

OpenCPI RCC Development Guide

Revision History

Revision	Description of Change	Date
0.1	Initial, from earlier doc sources	2015-10-31
0.9	Add all C++ content slave/proxy content, references to CDG	2016-01-27
0.95	Another editing pass with more consistent typography	2016-02-04
1.0	Accommodate additional issues, debug, opcodes, additional C++ container methods, debugging, C++ example	2016-04-13
1.0.1	Clarify run conditions, beforeQuery, firstRun, minbuffercount, port masks, ordinal()	2016-09-24
1.0.2	Clarify (non) interaction between minbuffercount OWD attribute and run conditions	2016-10-18
1.1	Update for 2017Q1	2017-02-21
1.2	Update for 2017.Q1	2017-08-08
1.3	More multi-threaded guidance and limited library/local service functions	2018-02-23
1.4	C++ log methods added, and setRunCondition clarifications, multiple slaves using slave elements	2018-09-24

Table of Contents

1	Introduction.....	6
1.1	References.....	7
2	Overview.....	8
3	XML Description Files (OWD) for RCC Workers.....	9
3.1	Attributes of a Top-level RCCWorker Element.....	10
3.1.1	Name Attribute — See the CDG.....	10
3.1.2	Spec Attribute — See the CDG.....	10
3.1.3	ControlOperations Attribute — See the CDG.....	10
3.1.4	Language Attribute.....	10
3.1.5	Slave Attribute — C++ Language Only.....	10
3.1.6	ExternMethods Attribute — C Language Only.....	10
3.1.7	StaticMethods Attribute — C Language Only.....	11
3.2	Attributes of Port Elements in the OWD.....	12
3.2.1	Name Attribute.....	12
3.2.2	MinBufferCount Attribute.....	12
3.3	Attributes of Slave Elements in the OWD.....	13
3.3.1	Worker Attribute.....	13
3.3.2	Name Attribute.....	13
4	The RCC Worker Interface.....	14
4.1	The RCC Execution Model.....	16
4.2	Worker Methods: Called by the Container, Implemented by the Worker.....	17
4.2.1	RCCWorker Structure Type – C Language Only.....	17
4.2.2	The Worker Derived Class – C++ Language Only.....	18
4.2.3	RCCResult Enumeration Type — C and C++ Languages.....	19
4.2.4	Worker Initializations.....	20
4.2.5	initialize — Worker Method.....	20
4.2.6	start — Worker Method.....	21
4.2.7	stop — Worker Method.....	22
4.2.8	release — Worker Method.....	22
4.2.9	run — Worker Method.....	22
4.3	Container Methods, Called by the Worker.....	24
4.3.1	firstRun — Container Method.....	24
4.3.2	setError — Container Method.....	25
4.3.3	log — Container Method.....	25
4.3.4	willLog — Container Method.....	25
4.3.5	time/getTime — Container Method.....	26
4.3.6	The RCC::RunCondition C++ Class and RCCRunCondition C Structure.....	26
4.3.7	getRunCondition — Container Method — C++ Language Only.....	29
4.3.8	setRunCondition — Container Method — C++ Language Only.....	29
4.4	Port Management Data Members and Methods.....	31
4.4.1	advance — Release Current Buffer, and Request Another.....	32
4.4.2	hasBuffer — Query a Port for Whether It Has a Current Buffer-to-release.....	32
4.4.3	isConnected — Query a Port for Being Connected.....	33
4.4.4	ordinal — Obtain an Ordinal for a Port.....	33

4.4.5	request — Request a New Buffer.....	33
4.4.6	send — Send an Input Buffer on an Output Port.....	34
4.4.7	setDefaultLength — Set Default Message Length at an Output Port — C++ Language Only .	34
4.4.8	setDefaultOpCode — Set Default Opcode at Output Port — C++ Language Only	35
4.4.9	take — Take a Buffer from an Input Port.....	35
4.5	Buffer Management Data Members and Methods.....	37
4.5.1	checkLength — Check Size of the Buffer — C++ Language Only.....	37
4.5.2	data — Access the Raw Contents of the Buffer.....	37
4.5.3	length — Retrieve the Length of the Message in a Buffer - C++ Language Only.....	38
4.5.4	maxLength — Retrieve the Maximum Available Space in the Buffer.....	38
4.5.5	opCode — Retrieve the OpCode of the Message in a Buffer.....	38
4.5.6	release — Release a Buffer.....	38
4.5.7	setLength — Set the Length of the Message in a Buffer — C++ Language Only.....	39
4.5.8	setOpCode — Set the OpCode for the Message in a Buffer — C++ Language Only.....	39
4.5.9	setInfo — Set the Metadata Associated with a Buffer — C++ Language Only.....	40
4.5.10	topLength — Retrieve the Size of the Single Sequence in a Message — C++ Language Only	40
4.6	Accessing the Contents of Messages.....	41
4.6.1	Accessing Messages in C Language Workers.....	41
4.6.2	Accessing Messages in C++ Language Workers.....	42
4.7	How a Worker Accesses its Properties.....	43
4.7.1	The Worker Property Structure.....	43
4.7.2	The Accessing Worker Properties in C.....	44
4.7.3	Accessing Worker Properties in C++.....	44
4.7.4	Property Access Notifications in C++.....	44
4.7.5	Accessing the Values of Parameter Properties.....	45
4.8	Controlling Slave Workers from Proxies — C++ Language Only.....	46
4.9	Worker Dispatch Structures — C Language Only.....	48
4.9.1	RCCDispatch Structure Type.....	48
5	Code Generation for RCC Workers.....	50
5.1	Namespace Management.....	51
5.1	Generated Data Types.....	52
5.1.1	The Enumeration Constants for the Worker's Ports.....	52
5.1.2	The Properties Structure Type.....	52
5.1.3	Structures for Message Payloads — C Language Only.....	53
5.1.4	Worker Base Class — C++ Language Only.....	54
6	RCC Local Services.....	55
6.1	RCC Local Services AEP as a Small Subset of POSIX and ISO-C.....	57
7	Debugging RCC Workers.....	59
8	Summary of OpenCPI RCC Authoring Model.....	61
9	Building RCC Workers.....	62
9.1	RCC Compiler and Linking Options.....	62
10	Worker Code Examples.....	65
10.1	C Language Examples.....	65

10.1.1	Worker Using the Default Run Condition.....	65
10.1.2	Worker Using the State-machine Style.....	66
10.2	C++ Language Examples.....	70
10.2.1	Worker Using the State-machine Style.....	72
11	Glossary.....	73
12	List of Abbreviations and Acronyms.....	74

1 Introduction

This document specifies the **OpenCPI Resource-Constrained C/C++ Language (RCC)** authoring model and describes how to write RCC workers in C or C++. This model is based on the C language and makes most design choices to minimize resources appropriate for resource-constrained embedded systems. DSP processors with on-chip memories, micro-controllers, and multi-core processors are natural targets for this authoring model. The RCC model is also an appropriate model for any general-purpose processor with a C compiler, when the developer is comfortable with the constraints of the C language.

This document also describes a C++ variant of this authoring model that takes advantage of the expressive power of the C++ language. Both the C and C++ language variants are considered to be based on this one authoring model as so many concepts and details are common.

This specification is based on the authoring model concept as defined in the **OpenCPI Component Development Guide**, and is a prerequisite to this document. That document introduces key concepts for all authoring models, including the configuration and lifecycle model of components, and the software execution model for most authoring models targeting general purpose software platforms.

All OpenCPI authoring models are required to coexist and interoperate with the other existing models that are more appropriate for their respective processing technologies. These include the OCL authoring model for GPUs, and the HDL authoring model for FPGAs. Other models and any unique aspects to their associated development workflow, are described in their own documents.

1.1 References

This document depends on several others. Primarily, it depends on the **OpenCPI Component Development Guide**, which describes concepts and definitions common to all OpenCPI authoring models. As the 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. The C++ authoring model is based on the language as defined in **ISO/IEC 14882:2003**, prior to Cxx11. While it is possible to develop workers enabling C++11 (or later) features, this reduces portability for embedded systems running older compilers. A later major release will be centered on C++11.

Table 1: Table of Reference Documents

Title	Published By	Link
OpenCPI Overview	OpenCPI	https://github.com/opencpi/opencpi/raw/2018.Q3/doc/pdf/OpenCPI_Overview.pdf
OpenCPI Component Development Guide	OpenCPI	https://github.com/opencpi/opencpi/raw/2018.Q3/doc/pdf/OpenCPI_Component_Development.pdf
ISO C Language Specification	C Language	ISO/IEC 9899:90
ISO C++ Language Specification	C++ Language	ISO/IEC 14882:2003

2 Overview

RCC workers are C/C++ language component implementations that do work. They are hosted in an OpenCPI **container**, which is responsible for:

- loading, executing, controlling, and configuring the worker
- effecting data movement to and from the data ports of the worker
- providing interfaces for the local services available to RCC workers.

RCC workers that are executing and colocated together in the same container can make use of local zero-copy approaches to move data between them. For connections between workers in *different* containers, the containers move data between each other using a common data transport mechanism. Containers make use of a default data transport between the two devices unless explicitly configured to do otherwise.

The **OpenCPI Component Development Guide (CDG)** contains sections for the general introduction to the control plane functionality of workers and containers, followed by the general execution model of software-based workers. The specifics of the RCC authoring model are included here, consisting of:

- **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
- **the local services:** how the worker uses local services and which ones are available.

Creating a component implementation (a.k.a. *authoring a worker*), includes writing source code as well as specifying certain characteristics of the implementation in a separate XML file. This XML file is called the **OpenCPI Worker Description (OWD)**. It refers to the **OpenCPI Component Specification (OCS)** being implemented, specifies the authoring model and language, and describes any non-default constraints or behavior of this particular component implementation. It includes attributes and information that are specific to the authoring model. RCC OWDs are initially generated using the **ocpidev** tool and then may be further customized. The **ocpidev** tool and the OWD aspects that are common to all authoring models are described in the **OpenCPI Component Development Guide**.

RCC workers can act as a **proxy** for other workers, which then act as **slaves**. Proxy workers provide a control interface via their properties and control operations that are translated into lower level controls for workers that are more primitive, usually device specific, and usually written using different more processor-specific authoring models.

Having a proxy for higher level, more generic control processing relieves the slave workers of the complexity and burden and footprint of adapting to common interfaces and control protocols. A common use-case for proxies is controlling **device workers**, which are fully described in the **OpenCPI Platform Development Guide**. In an application, a proxy is guaranteed to be configured and started before its slaves are started, allowing the proxy to privately configure the slave.

3 XML Description Files (OWD) for RCC Workers.

This section describes the format and structure of the RCC OWD. This XML document specifies for the worker:

- The OCS being implemented (required)
- The authoring model (required) and language (may default)
- Some generic worker attributes allowed in all authoring models (all optional)
- Some attributes and elements specific to RCC workers (all optional)
- Property information beyond what is in the OCS (all optional)
- Port information beyond what is in the OCS (all optional)

The top level XML element for a RCC worker is **RCCWorker**, which implies the authoring model. Using defaults the simplest OWD for an RCC worker would be:

```
<RCCWorker spec='myspec' />
```

The **RCCWorker** XML element includes or references an OCS, and then describes implementation information about this particular RCC implementation of that OCS. The **RCCWorker** element must either include as a child element a complete OCS, or include one by reference, using the **spec** attribute of the top-level element. For example, the “vsadd” RCC implementation of the “vsadd-spec” OCS would reference the component specification this way:

```
<RCCWorker spec="vsadd-spec"  
  ---other attributes---  
>  
  ---other child elements---  
</RCCWorker>
```

The **RCCWorker** follows the specification of OWDs in general as specified in the CDG. This section only defines the aspects of the RCC OWD that are *not* common to all OWDs. A more complete example is below:

```
<RCCWorker language='c++' spec='vsadd-spec'>  
  <SpecProperty name='control' readable='true' />  
  <Property name='debug' type='float' volatile='true' />  
  <Port name='in' minBufferCount='2' />  
</RCCWorker>
```

The **RCCWorker** element may also include the OCS as an embedded child element in the rare cases when there can never be alternative implementations of that OCS.

3.1 Attributes of a Top-level RCCWorker Element

The **name**, **spec**, and **controloperations** attributes are the same for all authoring models and are described in the CDG.

