OpenCPI
Component Development Guide

OpenCPI Release: v2.4.6

Revision History

Revision	Description of Change	Date
1.01	Creation	2010-06-21
1.02	Add ocpisca information and HDL application information	2010-08-05
1.03	Add more detail to HDL building, and general editorial improvements	2011-03-28
1.1	Editing, platform and device aspects of ocpigen, HDL details	2011-08-01
1.2	Update for latest HDL details	2013-03-12
1.3	Rename document, add VHDL coding details and ocpihdl utility	2013-06-15
1.4	Add new and complete assembly and container info, and add two new ocpihdl functions	2013-11-06
1.5	HDL coding practices, more HDL platform information	2013-12-10
1.6	Convert to ODT format, add more clarity for parameter properties and readonly properties	2014-03-31
1.7	Use standard template., minor updates, add generic OWD content.	2015-02-27
1.8	Incorporate the authoring model reference content into this document	2015-10-31
1.9	New section on projects, removed the HDL content to a different doc, removed authors cleaned up and refreshed all content.	2016-01-11
1.95	Processed additional clarity issues raised	2016-04-12
2.0	Clarify project dependencies in 13.5	2016-05-18
2.1	Update for 2017.Q2, major ocpidev update, minor unit test update.	2017-08-06
2.2	Update for 2018.Q1, projects, registries, package-IDs, pass/fail+remote containers in tests	2018-02-26
2.3	Update for 2018.Q3, ocpidev run/show	2018-09-28
2.4	Update for release 1.5. version 2, EOF, property accessibility, make variable deprecation	2019-04-16
2.5	Update for release 1.6	2020-01-02
2.6	Update for release 1.7; update spec property attributes, add data plane introduction	2020-06-04
2.7	Update for release 2.1: remove references to CentOS6, add ocpidev man page info	2021-01-08
2.8	Add shared Glossary section	2021-03-15
2.9	Update for release 2.2: fix bad heading in Component Specs chapter, add note about meaning of asterisks in ocpidev show hdl platforms, change Makefiles to XML files	2021-06-30
2.10	Clarify message payload format	2022-04-15
2.11	Clarify use of default property values	2022-07-24

Table of Contents

1 References

This document also refers to concepts and definitions in other documents, but does not depend on them.

Table 1: References to Related Documents

Title	Published By	Link
OpenCPI User Guide	OpenCPI	https://opencpi.gitlab.io/releases/v2.4.6/docs/OpenCPI_User_Guide.pdf
OpenCPI RCC Development Guide	OpenCPI	https://opencpi.gitlab.io/releases/v2.4.6/docs/OpenCPI_RCC_Development_Guide.pdf
OpenCPI HDL Development Guide	OpenCPI	https://opencpi.gitlab.io/releases/v2.4.6/docs/OpenCPI_HDL_Development_Guide.pdf
OpenCPI Application Development Guide	OpenCPI	https://opencpi.gitlab.io/releases/v2.4.6/docs/OpenCPI_Application_Development_Guide.pdf

The ***OpenCPI Application Development Guide*** has introductory material on XML and the OpenCPI conventions for using XML. It also contains the description of the syntax for property values and expressions that is also used in XML files described in this document.

2 Overview

This document describes how to create OpenCPI component implementations (*workers*) in a component library, so that they are available for OpenCPI application developers and users. It introduces a kit of tools to specify and develop OpenCPI workers in any supported authoring model, language and API. It also describes how to create, build, and manage libraries of heterogeneous components where the components may have multiple implementations.

This document describes tools and processes to development workers in component libraries in general. Other documents describe the process of developing workers for specific authoring models, which currently include Resource-Constrained C Language workers ***RCC*** (C and C++ workers for software targets), Hardware Description Language workers ***HDL*** (VHDL or Verilog workers for FPGAs), and OpenCL workers, ***OCL*** (OpenCL workers for GPUs).

The kit of tools, scripts, documents and libraries used for developing components and workers in libraries are part of the ***OpenCPI Component Development Kit (CDK)***. The CDK is not an integrated development environment (IDE), but rather is a set of commands, shell level tools, and scripts that support the development process.

The CDK also includes tools specific to OpenCPI that support heterogeneous code generation and component testing. OpenCPI's code generation significantly reduces the code that needs to be hand-written in implementing heterogeneous components, applications, and FPGA bitstreams.

The OpenCPI CDK relies on technology-specific compilers (e.g. gcc), synthesis tools (for FPGAs) and simulation tools (e.g. Xilinx Vivado and Xsim, Altera Quartus, Modelsim etc.). These tools are a mix of open-source/free and commercially available products. Specific supported tools and versions are found in the ***OpenCPI Installation Guide***.

Several key concepts are described in the following sections, followed by the development process for libraries of components.

Component Specification: an XML file that describes a component in such a way that it may be implemented using different languages and APIs for different processing technologies and environments. It specifies the properties and ports of the component.

Protocol Specification: an XML file that describes the allowable data messages and payloads that are used for communication between components.

Property: writeable and/or readable values that enable configuration, control and monitoring of workers by control software at run time.

Port: 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.

Authoring Model: one of several ways of creating component implementations (workers) in a specific language using a specific API between the component and its execution environment. Existing models described below are RCC, HDL and OCL.

Worker: a specific implementation of a *component specification*, with the source code written according to an *authoring model*.

Component Library: a collection of *component specifications* and *workers* that can be built, exported and installed to support applications.

Project: a work area in which to develop OpenCPI components, libraries, applications, and other platform and device oriented OpenCPI assets.

3 Introduction to Worker Development

This section introduces the aspects of the worker development process that are common across all types of workers and authoring models. There are separate documents for each authoring model which describe their differing aspects in more detail, including languages and APIs. After this section introduces the general development process, following sections provide details for the contents of the various directories and files that are involved.

A worker is developed in its own directory, based on a component specification that typically exists in a file elsewhere. The component specification is the basis for multiple potential alternative implementations (workers). A component specification is an XML file called an ***OpenCPI Component Specification (OCS)***, abbreviated as a ***spec*** file, is described in detail in the Component Specifications section. The spec file also typically references one or more ***OpenCPI Protocol Specification (OPS)*** **files**, defined in the Protocol Specification section, to indicate the types of messages allowed to flow into and out of the component's ports.

In addition to workers having their own directories, they are developed in a component library (a collection of workers). The worker directories are thus subdirectories of the component library's directory, and the OCS (and OPS) for a worker is typically found in the **`specs`** subdirectory of the component library.

Some authoring models (e.g. RCC) support creating a single binary artifact that implements multiple workers, but usually a single worker implementation is in its own subdirectory and when compiled, results in a single binary artifact for each compilation target (e.g. type of CPU and compiler).

The names of the worker directories have a suffix indicating the authoring model used for that implementation (e.g. **`.rcc`**, **`.hdl`**). For a component whose component specification file is named **`xyz-spec.xml`**, the RCC authoring model implementation of that component will typically be in a worker directory called **`xyz.rcc`**. The worker's directory must combine the name of the worker, before the "**.**", and the authoring model used, after the "**.**". A worker named **`abc`** written using the authoring model named **`rcc`**, would exist in a directory named **`abc.rcc.`**

The names of the spec file and the worker's directory do not have to match, but it is recommended and allows the use of more defaults to simplify the process. An HDL implementation (for FPGAs) of the component spec **`xyz-spec.xml`** would be in the subdirectory **`xyz.hdl`**.

An **`xyz.test`** directory, at the same level as the worker directories, is created for unit tests common to all implementations of the **`xyz`** component's spec file. This means that tests in this directory apply to all workers that implement the same spec.

It is possible to have multiple workers implementing the same component specification, written in the same authoring model. In this case, the worker names must be different

and at least one of them must be different from the name implied by the component specification. E.g. one might have **`big_fast_xyz.rcc`** and **`small_slow_xyz.rcc`**, both implementing the OCS in **`xyz-spec.xml`**.

Once an OCS is available, a worker can be created, in a library, by using the **`ocpidev`** command, which creates a worker's directory and populates it with an initial version of several files that can be edited later. The **`ocpidev`** tool is described in the ocpidev section (and man page). The initial files created that are then edited as necessary include:

- ***OpenCPI Worker Description File*** (the ***OWD*** XML file, **`xyz.xml`**)
- Worker initial source code file (named **`xyz.<language-suffix>`**)

These file types have initial, automatically generated, skeletal contents that may be subsequently edited by the developer as required. Frequently only the source code file requires editing. These files are described in detail below. There are additional optional files in a worker's directory that will also be discussed below.

The ***OWD*** file is an XML file that describes the worker itself, refers to an OCS, and includes any implementation-specific attributes needed by OpenCPI. The second file is the initial source code file for the worker's actual logic.

The component specification (OCS) referred to by the worker's OWD contains the description of the component's external behavior. This OWD-to-OCS reference will exist in all workers that implement the component specification. The OWD adds information about a particular worker.

The OWD XML file has the name of the worker and the **`.xml`** suffix. The primary source code file has the name of the worker, with the typical suffix for the programming language used (**`.c`**, **`.cc`**, **`.vhd`** etc.). The primary source file for the **`xyz.rcc`** worker written in the C language would be **`xyz.rcc/xyz.c`**. A worker may also reference additional source files or libraries.

The worker building process (using **`ocpidev build`**) automatically creates and populates two types of subdirectories in the worker directory.

The first, called **`gen/`**, holds automatically generated source code and XML files that are target-independent (architecture independent). The second type, with the name **`target-<platform>`** holds architecture-specific object/binary files usually generated for or by a compiler for a specific target. In this case, ***`<target>`*** is the name of the compilation target being built, such as **`centos7`** for CentOS7 Linux running on a 64-bit x86 processor. Both types of directories contain files resulting from the build process and are removed by the **`ocpidev clean`** command (analogous to the typical "make clean" command), as they are always regenerated and should never be manually edited. More details about these targets is in the Developing Workers section.

In the sections below, a simple example will be given, followed by a detailed description of the component specification files (OCS), followed by the types of worker files just introduced.

3.1 A Simple Example Worker

Here is a simple example of a worker written in C++. The component's function adds a constant, **biasValue**, to each unsigned 32 bit integer at its input, and put the value at its output, one message containing many such values at a time. The component specification XML file, the OCS file, is named "**specs/bias-spec.xml**", and contains

```
<ComponentSpec>
  <property name='biasValue' writable='true' type='ulong'/>
  <port name='in' protocol='u32-proto'/>
  <port name='out' producer='true' protocol='u32-proto'/>
</ComponentSpec>
```

The protocol specification XML file, the OPS file, indicated by the **protocol** attributes in the OCS above, would be found in the file **specs/u32-proto.xml** (or elsewhere in other libraries or projects), and contains:

```
<Protocol>
  <Operation name='info'>
    <Argument name='values' type='ulong' sequenceLength='0'/>
  </Operation>
</Protocol>
```

For the **bias.rcc** worker created by **ocpidev**, implementing the above spec using the C++ language, the OWD XML file is named **bias.rcc/bias.xml**, and contains:

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

This OWD indicates that the authoring model is RCC, the spec is **bias-spec**, and the language is C++. The source file that implements this **bias.rcc** worker, simplified without header files or error checking, is in the file named **bias.cc**, and contains:

```
class BiasWorker : public BiasWorkerBase {
  RCCResult run(bool /*timedout*/) {
    size_t length           = in.info().values().size();
    const uint32_t *inData  = in.info().values().data();
    uint32_t *outData       = out.info().values().data();

    for (unsigned n = length; n; --n)
      *outData++ = *inData++ + properties().biasValue;
    out.info().values().resize(length);
    out.setOpCode(in.opCode());
    return RCC_ADVANCE;
  }
};
```

If the worker was written to the HDL model, in the VHDL language, its OWD would be:

```
<HdlWorker language='vhdl' spec='bias-spec'/>
```

For a detailed explanation for creating HDL workers see the ***HDL Development Guide*** document for VHDL examples.

4 Authoring Models

This section introduces the concept of an ***OpenCPI Authoring Model***, and defines aspects common to all authoring models. It specifies the concepts, lifecycle states and related operations, and OWD XML information used and manipulated by OpenCPI tools and edited by OpenCPI component developers.

The use of the term ***component*** is to define the functionality and an abstract interface. The term ***worker*** is used for particular implementation of a component written, (authored), using a programming language source code.

The definition of an authoring model can be referred to as *a way to write a worker*. A key goal is to support different processing technologies available such as ***General Purpose Processors (GPPs)***, ***Field-Programmable Gate Arrays (FPGAs)***, ***Digital Signal Processors (DSPs)***, or ***Graphical Processing Units (GPUs)***.

Since there is no one language, or API, that allows all these processing technologies to be utilized with efficiency and utilization comparable to their native languages and tool environments, we define a set of authoring models that achieve native efficiency with sufficient *commonality* with other authoring models to be able to:

- Implement an OpenCPI worker for a class of processors in a language that is efficient and natural to users of such processors
- Be able to switch or replace the authoring model and processing technology used for a particular component in a component-based OpenCPI application without affecting the other components of the application.
- Combine workers into an application using a variety of authoring models and processing technologies.

An OpenCPI authoring model consists of these aspects:

- An XML structure/schema/definition for OWDs, which describes aspects of the worker needed by tools and runtime infrastructure.
- Three programming language interfaces used for interactions between the worker itself and its environment:
  1. Control and configuration interfaces for run-time lifecycle control and configuration, referred to as the ***control plane***.
  2. Data passing interfaces for workers to consume/produce data from/to other workers in the application (of whatever model on whatever processor), referred to as the ***data plane***.
  3. Local service interfaces/APIs used by the worker to obtain various services locally available on the processor on which the worker is running.
- Mechanisms to build workers and package them for execution.

4.1 Requirements for All Authoring Models

- Enable well-defined data plane interoperability with other authoring models
- Define an OpenCPI Worker Description (OWD) XML format.
- Define programming language interfaces for control, data, and local services.
- Define the packaging for delivering ready-to-execute workers.

The currently supported authoring models are:

RCC (for ***R***esource-***C***onstrained ***C***-language) is an authoring model used in the C or C++ language workers that execute on general purposes processors (GPPs). The C language model is a lean model well-suited to small resource-constrained processors such as embedded CPUs, DSPs or micro-controllers. The C++ variant is more powerful and more compact, carries a slightly higher resource footprint, and of course requires a C++ compiler. Developing workers according to the RCC authoring model (either C or C++) is fully described in the ***RCC Development Guide***.

HDL (for ***H***ardware ***D***escription ***L***anguage) is an authoring model using the VHDL (and less-supported Verilog) languages and is targeted at FPGAs. Developing workers according to the HDL authoring model is fully described in the ***HDL Development Guide***.

OCL (for ***O***pen***CL***) is an upcoming authoring model using the OpenCL (C subset/superset) language targeting graphics processors. It is fully described in the ***OCL Development Guide***. This support is experimental as of the current release.

4.2 Control Plane Introduction

The material in this introduction is common to all authoring models. We use the term ***control software*** to describe software that launches and controls OpenCPI applications. This is either the standard utility program, **`ocpirun`**, or custom C++ programs that perform the same function embedded inside them. Such custom programs use the ***Application Control Interface (ACI)***, an application launch and control API described in the ***OpenCPI Application Development Guide***.

We use the term ***Control Plane*** as the infrastructure to support how control software, usually running in a centralized host processing environment, can control worker instances at runtime. The entity that is doing the controlling (or managing) is the *control application*, or simply *control software*. The control software accesses all controllable worker instances the same, regardless of where they are running, on what type of processing technology, and with what authoring model they were written.

While control software sees a uniform view of how to control workers, each authoring model defines how this is accomplished from the point of view of the worker itself. In particular, each authoring model defines how the two key aspects of control are made visible to the worker's source code: *LifeCycle control* and *Configuration Property access*. The documents describing each authoring model give additional interface details of these interactions, but they all follow a common pattern which is defined here.

4.2.1 LifeCycle State Model

Most component-based systems have an explicit lifecycle state model, where workers are instantiated and then managed, according to a lifecycle state machine. Normally all components in the application are managed together and they all progress through the lifecycle together. However, there are cases where control software must control (start/stop etc.) some components in the application different than others.

The LifeCycle model is defined by the *control states* each worker may be in, and *control operations* which generally change the state a worker is in, effecting a state transition. The possible states and transitions are shown in the following diagram.

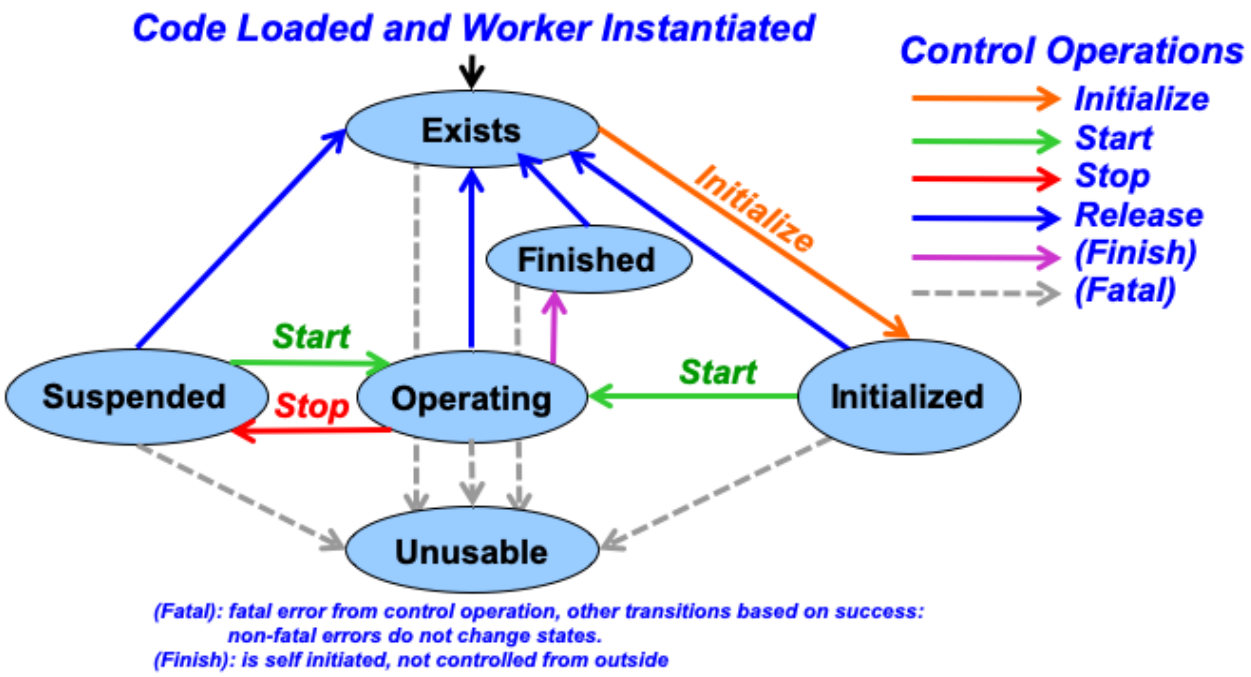

Diagram 1: Control States and Operations

4.2.2 LifeCycle Control Operations

Workers have default implementations for control operations that only perform the state transitions: workers code only needs to implement the operations that have custom behavior. A good example is the *initialize* control operation. If the worker has no runtime initialization to perform, it can have no implementation of this operation and not even have an empty "stub". Each authoring model describes which control operations must have implementations, if any.

Control operations can have two error types: transient or fatal. Transient errors imply that no state change occurred and the operation can be retried. Fatal errors imply that the worker instance has become unusable and needs to be reloaded.

Control software is required to issue control operations correctly, in sequence, so workers can avoid checking for valid states and transitions. State descriptions are listed in the table below:

Table 2: Control States

Control State	Previous State(s)	Allowable Operations	Description
Exists	*Initial state or all others after* ***release****, except* ***unusable***	Initialize	Follows instantiation or a successful ***release****.* Worker is loaded (if necessary), not fully initialized, with no properties valid, and property access not allowed.
Initialized	Exists	Start, Release	Initialization complete due to ***initialize.*** In a stable state "ready to start doing work", not "operational". Properties can be read and/or written.
Operating	Initialized, Suspended	Stop, (finish), Release	Member doing normal work using properties and data at ports. Properties can be read or written. Follows ***start***.
Suspended	Operating	(Re)start, release	Member not operating, will not produce or consume at data ports. Properties can be read or written. Can be resumed via ***start***.
Finished	Operating	Release	Member is finished and will not produce or consume at data ports. Properties can be read or written. Entered *autonomously*.
Unusable	All (fatal errors)	none	Fatal error state, may not be ever reusable without reloading (model-dependent).

4.2.3 Configuration Properties

Configuration properties are specified in the OCS or OWD, and are writeable and/or readable values that enable control and monitoring of workers by control software. They are logically the knobs and meters of the worker's "control panel". All authoring models provide an interface enabling control software to access (read and write) these values.

Some authoring models define a flat/linear configuration address space where the configuration properties are accessed by accessing this memory space, roughly as a data structure or register file.

The component specification for the worker contains the description of the configuration properties that are part of the component's external behavior. These will exist in all implementations (workers) that reference the component specification. However, each worker may also *add* to this set of configuration properties and define ***implementation-specific*** configuration properties. These can be useful for implementation debugging and testing, or in some cases to allow applications to configure properties specific to a particular implementation.

Each configuration property is defined with a name and data type from the data types listed in the table below. Each configuration property may vary in length (i.e. be strings or sequences) but must specify a maximum length. This enables components and workers to be compliant with a variety of component system standards, and enables authoring models for resource-limited embedded technologies.

Data types for configuration properties are based on the ***scalar types*** listed in the following table. A ***property data type*** can be one of the scalar types or a ***structure*** with typed members that are ***property data type***s. A property data type may have array dimensions, and also (after any array dimensions are applied) be defined as a variable length, single dimension ***sequence***. This recursive definition allows for complex types such as: *a sequence of a two-dimensional arrays of structures containing members that are themselves arrays or sequences or structures*.

Table 3: Scalar Data Types for Properties

Scalar Data Type	Support for Unsigned Version	Data Size	Notes
bool	*N/A*	8 bits	
char	*Yes*	8 bits	Unsigned version is **uchar**.
double	*N/A*	64 bits	Consistent with IEEE floating-point types
enum	*N/A*	32 bits	Types are represented by ulong values, but are associated with string names.
float	*N/A*	32 bits	Consistent with IEEE floating-point types
long	*Yes*	32 bits	Unsigned version is **ulong**.
longlong	*Yes*	64 bits	Unsigned version is **ulonglong**
short	*Yes*	16 bits	Unsigned version is **ushort**
string	*N/A*	fixed	Null terminated with a defined max length

When a property's type is a multidimensional ***array*** of the above types, the number of dimensions is fixed, and the length in each dimension is fixed.

When a property's type is a variable length ***sequence*** of the above types (or arrays of above types), it has a current length (number of valid values present), but is still required to have a defined maximum possible length (capacity). Sequences may have a current length of zero or any amount up to the specified maximum possible length. E.g. if a sequence is defined with a maximum length of 4, it means that it may hold zero, 1, 2, 3 or 4 values. Space is always reserved for the maximum number of values, but the current length is also recorded in the sequence and is set whenever a new value for the sequence is set.

Beyond data type, the configuration properties also have ***accessibility attributes*** indicating whether the value can only be:

- set at initialization time
- set any time during execution
- never set (read-only)

Similarly, a configuration property value can be described as volatile where the value may be changed by the worker during execution, or statically readable and will not change unless written by control software. These accessibility types are described in detail in the Accessibility Attributes section.

An OWD file allows additional properties to be defined unique to the worker, beyond those specified in the OCS. Additionally, the OWD may *add* to the accessibility of an OCS property. E.g., for debug purposes, the OWD may make a property readable that was not readable in the OCS. The accessibility added by the OWD results in the implementation having a superset of what was described in (and required by) the OCS.

4.2.4 Properties That Are Build-time Parameters

While the OCS specifies properties and their initialization-time and run-time accessibility, an OWD can further declare that a property is a compile time parameter in this worker. This is not allowed for properties declared as writable at runtime, either in the OCS or OWD. When an OWD declares properties to be parameters, this means that the worker will be built compiled, synthesized, elaborated for specific values of such properties. This has three implications:

- An application can only use the worker if it is built for a property value that matches the value requested for an “initial” property in the application.
- Binary component libraries may have multiple binary artifacts for the same worker, but with different combinations of parameter values.
- The component developer must decide which combinations of parameter values to build, in order to make alternative settings of such parameters available.

This parameter feature allows implementations to have compile time optimization for certain values. It also allows a single worker's source code module to be optimized for specific compile-time values.

Parameter property values are applied to the build process as per the language of the authoring model: e.g. by preprocessor symbol definitions for C, **`static const`** values for C++, generics for VHDL, and parameters for Verilog. A framework-generated (built-in) parameter property is the **`ocpi_debug`** Boolean property that specifies the typical debug build vs. production build. Parameter properties are described in more detail in the Parameter Attribute OWD section.

Each authoring model specifies how properties, at runtime and compile time, appear to the worker code. In addition it specifies how the worker may access property values written by control software, and set values that will be read by control software. Finally, different authoring models define how workers know when control software actively reads or writes these values.

4.3 Data Plane Introduction

Components (and thus workers) communicate with each other by sending and receiving messages via ports. Ports are specified as sending or receiving messages whose content is defined by a protocol. A protocol is simply a defined set of message types (see the Protocol Specifications section below). Thus a port will send or receive messages of the types defined in its protocol.

