ is the upper cased worker implementation name and PPP is the upper cased port name. RRR and DIR are only included with bidirectional component ports (when some protocol operations are *two-way*). RRR will be “REQUEST” or “REPLY” as appropriate. DIR will be “IN” or “OUT”. These rules fully describe port roles for authors and readers.

For aid in initializing the numInputs and numOutputs members of the RCCDispatch structure, the macros XYZ_N_INPUT_PORTS and XYZ_N_OUTPUT_PORTS shall also be generated with the appropriate values.

5.1.1 Property structure from OCS and OWD Property Elements

The layout of the property structure is derived from the Property elements found in (first) the OCS and (second) the RCCImplementation in the OWD. This layout is expressed as a normal C struct, with member names being the properties’ “Name” attributes from the OCS and OWD. This struct typedef can be used for accessing property values using the “properties” pointer in the *RCCWorker* structure. This struct is automatically generated by OpenCPI tools as part of the automatic generation of the complete Xyz_Worker.h file. The rules for generating this structure are:

- *The standard name of the struct type (typedef name) is XyzProperties where “Xyz” is the mixed case (usually with initial upper case) worker implementation name in the OWD.*

- *The standard name for the generated header file containing this definition (and others) is "Xyz_Worker.h". This precludes using "RCC" for the worker implementation name.*
- *Properties that are sequences are preceded by an unsigned long member whose name is the property name with "_length" appended. Padding may be added before and after the length member to achieve the required alignment of this length field as well as the sequence data following it.*
- *Sequence properties are represented by structure members that are C arrays whose length is the SequenceLength attribute from the Property element.*
- *Struct properties have structure tags the same name as the property name, preceded by the worker implementation name.*
- *The names of the integer types for simple properties: short, ushort, long, ulong, octet, char are mapped to the respective <stdint.h> types: int16_t, uint16_t, int32_t, uint32_t, uint8_t.*
- *The names of the non-integer types for simple properties are the OCS-defined type names capitalized and prefixed with "RCC": e.g. RCCBoolean, RCCChar, RCCFloat, RCCDouble.*
- *Properties that are string properties are "RCCChar" arrays whose size is one more than the StringLength attribute of that string property in the Property element, and the values are null terminated strings.*

5.1.2 Operation and exception ordinals

The Xyz_Worker.h file will supply an enumeration for each port, defining opcode or exception ordinals. The ordinals will have the typedef name XyzPqrOperation or XyzPqrException, and the constant name XYZ_PQR_ORD, where ORD is the name of the operation or exception. For any reply port, RCC_NO_EXCEPTION and RCC_SYSTEM_EXCEPTION are also valid. PQR is the upper-cased port name.

These ordinals represent the message types based on the Operation subelements of the Protocol subelement of the DataInterfaceSpec corresponding to the port.

Enumerations will also be generated for each protocol used on any port. These enumerations may be more meaningful and/or convenient since the worker code can use the protocol operation enumeration names regardless of the port. The typedef name for the protocol enumeration is PqrOperation, where "Pgr" is the capitalized name of the protocol (the name in the XML, not the file name). Each enumeration constant is of the form PQR_OP, where PQR is the upper cased protocol name and OP is the upper cased operation name.

5.1.3 Message structures

Since the Argument elements in an Operation element are allowed to be variable length sequences and strings, arguments in operation messages can be variable in length. Thus in the general case, when multiple arguments are unbounded, no C data structure can capture the entire message format. This specification defines a normative structure for message contents up to and including the first variable length argument, and, for

structures or unions, the first variable length member. This applies to request, reply and exception messages.

This partial solution to mapping the message contents into a normative structure is intended to facilitate many, but not all message content access. Since fixed size messages, or messages whose only variable element is the last argument, are very common, this solution adds portability value in many cases. More complex cases, after the first variable argument, must be accessed according to the message layouts as described in [FIXME not yet somewhere to reference, but should be in AMR].

