# OpenCPI
# RCC Development Guide

*OpenCPI Release: v2.3.4*

## *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 |
| 1.5 | Update for release 1.5 | 2019-04-25 |
| 1.6 | Update for release 1.6 | 2020-01-02 |
| 1.7 | Update for 1.7,minbuffer count, and send method clarification | 2020-06-16 |
| 1.8 | Add shared glossary of terms chapter | 2021-03-15 |
| 1.9 | Add slave assembly info, put C++ examples first | 2021-04-09 |
| 2.0 | Remove most Makefile details for release 2.2 sine variables are now in XML files. | 2021-06-29 |

## Table of Contents

# 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 that 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 User Guide | OpenCPI | https://opencpi.gitlab.io/releases/v2.3.4/docs/OpenCPI_User_Guide.pdf |
| OpenCPI Component Development Guide | OpenCPI | https://opencpi.gitlab.io/releases/v2.3.4/docs/OpenCPI_Component_Development_Guide.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. 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 collocated 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:*** they 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)
- Slave information: which ones and how they are configured and connected.

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' writable='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**, **controlOperations**, **onlyPlatforms**, **excludePlatforms**, **sourceFiles**, **includeDirs** and **componentLibraries** 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++ workers. 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.

This *attribute* cannot be used when a worker is a proxy for more than one slave. In that case the <**slaves**> *element* must be used. See Slaves Element. *This attribute also cannot be used if the required slave worker is not the default build configuration (or only build configuration) of the slave worker*. The **<slaves>** element can be used in that case, since it provides a way to specify parameter settings for slave workers.

### *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. This is not typical usage.

### *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`**.

### *3.1.8 StaticPrereqLibs Attribute*

This attribute provides a list of names of prerequisite libraries needed by this worker. Prerequisite libraries are only those built in to OpenCPI (at this time). Mentioning these libraries automatically adds the header file directory associated with the library to be searched when the worker source files are compiled.

The indicated libraries are statically linked into the binary artifact for this worker and are thus not shared by any other worker. This static mode makes using the worker simpler since there is no requirement for the library to be present, as a dynamically loaded library, in the runtime environment.

### *3.1.9 DynamicPrereqLibs Attribute*

This attribute provides a list of prerequisite libraries needed by this worker. Prerequisite libraries are only those built in to OpenCPI (at this time). Mentioning these libraries automatically adds the header file directory associated with the library to be searched when the worker source files are compiled.

The indicated libraries are dynamically linked with the binary artifact for this worker and are thus sharable by any other worker. This dynamic mode makes using the worker's use of the library require the library to be present, as a dynamically loaded library, in the runtime environment.

## 3.2 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 example 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. The worker must wait (via its run condition) for each buffer to arrive.

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 Slaves Element in the OWD

While the **slave** *attribute* can be conveniently used to indicate that this worker is a proxy for a single slave (worker) with its default/only build configuration, the **<slaves>** *element* is used when there are multiple slaves or specific build configurations (parameter property settings) are required. The **slaves** element introduces a set of slave instances, along with their static or initial properties and interconnections. The **slaves** element is essentially an "assembly of slave instances", very similar an application XML (OAS) as an assembly of component instances, or an HDL assembly as an assembly of HDL worker instances.

Here is an example (in a proxy's OWD) which specifies a single slave worker, which is equivalent to using the top-level **slave** attribute for a single worker:

```
<slaves>
  <instance worker='ocpi.devices.xyz_adc.hdl'/>
</slaves>
```

#### 3.3.1 Name Attribute of the Instance Element in a Slave Assembly

Each instance can have an optional **name** attribute whose purpose is to uniquely name the instance within the assembly. If the **name** attribute is not specified, the name of the instance is the worker's name (without package prefix or model suffix). If there is more than one instance of a worker, the default names have a zero-based ordinal as a suffix. I.e. if the slave assembly was:

```
<slaves>
  <instance worker='ocpi.devices.xyz_adc.hdl'/>
  <instance worker='ocpi.devices.xyz_adc.hdl'/>
</slaves>
```

The first slave would have the name **xyz_adc0** and the second would have **xyz_adc1**.

The slave instance's name is the name used in the proxy worker's source code to reference this particular slave. This instance naming is identical to naming in HDL assemblies and application assemblies.

#### 3.3.2 Worker Attribute of the Instance Element in a Slave Assembly

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

#### 3.3.3 Connections in Slave Assemblies

Slave instances may be connected using the techniques described for all types of assemblies (applications and HDL assemblies): using either the **connect** attribute of the instance (for simple connections), or the **connection** element. These are described in the OpenCPI Application Development Guide section on the OAS XML documents.

Slave assemblies may have external ports, the same as application assemblies or HDL assemblies.

Connections to external ports of a slave assembly are used during the execution of an application to *delegate* connections at a proxy worker's ports in the *application* assembly, to slave instance ports in the proxy's *slave* assembly. This means that if the proxy is selected to support an instance in the application, any connections to its ports are in fact connected to slave instance ports in the slave assembly instead, and not to the proxy worker itself.

For example, if the application is this (**`file_read => mycomp => file_write`**):

```
<application>
  <instance component='file_read' connect='mycomp'/>
  <instance component='mycomp' connect='file_write'/>
  <instance component='file_write'/>
</application>
```

and you have a proxy worker that implements the **`mycomp`** spec (with **`pxin`** and **`pxout`** ports), with this slave assembly:

```
<slaves>
  <instance worker='myfirst.hdl' connect='mysecond'/>
  <instance worker='mysecond.hdl'/>
  <external name='pxin' instance='myfirst' port='in'/>
  <external name='pxout' instance='mysecond' port='out'/>
</slaves>
```

This means that, at application execution/deployment time, when the proxy is selected for **`mycomp,`** the connection of **`file_read`** to the **`pxin`** port of **`mycomp`**, is actually connected to the **`in`** port of **`myfirst`**, and the connection to **`file_write`** from the **`pxout`** port of **`mycomp`** is actually connected from the **`out`** port of **`mysecond`** instead. So the way the proxy has delegated the connections at its ports, to the external ports of the slave assembly with the same names as the proxy's ports. And in this case the app is now:

```
file_read => myfirst => mysecond => file_write
```

### *3.3.4 Array Port Connections in Slave Assemblies*

Proxy ports that are delegated as described above may be ***array ports***, meaning they can have a **`count`** attribute in the **`<port>`** element which indicates the width or dimension of the port. This indicates that each such array port is actually a set of ports numbered from 0 to **`count`** -1. When array ports are referenced in **`<connection>`** elements, the **`index`** attribute is used to indicate which individual port(s) in the array are being connected. The **`mycomp`** spec (OCS) implemented by the proxy worker may declare ports as having count attributes, e.g.:

```
<ComponentSpec>
  <port name='pxin' protocol='myprot' count='2'/>
  <port name='pxout' producer='true' protocol='myprot'/>
</ComponentSpec>
```

This component spec could be for a proxy with a **pxin** port of count 2, and a **pxout** port that is not an array port. An RCC proxy worker might have an OWD like this:

```
<RccWorker language='c++' spec='mycomp'/>
  <slaves>
    <external name='pxin' count='2'/> -- external matches OCS port
    <external name='pxout'/>          -- external matches OCS port
    <instance worker='first.hdl' connect='adder' port='arg1'/>
    <instance worker='second.hdl' connect='adder' port='arg2'/>
    <instance worker='adder.hdl'/>
    <connection>
      <external name='pxin' index='0'/>
      <port name='in' instance='first'/>
    </connection>
    <connection>
      <external name='pxin' index='1'/>
      <port='in' instance='second'/>
    </connection>
  </slaves>
</RccWorker>
```

This slave assembly declares external ports consistent with the proxy's spec, and has two workers (**first.hdl** and **second.hdl**) that each connect to one of the **pxin** ports (index 0 and index 1). Both **first** and **second** have their outputs connected to an adder worker instance (at input ports **arg1** and **arg2**), whose output is connected to the **pxout** extern port. [diagram]

These array ports are *not* supported in all assemblies or for all instances, but *are* supported for proxy ports and the external ports of slave assemblies.