3.1.1 *Name Attribute — See the CDG*

3.1.2 *Spec Attribute — See the CDG*

3.1.3 *ControlOperations Attribute — See the CDG*

3.1.4 *Language Attribute*

The **Language** attribute of the component implementation for RCC workers should have the value **c** or **c++**. The default is **c**.

3.1.5 *Slave Attribute — C++ Language Only*

This attribute indicates that this worker is a proxy for one other worker, and that other worker is named as the string value of this attribute. This attribute is only supported for C++. The name of the slave worker may have a package prefix, denoted with periods, if the slave worker is not in the same namespace as this worker. The slave worker name must include the authoring model suffix. An example is:

```
<RCCWorker language='c++' slave='ocpi.devices.xyz_adc.hdl' />
```

This would indicate that this worker is a proxy for the HDL worker named **xyz_adc**, in the **ocpi.devices** package name scope. The indicated slave worker must be in the same library as this proxy.

This *attribute* cannot be used when a worker is a proxy for more than one slave. In that case the **slave element** must be used. See [Slave Elements](#).

3.1.6 *ExternMethods Attribute — C Language Only*

The default scope and name for RCC worker methods in C is to be declared static, i.e. name-scoped in the file, and have the method name in lower case, e.g.:

```
static RCCResult start(RCCWorker *self);
```

This attribute is used to change the name scope to external, and provide a pattern string to use when generating the names of methods. The pattern string value of this attribute is like a **sprintf** format string where various letter codes are preceded by % to insert values into the string. The letter codes are: **m** (lower case method name), **M** (capitalized method name), **w** (lower case worker name), and **W** (capitalized worker name). An example is the pattern **%W_%m**, which, for the start method of the XYZ worker, would be:

```
extern RCCResult Xyz_start(RCCWorker *self);
```

The code generator uses this pattern when generating the skeleton file for the worker. Making worker methods have externally-scoped names allows them to be implemented in separate source files.

3.1.7 *StaticMethods Attribute — C Language Only*

This attribute provides a pattern like the **ExternMethods** attribute, but leaves the worker methods in the file scope, declared **static**.

3.2 Attributes of Port Elements in the OWD

The **port** child element of the **RCCWorker** specifies information about a data port in the OCS. It references an OCS **port**, or **dataInterfaceSpec**, element by its **Name** attribute. The **Name** attribute of the **Port** element must match the **Name** attribute of a **Port** or **DataInterfaceSpec** element of the **ComponentSpec**. The **Port** element adds implementation-specific information about the port initially defined in that **ComponentSpec**.

A number of attributes available for the **port** element are common to all authoring models and are described in the CDG. These are usually used to override attributes inferred from the protocol associated with the data port.

3.2.1 Name Attribute

This attribute specifies the name used to reference the **Port** or **DataInterfaceSpec** element in the OCS. It must match the name used for the port in the OCS.

3.2.2 MinBufferCount Attribute

This numeric attribute specifies the minimum number of message buffers required by the worker for a port. The **Worker Interface** allows the worker code (typically in the **run** method) to **take** a buffer from an input port, and ask for a new buffer for that port while retaining ownership of the previous buffer from that port. This behavior requires the infrastructure to provide at least two buffers for that port.

The buffers are guaranteed to exist, so the worker can keep one that is full while waiting for another one and be guaranteed that it will be able to receive a second buffer while holding onto the first (i.e. not deadlock). If the worker wants to compare the previous buffer with the next buffer, it requires there to be two, but it does *not* require that when the first one arrives, the second one is also full of data ready to process.

This attribute informs the infrastructure as to the minimum buffering requirements of the worker implementation for that port. The default value is one. *This attribute should not be used to tune the buffer count for performance. It should only specify the actual minimum requirements for the correct function of the worker.*

MinBufferCount only applies to input ports, since there is no way to use the current worker interface to depend on the existence of multiple output buffers.

This value has no effect on the behavior or interface for run conditions. See the [RunCondition](#) and [run worker method](#) sections for more details. In particular it does not wait for more than one buffer to be available before running the worker.

The stronger contract, where the worker will only be run when *all* of the required buffers for a port are available, is *not* supported.

3.3 *Attributes of Slave Elements in the OWD*

While the **slave** *attribute* can be conveniently used to indicate that this worker is a proxy for a single slave, the **slave** *element* is used when there are multiple slaves. One **slave** *element* is required for each such slave, and it specifies both the worker for the slave, and the name that will be used to refer to that slave in the proxy worker's code. It is possible and allowed for the proxy to have multiple slaves based on the same worker.

3.3.1 *Worker Attribute*

This attribute specifies the worker for this slave. Its syntax is the same as the slave attribute described above. The attribute's value is the name of the worker to used as a slave of this proxy and may have a package prefix, denoted with periods, if the slave worker is not in the same namespace as this proxy worker. The worker name must include the authoring model suffix.

3.3.2 *Name Attribute*

This optional attribute specifies the name used in the proxy worker's source code to reference this particular slave. If not specified, it is set to the worker name without the model suffix and if there is more than one slave using the same worker, a zero-based ordinal will be added as a suffix, similar to instance names in application XML.

4 The RCC Worker Interface

This section defines the interface between the worker and its container: the API for this authoring model. The term **worker method** is used as a shorthand and language-neutral term for what is a member function in C++ or a worker entry point function in C. Methods in C++ have an implicit **this** argument that is hidden by the language. C language RCC workers have the first argument to all worker methods as an explicit **self** argument. This is a pointer to a structure containing context and state information for the worker as well as C function pointers to container methods

When discussing a worker's runtime behavior, the term **worker** is sometimes used as a runtime *instance* of the worker in contrast to referring to the source code that is written for the worker as a component implementation.

RCC worker code must avoid the prefixes **OCPI** and **RCC** (even without a trailing underscore) for compile-time constants and types as these are used by the authoring model. The RCC authoring model also specifies that all generated macros are upper case, and all generated data types are capitalized and mixed CamelCase. User code is recommended to follow these conventions, but is not required to do so.

The worker interface consists of control operation methods whose behavior is defined in the **Control Plane Introduction** section of the CDG. In addition, there is a required **run** method that supports the event-driven execution model defined in the **Software Execution Model** section of that same document. The **run** method is the only required method and all the other worker methods are optional. All processing of the worker occurs in the context of these methods. There are two categories of methods:

- **Worker methods** represent functionality of the worker, to be called by the container, and which may have default implementations. These include the **run** method, and the lifecycle control operation methods: **initialize**, **start**, **stop**, and **release**.
- **Container methods** represent functionality of the container, to be called by the worker, such as changing run conditions, and accessing ports and buffers. All C language container methods are dispatched through function pointers in the **container** member of the **RCCWorker** structure. C++ container methods are accessible member functions inherited from the worker's base class.

Default methods are the behaviors that are executed if the user supplies no code for a worker method. Default methods for C-language workers are indicated by **NULL** pointers for those methods in the [RCCDispatch structure](#). For C++, they are simply the methods in the base class.

Several integral typedefs and constants are defined and used throughout the interface:

- **RCCBoolean** is aliased to **uint8_t**, to match the defined size in property space and message layouts.
- **RCCChar** is a signed 8 bit type to represent property values of type "Char".
- **RCCOrdinal** is used for ordinals for ports, operations, and properties.

- **RCCPortMask** indicates a proper subset of a worker's ports, with ports indicated using, e.g.: **(1 << port1_ordinal) | (1 << port2_ordinal)**.
- Ordinals for ports are generated as enumeration constants **<WORKER>_<PORT>**.
- **RCC_NO_PORTS** is a mask of type **RCCPortMask** indicating no ports.
- **RCC_ALL_PORTS** is a mask of type **RCCPortMask** indicating all ports.

More complex data types used in worker and container methods are described in the following sections where they are used.

4.1 The RCC Execution Model

The CDG describes the fundamentals of software execution models for OpenCPI software authoring models. RCC workers executing in their containers operate according to this model, with the details described in the [RCC Worker Interface](#) section below.

The container executes workers such that all execution threads are supplied by the container. Thus there is no need or possibility for workers to create threads. Containers may arrange for workers to run concurrently, each in its own thread, or run one at a time with all workers in the container running in a single thread.

A simple non-preemptive single-threaded container implementation would have a loop, testing **run conditions**, and calling workers' **run** methods. A more complex environment might run workers in different threads for purposes of time preemption, prioritization, etc. This worker execution model allows a variety of container execution models while keeping the worker code simple.

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 full function GPP environments, providing a test environment and a migration path to more minimal embedded environments such as a single-threaded environment with no real operating system at all (i.e. "bare metal"). Such a minimal environment is possible, although none are currently supported.

The execution source code for an RCC worker should be written for a multi-threaded environment. Since the interfaces between the worker and container are already implicitly thread-safe, this concern mostly applies to workers calling library functions. Library functions such as **localtime** or **strtok** should be avoided since they are not thread-safe. Library functions such as **setenv** or even **exit**, should never be called since they modify the environment for other workers. More information about appropriate library functions is in the [RCC Local Services](#) section below.

4.2 Worker Methods: Called by the Container, Implemented by the Worker.

This section describes the methods that a worker may implement with only one being mandatory: **run**. All other methods are optional and have default behavior. These other methods are **control operations** that perform lifecycle state transitions. The transitions are described in the control plane introduction section of the CDG. All processing of the worker occurs in the context of these worker methods.

For C++, the code generation tools create a custom base class that the actual worker class inherits. The derived class that implements the worker is declared by the worker author directly in their source code. This derived class inherits the custom base class, but can otherwise contain any other member functions and data members, with the caveat that they do not shadow certain members in the generated custom base class (which are specifically mentioned below). All worker methods in C++ are member functions and this is where all worker processing takes place.

4.2.1 RCCWorker Structure Type – C Language Only

This structure type, a **typedef** name, represents the visible state of a worker. The container creates this structure (defined in **RCC_Worker.h**) with any content or member ordering as long as the documented members are supported. The structure 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 code. A pointer to this structure is the first argument to all C language worker methods, called **self**.

Workers (code in worker methods) access property values via the **properties** member of the **RCCWorker** structure, whose type is **void ***. Its usage is described in the [Property Access](#) section below.

The defined members of the **RCCWorker** structure are described in the table below. Some example code that uses these various members of **RCCWorker** is:

```
XyzProperties props = self->properties; // void * needs no cast

MyState *state = self->memory;           // void * needs no cast

self->container.setError("stuff happened");

if (self->runCondition == &mySearchCondition) ...

if (self->connectedPorts & (1 <= XYZ_IN)) ...

uint32_t *msgdata = self->ports[XYZ_IN].current.data;
```

Table 2: Members of the *RCCWorker* Structure for C Workers

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 declared in the OCS and OWD. The value is NULL if there are no such properties.
memories	void *const *const	container	An array of const pointers to the memory resources requested by the worker in the memSizes member of the RCCDispatch structure. Any memories that are not read-only are initialized to zero before a worker executes any method.
memory	void *const	container	A pointer to the memory resource as requested in the memSize member of the RCCDispatch structure.
container	const RCCContainer	container	A dispatch table of container method/function pointers.
runCondition	RCCRunCondition *	worker	Initialized from the RCCDispatch runCondition member. Checked by container after calling the start method.
connectedPorts	RCCPortMask	container	A mask indicating which ports are connected. A worker can check this to see if an optional port is connected.
ports	RCCPort[]	varies by member	An array of RCCPort structures, indexed by port ordinals.

4.2.2 The Worker Derived Class – C++ Language Only

All the worker methods are member functions of the worker's derived class. This class inherits a base class specifically generated for the worker and defined in the **gen/<worker>-worker.hh** file. Worker methods that must exist (based on the OWD), are declared pure virtual in this generated base class. Container methods are accessible member functions of the base class. There are several accessible data

members of the generated base class that can be used in worker method code. One is **m_properties**, which is a structure containing members for property values. Its usage is described in the [Property Access](#) section below.

There are also data members in the base class for each port. These are described in the [Port Management Data Members and Methods](#) section below.