When a message is sent or received, it is accompanied by two items of metadata, usually out-of-band (not embedded in the message payload):

- an opcode, which is an ordinal indicating which message type it is (if there is only one message type in the protocol, the opcode will always be zero).
- a length, in bytes, indicating how long the message payload is.

Messages are sent or received this way whether the other connected worker is in the same container (colocated) or on a different processor entirely. Messages are placed in buffers by the sending worker and retrieved from buffers in the receiving worker. When an application is run, the workers are instantiated in containers, and every connection between worker ports is established. Some connections are implemented with one or more fixed size buffers, so in general the sending worker must assume that the (output) buffer it is using for sending a message has a size limit.

Workers must assume fixed size output buffers since that is the case some of the time. Every worker can get the runtime size limits of its output buffers, using a method defined by its authoring model. If the protocol specifies fixed size messages, this worker can assume output buffers will always be large enough to contain any message type in the protocol. If the protocol specifies some variable size messages, usually using **`sequence`** data types, it must ensure that the output buffer size is respected, by limiting the (variable) size of output messages produced.

There is a built-in (always present) property for each output port, whose name is **`ocpi_buffer_size_<port>`**, that all workers can examine for this purpose. Some authoring models have separate convenience methods for getting this value. This output port buffer size property value is set externally, at application start time, which is why it is a runtime value, not a compile time constant value.

How the actual contents of messages are read from input ports and written to output ports is defined for each authoring model. Message payloads (when seen as a sequence of bytes), follow a conventional structure-like format, with all scalar values aligned per their size in bytes, and any array or structure is aligned for the maximum alignment of any of its members. Sequences (variable length) are preceded by an aligned 32 bit number-of-elements value, followed by aligned sequence element data. Strings are aligned on 4-byte boundaries, and are null terminated. More details about message payload formats is in the Message Payloads section.

A message type in a protocol may be specified as having no data, i.e. be zero length.

4.3.1 End-of-File (EOF) Indications on Data Ports

Regardless of protocol, the OpenCPI messaging system (its ***data plane***) supports an ***EOF*** indicator in the data paths between workers of all types that indicates that no more messages/data will arrive at input ports or that no more messages/data will be produced on output ports. This indication is a persistent condition that will only be removed when the application and workers are restarted. It is not a message or a protocol operation.

Such an indication only happens when some worker asserts it on an output port, which then propagates it (down-stream) to the input port of the worker that the output is connected to. The most common usage of this EOF indication is when data is being read from a file (e.g. by a **`file_read`** component), and when that component encounters the end of the file it is reading. Such a worker typically has three options:

- Assert an ***EOF*** on its output port after the last message/data is sent, and enter the ***finished*** control state.
- Do nothing on the port, and enter the ***finished*** control state.
- Start reading the file from the beginning again.

Thus in a typical execution of an application where end-of-data is a meaningful condition, the ***EOF*** will propagate through all workers to the final "sink" components (e.g. **`file_write`** or **`capture`**) that do not have output ports. In this way all workers know that the application is finished, and all can cleanly complete their work (perhaps flushing pipelines, closing output files etc.). In this way the EOF indication on the data plane is related to the action that a worker would take to enter the ***finished*** control state. I.e. the reception of EOF would commonly result in the worker entering that state.

For continuous applications where the concept of EOF is not meaningful, it will never be encountered by any worker since no worker will assert it on an output port.

The unit test framework described in Unit Testing makes use of EOFs to execute repeatable tests of components. Applications that process data sets of bounded size will also find the use of EOF to be convenient and useful, since it makes it clear to all components as well as the application as a whole, that the application is "finished".

For most workers, the propagation of the EOF indication from its input port(s) to its output port(s) is automatically done without their involvement. Input data will simply stop arriving. However, some workers have more complex interactions with data ports, and perhaps more complex internal pipelining or state that requires special treatment when the last data arrives. They will recognize the EOF indications and perform final processing tasks accordingly. This latter type of worker declares "I will deal with EOFs", (using the **`WorkerEOF`** attribute) and the OpenCPI infrastructure will not perform the automatic EOF propagation from input to output. The default propagation of EOFs (when no **`WorkerEOF`** attributes are set), is to propagate an EOF arriving on the first input port to all output ports.

Each authoring model has an interface for the worker to sense the EOF indication on input and assert it on output. These interfaces are described in the respective documents specific to the authoring models (e.g. RCC and HDL development guides).

The next section describes important issues of backward compatibility relating to EOF indications for workers implemented prior to release 1.5, which are still supported.

4.3.1.1 EOF Processing Prior to Release 1.5

Prior to release 1.5, the notion of EOF was overloaded with, and frequently confused with, the notion of a zero-length-message (a.k.a. ZLM). The pre-1.5 **`file_read`** component, upon encountering the end of the file it was reading, would (as a default), send a zero length message with the opcode of zero as an EOF indication.

This would directly conflict with protocols whose first operation happened to have no arguments. In general, users avoided the conflict by not defining protocols with the first operation having no arguments. And it was impossible to use the EOF concept with a "pure single event" protocol that only had one message with no arguments, which is useful for sending events (pulses) between components. These difficulties disappear with the introduction in release 1.5 of a cleanly separate EOF indication that has nothing to do with messages and protocols.

Workers written prior to release 1.5 that supported EOF for unit test or other reasons, did so by explicitly adding code to interpret zero-length messages with opcode zero (ZLM0s) as EOF, and passing them from input to output. A small number actually acted on the reception of ZLM0 to perform certain flushing operations. This ZLM0 recognition was required in order to make best use of the unit test framework and the extra worker code to process ZLM0s was considered "worth the extra code" in order to facilitate repeatable testing.

4.3.1.2 EOF-related Compatibility Measures for Pre-1.5 Workers

In order to maintain backward compatibility with all such workers, in release 1.5 a "worker interface version" attribute was introduced in OWDs which indicates whether a worker expects to see the older ZLM0 messages as EOFs or not. Newer workers set the version attribute to 2 (or later) in their OWD and get the benefit of the new, simpler, orthogonal EOF model (and other interface improvements introduced in release 1.5).

Workers that do not have this attribute are assumed to be "pre-version2", which means that if and when an EOF indication is asserted at its input port, a ZLM0 is injected at that input port so the worker sees the ZLM0/EOF as it did before. When such a worker sends a ZLM0 on its output port, it is interpreted as the assertion of EOF.

Workers that were written to not process ZLM0s as EOF (because they would never see them), and not have protocols whose first operation had no arguments, would never see or generate ZLM0, so they are also compatible with release 1.5.

These compatibility measures do not extend to workers that *do not* process ZLM0 as EOF but *do* use protocols whose first operation has no arguments. In this case when they produce a ZLM0 it will be incorrectly interpreted as EOF. A separate option to suppress this output-ZLM0-recognition-as-EOF may be developed in a patch release if it is discovered that this type of worker actually exists and deserves this additional compatibility treatment rather than requiring conversion to a new “version 2” worker.

There is a discussion in the various authoring model documents about such conversions/upgrades for migrating older workers to use version 2 worker interfaces.

4.4 Software Execution Model

The material in this section applies to most software-based authoring models (e.g. RCC and OCL). It does not apply to the HDL (FPGA) authoring model.

Execution of software workers is based on a construct called ***Containers***. Containers supply threads and execute the workers. This eliminates the need for workers to create or manage threads. This reduces the complexity of the worker code, eliminates any requirement to support a threaded API capability, and allows the container to determine the level of multithreading that is needed. The authoring model defines how the threading is provided.

Execution of a worker occurs when the worker is transitioned into the ***operating*** control state. Workers are only executed when either a combination of its ports are ready (port readiness), or an amount of time has elapsed. The combination of port readiness and the passage of time is referred to as the worker's ***run condition***: the condition under which it should be run.

Every worker has an entry point called its ***run method***. When a worker's run condition is satisfied, determined by its container, its run method is called. Worker execution is a series of “runs” initiated by the container. The worker's run method cannot block, but returns after doing some work, allowing the container itself to determine when the worker should be entered again: when its run condition is once again satisfied.

The container calls the worker's run method when the worker’s run condition is satisfied. Run conditions are satisfied based on 1) the availability of input buffers, with data, or output buffers, with space, at a worker's ports or 2) the passage of time. The worker’s run method performs some processing and may:

- process some available messages at some of its input ports
- produce messages at some of its output ports
- indicate when messages are consumed as input or produced as output
- make any changes to its run condition

It then returns control to its container. Workers never block. The container conveys the messages in buffers between collocated workers as well as into and out of the container as required by the application assembly’s connections between components.

The container determines when the worker should run, supplies it with buffers full of input messages, and buffers into which output messages may be placed.

4.4.1 Run Conditions

Workers declare a run condition which tells the container under what conditions the worker should run. The container evaluates the run conditions of all workers and runs them as resources and priorities allow.

The run condition object contains two aspects: *port readiness* and *time*. The worker is run when its port readiness requirements are satisfied, or a specified amount of time has passed. Either or both aspects can be specified.

While the worker is in the operating state, port readiness means that buffers are available at that port to be used by the worker. Input ports have available buffers 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 the worker may place new data. I.e. input ports are ready when the worker has data to consume, and output ports are ready when the worker has room to produce new data into a buffer. Workers may partially consume or partially produce messages in any given run.

This port readiness model implements simple data driven execution: code is run when data can flow. The ***default run condition*** specifies that the worker should run if data is available at *all* input ports and space is available at *all* output ports (or conversely, there are no ports that are not ready). Note that this default, for workers with *no* ports, means they are always ready to run.

The time aspect of run conditions, indicated by the worker, specifies the desired maximum time between invocations of the worker's run method. If no port readiness is also specified, this simply indicates periodic execution. If the time aspect is specified *with* port readiness, it indicates that execution should take place when either the indicated port readiness conditions are satisfied –*or*– the indicated amount of time has passed since the worker's run method was last entered.

The default time aspect of run conditions is: no such maximum time at all. In this case time passing does not affect worker execution, only port readiness.

A worker may change its run condition at any time during the execution of its run method by passing a new run condition to the container, to be considered after the run method returns to the container.

In summary, run conditions specify a combination of data-driven and time-driven execution. Most workers use one or the other, but both can be used together. The defaults allow most workers to never have to indicate any run condition at all.

4.4.2 Sending or Receiving Messages via Ports

The worker indicates data flow to the container under two conditions. The first occurs when the worker has consumed the message in an input buffer at a port. Notification of that consumption allows that buffer to be released and reused. The second happens when the worker has finished placing a message in an output buffer at a port. This allows the message in the buffer to be sent on to the other worker connected to that port.

4.4.3 Buffer Management

The container provides and manages all buffers and provides references to buffers to the worker. Input ports operate by the container providing buffers to the worker, filled with incoming messages. Output ports operate by the container providing buffers for the worker to fill with messages before being sent. Output buffers are either:

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

Workers may modify data in input buffers, allowing input buffers to be used for temporary storage, to reduce overall memory requirements. When reuse occurs, the buffers must be annotated in the worker description XML. This ensures the container does not share the buffer with another consumer of the same data.

Several more advanced buffer management requirements are supported for certain situations:

- To support sliding window algorithms, workers are allowed to retain ownership of previous buffers by not releasing them while new ones are requested; i.e. allow explicit in order input buffer release, not just the most recent buffer obtained. The worker must still release the buffers in the order received.
- To support zero copy from input ports to output ports, workers are allowed to send a buffer obtained from an input port to an output port. This method does not require an empty current buffer to fill on the output port. Such buffers must be sent, or released, in the order received. This avoids copying data from input buffers to output buffers.

The advanced features listed above are only needed in certain cases, and can be ignored for most simple workers. To support these more advanced modes, non-blocking interfaces for explicitly releasing, sending, and requesting buffers are available.

There are four operations performed with message buffers. These provide the basis for specific non-blocking functions in the APIs defined for each authoring model.

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

- ***Send*** (enqueue) a buffer on an output port, to be automatically released after the data is sent. The ownership passes from worker to container. If the buffer was originally obtained from an input port, it must be sent or released in the order received.
- ***Take*** the current buffer from an input port such that is it no longer the current buffer of the port, but ownership is retained by the worker. This allows new input buffers to be made available while the worker holds on to previous buffers. A ***take*** implies a request for new buffers. This function allows workers to use previous buffers to hold history data for algorithms such as sliding window or moving average, without allocating any additional storage.

The concept of the ***current buffer*** of a port supports a model for workers that have no need to be aware of buffer management. A port is ***ready*** if it has a current buffer. A current buffer on an input port is available to read data from. A current buffer on an output port is available to write data into. The concept of ***advancing*** a port, is simply a combination of releasing (input) or sending (output) the current buffer of the port, and requesting a new buffer to be made available on that port, to become the current buffer when it becomes available in the future:

- advance = release_or_send + request

Simple workers, using the default run condition, wait for all ports to be ready, process input buffers into output buffers, advance input and output ports, and return.

Worker APIs defined by the authoring model are designed to make this common pattern as simple as possible. Workers are run when ports are ready, and they advance ports after processing messages in current buffers.

5 Protocol Specifications (in OPS XML Files)

An ***OpenCPI Protocol Specification (OPS),*** describes, in one or more XML files, the set of messages that may flow between the ports of components. They are described separately from the OCS XML file because they are used by both sides of a connection. In a connection between component ports, the specs of both ports, in their Port elements, refer to the same OPS.

The OPS describes the set of messages defined in the protocol, as well as some top level attributes for the protocol.

In special cases, the messages in a protocol are not specified individually but rather a set of summary attributes is specified. This indicates the basic behavior of the ports using the protocol. The information is called a ***protocol summary***. When messages *are* specified, protocol summary attributes are inferred from the messages. If protocol summary attributes are present when messages *are* specified, they override the attributes inferred from the message specifications.

As an example, a set of messages of different lengths and different payload formats might be bounded (having a maximum length), or unbounded, depending on whether any message has no maximum length. This ***boundedness*** attribute is normally inferred from the set of messages. Another example is the smallest unit of data in any message. If all messages in a protocol deal only with 64 bit integers, then the smallest unit of data for all messages is 8 bytes. This ***minimum data granularity*** attribute is inferred from examining all the messages specified for the protocol.

A protocol summary is the set of all summary attributes, whether inferred from the messages specified for the protocol, overridden by explicit values, or given directly when messages are not specified. The summary attributes are used by the OpenCPI code generation tools and runtime environment to determine certain behaviors.

In OPS files, messages are called ***operations***, and fields of messages are called ***arguments***. This is terminology based on the Remote Procedure Call (RPC), or Remote Method Invocation (RMI), model of communications. However, this concept does not apply to OpenCPI inter-component communications, as all communications are simply unidirectional connections conveying messages.

The term ***opcode*** is used to represent a zero-origin ordinal of operations within a protocol. In the runtime environment, opcodes are used to indicate which operation of the protocol a given message represents. Thus if the opcode of a message is zero, then that message should be interpreted as the first operation in the protocol.

OPS files preferably carry the suffix **`-prot.xml`**, although **`-protocol.xml`**, **`_protocol.xml`** are also used.

5.1 Protocol Element as Top-level Element

The **`Protocol`** element is a top-level element in a separate file whose name (without the **`.xml`** suffix) is the value of the **`Protocol`** *attribute* in a **`port`** element in an OCS. It specifies a message protocol used at ports. The protocol will likely be reused across a variety of components and ports since it specifies how two components talk to each other. The **`Protocol`** element has **`Operation`** subelements to indicate the different message types that may flow out of or into data ports using this protocol.

A **`Protocol`** element may also use the **`<xi:include href='<file>'/>`** syntax as a child element, which allows the operations of one protocol to be included at that point in another protocol. I.e. the operations in the protocol file included are inserted into the protocol at the point in the sequence of **`operation`** elements where the **`xi:include`** element is placed. The **`xi:include`** element is usually placed first so that the opcodes of the included protocol remain the same in the new "derived" protocol. An example:

```
<protocol>
  <xi:include href='base-proto'/>
  <operation name='special_add_on' .../>
</protocol>
```

5.1.1 Name Attribute of Protocol Elements

When the **`Protocol`** element is the top-level element in a file, the optional name attribute is defaulted from the name of the file, with any **`-prot`**, **`_prot`**, **`-protocol`**, **`_protocol`**, **`.xml`** suffixes removed. Since protocols are usually defined in separate files, the names are usually not present in the XML and are derived from the file name.

5.1.2 Protocol Summary Attributes

These attributes are normally inferred from **`Operation`** elements in the protocol and are rarely used explicitly. They may be used to override the inferred values or they may be specified in the absence of **`Operation`** elements altogether. The following table defines the protocol summary attributes normally inferred from examining **`Operation`** elements.

Table 4: Protocol Summary Attributes

Name	Type	Description
DataValueWidth	**ulong**	Size in bytes of smallest unit of data in any message
DataValueGranularity	**ulong**	All messages have this multiple of data values. I.e. if messages always of an even number of values, it is 2.
DefaultBufferSize	**ulong**	Default buffer size for this protocol even if unbounded*
DiverseDataSizes	**bool**	Are there different size data values?
MaxMessageValues	**ulong**	Maximum number of data values in any message*
MinBufferSize	**ulong**	Minimum allowable buffer size in bytes
MinMessageValues	**ulong**	Minimum number of data values in any message
NumberOfOpCodes	**ulong**	Number of message types
UnBounded	**bool**	Do any messages have unbounded length?
VariableMessageLength	**bool**	Can messages be different lengths?
ZeroLengthMessages	**bool**	Are any messages zero length (have no arguments)?

* these attributes have no inferred value if protocol is unbounded.

5.1.3 Operation Element of Protocol Elements

The term Operation is loosely associated with the analogous concept in RPC systems where the message is invoking an operation on a remote object. In the context of OpenCPI it simply describes one of the messages that is legal to send on a port with this protocol. It has two attributes and some number of argument child elements, which describe data fields in the message.

5.1.3.1 Name Attribute of Operation Elements

This string attribute is a case insensitive name of the operation/message within this protocol. It should be an appropriate identifier for programming languages.

5.1.3.2 Argument Child Element of Operation Elements

This child element indicates a data field in the message payload for the given operation. Its attributes are the same as a **Property** element and describe an argument with: **Name**, **Type**, **StringLength**, **ArrayLength**, **SequenceLength**, etc. If no **argument** elements are present under an Operation element, the operation defines messages with no data fields, referred to as a ***Zero Length Message***. Argument elements are similar to **member** elements in **property** elements whose **Type** attribute is **struct**, but these arguments to an operation do not have to have bounded lengths. Here the **StringLength** attribute is not required for strings, and the **SequenceLength** attribute can be zero indicating no upper bound.

5.2 Protocol Specification (OPS) Examples

This protocol has one message type consisting of 1024 unsigned short values.

```
<Protocol>
  <Operation Name="mess1">
    <Argument Name="val" Type="uShort" ArrayLength="1k">
  </Operation>
</Protocol>
```

5.3 Message Payloads on Data Ports

The message payload for each operation message has a serialized format as a sequence of bytes that, when used in software, are laid out in byte-addressed memory. For example, if the operation element in a protocol contains:

```
<argument name='a1' type='uchar'/>
<argument name='a2' type='ushort' arraylength='2'/>
<argument name='a3' type='ulonglong'/>
```

And the values of this payload are:

```
a1: 1, a2: {0x2345,0x6789}, a3: 0xfedcba9876543210
```

Then the byte sequence (with proper alignment, and encoded little-endian), would be:

Sequence # ►	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Contents (hex)	01	x	45	23	89	67	x	x	10	32	54	76	98	ba	dc	fe
Argument	a1		a2[0]		a2[1]				a3							
Contents	1		0x2345		0x6789				0xfedcba9876543210							

This layout and these values are the same for all types of workers in all (little endian) environments and over all data paths. The **x** values are padding for alignment.

Every numeric type is aligned for its size. Structure types are aligned according to the largest alignment requirement of any of its members. Sequences are aligned according to the alignment requirement of their members and are preceded by a 32-bit unsigned value indicating the number of elements. If the alignment requirement of the sequence is greater than 32 bits, padding is added after the length to align the first element, even if there are zero elements. Padding inserted to achieve alignment may be any value.

The exception to this rule is when the message consists entirely of one sequence of fixed-length elements. In this case, the sequence length is implied by the overall message length at input and output ports and *not* by inserting a 32-bit unsigned value into the message buffer/structure. The message itself is the sequence and the number of elements in the sequence is derived from the length of the message supplied out-of-band by the OpenCPI data-plane.

For related information about message payloads on data ports that is specific to an authoring model, see the development guide for that authoring model.

6 Component Specifications (typically in OCS XML Files)

An OpenCPI ***component*** is a functional abstraction with a specifically defined control and configuration interface based on configuration properties, and zero or more data interfaces (***ports***), each with a defined messaging ***protocol***. An ***OpenCPI Component Specification (OCS)*** file describes both of these aspects of a component, establishing interface requirements for multiple implementations (workers) in any authoring model. Workers are developed based on an OCS.

In the unusual case where there is no expectation of multiple workers implementing the same OCS, the XML for the OCS may be embedded in the worker's OWD, as a **`ComponentSpec`** *element*.

The OCS describes two aspects: (1) the configuration properties of the component (how it is initially and dynamically configured and controlled), and (2) the (data) ports of the component (how it talks to other components). Based on these aspects, all components can be configured and interconnected in an application, regardless of component implementations. An OCS does not contain a behavioral description of the component, but only its interfaces, for use by both implementations and for applications.

A OCS file is the first step in having a component implementation built and ready for use in an application. This file is the basis for all the implementations. An OpenCPI worker is an implementation based on an OCS using a particular authoring model. The worker consists of two things:

1. A separate XML description called the OpenCPI Worker Description (***OWD***) of the particular implementation, indicating the worker's authoring model the worker is based on and the OCS is it implementing
2. The source code in some programming language that does the actual computing function of the implementation, written according to the authoring model.

The OCS XML contains some top-level attributes, configuration aspects and data port aspects. A component specification is contained in the XML element whose type is ***ComponentSpec"*** which should be a top-level element in a file, structured as:

```
<ComponentSpec
    ---attributes---
    >
  ---child elements---
</ComponentSpec>
```

The OCS XML file is called the ***spec file*** for the component, and has a **`-spec.xml`** suffix. The spec files for all components in a library are usually found the **`specs`** sub-directory of the library. When groups of properties or groups of message protocol operations, or message types, are shared between spec files they are placed in separate **`*-prot.xml`** or **`*-prop.xml`** files. This allows for references from multiple spec files. The suffixes and locations of the files are required for the component library

management tools (**`ocpidev`**). The spec files must be exported when applications use components in the library.

The spec files, and if necessary, the property and protocol files, are used by two different processes:

- The implementations, in worker subdirectories, need these files to ensure the implementation matches the specification.
- Applications need these files to correctly *use* the components and connect them to each other. Reference the ***OpenCPI Application Guide*** document.

It is strongly encouraged to use a common spec, and common unit tests, between different implementations of the same functionality defined by an OCS.

6.1 ComponentSpec Top-level Element

Below are the attributes and elements of the **`ComponentSpec`** top-level XML element. **`ComponentSpec`** elements may have **`name`** and **`noControl`** attributes, and may contain **`property`** and **`port`** child elements.

6.1.1 Name Attribute *of the ComponentSpec* Element

The optional **`Name`** attribute of the component specification provides a name that must be unique within its library. The attribute is case insensitive within a library or application. This means two different component specifications cannot differ only in case. When the **`ComponentSpec`** element is the top-level element of a file, the component name attribute is defaulted from the name of the file before any suffixes. Omitting this attribute and using this default is recommended since this eliminates any confusing mismatches between the name of the OCS file and the name of the component in the XML.

6.1.2 NoControl Attribute *of ComponentSpec* Elements

The **`NoControl`** attribute of the component specification is a Boolean attribute that indicates, when true, that components using this specification have no lifecycle/configuration interface at all. This is generally not allowed for application components but is specified for certain infrastructure components.

6.2 `Properties` Element of ComponentSpec Elements

The **`Properties`** element of a component specification has no attributes but consists of a list of **`Property`** child elements. The **`Properties`** element may be in a separate file and referenced using the **`<xi:include href="<file>"/>`** syntax. This is useful when groups of **`Property`** elements are shared among multiple component specifications. However, the most common usage is to have **`Property`** elements directly enumerated under the top level **`ComponentSpec`** element, without using the **`Properties`** element at all.

6.3 `Property` Element of ComponentSpec or Properties Elements

A **`Property`** element describes one configuration property. It occurs as a sub-element of either the **`ComponentSpec`** element or the **`Properties`** element. A **`Property`** element describes the name, data type and accessibility of a configuration property. Its data type can be a scalar type or a structure. Each property can also be an **`array`** and/or a **`sequence`** of its data type. The term **`array`** refers to a fixed number of data values of the specified type. The term **`sequence`** refers to a variable number of data values, up to a specified maximum length. All variable length data types used for properties must be bounded. See the sequences and strings data types for more information.

Property elements as described here may also be present in the OWD for a worker, to specify worker-specific properties beyond what is specified and required by the OCS.

6.3.1 Name Attribute of Property (and Member) Elements

The **`Name`** attribute is the case insensitive name of the property. A set of properties cannot have properties whose names differ only in case. Mixed case property names can be used for readability. When a **`Properties`** element includes other **`Properties`** elements there is still only one flat case-insensitive name space of properties for the component.

