OpenCPI
HDL Development Guide

OpenCPI Release: v2.4.7

Revision History

Revision	Description of Change	Date
1.01	Creation from content in previous OpenCPI Component Development document	2016-02-1
1.02	Add new content to be consistent with current system as of February 2016	2016-02-23
1.03	Processed review comments for 3 reviewers	2016-02-24
1.10	Missing details about environment, time service, dependencies	2016-04-15
1.2	2017Q1 reviews, changed simulator model	2017-02-24
1.3	Update for 2017.Q2 issues	2017-08-08
1.4	Updates for 2018.Q1: primitive library clarifications, more backpressure explanation	2018-02-26
1.5	Update for device2device container connections, other clarifications	2018-09-24
1.6	Update for 1.5, version 2 and EOF changes, and reorg stream interface	2019-04-23
1.6.1	Update for 1.6, especially multiple/split worker clocks	2019-12-10
1.7	Update for 1.7, clarifying use of primitive libraries, with packages etc.	2020-06-16
1.8	Add shared glossary of terms chapter	2021-03-15
1.9	Remove most Makefile-related content and replace with XML file, for release 2.2	2021-06-28
1.10	Add reference to updated message payload section in CDG, remove duplication of info	2022-04-15

Table of Contents

1 References

This document depends on several others. Primarily, it depends on the ***OpenCPI Component Development Guide (CDG)***, which describes concepts and definitions common to all OpenCPI authoring models.

Table 1: References to Related Documents

Title	Published By	Link
OpenCPI User Guide	OpenCPI	https://opencpi.gitlab.io/releases/v2.4.7/docs/OpenCPI_User_Guide.pdf
OpenCPI Component Development Guide	OpenCPI	https://opencpi.gitlab.io/releases/v2.4.7/docs/OpenCPI_Component_Development_Guide.pdf
OpenCPI RCC Development Guide	OpenCPI	https://opencpi.gitlab.io/releases/v2.4.7/docs/OpenCPI_RCC_Development_Guide.pdf

2 Overview

This document describes how to develop component implementations, known as ***HDL workers***, for FPGAs, using the ***Hardware Description Language*** (***HDL) Authoring Model***. Workers are typically created in a component library, so that they are available for OpenCPI application developers and users.

Some knowledge of FPGA terminology is assumed here, but this document is also useful for non-FPGA developers in understanding the OpenCPI FPGA development process.

This document builds on the information provided in the ***OpenCPI Component Development Guide (CDG)***, which introduces concepts and processes used for OpenCPI component development in general.

In addition to describing how to develop ***HDL workers***, this document also describes:

- how to create ***HDL primitive libraries***, which are libraries of smaller/simpler reusable code modules sometimes used in the design of HDL workers
- how to create ***HDL primitive cores***, which are prebuilt and possibly pre-synthesized modules sometimes incorporated into HDL workers
- how to assemble a group of connected HDL workers to form an ***HDL assembly***, which is realized in a complete FPGA bitstream.

HDL assemblies enable an FPGA to execute a subset of, or all of, the components specified in an OpenCPI application. Usually when an OpenCPI application uses an FPGA, it is using it to execute some of the components specified in the application, with the others executing on other processors typically using other authoring models. However, in some cases an HDL assembly provides workers for *all* the components required by an application.

For HDL development, OpenCPI utilizes technology-specific FPGA synthesis and simulation tools (e.g. Xilinx Vivado or ISE and Isim, Altera/Intel Quartus, Modelsim etc.).

The following sections describe the development of HDL workers, primitives (libraries and cores), assemblies and bitstreams to support the execution of parts of component-based applications on FPGAs. All these terms are prefixed with HDL here to avoid confusion when they are used elsewhere.

HDL Authoring Model: the OpenCPI authoring model targeting Hardware Description Languages that are appropriate for FPGA development, currently using VHDL, with some legacy support for Verilog. Full support for Verilog and System-Verilog is not currently included. VHDL workers calling primitives written in Verilog *is* supported. *New HDL workers should be written in VHDL*.

HDL Target: a particular type of FPGA device, usually what is considered a *part family*, that is the target of compilation or synthesis, where the result can be used for any architecturally similar device. Examples are “virtex6”, or “stratix4”, or “zynq”. Simulators are also HDL targets, including Mentor's Modelsim and Xilinx Isim and Xsim.

HDL Worker: an HDL (e.g. VHDL) implementation of a component specification, with the source code written according to HDL authoring model. While most HDL workers are *application* workers (usable in portable applications), a special type is *device* workers which are for controlling hardware physically attached to the FPGA. For application workers, it is common and recommended to have an RCC worker that also implements the same spec. Such workers are sometimes called "work-alikes".

HDL Primitive: an HDL asset that is lower level than workers, that is used as a building block for workers. HDL primitives can either be ***libraries*** or ***cores***.

HDL Primitive Library: a collection of low level modules compiled from source code that can be referenced in HDL worker code. An HDL worker declares which HDL primitive libraries it draws modules from.

HDL Primitive Core: a low level module that may be built and/or synthesized from source code, or imported as presynthesized and possibly encrypted from 3rd parties, or generated by tools like Xilinx CoreGen or Altera MegaWizard. An HDL worker declares which primitive cores it requires (and instantiates).

The following diagram shows the hierarchy of modules when an FPGA design is realized using OpenCPI:

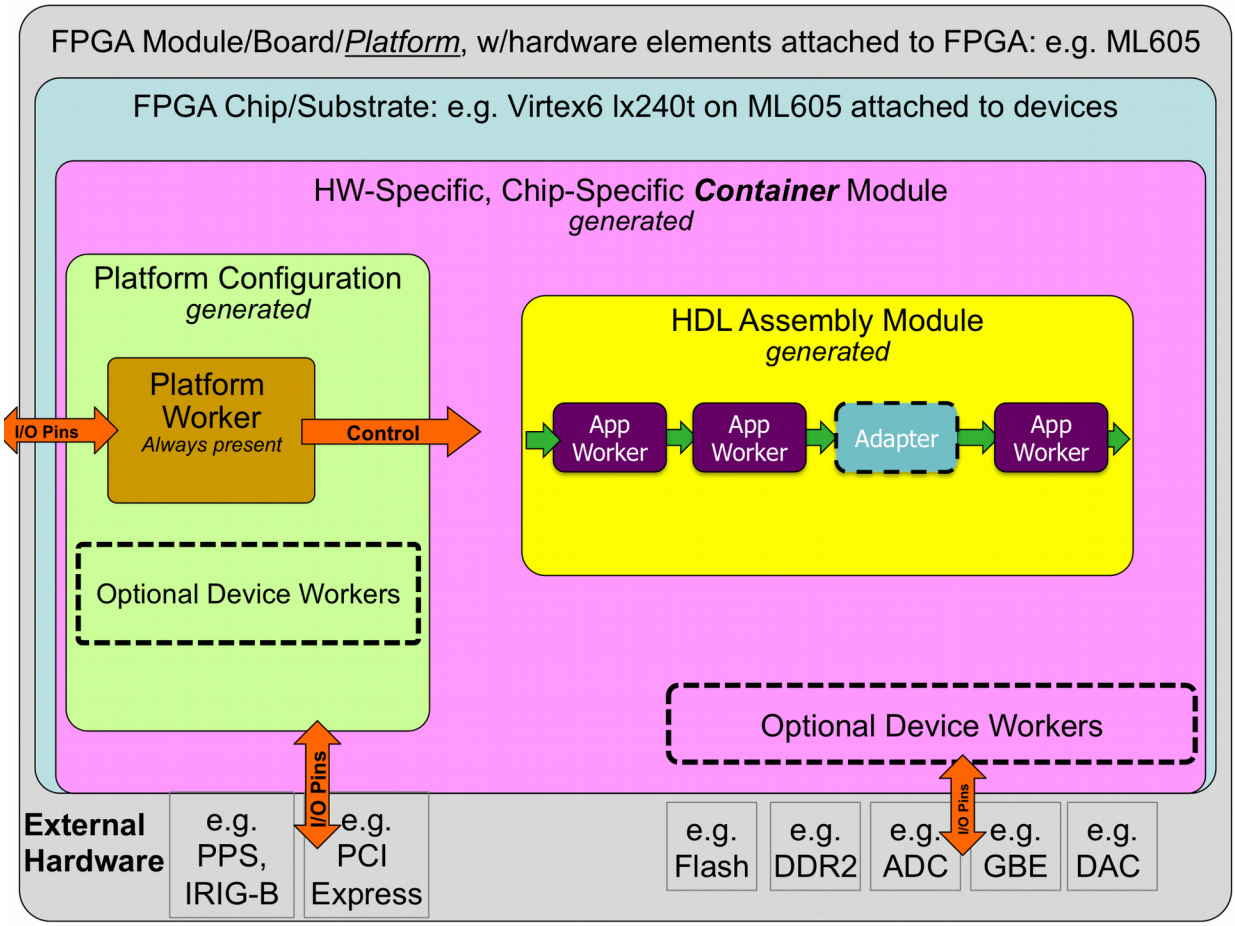

Figure 1: HDL Hierarchy

HDL Assembly: a composition of connected HDL workers that are built into a complete FPGA configuration ***bitstream***, acting as an OpenCPI artifact. The resulting bitstream is executed on an FPGA to implement some part or all of (the components of) the overall OpenCPI application. The HDL code is *automatically generated* from a brief description in XML.

HDL Platform: an OpenCPI platform based on an FPGA that is enabled to host OpenCPI HDL workers. Simulators are also considered HDL platforms.

HDL Platform Worker: a specific type of HDL worker providing infrastructure for implementing control/data interfaces to devices and interconnects external to the FPGA or simulator (e.g. PCI Express, Clocks). See the ***HDL Platform and Device Development*** section.

HDL Device Worker: a specific type of HDL worker that supports external devices attached to FPGAs. See the ***HDL Platform and Device Development*** section.

HDL Platform Configuration: a prebuilt (presynthesized) assembly of device-level HDL workers that represent a particular configuration of device support modules for a given HDL platform. The HDL code is *automatically generated* from a brief description in XML. See the ***HDL Platform and Device Development*** section.

HDL Container: a complete design for an entire FPGA, which includes an HDL assembly and an HDL platform configuration combined in a specified fashion. The HDL code is *automatically generated* from a brief description in XML.

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.

HDL device workers are developed as part of enabling an HDL platform (an FPGA on a particular board) for OpenCPI. See the ***HDL Platform and Device Development*** section.

The sections below for HDL/FPGA development are:

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

3 HDL Workers

This section describes how to write an HDL worker, and defines what distinguishes HDL worker development from developing workers using other authoring models. It builds on the worker development section in the Component Development Guide (CDG) that describes what is common to building workers for all authoring models.

HDL workers can consist of a single standalone source code module or reference and instantiate lower level models: HDL primitive libraries and HDL primitive cores. In either case a worker is compiled and (when appropriate) synthesized for a given ***HDL Target***, as described in detail in the ***HDL Build Targets*** section. HDL worker source code cannot reference other workers.

3.1 Execution Model for HDL Workers

An HDL worker executes when enabled from its control interface. When not in the operating control state a worker should only execute in response to control operations. When in the operating state, it is expected to operate continuously until a control operation changes its state (via ***stop*** or ***release*** control operation). As a convenience, the code generated worker "shell" has its own implementation of the start control operation and sets the `is_operating` signal accordingly. See Figure 2 below. It is strongly recommended that every worker use that signal to enable its operation. Execution of workers normally processes data arriving at input ports and produces data at output ports, possibly changing the values of volatile properties.

All workers are provided a reset signal, which is guaranteed to be asserted for at least 16 control clock cycles. If a worker needs more time or clock cycles to perform its initialization, then it must implement the ***initialize*** control operation as described in Lifecycle/Control Operation Signals.

3.1.1 Clocking and Clock Domains in HDL Workers

All HDL workers are provided the control clock which is used to control the operating state of the worker, access property values/registers, and may be used for data processing. However, this default single clock may restrict performance in some cases since the control clock may not be the fastest clock rate that the logic can support.

Workers may be written with different clocks used at different ports, allowing some to operate at clock rates different from the control clock. It is common for all data interfaces to use the same data clock that is different from the control clock. In some cases different data interfaces could use different clocks, all different from the control clock. Workers can also internally generate clocks that are used with one or more of its data interfaces.

It is up the worker author to perform appropriate clock domain synchronization inside the worker between control signals using the control clock and signals operating using other clocks. In many cases the only signals that might need to "cross a clock domain boundary" inside such a worker are:

- the control **`reset`** and **`is_operating`** signals, when used by logic operating in a data port clock domain.
- volatile property values, where the values are created in the data interface clock domain, and are read by the control interface using the control clock.
- writable property values, where new values are written to the worker from the control clock domain and the values are used in the data clock domain.

Note that ***initial*** properties have values that are stable before the worker enters the operating state and thus do not require clock-domain-crossing synchronization. Similarly, volatile properties that are only useful when viewed at the end of an application run may not need synchronization.

A worker author indicates which interfaces operate with which clocks in the OWD XML file, which is described below.

3.2 Creating an HDL Worker

The process of writing a new HDL worker (after the OCS exists), starts with using the **ocpidev create worker** command, as described in the CDG, to create the worker's directory and its initial content, usually as a subdirectory of a component library. This command can be executed in a project's directory or a component library's directory. The name of an HDL worker always has the **.hdl** suffix, and the language must be specified as **VHDL** or **Verilog**. Languages are case insensitive. The default is **vhdl**. Here are some examples of using **ocpidev** to create HDL workers:

```
ocpidev create worker xyz.hdl -L vhdl
ocpidev -v -l dsplib create worker
ocpidev -S fft2d-spec create worker fft.hdl
```

As described in the CDG, this will create an initial worker description XML file (OWD), ***<worker>*.xml**, in the worker directory, which is subsequently edited for worker-specific attributes. Creating an HDL worker also generates various files in the worker's **gen/** subdirectory, including the code skeleton file which is initially copied to the ***<worker>.<language-suffix>*** in the worker's directory. The first **ocpidev** command above would result in the following directory tree for the worker:

```
xyz.hdl/
        xyz.xml             # OWD for this worker
        xyz.vhd             # editable source code for this worker
        gen/xyz-skel.vhd    # initial skeleton for this worker
            xyz-defs.vhd    # definitions enabling instantiation
            xyz-impl.vhd    # the generated shell of the worker
            xyz.build       # the generated build configuration file
```

None of the files in the **gen** subdirectory should be edited. Since this directory and its contents are all generated by tools, all are deleted when the **ocpidev clean** command is issued here or at the project or library level. For HDL workers using VHDL, the initial worker code file (**xyz.vhd**), that you must subsequently edit, contains only the **architecture** of the worker. The **entity** declaration of the worker was automatically generated for you, and is found in the generated file **gen/xyz-impl.vhd**.

3.3 Building the Worker

The above command establishes the worker's directory (**xyz.hdl**), its OWD file (**xyz.xml**), and its initial source code file (**xyz.vhd**). The worker is built by issuing the

```
ocpidev build [--hdl-target=<target> | --hdl-platform=<platform> ]
```

command in the worker's directory, or

```
ocpidev build worker xyz.hdl \
    [--hdl-target=<target> | --hdl-platform=<platform> ]*
```

in the library directory containing the worker's directory. In both the above cases ***<target>*** would specify which HDL part families to build for, or ***<platform>*** would specify the platform that implies the target to build for.. If a default for ***<target>*** was already specified in the project's **Project.xml** file, nothing would be necessary here. Multiple targets or platforms may be specified to build for multiple targets with one command.

These commands will compile (or synthesize) the worker for the active set of ***HDL Targets***, such as **zynq**, **virtex6**, **stratix4**, **xsim** or **modelsim**. HDL targets are described in detail in the ***Building HDL Assets*** section.

The worker may be built as it was originally created by **ocpidev**, for any targets, before adding any code to implement the actual function of the worker.