4.2.3 RCCResult Enumeration Type — C and C++ Languages

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 described in the following table:

Table 3: RCCResult Return Values

Enumeration Identifier	Description
RCC_OK	The worker method succeeded without error.
RCC_ERROR	The worker method did not succeed, but the error is not fatal to worker or container: the method may be retried if defined to allow this.
RCC_FATAL	The worker should never run again and is non-functional. The container or other workers may be damaged. The worker is in an unusable state. The container may know that it, or other workers are protected from damage, but the worker indicates this condition in case there is no such protection.
RCC_DONE	The worker needs no more execution. This is a normal completion. The worker is entering the finished state. This allows the container release resources (such as I/O buffers) prior to the worker instance being destroyed or reused. Any messages remaining in input buffers will not be processed and may be discarded. Properties are still accessible.
RCC_ADVANCE	The worker is requesting that all ports that were ready when the run method was entered be <i>advanced</i> (applies only to the run method).
RCC_ADVANCE_DONE	The worker is requesting that all ports be advanced (run method only) and declaring that it is also “done”. The worker is entering the finished state like RCC_DONE .

These return 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 have also set a descriptive error message via the **setError** container method described below.

4.2.4 Worker Initializations

A worker may initialize its state in several places, as appropriate. Each is described below in detail, but here is a summary of the different places it may be done and why:

- *C++ Worker Constructor*: convenient for one time initializations that do not depend on initial property values and do not generate error conditions.
- *C++ Member Data Initializers*: convenient for one time member data initializations that do not depend on initial property values and do not generate error conditions.
- *The **initialize** control operation*: one time initializations that do not depend on initial property values for C, and those that might generate errors for C++.
- *The **start** control operation*: initializations that should happen when the worker is first started as *well* as when it is resumed after being suspended.
- *The **firstRun** test in the run method*: one-time initializations that depend on initial property values. See [firstRun container method](#).

Guidance to choose where initialization should take place is, in priority order:

1. In C++, use the constructor if the initializations do not depend on property values and do not generate errors, otherwise
2. Use the [firstRun container method](#) unless it is critical that the first execution of the run method should not be slower than subsequent executions.

If neither #1 nor #2 is suitable:

3. Use **initialize** if the initializations do not depend on property values.
4. Use **start** if the initializations depend on property values or must be repeated when the worker is resumed after being suspended.

4.2.5 **initialize** — Worker Method

This worker method implements the **initialize** control operation as defined in the CDG and it is optional. It cannot depend on any property settings. A worker should implement this method under these conditions, for one-time initializations that *do not* depend on properties:

- There are initialization errors that can be reported to the container.
- There is significant processing work involved in the initialization.
- There are significant resource allocations performed during initialization.

If none of the above conditions are true, this method can be unimplemented and the default implementation will be used.

Using the **initialize** method will allow the **run** or **start** methods to be more consistent in execution time. Without an **initialize**, all one-time initialization must be done on the first invocation of **start**, or if **start** is unimplemented, the first invocation of **run**. Sometimes this is unavoidable.

For one-time initializations that depend on property values, the [firstRun container method](#) can be called in the **run** method to know when it is being called for the first time.

4.2.5.1 Synopsis

```
static RCCResult initialize(RCCWorker *self); // C language
RCCResult initialize(); // C++ language
```

4.2.5.2 Returns

This method shall return a **RCCResult** value.

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.

4.2.6 **start** — Worker Method

This method implements the **start** control operation as defined in the CDG and is optional. Upon successful completion, the worker is in the **operating** state or the **finished** state. This operation is called for the first time after initialization and after initial property settings are done. It may also be called after a successful **stop** operation, to *resume* execution. Thus any truly one-time initializations must be protected by some means in the worker implementation to ensure they are executed only once.

4.2.6.1 Synopsis

```
static RCCResult start(RCCWorker *self); // C language
RCCResult start(); // C++ language
```

4.2.6.2 Returns

This method shall return a **RCCResult** value.

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**. It shall return **RCC_DONE** if no worker execution should proceed, to indicate that it is entering the **finished** state. Otherwise it shall return **RCC_OK** if normal worker execution should proceed.

Returning **RCC_DONE** indicates to the container that the worker will never require further 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.

4.2.7 **stop** — Worker Method

This method implements the **stop** control operation as described in the CDG. Upon successful completion, the worker is in the **suspended** state. It will be also called before the **release** method, and before destruction, if the worker is in the **operating** state.

4.2.7.1 Synopsis

```
static RCCResult stop(RCCWorker *self); // C language
RCCResult stop(); // C++ language
```

4.2.7.2 Returns

This method shall return a **RCCResult** value.

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**.

4.2.8 **release** — Worker Method

This method implements the **release** control operation as described in the CDG. After successful completion, the worker instance is in the **exists** state. This method is needed when there are resources allocated by the worker in any other worker methods, such as heap allocations or file handles.

4.2.8.1 Synopsis

```
static RCCResult release(RCCWorker *self); // C language
RCCResult release(); // C++ language
```

4.2.8.2 Returns

This method shall return a **RCCResult** value.

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**.

4.2.9 **run** — Worker Method

The **run** method requests that the worker perform its normal computation. The container only calls this method when the worker's *run condition* is satisfied.

4.2.9.1 Synopsis

```
static RCCResult run(RCCWorker *self,
                    RCCBoolean timedOut,
                    RCCBoolean *newRunCondition); // C language
RCCResult run(bool timedOut); // C++ language
```

4.2.9.2 Behavior

The **run** method shall perform the worker's computational function and return a result. This method may use information in its property structure, the state of its ports, and its requested local and global/persistent member data to decide what to do. When there are data ports, this typically involves using messages in buffers at input ports to produce messages in buffers at output ports.

The **timedOut** boolean input argument indicates when the **run** method is being invoked due to time passing (the **usecs** value of the run condition) and *not* due to the ports specified in the run condition being ready.

C Language: The **run** method may change the run condition by setting a TRUE value to the location indicated by the **newRunCondition** output argument, and setting a pointer to a new run condition in the **runCondition** member of **RCCWorker**. The **runCondition** member of **RCCWorker** is initially set to the value in the **RCCDispatch** structure (including **NULL** to indicate the default run condition).

C++ Language: The **run** method may change the run condition using the [setRunCondition container method](#) and retrieve it using the [getRunCondition\(\) container method](#).

If the current **runCondition** is set to **NULL**, the default run condition is restored. The run condition for a worker is initially set to this default. Since the worker code is managing the run conditions (passed by pointer), it can use it as a convenient “state variable” when its execution modes each have a different run condition.

Code in the **run** method accesses information about ports and current buffers by accessing objects/structures that are ports in the Worker object.

See the [port management](#) and [buffer management](#) sections below for container methods and port data members used to process input messages and create output messages.

The **firstRun** container method may be tested for one-time initializations in the **run** method.

4.2.9.3 Returns

This method shall return a **RCCResult** value. The most common value is **RCC_ADVANCE**, which indicates that all ports should be advanced that were ready on entry to the **run** method and were not subject to container methods called since then (e.g. advance, request etc. described below).

It should return **RCC_OK** or **RCC_ADVANCE** if normal worker execution should proceed (the **run** method will be called again when the run condition is true).

The run method should return **RCC_DONE** or **RCC_ADVANCE_DONE** if no further worker execution should happen.

See [RCCResult Enumeration Type](#) for a description of all allowed values.

4.3 Container Methods, Called by the Worker.

These methods are what a worker calls to invoke functionality provided by the container. The container methods are in three categories:

- container scope (this section),
- port scope (next section)
- buffer scope (following section).

The use of all container methods is optional, and typically unneeded for simple workers that only use run conditions and the **run** method. When used these provide additional flexibility and functionality in message handling.

All container methods are non-blocking.

C Language: container methods are usually accessed using function pointer members in the container structure member of the **RCCWorker** structure, e.g.:

```
self->container.setError(...)
```

C++ Language: container methods exist in an accessible base class for the worker and are called directly. Container methods relating to ports and buffers are methods of the worker's member data objects for ports, rather than on the worker (or container) objects.

4.3.1 **firstRun** — Container Method

This method is used by the worker to determine when the **run** worker method is being called for the first time. It can be used for initializations that must be done once after all initial property values are established. It can simply be used when it is more convenient or concise than other methods, e.g. (in C++):

```
RCCResult run(bool timedOut) {  
    if (firstRun())  
        do_this_once(); // whatever must be done only once  
}
```

Remember that the **start** control operation may be called more than once if the worker is suspended (using the **stop** control operation) and subsequently resumed using **start** a second time. The **initialize** control operation is also called exactly once, but is called *before* initial property values are established.

If the **firstRun** container method is called from other than the **run** method, its behavior is undefined.

4.3.1.1 Synopsis

```
RCCBoolean self->firstRun; // C Language, using structure member  
bool firstRun();          // C++ Language
```

4.3.1.2 Returns

This method (C++) or structure access (C) returns a boolean which is true when the run method is being called for the first time.

4.3.2 **setError** — Container Method

This method is used by the worker to convey an error message associated with returning **RCC_ERROR** or **RCC_FATAL**. It has semantics similar to `printf`.

4.3.2.1 Synopsis

```
RCCResult (*setError)(const char *fmt, ...); // C Language
RCCResult setError(const char *fmt, ...);    // C++ Language
```

4.3.2.2 Returns

RCC_ERROR is always returned, allowing a worker to simultaneously set the error and return the error indication. E.g., a worker method detecting an error could (in C) do:

```
return self->container.setError("Failure due to %d", arg);
```

Or, in C++:

```
return setError("Failure due to %d", arg);
```

Workers that must return **RCC_FATAL**, may call this prior to returning that value.

4.3.3 **log** — Container Method

This method is used by the worker to make an entry into the OpenCPI system log. After an initial argument to specify the log level associated with the message, it has semantics similar to **printf**. It is currently only supported in C++. A newline is automatically appended to all log messages so none is required in the message content.

Log levels are set on the command line of **ocpirun** or **ocpiserve**, and may also be set in the ACL. Using a log level of zero in this method will nearly always show in the system log. Log levels of 10 and above are used and shown only for detailed debugging output.

Symbolic constants for log levels are not available in RCC workers (yet).

4.3.3.1 Synopsis

```
void log(unsigned level, const char *fmt, ...); // C++ Language
```

4.3.4 **willLog** — Container Method

This method returns whether a certain level of logging will in fact be put into the system log. It can be used to conditionalize code that should only be run if certain level of logging is enabled. It is only available in C++.

4.3.4.1 Synopsis

```
bool willLog(unsigned level); // C++ Language
```

4.3.4.2 Returns

This method returns **true** if the logging level supplied in its argument is enabled for inclusion in the system in the current environment.

4.3.5 *time/getTime* — Container Method

This method retrieves the time of day as a 64 bit unsigned number that represents GPS time in units of 2^{32} seconds (~233 ps). The most significant 32 bits hold GPS time in seconds. The accuracy of this time is dependent on the container implementation.

Note that GPS time is monotonic, as opposed to UTC time (the POSIX standard), which is subject to leap seconds etc.

4.3.5.1 Synopsis

```
RCCTime (*time)(); // C Language
RCCTime getTime(); // C++ Language
```

4.3.5.2 Returns

This function returns GPS time in units of 2^{32} seconds.

4.3.6 The **RCC::RunCondition** C++ Class and **RCCRunCondition** C Structure

This class is used by the container to determine when it is appropriate to invoke the **run** method. The **RCC::RunCondition** class has convenience constructors, methods, and worker-writable data members. In most cases the convenience constructors are sufficient and no member data or method call is required. The **getRunCondition** and **setRunCondition** *container* methods described below are used to set and get the current run condition objects for the worker. This class, and those container methods, are only needed when the default run condition is not appropriate.

The **RCC::PortMask** (C++)/**RCCPortMask** (C) type in run conditions specifies a set of ports that must be ready at the same time. A mask is considered “true” when *all* ports whose bit is set in the mask (i.e. **mask & (1 << port_ordinal) != 0**) are ready or are not connected. The **RCC_ALL_PORTS** constant indicates all a worker's ports.

Table 4: RunCondition Constructors

RunCondition()	Specify the default run condition: no timeout, and all connected ports must be ready. If the worker has no ports, it will always run. The constructor is equivalent to: RunCondition(RCC_ALL_PORTS, RCC_NO_PORTS)
RunCondition (RCC::PortMask first, ...)	Specify a list of port masks as variable arguments that is terminated by the value RCC_NO_PORTS . No timeouts are indicated. Each mask indicates a set of ports. The run condition is considered <i>true</i> when <i>any</i> of the masks is true. RunCondition(RCC_NO_PORTS) indicates: <i>never run</i> .
RunCondition (RCC::PortMask *masks, uint32_t usecs = 0, bool timeout = false)	Specify a pointer to an array of masks (terminated by RCC_NO_PORTS) as well as the timeout, with defaults for timeout arguments. The masks in the array are copied during construction. RunCondition(NULL) indicates: <i>always run</i> .