The message structures generated for each port, will have the typedef name XyzPqrOpr. Xyz is worker name, Pqr is port name, and Opr is operation or exception name. The members of the structure are named the same as the property structure above, but with argument names from the Argument elements rather than from the Property elements. If the first variable length argument is a sequence of variable length types (sequence or string), the type of the member for that variable length type will be uint32_t, which will be the last member in the structure.

The message structure member types for the integer types: *short, unsigned short, long, unsigned long, long long, unsigned long long, octet, and enum* are mapped to the respective CDR-sized <stdint.h> types: int16_t, uint16_t, int32_t, uint32_t, int64_t, uint64_t, uint8_t, uint32_t.

There is one special exception for messages that consist entirely of one sequence of fixed length elements: the length of the sequence is implied by the overall message length as specified by the “length” elements of input and output ports, and *not* an inserted unsigned long value in the message buffer/structure. In all other cases (not-fixed sequence elements, or more than one top level data argument in the message), sequences in messages are preceded by the unsigned long number-of-elements value.

The member types for the non-integer basic types are the Type attribute value names capitalized and prefixed with “RCC”: e.g. RCCBoolean, RCCChar, RCCFloat, RCCDouble. String types are “RCCChar” arrays.

The structure generated is padded and packed. This means padding members are explicitly inserted to ensure all types are aligned on their own boundaries. It also means that the compiler is told to “pack” the structure, Given the insertion of padding, the “packing” simply means that there is no padding at the end of the structure to achieve any alignment. Thus the struct definition should not be used in any other context where this lack of overall size alignment is required (i.e. an array of such structures).

6 RCC Local Services

Local service APIs support RCC workers running in a container. Containers are required to supply them, and workers are constrained to use only them (other than the worker-to-container entry points defined below). The RCC local services are defined as a small subset of the POSIX and ISO-C runtime libraries.

6.1 Container Multithreading Support

Threading (in RCC environments) can generally be supplied by the container, and not require the worker developer to create or manage threads. The RCC base profile does not allow workers themselves to create or manipulate threads, and the worker's control interface is guaranteed not to have more than one of its methods entered at a time. The RCC multithreaded profile *does* allow the use of the POSIX pthread API to create and manipulate threads, although methods will not be called simultaneously in different threads. Only one method will be called at a time in any worker: the container will enforce this.

RCC environments are frequently configured without support for threads. The lack of thread support in the base profile is consistent with such environments. RCC containers supporting only the base profile may still run multiple base-profile workers in separate threads.

RCC containers supporting the multithreaded profile must support multithreading and support workers to wait/block for input or output buffers.

6.2 RCC Local Services AEP as a small subset of POSIX and ISO-C.

The local services available for RCC workers are typical runtime support functions without the functions that require heavy weight OS support. All I/O is excluded since portable RCC workers should be performing all I/O via the OpenCPI data plane ports. Some containers in fact allow many more functions to be called from a worker, but that makes such workers non-portable.

The POSIX Minimal Realtime System Profile (PSE51) from the IEEE Std 1003.13TM-2003 standard is used as the superset of functionality that is reduced (with subtractions) to from the RCC local services available functions. Using this subset requires setting the _POSIX_AEP_RT_MINIMAL_C_SOURCE feature test macro to the value 200312L during compilation.

The PSE51 specification defines POSIX.1 units of functionality in its table 6-1. This RCC Local Services AEP (application environment profile) removes these units of functionality from that table:

- POSIX_DEVICE_IO
- POSIX_FILE_LOCKING
- POSIX_SIGNALS
- XSI_THREAD_MUTEX_EXT
- XSI_THREADS_EXT

In the base profile (not multithreaded) it also removes:

- POSIX_THREADS_BASE

The retained units of functionality, and their defined symbols and functions are:

