OpenCPI
Component Development Guide

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

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 Overview	OpenCPI	https://github.com/opencpi/opencpi/raw/2017.Q2/doc/pdf/OpenCPI_Overview.pdf
OpenCPI RCC Development Guide	OpenCPI	https://github.com/opencpi/opencpi/raw/2017.Q2/doc/pdf/OpenCPI_RCC_Development.pdf
OpenCPI HDL Development Guide	OpenCPI	https://github.com/opencpi/opencpi/raw/2017.Q2/doc/pdf/OpenCPI_HDL_Development.pdf
OpenCPI Application Development Guide	OpenCPI	https://github.com/opencpi/opencpi/raw/2017.Q2/doc/pdf/OpenCPI_Application_Development.pdf

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 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).

References to 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, make and shell level tools, and scripts that support the development process. The CDK relies on several conventional tools, including GNU Make, and other basic POSIX command-line tools.

The CDK also includes tools specific to OpenCPI that support heterogeneous code generation and component testing. These tools are usually used indirectly, using the provided makefile scripts to build component libraries of workers (heterogeneous implementations), and, when applicable, building workers in each of the available authoring models. 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 (for FPGAs) and simulation tools (e.g. Xilinx XST and Isim, 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 creating component libraries.

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: (or *configuration property*) are 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 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 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 respective 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 ***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 an implementation.

In addition to a worker having their own directory, they are typically developed in a component library (a collection of workers). The worker directories are then 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.

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 of the component spec **`xyz-spec.xml`** would be in the subdirectory **`xyz.hdl`**. Note that these names "xyz" are not required to be the names that occur in the programming language source files (e.g. C, C++, Verilog, etc.), although that is usually the simplest.

An **`xyz.test`** directory, at the same level as the worker directories, should be 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, usually in a library, by using the **`ocpidev`** tool, 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. The initial files created that are then edited as necessary include:

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

All three of these file types have initial, automatically generated, skeletal contents that may be subsequently edited by the developer as required. Frequently only the source code files require editing. These files are described in detail below.

The OWD file is an XML file that describes the worker itself, by internally referring to an OCS and including implementation-specific attributes needed by the framework. The second file is a "make" file that describes how the worker is built, and the third is the initial source code file for the worker's actual logic.

The component specification (OCS) for the worker contains the description of the component's external behavior. These will exist in all implementations (workers) that reference the component specification. The OWD adds information about a particular implementation 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.

The worker building process invokes built-in scripts and makefiles, which automatically create and populate 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-<target>`** 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 ***`linux-c6-x86_64`*** for CentOS 6 Linux running on a 64-bit x86 processor. Both types of directories are files resulting from the build process and are removed by **`make clean`**, as they are always regenerated and should never 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 three types of worker files just introduced.

3.1 A Simple Example Worker

Here is a simple example of a software worker written in C++. The component's function is to add a constant, called the biasValue, to each unsigned 32 bit integer at its input, and put the value at its output, one message at a time. The component specification XML file, 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, OPS file, indicated by the **protocol** attributes in the OCS, would be found in the file **specs/u32-proto.xml**, and contains:

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

For the **bias.rcc** worker as created by **ocpidev**, which implements 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'>
</RccWorker>
```

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 length ? RCC_ADVANCE : RCC_ADVANCE_DONE;
  }
};
```

The **Makefile** in the worker's directory, automatically generated, would be:

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

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

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

For a detailed explanation for using 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 XML metadata used and manipulated by OpenCPI tools and OpenCPI component developers.

The use of the term component is to encompass the functionality and abstract interface aspects of a model. The term worker is meant for particular implementation of a component written, authored, using a programming language source code.

The definition of a authoring model can be casually 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 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 a processor
- Be able to switch, 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, component implementations, into an application using a multiplicity of authoring models and processing technologies.

An OpenCPI Authoring model consists of these specifications:

- An XML document, structure/schema/definition, to describe the aspects of the implementation that are specific to the authoring model being used and needed by tools and runtime infrastructure software.
- Three sets of 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 used for workers to consumer/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 used by the worker to obtain various services locally available on the processor on which the worker is running.

- Each authoring model also specifies how a worker is built (compiled, synthesized, linked) and packaged, to be ready for execution in an application.

4.1 Requirements for All Authoring Models

- Enable/support well-defined data plane interoperability with other authoring models
- Define its OpenCPI Worker Description (OWD) XML format.
- Define programming language interfaces for control, data, and local services.
- Must 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***, an application launch and control API described in the ***OpenCPI Application Development Guide***.

We use the term ***Control Plane*** to encompass the various aspects of 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 uses 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 are shown in the following diagram.

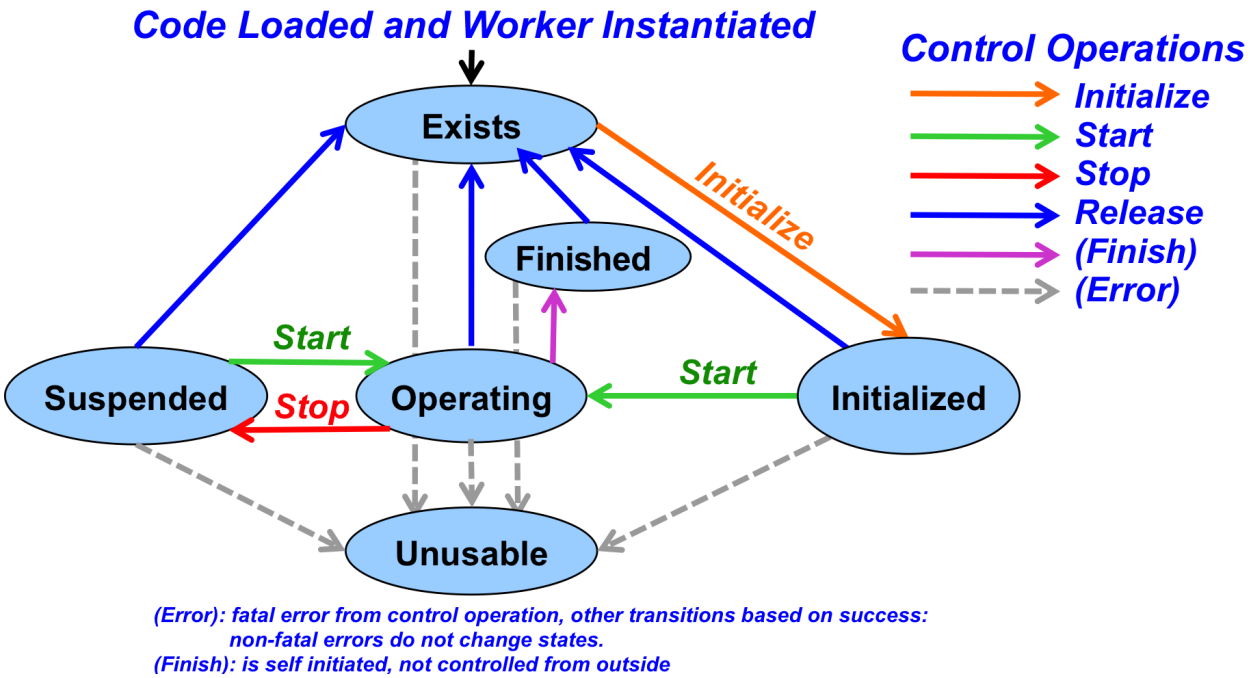

Diagram 1: Control States and Operations

4.2.2 LifeCycle Control Operations

Control operations have default implementations that only perform the state transitions: workers only need implement the states that have custom behavior. A good example is the *initialize* control operation. If the worker has no runtime initialization to perform, it can avoid any implementation of this state (or even an empty "stub"). Each authoring model describes which control operations must have implementations.

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 (except **unusable),** after release*	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. Follows ***initialize.*** In a stable state "ready to start doing work", not "operational". Properties can be read and/or written.
Operating	Initialized, Suspended	Stop, (done), 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 consumer 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 consumer at data interfaces. Properties can be read or written. Entered autonomously.
Unusable	All (fatal errors)	none	Fatal error state, may not be ever reusable without reloading (container-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 workers 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 must 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 different 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 read property values

written by control software, and write 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 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 the model is based on a construct called ***Containers***. Containers supply threads and execute software-based 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 and is detailed in the individual sections for each.

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 entered. Worker execution is a series of “runs” initiated by the container. The 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 the availability of its input buffers, with data, or output buffers, with space, at a worker's ports or the passage of time/ The worker’s run method executes some processing tasks 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.

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.

4.3.1 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 satisfies, 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 they 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.

4.3.2 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 which 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.

4.3.3 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 features list previously 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 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 OWD, as a **`ComponentSpec`** element.

The OCS describes two things: (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 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 implementations 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 and 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 component-global 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 scripts. This also enumerates what files must be exported when applications use components in the library.

The spec files, and if necessary 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.

5.1 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.

5.1.1 Name Attribute *of the ComponentSpec* element

The optional **`Name`** attribute of the component specification provides a name that is unique within its name scope. 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. That means this attribute is optional when the **`ComponentSpec`** is the top level element of a file. 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.

5.1.2 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.

5.2 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.

5.3 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 use of 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.

5.3.1 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.

5.3.2 Type Attribute of Property (and Member) Elements

The **`Type`** attribute specifies the data type of the property. The legal types are listed in table Data Types for Properties. When the **`Type`** attribute has the **`String`** value, the **`StringLength`** attribute must also be supplied. 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 the type is **`Struct`**, the **`Property`** element itself has **`Member`** sub-elements that indicate the types of the members of the **`struct`** property. No **`struct`** members can be of type **`Struct`**. 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.

All types have a maximum length and Properties cannot have unbounded length.

5.3.3 StringLength Attribute of Property Elements

The **`StringLength`** attribute is required when the **`Type`** attribute is **`String`**, and indicates the maximum length, excluding any null termination, string value that this property can hold.

5.3.4 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.

5.3.5 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.

5.3.6 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.*

5.3.7 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.

