OpenCPI
Component Developer Guide

DRAFT

Authors:
Michael Pepe, Mercury Federal Systems, Inc. (mpepe@mercfed.com)
Jim Kulp, Parera Information Services, Inc. (jkulp@parera.com)

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

Table of Contents

1 References

This document depends on several others. Primarily, it depends on the "OpenCPI Generic Authoring Model Reference Manual", which describes concepts and definitions common to all OpenCPI authoring models. This document also refers to concepts and definitions in the HDL and RCC authoring model reference documents.

Title	Published By	Link
OpenCPI Technical Summary	OpenCPI	Public URL: http://www.opencpi.org/doc/ts.pdf
OpenCPI Authoring Model Reference	OpenCPI	Public URL: http://www.opencpi.org/doc/amr.pdf
OpenCPI RCC Authoring Model Reference	OpenCPI	Public URL: http://www.opencpi.org/doc/rccamr.pdf
OpenCPI HDL Authoring Model Reference	OpenCPI	Public URL: http://www.opencpi.org/doc/hdlamr.pdf

2 Overview

This document describes how to create OpenCPI component implementations (a.k.a. *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. It also includes how to create, build, and manage libraries of heterogeneous components (components with multiple implementations/workers).

For HDL workers and applications, it also describes how to create libraries of primitives (smaller/simpler reusable modules) used to build HDL workers, and how to assembly a group of HDL workers to form an "HDL assembly", in order to build a complete FPGA bitstream that will support part or all of an OpenCPI component-based application.

We refer to the set of tools, and libraries used for developing components and workers as the: OpenCPI Component Developer's Kit (CDK). The CDK is not an IDE (although integration with the Eclipse IDE is a roadmap item), but rather is a set of commands, "make" 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. It includes several tools specific to OpenCPI including:

ocpigen: a source code and XML generation tool that processes various OpenCPI XML metadata files and generates various source code and other XML files. It supports development using OpenCPI's native XML metadata formats.

ocpisca: a tool that integrates OpenCPI into the SCA (DoD Software-Defined Radio framework) component development process by:

- Generating OpenCPI XML from the more complex and comprehensive SCA XML metadata (and CORBA IDL).
- Back-annotating SCA SPD files to properly reference and describe component implementations (workers).

ocpixm: a tool that generates code to wrap X-Midas primitives (written in C++ or Fortran) for use in OpenCPI as OpenCPI workers. It is described in a separate document.

These tools are usually used indirectly, by using the built-in provided "make" scripts that facilitate building component libraries of heterogeneous implementations, and building components in each of the available authoring models.

The OpenCPI CDK also relies on technology-specific compilers (e.g. gcc) and (FPGA) synthesis and simulation tools (e.g. Xilinx XST and Isim, Altera Quartus, Modelsim etc.).

3 Component libraries

OpenCPI components are developed in libraries. OpenCPI component-based applications are built as a composition (or assembly) of components, and the components are drawn from component libraries at execution time. 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" in the source library, "exporting" the results to a binary archive, and then "installing" that binary archive into a directory where applications can find and use the implementations in the library. As a shortcut during component development, the "export" and "install" steps can be skipped since a "binary" version of the library is actually built directly as a subdirectory of the source tree.

A component library, in source form for component developers, is a directory that contains:

- Component specifications (OCSs, "spec files") in a "specs" subdirectory.
- Component implementations (each in its own subdirectory)
- Component tests in *.test subdirectories.
- The Makefile for the component library as a whole
- When built, a library exports subdirectory (the "lib" subdirectory) is the binary form of the library (files specifically exported by the library as required by applications when they *use* components in the library).

The exported version of a component library (created in the subdirectory "lib"of the source tree) contains mostly binary files (a.k.a. artifacts), and some XML files. An exported library thus contains a heterogeneous collection of binary files containing built workers in various technologies.

When the developer of the library wishes to send its binary form to an application developer or user, it simply provides the contents of this "lib" subdirectory (usually in an archive format), which is then extracted on the user's system, typically in the "/opt/opencpi/lib/xxx" directory where "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 (e.g. linux for x86 and linux for ARM, or for Xilinx and Altera FPGAs, or for different simulators).

For application developers and users, libraries are accessed dynamically at runtime via the "OCPI_LIBRARY_PATH" environment variable, similar the "LD_LIBRARY_PATH" environment variable on UNIX/Linux systems, or DYLD_LIBRARY_PATH on MacOSX.

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

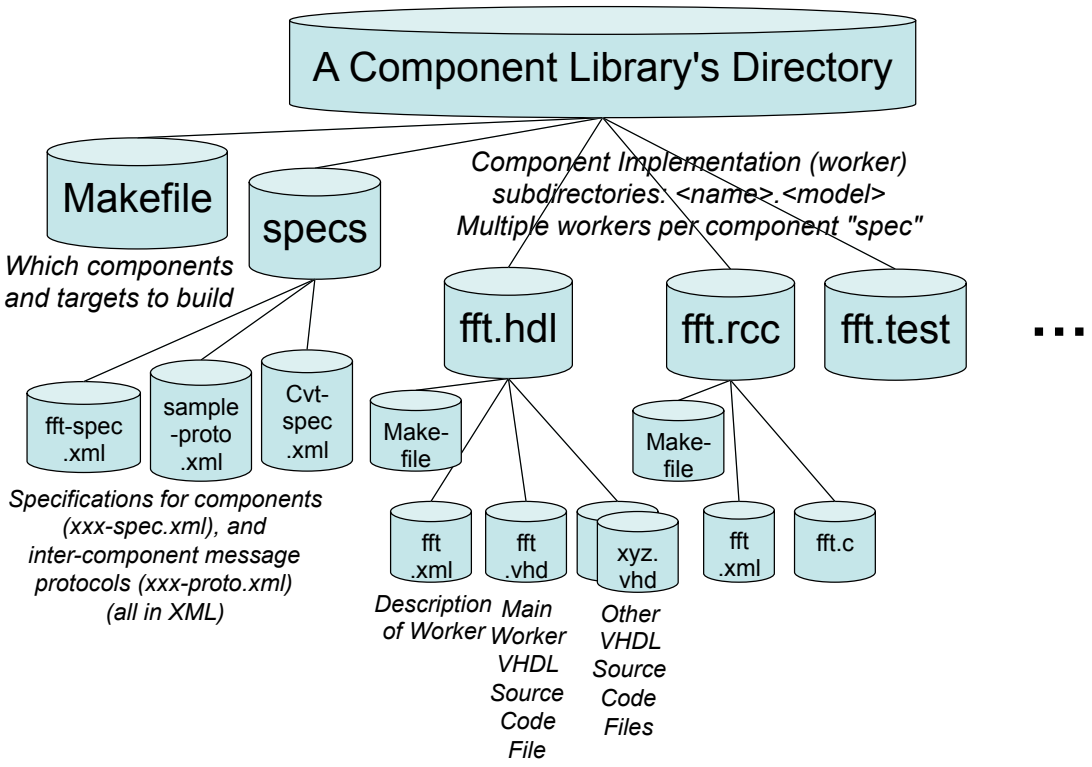

Figure 1: Component Library Directory Structure

3.1 Component Specifications

A “Component Specification” is a description that is common to all implementations of a component. It describes the component properties, ports, and message protocols at each port of a component. This XML file is called the “spec file” for the component, and has a “-spec.xml” suffix. The spec files for all components in the library are found the “specs” sub-directory of the library. When groups of properties or groups of message protocol “operations” (message types), are shared between spec files they are typically placed in separate “-prot.xml” or “-prop.xml” files that are referenced from multiple spec files. The suffixes and locations of these files are required for the component library management scripts to know what files must be exported when applications use components in the library. The contents of these XML files are described in detail in the [AMR].

These spec files (and property and protocol files, if they are used) are used by two completely 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.[See the OpenCPI Application Reference document].

To get the most value out of the heterogeneous component model of OpenCPI, it is strongly encouraged to use a common spec (and common unit tests) between different implementations of the same functionality.

3.2 Implementation/worker subdirectories

Component *implementations* are in implementation/worker subdirectories of the component library's directory. Some authoring models (e.g. RCC) support creating a single binary file ("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) file. The names of the worker subdirectories have a suffix indicating the authoring model used for that implementation. Thus if we have a component whose spec file is named "search-spec.xml", the RCC model implementation of that component would normally be in a subdirectory called "search.rcc". The names of the spec file and the implementation subdirectory do not have to match, but it is recommended and allows the use of more defaults to simplify the process. If there were an HDL implementation of the same component, it would be in the subdirectory "search.hdl". Note that these names ("search") 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. A "search.test" subdirectory would be created for a unit test for all implementations of the "search" component, as defined in the "spec" file.

The first step in creating a component implementation (after creating the spec file) is to create a subdirectory in the component library with the name of the implementation (before the ".") and the authoring model used (after the "."). The contents of these directories are different based on the authoring model (although they are as similar as conveniently possible), and are described in model-specific sections below. What is in common between all authoring models is that the implementation subdirectory contains these files:

1. Worker Description File (the OWD XML file)
2. Worker Makefile
3. Worker source code file(s)

When XML and source language files are automatically generated by the built-in make scripts, they are placed in a "gen" subdirectory of the worker's directory. When files are compiled, the resulting binary files are placed in subdirectories named: target-*<target>*, where *"<target>*" is the type of hardware the compilation is targeting. Thus "cleaning" (via "make clean") a worker directory simply removes the "gen" and all "target-*" subdirectories.

3.2.1 The Worker Description File

For each worker implemented in the subdirectory (usually only one), there is an XML file that describes the worker and references the spec file in the component library "specs" sub-directory. These files are called OpenCPI Worker Descriptions or OWDs in the [AMR]. The generic (across authoring models) aspects of these implementation description files (OWDs) are documented in detail in the AMR. The build scripts and makefiles automatically put the "specs" directory (the component library directory) into the search path when these worker description files are processed, so the spec files (OCSs) need only be referenced by their name and not any directory or pathname.

The worker description file essentially adds non-default implementation information to the basic information found in the spec file. Each authoring model defines what this

implementation-specific information might be (for example, whether certain lifecycle control operations are implemented at all).

If the worker in fact has no non-default behavior, there is no need for an OWD: a default one will be generated. This default OWD simply contains a reference to the spec file. E.g., if the RCC implementation based on the spec file "search-spec.xml" had no non-default implementation attributes, the OWD file would be, entirely:

```
<RCCImplementation>
  <xi:include href="search-spec.xml"/>
</RCCImplementation>
```

Note that the name of the implementation defaults from the name of the OWD file itself. Otherwise it can be overridden by the "name" attribute" of the "RCCImplementation" XML element in this file.

3.2.2 The Worker Makefile

The "make" file (named "Makefile") in the worker directory can be as simple as one line:

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

This simply tells "make" that this directory is for building a worker whose name and authoring model are derived from the name of the implementation directory, and whose worker description file (OWD) is named the same as the worker name, with ".xml" as the suffix, and whose source file for the actual implementation code for the worker also has the name of the worker with the suffix for the programming language. The OWD will be generated if it is not supplied.

Thus, if the name of the directory were "search.rcc", then the simplest Makefile above, would assume that the worker description file can be generated and the source code to compile for the worker was "search.c". Note that if the source file is missing, it will be created automatically as a "skeleton" of the implementation that does nothing, but compiles as a valid worker of the given authoring model.

The worker makefile can also have other variable settings specific to the authoring model, and also may list other source files and libraries needed to compile the worker into the appropriate binary file format (e.g. a shared object ".so" file for RCC workers on linux, or a ".ngc" file for HDL workers using Xilinx XST synthesis tool).

3.2.3 The Worker Source files

The worker source files must be written according to the authoring model, but as a starting point, if no such file exists, you can use "make skeleton" to create an empty skeleton of a worker implementation that will in fact compile, build and execute (doing nothing). If additional source files should be compiled and linked to make the binary file in an implementation directory, the "make" variable "SourceFiles" can provide a list of files in addition to the ones named for the workers being built. For example if the file "utils.c" was used in an RCC implementation directory that was building workers w1 and w2 into the same binary file, the makefile would be:

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

In this case 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. If both workers have default implementation aspects, the xml files will be generated.

3.2.4 Creating a new worker

To create a new worker (component implementation) called "*xyz*", with the authoring model "am", after the "specs" file exists in the specs directory (*xyz*-spec.xml), the following make command is used in the top level directory of the component library:

```
make new Worker=xyz.am
```

This generates the subdirectory for that implementation, creates the appropriate initial "Makefile", and builds a compilable skeleton of the worker implementation.

An additional "Language" attribute may be supplied for authoring models that support multiple languages. E.g.:

```
make new Worker=xyz.hdl Language=Verilog
```

At this point you could then edit the `xyz.rcc/xyz.c` file (assuming the RCC model) to make it contain something more interesting that then empty skeleton that does nothing. Even before editing the skeleton, it can be "built" using "make xym.am".

```
make xyz.am
```

If you need to use a spec file different than "*xyz*-spec.xml", you can set the SpecFile variable on the same command, e.g.:

```
make new Worker=xyz.hdl Language=Verilog SpecFile=specs/other-spec.xml
```

3.3 The Component Library Makefile

The make file (called "Makefile") in the top-level directory of the component library is basically a list of implementations that should be built, and the target platforms that each authoring model should be built for. The last line in the file should establish it as a Makefile for an OpenCPI component library and be:

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

Note that the OCPI_CDK_DIR variable must be set prior to this either in the environment or as a variable in the makefile to point to the CDK installation. Before the "include" line above, several "make" variables can be set in the file. The most important one is "Implementations" (or "Workers"), which is a list of which worker subdirectories to be built for this component library. If that variable is not set at all, then it is assumed that all subdirectories of the component library whose suffix is one of the known authoring models, should in fact be built. For example:

```
Implementations=fft.rcc fft.hdl fft-for-xilinx.hdl fir.rcc
```

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 implementations in this library. The default package name is "local", but for libraries used outside the local organization, it should generally be the reverse internet domain name similar to the name spaces used for Java classes. E.g.:

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

The package name "ocpi" is reserved for OpenCPI component specifications.

Finally, for each authoring model, there must be a list of targets to build for. I.e. for the RCC authoring model, the variable "RccTargets" would be set to a list of targets to build all RCC workers for. For all software (not HDL) authoring models, the default target, if none is specified, is the machine and operating environment of the machine doing the building. Other software targets would use cross-compilers.