6.3.2 Description Attribute/Element of Property (and Member) Elements

The **`Description`** attribute allows the author to add documentation of the property in XML. This text can also be supplied in a child element, which is more convenient if the test is multiple lines. To allow multi-line descriptions to be conveniently indented in an XML file, leading white space is removed from the start of all lines to the extent that it is common to all lines. Currently this description text is not used in OpenCPI, but it is intended to support automation for component data sheets. An example of the white-space removal is:

```
<property name='prop1'>
  <description>
     Hello this is some descriptive text
     Where we want to show
       - multiline features
       - white-space removal
  </description>
</property>
```

would result in the description test being:

```
Hello this is some descriptive text
Where we want to show
  - multiline features
  - white-space removal
```

6.3.3 *Type Attribute of Property (and Member) Elements*

The **Type** attribute specifies the data type of the property. The legal types are listed in the table Data Types for Properties. When the **Type** attribute has the value "**String**", the **StringLength** attribute must also be supplied unless the property is a parameter. This additional attribute indicates the maximum length of the string property values, *excluding* any terminating null character. If no **Type** attribute is present in the **Property** element, the type **ULong** is used as the default.

When the type is **Enum**, the actual values are zero-based **Ulong**, but the named values are indicated by strings found in the **Enums** attribute described below.

The **ArrayLength** attribute is used when the property is a fixed-length one-dimensional array of the indicated type. The **SequenceLength** attribute is used when the property is a variable length sequence of the indicated type. When a property is both an array and a sequence, this means it is a *sequence of arrays*.

When the type is **Struct**, the **Property** element itself has **Member** sub-elements that indicate the types of the members of the **struct** property. The **SequenceLength** and **ArrayLength** attributes may apply to **Struct** properties. **Member** child elements are similar to **Property** elements in that they describe the name and data type information for the member. Note that as of release 2.1, components with properties of type **Struct** are not yet implementable with the HDL authoring model for FPGAs.

All types have a maximum length and properties in general cannot have unbounded length. All properties have a default "empty" value, which is zero for numeric types, false for booleans, empty strings for strings, and zero length sequences for sequences.

6.3.4 *StringLength Attribute of Property Elements*

The **StringLength** attribute is required when the **Type** attribute is **String** unless the property is a parameter. It indicates the maximum length string value, excluding any null termination, that this property can hold. This length is like the value of the **strlen** function in C, or the **length()** member function of a std::string in C++.

6.3.5 *Enums Attribute of Property Elements*

This attribute is required when the **Type** attribute is **enum**, and its value is a comma-separated list of strings naming the enumerated values. The actual values are **Ulong** and are zero-based ordinals based on the position of the names in this list.

6.3.6 *ArrayLength Attribute of Property Elements*

The presence of this attribute indicates that the property values are a fixed length one-dimensional array of the type specified in the **Type** attribute, and that fixed length is indicated in the value of this attribute.

This attribute is a convenience shorthand for using the **ArrayDimensions** attribute described below. E.g. if it is used like this:

```
ArrayLength='5'
```

This is really shorthand for:

```
ArrayDimensions='5'
```

6.3.7 SequenceLength Attribute of Property Elements

The presence of this attribute indicates that the number of property values is a variable, but bounded, sequence of the type specified in the **Type** attribute. The maximum length is indicated in the value of the **SequenceLength** attribute. This property has the specified maximum length, and always contains a current length, up to that limit. *When both **SequenceLength** and **ArrayLength** attributes are present, the meaning is sequence of arrays, not array of sequences.*

6.3.8 ArrayDimensions Attribute of Property Elements

The value of this attribute is a comma-separated list of array dimensions indicating an array whose number of dimensions is the number of values in the list. If this attribute is set, then the **ArrayLength** attribute should not be set. This attribute implies that values are multidimensional arrays of elements whose type is indicated by the **Type** attribute. The ordering of array dimensions is the same as arrays in C and C++, namely that elements in the last dimension are contiguous in memory.

6.4 Accessibility Attributes; How and When Properties Get Their Values

The accessibility attributes of the **`Property`** element specify how and when properties get their values. The boolean attributes that affect property accessibility are: **`parameter`**, **`initial`**, **`writable`**, and **`volatile`**. They all have the default value of **`false`**. When described as being "set", it implies set to **`true`**. The attributes that specify property values in XML are **`default`** or **`value`**. How all these attributes are used is defined below and a summary of the valid combinations is in a table after that.

*6.4.1 **How** Do Properties Get Their Values?*

Property values come from a combination of:

- control software (an ACI application, the **`ocpirun`** utility, or the OAS XML)
- worker software (either as built/compile-time, or set by executing worker code).

Control software can provide the value for a property using any one of these methods, in precedence order (earlier ones override later ones):

- runtime (after start, during execution) setting of values using ACI methods or the **`<delay>`** element in the application's XML
- command line arguments to **`ocpirun`**
- **`PValue`** arguments to the C++/Python application constructor in the ACI
- set in the application XML (OAS)

The above methods can be considered "top-down" property settings, and are all described in the ***OpenCPI Application Development Guide***.

Worker software (and the OCS a worker is based on) can provide values using any of these (bottom-up) methods, in precedence order:

- set in a worker's own runtime code (after start, during execution, overriding any initial value)
- set in component OCS XML **`property`** elements as a *promise* that all workers implementing the spec will have the same *default initial* value if not overridden by control software.
- set in worker OWD XML **`property`** or **`specproperty`** elements (when *not* set in the OCS XML), indicating a worker-specific initial value.
- set in a worker's own initialization code

When the initial value is specified as *not* being settable by control software, the worker itself (the implementation) provides its initial value and subsequent runtime values.

*6.4.2 **When** Do Properties Get Their Values?*

Property values may be set/established at three different ***times***:

- At worker build/compile time.
- At initialization time, just before applications (and workers) are started
- After an application starts, during execution

6.4.3 Property Accessibility Attributes

Attributes in OCS **`property`** elements and OWD **`property`** and **`specproperty`** elements specify which of the setting/getting methods and times are allowed.

The **`parameter`**, **`initial`** and **`writable`** attributes specify where the value comes from (its source) and when it may be set (its time). None or one of them may be set: they are mutually exclusive.

`Parameter`: the value is established in XML and has a constant, compile-time value, either determined in the OCS or determined by the worker's OWD XML. Source is ***worker*** and time is ***compile-time***. No changes are allowed.

`Initial:` the value is established when an application is initialized, before it is started, according to the precedence rules. Source is ***control software***, and time is ***initialization*** (before start). No changes are allowed ***during execution***.

`Writable:` the value is established like **`initial`**, but may also be dynamically set at any time *during* execution. Source of dynamic/runtime values is ***control-software***, and changes are allowed any time ***during execution***: workers will see the value changing during execution.

`<None>`: If none of the **`parameter`**, **`initial`**, or **`writable`** attributes is set, the source is ***worker***, and the time is ***initialization***.

The **`volatile`** attribute specifies whether the worker may change the value *after* the application starts during execution. Control software will see value changes during execution. The **`volatile`** attribute may not be set when **`parameter`** is set.

The **`default`** attribute may specify a default value in the XML, i.e. the value that will be set if control software sets no value. This attribute is only valid when **`parameter`**, **`initial`**, or **`writable`** is set. As one exception to the precedence rules, a *parameter* with a **`default`** value may be overridden by the underlying worker, in its XML, but this override does not change the default value assumed by applications when no value is requested.

For parameters, the **`value`** attribute may be used instead of the **`default`** attribute which indicates a fixed and unchangeable value. I.e. control software can assume this value for all workers.

When the OCS specifies neither **`default`** nor **`value`** attributes, the default is the "empty" value for the data type, as described in the Type Attribute section above.

6.4.4 Reading Properties from the Control-Software Perspective

All properties are considered readable by control software. Parameter property values are read as the constant value set in XML (by OCS or worker XML). The worker is thus not involved in reading. Otherwise the following discussion applies.

If the property is indicated as **`volatile`**, the worker may change the value at any time and the value control-software reads is what the worker provides at that moment, with no caching. Even when the volatile property is specified as **`initial`** or **`writable`**, the value *read* by control software is what the worker provides at that moment and not necessarily what was written previously by control software.

If the property is *not* indicated as **`volatile`**, and it is specified as **`initial`** or **`writable`**, control-software simply reads back what it last wrote, using a cache and not involving the worker. If no boolean accessibility attributes are set, control software reads back the constant value that the worker provided after its own initialization, using a cache in the control-software after the first read.

6.4.5 The Order in which Properties are Written during Startup

This section only applies when properties are specified as i**`nitial`** or **`writable.`** Initial values are written prior to workers being started but after they have been initialized, with no inter-property ordering (of writes) guarantees. To process multiple properties as a group, a worker must wait until the start operation or the first time the worker is run, since that is the only way to be sure all the properties that will be written, have been written. I.e. if some worker startup action depends on a combination of the values of multiple properties, this cannot be done prior to the start operation.

Property write notifications (telling the worker when a property has been written, different in each authoring model) will still be made prior to the start operation, to allow workers to know that a property has been written at the earliest possible time. However, no assumptions on ordering between these early property writes can be made. A worker may keep track of what properties have been written to determine when multiple properties have been written based on write notifications. Otherwise, it should wait until the start operation.

6.4.6 Initial Attribute of Property Elements

See the introduction above on the semantics of this boolean attribute. This attribute precludes setting the value later by control software during execution. It only allows an initial value to be specified from any source but the worker itself.

The **`initial`** attribute setting in an OCS provides the most flexibility to workers implementing the specification, and is the most common property accessibility setting. It allows workers to choose whether to implement the property as compile-time (setting

parameter in the worker XML and perhaps building the worker for multiple compile-time values) or runtime (leaving it unchanged from the OCS, allowing any value). The **initial** setting allows there to be multiple workers that make this choice differently (compile-time/fast/small for a fixed set of values vs. runtime and flexible for all values).

The **initial** setting provides another benefit that makes workers easier to write: the worker does not have to deal with any changes in the property's value after the application starts, allowing implementations to *either*:

- Have the property be configurable to any value at execution startup time
  —or—
- Have the property value be fixed and compiled in to the implementation.

This enables some implementations to be flexible and allow different values at runtime, while other implementations can fix the value at compile time. When the application (control software) specifies a particular value, the worker with a compiled-in value can only be used if the requested initial value matches the compiled-in value. Workers with the non-parameter property can be used with any value. The usage of **initial** in the OCS, and **parameter** in the OWD is discussed further in the section on the OWD XML files.

6.4.7 Writable Attribute of Property Elements

See the introduction above on the semantics of this boolean attribute. It enables setting the value by control software *during* execution.

Setting the **writable** attribute requires that the worker accept and implement dynamic runtime changes in the property's value. This setting should only be used if this is a true requirement of the component since it is more demanding on all workers that implement the specification.

6.4.8 Parameter Attribute of Property Elements

See the introduction above on the semantics of this boolean attribute. It indicates that the property's value is set at compile/build time when source code is processed into a binary artifact to be loaded and executed at run-time.

There are two uses for properties designated as **parameters** using this attribute:

1. A convenience variable for defining other attributes in the same XML like string and sequence lengths or array dimensions, or as the basis for other property values when those values use expressions which use parameter properties as variables.
2. Making this property's value available for all workers as a compile-time constant (possibly based on an expression of other parameter values).

The convenience usage (1: expressions) allows properties defined as parameters to be used in expressions for the value of **stringlength**, **sequencelength**, **arraylength**, **arraydimensions**, **value** or **default** attributes. For the allowable syntax of such expressions see the attribute expressions section just below. An example is when multiple properties are to have the same array dimensions, or to have array dimensions that relate to each other, e.g. one twice as long as the other. An example is:

```
<property name='nbranches' default='14' parameter='true'/>
<property name='tree1' arraylength='nbranches'/>
<property name='tree2' arraylength='nbranches*2 - 1'/>
```

The second usage (2) makes the value for all implementations a compile-time value. If the property has a **value** attribute, that value is the fixed value for the property. If the property has a **default** attribute, that value is the value unless the worker specifies otherwise (see the OWD and **-build.xml** sections).

These **parameter** values are also made available to worker source code during the compilation process, in a way specific to the authoring model and described in the document for the authoring mode.

6.4.9 Expressions in Numeric Attributes

For attributes that take numeric values, such as **StringLength**, **ArrayLength**, **SequenceLength**, and **ArrayDimensions**, the values can be non-negative numeric values, and can be expressions that may use parameter properties as variables in the expression. They are parsed as the type **Ulong**. The full expression syntax is described in detail in Property Value Syntax section. An example is above in the Parameter Attribute section.

6.4.10 Default Attribute of Property Elements

The name of this attribute is **default**. This attribute provides a default value for the property for all implementations and for the initial value precedence rules described above. It is parsed based on the data type specified in the **Type** attribute, and if a numeric type, may be an expression (see the previous section). The format of the string value of this attribute is described in the *Property Value Syntax and Ranges* section.

For **initial** or **writable** properties, it specifies the initial value set in the absence of any other value supplied by control software when the application is started. For **parameter** properties it specifies the compile-time value when a worker specifies no other compile time value.

6.4.11 Value Attribute of Property Elements

The **value** attribute is similar to the **default** attribute described above, except that its usage indicates that the value may not be overridden elsewhere (not by the application,

not by the worker). Its primary use is simply as a convenient constant value that may be reused in other numeric expressions. It may only be used for **`parameter`** properties.

6.5 Port Element of ComponentSpec Elements

The component specification defines ports using the ***Port*** element. It specifies the direction/role of the port (producer or consumer), and the message protocol used at that port. For backward compatibility, **`DataInterfaceSpec`** can be used in place of **`Port`**.

The **`Port`** element has several attributes and one optional child element: the **`Protocol`**. The protocol is usually specified using the **`protocol`** *attribute*, but can also specified inline using the **`protocol`** *element* instead.

6.5.1 Name Attribute of Port Elements

This attribute specifies the name of this port of the component. The value of the **`name`** attribute is a string that is constrained to be valid in various programming languages. It must be unique and is case insensitive within the component specification.

6.5.2 Producer Attribute of Port Elements

This Boolean attribute indicates whether this port has the role of a producer, when true, vs. the default of false for a consumer. *There is no Consumer attribute*. This attribute indicates whether the port acts as a consumer (input) when it is not set at all (which defaults to false), or is explicitly set to false.

All ports are considered input/consumer ports unless this attribute is set to true.

6.5.3 Protocol Attribute of Port Elements

Not be confused with the **`Protocol`** *element,* described in the OPS section, this string attribute names an XML file containing the OPS for the port. The file is expected to contain a **`Protocol`** element at its top level. If the port being described is permissive, meaning it may produce/consume any protocol, then this attribute can be absent. An example of a permissive component is a file writing component that logs any messages from its input, regardless of protocol.

As with all attributes that refer to an XML file, the **`.xml`** suffix is assumed if not present, and the file is sought using the search path for XML files.

When a protocol is not specified, the following ***protocol summary*** attribute values are implied:

Table 5: Protocol Summary Attribute Values when there is no Protocol

Name	Value
DataValueWidth	**8**
DataValueGranularity	**1**
DefaultBufferSize	system default
DiverseDataSizes	**true**
MaxMessageValues	none
MinBufferSize	**0**
MinMessageValues	**0**
NumberOfOpCodes	**256**
UnBounded	**true**
VariableMessageLength	**true**
ZeroLengthMessages	**true**

6.5.4 Optional Attribute of Port Elements

This is the attribute whose name is **optional** which is an optional attribute. This Boolean attribute indicates whether the data port may be left unconnected in an application. The default value of **false** indicates that workers implementing this component require that this port have a connection to some other worker in the application. When true, this port may be left unconnected and all workers implementing this specification must support the case when the port is not connected to anything.

6.6 Component Specification Examples

Here is an example of a component specification that declares one float property that can be set during initialization, but not during operation. It has one output producer port that uses the protocol defined in the **`ushort_1k-proto.xml`** file.

```
<ComponentSpec>
   <Property Name=″size″ Type=″float″ Initial='true'/>
   <Port Name="lvds_tx" Producer="true" Protocol='ushort_1K-proto'/>
</ComponentSpec>
```

7 Property Value Syntax and Ranges

This section describes how textual property values are formatted to be appropriate for their data types. Property values occur in the **`default`** or **`value`** attributes of property elements described above. This syntax is also used when property values are specified in the worker XML build files and unit test files described below. The type names in the sections below are those acceptable to the **`Type`** attribute of the **`property`** element in the OCS file.

Remember that attribute values in XML syntax are in single or double quotes. The syntax described here is used inside these quotes. To have quotes inside attribute values, the other type of quotes is used to delimit the attribute value. In either case, inside the quoted attribute value, the **`&`** and **<** characters must be escaped using the official XML notions: **`&`** for **`&`**, **`<`** for **<**. If *both* types of quotes must be in an attribute value, then the official XML escape sequences can be used: **`&quot;`** for double-quote, and **`&apos;`** for single quote.

Property values are also used when running applications. That usage is described in the *Application Development Guide*, but the format is as described here.

7.1 Values of Unsigned Integer Types: uchar, ushort, ulong, ulonglong

These numeric values can be entered in decimal, octal with leading zero, or hexadecimal with leading 0x. The limits are the typical ranges for unsigned 8, 16, 32, or 64 bits respectively.

The **`uchar`** type can also be entered as a value in single quotes, which indicates that the value is an ASCII character, with backslash escaping as defined in the C language. The syntax inside the single quotes is as described for the **`char`** type below.

7.2 Values of Signed Integer Types: short, long, longlong

These numeric values can be entered in decimal, octal with a leading zero, or hexadecimal with a leading 0x, with an optional leading minus sign to indicate negative values. The limits are the typical ranges for signed 16, 32, or 64 bits respectively.

7.3 Values of the Type: char

This type is meant to represent a character, i.e. a unit of a string. In software it is represented as a signed char type, with the typical numeric range for a signed 8-bit value. The format of a value of this type is simply the character itself, with the typical set of escapes for non-printing characters, as specified in the C programming language and IDL:

`\n \t \v \b \r \f \a \\ \? \' \"`

A series of 1-3 octal digits can follow the backslash, and a series of 1-2 hex digits can follow **`\x`**.

OpenCPI adds two additional escape sequences as a convenience for entering signed and unsigned decimal values of type char. The sequence **`\d`** may be followed by an optional minus sign (**`-`**) and one to three decimal digits, limited to the range of -128 to 127. The sequence **`\u`** can be followed by one to three decimal digits, limited to the range of 0 to 255.

These escapes can also be used in a string value. Due to the requirements of the arrays and sequence values, the backslash can also escape commas and braces, i.e.:

```
\,    \{    \}
```

7.4 Values of the Types: float and double

These values represent the IEEE floating point types with their defined ranges and precision. The values are those acceptable to the ISO C99 **`strtof`** and **`strtod`** functions respectively.

7.5 Values of the Type: bool

These values represent the Boolean type, which is logical true or false. The values can be case insensitive: **`true`** or **`1`** for a true value, and **`false`** or **`0`** for a false value.

7.6 Values of the Type: string

These values are simply character strings, but also can include all the escape sequences defined for the **`char`** type above. Due to the requirements of the arrays and sequence values, the backslash can also escape commas and braces (**`\,`** and **`\{`** and **`\}`**). Double quotes may be used to surround strings, which protects commas, braces, and leading or trailing white space. To be interpreted this way, the first character must be a double quote. Two double quotes can represent an empty string.

7.7 Values in a Sequence Type

Values in a sequence type are comma-separated values. When the type of a sequence is **`char`** or **`string`**, backslash escapes are used when the data values include commas.

7.8 Values in an Array Type

When a value is a one-dimensional array, the format is the same as the sequence, with the number of values limited by the size of the array. If the number of comma-separated values is less than the size of the array, the remaining values are filled with the ***null*** value appropriate for the type. Null values are zero for all numeric types and the type ***char***. Null values for string types are empty strings.

7.9 Values in Multidimensional Types

For multidimensional arrays or sequences of arrays, the curly brace characters (**{** and **}**) are used to define a sub-value. For example, a sequence of 3 elements, each consisting of arrays of length 3 of type char, would be:

```
{a,b,c},{x,y,z},{p,q,r}
```

This would also work for a 3 x 3 array of type char. Braces are used when an item is itself an array, recursively.

7.10 Values in Struct Types

Struct values are a comma-separated sequence of members, where each member is a member name followed by white space, followed by the member value. A struct value can be "sparse", i.e. only have values for some members. If the struct type was:

```
struct { long el[2][3]; string m2; char c; };  // C pseudo code
```

A valid value would be:

```
el {{1,3,2},{4,5,6}}, c x
```

This struct value would not have a value for the **m2** member.

7.11 Expressions in Property Values

Both numeric and string typed scalar values can be specified using an expression syntax and operator precedence from the C language, where any parameter property with a value can be accessed as a variable. All C expression operators can be used except the comma operator, assignments or self-increments/decrements. The conditional operator using **?** and **:** *is* supported. Expressions can be used as elements of arrays or sequences, or as structure member values.

For example, if the **nbranches** property was a parameter, a valid expression elsewhere might be:

```
nbranches == 0x123 ? 2k-1 : 0177
```

7.11.1 Numeric Values

The numeric constant syntax is typical C language syntax (integer and floating point), with the following additions:

- Integers with explicit radix after a leading **0** can use **0t** for base 10 and **0b** for base 2, in addition to the normally used **0x** for base 16 and no letter for base 8. All these prefixes can be applied to the fraction and exponent for floating-point syntax.
- Integers can use a letter suffix of **K**, **M**, or **G**, upper or lower case, indicating 2^10, 2^20 or 2^30 respectively. E.g. **2k-1** is 2047.

- All arithmetic is done using a numeric data type exceeding the range and precision of **`uint64_t`**, **`int64_t`** and **`double`**, and then assigned to the actual data type of the property whose value is being specified.
- The ** binary operator (pow) from the python and FORTRAN languages is also supported.

When the value of the expression is assigned to the property value or numeric property attribute (e.g. **`ArrayLength`**), it is range checked for validity. Boolean properties are set to true if the value is non-zero. Fractions are discarded when assigning values to integer types.

7.11.2 String Values

Within expressions, string constants (using double quotes as in C) and string-valued parameter properties can be used. All comparison operators are case sensitive and result in boolean numeric values (0 or 1). All operators requiring boolean values (**`!`**, **`||`**, **`&&`**, **`?:`**) use the length of the string (zero being false, otherwise true). The **`+`** operator concatenates strings. There is no implicit or explicit conversion between string values and numeric values. E.g. if **`sparam`** is a string-typed parameter with the value **`abc`**, then this expression has the numeric value of 1:

```
sparam == "abc"
```

This expression would have the string value **`xyz_abc`**:

```
"xyz_" + sparam
```

7.11.3 Expressions Used for String Values

To distinguish when a string value is an expression, as opposed to a string that might look like an expression, the string value must have a special prefix of **`\:`** (backslash followed by colon). So if a property (or sequence/array element or structure member) is a string type, any value assigned to it is considered not to be an expression by default. To make the string value interpreted as an expression, use the **`\:`** prefix.

So if **`sparam`** is a parameter with value **`abc`**, then specifying the property's string value

```
sparam+"xyz"
```

would simply define that exact string (with plus sign and double quotes included) but if the string value was:

```
\:sparam+"xyz"
```

then it would be interpreted as an expression, and the actual string value would be:

```
abcxyz
```

8 Worker Descriptions in OWD XML Files

Each worker directory contains an XML file describing the worker and referencing the spec file typically located in the component library's **specs** sub-directory. This XML file is referred to as the ***OpenCPI Worker Description (OWD***). The generic aspects (common across authoring models) of these worker description files are described in this section. The OWD files also have aspects specific to different authoring models, which are described in the respective documents for each authoring model. Some authoring models allow multiple workers to be implemented in one worker directory. In these cases multiple OWDs may be in a single worker directory.

The worker description file adds non-default implementation information to the basic information found in the spec file. Each authoring model defines what the worker-specific information might be. An example would be the width of an FPGA data path for a port in the HDL model. All OWDs have as the top-level XML element an ***XYZ*Worker** element, where ***XYZ*** is the authoring model of the worker.

If the worker has no non-default behavior, there is no need for an edited OWD. The framework (**ocpidev**) will generate a default one. This default OWD simply contains a reference to the spec file and specifies the authoring model and language. For example, if an RCC worker based on the spec file **search-spec.xml** only had default implementation attributes, the entire OWD file would be:

```
<RccWorker spec='search-spec'/>
```

This section describes aspects common to the OWDs for all authoring models. Further OWD content specific to authoring models is described in their respective documents.

The top level element must refer to an OCS by either:

- indicating an OCS file by using the **spec** attribute (preferred)
- containing a **ComponentSpec** child element, thus not shared with any other OWD (rare)

Below is an example OWD for an HDL worker, that might be found in **fastcore.hdl/fastcore.xml**. The **fastcore** implementation of the **core-spec** specification is using the HDL authoring model. It references the component specification found in the ***core-spec.xml*** file, probably in the **specs** directory of the component library containing this worker:

```
<HdlWorker Spec='core-spec' Language='vhdl'
   ---other attributes---
                          >
   ---other child elements---
</HdlWorker>
```

OWD files contain the information (metadata) that the OpenCPI build process and runtime environment need to know about the worker that is not in its source code.

A second, optional, XML file in a worker's directory is its build file, named, e.g., **`fastcore-build.xml`**. These files are described in the Build Configuration File section below.