5.3.8 Accessibility Attributes of Property Elements

The attributes described here specify what property accesses are allowed under what conditions. They are all Boolean attributes and all have the default value of false. At least one of these attributes *must* be set to true.

5.3.8.1 Readable Attribute of Property Elements

When this attribute is true it indicates that this property can be read by control software but that the workers *cannot* and *will not* change the value. This means that control software can read back what it wrote (if it is also **`writable`** or **`initial`**). If set to false and the volatile attribute is false, attempts to read the property value at any time may result in an error. If **`readable`** and *not* **`writable`** nor **`initial`** is specified, the property value is a constant in the implementation. It is an error to set both **`readable`** and **`volatile`**.

Since control software caches all property values that are written (either initially or dynamically during execution), setting this attribute is *not necessary* to simple have control software be able to read back what was written (e.g. in **`ocpirun`** with the dump option, or in an ACI main program that reads property values). Thus this attribute is only needed when either:

- the property is not a parameter and is never written: i.e. the worker's value is readonly, not known in advance, but a constant inside the binary worker.

— or —

- the ability to read back (uncached) from the worker what was previously written is considered more valuable (for debug?) than the resource savings of avoiding such a readback capability (especially in HDL workers).

5.3.8.2 Volatile Attribute of Property Elements

This attribute indicates, when true, that the property is readable and that its value may be changed during execution by the worker, without it being written by control software. When neither readable nor volatile is true, attempts to read the property value may result in an error. Only one of the attributes **`readable`** or **`volatile`** should be set to true. When a property is volatile if cannot be cached by control software.

5.3.8.3 Writable Attribute of Property Elements

This attribute indicates whether this property can be written both before and while the worker is in the Operating state. If set to false, attempts to write the property value during operation will result in an error.

5.3.8.4 Initial Attribute of Property Elements

This attribute indicates whether this property can be set during initialization. If set to false and it is also not writable, attempts to specify an initial property value will result in an error. A property that has Initial as true, but writable as false, can be set at initialization-time, but not at run-time. Only one of the attributes **`writable`** and **`initial`** may be set to true. Initial properties are set prior to any worker being started, but *after* each worker is itself locally instantiated and initialized.

5.3.8.5 Padding Attribute of Property Elements

This attribute is used for properties that exist only for padding purposes, and are otherwise inaccessible. The use case for this is when properties need to match a register set and may require exact offsets for each property. These padding properties are not accessible and are marked with the **`padding`** attributes having the value of true. It is rarely if ever used in a component specification, but may be used in a worker description.

5.3.8.6 The Order in which Properties are Written during Startup

Initial or default property values are written prior to workers being started but after they have been initialized, with no inter-property ordering 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 startup action depends on a combination of the values of multiple different properties, this cannot be done prior to the start operation.

Property write notifications (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.

5.3.9 Parameter Attribute of Property Elements

This attribute indicates that the property's value is used at compile/build time when source code is processed into a binary artifact to be loaded and executed at run-time. It also allows the default value expressed in this property to be used by name elsewhere in the OCS and OWD XML files.

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

1. A convenience variable for defining other attributes 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. A performance/footprint enhancement for compiled constants vs. runtime settable values.

The convenience usage (1) allows properties defined as parameters to be used in expressions for the value of **`stringlength`**, **`sequencelength`**, **`arraylength`**, **`arraydimensions`** or **`default`** attributes. For the allowable syntax of such expressions see the attribute expressions section 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) is to allow a component specification to indicate a property as **`initial`** with (**`parameter == false`**), and allow some *implementations* to actually compile in the value for efficiency purposes. The component spec just says **`initial`**, allowing implementations to *either*:

- Have the property be configurable to any value at execution 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 specifies a particular value, the worker with the compiled-in value can only be used if

the requested value matches the compiled-in value. Workers with the non-parameter property can be used with any value. This second usage of the **`parameter`** attribute is discussed further in the section on the OWD XML files.

In either case, 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.

5.3.10 Default Attribute of Property Elements

The name of this attribute is **`default`**. This string valued attribute provides a default value for the property for all implementations. It is parsed based on the data type specified in the **`Type`** attribute, and it may be an expression (see the next section). This value is set by the infrastructure at runtime when any implementation is instantiated in the runtime environment, unless an initial property value is specified by the application. The purpose of this attribute is to advertise what value will be used if no initial value is provided by the application. Default values should only be specified for properties which are parameters or whose accessibility is initial or writable. The format of the string value of this attribute is described in the *Property Value Syntax and Ranges* section.

5.3.11 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 properties defined as parameters 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.

5.4 Port Element of ComponentSpec Elements

The component specification defines ports through the use of this ***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 also 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.

5.4.1 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 case insensitive within the component specification.

5.4.2 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 (not needed since it is the default).

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

5.4.3 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 named OPS XML file is expected to contain a **`Protocol`** element at its top level. If the port being described is permissive, meaning it can accept any protocol, then this attribute can be absent. An example of a permissive component is a file writing component that logs any types of messages as 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, several protocol attributes are implied:

- There can be up to 256 operations (opcodes).
- Messages are of unbounded size, up to 64KB.
- Messages may be of zero length.
- The granularity of messages is a single byte.

5.4.4 Optional Attribute of Port Elements

This is the attribute whose name is optional and is also 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.

5.4.5 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>
```

6 Property Value Syntax and Ranges

This section describes how property values are formatted to be appropriate for their data types. Property values occur in the **`default`** attribute of property elements described above. This syntax is also used when property values are specified in the worker Makefile described below. The type names presented 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.

6.1 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.

6.2 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.

6.3 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.:

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

6.4 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.

6.5 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.

6.6 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 white space. To be interpreted this way, the first character must be a double quote. Two double quotes can represent an empty string.

6.7 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.

6.8 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.

6.9 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.

6.10 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.

6.11 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 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 might be:

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

6.11.1 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 target data type.
- 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.

6.11.2 String Values

String constants (using double quotes) can be used in expressions as well as string-typed parameters. 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 or not). 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 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 as they are used by both sides of a connection. In a connection between component ports the specs of both components, 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***. Protocol summary attributes can also be present when messages are specified, and can 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, or specified directly as attributes of the protocol element. When messages are specified, summary attributes override the values inferred from the message specifications. When no messages are specified, the summary attributes are used by the OpenCPI code generation tools and runtime environment to determine certain behaviors, rather than having the attributes inferred from message specifications.

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. This 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.

7.1 Protocol Element as Top-level Element.

The **Protocol** element is a top-level element in a separate file whose name is the value of the **Protocol** attribute in a port element in an OCS. It specifies the message protocol used at a port. The protocol will likely be reused across a variety of components and interfaces 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.

7.1.1 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.

7.1.2 Protocol Summary Attributes

Other attributes of the protocol element are normally inferred from the **Operation** elements in the protocol as defined below and are rarely used explicitly. Under some circumstances they may be used to override the inferred values or they may be specified in the absence of **Operation** elements altogether.

This table defines the extra protocol summary attributes that are normally inferred from examining **Operation** elements.

Table 4: Protocol Summary Attributes

Name	Type	Description
NumberOfOpCodes	**ulong**	Number of message types
DataValueWidth	**ulong**	Size in bytes of smallest unit of data in any message
DataValueGranularity	**ulong**	Minimum number of data values in any message.
ZeroLengthMessages	**bool**	Are any messages zero length?
MaxMessageValues	**ulong**	Maximum number of data values in any message, or zero if not bounded
VariableMessageLength	**bool**	Can messages be different lengths?
DiverseDataSizes	**bool**	Are there different size data values?
UnBounded	**bool**	Do any messages have unbounded length?
DefaultBufferSize	**ulong**	Buffer size for ports using this protocol even if protocol is unbounded. Default is zero when protocol is unbounded.

7.1.3 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.

7.1.3.1 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.

7.1.3.2 Argument 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 a configuration property 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.

7.2 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>
```

7.3 Message Payloads on Data Ports

The message payload for each operation 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 is 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 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.

8 Worker Descriptions in OWD XML Files

Each worker directory contains an XML file describing the worker and references 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, common across authoring models, aspects of these implementation description files are described in this section. The OWD files are specific to different authoring models. These differences are described in the respective sections 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 essentially 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. All OWDs have as the top-level XML element an XYZWorker 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. In this case 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 OWD file would be:

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

The following description of the OWD is generic as it is augmented for each authoring model. Actual OWDs are defined for each authoring model, and the top-level element is named according to the authoring model. This section describes aspects common to the OWDs for all authoring models using an inherited schema.