Other non-software authoring models (for processors that will never be the one running the tools), have other default targets (see each one).

Software targets generally use the format: *os-version-processor*. The ***os*** part is something lower case like "linux" or "macos". The ***version*** part is determined by whoever sets up tools. For linux, it is typically a letter for a distribution followed by a major number (e.g. "c6" for centos 6, "r5" for RedHat 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 section below.

Thus if all subdirectories containing workers should indeed be built, and the desired build targets are the default ones, and the package name is the default, then the single "include" line above is sufficient to built a component library. Thus creating a new component library "mycl" can be accomplished by the script:

```
mkdir mycl mycl/specs
echo 'include $(OCPI_CDK_DIR)/lib.mk' > mycl/Makefile
```

3.4 An Example Component Library

Here is a file hierarchy of a component library “mycl” with a “search” component with RCC and HDL implementations, and a “transform” component with only an XM (X-midas fortran) implementation (before any compilation):

```
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.v     (HDL verilog source file)
    /transform.xm/Makefile
                 /transform.xml
                 /transform.f
```

3.5 Library Exports

When a component library is built, all the workers are compiled and the binary "artifact" files (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. Thus the build process creates an "export" directory to be used by application developers. The export subdirectory is thus 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 and referenced by applications.

4 HDL/FPGA Development

HDL development in OpenCPI includes both “application workers” in a component library, which perform functions independent of any specific hardware attached to the FPGA, as well as “device workers” that are designed to support specific external hardware such as ADCs, flash memories, I/O devices, etc. Device workers are developed as part of enabling a platform (an FPGA on a particular board) for OpenCPI. The development of device workers is described in a separate “HDL Platform Development” document.

The sections below for HDL/FPGA development are:

- Developing application workers in a component library
- HDL build targets: building for different target devices and platforms
- The HDL Build Hierarchy: how whole device “bitstreams” are created.
- Developing assemblies of workers on FPGAs.

4.1 Developing HDL application workers

The process of writing a new worker (after the OCS is written), starts with the "make new Worker=xxx" command described above, with the "Language" option set to "vhdl" or "Verilog" (case insensitive). Assuming you already wrote the "specs/foo-spec.xml" file, you could issue, in the component library's directory:

```
make new Worker=foo.hdl Language=vhdl
```

4.1.1 Worker directory contents

The above command establishes the worker's subdirectory (foo.hdl), its Makefile, its Worker description file (foo.xml), and its compilable skeleton source code file (foo.v or foo.vhd). After issuing this command, before writing any code, it is recommended that you compile the skeleton worker that is automatically generated. Assuming the desired HdlTargets variable is specified in the component library's Makefile, this just means issuing:

```
make foo.hdl
```

There should be no errors unless the specs file somehow used an unsupported feature of the tools. As with any type of worker, compilation output is placed in the target-*TTT* subdirectory of the worker, for each target mentioned in the HdlTargets variable. This variable can also be set on the command line:

```
make foo.hdl HdlTargets="modelsim virtex6"
```

This would compile (and for non-simulation targests, synthesize) the worker.

In the "foo.hdl" worker directory there will be a skeleton VHDL or Verilog file that you can edit to add the functional source code (sometimes called the (ugh) "business logic") of the worker.

The "gen" subdirectory of the worker's directory contains all the files automatically generated by the "make new" above, but none of these files should be edited.

For VHDL workers, the skeleton file that you *do* edit contains only the "architecture" of the worker. It is named "foo.vhd" in this "foo.hdl" directory. The "entity" declaration of the worker was automatically generated for you, and is found in the generated file "gen/foo-impl.vhd". So at this point, the directory representing the new worker looks like this:

```
foo.hdl/                          directory for new worker
        Makefile                  Makefile for this worker
        foo.vhd                   Source code to edit
        gen/foo_impl.vhd          File containing entity declaration
        gen/foo.xml               Default OWD for this worker
        gen/…                     other generated files
        target-TTT/…              compilation output files
```

In some special cases, you may have to edit the Makefile here, but you should never edit any files in the "gen" or "target-*" subdirectories.

4.1.2 Worker Makefile

The worker Makefile is usually not hand-edited, but there are a few cases where editing is necessary. If your worker will consist of multiple source files, you can add VHDL or

Verilog source files (in addition to the skeleton file) used by specifying the "SourceFiles" variable in the Makefile. This variable can refer to additional files placed in this directory that are subsidiary to the worker entity/architecture itself. Here are the variables you can set in this Makefile:

Variable Name in Makefile	**Override/augment component library Makefile?**	
SourceFiles	N	A list of *additional* source files for this worker.
Cores	N	A primitive core built elsewhere
Libraries	Y	A primitive library built elsewhere
OnlyTargets	Y	A list of the only targets for which this worker should be built
ExcludeTargets	Y	A list of targets 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 file in the component library)
VerilogIncludeDirs	Y	Searchable directories for Verilog include files
Worker	N	Name of worker, default is from directory name

When using the "Cores" and "Libraries" variables, if the name has no slashes, it is assumed to be in the OpenCPI hdl/primitives directory. The variables with "Y" in the table above are those than can be specified in the component library's make file to provide a default for all workers in the library.

4.1.3 Worker Description File: the OWD XML file.

This file specifies characteristics for this implementation of the spec file (OCS). In many cases it can be empty or left as it is. A default version is generated for you in the "gen" directory, called "foo.xml". If you don't need to modify it, you can leave it there. If you do need to modify it to add more information about this implementation/worker, you must copy it into the worker's directory and edit it *there*.

The primary reasons to customize this OWD XML file are:

- ***Add implementation-specific properties***
  You can add additional worker properties for the implementation, beyond what is in the spec file. The "property" XML elements accomplish this, with the same xml format as in the spec file.
- ***Add more accessibility to spec properties***
  You can add more access capabilities for existing properties, via the "specproperty" element. E.g. make a property that is write-only in the spec to be also "readable" in the implementation for debug purposes. The only attributes are "name" and the access properties: readable, writable, volatile, initial.
- ***Specify interface style and implementation attributes for data ports***
  You can specify whether the port uses a stream or message interface, and provide additional details for those interfaces (e.g. data path width, or whether the port supports aborting messages).
- ***Specify which control operations that this worker will implement***
  You can specify which control operations are in the implementation: only the "start" operation is mandatory and required, and has a default implementation provided.

The top level of the HDL OWD is the XML “HdlImplementation” (or “HdlWorker”) element, which can have the following optional XML attributes. None are typically needed, and are only specified when the default behavior must be overridden.

HdlImplementation Attribute Name	Data Type of value	
Name	string	The case insensitive name of this worker, using letters appropriate to language identifiers. The name is optional with the default taken from the name of the OWD file (without .xml). It must match the filename in any case at this time.
Pattern	string	An external signal naming pattern (described below this table) for all signals of the worker. The default is “%s_”, which indicates a prefix of the port name followed by underscore. “External” signals are those defined using the OCPIP interface standard, not the inner “worker” signals.
PortPattern	string	A port naming pattern used when port names and signal (not data) direction are used in the generated code. I.e. for each worker port, a naming pattern is defined both for input signals and output signals of the port. The default is “%s_%n”, which indicates a prefix of the port name followed by underscore, and then “in” or “out” for signals that are “input” to the worker or “output” from the worker.
Language	string	The case insensitive HDL language used for generated code, including the skeleton file. The default is “vhdl”.
DataWidth	unsigned	The default width of data ports for this worker. Any individual port can override this. The default value when this attribute is not specified is based on the protocol (messages) defined for the port.
SizeOfConfigSpace	unsigned 64-bit	The size of the configuration space in bytes is overridden to be the given value. The default is based on the actual properties.
ControlOperations	string list	A comma-separated list of control operations that this worker will support. The default is to only support the “start” operation.
Sub32BitConfig-Properties	Boolean	Whether this worker needs to address items smaller than 32 bits (and thus require byte enables in its interface). The default is based on the actual properties.
RawProperties	boolean	A boolean value indicating whether the implementation will use the “raw” property interface for all properties. The default is “false”, unless the “FirstRawProperty” attribute is set. The “raw” interface is described below under “control interface signals” and is normally only used for device workers.
FirstRawProperty	string	A string value indicating the name of the first property that requires the “raw” property interface. Properties before this use the normal property interface described below.

4.1.4 The Authored Worker: the VHDL "architecture" or Verilog "module body"

The "functional code" or "business logic" of the worker is in the architecture section (VHDL), in the "foo.vhd" file (and possibly subsidiary source files). In Verilog, it is in the body of the module, and the file is "foo.v". The entity/module being implemented at this level starts life as the generated skeleton of the "***inner*** worker", meaning that it is surrounded by an automatically generated logic "shell" to provide robust and composable interfaces compliant with the Open Core Protocol interfaces defined for the ***outside*** of all workers. Thus an outer shell is generated around this "inner worker". This shell and the entity declaration for the "inner worker" are found in the generated file: "gen/foo-impl.vhd" file for VHDL and "gen/foo-impl.vh" for Verilog. The skeleton file consisting of an empty "inner" worker, becomes the "authored worker" when the functional logic is written/edited into that file.

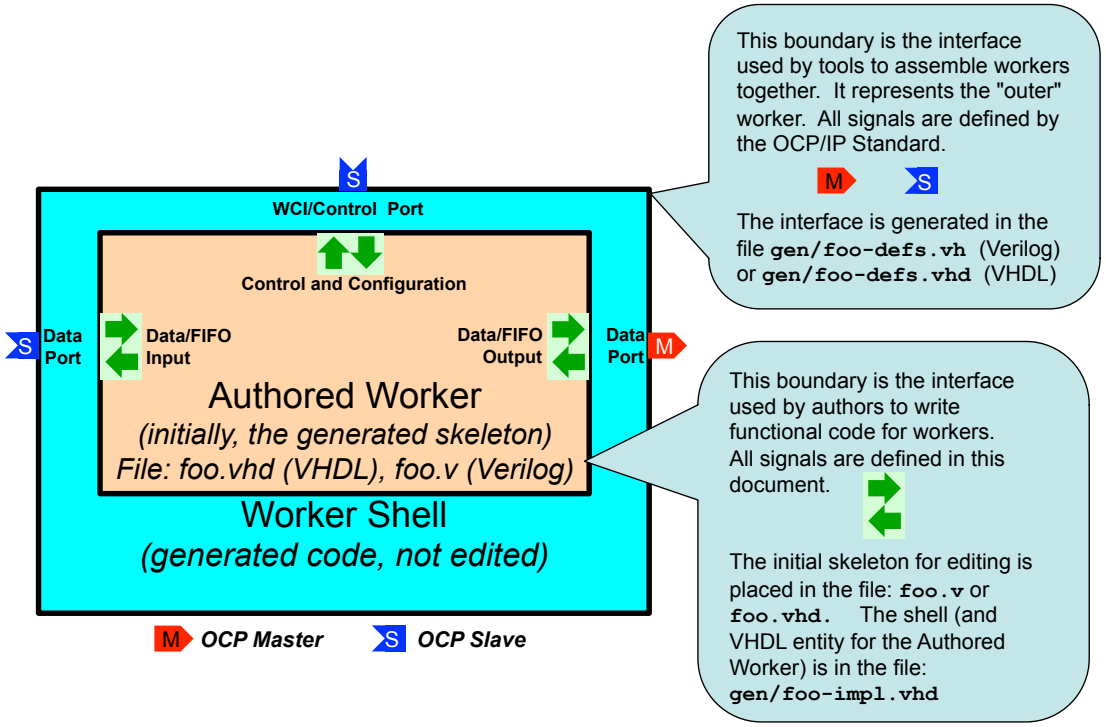

Figure 2: Worker Code and Files

All OpenCPI HDL workers are characterized by their properties, their ports and their clocks, and usually the clocks are simply associated with ports, or even more simply, a single clock is frequently used with all ports. Thus the job of implementing the inner worker is the job of processing the various ports' inputs to the worker to produce the various ports' outputs of the worker. For each port of the worker (including the control port) there are input signals (into the worker) and output signals (out of the worker).

In VHDL, these groups of inputs and outputs are in a record type. Thus for each port (whether control, data producing, data consuming or other service), there is an "input signal" record and an "output signal" record, named <port>_in and <port>_out, respectively. In Verilog there are no record types, so individual signals simply have the <port>_in_ and <port>_out_ prefixes instead.

E.g., with a “filter” worker that has a “sensor” input port, and a “result” output port, the VHDL entity declaration would be:

```
entity filter_worker is
  port(
    ctl_in             : in  worker_ctl_in_t;
    ctl_out            : out worker_ctl_out_t;
    sensor_in          : in  worker_sensor_in_t;
    sensor_out         : out worker_sensor_out_t;
    result_in          : in  worker_result_in_t;
    result_out         : out worker_result_out_t);
end entity filter_worker;
```

The actual individual signals in each record depend on the contents of the spec XML file (OCS) and the implementation XML file (OWD). These signals will be described below. Note that the name of the “control port” defaults to “ctl”. An example skeleton file for this worker would be:

```
library ieee; use ieee.std_logic_1164.all, ieee.numeric_std.all;
library ocpi; use ocpi.types.all;
architecture rtl of filter_worker is
begin
  -- put the logic for this worker here
end entity filter_worker;
```

Note that while the overall worker has the name “filter”, the entity being implemented in the architecture here is “filter_worker”, the “inner” worker.

4.1.5 Signal naming conventions and data types

Other than the property value signals described below, the signals in these interfaces are mostly a combination of std_logic_vector and a boolean type, “bool_t” that is used for various boolean indicator signals.

This VHDL type “Bool_t”, acts as much like the VHDL type “Boolean” as possible (with various operator overloading functions), while still being based on std_logic. The to_boolean and to_bool functions explicitly converts to and from the VHDL BOOLEAN type, respectively. The “its” function is a convenient synonym for the “to_boolean” function, enabling code like:

```
if its(ready) then
   …
end if;
```

There are also two constants for this type, btrue and bfalse. All VHDL types defined are in the “types” package of the “ocpi” library.

All signals into and out of the authored worker are in the “in” and “out” records of each port.

All data types created by OpenCPI use the “_t” suffix. All enumeration values created by OpenCPI use the “_e” suffix.