8.1 XML Attributes of the Top-level XYZWorker Element in the OWD

8.1.1 Name Attribute of the XYZWorker Element.

The **`Name`** attribute defaults to the name of the OWD XML file itself without the directory or extension and is normally omitted. The **`Name`** attribute of the component implementation is constrained to be an identifier in several contexts It is sometimes called the worker name or implementation name.

Worker names may include both upper and lower case for stylistic or programming language purposes. The OpenCPI framework identifies workers in a case insensitive manner. There should not be two workers using the same authoring model in the same package namespace whose names differ in case.

The name of the implementation may be the same as the name of the OCS. It is not required to have a unique name for the OWD different from the spec, unless there are multiple implementations of one OCS *that use the same authoring model*. I.e. OWD names are implicitly scoped by authoring model.

8.1.2 Spec Attribute of the XYZWorker Element

This string attribute specifies the name of the file for the OCS for this worker. The **`ocpidev build`** command automatically place the **`specs`** subdirectory (in the component library's directory), into the search path when these worker description files are processed. The spec files need only be referenced by their name and not any directory or pathname. If the spec file is outside the component library, it can be a relative or absolute pathname. The .**`xml`** suffix is assumed and not needed.

8.1.3 Language Attribute of the XYZWorker Element

This string attribute specifies the source code language used in this worker. The valid languages depends on the authoring model, and for each model there is a default language. Some authoring models have only one valid language in which case this attribute is not required.

8.1.4 Version Attribute of the XYZWorker Elements

This attribute has type **`UChar`** with a default value of zero if unspecified.

The language interface between workers and their environment (control and data) evolves, usually backward-compatibly. When there is an incompatible change, the worker interface version is incremented, allowing the tools and runtime environment to continue to support workers written to the older interface. Actual interface changes are described in the authoring model documents (e.g. RCC, HDL etc.).

The worker interfaces were updated incompatibly in OpenCPI release 1.5, to version 2, which means that workers written to the new version 2 interface (highly recommended)

must explicitly set this attribute to 2. While most changes are specific to authoring models, a significant change to end-of-file indications and zero-length-message handling was made and is described in the End-of-File (EOF) Indications section. New workers should be written to version 2, and older workers that undergo significant revisions should too. When the **`ocpidev`** command creates new workers, it sets this attribute to the latest version.

8.1.5 ControlOperations Attribute of the XYZWorker Element

This attribute contains a comma-separated list of strings identifying the implemented control operations. For operations that are mandatory for the authoring model, they are assumed. The default implies a minimal implementation that only implements those operations required by the authoring model. The control operations are listed in the LifeCycle Control section. Control operations that are required by the authoring model do not need to be mentioned. When only mandatory operations are implemented, this attribute need not be specified.

8.1.6 OnlyPlatforms Attribute of the XYZWorker Element

This attribute can contain a list of platforms that are the only ones this worker should ever be built for. This should be used to indicate a permanent characteristic of the worker (source code): that it is known to *not* be buildable or workable except for these. A (rare) example would be if the code had platform-specific code for a certain set of platforms.

To temporarily and frequently change the list of platforms to build for, it is preferable to use this same attribute in the worker's build configuration file in Build Configuration File since that simply indicates a temporary choice rather than a permanent characteristic of the worker's source code.

8.1.7 ExcludePlatforms Attribute of the XYZWorker Element

This attribute can contain a list of platforms that this worker should never be built for. This should be used to indicate a permanent characteristic of the worker: that it is known to not be buildable or workable for these platforms.

To temporarily and frequently change the list of platforms to build for, use this same attribute in the worker's build configuration file in Build Configuration Files.

8.1.8 OnlyTargets Attribute of the XYZWorker Element

This attribute can contain a list of targets that are the only ones this worker should ever be built for. This should be used to indicate a permanent characteristic of the worker: that it is known to not be buildable or workable except for these. A (rare) example would be if the code had target-specific code for a certain set of platforms.

To temporarily and frequently change the list of targets to build for, use this same attribute in the worker's build configuration file in Build Configuration Files.

8.1.9 ExcludeTargets Attribute of the XYZWorker Element

This attribute can contain a list of targets that this worker should never be built for. This should be used to indicate a permanent characteristic of the worker: that it is known to not be buildable or workable for these.

To temporarily and frequently change the list of targets to build for, use this same attribute in the worker's build configuration file in Build Configuration Files.

8.1.10 SourceFiles Attribute of the XYZWorker Element

This attribute contains a list of subsidiary files that should also be compiled and linked together with the worker. For FPGA code, the files are guaranteed to be compiled *before* the worker source file is compiled so that VHDL entities in these files can be directly instantiated in the worker source code file without redundant component declarations. The list is space or comma separated.

8.1.11 Libraries Attribute of the XYZWorker Element

This attribute contains a list of primitive libraries that the worker requires when it is built. They represent libraries that are subject to the search rules for primitive libraries, which is, in order:

- in the same project
- exported from projects the current project explicitly depends on
- exported by the ocpi.core project

If a library name is qualified by a project, then only the library in the indicated project is used. E.g. for the library named **`mylib`** in a project whose package id is **`myorg.myproj`**, by using:

```
libraries='myorg.myproj.mylib otherlib'
```

the **`mylib`** must be found in the **`myorg.myproj`** project, while the library **`otherlib`** will be found by searching as described above.

RCC primitive libraries are not yet supported.

8.1.12 IncludeDirs Attribute of the XYZWorker Element

This attribute contains a list of directories (via relative or absolute pathnames) that will be searched when including programming language files during building.

8.1.13 ComponentLibraries Attribute of the XYZWorker Element

This attribute contains a list a component libraries to be searched when building this worker. Component libraries are needed when this worker refers to others that are not in the same library. Most workers do not depend on any other components or

component libraries, but one example is when the spec for a worker is in a different library.

8.1.14 Endian Attribute of the XYZWorker Element

This attribute specifies the endian behavior of the worker code. When workers are built, the build process may be run in three different modes to create three different types of binaries:

- Little endian
- Big endian
- Dynamic endian based on an input supplied at runtime

The third way, dynamic, is generally not relevant for software since compilers only generate code for a specific assumed endianness. But it is relevant to the FPGA build process to support FPGA bitstreams that can operate in both modes. This OWD attribute specifies how the worker's code will work in these three build scenarios ,as specified by the ***ocpi_endian*** parameter which is present for all workers of all types. This attribute is rarely used. Most workers are either unaffected by endianness or are written for little endian.

Table 6 – Worker Endian Attribute Settings

Endian Attribute	Description
Neutral	The worker code is unaffected by endian parameter settings and is correct regardless of the setting. This is the default value, and is generally correct for software workers.
Little	The worker code is unaffected by endian parameter settings and is will only operate correctly in a little endian mode.
Big	The worker code is unaffected by endian parameter settings and will only operate correctly in a little endian mode.
Static	The worker code will respect the endian parameter when set to “little” or “big” and the resulting binaries will operate correctly according to the compile-time parameter setting.
Dynamic	Worker code will respect all three values of the endian parameter. If the parameter is **`little`** or **`big`**, the resulting binaries work in the requested mode. If the parameter is **`dynamic`**, the resulting binaries will work in an endian mode specified by an input signal or variable as specified in the authoring model. Not all authoring models have this option.

8.2 Property and SpecProperty Child Elements in the OWD

Properties specified in the component's OCS indicate the external configuration interface for all implementations of the same spec. Properties specified in the worker's OWD define additional worker-specific properties, beyond those in the OCS. The description of OCS **`property`** elements in Property Elements in OCS applies when defining worker-specific properties in the OWD, with some additional attributes described below that are valid *only* in OWD properties.

The OWD can also provide *additional* attributes to properties already defined in the OCS. E.g. the OWD might make an OCS property **`writable`** that was not writable in the component spec. The accessibility *added* would result in the worker supporting a superset of what was required (promised) by the component spec.

While a component spec can only contain **`Property`** elements, a worker description can contain both **`Property`** and **`SpecProperty`** elements. The **`Property`** elements in the OWD introduce new worker-specific properties unrelated to those defined in the OCS. The **`SpecProperty`** elements add worker-specific *attributes* to the properties *already defined* in the OCS.

There are a number of OWD-only attributes in the next section. They generally apply to both **`Property`** and **`SpecProperty`** elements in the OWD. Property elements in the OWD support all the attributes supported by Property elements in the OCS, plus the OWD-only attributes below. These OWD property attributes are divided into three categories:

- Attributes valid in both Property elements and SpecProperty elements
- Attributes only valid in Property elements, not valid in SpecProperty elements
- The subset of OCS property attributes that are allowed in SpecProperty elements.

8.3 Attributes Only Allowed in OWD Property and SpecProperty Elements

These attributes only exist in OWDs and never in OCSs and are allowed in both **`Property`** and **`SpecProperty`** elements.

`SpecProperty` elements can add OWD-only attributes to a property and augment the accessibility of the property, but cannot change the data type of the property.

8.3.1 Name Attribute for OWD Property or SpecProperty Elements

The **`Name`** attribute is the case insensitive name of the property. The **`Name`** attribute is used in **`SpecProperty`** elements to indicate which OCS property is being referenced. In the **`Property`** elements it indicates the name of the new worker-specific property, which must not be the same as any **`Property`** element in the OCS.

8.3.2 ReadSync and WriteSync for OWD Property/SpecProperty Elements

These Boolean attributes, defaulting to false, are used to indicate the properties that require the worker to be notified when they are read or written by control-software. The baseline behavior is that property accesses are directly made to property values in the worker, with no specific synchronization or notification implied. In ths case the worker accesses these values as local memory locations. When these attributes are **`true`**, the worker is notified when the access is made by control software.

The exact mechanism used for such worker notification is specific to the authoring model and is described in those documents. Some authoring models may not implement or require this attribute, but where needed, this definition is valid.

8.3.3 ReadError/WriteError Attributes for OWD Property/SpecProperty Elements

These Boolean attributes , default is **`false`**, indicate properties that may return errors when read, **`ReadError`**, or written, **`WriteError.`** If a worker does not return errors and always succeeds when property values are read or written, then leaving these values false allows control-software to avoid any error checking. In some models and systems, error checking can carry significant overhead. Most workers simply accept new values using the default of **`false`** for this attribute.

The exact mechanism used for such worker error reporting is specific to the authoring model and is described in those documents.

8.3.4 Readback Attribute for OWD Property/SpecProperty Elements

This attribute specifies that the worker supports reading back values written to non-volatile properties that are either **`parameter`**, **`initial`** or **`writable`**. It is never necessary for normal operation since control-software always caches such values when written, and provides those cached values when control-software reads the value.

This option allows such values to be read back directly from the worker when control-software specifically requests an uncached read access, or when property values are being read directly from the worker and bypassing control-software altogether. An example of this is the **`ocpihdl`** utility for HDL/FPGA platforms (described in the HDL Development Guide) when a raw loaded bitstream is being examined directly.

8.4 Attributes Allowed in OWD Property, but not SpecProperty, Elements

These attributes are allowed in OWD Property elements, but are not allowed in SpecProperty elements.

8.4.1 Padding Attribute for OWD Property but not SpecProperty Elements

Padding properties (with their **`padding`** attribute being **`true)`**, are those that only exist to force subsequent properties to proper alignment or offsets. No other accessibility attributes may be used with padding properties. The use case for this is when properties need to match a register map or data structure and may require exact offsets for each property. These padding properties are not accessible.

8.5 Attributes Allowed in OWD SpecProperty Elements

In addition to OWD-only attributes described above, these are the attributes available in **`SpecProperty`** elements that can be used to *add* accessibility or values beyond what is in the OCS. I.e. the OCS specifies what is guaranteed for applications that use this component. These attributes allow the worker to provide a superset of what is promised in the OCS.

`SpecProperty` elements can add OWD-only attributes to a property and augment the accessibility of the property, but cannot change the data type of the property.

8.5.1 Parameter Attribute for OWD SpecProperty Elements

This boolean attribute in a **`SpecProperty`** element is used to convert a property specified as **`initial`** in the OCS, to be a **`parameter`** property for this worker. The **`initial`** setting in the OCS allows the worker author to decide whether to implement it as a runtime property (i.e. **`initial`**) set at initialization time, or a parameter set at compile time. This attribute set to true makes the latter choice.

When a worker has **`initial`** (in the OCS) properties that are converted to be **`parameter`** in the OWD, it means that the worker must be built for specific values of such properties. This has three implications:

- An application can only use the worker if the worker is built for a property value that matches what is requested as an “initial” value by the application.
- Binary component libraries may have multiple binary built artifacts for the same worker, but with different combinations of parameter values.
- The worker author must decide which combinations of parameter values to build, in order to make alternative settings of such parameters available.

This parameter feature allows workers to have compile time optimizations for certain parameter values, and also allow a single worker source code module to be optimized for different values at compile time. A different worker that accepts runtime values can still be used when no workers with compile-time values have been built with the value requested by the application.

Parameter property values are applied to the build process as expected: e.g. by preprocessor symbol definitions and initialized **`const`** values for software, generics for VHDL, and parameters for Verilog.

8.5.2 Writable Attribute for OWD SpecProperty Elements

This boolean attribute in a **`SpecProperty`** element is used to convert a property specified as **`initial`** or not writable at all in the OCS, to be a **`writable`** property for

this worker. Thus it adds the possibility of dynamically changing the property value even if that is not what the spec promises.

8.5.3 Default Attribute for OWD SpecProperty Elements

The name of this attribute is **`default`**. In the SpecProperty element, this attribute is only valid for **`parameter`** properties (as set in the OCS) and when a value is not already specified in the OCS (via **`default`** or **`value`** attributes). It might be used when the worker author provides this default for the purposes of unit testing and then have build configurations where other values may be specified. See section Build Configuration Files for build configurations.

8.5.4 Value Attribute for OWD SpecProperty Elements

In the SpecProperty element, this attribute is only valid for **`parameter`** properties (as set in the OCS) that are defined as parameters in the OCS, but do not specify a fixed value there.

An example would be where a property is defined in the OCS as a parameter (and possibly a default value) with the expectation that workers would provide a worker-specific fixed value. This can be considered annotating the worker so that applications can get/read the static value associated with the worker. The OCS is essentially saying: all workers that implement this spec will supply a static value to the application.

8.6 Built-in Parameters of All Workers

OpenCPI automatically adds several parameter properties to all workers. The values of these parameters are usually set during the build process in various ways. Some of these parameters are set to values by the build process and are not intended to be set manually at all. Others may be set or overridden manually.

Each authoring model may also specify additional built-in parameters for all workers using that authoring model. The built-in parameters that apply to all authoring models are described below.

All built-in parameters use the **`ocpi_`** prefix to avoid collisions with component and worker developers' property names.

8.6.1 The ocpi_debug Built-in Parameter Property

This boolean parameter property indicates whether a debug build is being done. The default value is **`false`**. Setting this value to **`true`** indicates to worker source code that any debugging instrumentation or behavior should be enabled, at a potential cost of some resource usage and performance. This built-in parameter is always available, and should be used in worker code to enable things like extra logging or statistics keeping.

Setting this parameter to true will also in some cases enable some introspection or instrumentation capabilities of code that is in the OpenCPI infrastructure or is generated code used implicitly by the worker.

Properties can be defined with a debug attribute value of **`true`**, which indicates that those properties should only be present when the worker is built with this **`ocpi_debug`** parameter set to **`true`**.

These features allow debug behavior and debug properties to be permanently in the worker's source code and OWD while only being enabled as required.

8.6.2 The ocpi_endian Built-in Parameter Property

[This feature is preliminary/untested, but mentioned here as a roadmap item]

This parameter property indicates to worker code which endian mode is being used when the worker is being compiled. Its type is an enumeration of three values:

- ***little:*** The build is intended to generate binaries for little endian systems
- ***big:*** The build is intended to generate binaries for big endian systems.
- ***both:*** The build is intended to generate binaries that can be used in either little or big endian mode, selected at initialization time in the runtime environment.

Software authoring models normally set this mode implicitly as compilers generate binary code for a specific endianness based on the processor being targeted, e.g. little for x86 and ARM, big for PPC.

However some authoring models, such as HDL, can support all three compilation modes. The ability of a worker's code to support various endian modes is specified in the worker's **`endian`** attribute at the top level of its OWD.

8.6.3 Built-in Properties Associated with Ports

OpenCPI provides several properties that are associated with each port, all of which have the **`ocpi_`** prefix and have the port name at the end of the name. Most built-in properties do not get listed when properties are dumped by ocpirun unless the **`--hidden`** option is specified to also dump these “hidden” properties. One useful one, for output ports, provides the runtime buffer size in bytes: **`ocpi_buffer_size_<port>`**.

8.7 Port Elements of XYZWorker Elements

Ports are how workers communicate with each other. They define message-oriented, data-plane communication. Each authoring model defines how workers receive/consume and send/produce messages to or from other workers. This is independent of whether workers are collocated in the same container or executing elsewhere.

Each authoring model may have attributes and elements of this **`Port`** element specific to that authoring model, but there are a number of aspects common to all worker descriptions that are described here. A **`Port`** element in a worker description matches the **`Port`** element in the component spec by name, and adds worker-specific information about how the worker implements the port.

Some authoring models may use different names for **`port`** elements in order to imply additional information, but the attributes described below apply in any case.

8.7.1 Name Attribute of Port Elements

This string attribute is required and must match one of the names of the port elements in the component spec. It normally indicates for which component port the worker is providing additional implementation-specific information. In some special cases, a worker may use this element to define additional ports beyond what is in the spec, but this behavior would be specific to an authoring model.

8.7.2 Protocol Summary Override Attributes of Port Elements

The OCS file on which the worker is based defines protocols for ports, and various protocol summary attributes are derived from the protocol. These attributes may be overridden where the protocol is defined (see Protocol Summary Attributes), but they may also be overridden for a particular worker as attributes of the **`port`** element.

Care must be taken when doing such overrides since they may render a worker incompatible with other workers that implement the same spec. This could make unit testing of all such workers infeasible or difficult.

9 Worker Build Configuration XML Files

While the OWD file described above holds fundamental information about the worker's source code, and information *required* to build and run the worker, the optional ***build configuration file*** describes the various ways it should be currently built.

Thus the build configuration file narrows down the perhaps large number of ways the worker *might* be built (for different combinations of parameter values or targeting different platforms), to what will actually get built. When it is not present, the worker is built using the default values of parameter properties, and the platforms being targeted is limited by the fundamental limitations on platforms expressed in the top-level attributes of the OWD.

For portable workers with no parameters, the build file is not recommended or needed.

The use cases for this file include:

- Specifying useful combinations of compile-time parameters for unit testing
- Specifying known currently required combinations of compile-time parameters for applications
- Limiting platforms to build for, for temporary convenience rather than an expression of “what platforms are viable”, which is already specified in the OWD.

The name of this optional file is **`<worker>-build.xml`**, and the top level element is **`build`**. The top-level attributes controlling/limiting the build targets and platforms in the OWD are also valid here. Limitations in the OWD override those in this file since the OWD platform constraints are considered fundamental, while those in this file are considered for convenience to further limit what platforms or targets get built.

These platform/target constraint attributes that are defined for the OWD and are also valid in the build file are: **`OnlyPlatforms`**, **`ExcludePlatforms`**, **`OnlyTargets`**, **`ExcludeTargets`**. There are no other top-level attributes for this file.

9.1 Build Configurations

We use the term ***build configuration*** to be a set of parameter property values to use when building the worker. Each build configuration has an ***ID***. When the build configuration file does not exist, there is a single build configuration, with an ID of zero, with default values of all parameters (from OCS or OWD).

The **`target-*`** directories mentioned above are used to separate the files that result from building for different targets (e.g. **`centos7`** vs. xilinx**`13_4`**, or [Xilinx] **`virtex6`** vs [Altera] **`stratix5`**). A different **`target-*`** subdirectory is in fact created for each build configuration. If the ID is zero, the name of the directory is **`target-<platform>`**. If it is not zero, then the directory name is **`target-<id>-<platform>`**.

There are two valid child elements in the top level **`build`** XML element, which together determine the set of build configurations for the worker:

- ***parameter*** to specify parameter values for all configurations
- ***configuration*** to specify a build configuration.

If there are none of either element, a configuration with ID zero is generated using the default values of all parameter properties. The same thing happens when there is no file at all.

If there are **`parameter`** elements, but no **`configuration`** elements, then configurations are generated for all combinations of values mentioned in all the **`parameter`** elements, using default values for unmentioned parameter properties. Since **`parameter`** elements can provide multiple values (see below), this means that configurations are automatically generated for the cross-product of all parameter values. E.g. if property A is specified with values 1 and 2, and property B is specified with values 10, 11, and 12, and property C is unmentioned but having a default value of 77, then these build configurations will be generated:

ID	A	B	C
0	1	10	77
1	1	11	77
2	1	12	77
3	2	10	77
4	2	11	77
5	2	12	77

The file in this example would be:

```
<build>
  <parameter name='a' values='1,2'/>
  <parameter name='b' values='10,11,12'/>
</build>
```

When a **configuration** element is specified *with* an **id** attribute, all parameter properties unmentioned in the **configuration** element take on default values. Using the above example we might have:

```
<build>
  <configuration id='0'>
    <parameter name='a' value='2'/>
    <parameter name='b' value='12'/>
  </configuration>
</build>
```

Which would result in A being 2, B being 12 and C being 77.

When a configuration element is specified *without* an **id** attribute, configurations are generated that have the all the combinations mentioned in top-level **parameter** elements as in the above example, but with the values for parameter properties mentioned *inside* the configuration element taking those specific values. For example:

```
<build>
  <parameter name='a' values='1,2'/>
  <parameter name='b' values='10,11,12'/>
  <configuration>
    <parameter name='c' value='78'/>
  </configuration>
</build>
```

This would create 6 build configurations (like the table above), but with property C having the value 78 rather than its default of 77.

9.2 Parameter Elements in the <worker>-build.xml File

This element specifies one or more values for a parameter property. The **`name`** attribute specifies which property, and must match the name of the parameter property in the OWD or OCS (case insensitive). The values for the parameter can be specified using one of these attributes:

- The **`value`** attribute can specify the single value.
- The **`values`** attribute can specify a comma-separated list of values.
- The **`valueFile`** attribute can specify a value in a file.
- The **`valuesFile`** attribute can specify multiple values in a file.

When the **`parameter`** element is at the top level of the ***`<worker>`*`-build`** file, it is specifying values common to all configurations that *do not* have **`id`** attributes. If the same **`parameter`** element is mentioned as a child element of a **`configuration`** element, it overrides any top level value(s) for that parameter in that configuration. An example ***`<worker>`*`-build`** file is:

```
<build>
  <parameter name='debug' value='true'/>
  <configuration id='1'>
    <parameter name='mode' value='lownoise'/>
    <parameter name='taps' valueFile='taps.txt'/>
  </configuration>
</build>
```

Since there is an **`id`** attribute present for the configurations, it would use the default value of the debug parameter, not the value specified in the **`parameter`** element (which in this case would be ignored).

9.3 Configuration Elements in the <worker>-build.xml File

The **`configuration`** element has an optional numeric **`id`** attribute which will appear as the build configuration suffix in the name of the target directory for that configuration. If the **`id`** attribute is zero, no suffix is added. The configuration element also has **`parameter`** subelements indicating the specific parameter values for that configuration.

When there is no **`id`** attribute, the globally defined values for parameters are used to generate build configurations.

10 Component Libraries

OpenCPI components are developed in libraries. OpenCPI component-based applications are defined as a composition of components, and the components are developed and built in component libraries.

A component library is a directory that contains:

- Component specifications, OCSs, OPSs in a **`specs`** subdirectory.
- Component implementations (workers), each in its own subdirectory.
- Component tests in **`*.test`** subdirectories.
- An XML file, named ***`<library-name>`*`.xml`**, for some library options.
- When built, a subdirectory call **`lib/`** is created and populated with links to the built binaries and metadata files required to use components in the library, from outside the library.

The results of building a library that are made visible to users of the library are in the **`lib/`** subdirectory. That directory is removed when the library is cleaned using:

```
ocpidev clean
```

The binary results of building a library is a collection of heterogeneous built workers for various technologies, along with various XML metadata files.

Component libraries are developed inside OpenCPI projects, as described in the Developing OpenCPI Components in Projects section. This defines a larger directory structure containing a variety of OpenCPI assets, including component libraries and applications.

The basic directory structure of a component library is shows in the figure below.