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

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

Below is an example OWD for an HDL worker, 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'
   ---other attributes---
                           >
   ---other child elements---
</HdlWorker>
```

8.1 XML Attributes of the Top-level XYZWorker Element.

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 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 build scripts and makefiles automatically place the specs subdirectory, in the component library's top-level 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 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.

8.1.5 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 bitstream configurations that can operate in both modes. This OWD attribute specifies how the worker's code will work when subjected to these three build scenarios ,as specified by the ***ocpi_endian*** parameter which is present for all workers of all types.

The endian attribute values are described in the following table:

Table 5 – 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	The worker code will respect all three values of the endian parameter. If the parameter is “little” or “big”, the resulting binaries will 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 may have such an option.

8.2 Property and SpecProperty Child Elements in the OWD

Properties specified in the OCS indicate the external configuration interface for all implementations of the same spec. Properties specified in the OWD define additional worker-specific properties, beyond those in the OCS common to all implementations. From the external/application point of view, there are nine valid combinations of readability and writability. The writable aspect are:

- not writable at all
- settable at initialization
- settable during execution, after start

The readable aspect are:

- not readable at all
- readable for a static/unchanging value
- readable for a volatile value that can change during execution.

A worker may have additional properties beyond what is specified in the component spec. Workers may add to the accessibility of an existing property defined in the component spec. E.g. it might make a property readable that was not readable in the component spec. The accessibility added would result in the implementation having a superset of what was required by the component spec.

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

In the OWD, property attributes can occur either in the **`Property`** or **`SpecProperty`** elements here. **`Property`** elements support all the attributes for **`Property`** elements in the component spec as well as any implementation-specific attributes, while **`SpecProperty`** elements may only add a subset of these attributes, as specified below.

8.2.1 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 implementation-specific property, which must not be the same as any **`Property`** element in the OCS.

8.2.2 ReadSync and WriteSync for OWD Property or 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's memory, with no specific synchronization or notification implied. The worker

accesses these values as local memory locations. When these attributes are true, the worker is notified at the time of the access 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 used.

8.2.3 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.

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

8.2.4 Default Attribute for OWD Property/SpecProperty Elements

This **`string`** attribute provides a default value for the property for this implementation. If the **`Default`** attribute is specified in a **`SpecProperty`** element, it is providing a default value for the implementation only. It is not permitted to provide a default value in a **`SpecProperty`** when the property in the OCS already has a default value. This attribute's value is applied consistent with the access attributes, specifically:

- For an initial or writable property, the value is set into the implementation at initialization, if no other initial value is specified in the application.
- For a readable property, the value will be statically available to control software. If it is also writable, the value may be overridden at run-time after initialization.

8.2.5 Parameter Attribute for OWD Property/SpecProperty Elements

This Boolean attribute, in a **`SpecProperty`** element or a **`Property`** element, indicates that the property's value is used at compile/build time when the source code is processed into a binary artifact to be loaded and executed and run-time. Parameter properties are supplied to the compilation process for the worker, in a form appropriate for the language and authoring model used. The actual values supplied use the value specified in the default attribute, except when overridden by values specified in the build directory for the worker. The OWD specifies that the property is a build parameter, and can supply a default value for the parameter if the OCS does not specify a default.

Parameter properties are utilized for other purposes depending on the access attributes and whether the property is in the OCS or only in the OWD.

- When a parameter property is readable, the value supplied to the build process is also available as a readable property at runtime, with a constant value.

- When a parameter property is initial, the binary worker is only considered usable for an application if the value supplied to the build process matches the value specified by the application.

While the component spec defines properties and their initialization-time and run-time accessibility, a worker can further declare that a property is actually a compile time parameter of the implementation. This is not allowed for properties that are dynamically writable at runtime. When a worker has properties that are parameters, 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 it is built for a property value that matches what is requested as 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 worker developer 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 optimization for certain parameter values, and also allow a single worker source code module to be optimized for different values.

Parameter property values are applied to the build process as expected: e.g. by preprocessor symbol definitions for software, generics for VHDL, and parameters for Verilog. An example of a built-in parameter property for a worker is the **`ocpi_debug`** Boolean property.

8.3 Built-in Parameters of All Workers

OpenCPI automatically adds several parameter properties to all workers. The values of these parameters are 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 in the makefile.

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 developers.

8.3.1 The ocpi_debug built-in parameter property

This parameter property, of Boolean type, 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.

8.3.2 The ocpi_endian built-in parameter property

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. This means the binary object created after software compilation does not support multiple architectures.

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.

8.4 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 collocation in the same device or executing elsewhere.

Each authoring model has 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.

8.4.1 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 indicates for which component port the worker is providing additional implementation-specific information.

9 The Worker Makefile

Each worker requires a makefile for building its binary ***artifact***. The default **`Makefile`** is simply:

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

This makefile line indicates to the framework that this directory is for building a worker whose name and authoring model are derived from the name of the worker's directory. The directory will also contain a OWD and one or more source files for the worker's functional code. These will be generated when the worker is created using the **`ocpidev`** tool described in the ocpidev section.

For example if the name of the directory were **`search.rcc`**, then the simplest makefile would assume that the worker description file is in **`search.xml`**. If no language was specified in the OWD, the source code to compile for the worker would be **`search.c`**, since C is the default language for the RCC authoring model. The **`ocpidev`** tool will create the initial worker source file automatically as a skeleton of the implementation. This initial empty worker does nothing, but compiles as a valid worker of the given authoring model. This file can then be edited to add the logic to perform its function.

If additional subsidiary source files are needed to be compiled with the worker source file, the default makefile is modified a add the SourceFiles variable to indicate these other files, e.g. if the file **`utils.c`** is needed in addition to the worker file:

```
SourceFiles=utils.c
include $(OCPI_CDK_DIR)/include/worker.mk
```

If this worker directory is intended to build *multiple* workers into a single binary artifact file, the **`Workers`** variable is added to list the workers to be built into a single artifact e.g.:

```
Workers=w1 w2
SourceFiles=util.c
include $(OCPI_CDK_DIR)/include/worker.mk
```

The three files, **`w1.c`**, **`w2.c`**, and **`util.c`** will be compiled together to form the worker binary file implementing **`w1`** and **`w2`**, as described by **`w1.xml`** and **`w2.xml`**.

When there is only a single worker, the value of the **`Workers`** variable is inferred from the name of the directory.

Running the command **`make clean`** in the worker directory causes any generated code skeleton file to be removed if it has not been edited since it was generated. Even after the initial skeleton is edited, a copy of the initial skeleton will still be left in the **`gen`** subdirectory for reference purposes with the name **`gen/search-skel.c`**. This can be useful to look at if changes in the OWD, or OCS or OPS, cause changes to the generated skeleton. This applies to all authoring models and languages, not just RCC and C/C++

The worker makefile can also have other makefile variable settings specific to the authoring model. Such variables are described in the given authoring model documents. In Linux the appropriate binary for an RCC worker is a shared object **`.so`**

file. For an HDL worker it would be an **`.ngc`** file for HDL workers using Xilinx XST synthesis tool.

The worker **`Makefile`** is commonly left as created by **`ocpidev`**, but there are cases where some extra settings need to be present. The table below lists the variables settable in the makefile that are common to all authoring models.

Table 6: Variables in Worker Makefiles

Variable Name in Makefile	Override/ augment by library or project	
`SourceFiles`	N	A list of *additional* source files for this worker.
`Libraries`	Y	A list of primitive libraries built elsewhere. If a name has no slashes, it is assumed to be in the OpenCPI CDK installation directory.
`OnlyTargets` **`OnlyPlatforms`**	Y	A list of the only targets/platforms for which this worker should be built
`ExcludeTargets` **`ExcludePlatforms`**	Y	A list of targets/platforms for which this worker should NOT be built
`XmlIncludeDirs`	Y	A list of directories elsewhere for searching for xml files included from the OWD (in addition to the **`../specs`** directory in the component library containing this worker)
`Worker`	N	Name of worker; the default is from directory name
ComponentLibraries	Y	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).

The variables with **Y** in the table above are those than can be specified in the component library's Library.mk or project's Project.mk file to provide a default for all workers in the library or project.

9.1 Parameter Properties in Worker Makefiles

Parameter properties are build-time parameters which have default values specified in the OWD or OCS XML files. Additional or different values are specified through various methods and cause the worker to be built for different combinations of parameter values.

The **`target-*`** directories mentioned above are used to separate the files that result from building for different targets (e.g. **`linux-c7-x86_64`** vs. **`linux-x13_4-arm`**, or [Xilinx] virtex6 vs [Altera] stratix5). When a worker has parameter properties, a different target directory is created for each requested combination of values per requested target. Combinations of compilation target and parameter values are called ***build configurations.***

When a worker is built for non-default values of parameters, the target directory name is extended with a build configuration identifier, e.g.:

```
search.rcc/target-1-linux-c6-x86_64
```

The ***build-configuration*** identifier is a generated numeric value added to the target directory name. To determine the actual build configuration for the parameter values from this identifier, a list of built configurations is placed in the file **`gen/<worker>.build`**, an XML file.

There are two mechanisms to specify parameter values. The simplest case is based on makefile variables. The other, more capable, case is based on a separate XML file to specify any possible build configuration.

9.1.1 Specifying Parameter Values using Variables in Makefiles

To specify the value for a parameter, set a makefile variable whose name begins with **`Param_`** followed by the name of the parameter, such as:

```
Param_xxx=5
```

To specify multiple values for a parameter, makefile variables can be used to specify the values for any parameter for which the worker should be built. A makefile variable of the form:

```
ParamValues_xxx1=v1/v2/v3
```

specifies that the parameter whose name is **`xxx1`**, should be built for the values **`v1`**, **`v2`**, and **`v3`**. The forward slash character is used to separate the values rather than commas, since commas must be used within values when the parameter's data type is an array and/or sequence. This variable specifies these values independent of the values of any other parameter. It is also independent of the target variables that specify target hardware. In the case of the variables specified as:

```
ParamValues_p1=1/2/3
ParamValues_p2=abc/xyz
```

then the worker will be built for each target specified, for all these combinations:

```
p1=1, p2=abc
p1=1, p2=xyz
p1=2, p2=abc
p1=2, p2=xyz
p1=3, p2=abc
p1=3, p2=xyz
```

If the worker was built for two targets, this would result in the worker being built for twelve different build configurations. These are inferred from the settings of parameter and target make variables. These parameter variables can be specified on the command line when calling the makefile and at higher levels of makefiles. To build an entire component library with the p1 parameter set to 2, simply specify this in the top level directory of a component library:

```
make Param_p1=2
```

The syntax of parameter values is the same as described for default values for properties in the Property Value Syntax section, with the limitation that string values cannot have spaces, quotes, commas or backslashes. If those are needed, the XML file method of specifying parameter values must be used.

There are a number of limitations when using makefile variables to specify parameter values:

- Different parameters are independent of each other creating the cross-product of all combinations.
- The value syntax is not generally usable for string values with embedded spaces, commas, quotes or backslashes.
- As with most makefile variable uses, misspelling the name of a makefile variable does not result in any error.

9.2 Specifying Parameters Using Build Configuration XML Files

Using the makefile variables is the simplest way to specify build configurations. This method can result in a combinatorial explosion of builds when there are several parameters that can take several values. A limitation of the makefile variable method is that the parameter values must accommodate makefile syntax. To overcome these limitations, build configurations can be specified in an XML file. This file can indicate specific builds with specific parameter values and specific targets.

The name of this optional file is ***<worker>*.build**, and the top level element is build. If this file exists, it will be used to build configurations instead of any **Param_** variable in the makefile. This method of specifying different build configurations allows the makefile to be left unedited. There are two sub elements that can occur in this top level XML element:

- ***parameter*** to specify parameter values for all configurations
- ***configuration*** to specify a single configuration with its own parameter values.

9.2.1 Parameter elements in the <worker>.build file

This element specifies a value for the parameter. The **name** attribute specifies which parameter, and must match the name of the parameter property in the OWD (case insensitive). The values for the parameter can be specified one of two ways:

- The **value** attribute can specify the single value.
- The **valueFile** attribute can specify a value in a file.

When the parameter element is at the top level of the ***<worker>*.build** file, it is specifying a value common to all configurations. If the same parameter is mentioned as a subelement of a configuration element, it overrides any top level value 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>
```

9.2.2 Configuration elements in the build.xml file

The **configuration** element has a 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.