In all cases, a port's readiness is based on a single buffer being available, even if more than one buffer exists, and even if the **MinBufferCount** attribute for the port in the OWD is set to a value greater than 1. Knowing that more than one buffer exists at a port does *not* imply that more than one will be available in any given invocation of the **run** method.

A worker using **RunCondition** objects typically declares them as member data that are configured in member initializers in the worker's constructor. For example, in the following code a run condition data member is declared, and initialized appropriately. Then it is used dynamically as conditions warrant.

```
class XyzWorker : public XyzWorkerBase {
    RunCondition m_aRunCondition;
    XyzWorker() : m_aRunCondition(1 << Xyz_Inport, RCC_NO_PORTS) {
    }
    RCCResult run(bool timedOut) {
        if (need_to_change_run_condition)
            setRunCondition(&m_aRunCondition);
        else if (need_default_run_condition)
            setRunCondition(NULL);
    }
}
```

The following table describes the accessible methods that can be used when the convenience constructors above are not sufficient to specify or control RunConditions. All have no return value (i.e. are void).

Table 5: RunCondition Methods

Method Name and Arguments	Member Description
disableTimeout()	Disable the timeout aspect of the run conditions. The timeout time value (usecs) is unchanged.
enableTimeout()	Enable the timeout aspect of the run condition, using the previously specified usecs value.
enableTimeout (uint32_t usecs)	Enable the timeout aspect of the run condition, supplying a new usecs value, which will be stored in the object.
setTimeout (uint32_t usecs)	Set the timeout usecs value of the run condition, which will not necessarily enable the timeout if not enabled.
setPortMasks (RCCPortMask *)	Set the port masks for the run condition from the array pointed to by the argument, which must be terminated by RCC_NO_PORTS . The array is copied into the run condition.
setPortMasks (RCCPortMask, ...)	Use a variable argument list to set the port masks of the run condition. The arguments must be terminated by RCC_NO_PORTS .

The following table describes the accessible data members of the RunCondition structure. They apply to the C **RCCRunCondition** structure as well as the C++ **OCPI::RCC::RunCondition** class.

Table 6: RunCondition Members — C and C++

Member Name	Member Data Type	Member Description
portMasks	RCCPortMask *	A pointer to an array of port masks, terminated by RCC_NO_PORTS , each of which indicates a bit-mask of port readiness. The run condition is considered <i>true</i> when <i>any</i> of the masks is true. A mask is <i>true</i> when all indicated ports are ready (logical AND of port readiness). If the pointer itself is NULL , the run condition is always true. A mask bit set for an unconnected port is ignored: the default run condition can be used with unconnected ports.
timeout	bool	Indicates that the usecs member determines when enough time has passed to make the run condition true. This value is used to enable or disable the timeout, without changing <i>usecs</i> .
usecs	uint32_t	When enabled by the timeout member, if this amount of time has passed (in microseconds) since the run method was last <i>entered</i> , the run condition is true.

The overall run condition is the logical OR of the **portMasks** and the timeout. For C language workers, if the worker offers no **run** method in its **RCCDispatch** structure (see below), run conditions are ignored. If the **portMasks** member is **NULL**, it indicates that no port readiness check is performed and the run condition is always true. The default run condition is a single port mask with all ports enabled. Typical combinations are:

Table 7: Run Condition Combinations

Shorthand	Port mask	Timeout	RunCondition Description
Always run	<code>portMasks == NULL</code>	ignored	the worker is always ready, no timeout
Run when data ports are ready	<code>portMasks != NULL</code>	False	Run condition is TRUE when any mask is true. If there are no masks (<code>portMasks[0] == 0</code>), then the run condition is FALSE.
Run when data ports are ready or timeout.	<code>portMasks != NULL</code> and <code>portMasks[0] != RCC_NO_PORTS</code>	True	Run condition is TRUE when any mask is true OR if the timeout expires. The timeout will take effect if time passes without any mask being satisfied.
Periodic execution	<code>portMasks != NULL</code> and <code>portMasks[0] == RCC_NO_PORTS</code>	True	Since <code>portMasks[0] == RCC_NO_PORTS</code> , port masks can never be true, thus this establishes periodic execution independent of port readiness.

4.3.7 *getRunCondition* — Container Method — C++ Language Only

This method is used by the worker to retrieve its current run condition. If the current run condition is specified to be the default, **NULL** is returned. Otherwise the pointer returned is the most recent one passed from the worker to the container in the **setRunCondition** method. Because the value returned is the pointer that the worker provided in a previous call to **setRunCondition**, it can be used as a state variable in the worker, e.g.:

```
if (getRunCondition() == &m_myruncond2)
    // do things for state2
```

4.3.7.1 Synopsis

```
RCC::RunCondition *getRunCondition();
```

4.3.7.2 Returns

This function returns a pointer to the run condition most recently set by the worker using the **setRunCondition** call, or **NULL**.

4.3.8 *setRunCondition* — Container Method — C++ Language Only

This container method is used by the worker to set a new run condition. If the run condition supplied is **NULL**, the default run condition becomes active, which is to wait for all ports to be ready, with no timeout. If the worker needs to start execution with a non-default run condition, it can call this container method in its constructor, or in its **initialize** or **start** methods.

When not **NULL**, the **RCC::RunCondition** pointer supplied as the argument is stored in the container as a pointer and is expected to remain valid until a subsequent call to

this method or until the worker is destroyed. The object referenced by this pointer is not copied and must not be modified while it is the current run condition.

4.3.8.1 Synopsis

```
void setRunCondition(RCC::RunCondition *rc);
```

4.3.8.2 Returns

No return value.

4.4 Port Management Data Members and Methods

This section describes port management methods. The **current** buffer at a port is in fact part of the port object. This means methods on the **current** buffer are in fact methods on the port itself. Otherwise buffer methods are used on specific buffer objects. Those are described in the next section.

In the C language, port management methods are directly accessed function pointer members in the **RCCWorker** data structure, like all container methods. Each has a **RCCPort** pointer as an argument.

In C++, each port is a member data object (of the worker object) whose name is the name of port (from the OCS). Where port management methods are methods on that object.

As an example, to *advance* a port in C, and optionally access the data of the now-current buffer, is below:

```
RCCPort *inport = &self->ports[MY_IN_PORT];

if (self->container.advance(inport)) {
    void *p = inport->current.data;
    ...
}
```

In C++ this would be:

```
if (inport.advance()) {
    void *p = inport.data();
    ...
}
```

Note that in C++, the current buffer object is *inherited* by the port object, so the port data member is the object used to access the current buffer's **data()** method.

The **run** method can indicate that all ports should be advanced by the special return value **RCC_ADVANCE**. It can also indicate disposition of buffers and ports by using the **release**, **send**, **request**, **take**, or **advance** container methods.

C Language: The **run** method accesses information about current buffers through members of each port's **RCCPort** structure member in the **ports** array in the **RCCWorker** structure (e.g. **self->ports[n]**, where **n** is the ordinal for the port). The **current.data** member is the pointer to the message data for both input and output ports. For input ports, the **input.length** structure member of the **RCCPort** is the length of bytes of the message in the current buffer, and **input.u.operation** is the opcode for the current input message. For output ports, the **output.length** is the length of bytes of the message in the current buffer, and **output.u.operation** is the opcode for the current output message.

C++ Language: The **run** method accesses port status and buffer contents using the worker's object port data member methods.

If a port is in the worker's run condition, then it can assume a current buffer is present at that port when the run method is entered. If not, the worker can test whether there is a current buffer; in C, the port's **current.data** member is non-**NULL**, in C++, the port's **hasBuffer()** returns true. In either case, a port's readiness only depends on a single buffer being available. Even if multiple buffers exist, the port readiness is only based on the availability of a single buffer.

In the C language, and in some cases for C++, the message buffers are accessed as a raw array of bytes, where the worker is responsible for correctly formatting the message in the buffer. The **request**, **send**, **setDefaultLength**, and **take** methods in this section are used only for this raw buffer access.

In C++, there is alternative set of methods that format the message correctly for the worker, and these methods should be preferred in nearly all cases. These are described in the [Accessing Messages in C++ Language Workers](#) section below.

4.4.1 **advance** — Release Current Buffer, and Request Another

This method releases the current buffer at a port, *and* requests that a new buffer be made available as the current buffer on the port. This is a convenience/efficiency combination of the **release** (the current buffer) and **request** (a new current buffer) methods described below.

An optional minimum size in bytes may be requested (0 is the default in C++, 0 can be supplied in C). Zero means no minimum size is requested. The default size is normally based on the protocol at the port. When there is no protocol this argument allows the worker to specify a size that must be satisfied, or an exception is thrown.

4.4.1.1 Synopsis

```
RCCBoolean (*advance)(RCCPort *port, size_t minSize); // C language
bool Port::advance(size_t minSize = 0);                // C++
```

4.4.1.2 Returns

A boolean value is returned indicating whether the **request** aspect of this method was immediately satisfied.

4.4.2 **hasBuffer** — Query a Port for Whether It Has a Current Buffer-to-release

This method returns whether a port has a current buffer. If a port is already part of a worker's run condition, and it is in every port mask of the run condition, it can be assumed to have a current buffer whenever the worker is run. This is the case for 90% of all workers. *This method is only needed and used when a port is not in the run condition*, as might be the case for a port that received some exceptional condition message. No such testing is needed when the port is in the run condition since port readiness is implicit when the run condition is satisfied.

In C, this is a port structure member access. In C++ it is a port method.

4.4.2.1 Synopsis

```
Port->current.data != NULL      // C Language  
bool RCCUserPort::hasBuffer();  // C++ Language
```

4.4.2.2 Returns

For C++, a **bool** value is returned indicating whether the port has a current buffer.

4.4.3 *isConnected* — Query a Port for Being Connected

This method returns whether a port is currently connected or not. Component specifications (OCS) indicate whether each port of a component (and thus all workers) must be connected before workers are started or run. A port is considered “optional” for a worker if the worker can tolerate the port *not* being connected. If a port is *not* optional, the worker code can assume it is connected, and this method would not be needed or used.

In C, this is a Worker (self) structure member access. In C++ it is a port method.

4.4.3.1 Synopsis

```
self->connectedPorts & (1 << PortOrdinal); // C Language  
bool RCCUserPort::isConnected();           // C++ Language
```

4.4.3.2 Returns

For C++, a **bool** value is returned indicating whether the port is connected.

4.4.4 *ordinal* — Obtain an Ordinal for a Port

This method returns the ordinal of a port, which may be used wherever port ordinals are required such as in port masks for run conditions.

In C, this is simply an enumeration value. In C++ it is a port method.

4.4.4.1 Synopsis

```
<WORKER>_<PORTNAME>; // C Language  
RCCOrdinal RCCUserPort::ordinal() const; // C++ Language
```

4.4.4.2 Returns

For C++, a **bool** value is returned indicating whether the port is connected.

4.4.5 *request* — Request a New Buffer

This method *requests* that a new buffer be made available as the current buffer on a port. If the port already has a current buffer, the request is considered satisfied. This request indicates to the container that it should make a buffer available when possible. Without an explicit request, the container may not make a buffer available since that would dedicate resources when they may not be needed. An implicit request is made for all ports that are part of the current run condition.

An optional minimum size may be requested.

Note that this method is not used or needed when ports are **advanced**, only when buffers are being explicitly managed (e.g. released, etc.).

4.4.5.1 Synopsis

```
RCCBoolean (*request)(RCCPort *port, size_t minSize); // C Language
bool Port(size_t minSize = 0);                       //C++ language
```

4.4.5.2 Returns

A boolean value is returned indicating whether the request was immediately satisfied.

4.4.6 **send** — Send an Input Buffer on an Output Port

This method sends an input buffer on an *output* port. If the buffer is a current buffer for a port, this buffer will no longer be that port's current buffer. Buffer ownership passes back to the container. This method is used to effect zero-copy transfer of a message from an input port to an output port.

C Language: The operation and message length are supplied as arguments.

C++ Language: The operation and message length are attributes of the buffer object and can be changed before it is sent using the **setOpCode** and/or **setLength** methods.

The sent buffer is either a current buffer of an input port or a buffer taken from an input port.