As with any type of worker, compilation output is placed in the **target-*TTT*** subdirectory of the worker, for each target in the currently active set. An example explicitly specifying multiple targets (by specifying platforms) is:

```
ocpidev build --hdl-platform=modelsim --hdl-platform=zed
```

This would compile (and for non-simulation targets, synthesize) the worker for two targets for the indicated platforms. At this point, after building the initial generated worker code file, the directory representing the new worker looks like this:

```
xyz.hdl/
        xyz.xml               # OWD for this worker
        xyz.vhd               # editable source code for this worker
        gen/xyz-skel.vhd      # initial skeleton for this worker
            xyz-defs.vhd      # definitions enabling instantiation
            xyz-impl.vhd      # the generated shell of the worker
            xyz.build         # the generated build config file
        target-zynq/          # directory for results of zed build
            xyz.edf           # zynq synthesized netlist file
            xyz-xst.out       # log of tool output
        target-modelsim/      # dir for results of modelsim build
            xyz/*             # modelsim compilation result files
            xyz-modelsim.out  # log of tool output
```

The **gen** directory and all the **target-*** subdirectories are generated, and should not be edited. More details about these files are described below. As they are generated files, they are removed when the **ocpidev clean** command is issued.

3.4 HDL Worker Description File: the OWD XML File

This file specifies characteristics for an HDL worker which expand on those found in the spec file (OCS). In many cases it can be empty or left as it is. A default file is generated when the worker is created. Aspects of the OWD that are common to all authoring models are described in the CDG. Aspects specific to the HDL authoring model are described here.

Some reasons to customize the OWD XML file for an HDL worker are to specify:

- ***Additional implementation-specific properties***
  You can add additional worker properties for the implementation, beyond what is in the OCS/spec file. The **`property`** element accomplishes this, with the same XML syntax as in the OCS file.
- ***Additional accessibility to OCS 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 OCS to be also volatile in the implementation for debug purposes. The allowed attributes are described in the CDG.
- ***That an OCS property is in fact a parameter (compile-time) property***
  You can indicate that *in this worker*, the OCS property is actually a compile-time value. This only applies to **`initial`** properties in the OCS.
- ***Which control operations this worker will implement***
  You can specify which control operations are in the implementation: none are required. This uses the top-level **`ControlOperations`** attribute of the OWD.
- ***Interface style, and implementation and clocking attributes for data ports***
  You can specify whether the port uses a stream or message interface, and provide details for those interfaces (e.g. clock domain, data path width, or whether the port supports aborting messages). See Attributes of HDL Worker Data Ports.
- ***Specify that the control interface should support raw properties***
  This is an alternative method of accessing properties described in Raw Properties Access. The XML attributes used are described there.
- ***Specify build instructions, based on the needs of worker source code***
  These are described in the CDG, with a few HDL-specific ones here.

Only the last 3 items above may be in fact specific to HDL workers. All the rest apply to workers of all authoring models and are fully described in the CDG.

3.4.1 HDL Worker OWD Top Level Attributes

The top level of the HDL OWD is the XML **`HdlWorker`** element, which can have the XML attributes in the table below (beyond those defined for all OWDs in the CDG, such as **`Libraries`** and **`IncludeDirs`**). All are optional, and are only specified when the default behavior must be overridden.

Table 2: HdlWorker Attributes

HdlWorker Attribute Name	Data Type	Description
`DataWidth`	unsigned	The default physical 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 in the OCS.
`Cores`	list	A list of HDL primitive cores required by this worker.
`RawProperties`	boolean	A boolean value indicating whether the worker will use the ***raw*** property interface for all properties. The default is false. The ***raw*** interface is described below under Raw Access to Properties and is typically 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.
The attributes below are for HDL infrastructure workers coded to the "outer" or OCP interfaces and not supported for general users.		
`Outer`	boolean	Whether the worker implements the outer interface, used in internal OpenCPI modules or for legacy code.
`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 OCP 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.
`SizeOfConfig-Space`	unsigned 64-bit	Overrides the size of the configuration space in bytes. The default is based on the actual properties.
`Sub32BitConfig-Properties`	boolean	Whether this worker needs to support property values smaller than 32 bits (and thus requires byte enables in its interface). The default is based on the actual properties.

When using the **`Cores`** attribute, if a name in the list has no slashes, it is found by searching the path for HDL primitives as described in the ***HDL Search Paths*** section. If the name *does* contain slashes, it is the specific path name of the library or core, usually a relative path name within the same project. The **`IncludeDirs`** attribute, described in the CDG, can apply to Verilog includes. For HDL workers, the **`Libraries`** attributes described in the CDG refers to HDL primitive libraries, described in the ***HDL Primitive Libraries*** section.

The **Pattern** and **PortPattern** attributes are like **sprintf** format strings in C, with **%s** being the port name (**%S** capitalized), **%n** being **in** or **out** for signal, not data, direction (in to or out of the worker), **%m** being **m** or **s** for master/slave (and **%M** for **M**/**S**), **%0** or **%1** for port ordinal within OCP profile (0 origin or 1 origin), **%i**/**I** for using **i**/**I** for input, **o**/**O** for output, **%N** for **In** or **Out**, and **%w/W** for profile name (lower case or capitalized). These patterns are used when exporting or importing OCP-based workers.

3.4.2 Attributes of Worker Interfaces

Other aspects of the OWD for HDL workers are described in the sections describing each interface: the control interfaces, data interfaces (stream and message*), and service interfaces (time and memory*). They are summarized here, and described in detail later.

Table 3: HDL Worker Interface (Port) Attributes

Attribute Name	Which Interfaces?	Description
Timeout	Only control	Control clock cycles allowed for control ops
DataValueSize	Any data	The minimum unit of data, in bytes.
DataValueGranularity	Any data	The minimum multiple of data values
NumberOfOpCodes	Any data	Maximum number of opcodes to support
MaxMessageValues	Any data	Maximum message length in “data values”
ZeroLengthMessages	Any data	Declare support for zero length messages
Abortable	Stream data	Declare that messages can be aborted.
PreciseBurst	Stream data	Declare that messages will have known length at the start of the message.
SecondsWidth	Time	Bits in the **seconds** field of time-of-day
FractionWidth	Time	Bits in the **fraction** field of time-of-day.
AllowUnavailable	Time	Declare tolerance for time to be unavailable
Clock	Any data	The (other) interface to use the clock from
ClockDirection	Any data	The direction of the clock on this interface

* Support for memory interfaces is preliminary/partial in the current framework.

3.5 The Authored Worker: the VHDL Architecture or Verilog Module Body

This section describes how and where to write the actual VHDL or Verilog source code for a worker.

The functional code or "business logic" of the worker is in the **`architecture`** section (VHDL), in the **`xyz.vhd`** file (and possibly subsidiary source files). In Verilog, it is in the body of the **`module`**, and the file is **`xyz.v`**. This architecture/module is initially generated as a skeleton of the ***inner*** worker. It is surrounded by an automatically generated logic ***shell*** which provides robust and composable interfaces compliant with the Open Core Protocol (OCP) defined for the entire ***outer*** worker.

This shell and the **`entity`** declaration for the inner worker are found in the generated file: **`gen/xyz-impl.vhd`** for VHDL and **`gen/xyz-impl.vh`** for Verilog. The skeleton file, consisting of an empty inner worker, becomes the authored worker when the functional logic is written/inserted into that file.

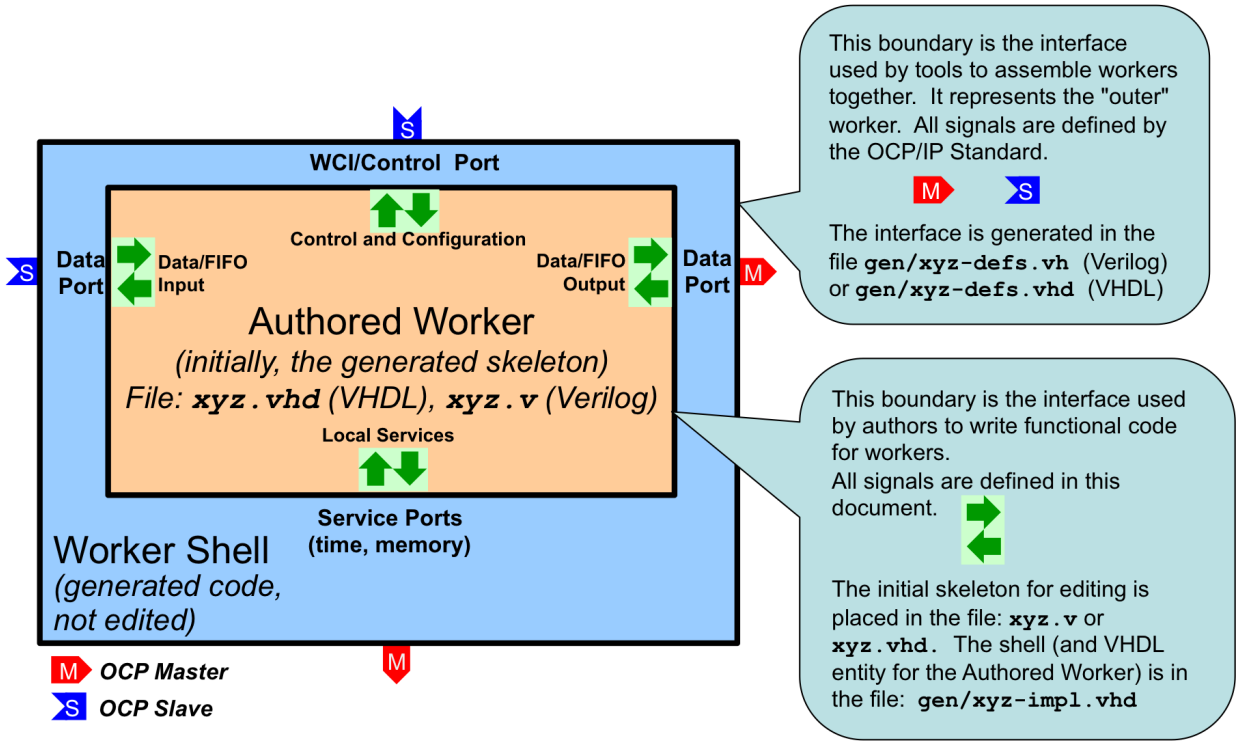

Figure 2: Worker Code and Files

All OpenCPI HDL workers are characterized by their properties, their data and service ports and their clocks, and usually the clocks are simply associated with ports, or even more simply, a single clock is commonly used with all ports. The job of implementing the inner worker is the job of:

- processing the various data ports' inputs to the worker to
- produce the various data ports' outputs of the worker,

- using the services provided at the service ports.

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.

E.g., with a "filter" worker that has a "sensor" input port, and a "result" output port, the VHDL entity declaration (in the **gen/filter-impl.vhd** file) 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 OCS and OWD files. These signals will be described below. Note that the name of the "control port" defaults to **ctl**. An example skeleton file for this worker, in the file **filter.vhd**, would be:

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

Note that while the overall worker has the name "filter", the entity being implemented in the architecture here is **worker**, the *inner* worker.

The clause **use ocpi.types.all** introduces all the data types in that package to the namespace for the architecture code of the worker. This is most convenient for using the built-in types provided by OpenCPI. However, if the author wants to avoid any collisions with their own types or functions, they can remove this **use** clause and fully qualify references to types provided by OpenCPI.

3.5.1 Signal Naming Conventions and Data Types

Other than the property access signals described below, the signals in these worker interfaces are mostly a combination of IEEE **std_logic_vector** and a boolean type, **bool_t,** that is used for various boolean indicator signals. All these VHDL types and related constants are defined in the **ocpi.types** package from the **ocpi** HDL primitive library.

The 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 convert 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**. These types and constants may also be used in user-written primitives, and are used in code automatically generated by OpenCPI.

In VHDL, 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 defined by OpenCPI use the **_e** suffix.

OpenCPI uses the term ***port*** to mean a high level data flow interface in and out of all types of workers. This conflicts with the use of the term in VHDL and Verilog, which means the individual signals (of any type) that are the inputs and outputs of an entity (VHDL) or module (Verilog).

In this section on HDL workers, this document uses the term ***interface*** to be the HDL worker's set of input and output port signals that correspond to the high level OpenCPI ports as defined in the OCS and OWD for the HDL worker. We also use the term ***interface*** for the implicit control port of all workers. An HDL worker has a control interface (for the implicit control port), data interfaces (for the explicit data ports defined in the OCS), and service interfaces (for service ports as defined in the HDL worker's OWD).

3.6 The Control Interface to the HDL Worker.

Every HDL worker has a control interface that at a minimum provides a *control* clock and associated reset into the worker. Normally, the control interface also is used to:

- 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 interface is used (**`ctl`**), the input signals are 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.

When the spec for the HDL worker (in its OCS) has the (rarely used) **`NoControl`** attribute set to true, only the clock and reset signals are present. In this case no signals associated with control operations or properties are present and there are no **`ctl_in`** or **`ctl_out`** signals. Only **`wci_clk`** for the clock signal and **`wci_reset`** for the reset signal are present.

3.6.1 XML attributes for the HDL Worker's Control Interface

Most aspects of the control interface are generically specified either in the OCS (e.g. the **`NoControl`** attribute), or at the top level of the OWD XML (e.g. the **`ControlOperations`** attribute). Several additional control interface attributes for HDL workers may be specified in a **`ControlInterface`** child element of the OWD. One example is the **`Timeout`** attribute described below. An example of an HDL OWD with this attribute would be:

```
<HdlWorker>
  <ControlInterface Timeout='100'/>
  ...
</HdlWorker>
```

The following table contains attributes that may be specified for the **`ControlInterface`** element:

Table 4: HDL Worker ControlInterface Element Attributes

Attribute Name	Value Type	Description
`Timeout`	ULong	The minimum number of control clock cycles that should be allowed for the worker to complete control operations or property access operations. When this number is exceeded the worker is considered inoperative and a timeout error is reported. The worker completes these operations using the **`ctl_out.done`** or **`ctl_out.error`** signals described below. The default value is 16.

3.6.2 Reset Behavior and Initializations in HDL Workers and Infrastructure

Per (at least) Xilinx recommended practice, OpenCPI uses and generates resets synchronously,. Thus resets are implemented active high and are asserted and deasserted synchronously. There are several reasons for this policy, but one is that less logic is typically needed to implement the resetting of register state.

At power up or reconfiguration, resets are asserted, so they will be asserted on the *first* clock edge. Per the OCP specification, resets will always be asserted for at least 16 clock cycles.

If registers (state) truly need an initial value (e.g. for simulation cleanliness or glitch-free initialization or sim-vs-synth consistency), it is preferred to set an initial *default expression* value in VHDL or Verilog, rather than using asynchronous reset. This is done by providing an initial value expression in the signal declaration. Note that current Xilinx (Vivado, ISE, and ISIM), Intel/Altera (Quartus), and Mentor (Modelsim) support such initialization without using any resources.

In the OpenCPI HDL infrastructure, applying resets to register state is only used to serve a functional purpose, and not the default practice.

3.6.3 Clock and Reset Input Signals in the Control Interface

The signal **`ctl_in.clk`** is the clock for all other control port signals as well as the *default* clock for all other data or service ports of the worker. The **`ctl_in.reset`** signal (asserted high) is asserted and deasserted synchronously with this clock. This **`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`** control operation (see below). The control reset, like all other resets generated by the OpenCPI infrastructure, is initially asserted.

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

3.6.4 Life Cycle/Control Operation Signals in the Control Interface

Other than the control **`reset`** signal, the life cycle of all workers is managed by life cycle control operations, according to the diagram below. When a worker's control reset is deasserted, it enters the ***exists*** state. Control operations cause state changes as shown. When control operations fail, the ***unusable*** state is entered. The worker autonomously enters the ***finished*** state, without any control operation.