# 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 **`self`** argument 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 reserved for use 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 (except C++ where it is optional) 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 (supplied as the **`self`** argument). 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, described below. For C++, they are simply 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 from the **`run`** method 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 (and only for C, not C++): ***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 OpenCPI 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 no particular content or member ordering, but with the documented members present. 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 C 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 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_FINISHED** | 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 *advanced* (applies only to the run method). |
| **RCC_ADVANCE_FINISHED** | The worker is requesting that all ports be advanced (run method only) and declaring that it is also "finished". The worker is entering the ***finished*** state like **RCC_FINISHED**. |

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, do not generate error conditions, and cannot use member data initializers.
- *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 (or member initializers) 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 or time 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 optional method implements the ***start*** control operation as defined in the CDG. Upon successful completion, the worker is in the ***operating*** state or possibly in 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. If this method is not provided the default implementation advances the worker to the ***operating*** state.

#### 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_FINISHED`** 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_FINISHED`** 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 optional 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. If not provided, the default implementation simply advances the worker to the ***suspended*** 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 (optional in C++) requests that the worker perform its normal computation. The container only calls this method when the worker's *run condition* is satisfied. The default C++ implementation of this method disables the worker's ***run condition*** causing this method to never be called again. This default behavior allows workers that have no need for run methods to avoid using a non-default run condition for this purpose. They simply do not implement the run method.

#### 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. In C++, the run method may be *omitted entirely*, which is equivalent to a permanent run condition which is never satisfied. This is useful when the only function of the worker is to implement other control operations or property notification methods.

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 the pointer value 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_FINISHED`** or **`RCC_ADVANCE_FINISHED`** if no further worker execution should happen.

See RCCResult Enumeration Type section 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:

- worker 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*** 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 **sprintf**.

#### 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 message 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**. 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 ACI. 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
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.

#### 4.3.4.1 Synopsis

```
bool (*willLog)(unsigned level);  // C Language
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 used with 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`** mask 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 *true* when *any* of the masks is true. **`RunCondition(RCC_NO_PORTS)`** indicates: *never run*. |
| **`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. The first argument *must* be pointer type, and *not* **`RCC_NO_PORTS`** itself or **`NULL`**. A value of **`nullptr`** *may* be used to indicate that the run condition is always true. |

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); // restore default run condition
  }
}
```

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 condition. 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. This argument itself must not be **`NULL`** or **`RCC_NO_PORTS`** since they are not pointers. The **`nullptr`** value may be used to indicate an aways-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 *true* when *any* of the masks is true. A mask is *true* when all indicated ports are ready (logical AND of port readiness). *If the pointer itself is **NULL** or **nullptr**, 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 *usecs*. |
| **usecs** | **uint32_t** | When enabled by the **timeout** member, if this amount of time has passed (in microseconds) since the run method was last *entered*, the run condition is true. |

The 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 **nullptr**, 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** | **portMasks == NULL** | ignored | The worker is always ready, no timeout |
| **Run when data ports are ready** | **portMasks != NULL** | False | Run condition is TRUE when any mask is true. If there are no masks, i.e. **portMasks[0] == RCC_NO_PORTS**, then the run condition is FALSE. |
| **Run when data ports are ready or timeout.** | **portMasks != NULL and portMasks[0] != RCC_NO_PORTS** | 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** | **portMasks != NULL and portMasks[0] == RCC_NO_PORTS** | True | Since **portMasks[0] == RCC_NO_PORTS**, 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 object's 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. In this case checking for availability of data is unnecessary and redundant. If the port's readiness is not implied by the run condition, 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.

In C++, there is alternative set of methods that format the message correctly for the worker, and these methods should be preferred in most 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. If so **`hasBuffer`** will now return true.

### *4.4.2 **hasBuffer** — Query a Port for Whether It Has a Current Buffer*

This method returns whether a port has a current buffer. If a port is already part of a workers'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++, an **RCCOrdinal** non-negative integer value is returned.

### 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, it 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.

Note: when a buffer is sent to an output port using **send**, it is bypassing (and not affecting) any current buffer on that port. Thus after **send**, the port should typically *not* be advanced (e.g. by returning RCC_ADVANCE), since that would essentially *also* send the current buffer in addition to the buffer that was explicitly sent. Furthermore, when **send** is used, the output port is normally not included in the run condition, since the availability of the output port's current buffer is irrelevant. When using **send** on an output port the default run condition for the worker is inappropriate.

##### *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 optionally 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.

***EOF*** indications may be received at input ports and sent at output ports. The role and purpose of EOFs is described for all types of workers in the End-of-File (EOF) Indications on Data Ports section in the CDG. However, most workers never see or generate EOFs because the default behavior for EOFs is that the container handles them without ever showing them to the worker. Unless a worker explicitly asks to handle EOFs by setting the **`workerEOF`** attribute at some of its ports to **`true`**, the container propagates the EOF indication from input to output without exposing EOFs to the worker at all.

The default propagation behavior is: when an EOF arrives at the first input port, it is propagated to all output ports automatically and the worker is not run any more. Workers can always send EOFs on output ports regardless, if that is useful. To suppress the automatic propagation of EOFs “behind the worker's back”, set the **`workerEOF`** attribute to **`true`** on the first input port, and EOFs will be delivered to that port and propagation is suppressed.

#### *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 ***eof*** *— Retrieve the EOF status of the Buffer*

This method retrieves the ***EOF status*** of the buffer (and port), for an input buffer. This is C++ only. In C, access to the EOF status is via the current buffer at a port, e.g.:

```
port->input.eof
```

#### *4.5.6.1 Synopsis*

```
RCCBool op = port->input.eof;     // C language
bool RCCUserBuffer::eof() const;  // C++ Language
```

#### *4.5.6.2 Returns*

This method returns the EOF status as a boolean value.

### 4.5.7 ***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.

#### *4.5.7.1 Synopsis*

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

#### 4.5.7.2 Returns

Nothing.

### 4.5.8 *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.

#### 4.5.8.1 Synopsis

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

#### 4.5.8.2 Returns

Nothing.

### 4.5.9 *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.

#### 4.5.9.1 Synopsis

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

#### 4.5.9.2 Returns

Nothing.

### 4.5.10 *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.

#### 4.5.10.1 Synopsis

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

#### 4.5.10.2 Returns

Nothing.

### 4.5.11 *setEOF* — Set the EOF status of the Buffer

This method sets the ***EOF status*** of the buffer (and port), for an output buffer. This is C++ only. In C, setting the EOF status is via the current buffer at a port, e.g.: **`port->output.eof`**

#### 4.5.11.1 Synopsis

```
port->output.eof = true;            // C language
void RCCUserBuffer::setEOF() const;  // C++ Language
```

#### 4.5.11.2 Returns

This method sets the EOF status as a boolean value.

### 4.5.12 ***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.

This method is only available in C++.