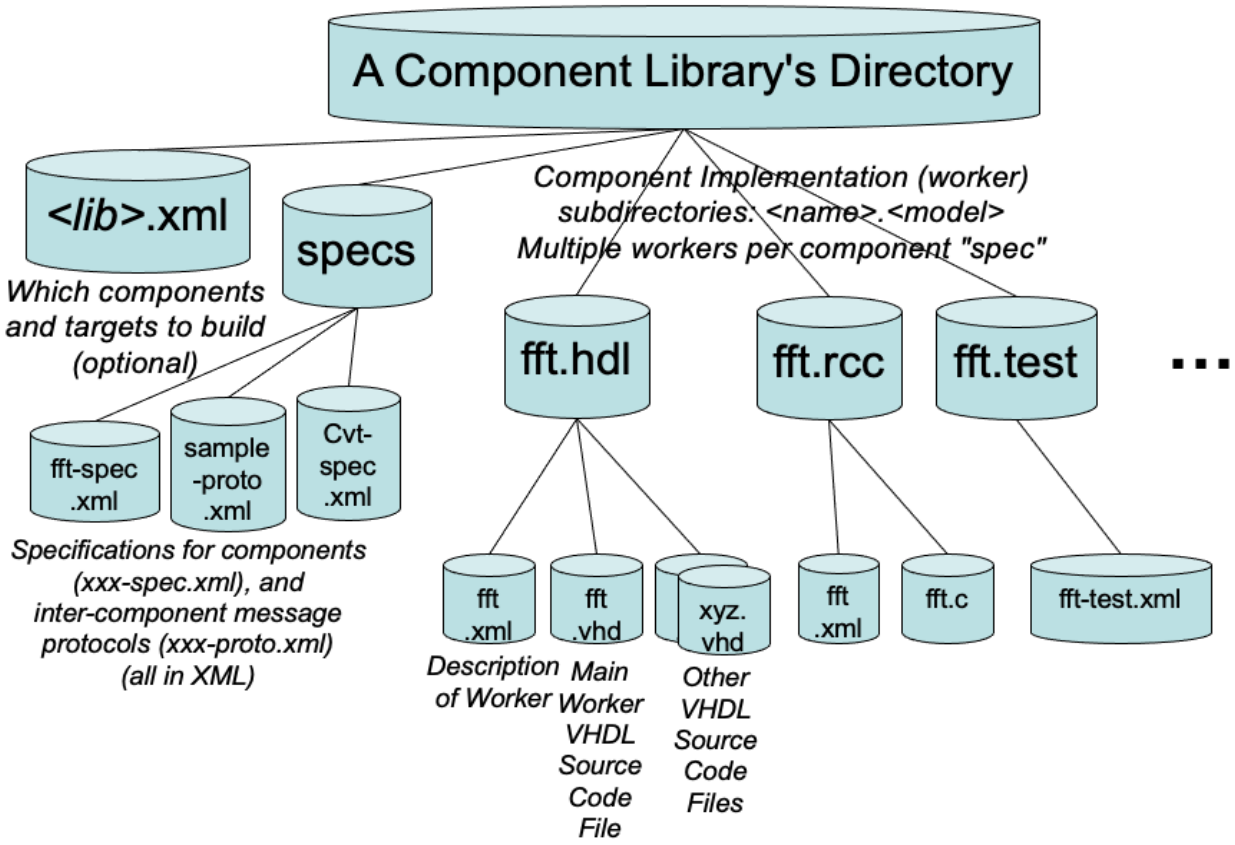

Figure 1: Component Library Directory Structure

The file hierarchy of a component library **`mycl`** is outlined below. The library contains a **`search`** component with RCC and HDL implementations, and a **`transform`** component with only an HDL implementation:

```
mycl/mycl.xml
    /specs/search-spec.xml
          /transform-spec.xml
    /search.rcc/search.xml
               /search.c      (RCC C source file)
    /search.hdl/search.xml
               /search.vhd      (HDL VHDL source file)
    /transform.hdl/transform.xml
                  /transform.vhd
```

10.1 The Component Library XML File

The ***<libname>*.xml** file in the top-level directory of the component library is generated automatically by the **ocpidev** command. The file contents look something like:

```
<library />
```

One optional attribute in this file is **Workers**, which is a list of which worker subdirectories to be built in this component library. When the **Workers** attribute is not set at all, it indicates that all workers found in the component library that contain workers should be built. For example:

```
Workers='fft.rcc fft.hdl fft-for-xilinx.hdl fir.rcc'
```

There are two reasons to set this attribute at all rather than using the default.

1. If you want to temporarily avoid building some workers in the library, you can set this attribute to only the ones you want to build, so any others are ignored.
2. If you want to specify the order in which the workers are built, you can set this attribute to the workers you want to build, in the order you want them to be built. There are two situations where the order of building workers is important. First, if a worker is a proxy (see the ***RCC Development Guide***) for another, the "slave" of the slave must be built before the proxy. Second, if a subdevice supports another device, the supported device must be built before the subdevice (see the ***Platform Development Guide***).

The **ExcludeWorkers** attribute can also be used to exclude certain workers from being built when the **Workers** attribute is unset.

In order to avoid name space collisions when using multiple component libraries, there is also a **PackageName** attribute that specifies what namespace should be used for the specs and workers in this library. See the Package IDs section for complete information about package IDs.

For all software (not HDL) authoring models, the default RCC platform, if none is specified, is the machine and operating environment of the machine doing the building. Otherwise build targets and platforms can be specified in arguments to the **ocpidev build** command.

Software targets use the software platform names. Examples are **centos7**, **ubuntu18_04**, **ubuntu20_04**, **ubuntu16_04**, **macos10_13**, **xilinx19_2_aarch32** (for Xilinx linux for Zynq).

HDL targets typically contain an architecturally compatible part family (e.g. **virtex6** or **stratix4**). See the ***HDL Development Guide*** for more details.

If all subdirectories containing workers should indeed be built, and the package name is the default, then no attributes need be set.

Creating a new component library **mycl** is accomplished by using the **ocpidev** tool, using the **create library** command. This tool is described in the ocpidev section below.

```
ocpidev create library <library-name>
```

If present, the ***<library-name>*.xml** file in the component library's directory contains settings that should apply to all the workers and tests in the library. This file is read **ocpidev**.

The table below lists the attributes settable in this file. They are also usable at the project level in the project's **Project.xml** file which has a similar purpose.

*Table 7: Attributes in **<library-name>.xml** files*

Attributes Name in <*libname*>.xml	**Description**
Libraries	A list of *primitive* libraries built elsewhere. It is searched in this project and in projects this project depends on.
OnlyTargets **OnlyPlatforms**	A list of the only targets/platforms for which this worker should be built
ExcludeTargets **ExcludePlatforms**	A list of targets/platforms for which this worker should NOT be built
XmlIncludeDirs	A list of directories elsewhere for searching for XML files.
IncludeDirs	A list of directories elsewhere for searching for files indicated by “include” directives in worker source files such as those in C, C++ or Verilog.
ComponentLibraries	A list of component libraries to search when the worker refers to another worker. The need for this is specific to the authoring model (e.g. proxies for RCC workers, or emulators for HDL device workers).

11 Developing Workers

This section describes the aspects of the worker development process that is common across all types of workers and authoring models. Previous sections above described the files involved in worker development, including:

OCS XML files: component specifications, usually in **`../specs`**

OPS XML files: protocol specifications, usually in **`../specs`**

OWD XML files: worker descriptions

Worker Build Configuration files: XML files for specifying build configurations

Worker Source files: Programming language source code for the worker.

This section describes the development process using these files.

The worker development details for each authoring model are described in the document for each authoring model. Some authoring models (e.g. RCC) support creating a single binary file ***artifact*** that implements multiple workers. However, usually a single worker implementation is in its own subdirectory, which when compiled, results in a single binary file *for each build configuration* (combination target and parameter values).

11.1 Creating Workers

A worker is created, in a component library, using the **`ocpidev`** tool, with the command:

`ocpidev create worker <name> [-S <spec>] [-L <language>]`

The authoring model is inferred from the ***`<name>`***, using the suffix of the name as the authoring model. The optional ***`<spec>`*** argument specifies the name of the OCS file, normally without any directory indicated (expected to be in the **`../specs`** directory). If ***`<spec>`*** is not specified, it is assumed to be ***`<name>`*****`-spec.xml`** in the library's **`specs`** directory. If the new worker will embed the component spec in its own OWD, then the ***`<spec>`*** argument can be set to **`none`**. While rare, some specialized workers will be the only implementation of a spec and there is no need for separate spec file.

The ***`<language>`*** is one of the programming languages allowed for the authoring model (e.g. **`c`**, or **`c++`** for RCC, **`vhdl`** or **`verilog`** for HDL). If not mentioned, the default language for the authoring model will be used.

The **`ocpidev`** command is usually executed in the directory where the new workers's directory will be created. Other options are fully described in the **`ocpidev`** section below.

The command:

```
ocpidev delete worker <name>
```

will remove the worker, and is roughly equivalent to (after asking for confirmation)

```
rm -r -f <name>
```

When a worker is created, the worker's XML and source language files are initially generated by **ocpidev.** Several internal files (not for user editing) are also placed in a **gen/** subdirectory of the worker's directory. When source files are compiled, the resulting binary files are placed in subdirectories named: **target-<target>**, where ***<target>*** is the platform the compilation is targeting. Cleaning a worker directory (via **ocpidev clean**) simply removes the **gen** and all **target-*** subdirectories. In almost all cases, files in the **gen** subdirectory should be considered read-only and not edited.

Creating a new worker creates initial versions of two files in the worker's directory:

3. the OWD XML file
4. the skeleton source file

These are the files the developer can edit as necessary. Although frequently the OWD XML file does not need any further editing.

The initial source file is termed the ***skeleton***, and is named

```
<worker-name>.<source-suffix>
  e.g.
xyz.c
```

It can be compiled, but has empty logic. The skeletal code allows the worker to be test-built even before any editing is done. Each authoring model describes how and where this skeleton source file should be edited and "filled out" with the logic that makes it perform its intended function. A copy of this initial skeleton file is always put in the **gen** subdirectory, with the name:

```
gen/<worker-name>-skel.<source-suffix>
  e.g.
gen/xyz-skel.c
```

This copy can always be examined to see what the skeleton was originally, before any editing. It can also be useful to examine, after the OCS, OPS, or OWD has changed, in case changes are require in the source file. After the initial skeleton (*not* the copy in **gen/**) is edited by the developer, it will never be overwritten or removed by **ocpidev clean** or any other command.

11.2 Editing Workers

Often it is useful to break the worker's logic into supporting code modules in other source files. Those files must be created manually and added to the **SourceFiles** attribute in the OWD. In some authoring models and languages, the files listed in the **SourceFiles** attribute must be in dependency order, with lower level modules/files

preceding those that depend on them. The primary source file is always considered the top level module for the worker and is put at the end of the list automatically.

Some changes to the OWD, OCS, and OPS files can result in changes that require corresponding changes in the worker's primary source file, which was initially generated as a skeleton. Since the developer has likely manually edited the primary source file, it is not touched when such changes are required. If it is clear to the developer when these changes are required, they can do it before any building. However, it is likely that the required changes will create build/compilation errors.

Examples would be such things are renaming ports or properties, adding or subtracting access attributes to properties, converting properties to parameters, etc.

When any changes are made, the skeleton is regenerated properly during the next build, and the result places in the file:

`gen/<worker>-skel.<source-suffix>`

The newly generated skeleton can be used as a guide when changes occur that might require changes in the edited worker source code.

The authoring model documents also list common changes to the OCS, OPS, and OWD files, and the corresponding changes required in the source file. An example is for VHDL workers using the HDL authoring model. The skeleton lists the lower level primitive libraries that the worker depends on. If such a primitive library is added to the worker XML file, the library also needs to be added to the list of libraries in the skeleton.

12 The Worker Source Files

The worker source files must be written according to the authoring model. As a starting point the **ocpidev** tool creates an empty skeleton of a worker implementation that will compile, build and execute, doing nothing.

12.1 How Parameter Value Settings Appear in Source Code

Parameter values are compile-time constants in all authoring models. The precise way that parameters and their values appear in source code varies by authoring model and programming language. There are standard data types for properties (see Data Types for Properties), and constants are defined that specify these values. Examples are:

- C and C++: A **static const** variable is defined which is initialized to the parameter value. The name of the variable is the property name prefixed with **OCPI_PARAM_**.
- VHDL: A generic with the parameter's name is set to the value.
- Verilog: A parameter with the parameter's name is set to the value.

12.2 Building Workers

Workers are normally built as part of building a whole component library, or as part of building a whole project. To simply compile new code and locate syntax errors, a worker can be built in the worker's directory, by typing **ocpidev build**.

The target or platform of a worker build is specified in several ways. For software authoring models the default target is always the local development machine on which the building is taking place. For other authoring models, there is no default. On the **ocpidev** command line, targets and platforms use options of the form:

```
--<model>-target <target>
--<model>-platform <platform>
```

Some examples are:

```
--rcc-platform centos7
--rcc-platform xilinx13_4
--hdl-target virtex6
--hdl-platform zed
```

Default target and platform settings can also be set in project's **Project.xml** file or the library's ***<library-name>*.xml** file. In this case the syntax is setting XML attributes of the form:

```
<Model>Target(s)
```

Some examples are:

```
HdlTargets='zynq spartan6'
HdlPlatforms='zed ml605'
RccPlatforms='centos7 xiinx13_4'
```

Workers can be built for multiple targets with one command. This can be useful to check whether the source code is acceptable to all the different compilers.

The worker build process has make-style dependencies such that rebuilds will only happen if any dependent files are changed, including the OCS/OPS/OWD XML files.

13 Unit Testing of Workers

OpenCPI supports unit testing where a ***<component>*.test** directory in a component library is created to hold a test suite for all the workers in the library that implement the same spec (OCS). The workers that are tested could be written to different authoring models or languages or simply be alternative source code implementations written to the same authoring model.

E.g. if a library contained **fft.hdl** and **fft.rcc** and **fft_xilinx_dsp.hdl** workers that all implemented the **fft-spec.xml** OCS file in the library's **specs** directory, a single **fft.test** directory would be created to hold a test suite that tested them all. Each ***<component>*.test** directory is associated with a single OCS,and has a test suite description XML file, called ***<component>*-test.xml**. The test XML file described below in Unit Test Description XML File.

The OpenCPI unit test framework manages multiple dimensions of worker testing, with automation to minimize test design and preparation efforts. The dimensions are:

- Test cases (individual parameterized tests, possibly using different runtime values)
- Platforms (HDL hardware and simulation platforms, RCC/OCL Platforms)
- Workers (different source code implementations, different models)
- Worker build configurations (compiled in vs. runtime settable property values)

The unit test framework allows complex test scenarios while providing layered complexity to keep simple test cases very simple to define and execute. Test inputs and outputs can be pre-prepared data files (i.e. test vectors), or be developer-provided scripts for input data generation and output data verification.

Unit testing proceeds in five phases, although it is most common to group them into two steps: build (phases 1-2) and run (phases 3-5). The phases are:

1. ***Generate*** — generate testing artifacts after finding the spec and the workers to test
2. ***Build*** — building HDL bitstream/executable artifacts for testing (if any HDL workers are found to be tested)
3. ***Prepare*** — examine available *built* workers and available runtime platforms, and create execution scripts to execute all feasible tests.
4. ***Execute*** — execute tests for all workers, configurations, test cases and platforms
5. ***Verify*** — verify results from the execution of test cases on workers and platforms

In most cases, **`ocpidev build`** is used to perform the generate and build phases, while **`ocpidev run`**, performs the ***prepare***, ***execute***, and ***verify*** phases.

The rest of this section will describe:

- the terminology used in this testing framework
- the phases of unit testing in detail
- what is required and possible in the test description XML file: **`<component>-test.xml`**
- how to support data generation and verification
- how to execute test cases and verify their results

A test suite, in a **`<component>.test`** directory, can be created and initially populated using this command:

```
ocpidev create test <component>
```

This will create the directory the one necessary file: **`<component>-test.xml`**.

13.1 Unit Test Concepts and Terminology

Here are the terms used in the OpenCPI unit test automation framework:

Test suite – as embodied in the ***`<component>.test`*** directory, is a suite of test cases for testing all workers implementing a spec across all available platforms for which the workers have been built

Test matrix —the virtual multidimensional space of testing, across:
Workers — the different source code or authoring model implementations
Worker configurations — different parameter value sets for worker builds
Initial property values — runtime property value configurations
Platforms — possible runtime environments
User-defined test cases — with property values, inputs and outputs

Test case – a parameterized test
Using a defined set of inputs or generation scripts
Using a defined set of outputs or verification scripts
Using a defined matrix of property values

Test subcase – a very specific test
Defined by and generated from a test case
Using a specific worker build configuration
Using a specific set of property values
Not bound to a specific platform or artifact

Generator — script to create input data files for ports, or property value files
Called for a subcase, with all property values supplied. Fixed input files can be provided that do not require generation scripts.

Verifier — script to verify test output data produced by output ports
Called for a subcase, supplied with each port's output data file, with all property values supplied, both initial and final (volatile). Fixed “golden” output files can be used in place of verification scripts.

Viewer — script to view the results of a subcase execution. (e.g. plot).

Default Test Case — the case that is automatically created when none are specified
Tests all parameter combinations as derived from all worker parameter/build configurations or all workers
Developer can supply runtime property settings with multiple values for each, resulting in the cross-product of subcases
One generation script (input) and one verification script (output), per port, parameterized by subcase property values

13.2 The Phases of the Unit Test Process

The ***generate(1)*** phase performs the following tasks automatically without any developer involvement other than to prepare the unit test XML file:

- Discovers the OCS associated with this test directory.
- Discovers workers in the same library that implement that OCS.
- Discovers the build configurations (parameter values) for each worker.
- Derives a baseline for test cases based on all the build configurations that have been used on any worker.
- Derives the actual tests appropriate for all parameter combinations vs. the actual worker build configurations they apply to.
- Generates XML applications (OAS files) that perform unit tests on all workers
- Generates HDL test assemblies (subdirectories and OHAD files) that can be built for HDL platforms (hardware and simulation).
- Creates a report in the **`gen/cases.txt`** file listing the parameter values used in all test cases and subcases.

Although there are many options described below, the default generated unit test applications feed specified input data to input ports of the worker being tested and cause EOF to be send to input ports after the input data is sent.

Generated test applications are considered finished when either:

- The component being tested has a single output port and an EOF is received from that port.
- The component being tested has multiple output ports and an EOF is received from the first one.
- The port designated as the **`finishPort`** and an EOF is received from that port.
- The worker itself is designated as the “finish” worker for the application and it enters the finished state.
- A specified time duration for the test has been exceeded.

After the test completes, the output data from all output ports as well as the final values of all properties are available to the verification process for assessment.

All the above tasks in the ***generate(1)*** phase are done “off-line”, without regard to:

- which worker artifacts are built

- which platforms are available on which to execute tests
- the availability of any build-related tools (compilers or FPGA synthesis tools or simulators).

The ***build(2)*** phase is used for testing HDL workers, and builds the *generated* HDL test assemblies for whichever platforms (including simulators) are specified. When building for hardware HDL platforms, this phase takes the longest. There must be tools available to build the specified platforms, but this phase does not require those platforms be available for execution. This phase does nothing when only RCC workers are available. Running **`ocpidev build`** on a unit test suite performs both the generate phase and the build phase.

The ***prepare(3)*** phase does the necessary work to prepare to actually execute test cases and perform associated verification. In this phase the unit test framework automatically does:

- Discovery of available execution platforms, local and remote (reachable via network)
- Discovery of available built artifacts that can be executed on available platforms (including generated HDL test assemblies generated in the ***generate(1)*** phase)
- Generation of test scripts to perform all feasible tests on all available platforms.

After preparation, the developer invokes the ***execute(4)*** and ***verify(5)*** phases. These phases can be sequential (all executions followed by all verifications), or interleaved (each test subcase is executed and verified before executing the next one).

During execution and verification, there are various filtering capabilities to subset which tests are run, which platforms should be tested, and whether a test case failure should immediately stop the testing process.

13.3 Unit Test Description XML File

The ***<component>*-test.xml** file specifies test cases and the defaults that apply to all test cases. As with all OpenCPI XML files, element names and attribute names are case insensitive. The top-level element in the file is **<tests>**, with the possible child elements being:

<input> to define an input file or generator script usable by any test case

<output> to define an output file or verifier script usable by any test case

<property> to define property values for all test cases

<case> to define a non-default test case when needed

If no **<case>** element is defined, the default test case is used. This is a common situation since the default test matrix is based on the parameters that workers are built with, and the available workers that implement this component spec. Here is an example file using the default test case for a component:

```
<tests>
  <input port='in' script='generate.py 16'/>
  <output port='out' script='verify.py 16384 16' view='view.sh'/>
  <property name='phs_inc' values='-4096'/>
  <property name='enable' values='0,1'/>
</tests>
```

It specifies that the default test case should be used (no **<case>** elements), the "**generate.py 16**" command should be issued to generate test data for port **in**, the "**verify.py 16284 16**" command should be issued to verify output data from port **out**, the **phs_inc** property should always be tested set to **-4096**, and the **enable** property should be tested with values **0** and **1**. All scripts are run per subcase and have access to the parameter properties as well as the runtime properties of the subcase being tested.

Several attributes described below refer to scripts that will be executed by the unit test framework. In all cases, scripts must properly return process/shell exit status, with zero indicating success and non-zero indicating failure. This is true regardless of the language used in the script.

13.3.1 Attributes for the Top-level <Tests> Element

The valid attributes for the top-level **tests** element apply to all test cases and are **spec**, **UseHdlFileIO**, **ExcludeWorkers**, **OnlyWorkers**, **ExcludePlatforms**, **OnlyPlatforms**, **DoneWorkerIsUUT**, **TimeOut**, and **Duration**. All are optional and are specified in special situations.

*13.3.1.1 Spec Attribute of the Top-level **<Tests>** Element*

Normally the spec (OCS) for all the workers being tested is inferred from the name of the **<component>.test** directory, and found in the file:

../specs/<component>-spec.xml

When this is not the case, this **spec** attribute can specify the name of the spec file for this test suite, much like the same attribute can be used in a worker's OWD.

*13.3.1.2 UseHdlFileIO Attribute of the Top-level **<Tests>** Element*

This boolean attribute applies only when HDL workers are being tested on simulation platforms. When true, it indicates that file I/O between the worker being tested and the input and output test files is done in the simulator using VHDL/Verilog file operations directly. When false (the default), the file I/O is being done by file reading and writing RCC workers running *outside* the simulator, with the data flowing in and out of the simulator. Both settings can be useful, but the **true** setting generally results in faster simulation times since less logic is being simulated for this file I/O.

*13.3.1.3 ExcludeWorkers Attribute of the Top-level **<Tests>** Element*

This string attribute specifies a list of comma-separated workers (e.g. **fft.hdl**) that should *not* be tested, even if they implement the spec of this test suite.

*13.3.1.4 OnlyWorkers Attribute of the Top-level **<Tests>** Element*

This string attribute specifies a list of comma-separated workers that should be the *only* ones tested. Any others found to implement the same spec will be ignored.

*13.3.1.5 ExcludePlatforms Attribute of the Top-level **<Tests>** Element*

This string attribute specifies a comma-separated list of platforms that should *not* be tested. Wildcard patterns may be used for any name. E.g. "***sim**" would exclude any platform that ended with the letters **sim**. Any other available platforms that have built artifacts will be used.

*13.3.1.6 OnlyPlatforms Attribute of the Top-level **<Tests>** Element*

This string attribute specifies a comma-separated list of platforms that should be the *only* ones tested. Wildcard patterns may be used for any name. E.g. "***sim**" would test only platforms that ended with the letters **sim**. Any other available platforms will be ignored.

*13.3.1.7 Duration Attribute of the Top-level **<Tests>** Element*

Normally unit test cases execute until the test subcase application is ***done***, which is usually when an EOF is asserted on the first or only output port of the worker being tested. See the EOF Assertion section.

When this default EOF behavior is not viable or desirable (e.g. when the workers being tested have no output ports), the **`duration`** attribute can be set to an amount of time the application should run before being considered successfully ***done***.

The duration value is in seconds, and represents wall clock time for the execution. After this amount of clock time, the execution is stopped, and the execution is considered successful and any outputs or final property values are provided for verification.

This attribute is separate from the **`timeout`** attribute, which indicates when the execution should be considered ***failed***. Only one of these attributes may be set.

*13.3.1.8 Timeout Attribute of the Top-level **<Tests>** Element*

The **`timeout`** attribute indicates an amount of wall clock time in seconds after which the execution of a test subcase is considered a failure. Setting this value prevents an execution from hanging and preventing completion (but failure) of the subcase. This is especially useful for non-interactive scripted regression testing. Only one of the **`duration`** or **`timeout`** attributes may be set. This attribute has a default of 3600 (one hour).

*13.3.1.9 FinishPort Attribute of the Top-level **<Tests>** Element*

The **`finishPort`** attribute specifies which of the component's output ports is used to indicate that the test case is finished. The default behavior of the unit test framework is to designate the **`file_write`** component connected to the *first* output port listed in a component's spec as the ***done worker*** in all the unit test applications. Setting **`finishPort`** will change the ***done worker*** to the **`file_write`** connected to the designated output port. The value of the **`finishPort`** attribute is the name of the output port. This top-level setting becomes the default for all test cases, but this setting can be overridden for any test case.

```
<tests finishPort='out3'>
```

*13.3.1.10 FinishedWorkerIsUUT Attribute of the Top-level **<Tests>** Element*

The **`finishedWorkerIsUUT`** boolean attribute indicates that, as a default for all test cases, the worker being tested (the “unit under test” or UUT) is the ***finished worker*** in unit test applications. The ***finished worker*** is the worker which, when finished, indicates that the application is finished. The default is that the **`file_write`** component connected to the first output port is the ***finished worker***. If the component being tested has no output ports, then the default is in fact as if this attribute is set to true. I.e., this attribute is only needed when the component *does* have output ports, but

for some reason they should not be used to indicate when the unit test applications are finished. This global setting can be overridden for any test case.

```
<tests finishedWorkerIsUUT='true'>
```

13.3.2 Input Element of Top-level <Tests> Element

An **<input>** element as a direct child of the top-level **tests** element specifies a source of input data that can be used by test cases. It is not specific to a test case but may be used by any test case for any input port. Its allowable attributes are: **name**, **port**, **file**, **script**, **messageSize**, **messagesInFile**, **suppressEOF**, and **stressorMode**.