4.4.6.1 Synopsis

```
void (*send)(RCCPort* port,
             RCCBuffer* buffer,
             RCCOrdinal op,
             uint32_t length); // C Language
void RCCPort::send(RCC::Buffer &buffer); // C++ Language
```

4.4.6.2 Returns

Nothing is returned.

4.4.7 **setDefaultLength** — Set Default Message Length at an Output Port — C++ Language Only

This method sets the default message length in bytes for a port. It is useful when messages produced at a port will always be the same size.

This method is used only when the message content is manipulated as a raw untyped buffer, rather than using the message content access methods described below.

The initial setting of the default message length is the length of the buffer.

This method is only available in C++.

4.4.7.1 Synopsis

```
void RCCUserPort::setDefaultLength(size_t length); // C++ Language
```

4.4.7.2 Returns

Nothing.

4.4.8 **setDefaultOpCode** — Set Default Opcode at Output Port — C++ Language Only

This method sets the default **opCode** for an output port. It is useful when messages produced at a port will typically have the same opcode, among those defined in the protocol.

The initial setting of the default opcode is zero (indicating the first operation specified in the associated protocol). Setting the default opCode does not prevent setting the opCode on a per-message basis at any time during execution.

This method is only available in C++.

4.4.8.1 Synopsis

```
void RCCUserPort::setDefaultOpCode(RCCOpCode opcode); // C++ only
```

4.4.8.2 Returns

Nothing.

4.4.9 **take** — Take a Buffer from an Input Port

This method *takes* the current buffer from a port, and optionally releases a previously taken buffer. This method is used when workers need to maintain a history of one or more previous buffers while still requesting new buffers, e.g. for *sliding window* algorithms. This method only applies to input ports.

The optional buffer to be released is provided by a possibly **NULL** pointer to a buffer. This buffer-to-release argument is a convenience feature of the API to allow cycling of buffers in a single call, e.g.:

```
class XyzWorker {
    RCCUserBuffer *m_prev;           // remember buff from previous run
    XyzWorker() : m_prev(NULL) {}
    void process_curr_and_prev(RCCUserBuffer *curr,
                              RCCUserBuffer *prev) {}

    run() {
        process_curr_and_prev(&in, m_prev);
        m_prev = &in.take(m_prev);
    }
}
```

In C, while we logically consider this a port method, it is actually a container method with three arguments:

- an **RCCPort** pointer indicating the port from which the buffer is taken
- an **RCCBuffer** pointer indicating, if not NULL, a buffer to release

- an **RCCBuffer** pointer as an output argument where the taken buffer structure will be copied (not the data, just the structure).

In C++, **take** is a port method with a single buffer-to-release argument and the taken buffer is simply returned via a reference.

Ownership of the taken buffer is passed to the worker. The current buffer now taken is no longer the current buffer. This method is used when the worker needs access to more than one buffer at a time from an input port. **Take** implies a **request** (to get another current buffer).

It is an error to call this method when the port does not have a current buffer.

4.4.9.1 Synopsis

```
void (*take)
(RCCPort* port, RCCBuffer* releaseBuffer,
 RCCBuffer* takenBuffer);           // C language
RCCUserBuffer &Port::take
(RCCBuffer *release = NULL);         // C++ Language
```

4.4.9.2 Returns

Nothing is returned in C, since the taken buffer object is copied to the location supplied by the **takenBuffer** argument. In C++, a reference to the taken buffer is returned.

4.5 Buffer Management Data Members and Methods

This section describes methods that apply to buffers separate from ports. As mentioned above, when a buffer is the current buffer at a port, the port object itself is normally used with these buffer methods. In C++, the port object inherits the buffer object that is the current buffer. In C, the current buffer is the **current** member of the RCCPort structure.

When methods deal with **opCodes**, they are dealing with an ordinal for the message in a buffer. The **opCode** identifies which type of message is in the buffer, among those defined by the protocol. OpCode values are zero-origin in the order of how the operations are defined in the protocol (OPS) XML file. Enumeration constants for the messages in a protocol are generated and are described in the [code generation](#) section.

In C++ buffer management is via buffer methods — member functions of buffer objects. In C, buffer management is either via container methods or direct access to structure members.

In the C language, and in some cases for C++, the message buffers are accessed as a raw array of bytes, where the worker is responsible for correctly formatting the message in the buffer. The methods in this section other than **setOpCode**, are used only for this raw buffer access.

In C++, there is alternative set of methods that format the message correctly for the worker, and these methods should be preferred in nearly all cases. These are described in the [Accessing Messages in C++ Language Workers](#) section below.

4.5.1 **checkLength** — Check Size of the Buffer — C++ Language Only

Check that a message of a given size in bytes will fit into the buffer, usually associated with an output port. An exception is thrown if the message will not fit. This is useful when the worker is creating a variable length message and wants to ensure it will fit into the available buffer. This check results in an error that the worker cannot check, and is intended to avoid unexpected buffer size mismatches, similar to the “assert” standard library function.

4.5.1.1 Synopsis

```
void RCCUserPort::checkLength(size_t neededSize); // C++ Language
```

4.5.1.2 Returns

Nothing.

4.5.2 **data** — Access the Raw Contents of the Buffer

This method returns a pointer to the raw data in a buffer. It is analogous to the **data** method in C++ STL container classes, although it has no type.

4.5.2.1 Synopsis

```
void *p = buffer->data;           // C Language
void *RCCUserBuffer::data();      // C++ Language
```

4.5.2.2 Returns

The value returned is a void pointer to the contents of the buffer.

4.5.3 **length** — Retrieve the Length of the Message in a Buffer – C++ Language Only

This method returns the number of bytes of the message in the buffer, which is typically an input buffer. This is C++ only. When using C, access to the length is via the current buffer at a port, e.g. **port->input.length**.

4.5.3.1 Synopsis

```
size_t RCCUserBuffer::length() const;    // C++ Language
```

4.5.3.2 Returns

The method returns the length in bytes of the message in the buffer.

4.5.4 **maxLength** — Retrieve the Maximum Available Space in the Buffer.

This method allows the worker to retrieve the actual size of the buffer, typically used on output buffers.

4.5.4.1 Synopsis

```
size_t len = buffer->maxLength;         // C language
size_t RCCUserBuffer::maxLength() const; // C++ language
```

4.5.4.2 Returns

The size of the buffer in bytes is returned.

4.5.5 **opCode** — Retrieve the OpCode of the Message in a Buffer

This method retrieves the *opCode* of the message in the buffer, typically an input buffer. This is C++ only. When using C, access to the *opCode* is via the current buffer at a port, e.g. **port->input.u.operation**

4.5.5.1 Synopsis

```
RCCOpCode op = port->input.u.operation;    // C language
RCCOpCode RCCUserBuffer::opCode() const;   // C++ Language
```

4.5.5.2 Returns

This method returns the *opCode* of the message in the buffer.

4.5.6 **release** — Release a Buffer

This method releases a buffer for reuse. If the buffer is the current buffer for a port, it will no longer be the current buffer. Buffer ownership passes back to the container.

Buffers for a port ***must be released in the order obtained***, per port. Note that this method is not used or needed when ports are “advanced”, only when buffers are obtained from a port using other port management functions such as ***take***.

Releasing a current buffer does not imply requesting a new current buffer: that request must be explicit. A release without a request might be useful if a worker is entering a mode where it no longer needs any more data from an input port or it no longer needs to send any more buffers on an output port.

In C, this is a function with the buffer pointer as an argument, e.g. called using:

```
self->container.release(buffer);
```

In C++ it is a buffer method invoked on the buffer object itself.

4.5.6.1 Synopsis

```
void (*release)(RCCBuffer* buffer);    // C Language  
void RCCUserBuffer::release();        // C++ Language
```

4.5.6.2 Returns

Nothing.

4.5.7 **setLength** — Set the Length of the Message in a Buffer — C++ Language Only

This method sets the length in bytes of the message in a buffer.

This is C++ only. When using C, access to the length of an output buffer is via the current buffer at a port, e.g. **port->output.u.length**, or via the **send** port method.

4.5.7.1 Synopsis

```
void RCCUserBuffer::setLength(size_t length);    // C++ Language
```

4.5.7.2 Returns

Nothing.

4.5.8 **setOpCode** — Set the OpCode for the Message in a Buffer — C++ Language Only

This method sets the opCode of the message in a buffer.

This is C++ only. When using C, access to the opCode of an output buffer is via the current buffer at a port, e.g. **port->output.u.operation**, or via the **send** port method.

4.5.8.1 Synopsis

```
void RCCUserBuffer::setOpCode(RCCOpCode op);    // C++ Language
```

4.5.8.2 Returns

Nothing.

4.5.9 **setInfo** — Set the Metadata Associated with a Buffer — C++ Language Only

This method is a convenient combination of setting both the **opCode** and the **length** of the message in a buffer. It is typically used on output buffers, when both opCode and length are being set at the same time.

4.5.9.1 Synopsis

```
void RCCUserBuffer::setInfo(RCCOpCode op, size_t length); // C++
```

4.5.9.2 Returns

Nothing.

4.5.10 **topLength** — Retrieve the Size of the Single Sequence in a Message — C++ Language Only

Normally, sequence lengths are retrieved using the message/operation access methods described below in the [Accessing the Contents of Messages](#) section. However, when message contents are accessed as an untyped raw buffer, this method allows the worker to retrieve the number of elements of a given size in the message without the container knowing the data type of the message. It performs the size conversion and error check in one API.

Untyped raw buffers are used when no protocol is defined for the port, e.g. when a worker needs to process any protocol or when it has some private mechanism for determining the contents of a message.

This method only applies to input buffers that contain messages consisting of a single sequence argument. It retrieves the length, *in elements*, of that sequence. The **elemSize** argument to **topLength** is the size in bytes of the elements of the sequence in the buffer. An error check is made to ensure that the size of the message in bytes is divisible by the **elemSize** value provided.

The message must consist of exactly one sequence for this method to be valid. Sequences can have zero elements.

This method is only available in C++.

4.5.10.1 Synopsis

```
size_t RCCUserBuffer::topLength(size_t elemSize); // C++ Language
```

4.5.10.2 Returns

For C++, this method returns a `size_t` value indicating the number of sequence elements in the message, given the size of elements known by the worker.

4.6 Accessing the Contents of Messages

The sections above described methods used to access the raw content and metadata (length and opCode) of messages in buffers. Those methods operate independent of the actual types of the data in the messages, and put the burden of accessing typed data on the worker code. I.e. the worker code would have to cast the pointer types and manually deal with the multiple data types for the arguments in a message.

Accessing the individual arguments (fields) in the payload for an input message (or setting the fields of an output message) is facilitated differently in C vs. C++.

4.6.1 Accessing Messages in C Language Workers

In C, data structures are generated that can be overlaid on buffers to access fields of a message up to and including the first variable length field (sequence or string). Each port has a set of data structures defined for the messages that may be in its buffers. These structures are described in the [code generation](#) section, but the basic pattern, for worker WXY, port PXY, with protocol PRXY, with operations Op1 and Op2, is:

```
typedef struct {
    ... fields in the message for Op1 messages ...
} WxyPxyOp1;
typedef struct {
    ... fields in the message for Op2 messages ...
} WxyPxyOp2;
```

Note that the structure typedefs are CamelCase, the opcode enumeration values are upper case, and the structure members are the case of the argument names in the protocol. In C, messages are accessed based on opCode like this:

```
switch (port->input.u.operation) {
case PRXY_OP1:
{
    WxyPxyOp1 *p = port->current.data;
    // use p-> to access the message structure
    break;
}
case PRXY_OP2:
{
    WxyPxyOp2 *p = port->current.data;
    // use p-> to access the message structure
    break;
}
}
```

If the messages are simply a sequence or array of one basic type (e.g. float), directly assigning the buffer pointer to that pointer type is sensible, e.g.:

```

switch (port->input.u.operation) {
case PRXY_OP1:
{
    float *p = port->current.data;
    // use p-> to access the message data
    break;
}
...

```

The major limitation of this scheme is that the structures only cover the fields of messages up to and including the first variable length field. After that, the worker must manually access the remaining fields using pointer arithmetic and casting. This covers most protocols used in simple systems, but is painful when the limitation is exceeded.

A second limitation for C language workers is that there is no error checking that the correct message structure is being used with its corresponding opCode.