#### 4.5.12.1 Synopsis

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

#### 4.5.12.2 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
- none of **`initial`**, **`writable`** or **`parameter`** — meaning the worker can set it during initialization only and *not after start*.

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 ***volatile*** properties or those 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 ***non-volatile*** 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's value is returned to control software, (when control software reads the property), to create the correct dynamic value to return. Similarly, a worker would run some code *after* control software writes a property, to cause some other side affect of the new written value. For written values, this eliminates the need for checking 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 worker skeleton code 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 all upper case ***<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. Internally, the parameter values are also defined as preprocessor macros of the form:

```
OCPI_PARAM_<worker-name>_<property-name>()
```

If the parameter's value must be used in a preprocessor conditional e.g. **#if *<expr>***, these macros can be used instead of the **static const** variables.

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

When the RCC OWD indicates that a worker is a ***proxy*** for other workers, as its slaves, it gives the proxy worker convenient access to each slave's control operations and property accesses. Proxy workers typically have no data ports and no code in their **`run`** method (in C++, no run method at all). 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 a clock generator device. This device worker 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 clock generators they might use.

Implementing normalization of the control/configuration interface in 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 implements 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, and its generic properties can be translated to actual device configuration settings.

For each slave worker mentioned in a **`slave`** attribute or in the **`<slaves>`** element, there is a member data object in the (proxy) worker's class that is used to control the slave. Each slave is controlled using the **`slaves.<slave-name>`** data member, which is a class specifically generated per slave. The first or only slave can also be accessed using a data member whose name is simply **`slave`**.

These slave objects have methods to perform control actions or property accesses on the slave workers. The control methods have fixed method names, while the property access methods have names based on the property names of the slave's properties. The property access methods are the roughly the same as the property access methods at the application level in the ACI except that properties are specified by the method name rather than a string name argument. As in the ACI, there are methods for accessing properties using text values in strings or using the underlying binary values.

These are all methods on the **`slave`** or **`slaves.<`*`slave`*`>`** member objects.

#### 4.8.1 **start** — Slave Method

This method performs the ***start*** control operation on the slave worker, which may start it after initialization or resume it after suspension (after it has been stopped). If the proxy does not call this method in its own **`start`** method, then the slave will be started after all proxies have been started in any case. Thus use of this method is optional.

##### 4.8.1.1 Synopsis

```
void start();            // C++ Language
```

#### 4.8.2 **stop** — Slave Method

This method performs the ***stop*** control operation on the slave worker. If the proxy does not call this method in its own **`stop`** method, then the slave will be stopped after all proxies have been stopped in any case. Thus use of this method is optional. A proxy may stop (suspend) or start (resume) its slaves during its own execution, independent of its own **`start`** or **`stop`** methods.

##### 4.8.2.1 Synopsis

```
void stop();             // C++ Language
```

#### 4.8.3 **isOperating, isFinished, isUnusable, isInitialized, isSuspended** — Slave Methods

These methods query the slave and returns whether it is currently in the indicated control state. Slaves are not started before their proxies, and if a proxy does not start a slave in its own start method, then the slave will be started after all proxies are started and enter the operating state. See the Control Plane Introduction section of the CDG.

##### 4.8.3.1 Synopsis

```
bool isOperating();           // C++ Language
bool isFinished();            // C++ Language
bool isUnusable();            // C++ Language
bool isInitialized();         // C++ Language
bool isSuspended();           // C++ Language
```

##### 4.8.3.2 Returns

These methods return a boolean which is true when the slave is in the indicated control state.

#### 4.8.4 **getProperty_<property>** — Slave Method

This method retrieves the value of a property in text format in a string passed as an output argument by reference. It has three optional arguments for controlling which part of the property to retrieve (if it is an array/sequence/struct) as well as other options. See the similar method in the ACI for the explanation of those optional arguments.

#### 4.8.4.1 Synopsis

```
const char *
getProperty_<property>(std::string &val,
                        OA::AccessList &list = <none>,
                        OA::PropertyOptionList &options = <none>,
                        OA::PropertyAttributes *attrs = NULL);
```

#### 4.8.4.2 Returns

This method returns a pointer to the **c_str()** contents of the first argument allowing the return value as a convenience to be used both in typical C++ output using **<<**, as well as C-style formatting operations (e.g. %s in **printf**).

### 4.8.5 *setProperty_<property>* — Slave Method

This method sets the value of a property in text format from a **const char *** or **std::string &** passed as an argument. It has an optional argument for controlling which part of the property to retrieve (if it is an array/sequence/struct etc.). See the similar method in the ACI for the explanation of the optional arguments.

#### 4.8.5.1 Synopsis

```
void
setProperty_<property>(const std::string &val,
                        OA::AccessList &list = <none>);
void
setProperty_<property>(const char *val,
                        OA::AccessList &list = <none>);
```

### 4.8.6 *get_<property>* — Slave Method

This method retrieves a scalar value of the property and returns it as its native binary type. It is only available for properties that are scalar types or arrays of scalar types. If the type is an array, there are arguments for indices of each array dimension.

It the scalar type is **string**, there are extra arguments (after index arguments for arrays), that either reference a **std::string** to fill or a **const char *** pointer and **size_t** length to use as a buffer for the string's value. In both these cases the return value is a **const char *** which simply points to the provided buffer. As with the **getProperty_<*property*>** method, this return value is for convenient use for C++ **<<** output or C formatted output.

#### 4.8.6.1 Synopsis

```
// E.g. for a scalar boolean property
bool get_<property>();
// For a float property with 2 dimensions
float get_<property>(unsigned idx0, unsigned idx1);
// For a string property
const char *get_<property>(char *buf, size_t length);
const char *get_<property>(std::string &buf);
// For a string property that is a single-dimension array
const char *get_<property>(unsigned idx0, char *buf, size_t length);
const char *get_<property>(unsigned idx0, std::string &buf);
```

#### 4.8.6.2 Returns

This method returns a scalar value from the property value in its native binary type, except for string properties which return a pointer to the provided buffer.

### 4.8.7 *set_<property>* — Slave Method

This method set a scalar value in the property. It is only available for properties that are scalar types or arrays of scalar types. If the type is an array, there are arguments for indices of each array dimension. After any index arguments, there is an argument for the value in its native binary type.

It the scalar type is **`string`**, the value is either a **`const char *`** argument or a **`const std::string &`** argument.

#### 4.8.7.1 Synopsis

```
// E.g. for a scalar boolean property
void get_<property>(bool value);
// For a float property with 2 dimensions
void set_<property>(unsigned idx0, unsigned idx1, float value);
// For a string property
void set_<property>(const char *value);
void set_<property>(const std::string &value);
// For a string property that is a single-dimension array
void set_<property>(unsigned idx0, const char *value);
void set_<property>(unsigned idx0, const std::string &buf);
```

### 4.8.8 *getLength_<property>* — Slave Method

This method retrieves the maximum allowable length for a string property (which could be an array).

#### 4.8.8.1 Synopsis

```
size_t getLength_<property>();
```

#### 4.8.8.2 Returns

This method returns a the maximum length of a string for this property.