13.3.2.1 Name Attribute of the Input Element

This optional string attribute specifies the name of this input source, so it can be referenced by test cases that use it, by name. If it applies to all cases, it doesn't need a name. If it applies only to a specific **port**, the **port** attribute can be set, which is more common. One of **name** and **port** must be specified.

13.3.2.2 Port Attribute of the Input Element

This optional string attribute specifies the name of the port that this input source will always apply to. It there is only one input source for a port, it will be used for all cases. One of **name** and **port** must be specified.

13.3.2.3 Script Attribute of the Input Element

This string attribute indicates a command to execute to produce data. When data is generated for a subcase and for a port, this command will be issued. The attribute value is not just the name of a file to execute, but of a command, so it can have a command name followed by some command arguments. When the command is executed in order to produce data, it will be appended with the name of the file to be written into; i.e. the script's job is to write into the file whose name is at the end of the command. Thus if the value of this attribute was:

```
echo hello >
```

then the source of data would always be a line of text containing **hello** since the actual command executed would be:

```
echo hello > <output-file-supplied-by-unit-test-framework>
```

The way these scripts become more useful is that all parameter and initial runtime property values are supplied to the script as environment variables. Thus this script is parameterized by these values for the subcase being generated. Accessing environment values is easy for the scripts, whether they are written as shell scripts, python, or C. When a script is executed for a subcase (and for a port), the value of each parameter and runtime property is the value of an environment variable named:

OCPI_TEST_*<prop>*

So, if the property's name was **myprop**:

In C or C++, the value (as a string) would be: **getenv("OCPI_TEST_myprop")**

In python, it would be: **os.environ.get("OCPI_TEST_myprop")**

In bash/shell, it would be: **$OCPI_TEST_myprop**

Only parameter properties or runtime properties that are ***initial*** or ***writable*** are present. Using scripts based on these values normally means one script can be applied to all test cases.

The command is executed by the shell in the ***<component>*.test** directory, and must have execute permissions.

13.3.2.4 File Attribute of the Input Element

This string attribute specifies the name of a file to be used as the source of data. It is not affected by any property values and is thus a "constant". This is useful if the same input data should be used for a port for all test cases, or if the file is not easily generated by a script, but is used for some test cases.

13.3.2.5 MessageSize Attribute of the Input Element

This positive integer attribute specifies the size of messages to be supplied to the port of the worker under test when this data source is being used. Since data flowing between ports always consists of messages, this determines their size. The data from this input source is split into messages of this size, in bytes. This attribute is the same as the property of the **file_read** worker.

13.3.2.6 MessagesInFile Attribute of the Input Element

This boolean attribute indicates that the data produced by this input source has message boundaries and opcodes embedded in the data. Each message in the input file is preceded by a header consisting of two 32 bit unsigned integers (little endian), with the first being the length of the message in bytes, and the second being the opcode (with only the low-order 8 bits used). This attribute is the same as the property of the **file_read** or **file_write** workers.

13.3.2.7 TestOptional Attribute of the Input Element

This boolean attribute indicates that an input is to be unconnected, to test its optionality. The default is that all optional ports will be connected to test components. Setting **testOptional** to true will force the port to be left unconnected. It can also be used in **case** elements.

13.3.2.8 SuppressEOF Attribute of the Input Element

This boolean attribute indicates no EOF indication should be provided to the input port after all the data (messages) has been provided as input. This attribute is the same as the property of the **`file_read`** worker and is only valid for inputs. The default behavior is to provide such EOF indications.

13.3.2.9 StressorMode Attribute of the Input Element

This attribute only applies when testing HDL workers and is only valid for input ports. The generated test benches for HDL workers include a ***stressor*** function at each input port which varies the range of input behavior to provide coverage of what the worker might see at its port in applications. This optional enumeration attribute specifies the stressor behavior at this port. The term ***metadata*** is used here to indicate the signals indicating message boundaries and whether the boundaries are separate from or coincident with, the first or last data word of a message. These signals are **`som`**, **`eom`**, and **`valid`**.

There are four modes available: **`bypass`**, **`throttle`**, **`metadata`**, and **`full`**. The meanings are:

`bypass` — all messages are passed through to the worker under test without change or delay.

`throttle` — the metadata of all received messages is passed through to the worker under test without change, but data within the message will be withheld at random intervals to create intra-message data starvation.

`metadata` — the ***stressor*** at this port will cycle through a pattern of all valid metadata combinations, but the data will not be throttled.

`full` — the ***stressor*** will cycle through a pattern of all valid metadata combinations, data within the message will be throttled at random intervals to create data starvation, and idle cycles will be inserted between messages.

The default mode is **`bypass`**.

*13.3.3 Output Element of **`<Tests>`** Top-level Element*

An **`output`** element as a direct child of the top-level **`tests`** element specifies how the output data from a port of the worker being tested may be verified for correctness. The valid attributes are: **`name`**, **`port`**, **`file`**, **`script`**, **`stopOnEOF`**, **`view`** and **`disableBackpressure`**. It may 1) be applied to all test cases, 2) be used as a default for test case that do not mention an **`output`** element for a port, or 3) be referred to by name by some test cases.

It is very similar to the **`input`** element:

- The **`name`** attribute allows this element to be referred to in test cases.
- The **`port`** attribute specifies that this element should be used for a particular port.
- The **`file`** attribute specifies an existing file to compare the output data to for correctness.
- The **`script`** attribute specifies a command that takes a file name as the data to *verify*.

13.3.3.1 Script Attribute of the Output Element

This attribute is similar to the script attribute of **`input`** and **`property`** elements. The major difference is that there are multiple arguments appended to the command instead of one. The first is an input file that contains the output of the given output port as a result of executing the worker in a subcase. After that first file name argument there are file name arguments for each *input* port of the component that contain the input data supplied to that port, in the order the ports are declared in the OCS. This allows the script to not only access the resulting output data from an output port, but also access the data supplied to each input port (if needed for the verification).

For example, if the component had input ports **`in1`** and **`in2`**, and an output port name **`out`**, and a script attribute value of ***`<command>`***, the actual command executed would be:

`<command> <output-from-port-out> <input-to-in1> <input-to-in2>`

A second important difference for the output script vs. an input script is that the final values of writable and volatile properties are available in the environment in addition to the initial values of all other properties. For generated properties (those with a **`script`** attribute in its **`property`** element), the name of the generated property file is placed in the environment variable named **`OCPI_TESTFILE_`*`<property-name>`***, while the final value is still in the **`OCPI_TEST_`*`<property-name>`*** environment variable.

The name of the test case is in the **`OCPI_TESTCASE`** environment variable and the name (which is numeric) of the subcase is in the **`OCPI_TESTSUBCASE`** environment variable. E.g. if the subcase being run was **`case43.03`**, the case name is **`case43`** and the subcase name is **`03`**.

As with all other scripts, a process/shell exit status of zero indicates success, while a non-zero exit status indicates failure. The script may write other informational messages about the failure to ***stderr*** which will be logged. The script should not write simple success and failure (PASS/FAIL) messages since the unit test framework does that already, using green/red colors for PASS/FAIL, based on the exit status.

13.3.3.2 View Attribute of the Output Element

This optional string attribute operates similar to the **`script`** attribute, but has a different purpose. It provides a convenient way for the developer to ask for a "view" of the data for the port. Taking all the same arguments as the verification script (in the **`script`** attribute) , it is expected to present the data is some useful way during test development, typically in some viewing or plotting window.

13.3.3.3 TestOptional Attribute of the Output Element

This boolean attribute indicates if an output is to be unconnected, to test its optionality. The default is that all optional ports will be connected to test components. Setting **`testOptional`** to true will cause the port to be unconnected during the test case.

13.3.3.4 StopOnEOF Attribute of the Output Element

This boolean attribute indicates that reception of an EOF indication should stop the recording of output data from an output port. The default value is true (which is unusual for a boolean attribute). This attribute is the same as the property of the **`file_write`** worker and is only valid for outputs. The reception of EOF indications on the first or only output port is used to indicate the end of execution of a test subcase unless the **`duration`** attribute is set for the case (or as a top-level default for all cases).

13.3.3.5 DisableBackpressure Attribute of the Output Element

This boolean attribute only applies to output ports of HDL workers under test and specifies the ***backpressure*** (i.e. flow control) behavior the worker will see at this output port during testing. Setting **`disableBackpressure`** to true will disable the backpressure applied to this port. When false (the default) the test bench will apply random backpressure to this port.

*13.3.4 Property Element of **`<Tests>`** Top-level Element*

This element specifies the default set of values for a property for all test cases, unless overridden in particular test cases. When multiple values are specified, the implication is that subcases should be generated that test each of the specified values.

For parameter properties, where the potential set of test values is normally derived from the values used to build the workers being tested, the values specified in this element act as a filter or subset of those values, since no tests can be performed for parameter values that are not used in any worker's build configuration.

The allowable attributes for the **`property`** element are: **`name`** (required), **`test`**, **`value`**, **`values`**, **`valueFile`**, **`valuesFile`**, **`generate`**, **`add`**, **`only`**, and **`exclude`**. Exactly one of the **`value*`** attributes must be specified. The textual syntax for property values is used, as described in Property Value Syntax.

13.3.4.1 Name Attribute of the Property Element

This required string attribute identifies a property defined in the OCS of the test suite or in the OWD for a worker being tested. The values specified in other attributes are applied to this specified property during testing (except when the **`test`** attribute is true – see below).

13.3.4.2 Value Attribute of the Property Element

This attribute specifies a single value to be tested.

13.3.4.3 Values Attribute of the Property Element

This attribute specifies a comma-separated sequence of values to be tested.

13.3.4.4 ValueFile Attribute of the Property Element

This attribute specifies the name of a file containing a single value to be tested. Multiple lines in the file are considered elements of a sequence or array value. In fact, new-lines in the file are equivalent to commas, so anywhere in the value syntax that commas occur, they can be newlines in the file.

Property value syntax is described in the Property Value Syntax section.

13.3.4.5 ValuesFile Attribute of the Property Element

This attribute specifies the name of a file containing multiple values to be tested. Multiple lines in the file are considered separate values to be tested. Multiple values can also be specified on a single line in the syntax of a sequence of values of the type of the property. E.g., if the type is Ulong, the **`ValuesFile`** file could contain a single line of **`1,2,3,4`** or four lines containing the four values.

13.3.4.6 Generate Attribute of the Property Element

This attribute specifies a command to execute to create a file containing a value to be tested. An argument is added to the specified command that is the name of the file to be written. All parameter and initial runtime property values are available to the script as environment variables. This feature is convenient when a property value depends on others in a complex way. Note that expressions can be used in the value attributes of **`property`** elements, so scripts are not necessary to perform simple arithmetic based on other parameters.

The generated file should be in the format used for the **`valueFile`** attribute, as described above in the ValueFile Attribute section.

13.3.4.7 Test Attribute of the Property Element

This optional boolean attribute, when true, indicates that this property is a ***test*** property that is *not* a property of the workers being tested. It is a property whose value is available to all input generation, output verification, output viewing and property generation scripts. Its name must be different than all property names in the OCS or in any of the workers' OWDs.

Values assigned to a test property are used to generate other test cases not defined simply by the values of worker properties. When this attribute is true, other data type attributes used in an OCS **`property`** element, such as **`type`** or **`arrayLength`**, may be applied to this **`property`** element since it is in fact *defining a property*.

13.3.4.8 Exclude/Only/Add Attributes of the Property Element

These attributes limit certain values to certain platforms. The **`exclude`** and **`only`** attributes provide value-specific restrictions on platforms similar to the effect of the top-level **`excludePlatforms`** and **`onlyPlatforms`** attributes: they *exclude* certain platforms from testing certain values or specify that some values should *only* be tested on certain platforms. The **`add`** attribute specifies that the values should be *added* to the set of tested values only for certain platforms.

The syntax of these attributes is the same as mentioned in the ExcludePlatforms Attribute section above.

13.3.4.9 `<Set>` Child Element of the `<Property>` Element for Delayed Values

Normally any non-parameter initial property values are set before the test case application is started (after ***initialize*** and before ***start***). This means that such values are stable during the life of the test. In some cases it is useful to set writable property values *during* execution.

The **`set`** child element can be used to "schedule" the setting of a property value at some time *after* the application is started. Delayed property settings in applications are described in the ***Application Development Guide***, and the syntax here is similar: the **`set`** child element can occur more than once, and have **`delay`** and **`value`** attributes to specify the delay after start (in seconds, floating point), and a value to set at that time (relative to when the application is started). A delay of zero causes the value to be set immediately after the application is started. The following example:

```
<property name='myctl'/>
  <set delay='1' value='10'/>
  <set delay='1.001' value='20'/>
</property>
```

sets the **`myctl`** property to 10 one second after start, and sets the value to 20 one millisecond later.

*13.3.5 Case Element of **<Tests>** Top-level Element*

The **case** element defines a non-default test case when the default test cases are not sufficent. It is necessary when the automatic parameterization of the default test case is insufficient for testing the worker(s).

The allowed attributes of a case element are: **Name**, **OnlyWorkers**, **ExcludeWorkers**, **OnlyPlatforms**, **ExcludePlatforms**, **DoneWorkerIsUUT**, **Duration** and **Timeout**. All but **Name** have the same function as previously defined for the top level **tests** elements, but only apply to this case.

If **ExcludePlatforms** is set at the top level, setting it again at the case level adds to the list of platforms that will not be tested. Setting **OnlyPlatforms** at the case level if **ExcludePlatforms** is set at the top level is valid, but **OnlyPlatforms** at the case level cannot list a platform that has been excluded at the top level.

If **OnlyPlatforms** is set at the top level, setting it again at the case level narrows the list of platforms that will be tested further, but it is not valid to set **OnlyPlatforms** at the case level to a platform not in the top level **OnlyPlatforms** list. It is valid to set **ExcludePlatforms** at the case level if **OnlyPlatforms** is set at the top level, but excluding a platforms not in the top level list has no effect.

ExcludeWorkers and **OnlyWorkers** at the top and case levels interoperate in the same way as **ExcludePlatforms** and **OnlyPlatforms**.

DoneWorkerIsUUT at the top level will apply the attribute to all cases, but can be set at the case level to apply to only select cases.

The allowed child elements under a case element are: **input**, **output**, and **property**.

Each case can override or use the default inputs and outputs for each port, and each case can override the property values tested for each property. If no **input** or **output** is defined for an input/output port, then the default **input**/**output** is used (the **input**/**output** defined for the port under the top-level **tests** element). If no **property** element is present for a property under a **case** element, then the values defined at the top level are used. For parameter properties, the default values tested are derived from the values defined in all the workers' build configurations, but this automatic default can still be overridden (limited) by a **property** element at the top level or under a **case** element.

13.3.5.1 Name Attribute of Case Elements

This optional string attribute specifies the name of the test case. If not present, the name of the case is **case** followed by a case number starting at zero, with at least 2 digits (i.e. the second case would be **case01**, and the 101st case would be **case100**).

The name of a case is listed in various reports, and can be used when specifying that only certain cases (rather than all cases) should be executed or verified.

13.3.5.2 Input Element under Case Elements

This element specifies how input data is generated for a port, in a test case. If not specified, the default input source for the port specified at the top level is used. The **port** attribute of the **input** element specifies the **port** this input element applies to. The **name** attribute, when present, indicates that a specifically named input source defined at the top level should be used for this port. If the named input source at the top level already has a **port** attribute, no **port** attribute need be supplied for this **input** element.

When the **name** attribute is specified, none of the **file**, or **script** attributes are allowed. If there was a top-level **input** element like this:

```
<input name='pulsegen' port='in' script='mygen.py'/>
```

then a **case** element could simply have:

```
<input name='pulsegen'/>
```

Similarly, it the top-level **input** element was this (with no **port** attribute, allowing it to be used for different ports):

```
<input name='pulsegen' script='mygen.py'/>
```

then a **case** element could have:

```
<input name='pulsegen' port='in'/>
```

13.3.5.3 Output Element under Case Elements

The **output** element for a test case acts the same as the **input** elements. They refer to a named **output** element at the top level or override the default, per port.

13.3.5.4 Property Element under Case Elements

The **property** element for a test case acts the same as the **property** elements at the top level: it specifies values to be used for the named property for this test case. If a property is not mentioned in a **case** element, the default top level values are used.

A single test case can have multiple values for any property. Subcases are automatically generated for all combinations of property values specified for the test case whether specified at the top level as default sets of values, or specified for the test case in the **case** element.

Individual values for a property for a test case may be restricted to certain platforms as described above in the Exclude/Only/Add Attributes section.

13.4 Preparing Unit Test Inputs

An **input** element must be specified for each input port of the component, either at the top level as a default for all cases, or for specific cases. It either specifies a pre-existing data file to use (using the **file** attributes) or a command to execute to generate the input data which can depend on the property settings for the specific subcase (using the **script** attribute). While a specified input file applies to all subcases regardless of property settings, a generator script can generate input data for each subcase that depends on all its property settings.

The format of the data in the (possibly generated) file is a series of message payloads as defined by the protocol for the port, as described in Message Payloads on Data Ports. The data must respect the setting pf the **ocpi_endian** built-in parameter property, whose value is available to all data generation scripts. All platforms currently supported use only little-endian data layout, but to test a worker that might be built for different endian systems, the layout of the data must match this parameter value.

The generator scripts for input ports are run for each subcase, with all property values for the subcase available to the script. The script is responsible for writing a file whose name is provided on the command line. Since the script command is executed as it would be on the shell command line, it can be written in any language, such as python, bash, or even compiled C or C++. It is executed in the context of the development system (not the target, potentially embedded system), so it can depend on any tools installed on the development system. However, scripts that depend on tools not installed or required as part of OpenCPI will make the project as a whole less portable.

An example input generator script written in the python language is below, for a FIR filter component. The script depends on two properties. The first **COEFF_WIDTH_p** is a parameter specifying the bit-width of samples. The second **NUM_TAPS_p** is the number of taps in the filter. The script generates an impulse, with a maximum value followed by zeroes. The file is binary 16 bit signed fixed point data.

```
#!/usr/bin/env python3
import sys, os, struct
max_tap  = pow(2,int(os.environ.get("OCPI_TEST_COEFF_WIDTH_p"))-1)-1
num_taps = int(os.environ.get("OCPI_TEST_NUM_TAPS_p"))
fo       = open(sys.argv[1], 'wb')
for j in range(num_taps):
    fo.write(struct.pack('h', max_tap))
    for i in range(1,num_taps*2):
        fo.write(struct.pack('h', 0))
```

If the script was in the local file **generate.py** and made executable (e.g. with **chmod a+x generate.py**), and the input port was named **in**, then the input specification that used this script would be:

```
<input port='in' script='generate.py'/>
```

An input generation script must return exit status of zero/non-zero for success/failure.

13.5 Preparing for Unit Test Output Verification

An **`output`** element must be specified for each output port of the component, either at the top level as a default for all cases, or for specific cases. It either specifies a pre-existing file to compare test output data against (using the **`file`** attribute), or a command that examines the data produced at an output port to decide whether the execution of the subcase was successful (using the **`script`** attribute). Output verification scripts have access to the output data produced, the input data provided to all input ports, and the final values of all properties at the end of execution (in the environment).

If the component had input ports **`in1`** and **`in2`**, and an output port name **`out`**, and a script named ***`<command>`***, in the script attribute, the actual command executed would be:

`<command> <output-from-port-out> <input-to-in1> <input-to-in2>`

The three filename arguments would be added by the unit test framework to run this output verification script for a given subcase, providing the file names for the data associated with the subcase (input and output). The script would run in the development environment and not in the environment of a potentially embedded target platform.

As with input data, the message payload formats will comply with the lay out as described in Message Payloads on Data Ports, and also respect the value of the built-in **`ocpi_endian`** parameter property.

An output verification script must return exit status of zero/non-zero for success/failure.

13.6 Summary of Steps Prior to Test Execution and Verification

After creating the test suite in the ***<component>*.test** directory using the **ocpidev create test** command, the following steps are taken prior to running any tests.

- Adding the elements (**input**, **output**, **property**, **case**) to the ***<component>*-test.xml** file.
- Prepare any input data files or input data generator scripts.
- Prepare any output data files (for comparison) or output verification scripts.
- Run the ***generate*** phase (see below)
- Examine the report created in **gen/cases.txt** to see the generated subcases.
- If any workers are HDL workers (which must be built for some HDL platforms), run the ***build*** phase to build the bitstream/executables required for testing.

After these steps, all applications, HDL assemblies, input data sets and verification scripts have been generated (in the **gen/** directory) and any required HDL assemblies have been built for the desired HDL platforms. The ***generate*** phase does not depend on which platforms any of the workers being tested have been built for. Prior to the ***build*** phase no compilers or other build tools are required or used. The ***build*** phase *does* require any HDL workers to have been built for the desired platforms.

Normally, both the generate and build phases are performed with the single command:

```
ocpidev build
```

in the test suite's directory own, or:

```
ocpidev build test <test-name>
```

in a library directory or the project directory.

For more fine-grained control, the ***generate*** phase can be run by itself using the

```
ocpidev build --generate
```

command. This can be useful when debugging the XML file and checking the generated results, i.e. in the **gen/cases.txt** file.

When the ***build*** phase is invoked, the ***generate*** phase may be re-invoked based on file dependencies. The **ocpidev** options **--hdl-platform**, **--only-platform**, **--exclude-platform** may be used on the command line for building to modify defaults specified in the project or library.

13.7 Phases for Test Execution and Verification

After the off-line steps described above, and any remote systems are enabled, there are three phases that relate to actually executing tests and performing verification, and these phases are based on available local and remote platforms on which to execute tests.

The **`ocpidev run`** command on a test suite normally runs the three execution-related phases: ***prepare***, ***execute*** and ***verify***. To run the phases individually, use the **`--phase`** option to indicate which phases to run. I.e.

```
ocpidev run --phase prepare --phase execute # stop before verify
ocpidev run --phase execute # rely on previous prepare, don't verify
ocpidev run --phase execute --phase verify # use previous prepare
ocpidev run --phase verify # just verify previous execution
```

Skipping the ***prepare*** phase has the benefit of avoiding some setup overhead and time, but runs the risk of something changing in the environment (e.g. a remote system becoming unavailable).

13.7.1 Preparing for Execution: Discovery and Execution Script Generation

The ***prepare*** phase, by itself, is invoked by the **`ocpidev run --phase prepare`** command. It considers all available platforms, local and remote, including available RCC, HDL hardware and HDL simulators.

This phase also considers which RCC built artifacts are available as well as which HDL bitstream/executable artifacts have been built locally from generated HDL test assemblies in the ***build*** phase. From the combination of available platforms and available artifacts, it determines which subcases can be run on which platforms, and generates execution scripts accordingly. These execution scripts are generated in the **`run/`** subdirectory of the test suite. The **`gen/`** subdirectory captures the results of the ***generate*** and ***build*** phases, and the **`run/`** subdirectory captures the results of the ***prepare*** and ***execute*** phases. Both subdirectories are removed by **`ocpidev clean`**.

When determining available artifacts for unit test execution, it does *not* look at the prevailing setting of the **`OCPI_LIBRARY_PATH`**, but forces a new environment value that limits the artifact search to local artifacts in the component library, the local **`gen/assemblies`** directory and the built-in **`ocpi.core`** project.

After determining available and appropriate test execution platforms, this phase generates per-platform scripts that run all feasible subcases on that platform. These scripts are placed in a subdirectory of the **`run/`** directory named after the platform.

13.7.2 Executing Tests on Available Platforms: the Execute Phase

The ***execute*** phase executes, for each available platform, test applications for all possible cases and subcases. The scope of execution is filtered by the

OnlyPlatforms and **ExcludePlatforms** XML attributes, and the **--only-platform**, **--exclude-platform**, and **--case** options to the **ocpidev run** command.

The ***execute*** phase iterates through platforms (sequentially), executing all subcases on each platform in turn, and recording the results for later verification. The recorded results are:

- the output data from output ports
- the final values of all properties, including volatile ones
- a log of the actual execution of the tests.

All these results are recorded in the platform's subdirectory of the **run/*<platform>*/** directory with different files for different subcases.

After the execution of each test on a given platform, a console message will indicate whether the execution succeeded or failed. If an execution fails, the execution of all tests is stopped, and the outputs can be examined for the cause of the failure.

13.7.2.1 Command Options that Control Execution of Tests

The **--accumulate-errors** option to **ocpidev run** can be used to continue execution of test cases even if some fail, with the final exit status of the **ocpidev run** command indicating whether any tests failed. This option also applies to during verification of test cases. Not setting this option is useful to solve each problem as it occurs, while setting it allows all errors to occur and be analyzed or examined later. There is no option that will accumulate errors across multiple test directories in a library or project.

The **--verbose** (**-v**) option to **ocpidev run** puts all detailed execution logging in the console output (as well we into the subcase log file).

The **--keep-simulations** option can be set to cause the runtime logging of simulator execution (in the **run/<simulator-platform>/*.simulation** directories) to be retained after successful executions on simulation platforms. Successful verification for a platform normally causes the associated simulation logging directory to be removed immediately. Keeping simulation output may use lots of file system space (100s of GBs in some extreme cases). This option is useful when examining simulation traces of successful subcase test runs.