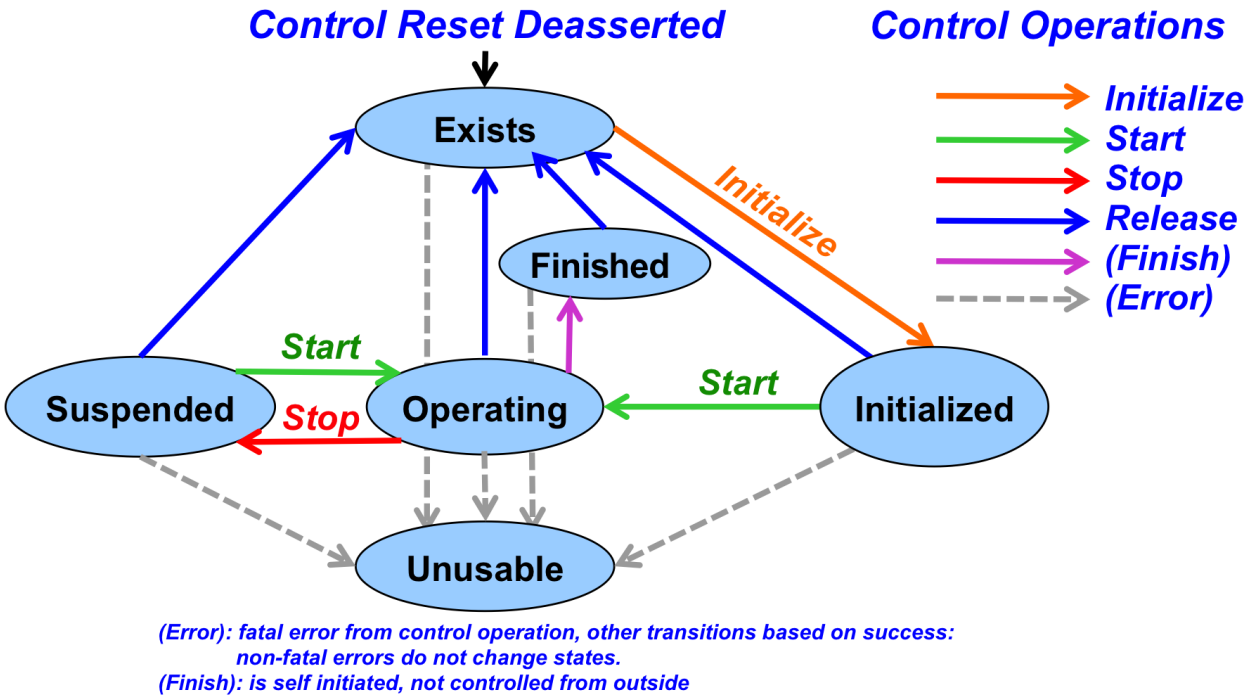

Figure 3: Control Operations and States

In simple and common cases, when the worker has no need to implement any of these operations, there is a single input signal indicating when the worker should *operate*, called **`is_operating`**. I.e., after **`reset`** is deasserted, the worker should *operate* only when this **`is_operating`** signal is asserted. Many HDL workers use this signal (as well as **`clk`** and **`reset`**) and no others in the control interface.

The **`is_operating`** signal indicates that the worker has been *started* and should now perform its function. *A worker must not perform any data transactions at its data ports unless* ***`is_operating`*** *is true.* This is necessary for robust system-level application control to suspend and resume all or parts of an application.

An example of a simple HDL worker would be:

```
architecture rtl of filter_worker is
  signal mystate_r : std_logic_vector(7 downto 0); -- some state
begin
  process (ctl_in.clk) is
  begin
    if rising_edge(ctl_in.clk) then
      if its(ctl_in.reset) then
        mystate <= "01010101";
      elsif its(ctl_in.is_operating) then
        -- do the clocked functions of this worker
      end if;
    end if;
  end process;
end rtl;
```

This allows the worker to be suspended and resumed since nothing happens during suspension, when **is_operating** is false.

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, the worker is considered in the ***exists*** state after **reset** is deasserted. The worker's OWD specifies whether **initialize** is implemented by this worker.

When a worker needs to explicitly support other control operations, there are two input and two output signals it may 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**. Otherwise the choices are: **INITIALIZE_e**, **START_e**, **STOP_e**, **RELEASE_e**, **BEFORE_QUERY_e**, and **AFTER_CONFIG_e**. The operation is terminated by the worker asserting the **done** or **error** output signals, after which the control operation is considered accomplished successfully (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**. 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**. The operation will be forced to complete with a timeout error if neither **done** nor **error** is asserted within the number of control clock cycles indicated in the **Timeout** attribute of the **ControlInterface** element.

All VHDL types specifically associated with the control interface are in the **wci** package of the **ocpi** library, including the enumeration values just mentioned. E.g.:

```
library ocpi;
...
if ctl_in.control_op = ocpi.wci.START_e ...
```

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** (or **error**) when the operation has completed. An example where initialization takes 10 clock cycles, would be:

```
architecture rtl of example_worker is
  signal init_count_r : unsigned(4 downto 0);
begin
  process (ctl_in.clk) is
  begin
    if rising_edge(ctl_in.clk) then
      if its(ctl_in.reset) then
        init_count <= (others => '0');
      elsif ctl_in.control_op = INITIALIZE_e then
        init_count <= init_count + 1;
      elsif its(ctl_in.is_operating) then
        -- do normal functions
      end if;
    end if;
  end process;
  -- initialize takes 10 clocks, all others take 1
  ctl_out.done <=
   '0' when ctl_in.control_op = INITIALIZE_e and init_count < 20
    else '1';
end rtl;
```

Another convenience input signal, **`state`**, indicates which life cycle 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`**, **`FINISHED_e`**, and **`UNUSABLE_e`**. These types are also in the **`ocpi.wci`** package.

Finally, there are two control output signals that the worker can use to indicate two other conditions. The first control output signal is **`finished`**. The worker uses this to indicate it has entered the ***finished*** state, and will perform no further work. This enables 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`**. Asserting **`finished`** will cause **`is_operating`** to become false, and **`state`** to become **`FINISHED_e`**.

The second, **`attention`**, allows the worker to indicate an interrupt or other condition to control software. This signal is for legacy compatibility and should not be used in new workers. It should be deasserted on **`reset`**.

Here is a summary of the control interface signals in the **ctl_in** record. The **bool_t** type is in the **ocpi.types** package, and the **control_op_t** and **state_t** are in ocpi.wci

Table 5: Control Input Signals

Signal	Type	Description
clk	**std_logic**	The clock for the control interface and the default clock for all other interfaces and ports.
reset	**bool_t**	Asserted high and synchronously, for the control interface, for at least 16 clocks. Initially asserted.
control_op	**control_op_t**	An enumeration type specifying the currently active control operation, with the value **no_op_e** when there is no active control operation. Control operations persist until **done** or **error** signal in the **ctl_out** record is true.
state	**state_t**	An enumeration type indicating the worker's current control state. Changes when control operation ends (via **done** or **error**) or **finished** is asserted.
is_operating	**bool_t**	Indicates the worker is started and is in an operating state. Persists until ***stop*** or ***release*** operation completes or **finished** or **reset** is asserted.
abort_control_op	**bool_t**	A command indicating that a long-duration control operation is being forcibly aborted. A pulse.
is_big_endian	**bool_t**	For dynamic endian workers, set at reset.

Here is a summary of the control interface signals in the **ctl_out** record:

Table 6: Control Output Signals

Signal	Type	Description
done	**bool_t**	Indicates successful end of a control operation. The default driven value is true indicating that all control operations complete in the same cycle they start.
error	**bool_t**	The signal indicating the unsuccessful end of a control operation. Default driven value is false.
finished	**bool_t**	A persistent indication, not deasserted after being asserted, until reset, that the worker has entered the ***finished*** state. Default is false.

These signals are assigned the default value in the entity port declaration.

3.6.5 Property Access

A worker's configuration properties are accessed via two additional record port 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 on what types of properties have been declared in the OCS and OWD.

When the worker shell is generated based on the OCS and OWD, the accessibility of properties determines which registers and signals are generated and made available to the worker code for each property. If any property is specified as being a ***raw*** property in the OWD, then the raw interface is also generated, and used for all such properties, as described below in raw access to properties. Otherwise the following rules apply:

Table 7: HDL Worker Property Logic Rules

Property is writable or initial	Property is Readback	Property is Volatile	Logic Description
Yes	No	No	Value is registered in the shell and register outputs are available in **`props_in`**
Yes	Yes	No	Value is registered in the shell and register outputs are available in **`props_in`**. The readback value is from the register outputs.
Yes	No	Yes	Value is registered in the shell and register outputs are available in **`props_in`**. Readback value is from the worker in **`props_out`**.
No	Yes	No	Readback value from the worker in **`props_out`**, but is cached by control software since worker is not expected to change it after it is operating.
No	No	Yes	The readback value is from the worker in **`props_out`**.

In the tables below, for a property called **`foo`**, the signals will be present as described. The types of the signals are all in the **`ocpi.types`** package.

The signals possibly present in the **`props_in`** record are in the following table.

Table 8: HDL Worker Property Input Signals

Signal in props_in	Included when:	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, type is sequence	**`ulong_t`**	Registered 32 bit unsigned number of elements when property is a sequence.
`foo_written`	Writable	**`bool_t`**	Indicates the entire value is being written. Persists until **`ctl_out.done`**, **`ctl_out.error`** or **`ctl_in.reset`**.
`foo_any_written`	Writable and (array or sequence or string)	**`bool_t`**	Indication that any part of the value is being written. Persists until **`ctl_out.done`**, **`ctl_out.error`** or **`ctl_in.reset`**.
`foo_read`	Volatile or (readback and not writable)	**`bool_t`**	Indication that the property is being read. Persists until **`ctl_out.done`**, **`ctl_out.error`** or **`reset`**.

The *indication* signals are valid during the access operation (until **ctl_out.done** or **ctl_out.error** is asserted). The operation will be forced to complete with a timeout error if neither **done** nor **error** is asserted within the number of control clock cycles indicated in the **Timeout** attribute of the **ControlInterface** element. Unless the OWD declares that the **readError** or **writeError** attributes are true, *control software will not expect and not check that errors have occurred*. The **readSync** and **writeSync** OWD property attributes currently have no function for HDL workers.

Any **writable** property is registered in the worker's ***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:

Table 9: HDL Worker Property Output Signals

Signals in props_out	Included when:	Type	Description
foo	Volatile or (readback and not writable)	*****	The worker-supplied value of the property, with the type dependent on the property declaration.
foo_length	Volatile or (readback and not writable) and sequence type	**ulong_t**	The worker-supplied 32 bit unsigned length (number of elements) when a sequence type.

3.6.6 Property Data Types

The **props_in** and **props_out** port signal records contain fields for property values with types that correspond to the property types defined in the OCS or OWD. All these types and associated conversion functions are defined in the **ocpi.types** package (in the **ocpi** HDL primitive library). This package is available in all workers. The worker author can decide to use the fully specified types (e.g. **ocpi.types.ulong_t**), or introduce the types into the worker architecture's namespace using:

```
library ocpi; use ocpi.types.all;
architecture rtl of worker is
```

For all property data types there is a:

- VHDL type name specified in the OCS with a **_t** suffix
- **from_*<type>*** conversion function from the type to **std_logic_vector**
- **to_*<type>*** conversion function from **std_logic_vector** to the type
- **to_*<type>*** conversion function from the related VHDL type (below) to the type
- ***<type>*_min** (for signed types) and a ***<type>*_max** constant for minimum and maximum values of the type
- ***<type>*_array_t** type for array or sequence property values, with a range of **(0 to length - 1)**
- **to_slv** conversion function from each ***<type>*_array_t** to **std_logic_vector**.

- **to_*<type>*_array** conversion function from **std_logic_vector** to ***<type>*_array_t**.

For example, for the **ushort** type, the **ocpi.types** package contains:

```
subtype ulong_t is unsigned (31 downto 0);
type ulong_array_t is array (natural range <>) of ulong_t;
constant ulong_max : ulong_t := x"ffff_ffff";
function to_ulong(c: natural) return ulong_t;
function to_ulong(c: std_logic_vector(31 downto 0)) return ulong_t;
function from_ulong(c: ulong_t) return std_logic_vector;
function to_slv(a: ulong_array_t) return std_logic_vector;
function to_ulong_array(a: std_logic_vector) return ulong_array_t;
```

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**.

The types are summarized in the following table, with extra conversion functions specific to each type.

Table 10: VHDL Types for Properties

VHDL type	Based on	Width	Extra Conversion Functions
uchar_t	IEEE unsigned	8	**to_uchar(n : natural)**
char_t	IEEE signed	8	**to_char(i : integer)** **to_char(c : character)** **to_character(c : char_t)**
ushort_t	IEEE unsigned	16	**to_ushort(n : natural)**
short_t	IEEE signed	16	**to_short(i : integer)**
ulong_t	IEEE unsigned	32	**to_ulong(n : natural)**
long_t	IEEE signed	32	**to_long(i : integer)**
ulonglong_t	IEEE unsigned	64	**to_ulonglong(n : natural)**
longlong_t	IEEE signed	64	**to_longlong(i : integer)**
float_t	std_logic_vector	32	**to_float(r : real)** (not synthesizable)
double_t	std_logic_vector	64	**to_double(r : real)** (not synthesizable)
string_t	char_t	8	**to_string(s : string, length : natural)** **from_string(s : string_t) return string;**
bool_t	std_logic	1	**to_bool(b : boolean)** **to_bool(b : std_logic)** **its(b : bool_t) return boolean**

3.6.7 Raw Access to Properties

There is an alternative property access method where a worker must manage the storage and addressing of individual property values itself. This is called the ***raw*** property interface. There are two primary use cases for this method:

- Device workers using properties to access hardware registers outside the FPGA, e.g. via I2C, SPI.
- Application workers that need to arrange the storage of property values for more efficient storage of large values, e.g. in a block memory managed by the worker.

In both cases, this avoids register duplication for property values, either off or on chip. The ***raw*** property interface does not apply to **`parameter`** properties.

The use of the raw property access interface is indicated in one of three ways:

1. Set the boolean **`raw`** attribute of the property (in **`Property`** or **`SpecProperty`** elements) to **`true`** to indicate those properties that should be accessed using the **`raw`** interface.
2. Set the boolean **`rawProperties`** top-level worker attribute to **`true`** to indicate that all properties defined for the worker (in OCS or OWD, except parameters), will use the raw interface.
3. Set the **`firstRawProperty`** string attribute to the name of the first property that should use the raw interface, as well as all subsequent properties.

All properties designated as raw will be accessed in an address space with each property aligned based on its type, with the first raw property having raw address 0. The **`padding`** property attribute can be specified in OWDs to explicitly align the addresses of raw properties.

The input signals (in the **`props_in`** record) for the raw interface are:

Table 11: Raw Property Input Signals

Signal in `props_in`	Signal included when:	Signal Description
`raw.address (31 downto 0)`	Always	The byte offset from the first raw property, of the property being accessed.
`raw.byte_enable (3 downto 0)`	Some raw property is less than 32 bits.	The (4) byte enables for reading/writing bytes in the 32-bit data path of the control interface.
`raw.is_read`	Some raw property is readable/volatile	Access operation is reading a raw property, valid until **`raw.done`** or **`raw.error`**.
`raw.is_write`	Some raw property is writable/initial	Access operation is writing a raw property, valid until **`raw.done`** or **`raw.error`**.
`raw.data (31 downto 0)`	Some raw property is writable/initial	The data being written to the raw property, on the appropriate byte lanes for the offset.