10 Component Libraries

OpenCPI components are developed in libraries. OpenCPI component-based applications are defined as a composition of components, and the components are drawn from component libraries at execution time.

A component library is a directory that contains:

- Component specifications, OCSs, OPSs in a **`specs`** subdirectory.
- Component implementations with workers, each in its own subdirectory.
- Component tests in *.test subdirectories.
- The makefile for the component library.
- When built, an exports subdirectory call **`lib`**, which contains the binaries and metadata files required to use components in the library.

A component library has two forms: source and binary. The source form is for component developers, and the binary form is for application developers and users. The binary form is the result of building the source library and exporting the results to a binary package. The package can then be installed onto a system such that the assets can be found and used by applications.

The exported version of a component library contains a combination of binary artifacts and XML files. The binary files can be a collection of heterogeneous built workers for various technologies.

Distribution of a library to an application developer or user requires the contents of the **`lib`** subdirectory for installation. It is recommended that the recipient should install this distribution to the **`/opt/opencpi/lib/xxx`** directory. Here the **`xxx`** is the name of the library. Note that OpenCPI component libraries in binary form can contain compilations for different operating systems, FPGA chips, or CPUs in the same binary directory tree. In OpenCPI libraries are accessed dynamically at runtime. To access this installed library the OpenCPI environment variable **`OCPI_LIBRARY_PATH`** needs to be set. This defines the search path, separated by colons, for the framework to find assets. **`OCPI_LIBRARY_PATH`** acts in a way analogous to **`LD_LIBRARY_PATH`** on Linux systems.

For larger development efforts it is recommended to review the Developing OpenCPI Component 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.

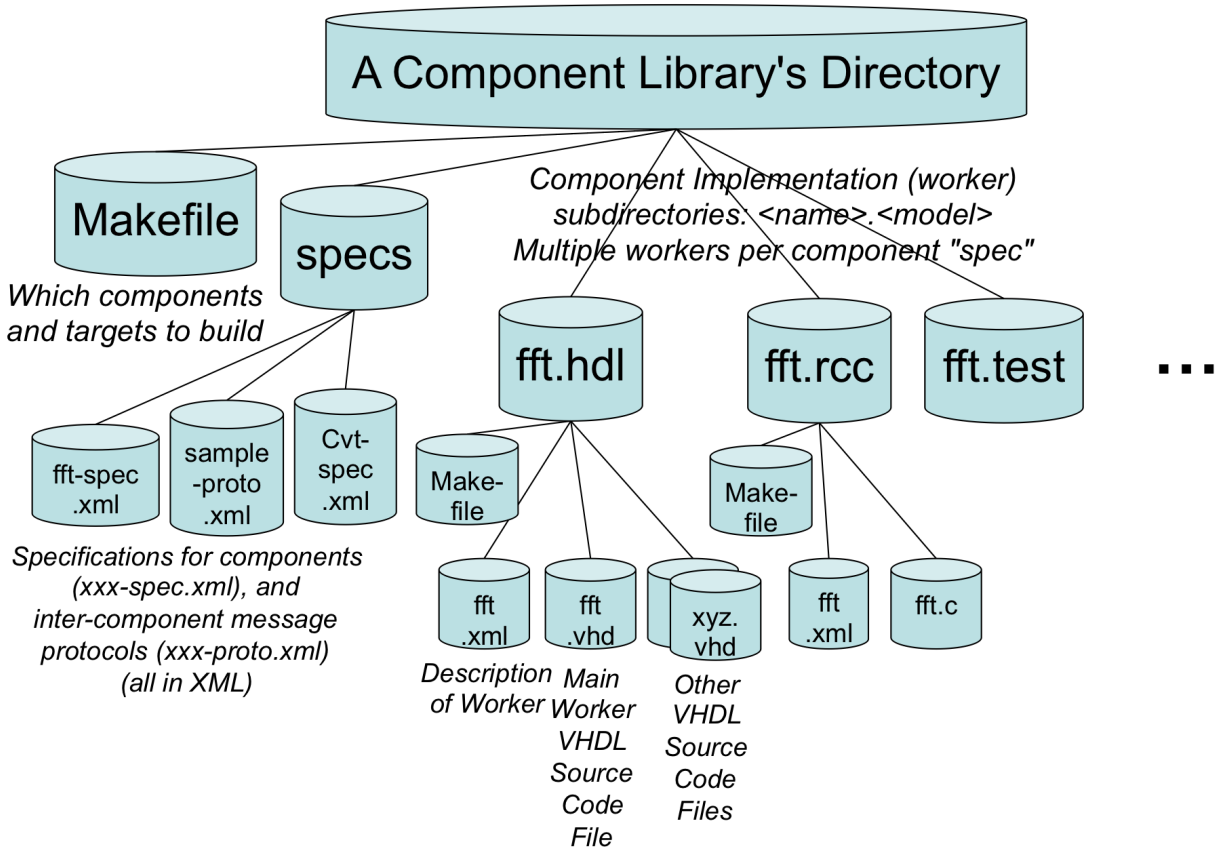

Figure 1: Component Library Directory Structure

10.1 The Component Library Makefile

The makefile in the top-level directory of the component library, usually generated automatically by the **`ocpidev`** command. The last line in the file establishes it as a makefile for an OpenCPI component library and should be:

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

The OCPI_CDK_DIR variable must be set in the environment to point to the OpenCPI CDK installation. The most important variable in this file is **`Workers`**, which is a list of which worker subdirectories to be built for this component library. When the **`Workers`** variable is not set at all, it indicates that all subdirectories of 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 variable at all:

If you want to temporarily avoid building some workers in the library, you can set this variable to only the ones you want to build, so any others are ignored.

If you want to specify the order in which the workers are built, you can set this variable 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 proxy 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***).

In order to avoid name space collisions when using multiple component libraries, there is also a “package” variable that specifies what namespace should be used for the specs and workers in this library. The default package name is **`local`**. For libraries used outside the local organization, this default should be changed. The recommended package naming policy should be something like the reverse internet domain name used for Java classes. E.g.:

```
Package=com.xyz-corp.siglib
```

If the component library is in a project (see the section on projects), a package name prefix can be specified at the project level with the package name for the library being the project's prefix followed by the library's name. Thus the *project* prefix might be **`com.xyz-corp`**, and this library's name might be **`siglib`** which would have the same result as the example above.

The package name **`ocpi`** is reserved for OpenCPI component specifications.

Finally, for each authoring model, there may be a (default) list of *platforms* to build for. I.e. for the RCC authoring model, the variable **`RccPlatforms`** would be set to a list of platforms to build all RCC workers for. 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. Other software targets would use cross-compilers. These target variables can always be overridden on the command line or in the project's **`Project.mk`** file.

Other non-software authoring models (for processors that will never be the one running the tools), have other default platforms (described in documents for the authoring model).

Software targets use the format: *os-version-processor*. The ***os*** part is something lower case such as **`linux`** or **`macos`**. The ***version*** part is usually an abbreviation of the distributor and major version of the operating system. For **`linux`**, it is typically a letter for a distribution followed by a major number (e.g. “c6” for CentOS 6, “r5” for RHEL 5, “u13” for Ubuntu 13). For **`macos`**, it is simply the major version (e.g. “10.8”). The processor part is a lowercased version of what the **`uname -p`** unix command would print, such as “x86_64”.

Typical examples are **`linux-c6-x86_64`** for 64-bit CentOS 6 Linux on x86, or **`macos-10_8-x86_64`**. 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 desired build targets are the default ones (or specified on the command line), and the package name is the default, then the single “include” line above is sufficient to built a component library.

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

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

10.2 Library Exports

When a component library is built, all the workers are compiled and the binary artifacts (the final result of the worker building process) are created. Different authoring models have many intermediate code and metadata files during the build process, but only a subset of these are required and essential for an application to *use* the component and its workers. The build process for a library creates an ***export*** directory to be used by application developers. The export subdirectory is the external view of the (built) library that could be sent to someone needing to *use* the library, but not to *build* or *modify* it.

The export subdirectory (called **`lib`**), is actually a hierarchy filled with symbolic links to the actual files as built for the component implementations in the library. To export it one might do:

```
tar czfLs ../mycl.tgz /lib/mycl lib
```

This would create a gzip-compressed tar file of the export tree, with symbolic links followed (taking the actual files rather than the links), and changing the top directory in the tar file to be **`mycl`** rather than **`lib`**. Such a file could be expanded in place using:

```
tar xzf mycl.tgz
```

and the resulting directory (called “mycl” in this case) referenced by applications using the OCPI_LIBRARY_PATH environment variable as described in the ***OpenCPI Application Guide***.

While exporting a library is useful and convenient, it may also be more appropriate to export a whole project, which is a larger collection of OpenCPI assets and could contain multiple component libraries. See the Project section below.

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 Makefiles: makefiles in worker directories

Worker Build 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.

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 artifact file *for each build configuration* (combination target and parameter values).

11.1 Creating Workers

A worker is created, either standalone or 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.