### 4.8.9 Examples

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);
slaves.mydev.set_arrayprop(3, x);
x = slaves.mydev.get_array2d(1, 2);
slaves.mydev.set_array2d(1, 2, x);
slaves.mydev.stop();
slaves.mydev.start();
```

### 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. May use **memSize** below when there is only one allocation required.<br>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 memory allocation for the lifetime of the worker. 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 "**ocpidev 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. This generated header file 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 OpenCPI 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 **sampledata**, 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 'sampledata' operation on port 'in'
  struct __attribute__ ((__packed__)) Sampledata {
    uint64_t sample[1];
  } sampledata;
};
```

### *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 and use the OpenCPI log methods rather than printf etc.

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(), feraiseexcept(), 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(), strtof(), strtoimax(),
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](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(),
feholdexcept(), 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(), strftime(), strlen(), strncat(), strncmp(), strncpy(),
strpbrk(), strrchr(), strspn(), strstr(), strtod(), strtof(),
strtoimax(), strtok(), strtok_r(), strtol(), strtold(), strtoll(),
strtoul(), strtoull(), strtoumax(), strxfrm(), time(), tolower(),
toupper(), tzname, tzset(), va_arg(), va_copy(), va_end(),
va_start(), vsnprintf(), vsprintf(), vsscanf();
POSIX_SINGLE_PROCESS: confstr(), environ, errno, getenv(),
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 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.

# 8 Building RCC Workers

The CDG explains how all workers are built (with **ocpidev build**), and this also applies to RCC workers.

The most common settings for building workers are set as top-level attributes in the worker's OWD, as described in the CDG. This includes these settings:

- any extra include search directories
- component libraries to search for slaves for proxy workers
- platforms to exclude when building
- extra source files to compile with the worker

In the section Attributes of the Top-level RCCWorker Element above, additional build-related attributes specific to RCC workers are defined, such as those for prerequisite libraries.

Occasionally, how RCC workers are built requires more detailed control, and this is accomplished by creating a **Makefile** for the worker, including the single line:

```
include $(OCPI_CDK_DIR)/include/worker.mk
```

The variables shown below can be set in the Makefile *before* the above single required line in the **Makefile**.

## 8.1 RCC Compiler and Linking Options

For most RCC workers, the built-in compiler options are suitable and no extra variables are needed, and no **Makefile** need be created.

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** (upper case) for C or C++, **<*os*>** is the operating system of target (e.g. **linux** or **macos**), and **<*platform*>** is the specific RCC target platform. 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 with 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 |
|---|---|
| **`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. |

# 9 Worker Code Examples

### 9.1 C++ Language Examples

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

- Initial run condition is 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 protocol operation **Op1** on input, which is an array of 100 “shorts”.
- Two protocol operations **Op2** and **Op3** 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>
  <operation name='op3'/>
</protocol>
```

Note the **op3** operation has no arguments, indicating that **op3** messages will have an opcode indicated, but no data (message length of zero). The OCS for the component would be (in the **xyz-spec.xml** file):

```
<componentSpec>
  <property name='center_frequency' type='float' initial='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 {               // structure for properties
    // internal stuff here
    const RCCFloat center_frequency; // const: not set to volatile
  };
  class XyzWorkerBase {              // class inherited by worker
  // 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; // access to RCC data types and constants
using namespace XyzWorkerTypes;  // for generated types

struct XyzWorker : public XyzWorkerBase { // struct avoids "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 freq, 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;
  }
};  // end of worker class
```

### 9.1.1 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 RCC data types and constants
using namespace XyzWorkerTypes;  // for worker's generated types

struct XyzWorker : public XyzWorkerBase {
  // 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_waitingResponse(1 << XYZ_RESPONSE, RCC_NO_PORTS) {
    setRunCondition(&m_awaitingInput);
  }
  RCCResult
  run(RCCBoolean /* timedout */) {
    // Use run condition as state indicator
    if (getRunCondition() == &m_awaitingInput) {
      // do 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 to output, same op and length,
      setRunCondition(&m_awaitingResponse); // await response
    } else {
      other.send(response);           // got response, send to other
      setRunCondition(&awaitingInput);// change state, await input
    }
    return RCC_OK;
  }
}; // end of worker class
```

## 9.2 C Language Examples

Here is a simple example of a C-language **xyz** worker whose:

- Initial run condition is 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 named **`in`**,  and one output port named **`out`**.
- One protocol operation **`Op1`** on input (i.e. can ignore "operation"), which is an array of 100 "shorts".
- Two protocol operations **`Op2`**, and **`Op3`** on output
- One simple property, called **`center_frequency`**, of type **`float`**.

The worker has automatically generated types and structures like this (based on OCS and OWD), generated 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 Op1
  int16_t ishorts[100];
} XyzInOp1;

typedef struct {// structure for message for operation Op2
  int16_t oshorts[100];
} XyzOutOp2;

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

typedef enum { // operation ordinals
  XYZ_OUT_OP2,
  XYZ_OUT_OP3
} XyzOutOperation;

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

The actual code for the worker is below.

### *9.2.1 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;
}
```

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

### *9.2.2 Worker Using the State-machine Style*

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

    /* Call Process op on user port - buffer ownership passes back
     * to container.  Input port is advanced by taking buffer away
     * from it. */
    c->send(computeOut, &inPort->current, 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 run condition /
  return RCC_OK;
}
/* continued on next page */
```

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

# 10 Glossary of Terms

This glossary provides definitions for OpenCPI-specific and industry-wide terms and acronyms used in OpenCPI documentation.

### *10.1 OpenCPI Terminology*

This section provides definitions for terms that are specific to OpenCPI.

***ACI***

See ***Application Control Interface***.

***adapter worker***

An ***adapter worker*** is the worker used when two connected HDL workers are not connectable in some way due to different interface choices in the OWD. Adapter workers are normally inserted automatically as needed, e.g. between a worker that has a 16-bit bus and one with a 32-bit bus, or HDL workers in different clock domains. These workers are considered part of the OpenCPI framework and not created by users. See also ***worker***.

***application***

[noun] In the context of component-based development, an ***application*** is a composition or assembly of connected components that, as a whole, perform some useful function. See also ***component***.

[adjective] The term ***application*** can also be used to distinguish functions or code from infrastructure to support the execution of a component-based application; e.g., an HDL device worker vs. an application worker.

***Application Control Interface (ACI)***

The ***Application Control Interface (ACI)*** is an application launch and control API for executing XML-based OpenCPI applications within a C++ or Python program. See the chapter on the ACI in the ***OpenCPI Application Development Guide*** for more information.

***application worker***

An ***application worker*** is the implementation of a component used in an application, generally portable and hardware independent.

***argument***

See ***operation argument.***

***artifact***

An ***artifact*** is a file that contains executable code for one or more workers, built for a specific platform. An artifact is a binary file that results from building some assets. See the overview in the ***OpenCPI Application Development Guide*** for more information.

***asset***

An ***asset*** is an object that is developed for OpenCPI, including applications, components, workers, protocols, platforms, primitives and other asset types. An asset is developed in a project and is defined by an XML file.

***authoring model***

An ***authoring model*** is a method for creating component implementations in a specific language using a specific API between the worker and its execution environment. An authoring model represents a particular way to write the source code and XML for a worker and is usually associated with a class of processors and a set of related languages. Existing models include RCC, HDL and OCL. See the chapter on authoring models in the ***OpenCPI Component Development Guide*** for more information.

***back pressure***

***Back pressure*** is the resistance or force opposing the desired flow of data through an application. Back pressure within an OpenCPI system is a common occurrence that happens when worker output is temporarily not possible due to processing or communication congestion from whatever the output is connected to. Back pressure can be the result of resource-loading issues or passing data between containers.

***build configuration***

A ***build configuration*** is a set of parameter property (compile-time) values to use when building a worker. See the chapter on worker build configuration XML files in the ***OpenCPI Component Development Guide*** for more information.