4.6.2 Accessing Messages in C++ Language Workers

C++ workers have data members for each port whose name is the port's name in the OCS. Each such port data member has an accessor for the information for each possible message in the port's protocol. Each field of the payload for each operation also has an accessor. Accessing the **arg1** field of the **op1** operation message on the protocol for port **in**, would simply be:

```
in.op1().arg1()
```

These C++ message argument access methods return a **const** reference to the field for input ports, and return a non-**const** reference for output ports. They will generate an exception if used when the message in the buffer has the wrong opCode for the operation accessor used. E.g., if **in.op1()** is used when the current opcode is not **op1**, an exception occurs. If the argument type was **float**, then the accessors defined for **in.op1()** would be:

```

const float &arg1() const; // for arguments in an input buffer
float &arg1();             // for arguments in an output buffer

```

When the argument type is a sequence or array, the field accessor returns a reference to an *object* that has the following methods, which act like the same-named methods defined in the ISO C++ STL container types (assuming the data type is **float**):

```

const float *data() const; // pointer to access the floats on input
size_t size() const;       // number of floats available on input
float *data();             // pointer to access the floats on output
size_t resize(size_t n);   // to set the number of floats on output
size_t capacity() const;   // to get the available space on output

```

For arrays or sequences in output buffers, the **resize** method must be called for each message.

If **arg1** in the above example was a sequence of unsigned shorts, access to this field is:

```

uint16_t *vals = in.op1().arg1().data(); // pointer to data
size_t nvals = in.op1().arg1().size();    // number of elements

```

4.7 How a Worker Accesses its Properties

4.7.1 The Worker Property Structure

A structure type (e.g. **XYZProperties**) is generated for the worker, and placed in the **gen/XYZ_Worker.h** (for C) or **gen/xyz-worker.hh** (for C++) file, and contains members for each property that is not a parameter. Structure members for properties are declared **const**, unless the property is defined as either:

- **volatile** — meaning the worker can change it at any time, or
- **readable** but not **writable** or **initial** — meaning the worker can determine/set its value before **start**, but not after

Thus setting the value of a property that is neither of the above will result in a compilation error.

The name of the structure type (**typedef** name in C) is **XYZProperties** where “Xyz” is the *capitalized* worker name.

Properties that are scalar values (not sequences, arrays, structures or strings), are simple scalar structure members with the property's name. Properties that are arrays are declared as such in this structure, including multi-dimensional arrays. The structure member for properties that are sequences are themselves structures, with two members: a **uint32_t length** member indicating the number of valid elements present, and a **data** member which is an array of the maximum number of elements declared for the property in the **sequenceLength** attribute that defines the property (its maximum number of elements). Similarly, structure members for properties that are of type **string**, are arrays of type **char** of dimension **stringLength+1**, with the **+1** to leave room for the terminating zero.

Remember that for properties (as opposed to protocols), sequences and strings always have a maximum size (i.e. are bounded).

All property values are initialized to zero when the worker is created at runtime and the worker's code can always assume this. This initialization is in effect prior to the **initialize** control operation (and prior to the C++ constructor). After the **initialize** operation, but before the **start** operation (or the first invocation of the **run** method if there is no start method) any application-supplied **initial** values are set by control software.

For **readable** or **volatile** properties that are not declared as **writable** or **initial** (and thus are read-only for control software), the worker is expected to set any non-zero initial value before the worker is started. It may do this initialization either in the **initialize** method, in the C++ constructor, or perhaps by a notification that some other property has been written. Note that control software may cache the **readable** property values since they are not expected to change after the worker is started (i.e. they are not **volatile**).

4.7.2 The Accessing Worker Properties in C

The **properties** member of the **RCCWorker** structure (which is supplied as the ***self** argument to all worker methods), points to the memory that holds the worker's properties, with type **void ***. When needed by the worker to access property values, it is first converted to the structure pointer type that defines the layout of the worker's properties, and then used to access individual properties, e.g.:

```
XYZProperties *p = self->properties; // implicit cast from void*
p->myprop1;           // scalar property
p->myprop2.length;    // number of elements in sequence for myprop2
p->myprop2.data[n];    // the nth element of the myprop2 sequence
p->myprop2.data;       // the address of the elements in myprop2
p->myprop3[a][b][c];  // an element of a 3d array property
p->myprop4[0];         // the first character of the myprop4 string
strlen(p->myprop4)     // the length of the myprop4 string.
```

4.7.3 Accessing Worker Properties in C++

The worker object has a member function, **properties()**, that returns a reference to the XYZProperty structure. Worker code accesses properties that are not parameters by simply using this member function. The accesses similar to the C example above is:

```
XYZProperties &p = properties();
p.myprop1;           // scalar property
p.myprop2.length;    // number of elements in sequence for myprop2
p.myprop2.data[n];    // the nth element of the myprop2 sequence
p.myprop2.data;       // the address of the elements in myprop2
p.myprop3[a][b][c];  // an element of a 3d array property
p.myprop4[0];         // the first character of the myprop4 string
strlen(p.myprop4)     // the length of the myprop4 string.
```

4.7.4 Property Access Notifications in C++

Workers sometimes need dynamic notifications when properties are written or read by control software. E.g. the worker would like to run some code just before a property read returns to potentially create the correct value to read, or run some code after a property write to cause some other side effect of the new written value. For written values, this eliminates the need for polling by the worker to determine when the value has changed.

These capabilities are enabled when the property in the OWD has the **readSync** or **writeSync** attributes set to true. The **readSync** attribute being true indicates that the worker would like to run some code *before* the value of the property is passed back to the control software reading the property. The **writeSync** attribute being true indicates that the worker would like to run some code *after* a new value is written, in order to implement some side effect when the property value is changed.

An example of using **readSync** is when the value is computed based on some dynamic condition, e.g. reading a real-time or physical sensor value. An example of **writeSync** is when writing a new value should atomically affect some other state of the worker.

Using the C language, there is no notification mechanism at this time.

In C++, when these attributes are set, the skeleton includes empty implementations of notification methods for each such property that the worker author can fill out. For **readSync** properties, a worker member function called **<property>_read** must be present. For **writeSync** properties, a member function named **<property>_written** must be present. Both notification member functions take no arguments and return **RCCResult**, which allows these functions to indicate errors. A **readSync** notification function can set the property value locally and then return, and that value will then be conveyed back to control software.

A **writeSync** notification is called after the new value has been written, so the function can locally access the new value when it is called. The previous value is no longer available. If a worker needs to access the previous value it could declare a data member and assign it at the end of the notification method, and perhaps initialize this “previous value” data member using **firstRun** or **start**. E.g. assuming **predicate** is a boolean writable property with the **writeSync** attribute set to true:

```
class XyzWorker {
    bool m_prevPredicate;           // remember previous value
    RCCResult predicate_written() {
        // use both properties().predicate and m_prevPredicate
        m_prevPredicate = properties().predicate;
    }
    run() {
        // we know that as of firstRun, initial setting is valid
        if (firstRun()) m_prevPredicate = properties().predicate;
        ...
    }
}
```

4.7.5 Accessing the Values of Parameter Properties

Properties which are indicated as parameters by setting the **parameter** attribute to **true** in the OCS or OWD are accessed differently than other properties. They are defined as **static const** values whose name is a fully capitalized **<WORKERNAME>_<PROPERTYNAME>**. Thus for worker **wxyz** and parameter property **pqr**, of type **float**, the value is:

```
static const float WXYZ_PQR;
```

These values are initialized at compile time and can be used accordingly.

4.8 Controlling Slave Workers from Proxies — C++ Language Only

When the RCC OWD indicates that a worker is a **proxy** for another worker as one of its slaves, it gives the proxy worker convenient access to the slave's control operations and property accesses. This allows proxy workers to standardize and/or simplify the control of the slave worker. The slave worker is usually device-specific, with device-level properties or configuration requirements that applications should not be concerned with.

An example would be a device worker that controls an RF front-end device. This device typically exposes the native functionality of the device in a high performance and/or resource-conservative fashion. However, most applications would only want a subset of those options that are uniform across all RF front-ends they might use.

Delegating normalization of the control/configuration interface to the RCC proxy worker provides users with a higher level standardized configuration interface. This relieves this burden from the embedded device worker which may be in an environment (e.g. on a small FPGA) that makes this normalization difficult or expensive. This also enables the device worker to faithfully and simply implement the device's native control interface. This enables applications to exploit lower level device-specific features. Such a low level device worker might be controlled in different ways using different proxies.

A proxy worker, acting as the software module that encapsulates specialized configuration and control sequencing of the device, can eliminate the need for control applications to use special non-component APIs: the proxy is just another component in the application's XML.

In a proxy, the slave is accessed one of two ways. If the single slave was specified using the **slave attribute**, it is accessed using the **slave** data member of the worker's class object. If one or more slaves were specified using the **slave element**, each slave is accessed using the **slaves.<slave-name>** data member. This data member has functions for control operations on the slave (except **initialize** and **release**). For each readable property of the slave, a **get_<property-name>** member function retrieves the current value of the slave's property, and for each writable property of the slave worker a **set_<property-name>** is available. If the property is an array, both of these **get** and **set** member functions have arguments specifying the index in each dimension. The **set** functions then have a value argument for the new value. Here are some proxy code examples when a single slave is defined using the **slave attribute**:

```
uint8_t x = slave.get_byteregA();
slave.set_byteregA(3);
x = slave.get_arrayprop(2);
slave.set_arrayprop(3, x);
x = slave.get_array2d(1, 2);
slave.set_array2d(1, 2, x);
slave.stop();
slave.start();
```

Here are the some examples when one or more slaves is specified using the **slave element**:

```
uint8_t x = slaves.mydev.get_byteregA();  
slaves.otherslave.set_byteregA(3);  
x = slaves.specialslave.get_arrayprop(2);
```

There is currently no way to get or set whole array values, structures, structure members, or sequences. Only scalar values and array elements are supported. Proxy workers typically have no data ports and no code in their **run** method.

4.9 Worker Dispatch Structures — C Language Only

When workers are loaded for execution by the container, the container finds the worker by getting access to its **RCCDispatch** structure. The only external symbol the worker code needs to define is the symbol that holds this structure.

The code generator generates this initialization in the skeleton file, and it can be further customized by the worker author.

4.9.1 RCCDispatch Structure Type

This type is the dispatch table for the operations of the C-language worker interface. It represents the 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 statically initialized by the worker source code. This structure also contains other descriptive information required by the container to use the worker.

The RCC C-language skeleton will contain a default initialization for this structure based on what is in the OWD, which looks like (for worker **xyz.rcc**):

```
XYZ_METHOD_DECLARATIONS;  
RCCDispatch bias = {  
    /* insert any custom initializations here */  
    XYZ_DISPATCH  
};
```

Any non-default member initializations must be placed after the XYZ_DISPATCH line, and be specified using the named member syntax, e.g.:

```
RCCDispatch bias = {  
    /* insert any custom initializations here */  
    XYZ_DISPATCH  
    .memSize = 16*sizeof(MyData),  
    .runCondition = &myRunCondition  
};
```


The members of this structure that may be set this way are defined in the following table:

Table 8: Members of the RCCDispatch Structure — C Language Only

RCCDispatch Member	Member Data Type	Member Description
memSizes	uint32_t*	This zero-terminated array of memory sizes indicates allocations required by the worker. Multiple allocations allow the worker to avoid aggregating its requirements in a single allocation. May be NULL, when no allocations are required. Use memSize below when there is only one allocation required. Result will be in self->memories .
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.
optionalPorts	RCCPortMask	A mask indicating ports that may be unconnected. The default, 0, means all ports must be connected before the worker is started.
memSize	size_t	This size indicates a required static memory allocation. May be zero, indicating no allocation is required. Result will be in self->memory .

5 Code Generation for RCC Workers

In the descriptions below, italicized names in angle brackets are names specific to the worker, as found in the worker's OCS and OWD XML files.

The worker's OCS and OWD are used to generate two code files, the **header** and the **skeleton**. The header file, which should not be edited, and is deleted by “**make clean**”, contains type definitions and declarations customized for the worker. This file is named **<workername>_Worker.h** in the C language or **<workername>-worker.hh** in the C++ language, with the worker name is its original case from the OWD. It is placed in the **gen** subdirectory of the worker's directory.