POSIX_C_LANG_JUMP: *longjmp(), setjmp();*

POSIX_C_LANG_SUPPORT: *abs(), asctime(), asctime_r(), atof(), atoi(), atol(), atoll(), bsearch(), calloc(), ctime(), ctime_r(), difftime(), div(), feclearexcept(), fegetenv(), fegetexceptflag(), fegetround(), feholdexcept(), feraiseexcept(), fesetenv(), fesetexceptflag(), fesetround(), fetestexcept(), feupdateenv(), free(), gmtime(), gmtime_r(), imaxabs(), imaxdiv(), isalnum(), isalpha(), isblank(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), labs(), ldiv(), llabs(), lldiv(), localeconv(), localtime(), localtime_r(), malloc(), memchr(), memcmp(), memcpy(), memmove(), memset(), mktime(), qsort(), rand(), rand_r(), realloc(), setlocale(), snprintf(), sprintf(), srand(), sscanf(), strcat(), strchr(), strcmp(), strcoll(), strcpy(), strcspn(), strerror(), strerror_r(), strftime(), strlen(), strncat(), strncmp(), strncpy(), strpbrk(), strrchr(), strspn(), strstr(), strtod(), strtof(), strtoimax(), strtok(), strtok_r(), strtol(), strtold(), strtoll(), strtoul(), strtoull(), strtoumax(), strxfrm(), time(), tolower(), toupper(), tzname, tzset(), va_arg(), va_copy(), va_end(), va_start(), vsnprintf(), vsprintf(), vsscanf();*

POSIX_SINGLE_PROCESS: *confstr(), environ, errno, getenv(), setenv(), sysconf(), uname(), unsetenv();*

POSIX_THREADS_BASE (in the multithreaded RC profile): *pthread_atfork(), pthread_attr_destroy(), pthread_attr_getdetachstate(), pthread_attr_getschedparam(), pthread_attr_init(), pthread_attr_setdetachstate(), pthread_attr_setschedparam(), pthread_cancel(), pthread_cleanup_pop(), pthread_cleanup_push(), pthread_cond_broadcast(), pthread_cond_destroy(), pthread_cond_init(), pthread_cond_signal(), pthread_cond_timedwait(), pthread_cond_wait(), pthread_condattr_destroy(), pthread_condattr_init(), pthread_create(), pthread_detach(), pthread_equal(), pthread_exit(), pthread_getspecific(), pthread_join(), pthread_key_create(), pthread_key_delete(), pthread_kill(), pthread_mutex_destroy(), pthread_mutex_init(), pthread_mutex_lock(), pthread_mutex_trylock(), pthread_mutex_unlock(), pthread_mutexattr_destroy(), pthread_mutexattr_init(), pthread_once(), pthread_self(), pthread_setcancelstate(), pthread_setcanceltype(), pthread_setspecific(), pthread_sigmask(), pthread_testcancel();*

The philosophy of the base RCC profile AEP subset is to allow functions that are simply libraries (rather than OS services), but remove services that could conflict with the lean container execution model in that profile. This basically leaves the typical ANSI-C (or ISO C99) runtime library (without I/O) – most DSPs have this available anyway, even those environments without any multithreading.

[FIXME: we should really dial this back to the IEEE Std 1003.13-1998 version].

7 OpenCPI Implementation Descriptions for RCC Workers (RCC OWD)

7.1 Introduction

This section describes the format and structure of the RCC OWD: the XML document that refers to or includes an OpenCPI Component Specification (OCS), and which specifies the implementation-specific aspects of an RCC worker. The AMR defines aspects common to all OWDs. This section defines aspects of the RCC OWD – specific to RCC workers. The AMR, and its descriptions of OCSs and OWDs is a prerequisite for this section.

7.2 Top Level Element: RCCImplementation

An RCCImplementation element contains information provided by someone creating an RCC worker based on a component specification (OCS). It includes or references an OCS, and then describes implementation information about a particular implementation of that OCS. Thus, its attributes, the RCCImplementation must either include as a child element a complete ComponentSpec, or include one by reference, for example, if the "fastcore" implementation of the "corespec1" specification referenced the component specification found in the "corespec1.xml" file:

```
<RCCImplementation
      xmlns:x="http://www.w3.org/2001/XMLSchema-instance"
      x:schemaLocation="http://www.omg.org/CPI WIP-schema1.xsd"
      xmlns="http://www.omg.org/CPI"
      xmlns:xi="http://www.w3.org/2001/XInclude"
   Name="fastcore"
   ---other attributes---
   >
   <xi:include href="corespec1.xml"/>
   ---other child elements---
</RCCImplementation>
```

The RCCImplementation follows the specification of OWDs in general as specified in the AMR. This section defines the aspects of the RCC OWD (RCCImplementation) that is not common to all OWDs (thus described in the AMR).