Similarly, the command:

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

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

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

When a worker is created, all the worker's XML, makefile and source language files are initially automatically 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 hardware the compilation is targeting. Cleaning (via **make clean**) a worker directory 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 three files in the worker's directory;

1. the **Makefile**
2. the OWD file
3. the skeleton source file

These are the files the developer can edit as necessary. Although frequently the makefile and the OWD XML file do 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 "make 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** make variable in the worker's **Makefile**. In some authoring models and languages, the files listed in the **SourceFiles** variable 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 essentially put at the end of the list automatically.

Some changes to the OWD, OCS, OPS and even makefiles can result in changes that require corresponding changes in the worker's primary source files, 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 makefile, the library 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 OpenCPI provides the **ocpidev** tool to create an empty skeleton of a worker implementation that will in compile, build and execute, doing nothing.

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

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. In most cases, there are standard data types, see Data Types for Properties, for the OpenCPI 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 **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 a whole project. To simply compile new code and locate syntax errors, a worker can be built in the worker's directory, by typing **make**.

The target of a worker build is specified in many 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, but one could be placed in the project's Project.mk file.

There are makefile variables named ***<Model>*Targets** and ***<Model>*Target**, which specify the targets that should be built. The ***<Model>*** is a capitalized version of the

authoring model of the worker, e.g. Rcc, or Hdl. The plural version can be set to a list of targets, while the singular one must be set to one target. These variables can be set in several ways:

- On the **make** command line, like **HdlTargets="zynq stratix4"**, or **RccTarget=linux-c7-x86_64**.
- Inherited from settings the project's or component library's makefile.
- Set in the environment

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 the typical **make** 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 simply be alternative source code implementations of the same spec.

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, has a **Makefile**, and has a test suite description XML file, called ***<component>*-test.xml**.

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

- Test cases (individual parameterized tests)
- Target platforms (HDL hardware and simulation platforms, and RCC Platforms)
- Multiple workers (different source code implementations, different models)
- Worker build configurations (compiled in vs. runtime settable property values)
- Alternative runtime property settings

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 in this framework proceeds in five phases:

1. ***Generate*** — generate testing artifacts
2. ***Build*** — building HDL bitstream/executable artifacts for testing (for HDL workers)
3. ***Prepare*** — examine available built workers and platforms, creating execution scripts
4. ***Run*** — execute tests for all workers, configurations, test cases and platforms
5. ***Verify*** — verify results from the execution of test cases on workers and platforms

The ***generate(1)*** phase performs the following tasks automatically without any developer involvement:

- 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 all the parameter configurations that have been used on any worker as a baseline for test cases.

- 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 assemblies (subdirectories, Makefiles and OHAD files) that can be built for HDL platforms (hardware and simulation).

[In the current release, the applications generated expect the workers being tested to pass a zero length message through from input to output, and the test will be considered completed when such a message is seen at the output of the worker's port.]

All the above tasks are done “off-line”, without regard to particular built artifacts or available platforms on which to execute tests and without requiring any build-related tools (compilers or FPGA synthesis tools or simulators). The ***build(2)*** phase is only necessary for testing HDL workers, and builds the *generated* HDL assemblies for whichever platforms (including simulators) are specified. When building for hardware HDL platforms, this phase takes the longest.

When asked to prepare to execute test cases and perform associated verification, the framework, in the ***prepare(3)*** phase, 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
- Generation of test scripts to perform all feasible tests on all available platforms.

After preparation, the developer invokes the ***run(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).

All this automation leaves the developer with a modest set of tasks to complete the test suite to be fully enabled for all phases. There are various filtering capabilities to subset which tests are run and which platforms should be tested.

The rest of this section will describe:

- what is required and possible in the test description XML file (**`<component>-test.xml`**)
- what can and should be specified in the **`Makefile`** for this directory
- how to provide data generation and verification scripts
- how to execute test cases and verify their results

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

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

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.

Verifier — script to verify test output data produced by output ports
Called for a subcase, for each port's output data file, with all property values supplied, both initial and final (volatile)

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
Test 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 and one verification script, per port, parameterized by subcase property values

13.2 Unit Test Description XML File

This ***<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.

13.2.1 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**. All are optional and are specified in special situations.

13.2.1.1 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.

13.2.1.2 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.

13.2.1.3 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.

13.2.1.4 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.

13.2.1.5 ExcludePlatforms Attribute of the Top-level Tests Element

This string attribute specifies a comma-separated list of platforms that should *not* be tested. Any other available platforms that have built artifacts will be used.

13.2.1.6 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. Any other available platforms will be ignored.

13.2.2 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**, and **messageSize**.

13.2.2.1 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.

13.2.2.2 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.

13.2.2.3 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-from-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.

13.2.2.4 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 one test case.

13.2.2.5 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.

13.2.2.6 Messages 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. *[This option is not supported in the current release, only raw data assuming opcode zero, divided into messages using the MessageSize attribute is supported in the current release.]*

13.2.3 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**, and **view**. It may be applied to all test cases, be used as a default for test case that do not mention an **output** element for a port, or 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.

13.2.3.1 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 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 command ***<command>***, in the script attribute, 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 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.

13.2.3.2 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.

13.2.4 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`**. Exactly one of these attributes, other than **`name`** and **`test,`** must be specified. The textual syntax for property values is used, as described in Property Value Syntax.

13.2.4.1 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).

13.2.4.2 Value Attribute of the Property Element

This attribute specifies a single value to be tested.

13.2.4.3 Values Attribute of the Property Element

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

13.2.4.4 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.

13.2.4.5 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.

13.2.4.6 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 command for 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 earlier **`property`** elements, so scripts are not necessary to perform simple arithmetic based on other parameters.

13.2.4.7 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.

13.2.5 Case Element of Tests Top-level Element

The **`case`** element defines a non-default test case when required. 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`**. All but **`Name`** have the same function as previously defined for the top level **`tests`** elements, but only apply to this case.

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.

13.2.5.1 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.

13.2.5.2 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'/>
```

13.2.5.3 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.

13.2.5.4 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.

13.3 Unit Test Makefile Contents

The Makefile in a ***<component>.test*** directory is normally untouched after being created with the **ocpidev create test** command. It is generated to contain only one line:

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

Several make variables can be used either on the make command line or specified in this Makefile to control the various phases of unit testing.

During the ***build*** phase, as with building projects, libraries and workers, these platform variables are applied: **HdlPlatform(s)**, **OnlyPlatform(s)**, **ExcludePlatform(s)**.

During later phases (***prepare***, ***run***, ***verify***) these platform variables apply: **OnlyPlatform(s)**, **ExcludePlatform(s)**.

The **View** variable can be set to 1, which will cause the "view" script to be run whenever verification is requested.

The **Verbose** variable can be set to 1, which will cause the execution and verification logs to be included in the console/shell output, rather than just placed in specific log files, per platform and per subcase.

The **KeepSimulations** can be set to one to cause the contents of the **simulations** directory to be retained after successful executions on simulation platforms. Successful verification for a platform normally causes the associated **simulations** directory to be removed immediately. Keeping simulation output may use lots of file system space (100s of GBs in some extreme cases).

The **Cases** variable is a wildcard pattern indicating which cases/subcases should be executed or verified. Subcases are named ***<casename>.<subcase#>***, so this variable may be set to patterns that affect certain cases or subcases. Subcase numbers are listed in the report in **gen/cases.txt** report produced by the ***generate*** phase. The default case name is **case00**. For example, to only run subcases that end in **3**, you could specify: **Cases='case*.*3'**. Multiple patterns are allowed, such as: **Cases='case00.01 case00.03'**. The quotes are not necessary in the Makefile, but are necessary on the command line.'

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 be laid out according to the setting of 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 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 in impulse, with a maximum value followed by zeroes. The file is binary 16 bit signed fixed point data.

```
#!/usr/bin/env python
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 command ***`<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 must 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 Off-line One-time Tasks 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.

- Making any necessary changes to the **`Makefile`** (rarely needed)
- 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 next section)
- Examine the report created in **`gen/cases.txt`** to see the generated subcases.
- If any workers are HDL workers, run the ***build*** phase to build the bitstream/executables.

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.

The ***generate*** phase is accomplished using the

```
make generate
```

command, and the build phase is accomplished using the

```
make build
```

command (or simply **`make`** with no goal specified). When the ***build*** phase is invoked, the ***generate*** phase may be re-invoked based on **`make`** dependencies. As mentioned above, the platform variables **`HdlPlatform`**(**`s`**), **`OnlyPlatform`**(**`s`**), and **`ExcludePlatform`**(**`s`**) may be used on the command line for building to modify defaults specified in the project, library or environment.

13.7 Defining Remote Systems for Executing Tests

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 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 provides 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 CentOS6 to run tests on that system also. Of course the CentOS6 system will only run RCC artifacts build for CentOS6.