***CDK***

See ***Component Development Kit***

***component***

A ***component*** is the interface "contract" specified by an OpenCPI Component Specification (OCS) and implemented by a worker. A component performs a well-defined function regardless of implementation. A component has ports and properties. See the chapter on component specifications in the ***OpenCPI Component Development Guide*** for more information.

***Component Development Kit (CDK)***

The OpenCPI ***Component Development Kit (CDK)*** is the set of OpenCPI tools, scripts, documents, and libraries used for developing components, workers and other assets in projects.

***component library***

A ***component library*** is a collection of component specifications, workers and test suites that can be built, exported, and installed to support applications. See the chapter on component libraries in the ***OpenCPI Component Development Guide*** for more information.

***component specification***

See ***OpenCPI Component Specification (OCS)***.

***component unit test suite***

A ***component unit test suite*** is a collection of test cases in a component library for testing all the workers in the library that implement the same spec (OCS) across all available platforms. The workers that are tested can be written to different authoring models or languages or simply be alternative source code implementations of the same spec. The OpenCPI unit test framework manages multiple dimensions of worker testing, with automation to minimize test design and preparation efforts. See the chapter on worker unit testing in the ***OpenCPI Component Development Guide*** for more information.

***configuration property***

A ***configuration property*** is a writeable and/or readable value specified in the OCS or OWD that enables control software to control and monitor a worker. Configuration properties (usually abbreviated to ***properties***) are logically the knobs and meters of the worker's "control panel". Each worker may have its own, possibly unique, set of configuration properties which can include hardware resources such registers, memory, and state. Properties can be specified as compile time or runtime. See the chapter on property syntax and ranges in the ***OpenCPI Component Development Guide*** for more information. See also ***configuration property accessibility***.

***configuration property accessibility***

***Configuration property accessibility*** is the set of declarations within an OCS or OWD that indicate when it is valid to read from or write to a configuration property. The various accessibility attributes (defined in the ***OpenCPI Component Development Guide***) establish the rules in relation to the worker's lifecycle and may declare the property as fixed at build time (see ***parameter***).

***container***

A ***container*** is the OpenCPI infrastructure element that "contains," manages, and executes a set of workers. Logically, the container "surrounds" the workers, mediating all interactions between the workers and the rest of the system. A container typically provides the OpenCPI runtime environment for a particular processor in the system. See the section on the RCC worker interface in the ***OpenCPI RCC Development Guide*** for more information on RCC containers. See the section on HDL container XML files in the ***OpenCPI HDL Development Guide*** for more information on HDL containers.

***control-application***

A ***control-application*** is the conventional application (e.g. "main program") that constructs and runs component-based applications. See the chapter on the ACI in the ***OpenCPI Application Development Guide*** for more information.

***control interface***

The ***control interface*** is the interface as seen by HDL worker code that an HDL container uses to provide (at a minimum) a control clock and associated reset into the worker, convey life cycle control operations like ***initialize***, ***start*** and ***stop***, and access the worker's configuration properties as specified in the OCS and OWD. See the section on the control interface in the ***OpenCPI HDL Development Guide*** for more information.

***control operations***

***Control operations*** are a fixed set of operations that every worker may have. These operations implement a common control model that allows all workers to be managed without having to customize the management infrastructure software for each worker, while configuration properties are used to specialize components. The most commonly used control operations are "start" and "stop". See the section on lifecycle control operations in the ***OpenCPI Component Development Guide*** for more information.

***control plane***

In OpenCPI, the ***control plane*** is the control and configuration infrastructure for runtime lifecycle control and configuration of worker instances throughout the system at runtime. See the control plane introduction in the ***OpenCPI Component Development Guide*** for more information.

***control software***

***Control software*** is the software that launches and controls OpenCPI applications, either the standard OpenCPI utility **`ocpirun`** or custom C++ or Python programs that perform the same function embedded inside them using the Application Control Interface application launch and control API. See the control plane introduction in the ***OpenCPI Component Development Guide*** for more information. See also ***control-application***.

Control software generally launches, configures and controls an application and the runtime workers that make up the application. A proxy, meanwhile, is a worker within an application that can control and configure other workers. See the ***OpenCPI RCC Development Guide*** for more information. See also ***device proxy worker***.

***core project***

The OpenCPI ***core project*** is a built-in OpenCPI project (package ID **`ocpi.core`**) that contains the minimum set of workers and infrastructure VHDL for OpenCPI framework operation on software and FPGA simulators.

***data interface***

A ***data interface*** is the set of ports defined in a worker's OCS that convey data, message boundaries, opcodes and EOF into and out of the worker and which implement flow control.

***data plane***

In OpenCPI, the ***data plane*** is a data-passing infrastructure that allow workers of all types to consume/produce data from/to other workers in an application regardless of the container on which the workers are executing in (or the processor on which they are executing). See the data plane introduction in the ***OpenCPI Component Development Guide*** for more information.

***device proxy worker***

A ***device proxy worker*** is a software worker (RCC/C++) that is specifically paired with one or more HDL device workers in order to translate a higher-level control interface for a class of devices into the lower-level actions required on a specific device. See the section on controlling slave workers from proxies in the ***OpenCPI RCC Development Guide*** and the section on device support for FPGA platforms in the ***OpenCPI Platform Development Guide*** for more information.

***device worker***

See ***HDL device worker***.

***Digital Radio Controller (DRC)***

The ***Digital Radio Controller (DRC)*** is a utility component in the OpenCPI built-in core project that is used when an application needs to use radio hardware in the system to control it and to stream sample data to and from it. The "digital radio" functionality in a system usually has antennas for transmitting and receiving RF signals and channels which convert the RF signals to and from baseband digital samples that are produced and consumed by the application. See the section on utility components for applications in the ***OpenCPI Application Development Guide*** for more information.

***HDL assembly***

An ***HDL assembly*** is a fixed composition of connected HDL workers that are built into a complete FPGA bitstream that can be executed on an FPGA to implement some or all of the components of an OpenCPI application. The HDL code is automatically generated from the HDL assembly's OHAD. See the chapter on preparing HDL assemblies for use by applications in the ***OpenCPI Application Development Guide*** and the ***OpenCPI HDL Development Guide*** for more information.

***HDL authoring model***

The ***HDL authoring model*** is the authoring model used by VHDL-language and less-supported Verilog-language workers that execute on FPGAs. See the ***OpenCPI HDL Development Guide*** for information about using this authoring model. See also ***Hardware Description Language (HDL)***.

***HDL build hierarchy***

The ***HDL build hierarchy*** is the structure in which FPGA bitstreams are created from other assets. See the section about the HDL build hierarchy in the ***OpenCPI HDL Development Guide*** for more information.

***HDL build process***

The ***HDL build process*** is the process of building HDL assets for different target devices and platforms. See the chapter on building HDL assets in the ***OpenCPI HDL Development Guide*** for more information.

***HDL card***

An ***HDL card*** is hardware that contains devices and plugs into a slot on an HDL platform. Devices are either directly attached to the pins on an HDL platform or attached to cards that plug into compatible slots on the platform. Devices on a card are considered to be part of the card, which can be plugged into a certain type of slot on any platform, rather than part of the platform itself. See the sections on device support for FPGA platforms and defining cards that contain devices that plug in to platform slots in the ***OpenCPI Platform Development Guide*** for more information.

***HDL data interface***

An ***HDL data interface*** is the set of ports defined in an HDL worker's OCS that convey data, message boundaries, opcodes and EOF into and out of the HDL worker and which implement flow control. Worker data ports can be implemented as stream or message interfaces. Stream interfaces are FIFO-like with extra qualifying bits along with the data indicating message boundaries, byte enables and EOF. Message interfaces are based on addressable message buffers. See the section on HDL worker data interfaces for OCS data ports in the ***OpenCPI HDL Development Guide*** for more information.