4.1.6 The control interface/port to the authored worker.

Every HDL worker has a control port interface that performs three functions:

- Provide a clock and associated reset
- Convey life cycle "control operations" like "initialize", "start" and "stop".
- Access the worker's configuration properties as specified in the OCS and OWD.

In VHDL, when the default name of the control port is used ("ctl"), the input signals are essentially prefixed with "ctl_in." and the output signals are prefixed with "ctl_out." I.e. the input signals are in the "ctl_in" record port, and the output signals are in the "ctl_out" record port.

4.1.6.1 Clock and reset in the control interface

The signal "clk" is the clock for all other control signals as well as the default clock for all other data ports of the worker. The "reset" signal (asserted high), is asserted and deasserted synchronously with this clock. The reset is guaranteed to be asserted for 16 clock cycles. When 16 clocks are not enough to perform initialization, the worker should implement the "initialize" life cycle control operation (see below). Reset is initially asserted.

If the worker (in its OWD) declares that other ports have clocks that are asynchronous to this control clock (i.e. those ports operate in a different clock domain), the worker implementation code takes responsibility for the appropriate synchronizations between this control clock (and its associated signals), and any other signals related to the data or service ports. In particular, it is the worker's responsibility to propagate this control reset to the reset outputs associated with those other ports, in the clock domain of those ports.

4.1.6.2 Lifecycle/control operation signals in the control interface

Other than the control "reset" signal, the lifecycle of all workers is managed by life cycle control operations, according to the diagram below. However, in the simplest common case, when the worker has no need to implement any of these operations, there is a single simple input signal that indicates when the worker should "run", called "is_operating". I.e., after reset is deasserted, the worker should "operate" only when this "is_operating" signal is asserted. Many HDL workers simply use this signal (as well as clk and reset) and no others in the control interface. The "is_operating" signal is a convenience indicating that the worker has been "started" and should now "do its thing".

If the "initialize" control operation is not implemented, then when the reset signal is deasserted, the worker is considered to be "initialized". If "initialize" is implemented, it is considered in the "exists" state after reset is deasserted.

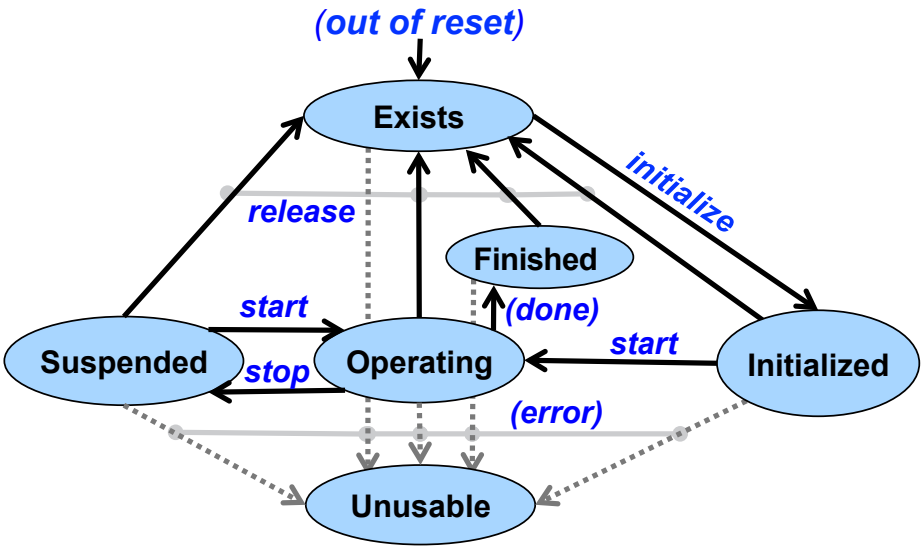

Figure 3: Control Operations and States

When a worker needs to explicitly support other control operations, there are 2 input and 2 output signals it can use. The "control_op" signal is a VHDL enumeration value that conveys which control operation is in progress. When there is no operation in progress, it has a value of "no_op_e". The operation is terminated by asserting the "done" or "error" output signals, after which the control operation is considered accomplished (if "done") or not (if "error"). Note that the "done" signal is driven to a default value of "btrue" in the entity declaration (and the "error" signal defaults to "bfalse"). Thus the worker does not need to drive these at all if it will always perform the control operations in a single cycle and will never need to assert "error".

A common example of a control operation might be when the worker needs multiple clock cycles to accomplish something like "initialize" or "start". In that case it notices when the "control_op" signal changes from "no_op_e", and then performs the operation, asserting "done" after the operation has completed (or "error").

There is another convenience input signal, called "state" that indicates which lifecycle state the worker is in. It changes when control operations succeed. It is a VHDL enumeration value: EXISTS_e, INITIALIZED_e, OPERATING_e, SUSPENDED_e, and UNUSABLE_e.

Finally, there are two control output signals that the worker can use to indicate two other conditions. The first, "attention", allows the worker to indicate an interrupt or other condition to control software. There is no particular definition of what this means, and its usage generally requires software to monitor it, and perhaps access some volatile property values to determine what caused it. When or how it is deasserted is also implementation-dependent, although it should be deasserted on reset.

The second additional control output signal is "finished". The worker uses this to indicate it has entered the "finished" state, and will perform no further work. This is a way for the worker to tell control software that its work is finished and perhaps that the

application the worker is part of can be considered “finished”. This signal should be deasserted upon reset.

Here is a summary of the control interface signals in the “ctl_in” record:

Signal	Type	Description
clk	std_logic	The clock for the control interface and the default clock for all other interfaces and ports.
reset	std_logic	The reset signal, asserted high and synchronously, for the control interface. Initially asserted.
control_op	control_op_t	An enumeration type that specifies the currently active control operation, with the value “no_op_e” when there is no active control operation. Control operations end when the “done” or “ready” signal in the ctl_out record is true.
state	state_t	An enumeration type that indicates the control state of the worker.
is_operating	bool_t	An indication that the worker has been started and is thus in an operating state, e.g. when state = operating_e
abort_control_op	bool_t	A command indicating that a long-duration control operation is being forcibly aborted.

4.1.6.3 Property accesses

A worker’s configuration properties are accessed via two additional “record” signals, called “props_in” and “props_out” (separate from the ctl_in and ctl_out records for the control interface). The individual signals within these records depend entirely on what types of properties have been declared in the OCS and OWD. In the tables below, for a property called “foo”, the signals will be present as described. The signals possibly present in the “props_in” record are:

Signal in props_in	Signal included when property is:	Type	Signal Description
foo	Writable or Initial	*	The registered value last written by control software. The type is dependent on the property type.
foo_length	Writable or Initial: and sequence	ulong_t	The registered 32 bit unsigned (ulong_t) length of the value (number of elements in the sequence) when property is a sequence type.
foo_written	Writable	bool_t	Indication that the entire value has been written.
foo_any_written	Writable and (array or sequence or string)	bool_t	Indication that any part of the value is being written.
foo_read	Volatile or (readable and not writable)	bool_t	Indication that the property is being read.

All the “indication” signals are valid for the length of the access operation (until “done” or “error”). Any writable property will be registered in the “shell” when written, even when the property is volatile and the worker is supplying a volatile value for reading in the props_out record.

The signals possibly present in the “props_out” record are:

Signal in props_out	Included when property is:	Description
foo	Volatile or (readable and not writable)	The worker-supplied value of the property, with the type dependent on the property declaration.
foo_length	Volatile or (readable and not writable)	The worker-supplied (ulong_t) length of the value (number of elements in the sequence) when a sequence type.

4.1.6.4 Property Data Types

The VHDL data type of the property value is the type as specified in the OCS XML definition of the property with the suffix “_t”. All signed and unsigned integer types are based on the IEEE numeric “signed” and “unsigned” types. I.e. uchar_t is the IEEE unsigned type of 8 bits. Also, a conversion function is defined to convert the VHDL “integer” and “natural” types into the numeric property types. I.e. to_uchar converts a VHDL “natural” to a uchar_t. These types are char_t, uchar_t, short_t, ushort_t, long_t, ulong_t, longlong_t, ulonglong_t. For the “char_t” type, which is an 8 bit IEEE signed type, there are also conversion functions (to_char) defined to convert from the VHDL CHARACTER type to char_t, and to_character to convert from char_t to the VHDL CHARACTER type.

The float_t and double_t types are just the appropriately sized std_logic_vector.

The string_t type is a null-terminated array of char_t types. The “to_string” conversion function can convert from a VHDL STRING type to a string_t.

When a property type is an array or sequence, the value is a VHDL array of the appropriate type, named <basetype>_array_t, with a range of (0 to size-1).

For each property type there is a constant declared for the minimum and maximum values, named <basetype>_min and <basetype>_max, respectively.

All these types, conversion functions and constants are in the ocpi.types package (in the ocpi library). This package is defined in the OpenCPI source tree at hdl/primitives/ocpi/types_pkg.vhd.

4.1.6.5 Raw access to properties

There is an alternative method for workers to support reading and writing of properties that is usually only used for device workers for devices with built-in hardware registers. Raw access is enabled by setting the “rawProperties” attribute or the “firstRawProperty” attribute in the OWD. If “rawProperties” is true, then all properties are “raw”. If “firstRawProperty” names a property, then properties before that one are accessed using the mechanism described above, but the named property and later properties are accessed using the “raw” interface.

The input signals (in the props_in record) for the raw property access interface are:

Signal in props_in	**Signal included when:**	**Signal Description**
raw_address	Always	The byte offset from the first raw property of the property being accessed.
raw_byte_enable	Some raw property is less than 32 bits.	The (4) byte enables for reading and writing bytes within the 32 bit data of the control interface.
raw_is_read	Some raw property is readable	An access operation is reading a raw property.
raw_is_write	Some raw property is writable	An access operation is writing a raw property.
raw_data	Some raw property is writable	The data being written to the raw property.

The output signals (in the props_out record) for the raw property access interface are:

Signal in props_out	**Signal included when:**	**Signal Description**
raw_data	Some raw property is readable	The data value for the raw property being read.

When raw properties are being accessed, the “done” and “error” control interface signals are used indicate when the access is complete (and for reading, when the props_out.raw_data is valid).

4.1.7 Streaming Data Interfaces/Ports to/from the Authored Worker

Worker data ports (as specified in the OCS) can be implemented in two different styles. One is “streaming” and the other is “message”. Streaming interfaces (WSIs) are basically FIFOs with some metadata. Message interfaces are based on addressable message buffers, and are described further below.

The default style of data interface is the streaming interface, and, if all attributes of the implementation are default, no indication of this style choice is needed in the OWD. When attributes are required, this style is indicated by including a <streaminterface> element in the OWD. Recall that the “datawidth” attribute at the top level of the OWD specifies the default width of all data interfaces. Thus if the only attribute you need to specify for an interface is a width that is the same as other data ports, it may be most convenient to specify this width at the top level HdlWorker.

An example of the per-port XML element is:

```
<StreamInterface name=”sensor”
                 datawidth=’64’
                 preciseBurst=’true’/>
```

The possible XML attributes of the stream interface element are:

Attribute in StreamInterface	Attribute data type	Attribute Description
name	string	The name of the port in the OCS being described. Required.
datawidth	unsigned	The width of the data path for this interface. The default is the smallest element in the message protocol indicated in the OCS, unless overridden by a default datawidth attribute at the top level of this OWD (HdlWorker)
impreciseBurst	boolean	Whether this interface requires (for input) or produces (for output) imprecise bursts. The default is true unless preciseBurst is set to true, in which case the default is false. When true it means it can accept imprecise bursts on input and may produce imprecise bursts on output. Most interfaces use imprecise bursts. Precise burst are those that present the length of the entire message at the start of the burst.
preciseBurst	boolean	Whether this interface requires (on input) or may produce (on output) precise bursts. The default is false. If a worker only supports precise bursts on input, it may require extra buffering to be inserted. If a worker can easily (without extra overhead) support precise bursts on output, it should.
abortable	boolean	On input, that aborted messages are tolerated. On output, that the worker may abort messages.
continuous	boolean	This port must receive, or will produce, continuous data, with no idle cycles within a message.
earlyrequest	boolean	For this port, this worker may receive, or may produce, the start of message indication before any valid data.
pattern	string	Signal naming pattern. This only applies to external signals, and not the inner worker signals. The default is “%s_”, which is simply the port name as prefix with an underscore.
clock	string	The name of the clock to be used with this port. The name is either the name of another port with a “myclock” attribute of true, or the name of a clock declared for the worker as a whole, that is not the control clock.
myclock	boolean	An indication that this port operates in its own clock domain, and will have its own “clk” signal as input.

Although a number of attributes of data interfaces are inferred/derived from the message protocol specified in the OCS, the OCS may not have a message protocol or certain attributes may need to be overridden. Below are those attributes that can be specified in a StreamInterface element. The indicated defaults apply when there is no message protocol in the OCS.

Attribute in StreamInterface	Attribute data type	Attribute Description
numberOfOpCodes	unsigned	The number of distinct message types (opcodes) for the interface. The default is 1, and the maximum is 256.

maxMessageValues	unsigned	Largest number of data values in a message. Default is 64K.
dataValueWidth	unsigned	The size of the atomic units of data on this interface (size of a byte). The default is the value of the "datawidth" attribute.
zeroLengthMessages	boolean	Whether the interface will support zero length messages. Default is false.

Specifying the StreamInterface attributes determines the signals present in the input and output records for the interface. The signals for stream interfaces (input or output) are:

Signal	**When Included**	**Direction**	**Signal Description**
clk	When myClock or Clock	Input	The clock for all signals in this interface. When no specific clock for this interface is specified, the ctl_in.clk signal is used.
reset	Always	Input	The connected peer worker is asserting its reset output into this worker for this port. This should reset any message state but perhaps still allow control accesses to properties.
ready	Always	Input	A data value and/or message boundary is present. On input: one or more of valid, som, eom, or abort is asserted; worker is allows to assert "take". On output: permission to "give", with worker asserting one of: valid, som, eom or abort.
data	If datawidth > 0	Data direction	The input or output data, when "valid" is true.
som	Always	Data direction	The start-of-message indication is present.
eom	Always	Data direction	The end-of-message indication is present
valid	If datawidth > 0	Data direction	The data signals hold message data
abort	If Abortable	Data direction	The message is being aborted
byte_enable	datavalue-width < datawidth	Data direction	Which data bytes are valid, usable when valid is true.
opcode	numberOf OpCodes > 1	Data Direction	The opcode (or message type) for the current message. Valid from start of message to end of message on input.
give	Port is output	Output	Indicates that data should be taken by output port, only permissible when "ready" is true. Worker is saying "I give". Like an "enqueue" signal to a FIFO.
take	Port is input	Output	Indicates that data is being taken by worker, only permissible when ready is true. Worker is saying "I take". Like a "dequeue" signal to a FIFO.