7.2.1 Attributes of an RCC Component Implementation

7.2.1.1 Name attribute

The "Name" attribute of the component implementation is used to generate the programming language name of the worker (e.g. used in the name of and the contents of the RCC generated header file). Per the AMR, when not specified the implementation name defaults to the name of the OWD file without directories or extensions.

7.2.1.2 Threaded attribute

The “threaded” attribute is a Boolean attribute that indicates whether this RCC worker is written to the multithreaded profile. If this attribute is false (the default if not specified), the RCC worker is written to the base profile.

7.2.1.3 ExternMethods attribute

This String attribute is used (set to true), in the cases where the methods of the worker (whose addresses are placed in the RCCDispatch structure) should have external linkage scope rather than be local to the file (i.e. declared “static”). In this case the string value of this attribute is used as the prefix to those external function names, rather than just being file-scoped static functions whose names are simply the method names (i.e. “initialize”, “start”, “run”, etc.).

7.2.2 Control Plane Aspects of an RCC Component Implementation

7.2.2.1 ControlInterface element

The control interface child element (ControlInterface) of the RCCImplementation specifies implementation aspects of the worker’s control functionality that are different from the default.

As stated in the AMR, Properties, Property, or SpecProperty elements can occur under this element. A Properties element can be referenced in an external file via xi:include.

7.2.2.1.1 ControlOperations attribute of ControlInterface

This attribute is as defined in the AMR. If not specified, no control operation methods are implemented by the RCC worker (only the “run” method). This information is used in the generation of the header file.

7.2.3 Data Plane Aspects of an RCCImplementation

7.2.3.1 Port element

The Port child element of the RCCImplementation specifies RCC-specific information about a data interface specified in the OCS. It references a DataInterfaceSpec by its Name attribute. Thus the Name attribute of the Port element must match the Name attribute of a DataInterfaceSpec element of the ComponentSpec. The Port element adds implementation-specific information about the interface initially defined in that DataInterfaceSpec.

7.2.3.1.1 Name attribute

This attribute specifies the name used to reference the DataInterfaceSpec in the ComponentSpec (OCS).

7.2.3.1.2 MinBuffers attribute

This numeric attribute specifies the minimum number of message buffers required by the RCC worker for a port. The RCC Worker Interface allows the worker code (typically in the “run” method) to “take” a buffer from a port, and ask for a new buffer for that port

while retaining ownership of the previous buffer from that port. This behavior requires that the infrastructure provide at least 2 buffers for that port. Thus this attribute informs the infrastructure as to the minimum buffering requirements of the RCC worker implementation for that port. The default is 1. This attribute should *not* be used to "tune" the buffer count for performance but only specify the actual minimum requirements for the correct functioning of the RCC worker.

8 Summary of OpenCPI RCC authoring model

- RCC workers are written to implement the Worker interface, called by the container, and optionally use the Container interface, called by the worker. The interface is conveyed to the container via the RCCDispatch type.
- RCC workers may call functions defined in the RCC AEP.
- RCC workers use data structures and ordinals as defined in the metadata code generation section (which will be automatically generated).
- RCC worker authors must make the following implementation-specific metadata available to the build process (see OWD below) for compiling/linking/loading worker implementation binaries on a given platform:
    - Implementation name
    - Which control operations are implemented
    - Which properties require beforeQuery or afterConfigure notification
    - Static memory allocation requirements of the implementation code
    - Profile used in the implementation (base or multithreaded).
    - Minimum number of buffers required at each port (default is one).
- The information in the OWD is used (which includes the implementation metadata above), to drive the build process and the code generation process for the RCC worker's header file.