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

Table 12: Raw Property Output Signals

Signal in props_out	Signal included when:	Signal Description
`raw.data (31 downto 0)`	Some raw property is readable/ volatile	The data value for the raw property being read, with values smaller than 32 bits (e.g. 8 or 16 bit values) aligned in the appropriate byte lanes. Valid and accepted when **`raw.done`** is asserted.
`raw.done`	Any raw properties	Indicates when the access cycle has completed successfully. Asserted for one cycle per access.

Signal in props_out	Signal included when:	Signal Description
raw.error	Any raw properties	Indicates when the access cycle has completed unsuccessfully. Asserted for one cycle per access.

When raw properties are being accessed, the **props_out.raw.done** and **props_out.raw.error** signals indicate when the access is complete (and for reading, when the **props_out.raw.data** signal is valid). These are analogous to the **done** and **error** signals in **ctl_out**, although they do not have default values and *must be explicitly driven by the worker*.

If the raw interface is accessing registers or block memories in the worker, the **raw.done** signal may be tied asserted if all access happens in a single cycle. When the raw interface is used to access external registers (e.g. accessing an I2C or SPI bus), it would be asserted by the worker for one cycle when the access is complete.

The following timing diagrams show examples of raw property accesses. Consider a worker with the following properties:

```
<Property Name="prop0" Initial="true" Raw="true"/>
<Property Name="prop1" Volatile="true" Raw="true"/>
<Property Name="prop2" Padding="true" Raw="true" Type="ushort"/>
<Property Name="prop3" Initial="true" Raw="true" Type="ushort"/>
```

The raw address map for these properties would be:

```
prop0: 0x00000000
prop1: 0x00000004
prop2: 0x00000008
prop3: 0x0000000A
```

The following timing diagram shows an example of a writing the value **0x3333** to **prop3**. In this case, the worker asserts **done** one cycle after the write data appears on the interface.

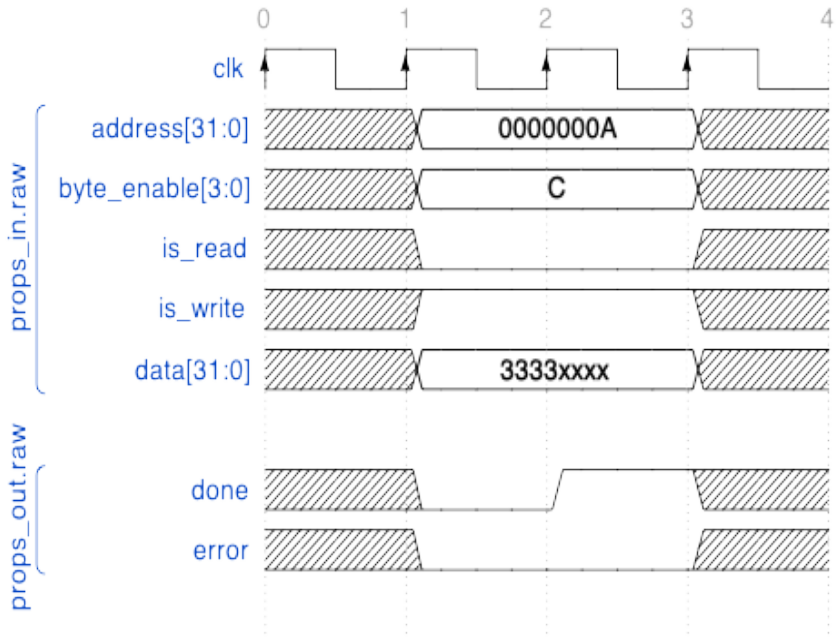

Figure 4: Raw Property Write to 16-bit Property

The following diagram shows an example of reading the value from **`prop1`**, which is assumed to be driven to **`0x1111`** by the worker. In this case, the worker asserts **`data`** and **`done`** one cycle after the read request appears on the interface.

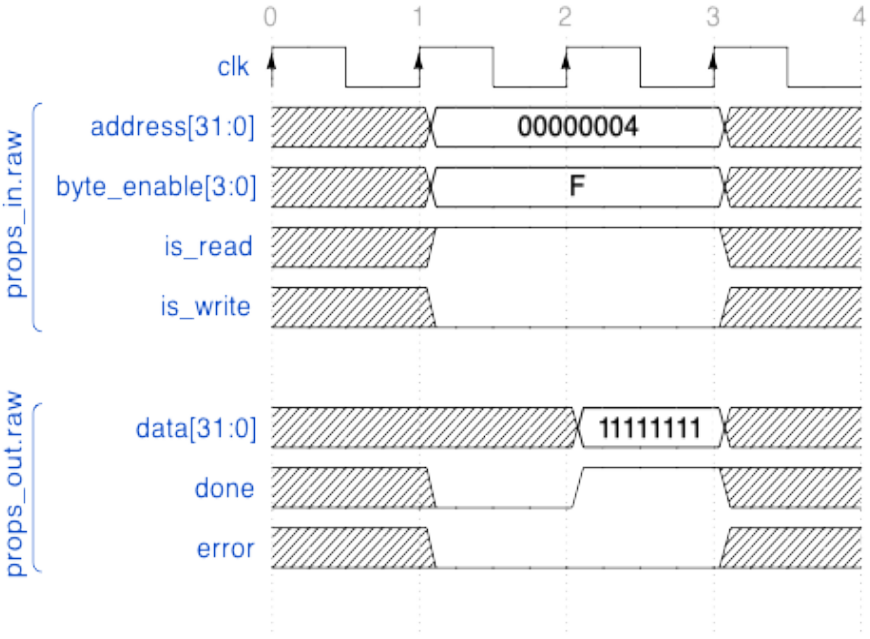

Figure 5: Raw Property Read of 32-bit Property

3.6.8 Summary of HDL Worker Control Interface

- Interface inputs are in the **`ctl_in`** signal port record.

- Interface outputs are in the **`ctl_out`** signal port record.
- Clock is **`ctl_in.clk`**.
- Reset is **`ctl_in.reset`**, asserted high, *synchronously*, for at least 16 cycles.
- Do no work unless **`ctl_in.is_operating`** is asserted, or a control operation is in progress.
- Optionally use **`ctl_out.done`** and **`ctl_out.error`** when control operations or property accesses take more than one cycle.
- Optionally set **`ctl_out.finished`**, if the worker has some semantic of being *finished*.
- Property inputs (written values, and access indicators) are in **`props_in`**.
- Property outputs (volatile values) are in **`props_out`**.
- If raw properties are used, the interface is in the **`raw`** member of **`props_in`** and **`props_out`**.
- If using **`ctl_out.done`** or **`ctl_out.error`** (or their raw equivalents), and more than 16 control clock cycles are required to complete the operation, set the **`Timeout`** attribute in the **`ControlInterface`** element.

3.7 HDL Worker Data Interfaces for OCS Data Ports

The OpenCPI ***data-plane*** is the collection of hardware and software infrastructure that conveys variable-length messages between workers of all types. Each message has an associated ***opcode***, which is an ordinal indicating the message type, among those defined in the protocol indicated for the port defined in the worker's OCS. The data-plane also conveys the EOF condition (see the section “EOF Indications on Data Ports” in the CDG). In summary, the data-plane conveys four things: data, message boundaries, opcodes, and EOF.

HDL worker data interfaces correspond to the ports defined in the worker's OCS and convey these four things in to and out of HDL workers. The signals in these interfaces carry data, opcodes, message boundaries and EOF. These interfaces have a *physical* data width, specified by the HDL-specific **`dataWidth`** attribute. It indicates the number of wires over which the message data will be conveyed. The data is conveyed as a sequence of fixed-width ***words*** of **`dataWidth`** bits. How message data is packed into these words is described in the Message Payloads section.

When there is only one message type in the protocol, no opcode is conveyed and no interface signals are present for opcodes. Data interfaces may have zero width when all messages in their protocol have no arguments and thus are all zero length: message opcodes are all that is conveyed. If there is only one message in the protocol, and it has no arguments, then there is no data, no opcode, but still an indication of a message being conveyed. This is essentially an “event pipe” or “pulse” interface.

Data interfaces implement flow control: an output cannot be produced unless permission is granted. HDL workers explicitly accept data at input interfaces *when offered*, and only produce data at output interfaces *when permitted*.

Data interfaces convey the EOF condition. Most workers ignore the EOF condition and it is automatically propagated from its input(s) to its output(s). Prior to worker interface version 2, EOF was not supported but was overloaded with zero-length messages with opcode zero.

Worker data ports can be implemented in two different styles: ***stream*** or ***message***. Stream interfaces are FIFO-like with extra qualifying bits along with the data indicating message boundaries, byte enables and EOF. Message interfaces are based on addressable message buffers. Each style has its own section below. The style of a port used in the HDL worker is indicated by the XML element that describes it in the OWD. The stream interface is the default style. If all attributes of the interface have default values, no indication of this port or style is required in the OWD.

The stream interface style is indicated by the **`<StreamInterface>`** element in the OWD. This element specifies interface characteristics and may also override any of the protocol summary attributes explained in the CDG (protocol and OCS port section).

An example of the per-data-port XML element is:

```
<StreamInterface name="sensor" dataWidth="64"/>
```

The example shows the **sensor** data interface port declared in the OCS is being further configured for a non-default **dataWidth**.

3.7.1 Message Payloads vs. Physical Data Width on Data Interfaces

As described above, message data is conveyed by the HDL worker data interfaces as a sequence of fixed-width words of **datawidth** bits. Message data is packed into these words according to the format described in the subsection "Message Payloads on Data Ports" in the "Protocol Specifications" section of the CDG. As shown in the example there, if the operation element in a protocol contains:

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

And the values of this payload are:

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

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

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

As mentioned earlier, HDL worker data interfaces have a *physical* width, specified by the HDL-specific **dataWidth** attribute. The width must be a multiple of the smallest data value in the protocol. In the example above, this would be 8 bits. If **dataWidth** was 8, the sequence of content bytes shown above would be how the payload appears on that byte-wide data interface. If the **dataWidth** was 16, the message would appear as:

Sequence # ►	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
15 downto 8	x	23	67	x	32	76	ba	fe
7 downto 0	01	45	89	x	10	54	98	dc

If the **dataWidth** was 32, the message would appear as:

Sequence # ►	**0**	**1**	**2**	**3**
31 downto 24	23	x	76	fe
23 downto 16	45	x	54	dc
15 downto 8	x	67	32	ba
7 downto 0	01	89	10	98

And if the **dataWidth** was 64, the message would appear as:

Sequence # ►	0	1
63 downto 56	x	fe
55 downto 48	x	dc
47 downto 40	67	ba
39 downto 32	89	98
31 downto 24	23	76
23 downto 16	45	54
15 downto 8	x	32
7 downto 0	01	10

The byte sequence remains the same regardless of **dataWidth**.

3.7.2 Byte Enables on Data Interfaces

Byte enables on data interfaces are only present when needed, and their presence is determined by a combination of the protocol summary attributes and the **dataWidth** of the interface.

Two relevant values are inferred from the protocol:

- ***DataValueWidth***: the smallest data value in the protocol.
- ***DataValueGranularity***: the least common multiple of data values among all messages in the protocol; all message lengths are a multiple of this number of data values.

The physical data width of the interface, **dataWidth**, must be a multiple of ***DataValueWidth***. When **dataWidth** is greater than ***DataValueWidth * DataValueGranularity***, byte enables are in the interface, since data words (of **DataWidth**) at the end of a message may be partially valid. E.g. if the **DataWidth** is 32, and ***DataValueWidth*** is 8 and ***DataValueGranularity*** is 1, the messages may have any number of bytes and thus the last 32 bit word of a message may have 1, 2, 3 or 4 valid bytes. In this context, a ***byte*** is a data value, and bytes *might* not be 8 bits. Here are some examples:

- Message is a sequence of short (16 bit) values, **dataWidth** is 16:
  DataValueWidth = 16
  DataValueGranularity = 1
  No byte enables required.
- Message is a sequence of short (16 bit) values, **dataWidth** is 32:
  DataValueWidth = 16
  DataValueGranularity = 1
  Byte enables (2) are required since sequences might be an odd number of shorts.
- Message is a sequence of *pairs* of short (16 bit) values, **dataWidth** is 32:
  DataValueWidth = 16
  DataValueGranularity = 2
  Byte enables not required since sequences are always a multiple of 2 shorts.

The exact naming and use of byte enable signals in data interfaces is described below.

3.7.3 Clocks for Data Interfaces

Data interface signals operate using the control clock provided on the control interface unless specified otherwise. There are two ways to override this behavior and specify that a data interface should *not* use the control clock:

1. Specify that the interface should have its own clock signal as part of the interface (and whether that signal is an input or output signal to/from the worker). This is indicated by the **`clockDirection`** attribute.
2. Specify that the interface should use the clock signal from another data interface on the same worker. This is indicated by the **`clock`** attribute.

The **`clockDirection`** attribute can have values of **`in`** or **`out`**. The presence of this attribute means there will be a signal in the interface named **`clk`**, which will be in the input record as an input signal if the value of the attribute is **`in`**, or in the output record if the value of this attribute is **`out`**. E.g. if the interface name is **`data`**, the signal in the worker will be **`data_in.clk`** or **`data_out.clk`**. Setting this attribute is saying that this interface operates in its own clock domain and will either accept as input, or drive as output, its clock.

The **`clock`** attribute is a string-valued attribute whose value is the name of another interface (port) on the worker. It indicates that the interface operates in the same clock domain as that other named interface, and thus the worker internally will use the clock signal associated with that other interface.

A common configuration of these attributes is when one interface (e.g. called **`data1`**) declares **`clockDirection='in'`**, and all other data interfaces declare **`clock='data1'`**. This means that all data interfaces of the worker are in one clock domain that is different from the control clock, and the data clock signal is fed into the **`data1`** interface. Note that the direction of the *clock signal* at **`data1`** is independent of the *data* direction at that interface.

Such workers are typically called “split clock” workers: the control clock is separate from the clock used for data flow and processing. See the section How Clocks are Connected in an HDL Assembly or Container for how split-clock workers are used in HDL assemblies. Split clock workers are generally considered best practice, but they must be written to ensure that any interactions between the control interface (e.g. volatile properties) and logic associated with the data interfaces, are handled with appropriate CDC (Clock Domain Crossing) logic.

3.7.4 Streaming Data Interfaces to/from the HDL Worker

The stream data interface is layered, with a simple default model usable by many workers, and a more advanced model that covers all possibilities.

The simple model:

- focuses exclusively on processing words of data and supporting flow control
- is designed to be similar to the AXI4-Stream interfaces found in other systems
- ignores signals used by the advanced model

- allows the worker to ignore message boundaries
- uses a small number of control signals to consume or produce data.

The advanced model gives the worker total control over message boundaries, byte enables and opcodes.

This data streaming interface is *versioned* according to the OWD **`version`** attribute described in the CDG. This attribute indicates the specific interface semantics. This distinction between simple and advanced interfaces was established with version 2 of this interface (introduced in release 1.5 of OpenCPI). Prior to that there was a single more complex interface model. The interface description here is the version 2 interface, with notes where a signal had a different semantic prior to version 2.

3.7.4.1 Attributes of Streaming Data Interfaces

Recall that the **`dataWidth`** attribute at the top level of the HDL OWD specifies the default physical width of all data interfaces. If most interfaces have the same width, it may be most convenient to specify this width at the top level **`HdlWorker`** element. Specifying **`dataWidth`** in the **`StreamInterface`** element applies only to that port. These **`dataWidth`** attributes may also be expressions based on the values of parameter properties.