4.1.8 Message Data Interfaces

This interface is used when an OWD specifies a MessageInterface element associated with a data port in the OCS. It provides an alternative mechanism to consume or produce data that uses an addressable message buffer interface. This enables the worker to produce or consume message data out of order, or to only access parts of a message. The signals provided with this interface allow the worker to address specific locations in the current buffer, and then signal that it is done with the current buffer.

[signals table here]

4.1.9 MemoryService Interfaces

This interface provides access to memory. [To be specified]

4.1.10 Time Service Interfaces

This interface provides “time of day” information to the worker, to the precision requested in the OWD via attributes to the “timeinterface” element. [To be specified]

4.2 HDL Build Targets

Build targets answer the question: what target device (or family of devices) am I building for? (compiling, processing, synthesizing, place-and-routing, etc.) When building any level of modules for FPGAs, the build targets are specified via the ***HdlTargets*** and ***HdlPlatforms*** variables. The "targets" are chips or chip families, whereas the "platforms" are actual FPGAs on specific boards. These build targets are defined in a hierarchy with these levels:

Top level, vendor level: this level specifies vendors (Xilinx, Altera, Achronix), as well as vendor-independent simulators (Icarus, Verilator, Modelsim). Thus HDL assets can be "built for Xilinx" or "built for Icarus". This implies building for all lower level targets under these top-level labels. The label "all" specifies all top level targets.

Family level: this level specifies the family of parts under the vendor level. Different part families typically have different on-chip architectures, and may drive tools differently. Building for a family target means generating libraries or cores that are suitable to any member (part) in the family. Examples would be "virtex5" or "spartan6" or "stratix4". Simulation targets at the top level don't have families (yet) so these top two levels are the same for simulation.

Part level: this level specifies the specific part that the design is targeted at, such as xcv5lx50t. This level does not include package information but may include speed grades.

Anywhere that HDL building takes place, these two Make variables can further filter the targets that are built:

ExcludeTargets/ExcludePlatforms: this variable specifies targets to be excluded, usually because they are known not to be buildable for one reason or the other (a tool error, or other incompatibility).

OnlyTargets/OnlyPlatforms: this variable specifies targets to be exclusively included, because it is known that only a limited set of targets should be built (e.g. a coregen core specific to a particular family or part).

The ***HdlPlatforms*** variable specifies HDL platforms (like Xilinx ml605), which imply the appropriate family and part. I.e., if you specify to build for a platform, it will build primitives and workers for the appropriate family.

4.3 The HDL Build Hierarchy

OpenCPI FPGA bitstreams (the files that configure entire FPGAs) are built in several layers. The same layers apply to building executables for simulation. The following diagram shows the build flow and hierarchy.

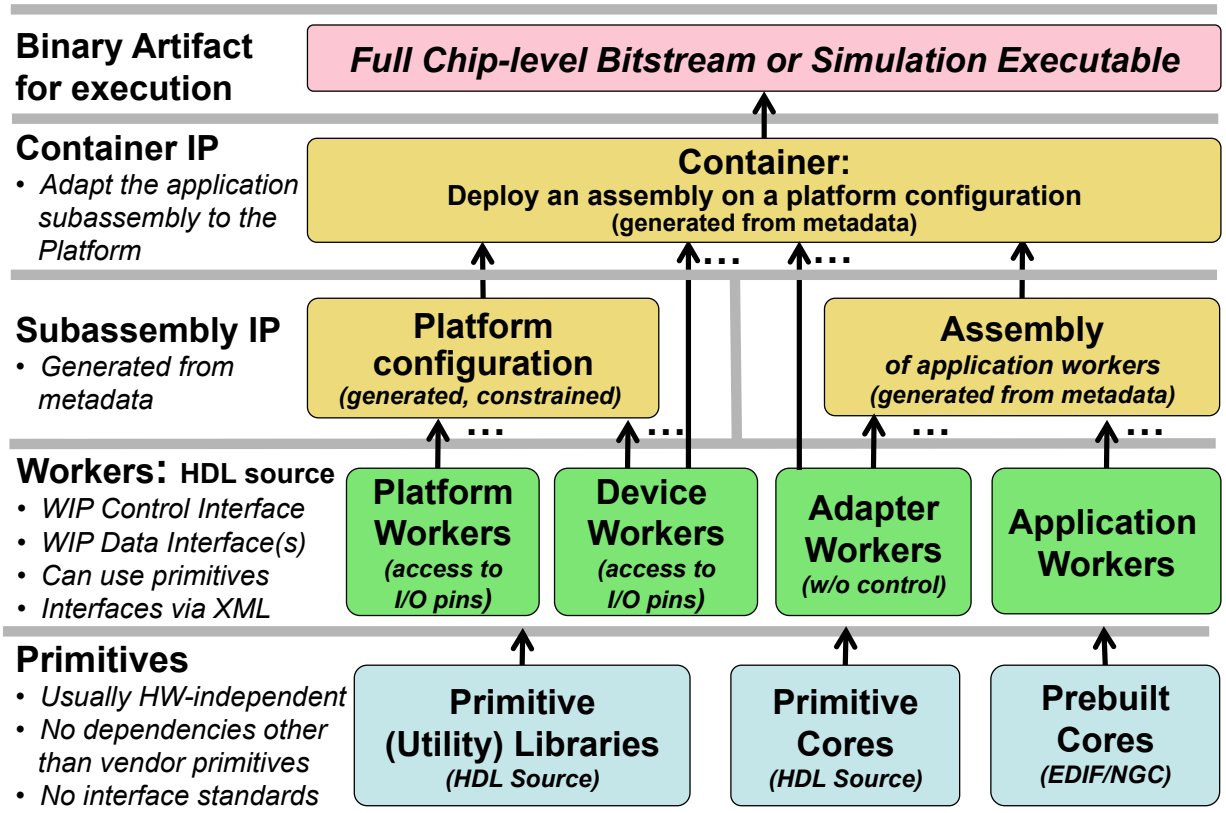

Figure 4: OpenCPI HDL Build Flow Layers

4.3.1 HDL Build Layers

At the bottom (built first, used by all other layers), are “primitives”. These are low level, “leaf” libraries and cores used by higher levels. Primitive libraries are libraries of modules built from HDL source code that are available to be used higher up the hierarchy; using a primitive library in a higher level module does not imply all the modules in the library are brought into the design, but only pulled in as needed by references in the higher levels of the design. Primitive “cores” on the other hand are single modules built from source or generated from tools such as Xilinx Coregen, which are also used in higher levels of design. They are explicitly included in higher-level designs. Primitives may in fact depend on each other: a core may depend on primitive libraries, and primitive libraries may depend on other primitive libraries. Circular dependencies are not supported.

There are primitive libraries specific to vendors and families that can be used for implementing primitives using vendor-specific elements. More detail on such primitive libraries are in the HDL Platform Development document.

Above the “primitives” layer there are workers (HDL component implementations) of several types. All types of workers can use primitive libraries or cores as required. Application workers are generally portable and hardware independent. Device workers are workers that also connect to the I/O pins of external hardware, and in some cases can attach to vendor-specific on-chip structures (e.g. ICAP on Xilinx). Adapter workers are used when two connected workers are not connectible in some way due to different interface choices in the OWD (e.g. width, stream-vs-message, clock domains). Platform workers are special device workers that perform platform-wide functions.

The application assembly is automatically generated HDL source code that uses application workers and adapter workers. The application assembly itself is described in metadata (XML) as an assembly of connected application workers. It represents the subset of an overall heterogeneous OpenCPI application: a subassembly that will be executed on a single FPGA.

The platform configuration is automatically generated HDL source code that uses platform-specific IP along with some device workers. It represents a platform configured with built-in support for some devices, and may include various constraints and physical design. For those familiar with Linux kernels, a platform configuration is analogous to a “built/configured kernel with some device drivers built-in”.

The container adapts the application assembly to a platform configuration and provisions any additional required device workers. It connects and adapts the “external I/O ports” of the application (sub)assembly to the available I/O paths and devices in the platform. When the deployment of the application assembly requires device workers that are not in the platform configuration, they are instanced in the container itself. Here the Linux kernel analogy is that these extra device workers are analogous to the “dynamically loaded device drivers used to run the application”.

Thus device workers can either be built into the platform configuration or instanced in the container.

The final design for the entire FPGA combines the container logic with the application assembly and the (presumably prebuilt) platform configuration. This hierarchy (except primitives) is shown in the following diagram.

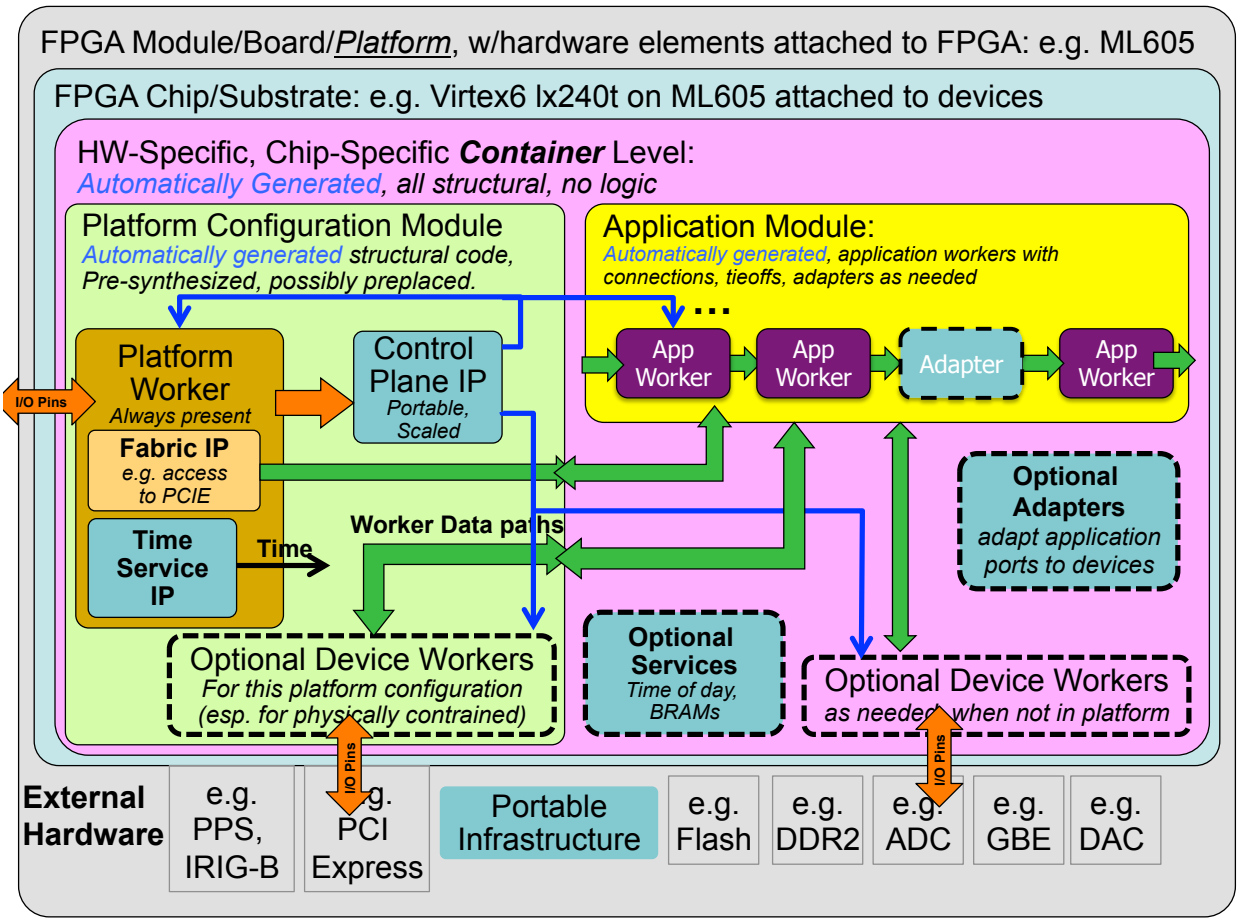

Figure : HDL Full FPGA Hierarchy

When tools support it, each layer in the build is actually synthesized or precompiled or elaborated as the tools allow (i.e. “prebuilt”). Thus:

- Each worker in a component library is prebuilt into a core (possibly using primitive libraries)
- Assemblies are prebuilt from generated VHDL or Verilog code and the required worker cores
- Platform configurations are prebuilt from platform IP, and device worker cores
- Container top levels are build from platform configurations and assemblies, with any additional device workers, service modules and adapters as required.
- Full bitstreams (or simulation executables) are built from the container modules.

This “layered prebuilding” allows the results to be reused at the next level without recompiling or resynthesizing, all in a vendor independent fashion. E.g. an assembly prebuilt for a Xilinx virtex6 part can be reused to target different virtex6-based platforms. The exact definition of “prebuilding” varies with different tool chains, and the level of “synthesis optimization” that happens at each step also varies by tool, and some of this level of “hardening” at each level is controllable for some tools.

At one extreme, “prebuilding” simply means remembering which source files must be provided to the next level (for tools that have no precompilation of any kind). At the other extreme are tools that can incrementally synthesize to relocatable physically mapped blocks on a family of FPGA parts.

Simulators are considered “platforms” that act as test benches for assemblies. This is described in more detail below in the simulation section.

4.3.2 HDL Directory Structure

In the OpenCPI code tree, there is a top- level “hdl” directory under which are the following directories:

primitives: This directory contains subdirectories for each primitive library or core.

devices: This directory is a component library containing subdirectories for each HDL device worker for devices optionally used based on application assembly requirements. Software emulators for some of the devices are also in this component library.

adapters: This directory is a component library containing subdirectories for each HDL adapter optionally instanced to adapt workers that otherwise could not be connected to each other. Adapters are workers with no control interface.

platforms: This directory contains subdirectories for each platform. Platforms are either a specific FPGA chip part on a circuit board with attached devices, or simulators. This is where platform-specific worker code exists, and where platform configurations are specified and built.

assemblies: This directory contains subdirectories for each assembly of application workers, and is where containers deploying these assemblies on platforms are built into bitstreams and simulation executables.