9 Code examples

9.1 Worker code example for base profile

Here is a simple example of an “Xyz” worker using the base profile whose:

- initial run condition was the default (condition == NULL, usecs == 0, run when all ports are ready, no timeout),
- initialize, release and test methods are empty
- one input port (0) with interface XyxIn (only oneways) and one output port (1) XyzOut (only oneways)
- one oneway IDL interface operation Op1 on input (i.e. can ignore “operation”), which is an array of 100 “shorts”.
- one oneway IDL interface operation Op2 on output
- one simple property, called center_frequency, of type float.

The worker would have automatically generated types and structures like this (based on OCS and OWD), and put in a file called “Xyz_Worker.h”:

```
#include “RCC_Worker.h”
typedef struct {  /* structure for defined properties /
   RCCFloat center_frequency;
} XyzProperties;
typedef struct {  // structure for message for operation
   int16_t ishorts[100];
} XyzInOp1;

typedef struct {  // structure for message for operation
   int16_t oshorts[100];
} XyzOutOp2;

typedef enum { // port ordinals
   XYZ_IN,
   XYZ_OUT
} XyzPort;

typedef enum { // operation ordinals
   XYZ_OUT_OP2
} XyzOutOperation;

typedef enum { // operation ordinals
   XYZ_IN_OP1
} XyzInOperation;
```

The actual code for the worker would look like this:

```
#include "Xyz_Worker.h"

/* Define the initialize method, setting output operation to be a */
/* constant, since it is the only one. Make it static because it */
/* doesn't need to be global */
 static RCCResult
initialize(RCCWorker *w) {
   w->ports[XYZ_OUT].output.u.operation = XYZ_OUT_OP2;
   return RCC_OK;
}

/* Define run method to call the "compute" function, reading from */
/* input buffer, writing to output, applying current value of the */
/* "center frequency" property.  Make it static because there is no */
/* reason to make it global. */
 static RCCResult
run(RCCWorker *w, RCCBoolean timedout, RCCBoolean *newRunCondition)
{
   XyzProperties *p = w->properties;
   XyzInOp1 *in = w->ports[XYZ_IN].current.data;
   XyzOutOp2 *out = w->ports[XYZ_OUT].current.data;

   /* Do computation based in ishorts, and frequency put results */
   /* in oshorts.  Extern is here simply for readability. */
   extern void compute(int16_t *, int16_t *, float);

   compute(in->ishorts, out->oshorts, p->center_frequency);
   /* Ask container to get new input and output buffers */
   return RCC_ADVANCE;
}

/* Initialize dispatch table for container, in a global symbol /
 RCCDispatch
Xyz = {
   /* Consistency checking attributes */
   RCC_VERSION, 1, 1, sizeof(XyzProperties), RCC_NULL, RCC_FALSE,
   /* Methods */
   initialize, RCC_NULL, RCC_NULL, RCC_NULL, RCC_NULL, RCC_NULL,
   RCC_NULL, run,
   /* Default run condition */
   RCC_NULL
};
```

Other than the compute function, the above example compiles to use less than 120 bytes on a Pentium processor.

A simple non-preemptive single-threaded container implementation would simply have a loop, testing run conditions, and calling run methods. A more complex environment might run workers in different threads for purposes of time preemption, prioritization,

etc. This model allows a variety of container execution models while keeping the worker model simple.

So, on each execution, the worker sees the status of all I/O ports, and can read from current input buffers, and write to current output buffers. It must return to get new buffers, after specifying whether buffers are consumed or filled during the execution.

This simple execution environment can be easily implemented in GPP environments, providing a test environment and a migration path to GPPs.

9.1.1 Worker code example for multithreaded profile

Consider a similar but less trivial worker than that presented in section XYZ above. This xyt worker which:

- does not require or support multithreading.
- has no processing to perform on initialize, release, test, start or stop.
- exports one SCA provides port with name IN that supports a single oneway IDL interface operation Op1 (i.e. can ignore “operation”), which is an array of 100 “shorts”. This maps to worker port(0).
- has two SCA uses ports:
    - one that has the name OUT that contains a single oneway interface operation Op2. This maps to worker port(1)
    - one that has the name Compute that contains a single two way interface operation Process. This maps to.
    - output worker port (2) for output commands
    - input worker port (3) for input responses
- has one property, called center_frequency
- responds to the oneway operation call on its provides port by:
    - manipulating the input data in some local Compute function based on the current center frequency value.
    - passing the intermediate result to another component via the two way Process operation on its one uses port for further processing.
    - forwarding the final result to its destination via Op2 oneway operation on its other uses port.
- Takes advantage of the fact that the data structure for the provides and uses operations is the same to avoid copying data.