The **--case** option (which can be indicated more than once) is a wildcard pattern indicating which cases/subcases should be executed or verified. Subcases are named ***<casename>.<subcase#>***, so this option may be set to patterns that affect certain cases or subcases. Subcase numbers are listed in the report in **gen/cases.txt** produced by the ***generate*** phase. The default case name is **case00**. For example, to

only run subcases that end in `3`, you could specify: **`--case='case*.*3'`**. Multiple patterns are allowed, such as: **`--case=case00.01 --case=case00.03`**.

13.7.3 Verifying Test Results

The ***verify*** phase relies on the previous execution of appropriate subcases for all platforms, and performs verification using the results recorded previously in the ***execute*** phase.

If both ***execute*** and ***verify*** phases are indicated (or ***prepare***, ***execute*** and ***verify*** — the default), the ***execute*** and ***verify*** steps are interleaved by subcase, with each subcase being verified immediately after it has been executed. Note that the ***verify*** phase can be run without access to the runtime platforms, and is sometimes conveniently done later, even on a different system, since most verification scripts are written in Python.

The command:

```
ocpidev run --phase verify
```

performs *only* verification.

The ***verify*** phase by itself does not involve any execution or access to local or remote execution platforms and thus can be performed offline. Whether the ***verify*** phase is executed with the ***execute*** phase or by itself, the view option will run the defined view scripts with verification.

If the **`--accumulate-errors`** option is *not* set, the ***verify*** phase will immediately stop and return an error if any subcase fails by returning non-zero exit status. If the option is set, all subcases will be verified with failures reported as they occur. The output of the verify phase makes it clear whether each subcase **`PASSED`** or **`FAILED`**. The individual verification scripts are required to return a standard shell command exit status of zero for success and non-zero for failure.

The **`--case`** option desribed bove applies to standalone verification as well.

13.7.4 Viewing Test Result Data

For interactive debug purposes, the output data can be “viewed”, by running the view script associated each output port. Viewing is enabled one of two ways:

The **`--view`** option simply adds viewing to whatever phases are being performed by **`ocpidev run`**. Alternatively, you can specify viewing as a phase, e.g.:

```
ocpidev run --phase view
```

which will *only* performing viewing (running the specified view scripts associated with any output port in the unit test description XML). Viewing is only possible after an execution that produces output data from a port.

When verification is specified, viewing happens per subcase execution before each verification. View scripts typically put up a window to display the data, and then they

may wait for user input or simply return, allowing the data to be displayed while verification proceeds.

13.8 Testing on Remote Systems

The default behavior of the unit test framework is to test on locally available platforms. This usually means using the local CPU and any locally attached GPUs or FPGAs. However, there are two methods to including non-local, network-reachable systems in the set of available platforms for testing.

The first and more common method is to use the ***remote container*** feature of OpenCPI that generally supports OpenCPI applications executing on a mix of local and remotely accessible platforms. This feature is fully described in the ***Application Development Guide***. In order to have the unit test execution framework consider remote systems as test platforms, the remote and local systems have to be enabled as described in that document. Using remote containers this way requires no common network mounts between the local and remote systems and prior download of OpenCPI software, but has some firewall-related limitations in some environments.

The second method is specific to unit testing and involves defining possible remote systems using the **`OCPI_REMOTE_TEST_SYSTEMS`** environment variable. This method is described in the next section.

13.8.1 Defining Remote Systems for Executing Tests without Remote Containers

When not using the simpler (no NFS) remote containers described above, the following alternative method may be used.

In order for the unit test framework to execute tests on platforms that are not available on the development system, remote systems with additional platforms must be specified. Such platforms are not discovered automatically but are specified in the **`OCPI_REMOTE_TEST_SYSTEMS`** environment variable.

Remote systems are accessed using the **`ssh`** remote execution command, with an **`ssh`** server capability required to be enabled on the remote system. This environment variable is a colon separated list of remote system specifications, and each remote system is specified by 5 fields separated by '**=**'. The four fields are:

1. Remote Host name/IP address
2. SSH user name
3. SSH password (yes, this is not a secure solution)
4. Project directory NFS mount path as seen on remote system
5. SSH Version (optional)

The remote system must meet the following requirements:

- SSH access from the development host, using the username and password in the first three (and optional fifth) fields. If the remote system is set up for public/private

key access control from the development system, the password is not used, but must still be non-empty.

then, after a successful SSH login from the development system

- The project's directory on the development system must be mounted (NFS or equivalent) for access from the remote system, using the path of the fourth field.
- The OpenCPI kernel driver must be loaded on the remote system
- The **OCPI_CDK_DIR** environment must be set up properly consistent with the development host. An OpenCPI CDK installation is assumed, with the remote system and development system using the same OpenCPI release.

The project directory mount, the OpenCPI CDK installation, and the kernel driver may either be established at boot time or at SSH login time on the remote system.

A remote system may support multiple platforms (e.g. both HDL and RCC). Whatever platforms are available on the remote system will be discovered when the remote system is contacted. Like local platforms, these discovered platforms are subject to the filters specified by **OnlyPlatforms** and **ExcludePlatforms**.

Remote systems are frequently embedded systems which do not host a development environment, but any system can be a remote system. E.g. if the development system is running CentOS7, the remote system could be Ubuntu 18.04 to run tests on that system also. Of course the Ubuntu 18.04 system will only run RCC artifacts built for Ubuntu 18.04.

Remote systems may be defined in the environment so that they are available for all test suites in the project, e.g.:

```
export OCPI_REMOTE_TEST_SYSTEMS:=10.0.1.16=root=root=/mnt/myproj
```

13.9 Summary of Unit Test Phases and `ocpidev` Options

The **`ocpidev`** "verbs" that apply to unit testing are:

`create`, **`delete`**, **`build`**, **`clean`**, and **`run`**

The **`ocpidev build`** command is used to perform the ***generate*** and ***build*** phases, and the **`--generate`** option performs the ***generate*** phase *without* building. The **`--hdl-platform`** option is used to cause HDL test assemblies to be built for HDL platforms.

The **`ocpidev clean`** command performs as expected, removing all files resulting from generation, building and running tests.. The **`--simulation`** option limits the cleaning to the simulation outputs produced by executions on simulation platforms, and the **`--execute`** option removes all files produced by test preparation and execution, including simulations (the **`run/`** subdirectory), but not results from building.

The **`ocpidev run`** command is used to perform the ***prepare***, ***execute***, and ***verify*** phases, with the **`--phase`** option used to be more selective as to which of those phases are actually performed. The **`--accumulate-errors`** option allows all tests to be run, even if some fail. The **`--keep-simulations`** option preserves simulation output even when test execution succeeds. The **`--only-platforms`** and **`--exclude-platforms`** options can filter which of the available platforms are actually used for executing tests. The **`--cases`** option can limit which test cases are executed and/or verified.

14 Developing OpenCPI Assets in *Projects*

In OpenCPI a ***project*** represents a work area in which a variety of assets are created and developed. Projects can contain all types of assets that are described fully either in this document or in others. A project can contain:

- Component libraries with specs and workers.
- Applications (described in the ***Application Development Guide***).
- HDL primitives and assemblies (described in the ***HDL Development Guide***)
- HDL devices, cards, slots (described in the ***HDL Development Guide***)
- Platform support assets (described in the ***Platform Development Guide***)

A project is a standard directory structure that holds the various OpenCPI assets in both source code form and built form, along with the XML files that describe how they are built. The project structure provides a means to bundle a collection of assets which may have a logical relationship or be created for a specific application.

The ability to develop assets outside of a project (a.k.a. *standalone*) is also supported, but is preliminary and subject to change, and not discussed further in this section.

The **`ocpidev`** tool is used to create and then populate a project directory structure with the various asset types. The created skeleton directory structure is always buildable.

The structure of a project, and types of assets (shown enclosed in <>), is shown in the following diagram (with the other files omitted except at the project level).

```
Project.xml
Project.exports
applications/<applicationXYZ>/
specs/
components/<componentlibXYZ>/<workerXYZ>/
                             /specs/
---OR---
components/<workerXYZ>/
          /specs/
hdl/primitives/<primitiveXYZ>/
hdl/assemblies/<assemblyXYZ>/
hdl/platforms/<platformXYZ>/
hdl/devices/<device-workerXYZ>/
           /specs/
hdl/cards/<card-device-worker>/
         /specs/
```

The optional top level **`specs`** directory is separate from the **`specs`** directory in any component library. It is a project-wide **`specs`** directory that is usable by all component libraries in the project. It can exist in the project, for use by other projects, even if there are no component libraries in the project.

Creation of a project (using **ocpidev**) creates a skeleton directory structure that is buildable, but it will build nothing initially as it contains no assets. All the intermediate directories are created by **ocpidev** as needed. If there are any component libraries created in the project (using **ocpidev create library *<libname>***) a **components** directory is created, under which those component libraries will be created. Alternatively, for simpler projects which only have a single component library, the **components** directory ***is*** the single component library (created using **ocpidev create library components**) and workers are created directly under **components**. A project currently cannot easily be changed between the "flat", single library structure and the "hierarchical", multiple library structure.

When creating a worker, if no library is specified and the current directory is not in a component library already, it is placed directly in the **components** directory. It is an error to create such workers if component libraries already exist under **components**. Conversely, it is an error to create a component library under **components** if workers already exist there or **components** was already explicitly created.

In addition to the various directories, two required files are generated at the top level when the project is created by **ocpidev**:

Project.xml: the XML file that specifies attributes that are used *project-wide*, for all assets at all directory levels.

Project.exports: a file that specifies which assets and files should be visible from outside the project, i.e. visible to other projects which use some of the assets in this project.

These files are automatically created when the project is created, but may be edited later as necessary. The **Project.xml** file must exist; the **Project.exports** is optional and created and edited manually.

14.1 Managing Project Assets

The **ocpidev** tool described in detail later (see also the **ocpidev(1)** man page) is used to manage all the asset types in a project. It is used to create, delete or test assets. Once created, the assets are based on text files that must be edited. Assets are created using the **ocpidev create** command and they are deleted using the **ocpidev delete** command. Most assets, including projects themselves, are based on a directory, with an XML file for descriptive information, in that directory. These include:

- Component libraries
- Workers
- Unit test suites
- HDL device workers
- HDL platforms
- HDL primitives (cores and libraries)
- HDL assemblies
- Applications (except the simplest ones)

When an asset is created, the appropriate directories are also created, an initial XML file is created in the new directory, and in some cases other initial files are also created.

Some assets are simply files and when created, an initial version of the file is created in the appropriate directory in the project. This type of asset includes:

- Component Specs
- Protocol Specs
- HDL card definitions
- HDL slot definitions
- Applications (for simple XML based applications with no ancillary files)

When creating specs, protocols, workers and unit tests, a library option (**-l <library>**) may be supplied to **ocpidev** indicating which component library the asset should be added to. If the project has a single library in the **components** directory, this option is not used. For hardware-specific HDL workers, the **-h <library>** option specifies a directory under the project's **hdl/** subdirectory where the device worker should be created (usually **hdl/devices**).

When adding a platform-specific device worker or device proxy, a platform option (**`-P <platform>`**) may be supplied to indicate which platform-specific device library to add the device worker to. Portable device workers that are *not* platform-specific should not use this option.

14.2 Package IDs

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 actually identified by their package-IDs in the current OpenCPI release. Eventually, package-IDs will be used uniformly and universally.

A package-ID is a hierarchical sequence of names separated by periods. This OpenCPI package naming scheme follows the Java package naming conventions, which are roughly based on a reversed Internet domain name with the top-level domain first, and the more specific or local names after that. Thus if CNN had an OpenCPI project full of news-related components, its package-ID might be **`com.cnn.news`**.

Warning: When changing the PackageName or PackagePrefix of an existing project the project needs to be both unregistered and re-registered then cleaned and rebuilt. This also includes cleaning and rebuilding any projects that depend on this project.

14.2.1 Package-ID Usage and Structure

Package-IDs are divided into two parts, the ***package-prefix*** and the ***package-name***. The package-prefix is the string before the last period, and the package-name is the name after the last period. Generally the package-ID of an OpenCPI asset consists of the package-prefix from its containing environment (e.g. the project of a library, or the library of a worker). Its package-name is simply the name given when the asset was created (using **`ocpidev`**). Thus the package-prefix acts as the name scope for the asset. This package naming scheme (from Java) provides for globally unique identifiers that are human readable (and type-able) and already administered by a globally known organization. A package-ID consists of ***`<package-prefix>.<package-name>`***.

OpenCPI defines two reserved top level package prefixes: **`ocpi`** and **`local`**. The **`ocpi`** prefix is managed by the OpenCPI maintainers and is used for assets that are located at the OpenCPI github repository. Projects that are maintained there all have names with this prefix (currently the three projects in use are **`ocpi.core`**, **`ocpi.assets`**, and **`ocpi.inactive`**). The second reserved prefix (**`local`**) is the default packageID for all projects that are created with no explicit package-ID. This means that basic development projects (or tutorial examples) are not required to have a global identity to simply get started. As soon as other projects need to use or depend on a project, it should have a more explicitly assigned package-ID.

The package-ID of a component specification is generally prefixed with the package-ID of its containing library or project (followed by a period). This prefix is the name scope in which the component is defined. This allows components to be specified and implemented by different organizations, while still allowing any implementation found in a library to satisfy any (other) organization's component specifications. E.g., my project can have an additional, alternative implementation of a component specified in another

organization's library, or can define its own specification for a component with the same package-name with a different package-prefix (name scope).

14.2.2 Package-ID attributes in XML Files

There are several cases where attributes are set in XML files to help specify package-IDs. In an application XML file, the top level **`package`** attribute provides a default package-prefix for all components mentioned in the file (that do not have periods in their name).

A component spec XML file can specify a package-prefix in a top-level **`package`** attribute, This would override the default, which is the package-ID of the library in which the spec is defined. If the spec is not in the library, but rather in the top level specs directory of a project, this would override the default prefix from the project's package-ID.

The package-ID of an asset is normally inferred from the name it was created with serving as package-name with the package-ID of its containing asset serving as package-prefix. When this inferred package-ID is not what is needed, certain attributes in the asset's XML file can override this default behavior. The **`PackagePrefix`** attribute can override the default package-prefix supplied from the package-ID of the containing asset.

The default package-prefix for projects is **`local`**. As soon as this default is not what is wanted, the **`PackagePrefix`** attribute can be set in the project's **`Project.xml`** file in the top level project directory. Similarly, a component library in a project could override its prefix (which would normally be the package-ID of its project) using the **`PackagePrefix`** attribute in the library's ***`<library-name>`*`.xml`** file.

In the unusual case where the package-name should be different than the name the asset was created with (which is also the name of its directory), the **`PackageName`** attribute can be used to override that part of the package-ID.

When it is required to simply set the entire package-ID for an asset, the **`PackageID`** attribute can be used. This overrides any logic for combining the package-prefix and the package-name.

14.2.3 How to Determine an Asset's Parent (and thus Default Package-Prefix)

The parent of a component library, any of:

```
components/
components/<library>
hdl/devices/
hdl/cards/
hdl/adapters/
hdl/platforms
```

is the project itself.

An HDL platform's parent is the **hdl/platforms** directory.

A **hdl/platforms/*<platform>*/devices** library's parent is the containing platform.

A project has no parent, and so its package-prefix defaults to **local**.

The default package-prefix of a component (which does not have its own directory) is ***<parent's package-ID>.<component-name>***.

14.2.4 Package-ID Examples

Within the **ocpi.assets** project the **Project.xml** file contains:

```
PackagePrefix='ocpi'
```

Thus the package-ID for the project is: **ocpi.assets**.

The **dsp_comps** library in the same project (located at **assets/components/dsp_comps**) has no attributes set in its ***<library-name>*.xml** file, so:

- The package-prefix defaults to package-ID of parent (project): **ocpi.assets**
- The package-name defaults to library's directory name: **dsp_comps**
- Thus the package-ID is: **ocpi.assets.dsp_comps**
- If we had **PackageID='full_package'** in the file **dsp_comps/dsp_comps.xml**, the package-ID of the library would be, entirely, **full_package**

The assets project has an HDL platform (**matchstiq_z1**) with its own devices library in:

```
assets/hdl/platforms/matchstiq_z1/devices
```

- With no package attributes set, the package-name defaults to the directory: **devices**
- The package-prefix defaults to the package-ID of parent (the **matchstiq_z1** platform), which is: **ocpi.assets.platforms.matchstiq_z1**
- Thus the package-ID for the library is: **ocpi.assets.platforms.matchstiq_z1.devices**

The **fir_real_sse** component spec in **assets** project's **dsp_comps** library would be referenced in an application as:

```
ocpi.assets.dsp_comps.fir_real_sse
```

This is because a spec's parent is the library it is in.

14.3 The Project Registry: How Projects Depend on and Find Each Other

Every project depends on some other projects. At a minimum all projects depend on the core project provided as part of OpenCPI. Its package-ID is **`ocpi.core`**.

When assets in a project depend on assets in other projects, the project must declare that it depends on such other projects. Here are some examples of how assets can depend on other assets, and thus how an asset in one project may depend on assets in other projects.

- A worker may depend on a spec (OCS), whether in the same library as the worker, a different library in the same project or in a different project.
- An HDL assembly may depend on workers in any library in the same project or workers from libraries in other projects.
- An HDL worker may depend on an HDL primitive library in the same project or in a different project.

A project declares its dependency on another project by either specifying the dependency when the project is created (using **`ocpidev`**), or adding the dependency to its **`Project.xml`** file later. In either case the dependency is declared by providing the package-ID of the other project. As mentioned above, the dependency on the **`ocpi.core`** project is always assumed.

A project is visible to other projects by being ***registered*** under its package-ID. A project can only depend on another if that other project is registered with the OpenCPI installation. The default condition of a project, when created, is to be ***unregistered*** and thus not visible to others. When a project is in a suitable condition to be depended on by others (i.e. it contains assets that are ready to be used by other projects), it can be registered at that time (using the **`ocpidev register`** command). This action can be reversed using the **`ocpidev unregister`** command.

The registry includes a mapping between package-IDs and the pathname of registered projects. You use the following ocpidev command to show the mapping:

```
ocpidev show registry --table
```

The built-in projects supplied with OpenCPI are automatically registered in the default project registry as part of the installation process. You can register new projects that you create with the command:

```
ocpidev register project <my-project-name>
```

and unregister them with:

```
ocpidev unregister project <my-project-name>
```

Unregistering a project does not delete the project. It only removes it from the registry. You can also create and register a project with a single ocpidev command:

```
ocpidev create project --register <project-name>
```

14.3.1 Projects That Implement Platforms

While platform assets (sometimes called *platform support packages*) are one of many types of OpenCPI assets, they are special in two ways:

- they enable other assets to be built (compiled, synthesized, etc.) targeting their platform
- when they are in a registered project, they are usable for asset building for *all other projects* regardless of whether those *other* projects declare a dependency on the project containing the platform asset

A project **A** does not have to explicitly depend on project **B** simply to build **A**'s assets targeting a platform implemented in project **B**. This allows building for platforms without knowing which project they are implemented in. For any other dependency between assets in one project and another, the project dependency must be declared when the project is created or later in the project's **`Project.xml`** file.

14.3.2 Project Registries for Sandboxed Project Development

Projects depend on other projects, and make themselves visible to other projects via a ***project-registry*** that is part of the OpenCPI installation. The registry includes a mapping between package-IDs and the pathname of registered projects. This mapping is shown using the **`ocpidev show registry --table`** command.

There are two different relationships between projects and registries:

- Every project is automatically ***associated with*** a project registry upon creation and thus can view and depend on other projects registered there.
- A project can be ***registered in*** its associated project registry, and thus be visible to other projects associated there. Projects are not initially registered.

In cases when these three things are true:

- multiple versions of some projects are developed simultaneously
- those projects are depended-on by others
- the different project versions all use the same OpenCPI installation

it is useful to create an additional registry as a sandbox for that set of copies or versions of projects. To create or delete an alternative registry, the **`ocpidev create|delete registry`** command can be used. This establishes a registry in a user-specified directory. To use such an alternative registry as the new default or to use platforms

defined and/or built in projects there (e.g. HDL simulators), the **`OCPI_PROJECT_REGISTRY_DIR`** environment variable must be set point to it. This overrides the default registry that is part of the OpenCPI installation.

Any projects created with this environment variable set will be associated with the alternative registry. This association of a project to its registry is persistent. This association does not depend on the current setting of the environment variable. To change an existing project's association with its registry, the **`ocpidev set registry`** command can be used. It can either specify an alternative registry directly or if none is specified, the default (possibly from the environment variable OCPI_PROJECT_REGISTRY) will be used. The association can be undone using the **`ocpidev unset registry`**, in which case the default registry will be used when needed (and that new association will persist).

So there are four uses of the relationship between projects and a registries:

1. A project's *associated* registry, set at creation or by **`ocpidev set registry`**, is the way a project *depends* on others. Normally this association is automatic and unchanged.
2. A project being *registered in* its registry, is the way for *other* projects, associated with the same registry, to depend on it.
3. A project depends on another *explicitly*, by the declaration in its **`Project.xml`** file, enabling assets in one project to depend on assets in other projects.
4. A project can be *built* for platforms implemented by any registered project in its associated registry, without an explicit dependency.

14.4 XML Files in Projects

Most of the directories in a project contain an XML file containing attributes pertaining to the directory and those below it. The name of the file is the name of the directory, minus any suffix in the directory name, plus the **`.xml`** suffix. E.g. for a directory named **`myworker.rcc`**, the name of the XML file in that directory is **`myworker.xml`**. For directories that are assets, the name is the name of the asset (plus **`.xml`**). For directories that are just containers for assets, the name is still the name of the directory, e.g. in the **`applications/`** directory in a project, there can be an XML file named **`applications.xml`**. There are two exceptions to these naming rules:

- the XML file for a project has the fixed name **`Project.xml`**, allowing the project to be renamed or moved without changing the name of the file.
- the XML file for a unit test suite, whose directory name is ***`<component-name>`*`.test`**, is ***`<component-name>`*`-test.xml`**.

The **`ocpidev create`** command creates the XML file for the asset, even when the asset does not have a directory (such as a protocol definition XML file).

When a directory contains a number of subdirectories for the same type of asset, an attribute can be set in its XML file which lists the assets to build, or the assets to exclude from building. These attributes are optional, and when not specified, all such assets are built. For example, in a component library where all the subdirectories contain workers, the default XML file contains no attributes listing workers, which implies that all worker subdirectories should always be built. If there are workers that should *not* be built, or they should be built in a particular order, then the **`Workers`** attribute can be specified to list the explicit set of workers that *should* be built, in order, e.g.:

```
<library Workers='fft.rcc fft.hdl'/>
```

Alternatively, if there are a few workers that should not be built, the **`ExcludeWorkers`** attribute can list them, in which case all workers *except* those listed will be built.

This same idea applies to directories in a project that contain HDL assemblies, HDL platforms, HDL primitives, applications, devices, cards, and slots, etc. The exact name of this attribute, and other optional attributes, are described in the section for each asset.

Some attributes set in the **`Project.xml`** file in the top-level project directory apply anywhere in a project, when **`ocpidev build`** is invoked in any of the project's directories.

14.5 The Project.xml File for Project-wide Settings

This XML file is required in the top level directory of a project. It contains attribute settings that apply to all levels of a project. Its existence indicates that the directory is in fact a project. *In all directories under a project, this file is found by looking in parent directories until the* **`Project.xml`** *file is found.* This is similar to how the **`git`** tool finds the top level of a git repository by searching for a directory named **`.git`**.

The build-related attributes in this top level project file or a similar file in subdirectories with assets under them, are applied to building lower-level assets automatically. An example is the **`ComponentLibraries`** attribute.

Within an OpenCPI project, the **`ocpidev build`** or **`clean`** command can be run directly in any asset's directory (without specifying a noun or asset name), or in any of the "plural" directories containing similar assets.

The project attributes that may be set in a **`Project.xml`** file are in the following table and are all optional. This file must be present, but may be empty.

Table 8: Attributes set in the Project.xml file

Attribute Name in `Project.xml`	**Default**	**Description**
`PackageName`	Project directory's name	The name used for this project in its package-ID
`PackagePrefix`	**`local`**	The package-prefix for this project's package-IDall. The default, **`local`**, is appropriate when the assets are intended to be used only in the local organization or prior to registration.
`PackageID`	<prefix>.<name>	The package-ID of the project, overriding **`PackageName`** and **`PackagePrefix`**
`ProjectDependencies`	""	A list of the package-IDs of registered projects that this project depends on. The **`ocpi.core`** built-in project is assumed.

In order to avoid name space collisions when using multiple projects or component libraries (e.g. spec names and worker names), the project's package-ID is the default namespace for all named assets in the project.

The **`ProjectDependencies`** attribute should be used to declare other projects that this project depends on. This will cause these other projects to be used when searching for assets that are subject to search paths, such as:

- OCS files (when building workers)
- HDL primitives (when building HDL workers)

- Component libraries when building workers (for slaves of proxies, or devices of emulators)

While project dependencies should generally be the package-IDs of registered projects, they may also be relative or absolute pathnames of other projects. However using such direct pathnames is deprecated and may not be supported in the future.