The XML attributes of the **`StreamInterface`** element are in the following table:

Table 13: XML Attributes of StreamInterface Elements

Attribute in StreamInterface	**Attribute data type**	**Attribute Description** ***(optional unless specified as required)***
`name`	string	The name of the corresponding port in the OCS. Required.
`dataWidth`	unsigned	The width of the data path for this interface. The default is the width of 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`**)
`workerEOF`	boolean	The worker will take responsibility for asserting EOF at this *output* port. If not set, this will be done automatically based on the first input port (see CDG). Only valid for output ports. Not valid for versions < 2.
`insertEOM`	boolean	End of message will be automatically asserted appropriately. The worker need not deal with or assert EOM. The worker may still assert EOM if it needs to. Only valid for output ports. Not valid for versions < 2.
`clock`	string	The name of the other interface that this interface will use for its clock. If not specified, the control clock is used.
`clockDirection`	string	The value must be **`in`** or **`out`**, and indicates that this interface has its own clock, whose direction is specified by this attribute.

3.7.4.2 Signal Summary for Streaming Interface

The signals are described in the following tables and explained in sections after the table. The term “qualified by xxx” means “is only usable when xxx is asserted”.

*Table 14: Stream Interface **Input** Signals for **Input** Ports*

Signal	When Included	Signal Description
clk	If clockDirection == in	Clock for the signals of the interface if **clockDirection** attribute is **in**. Otherwise use **ctl_in.clk**.
reset	Always	Port is being reset either by the worker connected to it or by the control port (ctl_in.reset, but in this port's clock domain).
data	If dataWidth > 0	The input data, qualified by **valid**. Width is **dataWidth**.
valid	If dataWidth > 0	The **data** signals hold message data, implies **ready** and **ctl_in.is_operating**. *Prior to v2, qualified by **ready**.*
byte_enable	If dataWidth > 0 **	Which data bytes are valid; qualified by **valid**. **Included when the **DataValueWidth * DataValueGranularity < dataWidth**. Width is **dataWidth**/**dataValueWidth**.
ready	Always	Can consume, and metadata bits are valid/usable. Implies **ctl_in.is_operating**.
som	Always	The start-of-message indication. Qualified by **ready.**
eom	Always	The end-of-message indication. Qualified by **ready.**
abort	If Abortable	The message is being aborted. Qualified by **ready.**
opcode	numberOf OpCodes > 1	Opcode for the current message. Valid from start of message to end of message on input. Initially qualified by **ready**. Width is ceil(log2(**numberOfOpCodes**)).
eof	Version > 1	Indicates EOF condition. Not qualified. Persists until reset when asserted. Only asserted after an **eom** or before first message.

All the above input signals are in the ***<port>*_in** VHDL record entity port.

*Table 15: Stream Interface **Output** Signals for **Input** Ports*

Signal	When Included	Signal Description
clk	If clockDirection == out	Clock for the signals of this interface, driven by worker, when **clockDirection** is **out**. Otherwise use **ctl_in.clk**.
take	Always	Indicates that the worker consumes the word if **ready** or **valid** (v2+) is also asserted. It can be asserted before **ready** or **valid** is asserted.

All the above input signals are in the ***<port>*_in** VHDL record entity port.

When using the port in the ***simple*** mode that ignores message boundaries, only the **data** and **valid** and **take** signals are used (and **byte_enable** when present).

When the **dataWidth** is zero, only the **ready** and **take** signals are used.

For interface versions prior to 2, **eof** signal did not exist and the **valid** signal was qualified by **ready** and could not be used unless **ready** was asserted.

*Table 16: Stream Interface **Input** Signals for **Output** Ports*

Signal	When Included	Signal Description
clk	If clockDirection == in	The clock for all the signals of this interface, if **clockDirection** is **in**, otherwise **ctl_in.clk** is used.
reset	Always	Port is being reset either by the worker connected to it or by the control port (**ctl_in.reset**, in this port's clock domain)
ready	Always	Indicates that the shell will consume the word (metadata and maybe data) if **give** or **valid** is also asserted. It implies **ctl_in.is_operating**. It may be asserted before **give** or **valid** is asserted.

All the above input signals are in the ***<port>*_in** VHDL record entity port.

*Table 17: Stream Interface **Output** Signals for **Output** Ports*

Signal	When Included	Signal Description
clk	If clockDirection == out	Clock for the signals of this interface, driven by the worker if **clockDirection** is **out**. Otherwise **ctl_in.clk** is used.
data	If dataWidth > 0	The output data, qualified by **valid**. Width is **dataWidth**.
valid	If dataWidth > 0	The **data** signals hold message data. Implies **give** . *Prior to v2, qualified by **give.***
byte_enable	If dataWidth > 0 **	Which data bytes are valid; qualified by **valid**. **Included only when the ***DataValueWidth * DataValueGranularity* < **dataWidth**. Width is **dataWidth/dataValueWidth.**
give	Always	Indicates metadata and maybe data is valid/usable. When asserted, one of ***som/eom/valid/abort*** must be asserted.
som	Always	The start-of-message indication. If previous **give** had **eom**, then **som** is assumed on the next **give**. Qualified by **give**.
eom	Always	The end-of-message indication. Qualified by **give**.
abort	If Abortable	The message is being aborted. Qualified by **give**.
opcode	numberOf OpCodes > 1	Opcode for the this message. Must be valid with ***som*** on output.
eof	Always	Indicates EOF condition (not present prior to v2) Not qualified. Persistent (until reset)

All the above output signals are in the ***<port>*_out** VHDL record entity port.

When using the port in the ***simple*** mode that ignores message boundaries, only the **data** and **valid** and **ready** signals are used (and **byte_enable** when present).

When the **dataWidth** is zero, only the **ready** and **give** signals are used.

For interface versions prior to 2, **eof** signal did not exist and the **valid** signal was qualified by **give** and could not be used unless **give** was asserted.

The **data** and **byte_enable** signals are **std_logic_vector**. The opcode signal is also **std_logic_vector** when there is no protocol in the OCS, otherwise it is an enumeration of type ***<protocol>*_OpCode_t**, with each operation having an enumeration constant **<protocol>_*<op>*_op_e**. These opcode types are in the **work.*<worker>*_worker_defs** package. The other signals are all **bool_t**, from the **ocpi.types** package.

3.7.4.3 Metadata and Message Boundaries Used for Stream Interfaces

The information flowing out of or into stream interfaces are variable length messages conveyed using fixed width ***words*** of data, along with metadata. The metadata associated with every word presented at the interface includes:

- **Valid**: indicates whether the data is present and valid
- **SOM**: start of message: indicates that this word is the first in a message
  version >= 2: on input will always be coincident with first valid data word
  version <2: may be present whether or not there is valid data present.
- **EOM**: end of message: indicates that this word is the last in a message
  version >= 2: will always be coincident with the last valid data word
  version <2: may be present whether or not there is data present.
- **Abort**: (optional) whether this word is indicating the end of an aborted message
- **Byte_enable**: (optional) indicates, *if **valid** is true*, which bytes in the data word are valid. This signal is all ones on all but the last valid word of a message.

These metadata signals, as well as **data**, are *all* registered on input ports, outside the worker's code, enabling simple workers to be written in a combinatorial style. Only the **valid** signal (and **data**) is used in the simple usage model.

On input, the optional **abort** indicator, when present and asserted, forces the **EOM** indicator on, and the **valid** indicator off. When **abort** is asserted on output, **EOM** and **valid** are ignored and assumed true and false, respectively. When there is no **abort**, the three metadata bits, **SOM**, **EOM**, and **valid**, can be in various combinations, with the following valid combinations:

Table 18: Metadata in Stream Interfaces

SOM	Valid	EOM	Signal Description
1	0	0	The start of a message, with no data (yet). *Only on output*.
1	1	0	The start of a message, coincident with the first word of data.
1	0	1	A zero length message, with no data, in a single word.
1	1	1	A single word message.
0	1	0	A data value in the middle of a message
0	1	1	A data value, coincident with the end of the message
0	0	1	An end of message with no data. *Only on output*

On input, the worker can assume that the sequence of metadata values will be correct, meaning all messages will have a SOM and an EOM. The valid sequences are:

- The first word of a message must have **110**, **101**, or **111** (i.e. the **SOM** bit set).
- After a message is started but not simultaneously ended by **101** or **111**, any of **010**, or **011** may occur (i.e. be data-without-**EOM**, or data-with-**EOM**).
- After a word with **EOM** set, the next word must have **SOM** set.

On output, the worker may also produce a **SOM** with no data (and no **EOM**) and an **EOM** with no data (without **SOM**), but a worker will never see these combinations on input.

3.7.4.4 Validity/Qualification of Interface Signals

On both input and output ports the **ready** signal is an input that allows data to move and is prequalified by the **ctl_in.is_operating** signal from the control interface. On input, the **som**, **eom**, **data,** and **byte_enable** signals are only meaningful if **ready** is asserted. However, the input port **valid** signal, when asserted, implies **ready**, and can thus be used as the sole control signal of the input interface when message boundaries are being ignored and all messages contain data.

The **take** signal on the input interface is the handshake to accept a word of data and metadata. It is not qualified. Similar to AXI interfaces, the **take** signal (like the AXI "ready" signal) may be asserted before input **ready** if the worker is prepared to accept data/metadata.

On output, none of the output signals are used unless the **ready** input signal (of the output interface) is asserted. Similar to input, the **valid** signal by itself indicates that valid metadata and data is being offered for output (and thus implies **give**)..

In summary, when message boundaries are ignored (and **insertEOM** is set for the output port), the entire interface is controlled by **valid** and **take** at an input port, and **valid** and **ready** at an output port. The semantics are the same as AXI-Stream signaling (with the OpenCPI **take** signal acting like the AXI **ready** signal).

When a more advanced worker needs to manage message boundaries in a more complex way, the **som** and **eom** signals can be used on either or both sides, along with the input **ready** and output **give** signals.

Prior to version 2, the **valid** input signal was also qualified by the **ready** signal and the **ready** signal was not qualified by **ctl_in.is_operating**. On output the **valid** was qualified by **give**. The changes introduced by version 2 can be summarized as:

- **valid** can be used on input and output without using **ready** (on input) or **give** (on output)
- **ready** can be used on input and output without using **ctl_in.is_operating**
- **take** can be asserted before **ready** on input, and **give** can be asserted before **ready** on output.

3.7.4.5 Output Message Sizes

The maximum permitted size of a message produced at an output port, in bytes, is always asserted by the system in the built-in initial property named **ocpi_buffer_size_<*port*>**. of type **ushort_t**. For bounded protocols, this value will always accommodate any message defined by the protocol. The worker can take one of the following approaches to managing the output message size:

- Let the system do it automatically by setting the **insertEOM** attribute and never driving the EOM signal at all.
- Let the system do it automatically (using **insertEOM**), but in some cases forcible terminate a message early by asserting EOM.
- Drive the EOM signal on output based on an EOM signal from an input port, when message sizes on input and output can and should be the same number of words.
- Intelligently drive the EOM per protocol, ensuring this maximum is respected.

Sometimes a worker must specifically determine and implement output message sizes based on some other criteria. An example would be a worker that produced a fixed size message regardless of the size of input messages, essentially accumulating data from input messages into fixed size output messages that should not necessarily be as large as allowed by the **ocpi_buffer_size_<*port*>** attribute.

Using **insertEOM** is best practice since it allows the application developer to adjust the message sizes to tune application performance for latency or throughput.

3.7.4.6 Flow Control (a.k.a. Back Pressure) On Streaming Interfaces.

Both input and output stream interfaces have a **`ready`** signal, that is *always input to the worke*r, indicating that data can be consumed or produced.

The rules for *input* interfaces are:

- **`ready`** indicates that the metadata and perhaps the data signals are valid
- **`valid`** indicates that metadata *and* data signals are valid
  Prior to version 2, this signal is qualified by **`ready`**.
- if **`ready`** is not asserted, none of the metadata signals are valid or meaningful
- the worker ***takes*** input data when **`ready`** or **`valid`** is asserted by asserting the **`take`** signal
- the **`take`** signal may be asserted early without **`ready`** (or **`valid`**) asserted

The rules for *output* interfaces are:

- **`ready`** indicates that metadata and perhaps data can be produced
- if the **`ready`** signal is not asserted, none of the metadata or data output signals are used.
- the worker ***gives*** data when **`ready`** is asserted by asserting the **`give`** signal; the **`give`** (or **`valid`**) signal may be asserted prior to **`ready`** being asserted.
- the worker may also use **`valid`** to imply **`give`**

Data flows according to FIFO semantics. Input data is presented as ***ready*** as if there is an input FIFO outside the worker that is ***not empty***. The worker consumes this data by ***taking*** it, which is as if it is ***dequeueing*** data from this ***not-empty*** FIFO. Similarly, output data can be produced when output is indicated to be ***ready*** as if there is an output FIFO outside the worker that is ***not full***. The worker produces this data by ***giving*** it, which is as if it is ***enqueueing*** data to this ***not-full*** FIFO.

These signals (**`ready/valid`**, **`take`**, **`give/valid`**) control the flow of data and metadata words through the interface. Here is a table of how this signal terminology compares to some other common interfaces with FIFO semantics: the classic FIFO interface, the AXI streaming interface, and the “native” Xilinx FIFO interface.

Meaning	OpenCPI	Classic FIFO	AXI	Xilinx FIFO
Data is available to consume	**`ready`**	**`not_empty`**	**`valid`**	**`!empty`**
Consume data	**`take`**	**`dequeue`**	**`ready`**	**`rd_en`**
Data can be produced	**`ready`**	**`not_full`**	**`ready`**	**`!full`**
Produce data	**`give`**	**`enqueue`**	**`valid`**	**`wr_en`**

In AXI interfaces, either signal (**`valid`** or **`ready`**) may be asserted early. The handshake (**`ready`**) can in fact be asserted early even when **`valid`** is not yet asserted.

With OpenCPI prior to version 2, it was invalid to assert **`take`** or **`give`** without **`ready`**. Version 2 brought alignment with AXI signaling. In Xilinx FIFO, **`rd_en`** and **`wr_en`** are ignored if the fifo is **`empty`** (input) or **`full`** (output).

In all cases (all these interfaces) data moves from producer to consumer when both sides assert their signals in the same cycle (at the same clock rising edge).

Since worker ports all have FIFO semantics, workers must be written to accommodate "back pressure". I.e. the **`ready`** signal on output interfaces may not always be asserted, so the output data is not accepted until then.

Example timing diagrams for this interface follow the signal descriptions below.

3.7.4.7 Timing Diagrams

The following diagram shows an input port where both the infrastructure (worker shell) and the worker respond one cycle after they see new input, resulting in a throughput of 3 clock cycles per data word. Both SOM and EOM are coincident with data.

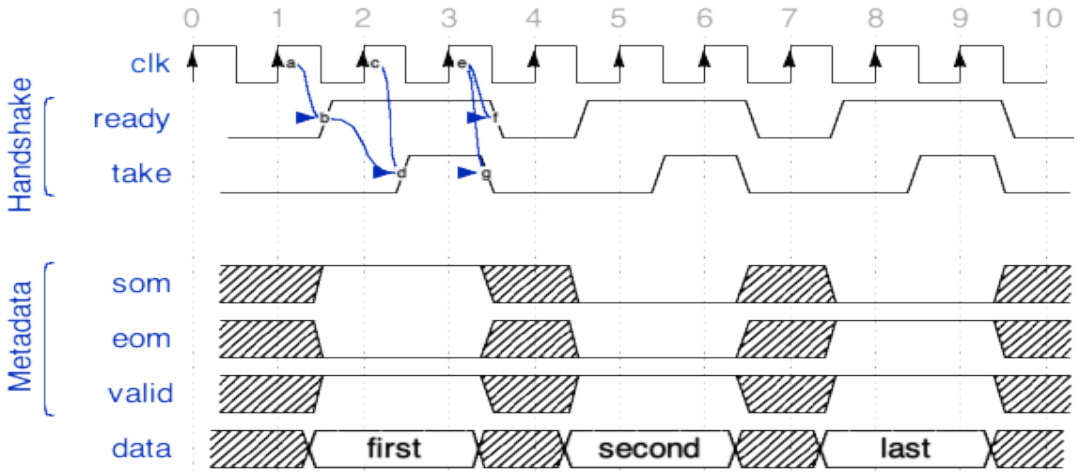

Figure 6: 3 word input message with delays on both sides

The following diagram shows an input port where only the worker responds one cycle after it sees new input, resulting in a throughput of 2 clock cycles per data word.