This worker would have a header file generated like this (from the OWD):

```
#include "RCC_Worker.h"

typedef struct { /* structure for defined properties */
   RCCFloat center_frequency;
} XytProperties;

typedef struct { /* structure for message for operation */
   int16_t ishorts[100];
} XytInOp1;

typedef struct { /* structure for message for operation */
   in16_t oshorts[100];
} XytOutOp2;

typedef struct { /* structure for message for operation */
   int16_t ishorts[100];
} XytProcessInOp3;

typedef struct { /* structure for message for response */
   int16_t oshorts[100];
} XytProcessOutOp3;

typedef enum {
   XYT_IN,
   XYT_OUT,
   XYT_COMPUTE_REQUEST_OUT,
   XYT_COMPUTE_REPLY_IN
} XytPort;

typedef enum {
   XYT_OUT_OP2
} XytOutOperations;

typedef enum {
   XYT_IN_OP1
} XytInOperations;

typedef enum {
   XYT_COMPUTE_PROCESS
} XytComputeOperations;
```

The actual code for the worker is presented below for each of the two profiles (most error handling omitted for sake of clarity):

9.1.1.1 Worker implementation using the multithreaded profile, with input callback

This illustrates an input-based callback function that has a sequential coding style in which the worker blocks on the two-way remote call until the response is received and processing can continue. No extra threads are used.

The code is:

```
#include "Xyt_Worker.h"

/* Define the input port callback method to call the local "compute"
 * function, reading from input buffer, applying the value of the
 * "center frequency" property, followed by two way  "process"
 * operation, finally writing the result to output port XYT_OUT */
 static RCCResult
computeInput(RCCWorker *this, RCCPort *inPort, RCCResult reason)
{
   XytProperties *p = this->properties;
   RCCContainer *c = &this->container;
   XytInOp1 *data = inPort->current.data;
   RCCPort *computeOut = &this->ports[XYT_COMPUTE_REQUEST_OUT],
           *computeIn = &this->ports[XYT_COMPUTE_REPLY_IN],
           *otherOut = &this->ports[XYT_OUT];
   extern void compute(int16_t *, int16_t *, float);

   if (reason != RCC_OK)
      return RCC_FATAL;

   /* Do computation based in ishorts, and frequency; put results
    * back in ishorts, in place */
   compute(data->ishorts, data->ishorts, p->center_frequency);

   /* Call Process operation on "uses" port: buffer ownership passes
    * back to container. */
   c->send(computeOut, &inPort->current, XYT_COMPUTE_PROCESS,
           inPort->input.length);
   /* Wait until response received (or 100 msec error timeout). /
   c->wait(computeIn, 100, 100000);
   if (computeIn->input.u.exception == RCC_NO_EXCEPTION)
      c->send(otherOut, &computeIn->current, XYT_OUT_OP2,
              computeIn->input.length);
   else
      c->release(&computeIn->current);
   return RCC_OK;
}
/* continued on next page */
```

```
/* Define the initialize method to perform worker initialization. */
 static RCCResult
initialize(RCCWorker *this)
{
  this->ports[XYT_IN].callBack = computeInput;
  return RCC_OK;
}

/* Initialize dispatch table provided to container. We only need the
 * initialize method to register the callback.  No run method needed */
 RCCDispatch
Xyt = {
   /* Consistency checking attributes */
   RCC_VERSION, 1, 1, sizeof(XytProperties), RCC_NULL, RCC_TRUE,
   /* Methods */
   initialize
   /* all remaining members zero/NULL */
};
```

Other than the compute function, the above example compiles to use less than 300 bytes on a Pentium processor.