Other useful XML attributes that can be specified in the **`Project.xml`** file include attributes providing default lists of build targets and platforms:

```
HdlTargets
HdlPlatforms
RccPlatforms
RccHdlPlatforms
```

Search list attributes may also be put in this file, although care should be taken to not unduly pollute search spaces for *all* assets in the project. These attributes include:

```
ComponentLibraries
HdlLibraries
```

Such attributes may be better placed in the ***`<library-name>`*`.xml`** files in component library directories.

14.6 Project Exports

When a project's assets are used by assets *outside* the project, they use the project's assets via its ***exports***. ***Exports*** of a project are the files within the project that are explicitly made visible and usable from outside the project. I.e., without exports, nothing in a project is intended to be visible outside the project. A project's complete directory structure contains source files and artifacts of the build process. The ***exports*** are the files needed by users of the project, and can be thought of as the installable and deliverable subset of the files in the project after it is built.

When other projects depend on a project, as specified in those projects' **`Project.xml`** file **`ProjectDependencies`** setting, that means they use project exports from those projects they depend on. The projects they depend on must be registered.

The ***exports tree*** is a directory containing the project's exports, and is constructed as a tree of symbolic links, under the directory named **`exports`** at the top level of a project. The structure of the exports tree is not necessarily the directory structure of the project itself, but is a structure appropriate and convenient for users of the project's assets. By constructing the **`exports`** tree using symbolic links, the exported view of a project uses no extra space (no copies). The assets are therefore used externally exactly where they exist in source form or where they are built (although indirectly via the symbolic links in the exports tree).

Much like there is a standard directory structure for OpenCPI projects, there is an implied standard exports tree based on the contents of a project. This directory, **`exports/`**, is updated when builds are done at the project level.

The exports tree has two different uses. One is to allow the project's intended deliverable results to be used in-place, without any copying or "installing". The other is to provide an implicit recipe or bill-of-materials for creating an installable binary package for users of the project. In this latter case a simple single file deliverable package can be created (see below).

14.6.1 The Exports Tree

The top level directory **`exports`**, is created and populated automatically based on the file **`Project.exports`**. This **`exports/`** directory can always be deleted and recreated. It is never manually constructed or changed. If the **`Project.exports`** file is empty or does not exist no exports are created. The default export tree is created based on the assets in the project. It is the set of exports when the **`Project.exports`** contains the single line containing the single word: **`all`**.

The next sections describe the default **`exports`** tree and the format of the **`Project.exports`** file that can be used to add or subtract from the default exports.

Here are the rules used to populate the default exports tree when project-level builds are invoked at the top level of a project and the **`Project.exports`** file contains the single line: **`all`**.

For an example of how the exports tree works:

Note: This structure is subject to change between Major releases of OpenCPI and should not be depended on by any end user developed code.

Component library deliverables are made available in the exports tree under the **`lib`** directory, using the name of the library. If there is a component library in the project in the directory **`components/util_comps`**, where its own locally built deliverables are in its **`lib`** subdirectory (**`components/util_comps/lib`**), then these deliverables are available in the exports tree using **`lib/util_comps`**, which is a symbolic link to **`components/util_comps/lib`**, i.e.:

```
exports/lib/util_comps -> ../../../components/util_comps/lib
```

This directory structure is subject to change but creates an export *view* of the deliverables of the project using a sparse set of symbolic links. Users of the project, seeing only the exports tree, see **`lib/util_comps`** for this library.

All assets have a default place in the exports tree where other projects that depend on them can find them, if the project owner decides to export them.

14.6.2 The Project.exports File

The **`Project.exports`** file specifies which assets and files in the project should be made visible, usable, and accessible from outside the project (usually by other projects). If the file is missing or empty (or only contains blank lines or **`#`** comments), the project is not intended to be used by others outside the project.

This *nothing-is-exported* condition is appropriate for projects in development before anything is ready for use by others (and before a package-ID is assigned). It is also appropriate for projects that contain end products like applications for test purposes.

At the other end of the “visibility” spectrum, the **`Project.exports`** file can contain a single line containing the word **`all`**, which indicates that all assets in the project should be visible/usable from outside the project.

Between these two extremes we can selectively export assets and/or files in the project.

14.6.2.1 Exporting Assets by Type or Name

To export assets, we simply provide their type and name, or in some cases the plural form of their type to indicate all assets of that type. Here is a list of the lines you can use to export assets by listing their types and names. When ***`<name>`*** is missing, it indicates that all the assets of the indicated type should be exported.

```
all
spec <name>          # a spec in the top level specs/ directory
specs                # all specs in the top level specs/ directory
libraries            # all component libraries under "components"
library <name>       # <name> can also be a path within the project
hdl platforms
hdl platform <name>
platforms            # all possible platforms
rcc platforms
rcc platform <name>
hdl primitives
hdl primitive library <name>
hdl primitive core <name>
hdl assemblies
hdl devices
```

14.6.2.2 Exporting Individual Files and Directories

The **Project.exports** file may also contain lines that add and subtract files from the asset exports of a project. This capability it rarely used but covers some edge cases to augment or prune the exports tree created based on asset exports. File additions are lines that start with a plus sign (**+**), and subtractions are lines that start with a minus sign (**-**). White space (spaces or tabs) can precede the **-**, **+**, or **#** characters.

The format of addition (**+**) lines is two fields separated by white space. The first field is the relative pathname within the project for the file to be exported, and the second field is the location in the export tree where the file should be linked. If the second field ends in a slash, then the filename part of the first field is used as the file name in the exports tree. Pathnames or other names with embedded spaces are not currently supported.

The line:

```
+special_dir/special_file include/ # this exports my special file
```

would export the file in the project named **special_dir/special_file** as:

```
include/special_file
```

If the name of the exported file should be different, it can be included in the second field, e.g.

```
+special_dir/special_file include/different-file
```

If the second field is blank (doesn't exist), then the project file or directory is exported in the same place is it exists,

```
+special_dir # export this directory where it is in the project
```

would simply make **special_dir** a top level directory in the exports tree.

Any directory that is exported implicitly exports all files underneath it.

The first field can also have wildcard patterns using normal sh/bash wildcard patterns. A special string, **<target>**, indicates the software target that is currently being

exported. When a project is exported, it is done in the context of a particular software target. The line below

```
+applications/myapp/target-<target>/myspecial_exe  bin/<target>/
```

exports a secondary executable in the application.

Subtraction lines start with a minus sign (`-`). As with addition lines, the first field is the pathname within the project, with possible wildcards and **`<target>`** strings.

In both additions and subtractions the first field can actually be an extended regular expression if the field starts with a vertical bar (`|`). For example, an addition line might be:

```
+|^(abc|xyz) special_dir/
```

which would export anything starting with **`abc`** or **`xyz`** in the exported directory **`special_dir`**.

14.7 Archiving a Project to be used Elsewhere

[This feature is under development as of release 2.2]

A project can be archived and used elsewhere in source form or in built/exported form without source files. In both cases a compressed tar file is created from the current project.

To export the project in source form, in the project's directory, use the command:

```
ocpidev archive [ --exports|--verbose ] --output <archive-file>
```

The **`--exports`** option indicates that the archive file contains only the project's exported files and assets — essentially a non-source archive of what the project exports. This should be sufficient for users and other projects to use the assets in the project, assuming they have been built.

Without the **`--exports`** option, the archive is essentially a snapshot of the source tree, excluding any references to its environment (e.g. its registry and its imports from other projects).

In both cases the resulting file is an archive that can be simply extracted in another location using **`tar xzf <archive-file>`**. Note that the archive will be the project's directory under its current name, so that extracting it will create the project's directory under its original name in the current directory when ocpidev archive is invoked (including after any **`-d <directory>`** option is used).

If you are using the default registry (or setting the default in the environment) you can set the registry for the project using the **`ocpidev set registry`** command. Make sure that the registry contains any projects this exported project depends on.

If you want other projects to use or depend on *this* project, you can register it using the **`ocpidev register project`** command.

14.8 Using Other Projects that Exist Outside the Project Being Developed

The convenient and recommended way to use a project **A** from another project **B** is to add the package-ID of **A** to the **ProjectDependencies** attribute in the project **B**'s **Project.xml** file. This declares the dependency without using any environment settings or pathnames. The dependency is persistent (in **Project.xml**) and stays with the project even if it copied or moved. Since the other project is specified in **ProjectDependencies** by its package-ID, it can also be moved (and re-registered) and the dependency is still valid.

The actual directories searched within a project depends on the type of asset being sought. E.g., if a component library is being sought, the search will look for that component library within all projects mentioned in **ProjectDependencies**. E.g.:

```
ProjectDependencies='local.utils ocpi.assets'
```

would indicate that this project depends first on the **local.utils** project and then also depends on assets in the **ocpi.assets** project (as well as **ocpi.core** which is added to the dependency list). Any component library being sought might be in either one, and if found in **local.utils**, the **ocpi.assets** project will not be used.

When a more dynamic setting is needed, e.g. for temporarily using one version of a project vs. another, the **OCPI_PROJECT_PATH** environment variable can be used. This variable specifies a colon-separated set of other projects (by pathname) to be searched in order. These are searched *before* projects mentioned in the **ProjectDependencies** attribute to allow the environment variable to temporarily override what is declared persistently in the project. Projects in **OCPI_PROJECT_PATH**, are examined by looking in their **exports** subdirectory if it exists.

To summarize searching at the project level, the order is:

1. Use assets in the local project.
2. Use assets in projects in **OCPI_PROJECT_PATH** to temporarily override anything later in this list.
3. Use assets in **ProjectDependencies**, in the order given in that attribute (with **ocpi.core** automatically added at the end, as previously mentioned)

15 The `ocpidev` Tool for Managing Assets

The **`ocpidev`** command-line tool is used to perform various development-related tasks inside projects as well as retrieving information about the environment. When used in projects, it may be invoked at the top level of a project, or in lower level directories of the project as appropriate to the particular command being used. The **`ocpidev`** command has full tab completion for its options and arguments.

The general usage of the **`ocpidev`** command is:

`ocpidev [<options>] <verb> [<noun> [<name> [<arguments>]]]`

The options can in fact occur anywhere in the command for the user's convenience. The general usage concept is:

*perform the **`<verb>`** operation on the **`<noun>`** asset type whose name is **`<name>`**.*

When issued in an asset's directory, the ***`<noun>`*** and ***`<name>`*** arguments are implied and not required.

The verbs are:

`create` — create the named asset, creating files and directories as required, and creating any skeleton files for future editing.

`delete` — remove all directories and files associated with this named asset

`build` — build the asset(s), running appropriate tools to create the binary files

`clean` — remove all the generated and compiled files for the asset(s)

`show` — display information about assets in registered projects and the current build environment (preliminary)

`[un]register` — register/unregister a project in its registry

`[un]set` — set/unset the registry used by the current project

`run` — execute the unit test or application

`refresh` — manually regenerate the metadata associated with the project

See the **`ocpidev(1)`** man page and its associated man pages for verbs (named **`ocpidev-<verb>`**, for example, **`ocpidev-create(1)`**) and noun asset types (named **`ocpidev-<noun>`**, for example, **`ocpidev-project(1)`**) for detailed syntax descriptions and usage examples.

15.1 Assets Managed by `ocpidev`

Below are two tables summarizing all the types of assets that the **`ocpidev`** command operates on, and which nouns apply to it. The first table is for assets that are not HDL specific. The second is for HDL-specific assets located under the **`hdl`** sub directory.

Table 9: Non-HDL Asset Types (Nouns) for the `ocpidev` Command

Name/ Noun	In Library	Description
`application`	no	A component application, specified either in XML or C++
`applications`	no	All applications in a project, in its **`applications`** directory
`library`	no	A component library.
`project`	no	A project containing all other asset types.
`properties`	yes	A properties XML file, which can be at the project or library level.
`protocol`	yes	A protocol specification XML file, which can be at the project or library level.
`registry`	no	A registry for projects to import/export dependencies from/to.
`component`	yes	A component specification XML file (OCS), which can be at the project or library level.
`test`	yes	A unit test suite for a component specification.
`worker`	yes	A worker, that implements a component spec.

The following table describe HDL asset types. When these assets are described as being in an HDL library, it means in one of the fixed HDL libraries in a project (**`hdl/adapters`**, **`hdl/cards`**, and **`hdl/devices`**) or the **`devices`** library within an HDL platform's directory.

Table 10: HDL Asset Types (Nouns) for the `ocpidev` Command

Name/ Noun	In Library	Description
`hdl assembly`	no	An assembly of HDL workers, used to build HDL containers
`hdl assemblies`	no	All the assemblies in the project or in an HDL assemblies directory.
`hdl card`	yes	A card specification XML file, which can be at the project or HDL library level.
`hdl device`	yes	A HDL device worker, in an HDL component library (**`adapters`**, **`cards`**, **`devices`** or a HDL platform's **`devices`**).
`hdl platform`	no	An HDL platform worker, in an HDL platforms directory, including its platform configurations.
`hdl platforms`	no	All HDL platforms in a project or in its **`hdl/platforms`** directory.
`hdl primitive core`	no	A HDL primitive core in a project or in its **`hdl/primitives`** directory

Name/ Noun	In Library	Description
`hdl primitive library`	no	A HDL primitive library in a project or in its **`hdl/primitives`** directory
`hdl primitives`	no	All HDL primitives (cores or libraries) in a project.
`hdl signals`	no	A signals specification XML file, which can be at the project or HDL library level.
`hdl slot`	no	A slot type specification XML file, which can be at the project or HDL library level.
`hdl subdevice`	yes	An HDL subdevice worker, that supports other HDL device workers, in an HDL devices library.

16 Environment Variables used in Component Development

All OpenCPI environment variables start with the prefix `OCPI_`. While some defined variables apply only to certain authoring models or targets, a master list is kept here. Some may be described as documented elsewhere.

There also may be environment variables starting with `OCPI_` that are used internal to OpenCPI and not documented for users.

In the table below, environment variables that are boolean options have the value `0` or `1`. When they are unset, or set to the empty string, it is equivalent to the value `0`.

In the table below a ***list*** means a white-space-separated list of items. A ***path*** means a colon-separated list of directories. All these variables have commonly used defaults so that most users have very few settings.

Most of the items are set at installation time, several are simply convenience variables derived from others. The ones that a developer might actually set *during* development are shaded.

Name	When set?	Data type	Description
OCPI_ALTERA_DIR	install	directory	Top level directory for all Altera tools. Default: **`~/altera`** or **`/opt/Altera`**
OCPI_ALTERA_VERSION	install	directory	Directory under **`OCPI_ALTERA_DIR`** for current/desired Altera tools version. Default is highest numeric version present under **`$OCPI_ALTERA_DIR`**
OCPI_ALTERA_TOOLS_DIR	install	directory	Directory for using Altera tools. Default: **`$OCPI_ALTERA_DIR/$OCPI_ALTERA_VERSION`**
OCPI_ALTERA_LICENSE_FILE	install	file	Altera license file. Default: **`$OCPI_ALTERA_DIR/Altera-License.lic`**
OCPI_ALTERA_KITS_DIR	install	directory	Directory for Altera development kits. Default is **`$OCPI_ALTERA_TOOLS_DIR/kits`**
OCPI_ASSERT	build	boolean	Enable assertions in any language (when != 1, for C and C++, sets NDEBUG). Default: **`1`**
OCPI_CDK_DIR	install	directory	The root of the installed CDK. Default: **`/opt/opencpi/cdk`**
OCPI_CFLAGS	build	list	Flags when compiling C code. Default set per target, but may be overridden.
OCPI_CXXFLAGS	build	list	Flags when compiling C++ code. Default set per target, but may be overridden.
OCPI_DEBUG	build	boolean	Controls debug logging, etc. For C and C++, enables “-g” also, or if not set, -O2. Is defined as a preprocessor macro for core software or executables. Default: **`1`**
OCPI_DYNAMIC	build	boolean	For main programs and software libraries: use dynamic linking and dynamic libraries. For software components, build for use in dynamic executables. Currently only **`0`** is supported.
OCPI_LIBRARY_PATH	run	path	Runtime search path for binary artifacts. Default: for applications in projects: current and depended-on project artifacts Default: for unit tests: a unit test's own generated artifacts and the core project's artifacts
OCPI_PREREQUISITES_DIR	install	directory	Location for using prerequisites, default locations for building and installing prerequisites. Default: **`/opt/opencpi/prerequisites`**
OCPI_PROJECT_REGISTRY_DIR	build	directory	Location of current project registry, overriding the default in the OpenCPI installation.
OCPI_SMB_SIZE	run	number	To override size in bytes of data plane endpoints. Default: **`100000000`**.
OCPI_TOOL_PLATFORM	build	string	The platform on which the current environment is executing. Automatically determined by the **`platforms/getPlatforms.sh`** script.
OCPI_TARGET_PLATFORM	build	String	The default target software platform to build for.
OCPI_XILINX_DIR	install	directory	Top level directory for all Xilinx tools. Default is from Xilinx: **`/opt/Xilinx`**
OCPI_XILINX_VERSION	install	directory	Directory under **`OCPI_XILINX_DIR`** for current/desired Xilinx tools version. Default is highest numeric version present under **`$OCPI_XILINX_DIR`**.
OCPI_XILINX_TOOLS_DIR	install	directory	Directory for using Xilinx tools. Default: **`$OCPI_XILINX_DIR/$OCPI_XILINX_VERSION`**
OCPI_XILINX_LICENSE_FILE	install	file	Xilinx tool license file. Default is: **`$OCPI_XILINX_DIR/License.lic`**
OCPI_XILINX_LABTOOLS_DIR	Install	directory	Where a Xilinx Lab Tools installation is, default is **`$OCPI_XILINX_TOOLS_DIR`**.

17 JSON Format Produced by the `ocpidev show` Command

All nouns (asset types) used with the **`ocpidev show`** command have the option to output information in a JSON format and the format of the JSON output will be different for each noun. This appendix details the format of JSON output for each noun.

If the data type of an asset in the JSON file is a JSON String this will be the directory of the asset, otherwise it will be a dictionary with an entry of **`directory`** that is the directory of the asset. (It is possible that Strings may be changed to dictionaries as needed as the interface matures.)

Libraries can override the default package-id and this can cause multiple libraries within the same project to have the same package-id. In order to express this in a JSON file, numbers are added to the string of the package-id if there are more than one library with the same package-id. E.g. the core project has two libraries (**`hdl/adapters`** and **`components`**) with the package-id of **`ocpi.core`**. The libraries are named **`ocpi.core`** and **`ocpi.core:1`** in the JSON file for thge project. It is up to the JSON data consumer to parse these package-ids and use them in the correct way.

Levels of verbosity are used to ask **`ocpidev`** for more information, the default being less information. The rule of thumb for what is output when showing the minimal information for an asset is to not include anything about the assets that are contained by it (i.e. a library contains components etc.). Each verbosity level that is added will show the information for one more level of contained assets. E.g. very verbose (-vv) for a project will show information on the project, libraries in the project, and workers/components/test in each library but verbose (-v) will not show workers/components/test in each library.

17.1.1 `ocpidev show project --json`

```
{
  "project": {
    "package": <String – Project package-id>,
    "dependencies": <Array of strings – package-ids of projects to
                                        depend on>,
    "directory": <String – Project directory>
  }
}
```

17.1.2 *ocpidev show project -v --json*

```
{
  "project": {
    "directory": <String – Project directory>,
    "package": <String – Project package-id>,
    "dependencies": <Array of strings – package-ids of projects to
                                        depend on>,
    "libraries": {
      <String- package id of library>: <String- directory of
library>,
      … more libraries
    },
    "components": {
      <String- name of component>: <String - directory of library>,
      … more components
    },
    "platforms": {
      "rcc": {
        <String- name of rcc platform>: <String - directory of rcc
                                         platform>,
        … more rcc platforms
      },
      "hdl": {
        <String- name of rcc platform>: <String - directory of hdl
                                         platforms>,
        … more hdl platforms
      }
    }
  }
}
```

17.1.3 *ocpidev show project -vv --json*

```
{
  "project": {
    "directory": <String – Project directory>,
    "package": <String – Project package-id>,
    "dependencies": <Array of strings – package-ids of projects to
depend on>,
    "libraries": {
      <String- package-id of Library>: {
        "package": <String- package-id of Library>,
        "directory": <String – Library directory>,
        "components": {
          <String- component name>: <String – Components directory>,
          … more components
        },
        "tests": {
          <String- test name>: <String – Test directory>,
          … more tests
        },
        "workers": {
          <String- worker name>: <String – Worker directory>,
          … more workers
        }
      },
      … more libraries
    },
    "primitives": {
      <String – name of primitive>: <String – Primitive directory>,
      … more primitives
    },
    "components": {
      <String- name of component>: <String - directory of library>,
      … more components
    },
    (Continued on next page)
(Continued from previous page)
    "platforms": {
      "rcc": {
        <String- name of rcc platform>: <String - directory of rcc
platform>,
        … more rcc platforms
      },
      "hdl": {
        <String- name of rcc platform>: <String - directory of hdl
platforms>,
        … more hdl platforms
      }
    }
  }
}
```

17.1.4 ocpidev show tests --local-scope --json

The **--local-scope** option will act upon the project that the command is executed in so this command shows all the tests in the specified project.

```
project": {
    "libraries": {
      <String-Library name>: {
        "directory": String – Library directory>,
        "package": String – Library package-id>,
        "tests": {
          <String - Test name>:<String - Test directory>,
          … more tests
        }
       … more libraries
      }
}
```

17.1.5 ocpidev show platforms/registry --json

```
{
  "rcc": {
    <String-platform name>: {
      "target": <String-target associated with this platform>,
      "package_id": <String - platform package-id>,
      "directory": <String - platform directory>
    },
     … more rcc platforms
    }
  },
  "hdl": {
    "xsim": {
     "target": <String - target associated with this platform>,
      "package_id": <String - platform package-id>,
      "directory": <String - platform directory>,
      "part": <String-part associated with platform>,
      "built": <Boolean - if this platform is built>,
      "tool": <String - FPGA tool associated with this platform>,
      "vendor": <String - FPGA vendor associated with this platform>
    },
    … more hdl platforms
  }
 }
```

17.1.6 *ocpidev show targets --json*

```
{
  "rcc": {
    <String - platform name>: {
      "target": <String-target name>
    },
   … more rcc targets
  },
  "hdl": {
    <String - vendor name>: {
      <String - target name>: {
        "tool": <String - FPGA tool associated with this target>,
        "parts": <Array of Strings - FPGA parts associated with this
                                     target>
      },
   … more hdl targets
  },
  … more hdl vendors
}
```

17.1.7 *ocpidev show component --json*

```
{
  "directory": <String - directory of component>,
  "name": <String - name of component>,
  "package_id": <String - package-id of component>,
  "ports": {
    <String- name of port>: {
      "protocol": <String - name of protocol for port>,
      "producer": <String - "0" or "1" for direction of port>
    },
    … more ports
  },
  "properties": {
    <String - name of property>: {
      "accessibility": {
        "volatile": <String - "0" or "1">,
        "readback": <String - "0" or "1">,
        "parameter": <String - "0" or "1">,
        "initial": <String - "0" or "1">,
        "writable": <String - "0" or "1">,
        "padding": <String - "0" or "1">
      },
      "name": <String - name of property>
      "type": <String - data type of property>
    },
    … more properties
  }
}
```

17.1.8 *ocpidev show worker --json*

```
{
  "directory": <String - directory of worker>,
  "name": <String - name of worker>,
  "package_id": <String - package-id of worker>,
  "ports": {
    <String- name of port>: {
      "protocol": <String - name of protocol for port>,
      "producer": <String - "0" or "1" for direction of port>
    },
    … more ports
  },
  "properties": {
    <String - name of property>: {
      "accessibility": {
        "volatile": <String - "0" or "1">,
        "readback": <String - "0" or "1">,
        "parameter": <String - "0" or "1">,
        "initial": <String - "0" or "1">,
        "writable": <String - "0" or "1">,
        "padding": <String - "0" or "1">
      },
      "name": <String - name of property>
      "type": <String - Datatype of property>
    },
    … more properties
  }
  "slaves": {
    <String - name of slave worker>: {
      "name": <String - name of slave worker>
    }
    … more slaves
  }
}
```

17.1.9 *ocpidev show registry/projects --json*

```
{
  "registry_location": <String - directory of registry>,
  "projects": {
    <String - package-id of project>: {
      "exists": <Boolean - if project exists on the file system >,
      "real_path": "<String - directory of project>
    },
    … more projects
}
```

17.1.10 ocpidev show workers --global-scope --json

```
{
  "projects": {
    "ocpi.assets": {
      "package_id": <String - package-id of project>,
      "directory": <String - directory of project>,
      "libraries": {
        <String - package-id of library>: {
          "package_id": <String - package-id of library>,
          "directory": <String - directory of library>,
          "workers": {
            <String - name of worker>:<String - directory of
                                                  worker>,
            … more workers
          }
          … more libraries
        },
  },
  … more projects
}
```

17.1.11 *ocpidev show components --global-scope*

```
{
  "projects": {
    "ocpi.assets": {
      "package_id": <String - package-id of project>,
      "directory": <String - directory of project>,
      "components": {
         <String - name of component>: <String - directory of
                                        component>,
         … more components
       }
      "libraries": {
        <String - package-id of library>: {
          "package_id": <String - package-id of library>,
          "directory": <String - directory of library>,
          "components": {
            <String - name of component>: <String - directory of
                                           component>,
            … more components
          }
          … more libraries
        },
  },
  … more projects
}
```

18 Glossary of Terms

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

18.1 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.

18.2 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***.