***HDL device emulator worker***

An ***HDL device emulator worker*** is a special type of HDL device worker that acts like a device for test purposes. A device emulator worker provides a mirror image of an HDL device worker's external signals so that it can can emulate the device in simulation. See the section on testing device workers with emulators in the ***OpenCPI Platform Development Guide*** for more information.

***HDL device worker***

An ***HDL device worker*** is a specific type of HDL worker designed to support a specific external device attached to an FPGA such as an ADC, flash memory, or I/O device. HDL device workers are typically developed as part of enabling an HDL platform. See the chapter on device support for FPGA platforms in the ***OpenCPI Platform Development Guide*** for more information.

***HDL interface***

(1) For HDL workers, an ***HDL interface*** is the set of input and output port signals that correspond to a high-level OpenCPI port as defined in the OCS and OWD for the HDL worker. An HDL worker has a control interface (for the implicit control port), data interfaces (for the explicit data ports defined in the OCS), and service interfaces (for service ports as defined in the HDL worker's OWD).

(2) For all worker types, an ***HDL interface*** is the implicit control port.

***HDL platform***

An ***HDL platform*** is an OpenCPI platform based on an FPGA that is enabled to host OpenCPI HDL workers. An HDL simulator is also considered to be an HDL platform. See the chapter on enabling FPGA platforms in the ***OpenCPI Platform Development Guide*** for more information.

***HDL platform configuration***

An ***HDL platform configuration*** is a pre-built (usually pre-synthesized) assembly of HDL device workers that represents a particular reusable configuration of device support modules for a given HDL platform. The HDL code is automatically generated from a brief description in XML. See the section on enabling execution for FPGA platforms in the ***OpenCPI Platform Development Guide*** for more information.

***HDL platform worker***

An ***HDL platform worker*** is a specific type of HDL worker that enables an HDL platform for use with OpenCPI and provides the infrastructure for implementing control/data interfaces to devices and interconnects external to an FPGA or simulator, such as PCIe or clocks. See the section on enabling execution for FPGA platforms in the ***OpenCPI Platform Development Guide*** for more information.

***HDL primitive***

An ***HDL primitive*** is an HDL asset that is lower level than a worker and can be used (and reused) as a building block for HDL workers. An HDL primitive is either a library or a core. See the chapter on HDL primitives in the ***OpenCPI HDL Development Guide*** for more information.

***HDL primitive core***

An ***HDL primitive core*** is a low-level module that can be built and/or synthesized from source code, or imported as pre-synthesized and possibly encrypted from third parties, or generated by tools like Xilinx CoreGen or Intel/Altera MegaWizard. An HDL worker declares which primitive cores it requires (and instantiates). See the chapter on HDL primitives in the ***OpenCPI HDL Development Guide*** for more information.

***HDL primitive library***

An ***HDL primitive library*** is a collection of low-level modules compiled from source code that can be referenced in HDL worker code. An HDL worker declares the HDL primitive libraries from which it draws modules. See the chapter on HDL primitives in the ***OpenCPI HDL Development Guide*** for more information.

***HDL slot***

An ***HDL slot*** is an integral part of an HDL platform that enables an HDL card to be plugged in so that its attached devices are accessible to the platform. An HDL platform has defined slot types; HDL cards that are designed for the same slot type can be plugged in to the defined slots on the platform. See the sections on device support for FPGA platforms and defining cards that contain devices that plug in to platform slots in the ***OpenCPI Platform Development Guide*** for more information.

***HDL subdevice worker***

An OpenCPI ***HDL subdevice worker*** is a special type of HDL application worker that supports an HDL device worker defined in another library. See the section on subdevice workers in the ***OpenCPI Platform Development Guide*** for more information.

***HDL worker***

An ***HDL worker*** is an HDL implementation of a component specification with the source code (for example, VHDL) written according to the HDL authoring model. An HDL worker is usually a hardware-independent, portable application worker but can alternatively be an HDL device worker that controls a specific piece of hardware attached to an FPGA. See the chapter on the HDL worker in the ***OpenCPI HDL Development Guide*** for more information.

***lifecycle state model***

The OpenCPI ***lifecycle state model*** specifies the control states each worker may be in and the control operations which generally change the state a worker is in, effecting a state transition. See the section on the lifecycle state model in the ***OpenCPI Component Development Guide*** for more information.

***OAS***

See ***OpenCPI Application Specification***.

***OCS***

See ***OpenCPI Component Specification***.

***OHAD***

See ***OpenCPI HDL Assembly Description***.

***OHPD***

See ***OpenCPI HDL Platform Description***.

***opcode***

See ***operation code***.

***OpenCL (OCL) authoring model***

The ***OpenCL (OCL) authoring model*** is the authoring model used by Open Computing Language (OpenCL) (C subset/superset)-language workers usually executing on graphics processors. See the ***OpenCPI OCL Development Guide*** for more information. This OpenCPI authoring model is currently experimental.

***OpenCPI Application Specification (OAS)***

An ***OpenCPI Application Specification (OAS)*** is an XML document that describes the collection of components along with their interconnections and configuration properties that defines an OpenCPI application. See the chapter on OpenCPI application specification XML documents in the ***OpenCPI Application Development Guide*** for more information.

***OpenCPI Component Specification (OCS)***

An ***OpenCPI Component Specification (OCS)*** is an XML file that describes both configuration properties and zero or more data ports (referring to protocol specifications) of a component, establishing interface requirements for multiple implementations (workers) in any authoring model. Also referred to as a *component spec* or *spec file*. See the chapter on component specifications in the ***OpenCPI Component Development Guide*** for more information.

***OpenCPI HDL Assembly Description (OHAD)***

An ***OpenCPI HDL Assembly Description (OHAD)*** is an XML file that describes an HDL assembly. See the chapter on HDL assemblies for creating and executing FPGA bitstreams in the ***OpenCPI HDL Development Guide*** for more information.

***OpenCPI HDL Platform Description (OHPD)***

An ***OpenCPI HDL Platform Description (OHPD)*** is an XML file that describes an HDL platform. An OHPD contains the same information as the OWD for the HDL platform worker and also describes the devices (controlled by HDL device workers) that are attached to the HDL platform and available for use. See the section on enabling execution for FPGA platforms in the ***OpenCPI Platform Development Guide*** for more information.

***OpenCPI Protocol Specification (OPS)***

An ***OpenCPI Protocol Specification (OPS)*** is an XML file that describes the allowable data messages (operation codes) and payloads (operation arguments) that may flow between the ports of components. See the chapter on protocol specifications in the ***OpenCPI Component Development Guide*** for more information.

***OpenCPI System support Project (OSP)***

An ***OpenCPI System support Project (OSP)*** is an OpenCPI project that contains OpenCPI assets whose purpose is to enable and test a particular system (of platforms) to be used by OpenCPI. OSPs fulfill what is generally meant by the more generic industry term: Board Support Package. An OSP may contain assets to support multiple related systems. See also ***Board Support Package (BSP)***.

***OpenCPI Worker Description (OWD)***

An ***OpenCPI Worker Description (OWD)*** is an XML file that describes the worker and references the component specification it is implementing. See the chapter on worker descriptions in OWD files in the ***OpenCPI Component Development Guide*** for more information.

***operation argument***

An ***operation argument*** is one of the data values in the payload data defined for a particular operation (message type) within a protocol specification whose type information is determined by the protocol XML.