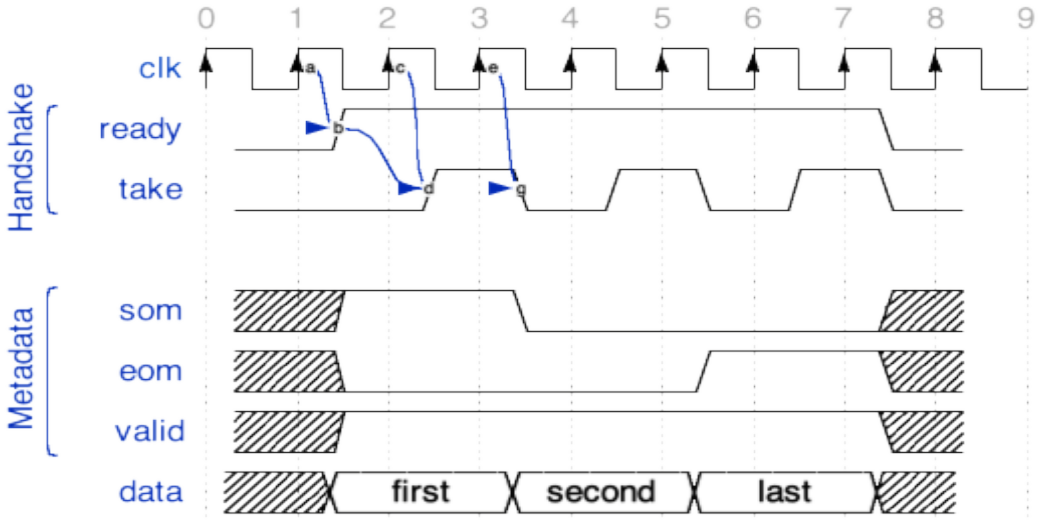

Figure 7: 3 word input message width worker adding delay

The following diagram shows an input port where both sides respond in the same cycle, with no delays, resulting in a throughput of 1 clock cycles per data word.

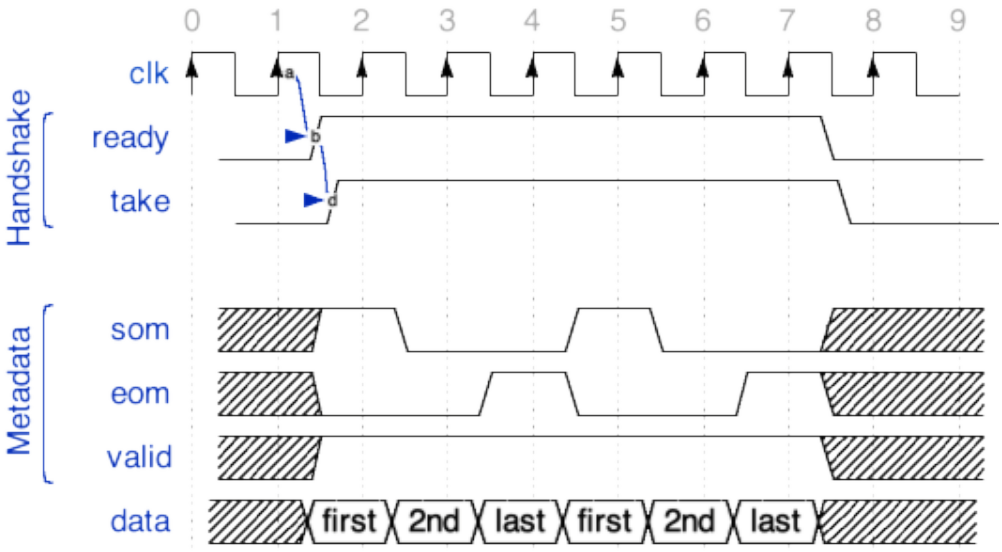

Figure 8: Two 3 word messages, with no delays on either side.

3.7.4.8 Source Code Examples

Here is a set of examples, showing the use of these signals. The first is a complete worker that is purely combinatorial, and simply adds a constant (3) to every data value from input to output. It is simply processing data and ignoring message boundaries. It sets **insertEOM** in its OWD since it is not interested in message boundaries.

No VHDL process or clocking or even reset is used since the computation takes place in a single clock cycle. No opcode or byte_enable is used since the protocol has a single operation and the data in that message is all the same size.

```
architecture rtl of worker is
begin
  out_out.data  <= std_logic_vector(unsigned(in_in.data) + 3);
  in_out.take   <= out_in.ready;
  out_out.valid <= in_in.valid;
end rtl;
```

If the same worker wanted to be sure that the output messages were the same size as the input messages, it would not set **insertEOM** in its OWD.

```
architecture rtl of worker is
begin
  out_out.data  <= std_logic_vector(unsigned(in_in.data) + 3);
  in_out.take   <= out_in.ready;
  out_out.give  <= in_in.ready;
  out_out.som   <= in_in.som;
  out_out.eom   <= in_in.eom;
  out_out.valid <= in_in.valid;
end rtl;
```

The next example shows a worker that inserts a special single-word all-ones message, with operation **xy**, every 8 messages it passes.

```
architecture rtl of worker is
  signal count     : unsigned(3 downto 0);
  signal inserting : bool_t := count = 8;
begin
  process (ctl_in.clk) is begin
    if rising_edge(ctl_in.clk) then
      if its(ctl_in.reset) then
        count <= (others => '0');
      elsif its(out_in.ready) then
        if its(inserting) then
          count <= (others => '0');
        elsif in_in.ready and in_in.eom then
          count <= count + 1;
        end if;
      end if;
    end if;
  end process;
  in_out.take    <= out_in.ready and in_in.ready and not inserting;
  out_out.give   <= out_in.ready and (count = 7 or in_in.ready);
  out_out.data   <= (others => '1') when inserting else in_in.data;
  out_out.som    <= inserting or in_in.som;
  out_out.eom    <= inserting or in_in.eom;
  out_out.valid  <= inserting or in_in.valid;
end rtl;
```

3.7.6 HDL Worker Data Interface Summary

- Only move data when the port is ready.
  The **`valid`** input signal implies the input port is **`ready`** (v2+).
- Input data is valid only when explicitly indicated at the input interface.
- Output data can only be moved when flow control allows it at the output interface.
- Output message boundaries must be supplied and respect maximum message sizes, unless **`insertEOM`** is set in OWD.
- Output message boundaries can be derived from input message boundaries.
- Remember to (in most cases) convey zero length messages from input to output (In V1 workers only. Otherwise simply respect the protocol).

3.8 Time Service Interface

This interface provides "time of day" information to the worker, to the precision requested in the OWD via attributes to the **`TimeInterface`** element. Time of day values are supplied to the worker in the clock domain of this interface, which defaults, like all interfaces, to the control clock. The **`clock`** and **`clockDirection`** attributes are valid and have the same meaning as in the Clocks for Data Interfaces section above.

The signals for the time service interface are in the **`time_in`** signal port record and are described below. If the default port name is overridden, the signal port record could be ***`<port-name>`*`_in`**.

Table 19: Time Service Signals

Signal in `time_in`	**Width**	**Signal Description**
`seconds`	**`SecondsWidth`** attribute of **`TimeService`** element	The entire seconds part of the time-of-day, in GPS time (no leap seconds). If the width is 32 it is absolute time. If width less than 32, it is just a relative time truncated preserving the LSB, to that value, and wraps. The LSB is always 1 second; VHDL type is IEEE numeric unsigned. Width may be zero, in which case this signal is not present.
`fraction`	**`FractionWidth`** attribute of **`TimeService`** element	The binary fraction of a second, with the radix point to the left of the MSB. If width is 32 bits, the LSB represents 2^-32 seconds, or ~233 ps. If width is less than 32, the MSB are preserved, such that the MSB is always ½ second. Width may be zero, in which case this signal is not present.
`valid`	1 (**`bool_t`**)	Indicates when the time of day is valid. Present only when the **`AllowUnavailable`** attribute is true.

3.9 Memory Service Interfaces

This interface provides access to memory. [Not supported in 2021Q2]

4 Building HDL Assets

Building workers is similar across different authoring models and languages: typing the **`ocpidev build`** command in a worker, library, or project directory builds workers for a specified set of targets. If the worker depends on lower level "primitive" libraries, those libraries are specified in the worker's OWD XML file, using the **`Libraries`** attribute. To make a declaration of such primitive libraries common to all workers in a library, or in a project, the **`HdlLibraries`** attribute can be set in the library's ***`<libname>`*`.xml`** file or in the project's **`Project.xml`** file. This attribute applies only to HDL workers. Lower level libraries must be built before building a worker which references them.

Similar to other authoring models (e.g. the RCC model), the build targets are specified by setting options to **`ocpidev build`** specifying targets for that model. For HDL, these target options are: **`--hdl-target`** or **`--hdl-platform`**. These options can be supplied multiple times for multiple targets/platforms. HDL build targets are discussed in detail in the next section.

The default platform for RCC workers is the development system itself (e.g. **`centos7`**). Unlike RCC workers, HDL assets have no inherent default target. However, a default value for HDL targets/platforms can be set in the **`HdlPlatforms`** and/or **`HdlTargets`** attributes in the library's ***`<libname>`*`.xml`** file, or in the project's **`Project.xml`** file.

For software workers, this is usually the end of the build process: deployable artifacts for these workers are created and ready for export and/or use in applications.

For HDL workers, it is different. FPGAs are generally not subject to dynamic, partial loading: the whole FPGA must be reloaded with a full "configuration bitstream". [OpenCPI does not support *partial reconfiguration* of FPGAs as of this document version]. As with any authoring model, primitives are built first. Then HDL workers are built and, for targets that are real FPGAs rather than simulators, synthesized. Finally, there are two additional steps in the build process in order to create the final, dynamically loadable configuration bitstream:

- Composing workers into an ***HDL assembly***.
- Finalizing the bitstream as an ***HDL container***.

This final step creates the deployable artifact usable for export and/or use in applications. These steps are defined in the sections below.

4.1 HDL Build Targets

Build targets specify the target device, family of devices, or platform for which the asset should be built (compiled, synthesized, place-and-routed, etc.) When building any level of modules for FPGAs, the build targets are specified via the **`--hdl-target`** or **`--hdl-platform`** options to the **`ocpidev`** command, or sometimes using the ***HdlTargets*** or ***HdlPlatforms*** attributes in the library or project-level XML files. The *targets* are chips or chip families, whereas the *platforms* are actual FPGAs on specific boards. HDL primitives, workers, and assemblies, are built for HDL targets, and HDL

platforms and containers (final bitstreams) are built for HDL platforms. These build targets are defined in a hierarchy with these levels:

Top level, vendor level: this level specifies vendors (Xilinx, Altera), as well as vendor-independent simulators (Modelsim). This enables HDL assets to be built for all Xilinx and Altera targets or built for Modelsim. This implies building for all lower level targets under these top-level labels. The value **`all`** specifies all top level supported 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 “virtex6” or “zynq” 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: specifies the exact part the design is targeted at, e.g. xcv5lx50t. This does not include package information but may include speed grades.

The following two XML attributes can further filter the targets that are built anywhere that HDL building takes place.

ExcludeTargets/ExcludePlatforms: these attributes specify 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 attribute specifies targets to be exclusively included, because it is known that only a limited set of targets should be built (e.g. a Xilinx coregen core specific to a particular family or part).

The **`--hdl-platform`** option for **`ocpidev`** (or the ***HdlPlatforms*** attribute in library or project XML) specifies HDL platforms to build (like Xilinx **`ml605`** and **`zed`**), 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 part family. Except for the final bitstream build, the HDL target(s) are implied by the specified HDL platform(s).

If no HDL target or HDL platform options for **`ocpidev`** are set, and no HDL target or HDL attributes are set in library or project XML files, the OCPI_HDL_PLATFORM environment variable can be set to an HDL platform. That will be used for all HDL builds.

In some synthesis cases, tools that target a *part family* actually target the *smallest* part in the family and try to limit use of some on-chip resources (e.g. DSP blocks) to the amount that exists on the smallest part. While this usually correctly generates a resulting file that can be used on any part in the family, it is not always desirable when the target *platform* in fact has a *larger* part.

Similarly, some tools that target a part family do *not* target the smallest part, so that the resulting design will not work on the smallest part.

To force a worker to be built for a specific part for an HDL target, you can set the **`ExactParts`** attribute in the worker's XML file. The value of this attribute is a list of

colon-separated pairs of ***<family>:<part>***, indicating the exact part to use when building for the mentioned part family. For example, this worker XML:

```
<HdlWorker ExactParts='zynq:xc7z020-1-clg484x
                       virtex6:xc6vlx240t-1-ff1156
                       stratix4:ep4sgx230k-c2-f40'/>
```

would choose the indicate parts when the HDL target of the build was **`zynq`**, **`virtex6`**, or **`stratix4`**.

Note the parts are all of the form ***<part>-<speed>-<package>*** even though various vendor tools have different part name formats for different lower level tools.

4.2 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 FPGA simulation. The following diagram shows the build flow (bottom to top) and hierarchy.

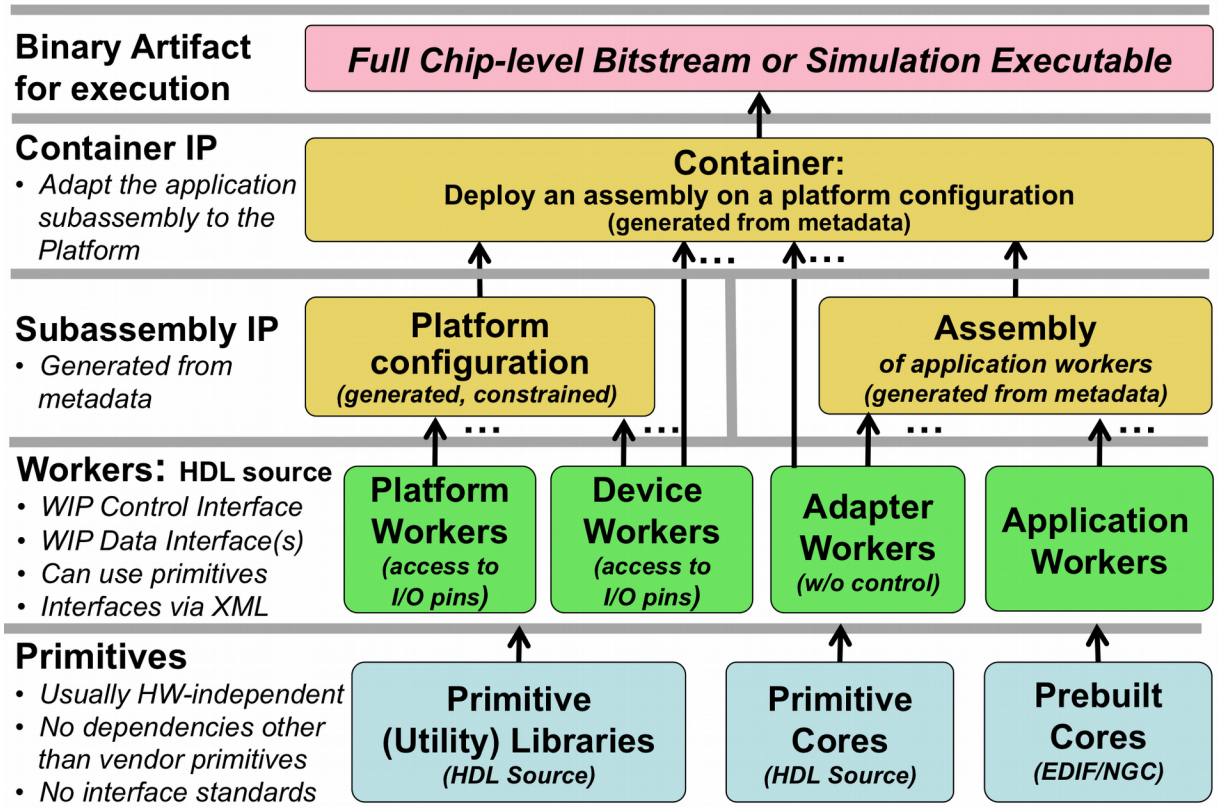

Figure 9: OpenCPI HDL Build Flow Layers

At the bottom layer (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 instantiated in workers. 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 creating such primitive libraries are in the ***OpenCPI HDL Platform Development*** document.