Remote systems may be defined in the project's **Project.mk** file 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
```

13.8 On-line Tasks for Test Execution and Verification

After the off-line steps described above, and any remote systems are defined, there are four steps that relate to actually executing tests and performing verification, and these steps are aware of available local and remote platforms on which to execute tests.

13.8.1 Preparing for Execution: Discovery and Execution Script Generation

The ***prepare*** phase is invoked by the **`make prepare`** command. It considers all available platforms, local and remote, including available RCC, HDL hardware and HDL simulators. The **`OnlyPlatform`**(**`s`**) and **`ExcludePlatform`**(**`s`**) make variables are used to filter the set of available platforms.

This step also considers which RCC built artifacts are available in the component library 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 ***run*** phases. Both subdirectories are removed by **`make clean`**.

When determining available artifacts for 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 **`gen/assemblies`** directory and the installed CDK on the local and remote systems.

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.

13.8.2 Executing Tests on Available Platforms

The ***run*** phase executes, for each available platform, filtered by **`OnlyPlatform`**(**`s`**) and **`ExcludePlatform`**(**`s`**), test applications for all possible cases and subcases.

Three different make goals can be used to invoke the ***run*** phase:

`make runonly` — perform the ***prepare*** phase and then the ***run*** phase.

`make runnoprepare` — manually rely on a ***prepare*** phase that has already be performed, and only perform the ***run*** phase. This 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).

`make run` — perform the ***prepare***, ***run*** and ***verify*** phases all together, with the ***run*** and ***verify*** steps interleaved by subcase with each subcase being verified immediately after it has been executed.

The ***run*** 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`** directory with different files for different subcases.

13.8.3 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 ***run*** phase.

The **`make verif`**y command performs verification for all platforms. The **`make run`** command will perform both ***run*** and ***verify*** phases in an interleaved mode where each subcase is verified after it is executed in order to show the results for subcases as they are executed.

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 ***run*** phase or by itself, the view option will run the defined view scripts with verification. The view option is enabled by specifying **`View=1`** on the make command line, and happens per subcase before each verification. When view scripts typically put up a window to display the data, they may wait for user input or simply return, allowing the data to be displayed while verification proceeds.

The ***verify*** phase will return an error at the end (will fail) if any subcases failed by returning non-zero exit status. 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.

13.8.4 Viewing Test Result Data

The **`make view`** command can be used to specifically run any view script defined for output ports. If the **`View`** variable in the **`Makefile`** (or on the **`make`** command line) is set to 1, this will always happen as a side effect of running the test, but when it is not, **`make view`** can be used to view results on demand (assuming a view script was defined).

13.9 Summary of Make Goals and Variables

The **make** command is used with the following goals to invoke the phases:

build — (the default goal if none is specified) build locally generated HDL assemblies, implies **generate** if needed

run — perform execution and verification, interleaved per subcase

clean — clean all generated, built and execution directories and files

The following goals control the process at a finer granularity:

generate — perform all offline generation tasks: scripted input data and property values, applications, HDL assemblies, generation report

prepare — discover platforms and artifacts and generate execution scripts per platform, assumes required artifacts are built

runnoprepare — execute tests assuming a previous prepare was done

runonly — prepare and execute tests, but do not perform any verification

verify — perform verification for whatever platforms have been executed

view — run the view scripts and put the results in separate windows

cleanrun — clean all run results

cleansim — clean all simulation output (but not run results)

The **View** and **KeepSimulations** variables apply to verification.

The **Case**(**s**) variables apply to both execution and verification.

The **OnlyPlatform**(**s**) and **ExcludePlatform**(**s**) applies to preparation, execution and verification.

The following make goals can be executed in a component library's directory to apply to all ***<component>*.test** directories in the library, or at the project level to apply to all component libraries in the project: **test**, **cleantest**, **runtest**, **verifytest**, **runonlytest**. They perform the same function as the corresponding goal in the ***<component>*.test** directory, without the **test** suffix. The **cleanrun** and **cleansim** goals can also be used at the library and project levels without a **test** suffix.

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. 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 source code form, along with the makefiles 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 not discussed 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 makefiles and other files omitted except at the project level).

```
Project.mk
Project.exports
Makefile
applications/<applicationXYZ>/
specs/
components/<componentlibXYZ>/<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 applies to 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 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 in the project, 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 and workers are created directory under **components**. When creating a worker, if no library is specified, 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.

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

Makefile: the top level Makefile for the project which supports building all assets from the top level project directory.

Project.mk: the make file fragment that defines make variables and options 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 users of the deliverables of the project.

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

14.1 Managing Project Assets.

The **ocpidev** tool described in detail later is used to manage all the asset types in a project. It is used to create or delete 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 a makefile in that directory. These include:

- Component libraries
- Workers
- 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 **Makefile** is created in the directory, and in some cases other initial files are also created. The makefiles each indicate which type of asset is in that directory.

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 (simple XML based applications with no ancillary files)

When creating specs, protocols, and workers, 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 the directory under the project's **hdl/** subdirectory where the device worker should be created.

When adding a 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 too. Portable device workers that are *not* platform-specific don't use this.

14.2 Project Makefiles

Most of the directories in a project contain a file named ***Makefile***, which is used to build the assets in that directory. The generated **`Makefile`** has the same form, setting optional **`make`** variables, and including a standard makefile fragment from the OpenCPI CDK. These makefiles are automatically generated when a project is created using **`ocpidev`**, and can subsequently be edited by the developer to specify additional optional variable settings or add customized **`make`** targets etc. The **`Makefile`** can be left unmodified, with the default behavior being adequate and appropriate. Each **`Makefile`** has initial single line content and for the top-level **`Makefile`** in a project, it is:

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

The **`project.mk`** file thus included from inside the CDK is *not related* to the **`Project.mk`** file mentioned above.

An example of an optional variable setting is the list of targets for which this project should normally be built *when built from the top-level project directory*. E.g. the following file would be a top level project makefile that, when building HDL (FPGA) assets *from the top level of the project*, would only build for the Modelsim simulator and the ml605 Xilinx virtex6 development board:

```
HdlPlatforms=modelsim ml605
include $(OCPI_CDK_DIR)/include/project.mk
```

When a makefile is in a directory with a number of subdirectories for the same type of asset, a variable can be set which lists the assets to build. This variable is optional, and when not specified, all such assets are built. For example, in a component library where all the subdirectories contain workers, the default Makefile is simply:

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

This 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`** variable can be specified to list the explicit set of workers that *should* be built, in order, e.g.:

```
Workers=fft.rcc fft.hdl
include $(OCPI_CDK_DIR)/include/library.mk
```

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 variable, and other optional variables, are described in the section for each asset.

Each standard make file (fragment) has a default make target (**`all`**) that builds all assets, when **`make`** is invoked with no targets. There is also always a **`clean`** target which removes all generated/temporary files, except those initially created by **`ocpidev`**. When a directory holds assets of the same type, they all have a corresponding make target. E.g., in a component library, any worker can be built be specifying it as a make target. Additional **`make`** targets may be available, such as in the top level project directory which has these make targets:

components — build all component libraries, and all workers in each

applications — build the applications in the **applications** directory

hdldevices — build the HDL device workers in the **hdl/devices** directory

hdlplatforms — build the HDL platforms in the **hdl/platforms** directory

hdlassemblies — build the HDL assemblies in the **hdl/assemblies** directory

hdl — build all HDL assets (primitives, components, devices, platforms, assemblies)

test — build all tests in all component libraries

The makefiles for directories that hold multiple assets of the same type have makefiles that indicate this by including the appropriate makefile fragment from the CDK:

- The **components** directory makefile, when there are multiple libraries, contains:
  ```
  include $(OCPI_CDK_DIR)/include/libraries.mk
  ```
- The **applications** directory makefile contains:
  ```
  include $(OCPI_CDK_DIR)/include/applications.mk
  ```
- The **hdl/platforms** directory makefile contains:
  ```
  include $(OCPI_CDK_DIR)/include/hdl-platforms.mk
  ```
- The **hdl/assemblies** directory makefile contains:
  ```
  include $(OCPI_CDK_DIR)/include/hdl-assemblies.mk
  ```
- The **hdl/primitives** directory makefile contains:
  ```
  include $(OCPI_CDK_DIR)/include/hdl-primitives.mk
  ```

The variables in the top level project **Makefile** apply when make is invoked in that top level project directory. The variables set in the Project.mk file in the top-level project directory apply anywhere in a project, when make is invoked in any of the project's directories.

The variables supports in the top-level project **Makefile** are:

RccTargets, **RccTarget**

HdlTargets, **HdlTarget**

HdlPlatforms, **HdlPlatform**

[soon: OcpiTargets, OcpiDynamic, OcpiDebug, OcpiEndian]

14.3 The Project.mk File for Project-wide Variable Settings

This file is required in the top level directory of a project. It contains variable 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.mk`** *file is found.* This is similar to how the **`git`** tool finds the top level of a git repository by searching for a file named **`.git`**.

This top level project file is included in all the makefiles automatically (by the included standard CDK makefile fragment at each level). It is *not* the file in the CDK that is included via the directive at the end of the project's top-level **`Makefile`** using

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

Variable settings in the project's **`Project.mk`** file are used even when the developer runs **`make`** in subdirectories of the project (i.e. not running **`make`** in the top level project directory). Within an OpenCPI project, the **`make`** command can be run directly in any asset's directory with the exception of HDL primitives, which must be built from the **`hdl/primitives`** directory or the top level project directory.

Variable settings that apply only when running **`make`** from the top-level project directory can be put in the **`Makefile`** in the top-level directory rather than in **`Project.mk`**. Variable settings in this **`Project.mk`** file can either override settings made in a given Makefile, add to lists, or conditionally set the variables if not already set. For example putting the following lines in **`Project.mk`**:

```
ifndef HdlPlatforms
  HdlPlatforms=zed
endif
```

indicates that when any level of the project is built, if **`HdlPlatforms`** has no value, use this definition.

The project variables that may be set in a **`Project.mk`** file are in the following table are all optional. This file must be present, but may be empty.

Table 7: Variables set in the Project.mk file

Variable Name in `Project.mk`	Default	Description
`ProjectName`	Project directory's name	The name used for this project.
`ProjectPackage`	**`local`**	The namespace prefix for all assets in the library. The default, **`local`**, is appropriate when the assets are intended to be used only in the local organization.
`ProjectPrefix`	***`<ProjectName>_`***	The prefix applied to any primitive software libraries in the project. E.g. an **`xyz`** library would be **`lib`*`<prefix>`*`xyz.a`**.
`ProjectDependencies`	**`""`**	A list of the directories of other projects that this project depends on.

In order to avoid name space collisions when using multiple projects or component libraries (e.g. spec names and worker names), this **`ProjectPackage`** variable specifies what namespace should be used for all named assets in the project. The default project package name is **`local`**. However, for libraries used outside the local organization, the prefix should generally be the reverse internet domain name similar to the name spaces used for Java classes. E.g., for assets in the **`dsp`** project of the **`xyz`**-corporation, the setting would be:

```
ProjectPackage=com.xyz-corp.dsp
```

The package prefix **`ocpi`** is reserved for assets defined in the OpenCPI core project found in the (CDK).

This value is used as a prefix to the package names defined for any component library in the package (when specified). Libraries can override or append to this **`ProjectPackage`**.