***operation (within a protocol)***

An ***operation*** is a message type encapsulating zero or more operation arguments within an OpenCPI protocol specification.

***operation code (opcode)***

An ***operation code*** (***opcode***) is an ordinal that indicates which of the possible operations in a protocol is present.

***OPS***

See ***OpenCPI Protocol Specification***.

***OSP***

See ***OpenCPI System support Project***.

***OWD***

See ***OpenCPI Worker Description***.

***package ID***

A ***package ID*** is the globally-unique identifier of an OpenCPI asset. A project's package ID is used when it is depended on by other projects. A component's package ID is used to reference it in applications or workers. While all assets have package IDs (either explicitly specified or inferred from the directory structure), only certain assets are currently identified by their package IDs. See the section on package IDs in the ***OpenCPI Component Development Guide*** for more information.

***parameter***

A ***parameter*** is an immutable configuration property that is set at build time, allowing software compilers and hardware compilers to optimize accordingly. See the sections on properties that are build-time parameters, property accessibility attributes, and the parameter attribute of property elements and SpecProperty elements in the ***OpenCPI Component Development Guide*** for more information.

***platform***

An OpenCPI ***platform*** is a particular type of processing hardware and/or software that can host a container for executing OpenCPI workers based on artifacts. Platforms may be based on CPUs, GPUs or FPGAs. See the chapter on OpenCPI systems and platforms in the ***OpenCPI Platform Development Guide*** for more information.

***platform worker***

See ***HDL platform worker***.

***port***

An OpenCPI ***port*** is an interface of a component that allows it to communicate with other components using a protocol. Ports are unidirectional: input or output, consumer or producer. In OpenCPI, a port is a high-level data flow interface in and out of all types of workers. In the VHDL and Verilog languages, however, a "port" refers to the individual signals (of any type) that are the inputs and outputs of an entity (VHDL) or module (Verilog). See the chapter on component specifications in the ***OpenCPI Component Development Guide*** for more information.

***port readiness***

***Port readiness*** indicates whether an input port has data to be consumed or an output port has capacity to produce data (e.g. no back pressure). Input ports are ready when there is message data present that has not yet been consumed by the worker. Output ports are ready when buffers are available into which they may place new data.

***project***

An OpenCPI ***project*** is a work area (and directory) in which to develop OpenCPI components, libraries, applications, and other platform- and device-oriented assets. See the chapter on developing OpenCPI assets in projects in the ***OpenCPI Component Development Guide*** for more information.

***project registry***

An OpenCPI ***project registry*** is a directory that contains references to projects in a development environment so they can refer to (and depend on) each other. Development activity takes place in the context of a project registry that specifies available projects to use. See the section on the project registry in the ***OpenCPI Component Development Guide*** for more information.

***property***

See ***configuration property***.

***protocol specification***

See ***OpenCPI Protocol Specification (OPS)***.

***protocol summary***

A ***protocol summary*** is the set of summary attributes, whether inferred from the messages specified for the protocol, or specified directly as attributes of the protocol, and indicates the basic behavior of a port using a protocol. A protocol summary can also be present when messages are specified, and can override the attributes inferred from the message specifications. See also ***Component Development Kit (CDK)***.

***RCC, RCC authoring model***

See ***Resource-Constrained C (RCC) authoring model***.

***Resource-Constrained C (RCC) authoring model***

The ***Resource-Constrained C (RCC) authoring model*** is the authoring model used by C or C++ language workers that execute on General-Purpose Processors (GPPs). The "Resource Constrained" prefix indicates that the environment may be constrained to use a limited set of library calls; see the ***OpenCPI RCC Development Guide*** for more information.

***registry***

See ***project registry***.

***RCC worker***

An ***RCC worker*** is an RCC implementation of an OpenCPI component specification with the source code (for example, C++ or Python) written according to the RCC authoring model. An RCC worker can act as a device proxy worker. See the ***OpenCPI RCC Development Guide*** for more information.

***run condition***

A ***run condition*** is the specification by an RCC worker as to when it should execute, based on a combination of port readiness and/or some amount of time having passed. The commonly-used default run condition is when all ports are ready, with no consideration of time passing.

***run method***

A ***run method*** is a non-blocking software method that is executed when a worker's run condition is satisfied, as determined by its container.

***spec file***

***Spec file*** (and *component spec*) is shorthand notation for an OpenCPI Component Specification file.

***SpecProperty***

A ***SpecProperty*** is an XML element that adds a worker-specific attribute to a configuration property already defined in the component spec. See the section on worker descriptions in OWD XML files in the ***OpenCPI Component Development Guide*** for more information.

***system***

In OpenCPI, a ***system*** is a collection of platforms usually in a box or on a system bus or fabric.

***target***

An OpenCPI ***target*** is the entity for which an asset should be built (compiled, synthesized, place-and-routed, etc.) In OpenCPI, build targets are usually platforms (particular products or particular operating system releases and architectures). When a set of platforms shares a common processor architecture family, it is *sometimes* possible to build for the "family" and the results of that build can be used for all the platforms. See the section on RCC compilation and linking options in the ***OpenCPI RCC Development Guide*** and the section on HDL build targets in the ***OpenCPI HDL Development Guide*** for more information.

***worker***

An OpenCPI ***worker*** is a specific implementation (and possibly a runtime instance) of a component specification with the source code written according to an authoring model. See the introductory chapter on workers in the ***OpenCPI Component Development Guide*** for more information.

***worker property***

A ***worker property*** is a configuration property related to a particular implementation (design) of a worker; that is, one that is not necessarily common across a set of implementations of the same high-level component specification (OCS). A worker property is additional to the properties defined by the component specification being implemented. See the section on how a worker access its properties in the ***OpenCPI RCC Development Guide*** and the sections on property access and property data types in the ***OpenCPI HDL Development Guide*** for more information.

***unit test***

See ***component unit test suite***.

***Zero-Length Message (ZLM)***

A ***Zero-Length Message (ZLM)*** is a data payload with no operation arguments present when a protocol specification specifies such an operation code with no data fields.

### *10.2 Industry Terminology*

This section provides definitions for industry-wide terms relating to OpenCPI.

***Advanced eXtensible Interface (AXI)***

***Advanced eXtensible Interface (AXI)*** is an industry-standard bus used by ARM processors.

***Advanced RISC Machine (ARM)***

***Advanced RISC Machine (ARM)*** is a widely-used processor architecture originally based on a 32-bit reduced instruction set (RISC) computer.

***ARM***

See ***Advanced RISC Machine***.

***AXI***

See ***Advanced eXtensible Interface***.

***Board Support Package (BSP)***

A ***Board Support Package (BSP)*** is the layer of software in an embedded system that contains hardware-specific drivers and other routines that allow a particular operating system (usually a real-time operating system) to function in a particular hardware environment integrated with the operating system itself. An OpenCPI System support Project (OSP) performs the function of a BSP in OpenCPI.

***Central Processing Unit (CPU)***

A ***Central Processing Unit (CPU)*** is the electronic circuitry that executes instructions comprising a computer program. A CPU performs basic arithmetic, logic, controlling, and input/output (I/O) operations specified by the instructions in the program, in contrast with external components such as main memory and I/O circuitry and specialized processors such as Graphics Processing Units (GPUs).

***CPU***

See ***Central Processing Unit***.

***Digital Signal Processor (DSP)***

A ***Digital Signal Processor (DSP)*** is a specialized microprocessor chip with an architecture that is optimized for the operational needs of digital signal processing.