Above the ***primitives*** layer is the HDL ***worker*** layer, with workers 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 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 connectable in some way due to different interface choices in the OWD (e.g. width, stream-vs-message, clock domains). Adapter workers are normally inserted automatically as needed.

A platform worker is the special type of device worker that performs necessary platform-wide functions for the platform.

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

The ***platform configuration*** is automatically generated HDL source code that uses platform workers along with some device workers. It represents a platform configured with built-in support for some attached 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.

At the top layer, 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 HDL assembly to the available I/O paths and devices in the platform. When the deployment of the HDL assembly requires device workers that are not in the platform configuration, they are instanced in the container itself. The Linux kernel analogy is that these extra device workers are analogous to the dynamically loaded device drivers used to run the application.

Device workers can either be built into the platform configuration or instanced in the container. Platform configurations provide the way to share configurations of devices and device workers to avoid redundancy and duplications of such things in container XML.

The final design for the entire FPGA is the container logic. This hierarchy (except primitives) is shown in the following diagram.

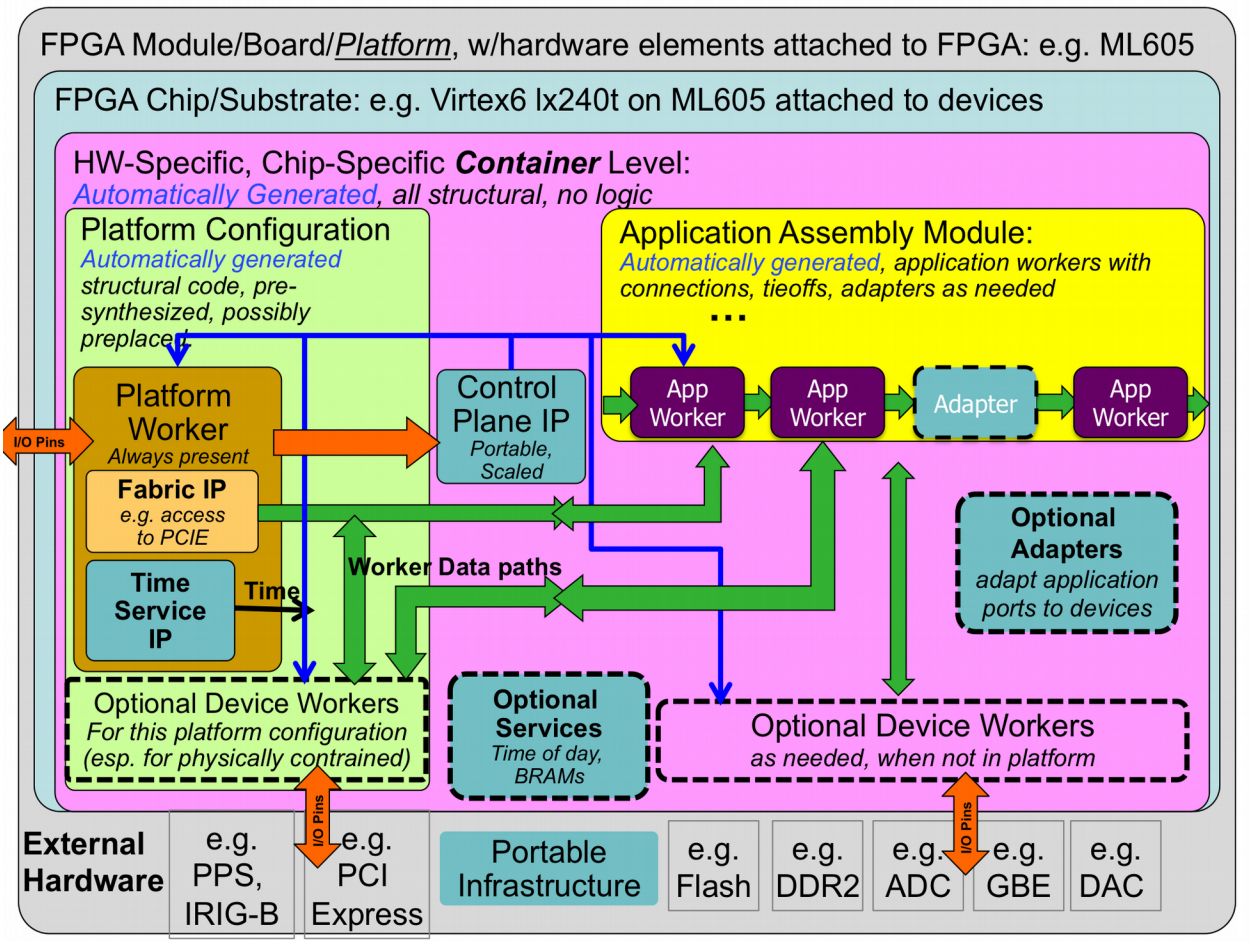

Figure 10: HDL Full FPGA Hierarchy

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

- Each worker in a component library is prebuilt (possibly using primitive libraries and cores)
- Assemblies are prebuilt from generated VHDL or Verilog code using the required worker cores
- Platform configurations are prebuilt from platform workers and device worker cores
- Container top levels are built from platform configurations and HDL 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 HDL 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 HDL platforms that act as test benches for assemblies. This is described in more detail below in the simulation section.

4.3 HDL Directory Structure

In OpenCPI projects, HDL workers are in component libraries with other non-HDL workers. Component libraries are thus heterogeneous, where different workers, possibly using different authoring models, may implement a common spec. A project may have a single component library called 'components', where worker directories are located. Alternatively, a project may have multiple uniquely named libraries under the **`components`** directory, each of which contains workers with some common theme. When projects have a single component library, it is in the **`components`** directory of the project. There can also be multiple separately named component libraries *under* the **`components`** directory. Additionally, the top-level **`hdl`** directory in a project contains the following directories as needed:

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

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

devices: This directory is a component library containing HDL device workers for devices that are potentially usable on different platforms. HDL device emulators and software proxies for some of the devices may also be in this component library.

platforms: This directory contains subdirectories for each platform implemented in the project. Platforms are 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. There may also be a subdirectory under the platform's directory, called **`devices`**, containing a library of HDL device workers, proxies and emulators specific to that platform.

cards: This directory contains HDL device workers (and their proxies and emulators) that are specific to cards, rather than those generally useful on different platforms and cards. It also contains specification files for cards.

Development for HDL devices, platforms, and cards is described in the ***OpenCPI Platform Development Guide***.

Application workers for all authoring models are found in component libraries in projects, some of which are part of OpenCPI.

The project directory hierarchy is shown in the following diagram. All directories are optional and are created as needed by the **`ocpidev`** tool described in the CDG.

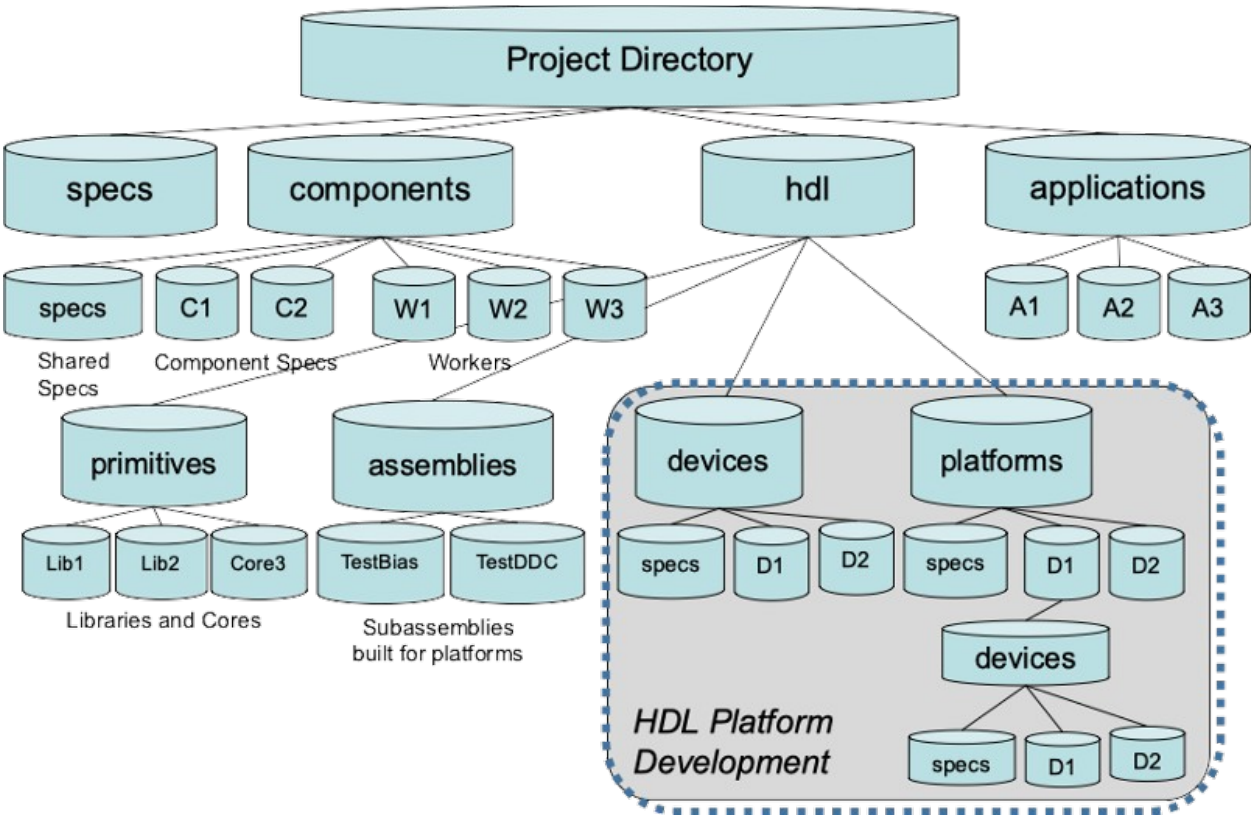

Figure 11: OpenCPI HDL Directory Structure

4.4 HDL Search Paths when Building

When building at any level of the HDL build hierarchy, the asset being built can depend on other assets, usually at lower levels of the hierarchy. HDL primitive libraries and cores can depend on other HDL primitive libraries. Workers can depend on primitives, specs, protocols and other workers. HDL assemblies always depend on application workers in libraries. The assets being depended on may be in the same library or same project, or in a different project.

In each case of a dependency, the underlying asset is found using search rules. The built-in search rules automatically find assets that are in the component library, when building workers in that library. They also automatically find assets that are in the current project, when building HDL primitives, workers, and assemblies in projects. Special action by the developer is only required when there are dependencies on assets outside the current library or project.

If assets in one project (A) depends on assets in another (B), that should be stated in the **`ProjectDependencies`** attribute in the project A's **`Project.xml`** file. If a special version of that second project (B) was needed temporarily, an environment setting would cause the special version to be searched first, shadowing the assets in the project (B) explicitly stated in **`ProjectDependencies`** of A. By "shadowing" we mean that the environment setting causes the search to look elsewhere *before* looking at the projects specified in the **`ProjectDependencies`** attribute.

Environment variables are normally used only to temporarily replace/shadow assets in the default search path. Dependencies are normally stated at the point of dependency and stored in the appropriate file that is part of that asset (e.g. **`Project.xml`** for project-level dependencies). Environment variables essentially override these stated dependencies.

For each of the search rules defined below, the following principles are applied:

- Search "closer" first, then "farther away" (e.g. in library, then in project, then other projects).
- Search within the project before searching outside the project.
- Search using the project path in the environment (**`OCPI_PROJECT_PATH`**) before using the explicit dependencies in the project's **`Project.xml`** file.
- Search the built-in **`ocpi.core`** project last.
- Search the directories specified in any path environment variables in the order they appear in the colon separated list.

4.4.1 Searching for HDL Primitives

When a worker depends on primitive libraries, it specifies this by declaring the library name in a list in the **`Libraries`** attribute in its OWD XML file. Similarly, when an HDL worker depends on primitive cores, it specifies this by putting the core name in a list in

the **Cores** attribute. When the worker is being built, these lists are used in the following way.

If the library or core name has slashes in it, it is treated as a pathname (absolute or relative) to the specific directory of the primitive library or core. Examples are:

```
Libraries=../../myprims  /home/colleague/hisprims/funprims
Cores=../../../ourcores/fft
```

Or for tools that require Cores to be explicitly mapped to HDL instances:

```
Cores=../../../ourcores/fft:fft_i
```

If a name in the primitive library or core list does *not* have a slash (the common case), it is found by searching in the following places, in the following order:

- The HDL primitive libraries or cores in the same project as the worker.
- The HDL primitive libraries/cores in the projects listed in the environment variable **OCPI_PROJECT_PATH**, which are colon separated and searched in order.
- The HDL primitive libraries or cores in the projects listed in the **ProjectDependencies** attribute in the project's **Project.xml** file.
- The HDL primitive libraries or cores in the **ocpi.core** project.

When it is convenient to put a list of primitive libraries in the library or project's XML file, (making them available to all workers in the library or project), the name of the attribute is specific to the authoring model. Thus to make a list of libraries available to all the HDL workers in a component library, you would put the following line in the *library's* XML file.

```
<library HdlLibraries='gprims'/>
```

Rarely, HDL Compilation tools may require that cores be explicitly mapped to HDL instances in a design. For such tools (*e.g.* Quartus Prime Pro Edition), the format **Cores=*<core-name-or-path>*:*<hdl-instance>*** can be used.

4.4.2 Searching for XML files (OCS, OPS) when Building Workers

As described in the CDG, all workers have an OWD, and all OWDs depend on a component spec, normally found in a separate OCS XML file. Furthermore, OCS files frequently depend on a separate OPS (protocol spec) file. It is also possible that an OWD could include an XML file to incorporate a list of properties defined elsewhere.

When looking for these XML files, when their name has no slashes, it is found by looking in the following places, in order:

- The worker's directory
- The **gen** subdirectory of the worker's directory (for files generated by tools)
- Directories specified in the space-separated list in the XML attribute **XmlIncludeDirs**
- The component library's export directory (**lib/hdl**) for referencing other HDL workers (e.g. for **slave** and **emulate** attributes)

- The component library's **specs** subdirectory
- The export directories (**lib/hdl**) of all component libraries in the **ComponentLibraries** attribute (see below) for referenced HDL workers (e.g. for **slave** and **emulate** attributes)
- The directories listed in the environment variable **OCPI_XML_INCLUDE_PATH**, which are colon separated and searched in order
- The **specs** subdirectories of all projects in the environment variable **OCPI_PROJECT_PATH**, which are colon-separated and searched in order
- The **specs** subdirectories of the projects listed in the **ProjectDependencies** attribute in the project's **Project.xml**
- The **specs** directory of the built-in **ocpi.core** project.

4.4.3 Searching for Workers in Component Libraries

The XML **ComponentLibraries** attribute specifies a list of places to look when searching for workers. The most common use case for **ComponentLibraries** is for creating HDL assemblies, where the workers specified in the assembly must be found by searching for them in component libraries. Other uses include device workers, platform workers and platform configurations, as described in ***HDL Platform Development***.