Application workers (for all authoring models) are found in component libraries, either private ones, or in the “components” top level directory of the OpenCPI code tree, which is a heterogeneous library of workers in various authoring models. For application component development separate from the opencpi tree, you might have some or all these directories in your own development area.

This directory hierarchy (at the time of the writing of this document) is:

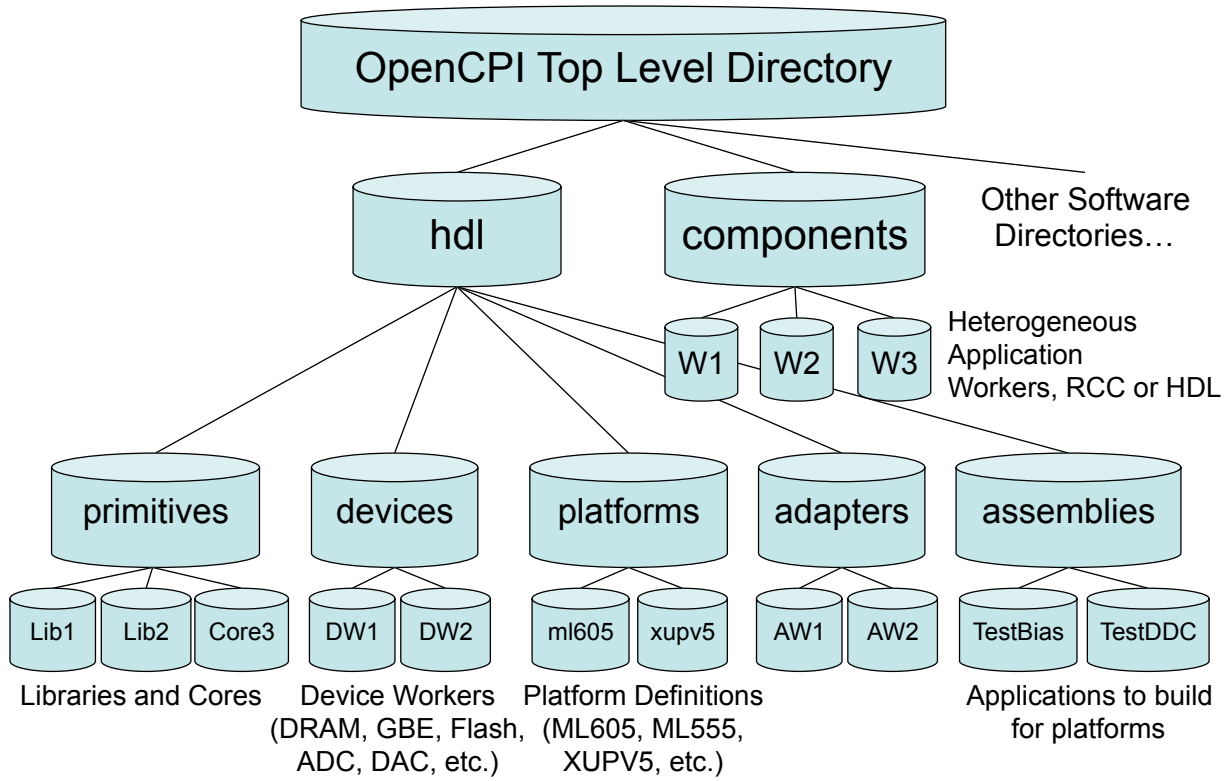

Figure 5: OpenCPI HDL Directory Structure

4.4 HDL Primitive and Assembly build scripts (and Makefiles)

While building typical software workers consists of compiling and linking workers into dynamically loadable modules used at the time applications are run, the HDL authoring model (and typical practice in HDL/FPGA development) involves a few more steps. These steps are supported by additional scripts:

- building HDL primitives used as submodules *before* building workers in a component library
- building HDL assemblies using workers *after* building workers in a component library

The first items (primitives) are built *before* building HDL workers, and the second (application assemblies) are built *after* building HDL workers (but before building final FPGA bitstreams). This sequence and associated dependencies are automatically managed by the OpenCPI build process for the top level “hdl” directory: primitives are built first, then application, adapter and device workers, then platforms, then assemblies, and containers. This entire process is the same across vendors, and across synthesis for hardware vs. compiling for simulation.

4.5 Building HDL Primitives

HDL primitives are organized into libraries of smaller units of modularity than workers (workers are built in OpenCPI component libraries). HDL primitives are essentially either a precompiled group of source files, or a prebuilt/presynthesized core, either generated by other tools (i.e. Xilinx CoreGen), or a core for which source code is unavailable (a 3rd party core in edif or ngc format or possibly qxp format for Altera). In both cases the exported/installed library or core that results from building a primitive is something that can be referenced by worker designs simply by including the following lines in the HDL worker makefile:

```
Libraries=myutils
```

If the library name is a simple name (no slashes), then it is assumed to be installed in the CDK installation directory itself. Then the OpenCPI source tree is present, this directory is actually the "ocpi" subdirectory of the source tree. If the name has slashes, it is used as a specific directory elsewhere where the primitive has been built and installed (e.g. your own library of primitives).

So, if the worker source code instantiates a module from the primitive library, no further action need be taken other than including the line above in the HDL worker's Makefile (or once for a component library in the component library's Makefile). In particular, no other "black box" module or VHDL component declaration need be created or referenced by the worker. The CDK itself includes several primitive libraries, some only have source code, and others have prebuilt cores. Several HDL primitive libraries in OpenCPI are always included even when the Libraries variable is not set.

4.5.1 Building primitive libraries from source files

To create an HDL primitive library from source code, create a directory containing the source files to be precompiled, and include a "Makefile" that includes this line:

```
include $(OCPI_CDK_DIR)/include/hdl/hdl-lib.mk
```

With the "Makefile" above, just running "make", will create the primitive library for the targets indicated in the HdlTargets "make" variable.

When you need to export the results of building the library somewhere other than where the sources are (in the target-* subdirectory), you set the "HdlInstallDir" variable to indicate where this and other libraries or cores should be "installed", and then running "make install". The default value of "HdlInstallDir" for the primitive library is "lib" in the library's directory, which is not particularly useful. In the OpenCPI core directory tree, all primitive libraries are installed under hdl/primitives/lib.

A primitive library from source code is built for a variety of targets in per-target directories such that any concrete target that a *worker* is built for will use appropriate files resulting from the build of the primitive libraries for that same target.

Primitive libraries may be written in VHDL or Verilog, but a variety of issues arise when workers written in one language access primitives written in the other language. The constraints vary widely with tools and their ability to precompile libraries etc. To make a VHDL primitive library work across most tools and be callable/instantiable from Verilog or VHDL, the library must include a VHDL file *_pkg.vhd containing component declarations for entities in the library. Even if the library is Verilog source code, the

*_pkg.vhd VHDL file with component declarations should be present if the Verilog modules in the library will be used from VHDL. The package name in the file should normally be the same as the library's name, but doesn't have to me. There can be multiple *-pkg.vhd files in each library if multiple packages are required. Finally, if the packages have package bodies in separate files, those files should be named "*-body.vhd".

When a VHDL worker is written to use a primitive in a library, it must include a line to access the library. For a primitive library "foo", and package named "bar", the worker would contain the lines:

```
library foo; use foo.bar.all;
```

4.5.2 Building primitive cores from prebuilt/presynthesized files

Making a prebuilt/presynthesized core available for use by workers is similar to creating a primitive library from source, except that the make file contains the lines:

```
include $(OCPI_CDK_DIR)/include/hdl/hdl-core.mk
```

The difference is that, in addition to source files that are just precompiled for use by workers, the core is also synthesized for specific targets and made available for building workers as a prebuilt/presynthesized core. The files used to build the core can be a mix of source and prebuilt files. Such a primitive may have no source code at all (except the black-box module definition), in this case the Makefile might look like this (in a directory called "mycore"):

```
PreBuiltCore=mycore.ngc # suppress rules to build from source files
OnlyTargets=xcv6lx240t  # this core is only good for this part
include $(OCPI_CDK_DIR)/include/hdl/hdl-core.mk
```

In this case the scripts simply expect the core file to already exist, with only the black-box file in source form: mycore_bb.v.

4.6 Building HDL assemblies into bitstreams/executables

An HDL assembly is simply the subset of a heterogeneous OpenCPI application that will execute on one FPGA, and thus is the part of some OpenCPI applications that will ultimately be put into an FPGA bitstream. This section describes how to define and build (synthesize) these modules and to ultimately turn them into bitstreams.

This process is essentially *deploying the assembly onto the platform*. More specifically, it is deploying the assembly onto a *platform configuration* that has been defined, prebuilt, and possibly preplaced/constrained in advance. We call the result of this "deployment" a "container" – the outer module that contains the application as well as the platform support modules in the platform configuration. Note that a platform configuration may be a simulation platform or a real platform.

Given that the platform configurations already exist, the steps taken to go from the assembly (described in XML) to a bitstream are:

1. Describe the assembly in XML
2. Select the platform configuration that the assembly will be deployed on
3. Specify how the assembly's external ports connect to the platform (i.e. to an interconnect like PCIE for off-platform connections, or to local devices). This is called "defining the container".
4. Run "make", to generate the bitstream.

The actual steps taken by the OpenCPI scripts and tools are:

1. Generate the Verilog/VHDL code that structurally implements the assembly
2. Build/synthesize the assembly module, that has some "external ports".
3. Generate the Verilog/VHDL container code that structurally combines the assembly and the platform configuration, as well as any necessary adapters.
4. Build/synthesize the container code, incorporating the assembly and platform configuration. This is the top level module.
5. Run the final tool steps to build the bitstream (map, place, route, etc.).

4.6.1 The HdlAssembly XML file that describes the assembly

The assembly is described in an XML file containing an HdlAssembly top-level element, which contains ***worker instances*** and ***connections*** and ***external ports*** . It is similar to the Application XML file that describes the whole OpenCPI heterogeneous application (as documented in the OpenCPI Application Guide).

The worker instances ("instance" subelements of the HdlAssembly) simply reference HDL workers in a component library, and optionally assign names to each instance. The "worker" attribute is the worker's OWD name (without directory or model), and the optional "name" attribute is the instance name. When not specified, instance names are either the same as the worker name (when there is only one instance in the assembly), or the worker name followed directly by a zero-based decimal ordinal (when there is more than one instance of the same worker).

Connections among workers in the assembly use "connection" XML elements. They define connections among worker data interfaces. A trivial example would be:

```
<HdlAssembly>
  <Instance Worker="generate"/>
  <Instance Worker="capture"/>
  <Connection>
    <port name='out' instance='generate'/>
    <port name='in' instance='capture'/>
  </Connection>
</HdlAssembly>
```

For convenience, internal connections between the output of one instance to the input of another, can simply be expressed using the "connect" attribute of the instance, indicating that the instance's only output should be connected to the only input of other instance whose name is the value of the "connect" attribute. Like this:

```
<HdlAssembly>
  <Instance Worker="generate" connect='capture'/>
  <Instance Worker="capture"/>
</HdlAssembly>
```

Furthermore, when this shortcut is used, you can in fact specify the "from" port using the "from" attribute, and the "to" port using the "to" attribute. If these instance had multiple other input and output ports, you can also specify it this way:

```
<HdlAssembly>
  <Instance Worker="generate" connect='capture' from='out' to='in'/>
  <Instance Worker="capture"/>
</HdlAssembly>
```

To specify external ports, the "external" element is used, which allows the name of the external port to be different from the worker port it is connected to:

```
<HdlAssembly>
  <Instance Worker="generate" connect='process' from='out' to='in'/>
  <Instance Worker="process"/>
  <external name='procout' instance='process' port='out'/>
</HdlAssembly>
```

But there is also a shortcut when the external port name is the same as the worker's port name, by simply using the "external" attribute of the instance:

```
<HdlAssembly>
  <Instance Worker="generate" connect='process' from='out' to='in'/>
  <Instance Worker="process" external='out'/>
</HdlAssembly>
```

However, this is only usable for one port of the worker. If the assembly is in fact a single worker where both "in" and "out" ports should be external, you would need:

```
<HdlAssembly>
  <Instance Worker="process" external='out'/>
  <external instance='process' port='in'/>
</HdlAssembly>
```

Below is a diagram of a simple assembly, and the corresponding HdlAssembly XML file. The application has a "switch" worker that accepts data either from its "in0" or "in1" interface, and sends the data to its "out" interface. The "delay" worker sends data from "in" to "out" implementing a delay-line function that requires attaching memory to it. The "split" worker takes data from its "in" interface and replicates it to both its "out0" interface as well as its "out1" interface. The local application has 4 external ports (ADC, SWIN,

SWOUT, DAC), as well as a requirement of attached memory (to satisfy the memory requirements of the “delay” worker).