The **`ProjectDependencies`** variable should be used to declare other projects that this project depends on. This will automatically use these other projects 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)

Other useful *make* variables that can be specified in the **`Project.mk`** file include:

```
HdlTargets
HdlPlatforms
RccTargets
```

```
OcpiDebug
OcpiDynamic
```

14.4 Project Exports

When a project's assets are used by applications or assets *outside* the project, they access 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.mk`** file **`ProjectDependencies`** setting, that means they use project exports from those projects they depend on.

The ***exports tree*** is a directory containing the project's exports, and it 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. At the top level of a project, the make target **`exports`** creates the exports tree. i.e.:

```
make exports
```

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 package for users of the project. In this latter case a simple single file deliverable package can be created as a tarball using the command:

```
make archive
```

which performs this command

```
tar czfLs exports/myproject.tgz /exports/myproject exports
```

This creates a single compressed archive file in **`exports/myproject.tgz`** of all the deliverables of the project (following the links to the actual files), which can then be installed anywhere using the command:

```
tar xzf myproject.tgz
```

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, the default export tree is created based on the assets in the project. The **`Project.exports`** file adds or subtracts from the default exports tree.

The next section describes the default **exports** tree and the format of the **Project.exports** file that can be used to add or subtract from the default exports.

The most complete example of exports is the core project of OpenCPI, referred to as the CDK.

14.4.1 The default exports tree

The default exports tree is a directory structure convenient and appropriate for making the built results of a project available to be used and/or installed by external users of the project. It is related to, but not the same as, the directory structure of a project in which assets are developed.

Here are the rules used to populate the default exports tree when **make exports** is invoked at the top level of a project:

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**, e.g.:

```
exports/lib/util_comps -> ../../../components/util_comps/lib
```

So, users of the project, seeing only the exports tree, see **lib/util_comps** for this library.

Software deliverables are made available in one of these export directories:

bin/*<target>*	Executable files
lib/*<target>*	Library files
scripts	Architecture-independent scripts
include	Architecture independent header/include files
specs	Spec and protocol XML files for component specifications (separate from the specs of a given library)

Software targets generally use the format: ***<os>-<version>-<processor>***. The ***os*** part is lower **linux** or **macos**. The ***version*** part is an abbreviation of the distribution and major version. For Linux, it is typically a letter for a distribution followed by a major number (e.g. **c6** for CentOS 6, **c7** for CentOS 6, **r5** for RHEL 5, **u14** for Ubuntu 14). For OS/X it is simply the major version (e.g. **10_11**). The processor (or *architecture*) variable is a lowercased version of what the **uname -m** Unix command would return, such as **x86_64**.

Typical examples are **linux-c6-x86_64** for 64-bit CentOS 6 Linux on x86, or **macos-10_11-x86_64**. HDL targets typically contain an architecturally compatible part family (e.g. virtex6 or stratix4). See the ***OpenCPI HDL Development Guide*** for more information.

14.4.2 The Project.exports file

The **Project.exports** file contains lines that add and subtract from the default exports tree of a project. Additions are lines that start with a plus sign (**+**), and subtractions are lines that start with a minus sign (**-**). Lines that begin with **#** are comments. Comments can also appear at the end of the addition or subtraction lines after a **#**. White space (spaces or tabs) can precede the **-**, **+**, or **#** characters.

The format of addition (**+**) lines is two fields separated by spaces. 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 would be exported 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.

Remember that 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 "**make exports**" is executed in the project directory, it is done in the context of a particular software target (usually set in the environment). 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 (**-**) have can have two fields. As with addition lines, the first field is the pathname within the project, with possible wildcards and **<target>** strings. The second field is a qualifier that limits the scope of the subtraction. The possibly qualifiers are: **main, library, script, include, platform, component, primitive**. These indicate that the subtraction only applies to assets of the given type.

14.5 Using Other Projects that Exist Outside the Project Being Developed

The most convenient and recommended way to use a project (A) from another project (B) is to use the **ProjectDependencies** variable in the project B's **Project.mk** file. This declares the dependency and does not depend on any environment settings. 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.

Like the space-separated list of projects in the **ProjectDependencies** variable in the **Project.mk** file, the **OCPI_PROJECT_PATH** environment variable specifies a colon-separated set of other projects to be searched in order, when various types of assets are being located during development. These are searched *before* projects mentioned in the **ProjectDependencies** variable to allow the environment variable to temporarily override what is declared by the project. Like most other "PATH" environment variables, it is a colon-separated list of project directories to be searched when looking for various asset types.

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 at the component libraries within the project. When a project is examined based on being in **OCPI_PROJECT_PATH**, its **exports** tree is actually used if it exists.

The OpenCPI CDK, located using the **OCPI_CDK_DIR** environment variable, is always searched last — as the *last project*.

To summarize searching at the project level, the order is:

1. Use assets in projects in OCPI_PROJECT_PATH to override anything later in this list.
2. Use assets in the local project.
3. Use assets in ProjectDependencies, in the order given in that variable.
4. Use assets in the CDK,

In cases where a standalone component library or application is being developed outside the project structure, the **OCPI_PROJECT_PATH** variable can also be used when these non-project assets depend on projects other than the CDK.

15 The `ocpidev` Tool for Managing Assets

The **ocpidev** command line tool is used to perform various development-related tasks, both inside projects as well as when projects are not being used. 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 subcommand being used.

The general usage of the **ocpidev** command is:

```
ocpidev [options] <verb> [<noun> [<name> [<arguments>]]]
```

The general usage concept is:
*perform the **<verb>** operation on the **<noun>** asset type whose name is **<name>**.*

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 resulting binary files (compiled object files, synthesis netlists for HDL assets, etc.)

clean — clean all the generated and compiled files for the asset(s)

The nouns are:

project, applications, application, spec, protocol, properties, test, library, worker, and hdl *<noun>*

After the **hdl** qualifier, the nouns are:

assemblies, assembly, card, slot, device, signals, subdevice, platforms, platform, primitives, and primitive *<noun>*

After **hdl primitive**, the nouns are:

library, or core

When the **build** or **clean** verbs are used without nouns, they operate on the asset(s) implied by the current directory. The **create** and **delete** verbs always require a noun. The **build** or **clean** verbs do not operate on specs, protocols, properties, hdl cards, hdl slots or hdl signals.

When the noun is one of these plurals: **applications**, **hdl assemblies**, **hdl platforms**, or **hdl primitives**, no name is expected, only **build** and **clean** are allowed, and all assets of that type are operated on. For **create** and **delete** a ***<name>*** command argument identifying the asset is required after the noun.

All these commands can be issued at the top level directory of a project. In a component library directory, the commands for operating on assets in the library can be issued for these asset types:

spec, protocol, test, worker, hdl device

When in a plural directory (applications, HDL platforms, HDL primitives or HDL assemblies), the appropriate assets can be operated on by name.

15.1 Assets Managed by `ocpidev`

Below are two table 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 assets.

Table 8: Non-HDL Asset Types (Nouns) for the `ocpidev` Command

Name/ Noun	Create/ Delete?	Build/ Clean?	In Library?	Description
`application`	yes	yes	no	A component application, specified either in XML or C++
`applications`	no	yes	no	All applications in a project or in an applications directory
`library`	yes	yes	no	A component library.
`project`	yes	yes	no	A project containing all other asset types.
`properties`	yes	no	yes	A properties XML file, which can be at the project or library level.
`protocol`	yes	no	yes	A protocol specification XML file, which can be at the project or library level.
`spec`	yes	no	yes	A component specification XML file (OCS), which can be at the project or library level.
`test`	yes	yes	yes	A unit test suite for a component specification.
`worker`	yes	yes	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 9: HDL Asset Types (Nouns) for the **`ocpidev`** Command*

Name/ Noun	Create/ Delete?	Build/ Clean?	In HDL Library?	Description
`hdl assembly`	yes	yes	no	An assembly of HDL workers..
`hdl assemblies`	no	yes	no	All the assemblies in the project or in an HDL assemblies directory.
`hdl card`	yes	no	yes*	A card specification XML file, which can be at the project or HDL library level.
`hdl device`	yes	yes	yes*	An HDL device worker, in an* HDL devices library.
`hdl platform`	yes	yes	no	An HDL platform worker, in an HDL platforms directory.
`hdl platforms`	no	yes	no	All HDL platforms in a project or in an HDL platforms directory.
`hdl primitive core`	yes	yes	no	An HDL primitive core in a project or HDL primitives directory
`hdl primitive library`	yes	yes	no	An HDL primitive library in a project or HDL primitives directory
`hdl primitives`	no	yes	no	All HDL primitives (cores or libraries) in a project.
`hdl signals`	yes	no	no	A signals specification XML file, which can be at the project or HDL library level.
`hdl slot`	yes	no	no	A slot type specification XML file, which can be at the project or HDL library level.
`hdl subdevice`	yes	yes	yes	An HDL subdevice worker, that supports other HDL device workers, in an HDL devices library.

15.2 Options for the `ocpidev` Command

The available options are in the following tables. All **`delete`** commands prompt for confirmation unless the **`-f`** option is supplied. When **`create`** fails all partially created directories and/or files are removed unless the **`-k`** (keep) option is specified.

The options that are specific to workers for a specific authoring model are more fully described in the development guide document for that authoring model. Options relating to platform development (enabling new platforms and devices), are more fully described in the ***OpenCPI Platform Development Guide***.