The skeleton file is a small file that is generated as the basis for writing the worker's execution code (e.g. the **run** method implementation). It relies heavily on the generated header file and is as small as possible to avoid requiring code changes when the OCS or OWD is changed. The skeleton is generated in the **gen** subdirectory, with the name **<workername>-skel.c** (C language) or **<workername>-skel.cc** (C++ language). It is also copied into the worker's directory as the worker's primary source file: **<workername>.c** or **<workername>.cc**. This copy is what should be edited to add the worker's functional code. This copy is only made *if the worker's code file does not exist or has not been edited since it was previously copied from gen/*.

When the OCS, OPS or OWD XML files are changed, the header and skeleton are regenerated, *but this does not affect the primary source file*. Because most of the code is generated in the header, and only a little in the skeleton, it is rare that significant code changes are required in the primary source file based on XML changes.

The **gen/<workername>-skel.{c,cc}** skeleton file should not be edited, but left for reference to see exactly what the code generator produced.

5.1 Namespace Management

To avoid name space collisions, some rules are used by the code generator when creating the header and skeleton files.

In the C++ language, generated data types in the header are placed in a namespace whose name is: **<Worker>WorkerTypes**. In the actual worker source file, this namespace is normally imported, via the “**using namespace**” directive. The actual worker derived class is a class in the global namespace whose name is **<Worker>Worker**. In both cases, **<Worker>** is a capitalized version of the worker's name. The dispatch entry point, which is necessarily a C external symbol, has the worker's name as its external symbol, with the prefix **ocpi_**.

In the C language, the worker's dispatch table is given the external symbol name of the worker, and all methods are declared **static**. All data types use the capitalized worker name as a prefix and all constants and macros use the upper cased worker name followed by an underscore as a prefix.

A summary of the RCC authoring model name space management is:

- The C++ namespace **OCPI** is reserved.
- The prefix **ocpi_** is reserved in the C external linkage namespace.
- C workers use their name as an external symbol.
- C++ workers use two global namespaces that use the **<Worker>** prefix.
- C and C++ workers use the **PARAM_** and **OCPI_** prefixes for preprocessor symbols.
- C workers use the **<Worker>** prefix for type names and **<WORKER>_** for constants.

5.1 Generated Data Types

Various data types are generated in the header file for the worker. They are :

- Port name enumeration
- Property data structure
- Opcode enumeration for each protocol used
- Opcode enumeration for each port (C only)
- Message structures (C only)
- Worker base class (C++ only)

5.1.1 The Enumeration Constants for the Worker's Ports

An enumeration type is generated in the header file defining constants for the ordinal of each port, in the form **<WORKER>_<PORT>**, in all upper case. These ordinals can be used when creating port masks for run conditions, or, for C workers, indexing into the port array in the **RCCWorker** structure.

5.1.2 The Properties Structure Type

This is the generated structure definition that reflects the properties declared for the worker in its OCS and OWD XML files. Each member of this structure has the data type corresponding to the property's description in the OWD and OCS XML files. Members for properties that are *not* declared **volatile** in XML are **const** in this structure, indicating that the worker is not expected to change their values.

The standard name of the struct type (typedef name in C) is **XYZProperties** where "XYZ" is the *capitalized* worker implementation name.

Properties which are sequences are generated as a structure containing a **uint32_t** member whose name is **length**, holding the number of elements in the sequence. 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. The member name of the structure is the property name. The members of the structure are **length** and **data**. The data member is a C array whose length is the **SequenceLength** attribute from the **Property** element in the XML. Remember that for properties (as opposed to protocols), sequences and strings always have a maximum size (i.e. be bounded).

Struct properties have structure tags with the same name as the property name, preceded by the worker name.

The correspondence between property types in OCS and OWD files and the C/C++ language data types are in the following table:

Table 9: Property Data Types in C and C++

OCS/OWD Property Type	C/C++ Data Type	Comments
bool	RCCBool	Must be 8 bits, consistent across languages.
char	RCCChar	Must be signed type, and basis for string
uchar	uint8_t	
short	int16_t	
ushort	uint16_t	
long	int32_t	
ulong	uint32_t	
longlong	int64_t	
ulonglong	uint64_t	
string	RCCChar[]	Null terminated string
enum	uint32_t	
float	RCCFloat	Must be IEEE 32 bit float
double	RCCDouble	Must be IEEE 64 bit double

The names of the non-integer types for scalar properties are the OCS-defined type names capitalized and prefixed with **RCC**, specifically:

RCCBoolean, RCCChar, RCCFloat, RCCDouble

Properties that are string properties are **RCCChar** arrays whose size is one more than the declared **StringLength** attribute of that string property in the **Property** element, and the values are null terminated strings.

5.1.3 Structures for Message Payloads — C Language Only

For each port a union type is defined for all possible messages at that port. Members of the union type are generated structures for each possible message in that protocol. The name of the union type is **<Port>Operations**, with port name capitalized.

For each of the message types specified in the protocol, a structure is defined. The structure's type name is the capitalized name of the operation in the protocol, and whose member name is the lower cased name. Arguments to the operation in the protocol are structure members of that per-operation structure. The structure layout is as defined above for the property structure, with the exception that variable size elements (strings and sequences) are allowed, and are sized as **[1]**, with no further members generated after the first variable sized member.

The structure generated is padded and packed, such that padding members are explicitly inserted to ensure all types are aligned on their own boundaries. As part of the

packing, 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. The struct definition should not be used in any other context where this lack of overall size alignment is required (e.g. an array of such structures).

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 value of buffers at input and output ports, and *not* represented by inserting a `uint32_t` 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 represented by a struct containing **length** and **data** members.

For example, for a port named **in**, with a protocol whose first operation was **sampladata**, and whose first argument was **sample** - a sequence of **Ulonglong** values, the worker's custom port union would look like:

```
union InOperations {  
    // Structure for the 'sampladata' operation on port 'in'  
    struct __attribute__((__packed__)) Sampladata {  
        uint64_t sample[1];  
    } sampladata;  
};
```

5.1.4 Worker Base Class — C++ Language Only

For C++ workers, a base class is generated in the header file that is inherited by a derived class in the skeleton. The base class is generated to maximize the code generated in the header file and minimize the code generated in the skeleton.

The worker base class is generated to support all the documented features described above, including access to properties, and contains pure virtual member function declarations for all the member functions required to be implemented in the derived class generated in the skeleton. While the documented interfaces used by the worker author in the derived class are stable and not subject to change, the details of the code generation and the contents of header files (generated or not) are implementation defined and subject to change.

6 RCC Local Services

Local services APIs are standard library functions available to RCC workers running in a container. Containers are required to supply them, and portable workers are constrained to use only them, in addition to the worker-to-container methods defined above. The RCC local services are defined as a small subset of the POSIX and ISO-C runtime libraries.

The subset avoids functions that require significant operating system support while providing the author with common, convenient and standard library functions. All I/O is excluded since portable RCC workers should be performing all I/O via the OpenCPI data plane ports (except **printf**).

These local services APIs represent a minimal environment required of embedded systems. Since RCC workers may execute in a multi-threaded environment, non-reentrant (or thread-unsafe) functions are excluded (e.g. **localtime**). Similarly, functions that affect the global (process) environment are excluded since they could damage the execution of other workers in the same process (e.g. **setenv**).

If other APIs are used, there is no guarantee they will be available in all containers. Some containers in fact allow many more functions to be called from a worker, but that makes such workers non-portable. In the future there may be optional explicit error checking for using library functions not in the list below.

In other standards documents, a list of available, standard, functions is sometimes called an Application Environment Profile (AEP).

The actual list of standard functions and global variables usable in portable RCC workers is, in alphabetical order:

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

The C++ standard library, especially in later versions, has not been analyzed for appropriate restrictions to ensure one worker does not damage the environment for other workers in the same process/container.

Using heap allocation functions (e.g. **malloc**, **calloc**, **realloc**) requires that the worker release any allocated memory in its **release** method when the allocations persist across methods.

6.1 RCC Local Services AEP as a Small Subset of POSIX and ISO-C.

This section explains the rationale for the RCC AEP and its connection to POSIX standards. The POSIX Minimal Realtime System Profile (PSE51) from the **IEEE Std 1003.13™-2003** standard is used as the superset of functionality that is reduced (with subtractions) to the RCC local services available functions. It can be found at <https://standards.ieee.org/findstds/standard/1003.13-2003.html>.

This subset is defined to require setting the **_POSIX_AEP_RT_MINIMAL_C_SOURCE** feature test macro to the value **200312L**, before including any standard header files. This is not necessary in RCC workers. The PSE51 profile defines POSIX.1 units of functionality in table 6-1 of the IEEE Std 1003.13™-2003 document. The RCC Local Services AEP removes these units of functionality from that table:

- **POSIX_DEVICE_IO**
- **POSIX_FILE_LOCKING**
- **POSIX_SIGNALS**
- **XSI_THREAD_MUTEX_EXT**
- **XSI_THREADS_EXT**
- **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_r(), atof(), atoi(), atol(),  
atoll(), bsearch(), calloc(), ctime_r(), difftime(), div(),  
feclearexcept(), fegetenv(), fegetexceptflag(), fegetround(),  
fholdexcept(), feraiseexcept(), fesetenv(), fesetexceptflag(),  
fesetround(), fetestexcept(), feupdateenv(), free(), gmtime_r(),  
imaxabs(), imaxdiv(), isalnum(), isalpha(), isblank(), iscntrl(),  
isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(),  
isupper(), isxdigit(), labs(), ldiv(), llabs(), lldiv(),  
localeconv(), 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(), strptime(), strlen(), strncat(), strncmp(), strncpy(),  
strpbrk(), strrchr(), strspn(), strstr(), strtod(), strtod(),  
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(),  
sysconf(), uname();
```

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, even those environments without multithreading.

7 Debugging RCC Workers

RCC workers are built as dynamically loadable “shared object” files, with the **.so** suffix (at least on Linux systems). When an application uses a worker, it will be loaded on demand, even when the executable is statically linked itself. To debug a worker, it is necessary to first start the debugger on the executable, which is usually the **ocpirun** utility program that runs OpenCPI applications based on a top-level XML file specifying which components are needed. It may also be a custom main program that uses the OpenCPI Application Control Interface to internally launch an application.

In either case the first step is to run the executable under the debugger, establishing a generic breakpoint to enter the debugger at a point after workers are loaded, but before they are actually run. Then breakpoints can be placed in the worker code itself.

The two pieces of “internal” information needed to do this are:

- A breakpoint that happens just before any RCC worker is initialized
- A way of knowing when the worker that hits the breakpoint is the worker you want.

The initial breakpoint should be placed on the **OCPI::RCC::Worker::Worker** member function (an internal constructor). This breakpoint will be hit for every worker in the application, after it is loaded, but before it ever is initialized (C) or constructed (C++). Note that although this initial breakpoint is at a constructor, it is not the actual constructor of the C++ worker, and not even in its inheritance hierarchy.

To determine whether the worker about to be constructed is the worker of interest, simply examine the “name” argument at this breakpoint. This is the instance name for the worker within the application. If the name indicates a worker of interest you can now establish a breakpoint in the worker, either based on a source line number, or symbols in the worker.

Assuming the use of the **gdb** debugger, and running the application with the **ocpirun** utility, here are the steps:

5. Start gdb on ocpirun:
% gdb ocpirun
6. Set the generic breakpoint:
(gdb) b OCPI::RCC::Worker::Worker
At this point gdb may not find it and say:
Make breakpoint pending on future ... load? (y or [n])
Say yes (y).
7. Run ocpirun with appropriate arguments:
(gdb) run -v myapp
8. When the generic breakpoint is hit, see which worker is being initialized by printing the **name** argument. If not a worker of interest, continue.
(gdb) p name
(gdb) c

9. When a worker of interest is about to be initialized/constructed, set a breakpoint in your worker, e.g.:
(gdb) b myfft.c:run (e.g. for C, set breakpoint on run method)
(gdb) b myfft.cc:135 (e.g. for a C++ worker, by line number)
(gdb) b MyfftWorker::MyfftWorker (e.g. break on C++ constructor)
(gdb) c
10. Continue past any other worker initializations (or disable the first breakpoint).

At this point the breakpoint in the worker will be hit and debugging can proceed.

8 Summary of OpenCPI RCC Authoring Model

- RCC workers are written to implement worker methods, called by the container, and optionally use the container methods, called by the worker.
- RCC workers may call the local services functions defined in section 6 above.
- RCC workers use symbols, data types and ordinals defined in the generated header file, which is based on the OCS, OPS, and OWD files, including
 - *Implementation name*
 - *Which control operations are implemented*
 - *Which properties require notification*
 - *Static memory allocation requirements of the implementation code*
 - *Minimum number of buffers required at each port*
- The information in the OCS/OPS/OWD files is used to drive the code generation and build process.
- The worker uses the software execution model as described in the CDG.

9 Building RCC Workers

The CDG explains how all workers are built, and this also applies to RCC workers. There are some **Makefile** variables that apply only to RCC workers. They can be set:

- in the worker's **Makefile**
- on the command line when invoking **make**
- in a project's **Project.mk** file

If they are specific to the worker, they should only be in the worker's **Makefile**.

9.1 RCC Compiler and Linking Options

For most RCC workers, the built-in compiler options are suitable and no extra variables are needed. In special cases, it may be useful to control these options for a given worker, library or project.

Compiler and link option variables exist in several variants, with suffixes to indicate how narrowly to apply them. For each such option variable mentioned in the table below, there are actually 6 potential variables with different suffixes. In the suffixes:

<language> is **C** or **CC** for C or C++, **<os>** is the operating system of target (e.g. **linux** or **macos**), and **<platform>** is the specific RCC target platform. A typical example of the fully specific target is **linux-c7-x86_64**, indicating the CentOS7 distribution of linux, compiled for the **x86_64** architecture. E.g., for the **RccCompileWarnings** variable (in the table below) the variations are:

```
RccCompileWarnings
RccCompileWarnings<language>
RccCompileWarnings_<os>
RccCompileWarnings<language>_<os>
RccCompileWarnings_<platform>
RccCompileWarnings<language>_<platform>
```

The most specific one overrides all the less specific ones above it. For example, specifying:

```
RccCompileWarnings_linux=-Wabi
```

would indicate that you want **-Wabi** to be the only warning option when building RCC workers for the linux OS. If you also set:

```
RccCompileWarnings=-Wall
```

then for all OSs other than linux, the warnings would be **-Wall**. If you wanted the default for all targets to be **-Wall**, but to *add* **-Wabi** for linux, you would specify:

```
RccCompileWarnings=-Wall
RccCompileWarnings_linux=$(RccCompilerWarnings) -Wabi
```

For each variable, like the **RccCompileWarnings** example above, there are actually 2 variables that each use the suffixes scheme:

- **RccCompileWarnings** – which OpenCPI controls and sets as appropriate

- **RccExtraCompileWarnings** – which OpenCPI never sets and whose contents is placed on the compile or link command line *after* the one without “**Extra**”.

Thus it is strongly recommended to use the **Extra** variants when possible. Specifying incorrect options for compilation and linking can easily result in errors. There is no guarantee that any additional options will be compatible with OpenCPI or result in functional worker binary artifacts. When changing these options, it is recommended to first run with default settings and use the **showvars** make target on the **make** command line to see how the options are used, before and after modifying these variables.

Table 10: RCC make variables for inside RCC worker Makefiles

Makefile Variable	Description
RccIncludeDirs IncludeDirs	Directories to search for C/C++ header files outside OpenCPI. Either IncludeDirs or RccIncludeDirs can be used in the worker Makefile, but RccIncludeDirs is specific to RCC workers when specified at the library or project level. Can be relative or absolute pathnames. No Extra or suffixes are used.
RccCompileWarnings	For controlling compile warnings. Uses option suffixes and Extra .
RccCompileOptions	For controlling compile options (not warnings). Uses option suffixes and Extra .
RccLinkOptions	Options applied when creating the RCC worker shared object file. Uses option suffixes and Extra .
RccCustomLibs	Library options to specify libraries unknown to OpenCPI. Uses option suffixes but not Extra . OpenCPI never sets this.
RccStaticPrereqLibs	Names of libraries installed as OpenCPI prerequisite libraries that this worker should link against, statically, which will incorporate (copy) any referenced objects in that library into the worker's binary artifact. This automatically puts the library's include directory in the search path for headers.
RccDynamicPrereqLibs	Names of libraries installed as OpenCPI prerequisite libraries to link against, dynamically, which requires that the library be available at runtime. This automatically puts the library's include directory in the search path for headers.
RccPlatform(s) OnlyPlatforms ExcludePlatforms	Which platform(s) should the RCC worker be built for? Only one of RccPlatform or RccPlatforms should be specified. If none are specified, the development host running make is assumed to be the target. In a worker Makefile , OnlyPlatforms and ExcludePlatforms can be used to restrict which platforms the worker is built for, even when RccPlatforms is set on the command line for many platforms.
RccHdlPlatform(s)	This option indirectly specifies the RccPlatform(s) in terms of its association with an HDL platform. E.g. setting RccHdlPlatform=zed would specify the RccPlatform currently associated with the zed HDL platform, which may or may not be the same as used on other HDL Xilinx Zynq-based platforms.

10 Worker Code Examples

10.1 C Language Examples

Here is a simple example of a C-language “Xyz” worker whose:

- Initial run condition was the default (condition == NULL, usecs == 0, run when all ports are ready, no timeout),
- **start** and **release** methods are not needed or implemented.
- One input port (0) with interface XyzIn, and one output port (1) XyzOut
- One interface operation Op1 on input (i.e. can ignore “operation”), which is an array of 100 “shorts”.
- One 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 is below.

10.1.1 Worker Using the Default Run Condition

This C language worker uses the default run condition and thus has no checking for port readiness, but does set a default operation for output messages. It always produces an output message for every input message, so the run method can simply return **RCC_ADVANCE**.

```

#include "Xyz_Worker.h"

/* Define the initialize method, setting output operation to be a */
/* constant, since it is the only one.*/
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.*/
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 an ARM processor.

10.1.2 Worker Using the State-machine Style

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

```

#include "Xyz_Worker.h"
/* Define two different run conditions to represent two states */
static uint32_t
    state1Ports[] = {1 << XYZ_IN, RCC_NO_PORTS},
    state2Ports[] = {1 << XYZ_COMPUTE_REPLY_IN, RCC_NO_PORTS};
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;
}
/* 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[XYZ_IN],
            *computeOut   = &this->ports[XYZ_COMPUTE_REQUEST_OUT];
        XyzInOp1 *in      = inPort->current.data;
        XyzProperties *p = this->properties;

        /* do some computation based in is shorts, 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, XYZ_COMPUTE_PROCESS,
            inPort->input.length); /* length of message */
        this->runCondition = &awaitingResponse; /* update state */
    } else {
        RCCPort *computeIn = &this->ports[XYZ_COMPUTE_REPLY_IN];
        RCCPort *otherOut = &this->ports[XYZ_OUT];

        if (computeIn->input.u.exception == 0)
            c->send(otherOut, &computeIn->current, XYZ_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 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,
    /* all remaining members zero/NULL */
};

```

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

10.2 C++ Language Examples

Here is a simple example of a C++ language “xyz” worker whose:

- Initial run condition was the default (condition == NULL, usecs == 0, run when all ports are ready, no timeout),
- **initialize**, **start** and **release** methods are not needed or implemented.
- One input port (**in**), and one output port (**out**)
- One interface operation Op1 on input, which is an array of 100 “shorts”.
- One interface operation Op2 on output, a sequence of shorts.
- One simple property, called center_frequency, of type float.

The OPS for the **in** port would be (in the **shortarray-prot.xml** file):

```
<protocol>
  <operation name='op1'>
    <argument name='shorts' type='short' arraylength='100' />
  </operation>
</protocol>
```

The OPS for the output port would be (in the **shortseq-prot.xml** file)

```
<protocol>
  <operation name='op2'>
    <argument name='shorts' type='short' sequencelength='0' />
  </operation>
</protocol>
```

The OCS for the component would be (in the **xyz-spec.xml** file):

```
<componentSpec>
  <property name='center_frequency' type='float' writable='true' />
  <port name='in' protocol='shortarray' />
  <port name='out' protocol='shortseq' />
</componentSpec>
```

The OWD would be (in the file **xyz.xml**):

```
<RccWorker spec="xyz-spec" language='c++' />
```

The worker would have automatically generated types and structures like this (based on OCS and OWD), and put in a file called **xyz-worker.hh**:

```
#include <RCC_Worker.h>
namespace XyzWorkerTypes {
  struct Properties {
    // internal stuff here
    const RCCFloat center_frequency; // const since not volatile
  };
  class XyzWorkerBase {
    // internal stuff here, including port members
  };
}
```

The actual code for the C++worker would look like this (in the file **xyz.cc**):

```
#include "xyz-worker.hh" // include generated declarations

using namespace OCPI::RCC; // for access to RCC data types and constants
using namespace XyzWorkerTypes; // for generated types

struct XyzWorker : public XyzWorkerBase { // use struct to avoid "public"
    // In constructor, set output operation to be a constant,
    // since it is the only one used, but not the default (0)
    XyzWorker() {
        out.setDefaultOpCode(ProtOp2_OPERATION);
    }
    void
    compute(int16_t *in, int16_t *out, size_t n, float freq) { ... }
    // Define run method to call the "compute" function, reading from
    // input buffer, writing to output, applying current value of the
    // centerFrequency" property
    RCCResult
    run(RCCBoolean /* timedout */) {
        // Do computation from in.op0.shorts(), using frequency, put results
        // in out.op2.shorts().
        out.op2().shorts().resize(in.op0().shorts().size());
        compute(in.op0().shorts().data(), out.op2().shorts().data(),
                in.op0().shorts().size(), properties().center_frequency);
        // Ask container to send output and get new input/output buffers */
        return RCC_ADVANCE;
    }
};
```

10.2.1 Worker Using the State-machine Style

This illustrates a finite state machine coding style in a C++ worker that maintains an internal state to simulate blocking on a two-way remote call until the response is received and processing can continue. It accepts data on its **in** port, performs computation in-place (in the input buffer), then sends this buffer to an **out** port, and starts awaiting a response from a **response** input port. When the response arrives, it sends the response to an **other** output port, and reverts to its original condition, waiting for input.

```
#include "xyz-worker.hh" // include generated declarations

using namespace OCPI::RCC; // for access to RCC data types and constants
using namespace XYZWorkerTypes; // for this worker's generated types

struct XYZWorker : public XYZWorkerBase { // use struct to avoid "public"
    // Define two different run conditions to represent two states
    // These could be static and initialized outside the class too.
    RCCRunCondition m_awaitingInput, m_awaitingResponse;
    XYZWorker()
    : m_awaitingInput(1 << XYZ_IN, RCC_NO_PORTS),
      m_awaitingResponse(1 << XYZ_RESPONSE, RCC_NO_PORTS) {
        setRunCondition(&m_awaitingInput);
    }
    RCCResult
    run(RCCBoolean /* timeout */) {
        // Use run condition as state indicator
        if (getRunCondition() == &m_awaitingInput) {
            // do some computation based in in.op0().shorts(), and frequency;
            // put results back in same buffer (in-place)
            compute(in.op0().shorts().data(), in.op0().shorts().data(),
                    in.op0().shorts().size(), properties().center_frequency);
            // Input port is advanced by taking buffer away from it.
            out.send(in); // send input buffer to output, same op and length,
            setRunCondition(&m_awaitingResponse); // await response
        } else {
            other.send(response); // got response, send to other
            setRunCondition(&m_awaitingInput); // change state to await input
        }
        return RCC_OK;
    }
};
```


11 Glossary

Configuration Properties – Named values associated with a worker that may be read or written by the control application and/or the worker. 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. Some properties come from the OCS; they are common to all implementations that implement that OCS. Other properties can be added in the OWD that are specific to that particular worker implementation.

Container – An OpenCPI infrastructure element that “contains”, manages and executes a set of application workers. Logically, the container “surrounds” the workers, mediating all interactions between the worker and the rest of the system.

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

Worker – A concrete implementation of a component. Also, a runtime instance of a worker source code implementation, generally existing within a container. A worker is implemented consistent with its authoring model and its OWD.

12 List of Abbreviations and Acronyms

ACI	Application Control Interface
API	Application Programming Interface
CDG	OpenCPI Component Development Guide
CDK	Component Developer's Kit
OCS	OpenCPI Component Specification
OpenCPI	Open-Source Component Portability Infrastructure
OWD	OpenCPI Worker Description
RCC	Resource Constrained C/C++ Language
XML	Extensible Markup Language