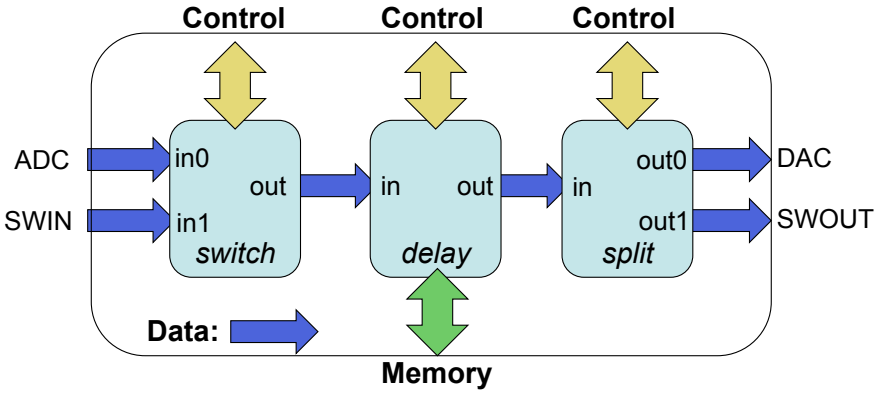

Figure 4: Example Local Application

Given that these three workers are already in a component library, the XML description of the above is below. Note that one of the workers (delay) in fact requires memory and thus would have a WMemI interface declared in its OWD, but this is not mentioned in the HdlAssembly below since no additional information needs to be provided: the container will provide memory to the worker.

```
<HdlAssembly>
  <Instance Worker="switch" connect=”delay”/>
  <Instance Worker="delay" connect=”split”/>
  <Instance Worker="split"/>
  <External name=’adc’ instance="switch" port="in0"/>
  <External name=’swin’ instance="switch" port="in1"/>
  <External name=’dac’ instance="split" port="out0"/>
  <External name=’swout’ instance="split" port="out1"/>
</HdlAssembly>
```

Figure 5: HdlAssembly for the example local application

4.6.2 The Makefile for building an assembly

Assemblies are built in their own directory, where the name of the assembly defaults from the name of the directory, and the contents of the assembly is described in the HdlAssembly XML file described above. The name of the XML file is simply the name of the assembly, with the “.xml” file extension. In addition to this description file, the Makefile must indicate which component libraries should be used to find the workers mentioned in the HdlAssembly file. A Makefile for the above application might be:

```
ComponentLibraries=/home/fred/project/components
include $(OCPI_DIR)/include/hdl/hdl-assembly.mk
```

In this case the assembly will be based on the indicated component library, and the automatically generated assembly Verilog module will be synthesized for all specified targets and platforms. If the OnlyTargets=virtex5 was specified, the assembly would only be synthesized for use on virtex5 parts. Alternatively, the “HdlPlatforms” variable could be set to a particular platform (e.g. ml605), in which case the assembly would only be synthesize for the family associated with that platform (virtex6).

The component libraries indicated in the ComponentLibraries variable provide a pathname to the library. If there are no slashes in the name of a component library it is assumed to be in the OpenCPI CDK.

The assemblies in the OpenCPI tree are built in the hdl/assemblies subdirectory of the OpenCPI source tree, but it could be anywhere as long as the OCPI_CDK_DIR variable is set.

When you have a directory full of assemblies like this (a “library of assemblies”), the top level Makefile can simply be:

```
Assemblies=assy1 assy2
include $(OCPI_DIR)/include/hdl/hdl-assemblies.mk
```

This would build the listed assemblies. The HdlPlatforms variable could be set here also.

4.6.3 Specifying the container that deploys the application on the platform.

The container is the top-level module and deploys the assembly onto the platform. Separating the “assembly” from the “container” in this way keeps the assembly portable and hardware-independent. It doesn’t know what it’s input and outputs are connected to, and it would easily be run in a simulation testbench with no hardware.

In an assembly’s directory, containers are specified in two ways. The first, is “default containers”. Default containers are generated by looking at the assembly, and connecting all the external ports to the platform’s interconnect (e.g. PCIE). Thus the “default” container specification is to connect every external port of the assembly such that it connects external to the platform, to connect to workers running on containers on other platforms. [Note that external connections like this are currently not supported in simulation platforms].

The “DefaultContainers” variable is used to list platform configurations for which default containers (and thus bitstreams) should be automatically generated for this assembly. The format of the items in this list is <platform> or <platform>/<configuration>. If this variable is defined as empty:

```
DefaultContainers=
```

Then there are no DefaultContainers for this assembly at all. If the DefaultContainers variable is not set at all (not mentioned in the assembly’s Makefile), then default containers (and bitstreams) will be generated for whatever platform the assembly is built for. In this case the platform configuration is assumed to be the “base” platform configuration for the platform (the one with no device workers at all).

So, if nothing is said at all about containers in the Makefile, default container bitstreams will be built for whatever platforms are mentioned in “HdlPlatforms”. In many cases this completely default simply case is all that is required (no container variables at all).

Default containers cannot be used to connect the external ports of the assembly to something *other than* the single system interconnect of the platform. I.e. if there are multiple interconnects (say PCIE and Ethernet), or if connections to local devices are required, then a container must be specified in an XML file. The “Containers” variable specifies a list of containers (container xml files) that should be built in addition to those indicated by the “DefaultContainers” variable (or absence thereof). The “Containers”

variable does not suppress the building of the default containers. If you want that, you should specify an empty "DefaultContainers" variable.

An example Makefile that relies only on the CDK component library and builds default containers for whatever platform is specified is:

```
include $(OCPI_DIR)/include/hdl/hdl-assembly.mk
```

A Makefile that builds a default container on the ml605 base platform and the "lime_adc" configuration of the "alst4" platform, and further generates a specific container called "in_2_adc" for connecting some external port to the ADC device on that latter platform configuration might look like:

```
DefaultContainers=ml605 alst4/lime_adc
Containers=in_2_adc
include $(OCPI_DIR)/include/hdl/hdl-assembly.mk
```

4.6.4 Specifying a container in an XML file

A container XML file is required to connect to multiple interconnects (or not the first one) or to local devices.

The top-level element of the container XML file is the <HdlContainer> element. Its primary attribute is the "platform" attribute to specify the platform configuration that should be targeted by the container. It is of the form <platform> or <platform>/<configuration>. Beyond specifying the platform configuration in the "platform" attribute, the subelements of the < HdlContainer> top-level element are special connection elements for containers.

They are "<connection>" elements with these attributes:

- external: specify an external port on the assembly
- device: specify a device on the platform
- interconnect: specify an interconnect on the platform
- port: when a device has more than one data port, specify it

Using these attributes you can specify these connections:

- external to interconnect
- external to to the only port of a device
- external to a specific port of a device
- a device port to an interconnect (bypassing the assembly altogether) [not yet supported]

So, in a case where the "in" of the assembly connects to a locally attached "adc" device, but the "out" of the assembly is attached to the interconnect for communicating to other FPGA or software containers, you would have:

```
<HdlContainer platform='alst/lime_adc'>
   <connection external='in' device='adc'/>
   <connection external='out' interconnect='pcie'/>
</HdlContainer>
```

4.6.5 Service connections in a container

OpenCPI HDL workers have three types of ports: control, data, and service. Service ports are connected locally in the container to provide the required services to workers. Service ports are implementation-specific and not found in "spec" files. The services currently defined are memory and time (time of day).

If workers in the assembly have service ports, they automatically become external ports of the assembly. When generating a container, all services required by the assembly, as well as any services required by device workers instanced in the assembly, must be satisfied by instancing the appropriate modules in the generated container code.

Time service requirements are satisfied by instancing a "time client" module for each worker that needs "time", and also connecting that "time client" to the timekeeping infrastructure on the platform. Each time client is instanced based on requirements of the associated worker's time port. These requirements are precision and whether the worker can tolerate time-of-day becoming available after it starts running, or whether it simply assumes it is always available.

Memory service requirements [which are currently not automated as of this writing], are satisfied by instancing either private BRAM modules (private to the worker), or instancing external memory access interfaces, connected to device workers for external memory. Memory access may also be multiplexed to support multiple workers sharing the same memory.

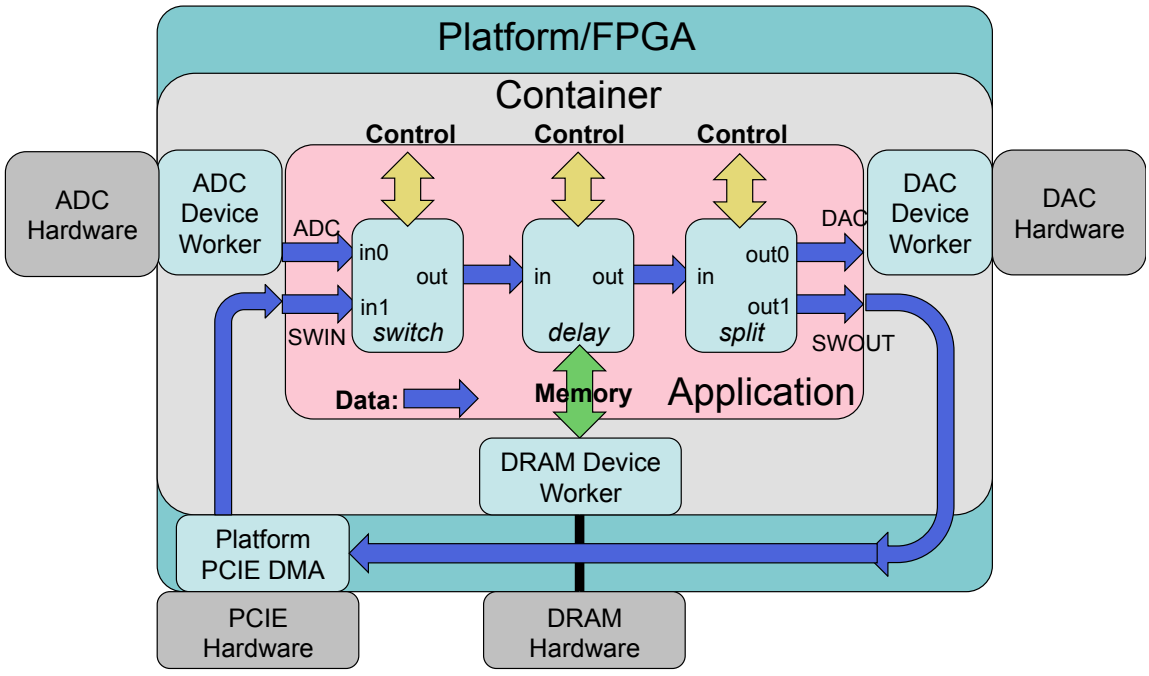

Figure 6: OpenCPI HDL Assembly on Container and Platform

4.6.6 Preparing the OpenCPI runtime metadata for the bitstream

When OpenCPI runs an application that includes HDL workers and thus includes an HDL assembly bitstream, it requires a small runtime metadata file that tells it what the bitstream contains (which workers and how they are connected, etc.). This file is

automatically generated as part of the same process that builds the bitstream, and is attached to the bitstream file after it is created by the FPGA back-end place-and-route tools. This file is also compressed and placed in a BRAM on chip. Thus software can discover the contents of a bitstream either by looking at the file to be loaded, or, looking at the file inside the loaded bitstream at runtime. This allows the control software to know what a bitstream can offer before it is loaded, and also know what the currently loaded bitstream can offer, without having access to the file that loaded it.

This runtime XML is generated whenever a bitstream is generated, and placed in a BRAM (compressed) and attached to the file (uncompressed).

The bitstreams are in fact compressed, thus the resulting ".xxx.gz" file has the metadata attached to the file (after the file is compressed, but with the XML not compressed). This compressed bitstream file, with uncompressed metadata attached, is what is provided as the "binary artifact" for HDL assemblies, as installed in an OpenCPI runtime component library as referenced by the OCPI_LIBRARY_PATH environment variable. It is the format expected by the internal mechanisms to load bitstreams onto platforms at runtime.

4.7 HDL device naming

HDL devices that can support OpenCPI containers have unique names within an OpenCPI system context. Each name starts with a prefix indicating how the device is discovered and controlled by OpenCPI software. Note that "device" in this context is an instance of an HDL platform in an OpenCPI system, rather than a device inside of an HDL platform attached to the FPGA chip.

The mechanisms currently supported are:

PCI
: FPGA devices/boards accessible by PCI express

Ether
: FPGA devices accessible via link-layer Ethernet

Sim
: FPGA devices that accessible via UDP that are in fact simulation servers (see "simulation devices" below)

UDP
: FPGA devices that are accessible via IP/UDP

The full device name is of the form:

```
<control-scheme>:<address-for-control-scheme>
```

As a convenience, if there is no known prefix in the name, then if there are 5 colons in the device name, it is assumed to be an **Ether** device name, otherwise it is assumed to be a **PCI** device name.

4.7.1 PCI-based HDL devices

HDL devices on the PCI express bus/fabric are identified by the syntax common to many PCI utilities such as "lspci" on Linux, namely:

```
<domain>:<bus>:<slot>.<function>
```

An example is:

```
0000:05:00:0
```

Since it is common to have "domain", "slot", and "function" all being zero, if the address field in the device is simply a number with no colons, it is assumed to be the bus number with the other fields being zero. Thus "pci:5" implies "pci:0000:05:00.0". Since an identifier with no prefix and not having 5 colons is assumed to be a PCI device, the identifier "4", is assumed to be "pci:0000:04:00.0".

A common example of a PCI device is the Xilinx ML605 development board. Another is the Altera Stratix4 development board (called alst4 in OpenCPI).

4.7.2 Ethernet-based HDL devices

HDL devices that are attached to Ethernet and operate at the link (or MAC) layer (OSI layer 2) use the "ether" prefix. This prefix implies access without using any routing or transport protocols such as IP/UDP or IP/UDP. It is the fastest and lowest latency way to use a network, with the drawback that it cannot be "routed" through IP routers, but

can only be “switched” by L2 Ethernet bridged and switches. The syntax for naming such devices is:

```
Ether:[<interface>/]<mac-address>
```

The MAC-address is the typical 6 hexadecimal bytes separated by colons, such as:

```
c8:2a:14:28:61:86
```

The optional <interface> value is the name of a network interface on the computer accessing the device. Typical examples are “en0” or “eth0”. It is optional when there is only one such device. On systems with multiple interfaces it indicates which one should be used to reach the device. This is needed since there is no routing at this level of the network stack: you much use the right network interface to reach the addressed device.

The available network interfaces can usually be identified by the “ifconfig” linux command. There is a more special purpose command of the ***ocpihdl*** utility described below that lists only the network interfaces available and usable for OpenCPI (the “ethers” command to ***ocpihdl***).

4.7.3 Simulator device naming

OpenCPI runs HDL simulators in a “server” that makes them look like any other device to software. When they are available (the server is running), they are discoverable like any other device. They are accessed using the IP/UDP network protocol, and thus use IP addresses. They are named according to this syntax:

```
Sim:<IP-address>:<IP-port>
```

The IP-Address is the typical “dotted” notation (e.g. “10.0.1.100”), and the IP port is a decimal number less than 65536.