***digital signal processing***

***Digital signal processing*** (also abbreviated to “DSP”) is the use of digital processing by General-Purpose Processors (GPPs) or Digital Signal Processors (DSPs) to perform a wide variety of signal processing operations. The digital signals processed in this way are a sequence of numbers that represent samples of a continuous variable in a domain such as time, space, or frequency.

***DSP***

See ***Digital Signal Processor***. This acronym is sometimes also used more generically for digital signal processing as a class of computational algorithms.

***eXtensible Markup Language (XML)***

***eXtensible Markup Language (XML)*** is a standardized markup language that defines a set of rules for encoding documents in a format which is both human- and machine-readable.

***Field-Programmable Gate Array (FPGA)***

A ***Field-Programmable Gate Array (FPGA)*** is an integrated circuit that is designed to be configured by a customer or a designer after manufacturing. The FPGA configuration is generally specified using a hardware description language (HDL), similar to that used for an Application-specific Integrated Circuit (ASIC).

***FPGA***

See ***Field-Programmable Gate Array***.

***FPGA bitstream***

In the context of FPGA development, an ***FPGA bitstream*** is a single, standalone artifact, resulting from building an HDL assembly, that is ready for loading onto an actual, physical FPGA.

***framework***

A ***framework*** is a development and runtime tool set for a particular class of software, firmware, or gateware development. OpenCPI is a framework.

***gateware***

***Gateware*** is source code written in an HDL for an FPGA. Gateware is like software because it is fully programmable, but it compiles to fully parallel logic, which allows it to to compute efficiently like hardware. Gateware solutions achieve performance and flexibility by running on FPGAs.

***General-Purpose Processor (GPP)***

A ***General-Purpose Processor (GPP)*** is a processor designed for general-purpose computers such as PCs or workstations and for which computation speed is the primary concern. See also ***Central Processing Unit (CPU)***.

***GPP***

See ***General-Purpose Processor***.

***GPU***

See ***Graphics Processing Unit***.

***Graphics Processing Unit (GPU)***

A ***Graphics Processing Unit (GPU)*** is a chip or electronic circuit capable of rendering graphics for display on an electronic device. In the last decade, GPUs have also been used for more general-purpose computing when algorithms can exploit the same highly parallel architectures.

***Hardware Description Language (HDL)***

***Hardware Description Language (HDL)*** is a specialized language used to program the structure design and operation of digital logic circuits. In OpenCPI, it is an authoring model using the VHDL language and is targeted at FPGAs. HDL workers should be developed according to the HDL authoring model described in the ***OpenCPI HDL Development Guide***.

***HDL***

See ***Hardware Description Language***.

***Integrated Synthesis Environment (ISE®) Design Suite***

The Xilinx ***Integrated Synthesis Environment (ISE) Design Suite*** is a discontinued software tool for synthesis and analysis of HDL designs, which primarily targets development of embedded firmware for Xilinx FPGA and Complex Programmable Logic Device (CPLD) integrated circuit (IC) product families. Use of the last released edition continues for in-system programming of legacy hardware designs containing older FPGAs and CPLDs otherwise orphaned by the replacement design tool, Vivado® Design Suite.

***ISE® Simulator (Isim)***

The ***ISE Simulator (ISim)*** is the HDL simulator provided with the Xilinx ISE® Design Suite. In OpenCPI, this simulator is called the **`isim`** HDL platform.

***isim***

See ***Integrated Synthesis Environment (ISE®) Simulator (ISim)***.

***OCL, OpenCL***

See ***Open Computing Language***.

***Open Computing Language (OCL, OpenCL)***

The ***Open Computing Language (OCL, OpenCL)*** is a language and runtime for writing programs that, subject to the availability of appropriate tools, may execute on different types of processors, e.g. Central Processing Units (CPUs), Graphics Processing Units (GPUs), Digital Signal Processors (DSPs), Field-Programmable Gate Arrays (FPGAs) and other processors or hardware accelerators. OpenCL is an open standard maintained by the non-profit technology consortium Khronos Group.

***OSS***

See ***Open Source Software***.

***Open Source Software (OSS)***

***Open Source Software (OSS)*** is a type of computer software in which source code is released under a license in which the copyright holder grants users the rights to use, study, change, and distribute the software to anyone and for any purpose. Open Source Software may be developed in a collaborative public manner.

***PCI***

See ***Peripheral Component Interconnect***.

***PCIe***

See ***Peripheral Component Interconnect Express***.

***Peripheral Component Interconnect (PCI)***

***Peripheral Component Interconnect (PCI)*** is a local computer bus for attaching hardware devices in a computer and is part of the PCI Local Bus standard. The PCI bus supports the functions found on a processor bus but in a standardized format that is independent of any given processor's native bus. Devices connected to the PCI bus appear to a bus master to be connected directly to its own bus and are assigned addresses in the processor's address space. PCI is a parallel bus, synchronous to a single bus clock. Attached devices can take either the form of an integrated circuit fitted onto the motherboard (called a planar device in the PCI specification) or an expansion card that fits into a slot.

***Peripheral Component Interconnect Express (PCIe)***

***Peripheral Component Interconnect Express (PCIe)*** is a high-speed serial computer expansion bus standard that is designed to replace the older PCI, PCI-X and AGP bus standards. Improvements over the older standards include higher maximum system bus throughput, lower I/O pin count and smaller physical footprint, better performance scaling for bus devices, a more detailed error detection and reporting mechanism and native hot-swap functionality. More recent revisions of the PCIe standard provide hardware support for I/O virtualization.

***RPM***

See ***RPM Package Manager***.

***RPM Package Manager (RPM)***

The ***RPM Package Manager (RPM)*** is a free and open-source package management system used in some Linux distributions. The name “RPM” refers to **`.rpm`** file format and to the Package Manager program command itself.

***System on a Chip (SoC)***

A ***system on a chip (SoC)*** is a single integrated circuit (IC, or “chip”) that integrates all or most components of a computer or other electronic system. SoC is a complete electronic substrate system that may contain analog, digital, mixed-signal or radio frequency functions. Its components usually include a Graphics Processing Unit (GPU), a Central Processing Unit (CPU) that may be multi-core, and system memory (RAM). SoCs are in contrast to the common traditional motherboard-based PC architecture, which separates components based on function and connects them through a central interfacing circuit board. SoCs used with OpenCPI typically also contain FPGAs.

***Verilog***

***Verilog*** is a hardware description language (HDL) used to model electronic systems. Verilog is standardized as IEEE 1364.

***VHSIC Hardware Description Language (VHDL)***

***VHDL*** is a hardware description language used in electronic design automation to describe digital and mixed-signal systems such as FPGAs and integrated circuits (ICs). VHDL can also be used as a general-purpose parallel programming language.

***Vivado® Design Suite (Vivado, Xilinx Vivado)***

The Xilinx ***Vivado Design Suite*** is a software suite for synthesis and analysis of HDL designs. Vivado is an integrated design environment (IDE) with system-to-IC level tools built on a shared scalable data model and a common debug environment. Vivado supersedes Xilinx ISE with additional features for system on a chip development and high-level synthesis. Vivado WebPACK Edition is a free version of Vivado that provides designers with a limited version of the Vivado Design Suite environment.

***Vivado® Simulator***

***Vivado Simulator*** is an HDL event-driven simulator that Xilinx provides with Vivado Design Suite and WebPACK Edition. In OpenCPI, this simulator is called the **`xsim`** HDL platform.

***XML***

See ***eXtensible Markup Language***.

***Xsim***

See ***Vivado® Simulator***.