The options are numerous, but in most cases very few are required. The more complex cases are normally only used by higher level tools and scripts that are out of scope for this document. In the tables below, each option indicates:

- whether it takes a value after the command option flag (Value?)
- whether it can be used multiple times to provide multiple values (Multiple?)
- the default value if the flag is not supplied at all

*Table 10: General Options for the **ocpidev** command*

Option	Value ?	Multiple ?	Default Value	Description
-d	yes	no	.	Specify the directory in which this command should be run. Analogous to the **-C** option in the POSIX make command.
-f	no	no		Force deletion, do not ask for confirmation when deleting assets. Similar to the same option in the POSIX **rm** command.
-h	yes	no		The library of the HDL asset (specs, protocols, devices or proxies) when it is under the **hdl/** directory . Value is **cards**, **devices** or **adapters**.
--help	no	no		Print help message, equivalent to providing no arguments at all.
-k	no	no		Keep files and directories created after a creation fails. Normally all such files and directories are removed upon such a failure.
-l	yes	no		The library of the asset when there is more than one. This applies to specs, protocols, properties and workers.
-ll	yes	no	*	The log level to use when executing the ocpidev command, for internal debugging. *The default is set by the OCPI_LOG_LEVEL environment variable.
-p	no			Create specs or protocols at the top project level, not in a library's **specs** directory.
-s	no	no		Run standalone, not part of a project; applies to all but projects.
-v	no	no		Be verbose, describing what is happening in more detail.
-G	yes	yes		An “onlyplatform” to limit build platforms for this asset.
-P	yes	no		The HDL platform for operating on device workers or proxies in the HDL platform's own **devices** library.
-Q	yes	yes		An “excludeplatform” to exclude when building this asset.
-T	yes	yes		An “onlytarget" to limit build targets for this asset.
-Z	yes	yes		An “excludetarget” to exclude when building this asset.

*Table 11: Creation Options for the **ocpidev** command*

Option	Value ?	Multiple ?	Default Value	Description
Options when creating projects				
-D	yes	yes		The pathname of another project that this project depends on. The order of these other projects becomes the order they are searched for finding various asset types during the build process.
-F	yes	no	***<name>*_**	The project prefix when creating a project.
Options when creating projects or libraries **-k**				
-K	yes	no	**local**	The package name. For libraries, a name with no periods or starting with a period is relative to the package name of the project.Run standalone, not part of a project; applies to all but projects.
Options when creating projects, libraries, workers, platforms, and assemblies				
-G	yes	yes		Specify a library that contains workers/devices/specs that this asset will need to reference. Proxy and subdevice workers may need this.An “onlyplatform” to limit build platforms for this asset.
Options when creating workers of all types (the -S option also applies to creating tests)				
-A	Yes	yes		A directory to include in the search path for XML files.
-L	yes	no	**	The language for the worker being created. **Defaults to the default language for the authoring model.
-N	yes	no		The worker's name attribute in its OWD (XML file) when it is different from the name before the model suffix in the <name> argument of the command. Rarely used and not recommended.
-O	yes	yes		A source file to include in addition to the primary worker source file.
-S	yes	no	***<name>*-spec**	The spec for a worker. The default is ***<name>*-spec** or ***<name>*_spec** depending on what is found in the **specs** directory of the library or project.
Options when creating specs				
-n	no	no		The spec has no control interface or non-parameter properties.
-t	no	no		When creating a spec, also create a test for it, in the same library.
Options when creating applications				
-x	no	no		The application is an XML file in its own directory.
-O	yes	yes		A source file to include in addition to the primary worker source file.
-X	no	no		The application is just an XML file, and not in a directory of its own.

Option	Value ?	Multiple ?	Default Value	Description
Options when creating RCC Workers				
-r	yes	yes		A dynamic prerequisite library needed by this RCC worker.
-I	yes	yes		A directory to include in the search path for header files for this worker.
-R	yes	yes		A static prerequisite library needed by this RCC worker.
-V	yes	no		The slave worker when the RCC worker being created is a proxy.
-W	yes	yes		One of multiple workers implemented in this worker's directory when a single RCC worker directory is creating a multi-worker artifact.
Options when creating HDL device-related specs, protocols, workers				
-C	yes	yes		A primitive core needed by this HDL worker (device or platform).
-E	yes	no		The device worker for the device this HDL worker will emulate.
-U	Yes	yes		An HDL device worker that this (subdevice) worker supports.
-Y	yes	yes		A primitive library needed by this HDL worker or assembly.
Options when creating HDL primitive libraries or cores				
-H	no	no		This HDL primitive library does not depend on any other primitive libraries, even the internal OpenCPI ones that are normally made available by default.
Options when creating HDL primitive cores				
-B	yes	no	**<name>**	Indicates a prebuilt (not source code) core
-M	yes	no	**<name>**	For HDL primitive cores, the top level module name for the core when it is different from the name of the core.
Options when creating HDL platforms				
-g	yes	no		The actual part (***<die>-<speed>-<package>***) used on the platform.
-q	yes	no	100e6	The control clock frequency driven by the platform, in Hz.
-u	no	no		The HDL platform does not have an SDP dataplane interconnect.

The following table lists the options that may be used when the verb is **build**. They all start with **--build-**. The "**Where"** column indicates what asset directories are valid for using the options. Note that using **build** at the project level builds all assets except unit tests. Similarly, **build** in a library builds all types of workers in the library.

*Table 12: Build Options for the **ocpidev** command*

Option	Value ?	Multiple ?	Where Valid?	Description
--build-rcc	no	no	Project, Library	Build RCC assets. Currently this is only RCC workers.
--build-hdl	no	no	Project, Library	Build HDL assets: primitives, workers, platforms, assemblies
--build-no-assemblies	no	no	Project	When building HDL assets, suppress building assemblies.
--build-hdl-assembly	yes	yes	Project, Assemblies	When building HDL assets, explicitly specify an assembly to build.
--build-hdl-target	yes	yes	Anywhere except Applications	When building HDL assets, include building for the target(s) specified with this option. When not mentioned, the default set of targets are implied by those needed for the specified HDL platforms.
--build-hdl-platform	yes	yes	Anywhere except Applications	When building HDL assets, include building for the platform specified with this option. When not mentioned, the default set of platforms are those optionally specified in the project's or library's Makefile.
--build-rcc-platform	yes	yes	Project, Library, Worker, Application	When building RCC assets, include building for the RCC platform specified with this option. When not mentioned, the default set of RCC platforms is the development host the command is run on.
--build-hdl-rcc-platform	yes	yes	Project, Library, Worker, Application	When building RCC assets, include building for the RCC platform associated with the HDL platform supplied as the value of this option.

15.3 Using `ocpidev` in Standalone Mode

The **`ocpidev`** command can be used when not using projects. When the **`-s`** option is specified, for *standalone* operation, **`ocpidev`** does not require that it be run inside of a project. This applies to libraries and all types of workers.

Standalone mode may be useful when a single asset needs to be created in some directory by itself. Normally multiple related assets of multiple types are developed together in a project.

Specs, protocols and workers can be created or deleted when in a library directory that is not part of a project.

15.4 Examples Using ocpidev

Here some examples using the ocpidev command.

```
# Create a project named "myproject" in the directory /somewhere
ocpidev create -d /somewhere project myproject

# In a project directory, build all assets other than tests, or
# in a library, build all the workers in the library, for the
# default set of platforms.
ocpidev build

# In a project directory, delete the worker aaa.rcc in the single
# component library in the "components" directory.
ocpidev delete aaa.rcc

# In a project directory, delete the worker bbb.hdl in the "mylib"
# component library
ocpidev delete -l mylib bbb.hdl

# Clean the build results from mylib library
ocpidev clean library mylib
```

16 Environment Variables used in Component Development

While any **`make`** variable can be set in the environment, only those specifically mentioned as usable in the environment should be set, since setting arbitrary **`make`** variables in the environment can lead to unpredictable/undefined behavior.

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: **`/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/license.dat`**
OCPI_ALTERA_KITS_DIR	install	directory	Directory for Altera development kits. Default is **`$OCPI_ALTERA_TOOLS_DIR/kits`**
OCPI_ASSERT	build	boolean	Enable assertions in any language (when != 1, for C and C++, sets NDEBUG). Default: **`1`**
OCPI_CDK_DIR	install	directory	The root of the installed CDK. Default: **`/opt/opencpi/cdk`**
OCPI_CFLAGS	build	list	Flags when compiling C code. Default set per target, but may be overridden.
OCPI_CXXFLAGS	build	list	Flags when compiling C++ code. Default set per target, but may be overridden.
OCPI_DEBUG	build	boolean	Controls debug logging, etc. For C and C++, enables “-g” also, or if not set, -O2. Is defined as a preprocessor macro for core software or executables. Default: **`1`**
OCPI_DYNAMIC	build	boolean	For main programs and software libraries: use dynamic linking and dynamic libraries. For software components, build for use in dynamic executables. Currently only **`0`** is supported.
OCPI_LIBRARY_PATH	run	path	Runtime search path for binary artifacts. Default: **`$OCPI_CDK_DIR/lib/components/rcc`**
OCPI_PREREQUISITES_BUILD_DIR	install	directory	Where prerequisite packages are built during installation. Default: **`$OCPI_PREREQUISITES_DIR`**
OCPI_PREREQUISITES_INSTALL_DIR	install	directory	Where prerequisite packages are installed during installation. Default: **`$OCPI_PREREQUISITES_DIR`**

Name	When set?	Data type	Description
OCPI_PREREQUISITES_DIR	install	directory	Location for using prerequisites, default locations for building and installing prerequisites. Default: **`/opt/opencpi/prerequisites`**
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_TOOL_HOST	build	String	A target triple (e.g. **`linux-c7-x86_64`**) determined from **`OCPI_TOOL_PLATFORM`**
OCPI_TOOL_OS/OS_VERSION/ARCH	build	String	For convenience, the components of **`OCPI_TOOL_HOST`**
OCPI_TARGET_PLATFORM	build	String	The requested target software platform to build for.
OCPI_TARGET_HOST	build	String	A target triple (e.g. linux-c7-x86_64) determined from **`OCPI_TARGET_PLATFORM`**
OCPI_TARGET_OS/OS_VERSION/ARCH	build	string	For convenience, the components of **`OCPI_TARGET_HOST`**
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 Tools Used in Component Development

The most commonly used tools used during OpenCPI component development are **`make`**, **`ocpidev`**, and **`ocpirun`**. The latter is used for executing component-based applications, and is fully described in the ***Application Development Guide***. The **`make`** tool invokes a variety of other tools to build workers, all behind the scenes. When there are errors or anomalies during the build process, the logs from the various tools used can be examined to understand what went wrong. The **`ocpidev`** tool is used to create and delete various OpenCPI assets, usually within a project.