4.8 The ocpihdl command-line utility for HDL development

The ***ocpihdl*** utility program performs a variety of useful functions for OpenCPI HDL development. These include:

- Searching for available FPGA devices (via PCI, Ethernet, UDP, etc.)
- Testing the existence of a specific FPGA device
- Reading and writing specific registers in an FPGA device
- Running a simulator server that executes simulators and acting as FPGA devices
- Loading bitstreams on a device
- Extracting the XML metadata from a running device

The general syntax of ocpihdl is:

```
ocpihdl [<options>] <command> [<options>] [<command arguments>]
```

Options are the typical hyphen-letter options, some with arguments after the hypen-letter argument. The following sections describe each command and its associated options and arguments. Here are the options that apply to many commands:

-v be verbose – put progress messages on standard output

-d <device>
specify the HDL device for the command. See "HDL device naming" above.

-p <platform>
specify the HDL platform type (e.g. ml605) for the command

-l <log-level>
specify the OpenCPI logging level that should apply during execution

-P produce "parseable" output for some commands that read registers

-i <network-interface>
the network interface (e.g. "en0") to use for the comment

The commands provided by the ***ocpihdl*** utility are summarized in the following table.

Name	Device	Worker	Interface	Description
admin	✔			Dump HDL device admin information
bram				Create BRAM file from input (XML) file
deltatime	✔			Measure round-trip time for time synchronization
dump				
emulate			✔	Emulate a UDP or Ethernet device (admin space)
ethers			✔	List available, up, and connected Ethernet interfaces

probe	✔			See if a specific device exists and responds
load	✔			Load a bitstream onto the device
getxml	✔			Extract XML metadata from running device
radmin	✔			Read specific addresses in HDL device admin space
reset	✔			Reset device (via control plane access)
rmeta	✔			Read metadata space from device
search			✔	List all discovered and responding HDL devices
settime	✔			Set the device's GPS time to the system time
simulate				Run a simulator server
unbram				Create an XML file from a BRAM file
wadmin	✔			Write admin space
wclear	✔	✔		Clear worker status errors
wdump	✔	✔		Dump/print worker control/status registers (not properties)
wop	✔	✔		Perform control operation on worker (e.g. start)
wread	✔	✔		Read worker configuration property space
wreset	✔	✔		Assert reset for worker
wunreset	✔	✔		De-assert reset for worker
wwctl	✔	✔		Write worker control registers
wwpage	✔	✔		Write worker page register (to reach full 32 bit space)
wwrite	✔	✔		Write worker configuration property space

Table 1: ocpihdl commands

*4.8.1 **admin** command – Print the device's administrative information*

This command is used to dump all the information and state in the devices "admin" space, which is the information for the device and bitstream as a whole. This information is specified in the OpenCPI HDL Infrastructure Specification document. A device flag must be specified. There are no command arguments.

*4.8.2 **bram** command – Create a configuration BRAM file from an XML file*

This command converts an XML file that is expected to contain the "artifact description XML" into a format that will be processed into a read-only BRAM inside the FPGA bitstream. This allows software to know what is inside a bitstream by only looking at the device, without needing a separate bitstream file. This command is used by the scripts that create bitstreams. The output format is an ASCII format that the tools can use to initialize memories. The two command arguments are <input-file> and <output-file>, e.g.:

```
ocpihdl bram my.xml mybram.bin
```

*4.8.3 **deltatime** command – Perform time synchronization test on device.*

This command, which requires a device option, uses special time-difference hardware in the OpenCPI FPGA bitstream to measure the round-trip time for accessing the FPGA in order to reduce the time-skew between the system time-of-day and the FPGAs time-of-day. It takes 100 samples, averages them, eliminates 10% outliers, and then retests the time-skew after applying the correction.

*4.8.4 **dump** command – unimplemented*

This command, which requires a device option, uses the configuration BRAM to dump the state of all workers and their properties.

*4.8.5 **emulate** command – emulate a network-based device (admin space only)*

This command acts as an Ethernet or UDP-based HDL device and responds to discovery and admin-space accesses. It is used to test software controls and network connectivity. It can optionally be supplied with a network interface option (**-i**) to specify on which network interface should the emulated device appear. With no network interface specified, it uses the first available that it finds (that is "up" and "connected"). A special case is the network interface whose name is "udp", which means the command emulates an OpenCPI HDL device attached to the IP subnet of the host computer implementing discovery and control via UDP, e.g.:

```
ocpihdl -i udp emulate
```

*4.8.6 **ethers** command – display available (up and connected) network interfaces*

This command is used to list all the available Ethernet network interfaces on the system and whether they are "up" and "connected". It also shows what the "default" interface is (the first listed that is up and connected), for commands that could take an interface (**-i**) option.

*4.8.7 **probe** command – test existence and availability of device*

The probe command takes a device (as option or argument) and tries to contact it and see if it is responding. This should work whether it is running an application or not.

*4.8.8 **load** command – load bitstream*

The load command takes a device (as option or argument) and a bitstream file name argument and loads the bitstream onto the device.

*4.8.9 **getxml** command – retrieve XML metadata from device*

The getxml command takes a device (as option or argument) and a filename to write the XML data to. It retrieves and uncompresses the XML data stored in the running device and writes it to a file.

*4.8.10 **radmin** command – read a specific address in device's admin space*

This command reads an individual word of data from the admin space of the specified device. The command argument is the hexidecimal (starting with 0x) or decimal offset in the admin space. If the offset ends in "/n" where n is 4 or 8, then that specifies the size of the access in bytes. If there is no "/n", then the access is a 32 bit access.

If the parseable option (-P) is specified the output is just the value returned in hexadecimal format. Otherwise a prettier message with the offset and value is printed. An example that does a 2 byte read of offset 12 would be:

```
ocpihdl –d pci:5 radmin 12/2
```

The layout of the admin space is defined in the OpenCPI HDL Infrastructure document.

*4.8.11 **reset** command – perform soft reset on device*

This reset command resets the device in a way that does not affect the control path to the device. For example on a PCI-based device, it would not damage or reconfigure the PCI express interface. All application workers and most other (infrastructure) workers are placed in reset and need to be specifically taken out of reset. [unimplemented].

*4.8.12 **rmeta** command – read specific addresses in the metadata space of the device*

This command works exactly like the **radmin** command except it reads the configuration BRAM space rather than the admin space.

*4.8.13 **search** command – search for all available HDL devices*

This command searches for all reachable HDL devices and reports what it finds. It uses all the supported control paths (PCI, Ether, UDP/Sim). If an interface **(-i**) option is specified it limits the Ether search to that one interface.

*4.8.14 **settime** command – set device's time from system time*

This command sets the time on the device from the current system time. It requires a device to be specified. The current time of the device is shown in the output of the **admin** command.

*4.8.15 **simulate** command – run a simulation server for a specific simulator "platform"*

This command creates a simulated HDL device that is discoverable (by the "ocpihdl search" command, by the "ocpirun" utility, and via the Application Control Interface (ACI) C++ API). It is thus an available HDL device for running assemblies of workers for such a device, compiled into "bitstreams". When "ocpihdl simulate" is running, the device is ready to "load and run a bitstream", where the "bitstream built for the device" is in fact a "simulation executable" for running under a particular FPGA simulator (such as Modelsim or Isim).

This command initially emulates the device (similar to the “emulate” command, except as a “sim” type device) and is thus discoverable before loading any particular simulator executable. The platform option (-p) is required to specify which type of simulator should the simulated device use as its “platform”. The current choices are isim (Xilinx ISE), xsim (Xilinx Vivado), and modelsim (Mentor). [xsim currently non-functional due to limitations reported to Xilinx].

When some OpenCPI application (e.g. executed using the ***ocpirun*** utility command) decides to run an assembly of workers on this simulator-based device (as it would with any other discovered and available HDL device/FPGA), it would request that the bitstream (executable) be “loaded” and “started” on this device. This would cause this simulated HDL device to run the actual simulator (e.g. modelsim) with that executable. If the application was re-run, it would notice that the “bitstream” was already loaded and continue to use it, which would in fact re-use the device in the same simulation run.

If the simulated device is currently running when requested to “load a new bitstream”, it shuts down the simulator, preserving its results, and starts a new one with the newly requested bitstream, in a newly created subdirectory.

Each time a simulator is actually run under this “server”, it will execute in a new subdirectory created for that simulation run, with the name:

```
<assembly-name>.<sim-platform>.<date-time>
```

Thus running the “ocpihdl simulate” command in a directory will result in one or more subdirectories holding simulation results for each simulation run. If this simulated device is never asked to “load and run” a bitstream, no subdirectories will be created, even though the device will still be discoverable while this command is running.

The simulation run starts with the first successful “load and run” of a bitstream. Each simulation will continue until one of the following occurs:

- The code being simulated explicitly asks for the simulation to terminate via the $finish system task in Verilog or an assertion in VHDL.[check VHDL details].
- The simulation run exceeds the control-plane clock cycle count provided by the ‘-T’ option to “ocpihdl simulate”
- This command receives a control-C.
- The simulated device is asked to load and run a new bitstream.

The results of any simulation run can be viewed by opening the file that records the execution with the viewer associated with the simulator being used. For Xilinx “isim”, the command (in the subdirectory for the simulation run) would be:

```
isimgui –view sim.wdb
```

For modesim, it would be:

```
vsim [fix this]
```

A separate command, “ocpiview” is used to conveniently view the results of simulation. If run in the same directory as “ocpihdl simulate”, it finds the most recently created simulation subdirectory, and runs the appropriate views based on which simulator was used. It can be given an optional argument specifying the directory name of the

simulation to view. This a typical sequence is to run “ocpihdl simulate”, and after that completes (based on control-C or other action), run “ocpiview” to look at the results.

*4.8.16 **unbram** command – create an XML file from a config BRAM file*

This command reverses the function of the ***bram*** command, by converting a file formatted for initializing (during build) the configuration bram in a bistream back to the original XML file. The two command arguments are <input-file> and <output-file>, e.g.:

```
ocpihdl unbram mybram.bin my.xml
```

4.8.17 wadmin command – write specific addresses in the device’s admin space

This debug command writes a 4-byte/32-bit or 8-byte/64-bit value in the device’s admin space at the specified offset. The size in bytes (4 or 8) is optionally specified in the first argument with a slash. The default size is 4. The syntax is:

```
ocpihdl wadmin –d <hdl-device> <offset>[/<size-in-bytes> <value>
```

The value can be in hex (preceded by 0x) or decimal. An example, to write an 8 byte value at offset 0x20 with the value 12345, would be

```
ocpihdl wadmin –d PCI:5 0x20/8 12345
```

4.8.18 Worker Commands: commands that operating on individual workers

The following worker commands operate on either a single worker or a sequential range of workers. They all require a device to be specified, and the first argument is either a single worker number or a set of worker numbers separated by commas. Thus to perform a worker command on worker 5 would be:

```
ocpihdl w<cmd> -d pci:5 5 <worker command args if any>
```

To perform the command on workers 2, 5, and 6, would be:

```
ocpihdl w<cmd> -d pci:5 2,5,6 <worker command args if any>
```

The worker commands are:

4.8.18.1 wclear command – clear a worker’s error registers

Each worker has a set of control and status registers that are part of the control-plane infrastructure IP (not in the worker itself). This command clears the error and attention bits in the worker’s status register. A worker’s status can be displayed by the “wdump” command. An example for clearing error and attention status for worker 5 is:

```
ocpihdl wclear –d PCI:5 5
```

4.8.18.2 wdump command – dump a worker’s status registers (not properties)

Each worker has a set of control and status registers that are part of the control-plane infrastructure IP (not in the worker itself). This command displays the current status of the worker by dumping these registers.

4.8.18.3 wop command – perform a control operation on a worker

This worker command executes a control operation for the worker, directly accessing the hardware that makes the control operation request of the worker. Only the “start” operation is implemented on all HDL workers. Any others may have unpredictable or

erroneous results when requested on a worker that doesn't implement them. The available operations are:

- **initialize** – after reset is deasserted (see wunreset), request worker initialize itself
- **start** – put the worker into an operational state, after stop or initialize
- **stop** – suspend operation of the worker
- **release** – return the worker to the "pre-initialized" state
- **test** – run the worker's built-in test
- **before** – inform worker that a batch of property settings will happen
- **after** – inform a worker that a batch of property reads has completed

4.8.18.4 wread command – read a worker's configuration property space

This command reads a value from the worker's property space. The first argument is the offset in the worker's property space (in bytes), with an optional size-in-bytes for the access (1, 2, 4 or 8), and the second (optional) argument is how many sequential accessed to make. The default for the size is 4, and the default for the second argument is: 1. E.g., to read 3 single bytes at offset 6 from worker 11 would be:

```
ocpihdl read –d 5 11 6/1 3
```

See the "wwpage" command for workers whose property space is larger than 1 MByte (2^20).

4.8.18.5 wreset command – assert reset for a worker

This command asserts the control reset signal into the worker. It stays asserted until the wunreset command is used.

4.8.18.6 wunreset command – deassert reset for a worker

This command deasserts the control reset signal into the worker, after which the "wop initialize" command can be issued (or, it the worker does not implement the "initialize" command, the "wop start" command can be issued).

4.8.18.7 wwctl command – write a worker's control register

This command writes the worker's control register, which is in the control plane infrastructure IP, not in the worker itself. The argument is a 32 bit value. The bit definitions are described in the "OpenCPI HDL Infrastructure" document.

4.8.18.8 wwpage command – write a worker's window/page register

A worker's property space can be a full 32 bit space (4 GBytes), but to access more than the first 1MByte (2^20 bytes), a "window" register in the control-plane infrastructure must be set. This command sets that register with the value in the first argument, which sets the high order 12 address bits (31:20) of the effective address when the **wread** and **wwrite** commands are issued. The offset in those commands supplies bits (19:0) of the effective address. Thus to read location 0x12345678 in worker 7 with a full 4GByte property space, two commands would be used:

```
ocpihdl wwpage –d pci:5 7 0x123
ocpihdl wread –d pci:5 7 0x45678
```

4.8.18.9 wwrite command – write a worker's configuration property space

This command writes a single value into a worker's property space at an offset (and size) specified in the first argument and a value specified in the second argument. The wwpage command applies to this command also, for workers with large property spaces. Thus to write location 0x20 in worker 6 with the 64-bit value 0x123456789abc:

```
ocpihdl wwrite –d pci:5 6 0x20 0x123456789abc
```

5 Utility programs used internally

The following sections describe utility programs used internal to the various building scripts in OpenCPI. They are generally not used directly.