9.1.1.2 Worker implementation using the base profile using state-machine style

This illustrates a finite state machine coding style in a worker that maintains an internal state to simulate blocking on the two-way remote call until the response is received and processing can continue.

```
#include "Xyt_Worker.h"
/* Define two different run conditions to represent two states */
static uint32_t
  state1Ports[] = {1 << XYT_IN, 0},
  state2Ports[] = {1 << XYT_COMPUTE_REPLY_IN, 0};
static RCCRunCondition
  awaitingInput = {state1Ports},
  awaitingResponse = {state2Ports};

/* Define start method, set run condition, which is also state. /
 static RCCResult
initialize(RCCWorker this)
{
   this->runCondition = &awaitingInput;
   return RCC_OK;
}
/* continued on next page */
```

```
/* Define run method to call the local "compute" function
 * reading from input buffer, applying current value of the
 * "center frequency" property, followed by remote two way
 * "process" operation, finally writing result to output. */
 static RCCResult
run(RCCWorker *this, RCCBoolean timedout, RCCBoolean *newRunCondition)
{
   RCCContainer *c = &this->container;

   /* Use run condition as state indicator */
   if (this->runCondition == &awaitingInput) {
      RCCPort
         *inPort       = &this->ports[XYT_IN],
         *computeOut   = &this->ports[XYT_COMPUTE_REQUEST_OUT];
      XytInOp1 *in     = inPort->current.data;
      XytProperties *p = this->properties;

      /* do some computation based in ishorts, and frequency;
       * put results back in same buffer (in-place) */
      extern void compute(int16_t *, int16_t *, float);
      compute(in->ishorts, in->ishorts, p->center_frequency);

      /* Call Process op on user port - buffer ownership passes back
       * to container.  Input port is advanced by taking buffer away
       * from it. */
      c->send(computeOut, &inPort->current, XYT_COMPUTE_PROCESS,
              inPort->input.length);      /* length of message */
      this->runCondition = &awaitingResponse; /* update state */
   } else {
      RCCPort *computeIn = &this->ports[XYT_COMPUTE_REPLY_IN];
      RCCPort *otherOut = &this->ports[XYT_OUT];

      if (computeIn->input.u.exception == 0)
         c->send(otherOut, &computeIn->current, XYT_OUT_OP2,
                 computeIn->input.length);
      else
         c->advance(computeIn, 0);
      this->runCondition = &awaitingInput; /* update state */
   }
   *newRunCondition = RCC_TRUE; /* to container: new runcondition /
   return RCC_OK;
}
/* continued on next page */
```

```
/* Initialize dispatch table provided to container. We only need the
 * initialize method to register the callback.  No run method needed */
RCCDispatch Xyt = {
   /* Consistency checking attributes */
   RCC_VERSION, 1, 1, sizeof(XytProperties), RCC_NULL, RCC_FALSE,
   /* Methods */
   initialize, RCC_NULL, RCC_NULL, RCC_NULL, RCC_NULL, RCC_NULL,
   RCC_NULL, run,
   /* all remaining members zero/NULL */
};
```

Other than the compute function, the above example compiles to use less than 330 bytes on a Pentium processor.

10 Glossary

Configuration Properties – Named storage 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.

Container – A collection of "infrastructure" IP configured to "contain" a set of application workers, such that the two combined represent a complete FPGA design. I.e. a "donut" of IP with chip pins on the outside, and the application workers residing in the "hole". Interfaces that cross the inside container bounds (between the hole and the donut) are between application and infrastructure. Interfaces that stay within the hole are between application workers. The outside of the container represents off-chip interfaces.

Implementation Attribute – An attribute related to a particular implementation (design) of a worker, for one of its interfaces. I.e. one that is not necessarily common across a set of implementations of the same high level component definition (OCS).

Opcode – Synonymous with "message type" in the context of this document. Opcodes allow multiple message types to be sent across worker data interfaces.

Worker – A concrete implementation (and possibly runtime instance) of a component, generally existing within a container. "Application IP blocks" (components) are termed "workers" in this document.