This section is not complete or up to date as of this document version.

5.1 ocpigen: the OpenCPI tool for code and metadata generation

The ocpigen tool is a command line tool for implementing certain parts of the OpenCPI component development process. Normally a component developer never uses **ocpigen** directly since it is used as needed by the make scripts used when libraries and components are created as described above. Below is a diagram depicting how ocpigen takes the XML description files (defined in detail in the [AMR]) and generates various files to support the creating of workers (component implementations):

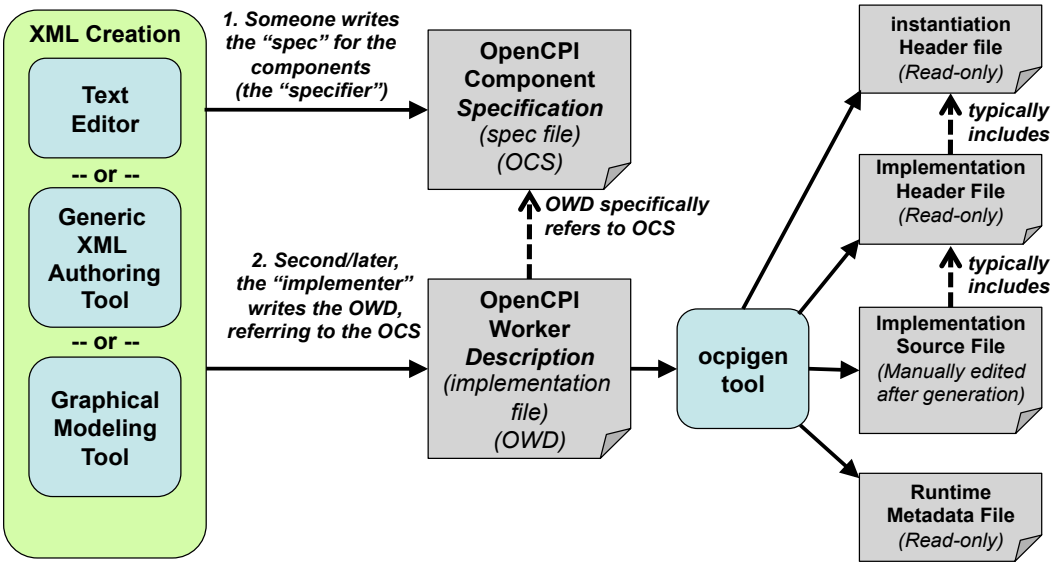

Figure 6: ocpigen Role in the Worker Development Process

5.1.1 Outputs from ocpigen

During the component implementation process, *for all authoring models*, ocpigen takes as input the XML description files, and generates these outputs:

Instantiation Header File: this read-only file is what is required for instantiating the worker in a container (or sometimes in test harnesses). It is required for some authoring models (e.g. HDL Verilog empty module or VHDL component) and not for others (e.g. RCC). It is sometimes also used in compiling the implementation (e.g. HDL Verilog, but not VHDL). It contains the definitions that containers require to instantiate the workers, and no more.

Implementation Header File: this read-only file is the basis for compiling the worker implementation source code, and has definitions beyond what is available from the Instantiation Header File, including various convenience definitions generated from the XML description files. It is required for compiling the worker, is not expected to be modified by the implementer, and contains as much many conveniently generated definitions as possible to minimize redundant and error prone efforts in the source code. Whenever possible, definitions, declarations, macros, etc. are defined here to minimize the lines of code in the Implementation Skeleton file (below), since that is edited by hand and any definitions are subject to hand editing mistakes and merge issues.

Implementation Skeleton File: this generated file is expected to be edited and contains a "skeleton" of the implementation, based on the XML description files. It is compilable as ist, and is generally "functional" in most authoring models, meaning that it can be compiled and included in real applications, but will not do anything interesting or useful. This is generally helpful to test the integration into applications before the implementation skeleton is "fleshed out" with real functional code (a.k.a. "business logic").

The **ocpigen** tool, for each authoring model, tries to put as much code into the implementation header file as possible since it is read-only and thus can be easily overwritten/regenerated by the tool. This minimizes the problems and challenges associated with changes in the XML description files (like adding a configuration property to an OCS), since the regenerated code will usually not invalidate the skeleton file (which has been edited by hand). In such cases any changes in the generated skeleton code (which are rare), can easily be compared against previously skeletons.

5.1.2 How ocpigen is used

The makefile scripts included in the CDK normally run **ocpigen** in the appropriate way with the appropriate options, but to run ocpigen as a command line tool separate from the CDK makefile scripts, the syntax is:

```
ocpigen [options] owd.xml
```

The owd.xml file is the worker description XML file, which contains or refers to an OCS (OpenCPI Component Specification) file. The options specify what output to produce as well as other options. All options are single letter flags following a hyphen. When options require a filename or other string, they follow as separate arguments, not directly following the flag "letter".

The options that tell ocpigen what output to produce are:

- **`-d`** Produce the *instantiation header file* (the "defs" file), which will have the same name as the OWD with the suffix "_defs" followed by an appropriate file name extension for included files of the authoring model (e.g. ".h" for C code, ".vh" for verilog code etc.)
- **`-i`** Produce the *implementation header file* (the "impl" file), which will have the same name as the OWD with the suffix "_impl" followed by an appropriate file name extension for included files of the authoring model.
- **`-s`** Produce the *implementation skeleton file* (the "skel" file), which will have the same name as the OWD with the suffix "_skel" followed by an appropriate file name extension for the source code of the authoring model.
- **`-A`** Produce the runtime metadata ("artifact XML" file), which will have the same name as the OWD with the suffix "_art.xml". This file is not used by itselfdirectly but is attached to the runtime binary file that implements some workers.

Any number of these options can be used together and all the requested files will be generated. The other options which control ocpigen are:

-D *outDir*
Specify the directory where the generated files are placed, rather than in the current working directory where ocpigen is being run.

-I *includeDir*
Add a directory to search when finding XML files that are referenced from any XML input files (or files they reference). This option is commonly used to specify the location of "spec" files when processing OWD files, since they typically do not (and should not) be absolute or relative pathnames. This is similar to the same gcc option.

-M *dependencyFile*
Specify a file to be produced for processing by "make" in which to put any dependencies discovered (via xi:include in XML files) or generated (when output files depend on input files). Such files are then referenced from Makefiles (if the files exist – which they will not before the first run of make in that directory).

Several options only apply when artifact XML files are generated:

-P *platform*
Specify the platform name that the artifact is built for. This is used at runtime to make sure the binary file is used on the appropriate platform.

-e *device*
Specify the device (or "part") name that the artifact is built for. This is used at runtime to make sure the binary file is used on the appropriate device.

-D *outDir*
Specify the directory where the generated files are placed, rather than in the current working directory where ocpigen is being run.

Finally, several options are special operations for the HDL authoring model. These are:

-l *libraryName*
Specify a library for VHDL or Verilog "defs" files that indicates where the entity declaration should be placed.

-c *containerDescriptionFile*
Specify the name of an XML file that describes the (fixed) container interfaces when generating HDL artifact files. I.e. this option is required when the "-A" options is used for HDL workers. The contents of this file are defined in the [HDLAMR]. This is required when the –A option is used to generate the HDL artifact XML file.

-b Generate a BSV (Bluespec System Verilog) declaration corresponding to an underlying Verilog worker. This is used when integrating HDL workers written in Verilog into a container written in BSV. The output filename will have a "I_" prefix and a ".bsv" file extension. [experimental].

-w Generate a cross-language "wrapper" file for the "defs" file. If the worker implementation language is Verilog, the wrapper file is VHDL and vice versa.[unimplemented]

-a Generate the “assembly” file, given an input XML file that specifies an HDLAssembly rather than an HDLImplementation. Thus the input is not really a worker description (OWD), but rather a composition of workers that represent a static configuration inside an FPGA design. The output is a Verilog file that instantiates the workers in the assembly, interconnects them (with required adaptations/tie-offset etc.), and creates the higher level “application” module that will be used to build a complete FPGA design bitstream.

5.2 ocpisca: the OpenCPI tool for integration with SCA XML files

The ocpisca tool is a simple command line tool for integrating the OpenCPI component development process with an SCA development process that involves creating SCA metadata, known as Software Package Descriptor (SPD) files, either from modeling tools or by hand. This tool then uses those SCA SPD files (and others that the SPD refers to), as input to create the OpenCPI XML metadata files that are common across OpenCPI authoring models. This tool is only used when using SCA metadata (from tools or hand-created), rather than the native OpenCPI metadata.

In addition to this SCA-to-OpenCPI metadata transformation, the tool can also update (or "back annotate") the SCA SPD files with implementation-specific information (which is necessary because in the SCA, there is no separate file for each implementation, but only an XML subelement in the SPD files which contains information that is generated by the build and test process.

Below is a diagram depicting how ocpisca takes the SCA XML description files and generates the OpenCPI "spec" and "implementation" files.

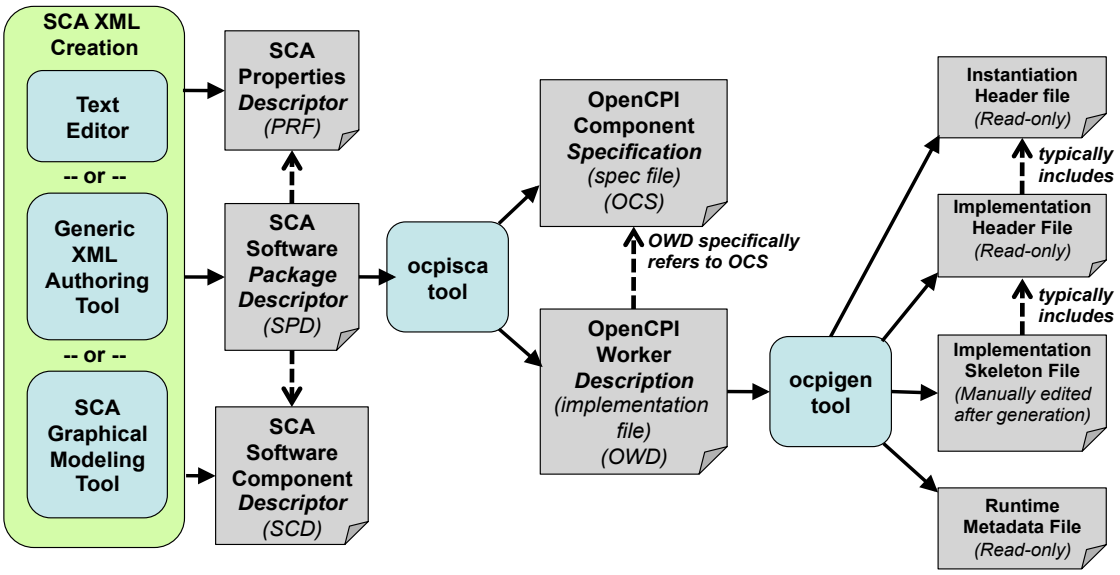

Figure 7: ocpisca Role in Worker Development Process when SCA XML is available

5.2.1 Outputs from ocpisca

The ocpisca tool creates the standard OpenCPI OCS and OWD files. In general the OCS will not have to be touched after being generated (unless the SCA inputs change). However, it is likely that the OWD will change to add more implementation information that is unavailable in SCA XML files.

The property and protocol aspects of the OCS and OWD are generated inline in those files rather than being placed in separate files.

5.2.2 How ocpisca is used

The makefile scripts included in the CDK normally run ocpisca in the appropriate way with the appropriate options, but to run ocpisca as a command line tool separate from the CDK makefile scripts, the syntax is:

```
ocpisca [options] spdfile [idl-options] idl-files
```

The *spdfile* is the SCA Software Package Descriptor (SPD) file, containing the "softpkg" element as the top-level element in the file. This file should reference an SCA Software Component Descriptor (SCD) file. The ocpisca tool uses the first implementation found in the SPD file as its target implementation (this can be overridden by an option). Thus the tool processes the SPD file, the SCD file (referenced from the SPD file), along with potentially three different SCA Property Descriptor (PRF) files:

- The PRF referenced from the SCD file
- The PRF referenced from the SPD file under a top level "propertyfile" element.
- The PRF referenced from a "propertyfile" element inside the selected "implementation" element.

The first two property files contribute configuration properties to the OCS (the "spec" file), while the third (implementation PRF) file contributes configuration properties to the OWD (the "impl" file).

The *idl-files* are IDL files that define the interfaces referenced by the "repid" attributes (interface repository IDs) for ports defined in the SCA SCD. These are necessary to generate the appropriate "protocol" or "protocolsummary" elements in the OCS. The idl-options are options used when processing IDL files normally passed to an IDL compiler, such as "-D" options to control the C preprocessor used while processing IDL files.

The ocpisca *options* listed before the SCA SPD file argument are:

--name=*name*
: Specify the implementation's name in the resulting OWD XML contents. Since the SCA SPD has no name attribute for an implementation, there is no default from the implementation element in the SPD. The default for this name is the name attribute of the SPD itself. Note that this implementation name, which appears in generated source code, should generally be unique within a authoring model, but, e.g., the implementation name for an RCC implementation and an HDL implementation can certainly be the same, and be the same as the name in the OCS (spec file XML).

--implementation=*id*
: This id value is used to identify which implementation in the SPD is the basis for the generated OWD. Since each SCA implementation id must be a DCE UUID string, these ids are cumbersome. The default is the first implementation in the SPD file. If this id is simply a decimal number, then it is the index (0 origin) of the implementation in the file. Thus the default value for this option is in fact: "0".

--specFile=*file*
: This option specifies the name of the file (without directory or extension) used for the OCS produced by the tool. The "_spec.xml" is always applied as a suffix. The default value is the name (without directory or extension) of the SPD file itself — with the "_spec.xml" suffix.

`--implFile=`*`file`*
: This option specifies the name of the file (without directory or extension) used for the OWD produced by the tool. The ".xml" suffix is always applied. The default value is the name (without directory or extension) of the SPD file itself — with the".xml" suffix.

`--specDIR=`*`file`*
: This option specifies the directory in which the spec file (OCS) is placed.

`--implDir=`*`file`*
: This option specifies the directory in which the impl file (OWD) is placed.

`--verbose`
: This option causes the tool to produce informational messages on its standard output.