While it might seem counter-intuitive for a worker *inside* a component library to depend on other component libraries, there are three cases where this occurs:

- A worker's OWD depends on specs (OCS and/or OPS) in another library.
- A worker is a proxy for a worker defined in another library (HDL workers cannot act as proxies).
- A worker is a device emulator for a device worker defined in another library.
- A worker is a subdevice which supports a device worker defined in another library.

In all these cases, a worker or a component library might define the **ComponentLibraries** XML attribute to specify this dependence.

To find workers, the search first looks in the library that the worker is already a part of. After this, the **ComponentLibraries** attribute is used, which holds a list of component libraries to search.

If the component library name in **ComponentLibraries** has slashes in it, it is treated as a path name (absolute or relative), to the specific directory of the component library. If a name in the list does *not* have a slash, the component library is found by looking in the following places, in order:

- The other component libraries in the same project.
- The component libraries exported by the projects listed in the environment variable **OCPI_PROJECT_PATH**, which are colon separated and searched in order.

- The component libraries of the projects listed in the **`ProjectDependencies`** attribute in the project's **`Project.xml`** file.
- The component library in the **`ocpi.core`** project.

In all cases, for a worker to be found, it must have been built, or for HDL workers, their project must have been built for RCC workers (using the **`--rcc`** option to the **`ocpidev`** build command in the project). Building a project's RCC workers has a side effect of making all of its worker (RCC, HDL etc.) visible to other workers and projects even though they are not actually built. This happens automatically when a platform is installed using the **`ocpiadmin install platform`** command.

5 HDL Primitives

HDL Primitives are HDL assets that are lower level than workers and may be used as building blocks for workers. HDL primitives can either be ***libraries*** or ***cores***.

HDL primitives are useful for HDL workers when there is lower level code that is reused or shared in different workers. Using HDL primitives is also useful when there are non-OpenCPI code modules that are imported and should be left untouched in order to remain useful outside of OpenCPI. The use of HDL primitives is *not* required for HDL workers. HDL primitives cannot have the same name as a worker.

When lower level code modules and files are used in only a single worker, there is no need for a primitive library: such files can be simply put in the worker's directory and added to the **`SourceFiles`** attribute in the worker's OWD XML file. Such files are built before the worker source files so that they can easily be referenced without forward declarations (e.g. **`component`** declarations in VHDL).

An ***HDL Primitive Library*** is a collection of modules compiled from source code that can be referenced in HDL worker code.

An ***HDL Primitive Core*** is a single low level module that may be:

- Built and/or synthesized from source code
- Imported as presynthesized and possibly encrypted, from a third party.
- Generated by tools such as Xilinx CoreGen or Altera MegaWizard.

An HDL worker must declare which primitive cores it requires (and instantiates).

The built and exported library or core can be referenced by workers simply by including the following attributes in the HDL worker OWD XML file (or primitive's XML file):

```
Libraries='myutils'
```

or

```
Cores='mycore'
```

When the worker source code instantiates a primitive core or a module from a primitive library, no further action needs to be taken other than including the attribute above in the HDL worker's OWD (or once for all workers in a component library in the component library's **`<libname>.xml`** file). In particular, no other "black box" module or VHDL component declaration needs to be created by the worker. In VHDL, the library and uses are used in the source code, e.g.:

```
library xyz; use xyz.all;
...
my_fifo_inst: xyz_pkg.fifo
```

The **`HdlLibraries`** attribute can be set in the library's **`<libname>.xml`** file or the project's **`Project.xml`** file to make HDL primitive libraries available to all HDL workers in the library or project. The built-in **`ocpi.core`** project includes several HDL primitive libraries, and some are always available for use by all workers, even when the **`Libraries`** attributes is not set.

5.1 Naming Rules for HDL Primitives

Due to the constraints of HDL languages and tools, primitives (libraries or cores) are essentially in a single global namespace. When an HDL primitive is created in a project, it is still in the single global namespace. Thus using the primitive from another project uses search rules to find it, but the creator or user of a primitive has no way of knowing whether the name they use will conflict with other primitives that happen to have the same name, in other projects.

To provide more certainty and uniqueness when naming a primitive, the **`namespace`** attribute in the primitive's XML file can be set to **`qualified`**, e.g.:

```
<HdlLibrary namespace='qualified'/>
```

This “qualifies” the primitive's name with the package-ID of the project to make the name globally unique, since the project's package-ID must already be unique. When using a primitive, its name is used in two places: the XML file of the asset using it and the HDL language source code of the asset using it. So if the **`xyz`** primitive in the **`com.cnn.proj1`** project has this **`namespace`** attribute set to **`qualified`**, an asset using it would have this attribute, e.g.:

```
<HdlWorker libraries='com.cnn.proj1.xyz'/>
```

And in the corresponding VHDL source file you would use, e.g.:

```
library com_cnn_proj1_xyz; use com_cnn_proj1_xyz.all;
...
my_module_inst: xyz_pkg.my_module
```

If a primitive's name is qualified like this, even uses within the same project must use the qualified names in XML files and source code.

5.2 HDL Primitive Libraries

Primitive libraries are normally created using the **ocpidev** tool in a project, e.g.:

```
ocpidev create hdl primitive library myprims
```

This creates a directory for the primitive library in the project, in the directory **hdl/primitives**, with two files:

- The primitive library's XML file, **myprims.xml**
- A VHDL file, **myprims_pkg.vhd**, containing the libraries package declaration.

The **ocpidev create** command can also be issued in the **hdl/primitives** directory itself.

Primitive libraries may be written in VHDL or Verilog, but there are specific rules to follow in order for the library to be usable with all supported tools, and from VHDL or Verilog. Primitive libraries can depend on other primitive libraries, and this must be indicated by setting the **Libraries** attribute in the primitive's XML file. Circular dependencies among primitive libraries are not supported. Some internal OpenCPI primitive libraries are always available to other primitive libraries libraries. This is suppressed if the **NoLibraries** attribute is set non-empty in the primitive library's XML file.

5.2.1 Source Files in Primitive Libraries

If there are no ordering dependencies between source files, just creating or copying source files into the directory will cause them to be built there, together as the library. Thus without mentioning source file names, all source files in the top level directory of the primitive library will be built and included in the library.

The default build order for the source files in a primitive library directory is to first build any ***_pkg.vhd** and ***_body.vhd** files (see below) and then build all other source files (***.vhd** and ***.v**). There are several conditions where all source files must be explicitly mentioned in the **SourceFiles** attribute in the library's XML file. These are:

- There are ordering dependencies between source files (other than dependencies on the ***<pkg>*_pkg.vhd** files which are aways compiled first).
- Some source files are not in the top level directory of the library *and* are not in target-specific subdirectories for shadowing purposes (see Target-Specific Modules below).
- Some source files that are in the library's directory should not be built into the library (i.e. extraneous unbuilt source files)

It is recommend that source files *not* be placed in subdirectories *unless they are there as target-specific modules*. The reasons this is not recommended are:

- there is then a potential name collision between the names of the subdirectories and the names of the HDL targets, tools, and vendor names used for target-specific modules

- all the files must be listed in the SourceFiles attribute, which is otherwise usually unnecessary

All modules in a library intended to be used from outside the library must be in separate source files with the name of the file matching the name of the module, including case (before the language suffix). While not strictly required, this practice is also recommended for modules instantiated by other modules in the same library.

5.2.2 Package Declarations for Primitive Libraries

The library must include a VHDL file **`<libname>_pkg.vhd`**, containing component and data type declarations for *all* modules externally referenced (from outside the library). Even if the library has Verilog source code modules, the **`<libname>_pkg.vhd`** VHDL file with component declarations must be present for all of the Verilog modules in the library that are usable from outside the library.

The VHDL package name in the **`<libname>_pkg.vhd`** file should normally be the same as the library's name with a **`_pkg`** suffix. Thus the source file name that defines the package has the same name as the package defined in it. There can be multiple **`<pkg>_pkg.vhd`** files in a library if multiple packages are required. Finally, if the packages have package bodies in separate files, those files should be named **`<pkg>_body.vhd`**.

When modules in the library are instantiated by other modules in the library, but *not* intended for external usage by code *outside* the library, they must still have component declarations, but they can be placed in a different package to keep them separate from those in the **`<libname>_pkg.vhd`** file that *are* intended for external use. Perhaps such a package would be called **`<libname>_internal_pkg.vhd`**.

For example, consider the source file implementing module **`outer`**, in the file **`outer.vhd`**, in the **`mylib`** primitive library.

```
entity outer is
  port (clk : in std_logic;...);
end entity outer;
architecture rtl of outer is begin
  fifo: work.mylib_pkg.myownfifo port map(....);
end rtl;
```

For this module to be usable from outside the library, the component declaration must be in the library's package file, **`mylib_pkg.vhd`**:

```
package mylib_pkg is
  component outer is
    port (clk : in std_logic; ...);
  end component outer;
end package mylib_pkg;
```

In the example above, the **`outer`** module uses another module, **`myownfifo`**, also from this library. Any module in the library referenced between files, must also have a component declaration in *some* package file, so the package file might in fact be:

```
package mylib_pkg is
  component outer is
    port (clk : in std_logic; ...);
  end component outer;
  component myownfifo is
    port (...);
  end component myownfifo;
end package mylib_pkg;
```

Alternatively an internal package file could be defined, in **mylib_internal_pkg.vhd**, containing just the internally-used module:

```
package mylib_internal_pkg is
  component myownfifo is
    port (...);
  end component myownfifo;
end package mylib_internal_pkg;
```

5.2.3 Instantiating Modules in Primitive Libraries

Modules referenced from VHDL must use the component instantiation syntax. The instantiation does that in the **outer** entity example above.

When a VHDL worker or code in another primitive library is written to *use* a module in a primitive library, it must include a line to access the library. For a primitive library **mylib**, the calling module file might contain the lines.

```
library mylib; use mylib.mylib_pkg.all;
...
inst1 : outer ...
```

When it is desirable to use better namespace control (e.g. if there is an **outer** module from two different libraries), it may be preferable to use:

```
library mylib; use mylib.all;
library otherlib; use otherlib.all
...
inst1 : mylib_pkg.outer ...
inst2 : otherlib_pkg.outer
```

Finally, when instantiating a module from within the same library, the work library can be used and no library declaration is required, e.g.:

```
inst1 : work.mylib_pkg.outer ...
```

5.2.4 Providing Target-specific or Vendor-specific Versions of Primitive Modules

The modules in a primitive library are normally each in their own files in the top level directory with the file name being the same as the module name. Sometimes it is useful to have special versions of source code for a module that is specific to a particular part family, vendor or tool. This allows alternative source code files for a module that uses vendor-specific primitives or are written in a way to infer vendor-specific hardware features (e.g. BRAMs, DSPs, IO features).

A primitive library is normally built using each source file indicated in the **SourceFiles** attribute, or if that attribute is not specified, each source file found in the primitive library's directory. However, when building for a specific target, it first looks for a file of the same name in a subdirectory with the name of the target being built (e.g. **zynq** or **isim**). If it finds that file, it uses it *instead* of the file in the top level directory. If there is no such file in a target-specific directory, it next looks in a vendor-specific directory (e.g. **xilinx** or **altera**). If the file does not exist in either target-specific or vendor-specific subdirectories, then the (default, generic) file in the top level directory is used.

Note that simulator targets also have vendors. Thus if the target is **isim** or **xsim** (both **xilinx** simulators), and there is a module file in the **xilinx** subdirectory, that file will be used for those simulators in preference to the default file at the top level.

The files in target-specific or vendor-specific directories should never be mentioned in the **SourceFiles** attribute. The file in the top level directory serves as the default implementation of the module, which will be ignored (shadowed) in preference to target-specific or vendor-specific versions when they exist.

Two examples of this feature are below.

5.2.4.1 Shadowing Example for Correct Inference of Resources

A synchronous ROM module is defined to take CLK and ADDR as input, and provides output DATA synchronously after two clock edges, based on the address valid at that time, with no overlap of access cycles. A simply default (Verilog) implementation, in **ROM.v**, assuming single-cycle access times, would simply ignore the clock and drive data continuously:

```
assign DATA = ROM[ADDR];
```

This is correct functionality, perhaps suitable for simulation, but neither Xilinx nor Altera tools will synthesize this into their respective block ram resources. For Xilinx, the output data must be registered for the synthesis tool to infer/use a block ram, thus the Xilinx code should be:

```
always @(posedge CLK) begin
  DO_R <= ROM[ADDR];
end
assign DO = DO_R;
```

For Altera, the input address must also be registered, thus the code should be:

```
always @(posedge CLK) begin
  ADDR_R <= ADDR;
  DO_R <= ROM[ADDR_R];
end
assign DO = DO_R;
```

The contract of the module allows for all three implementations:

- **ROM.v** for a simple default implementation suitable for simulation.
- **xilinx/ROM.v** to correctly infer BRAMs using ISE or Vivado tools.
- **altera/ROM.v** to correctly infer BRAMs using Quartus tools.

5.2.4.2 Example of Shadowing using Vendor-specific Primitives

A clock buffer is an important resource for clock distribution, and it is desirable to write higher level code that uses portable primitives to instance one. This example takes CLK as input and produces CLK_BUFFERED as output. The easiest way to use clock buffer resources is to directly instance the vendor-specific primitives. A generic clock buffer default implementation, in **clkbuffer.vhd**, would be:

```
clk_buffered <= clk;
```

This is functionally correct (ignoring delta cycle issues in simulators), but does not take advantage of clock buffering or routing features unless recognized automatically due to fanout etc. For Xilinx, the explicit implementation would use a BUFG primitive, e.g.:

```
buf : BUFG port map(I => clk, O => clk_buffered);
```

For Altera, an attribute declaration might be sufficient for this purpose:

```
attribute altera_attribute of clk_buffered :
  signal is "-name GLOBAL_SIGNAL REGIONAL_CLOCK";
```

The module would have three implementations:

- **CLK_BUFFERED.vhd** for a default/simple/portable implementation for simulation
- **xilinx/CLK_BUFFERED.vhd** to instance BUFG primitive for Xilinx
- **altera/CLK_BUFFERED.vhd** to use an explicit Altera attributes

5.2.5 Exporting and Using the Results of Building HDL Primitive Libraries

When HDL primitive libraries are built, their immediate per-target results are in target-specific subdirectories (**target-*<hdl-target>***) whose format varies depending on the tools used for that target. The log output of the tools is usually collected in a ***<libname>-<tool>*.out** file in the target directory, which can be examined when errors occur or to examine warnings, etc.

HDL Primitives are always built as a group under the **hdl/primitives** directory in a project, with its own XML file called **primitives.xml**, which is automatically created whenever primitives are created using **ocpidev** in a project.

Building a primitive library in its own directory is useful for rapidly getting to a clean build across all relevant targets. When primitives are built, the exportable results are placed in **hdl/primitives/lib**, much like the **lib** subdirectory of component libraries. The files in that directory are automatically used as the project's exported primitive libraries and cores when a project is built.

When developing a primitive library it is highly recommended to at least occasionally build for all available tools, both for synthesis to hardware and for simulation. This ensures that the code is nominally portable.

5.3 HDL Primitive Cores

Making a prebuilt/presynthesized core available for use by workers is similar to creating a primitive library from source files. The **ocpidev** command, for creating **mycore**, is:

```
ocpidev create hdl primitive core mycore
```

This creates a directory for the primitive core in the project, in the directory **hdl/primitives**, with an XML file **mycore.xml** in that directory for any required attributes (all are optional).

This command can also be issued in the **hdl/primitives** directory itself.

Whereas an HDL primitive library is built as a collection of source modules that are not fully elaborated or synthesized, HDL primitive cores are built into a *single* module that may have no source files other than those that define the interface.

The files used to build the core can be a mix of source and prebuilt files. There may be presynthesized core files (e.g. Xilinx **.ngc** or Altera **.qxp**), or source files. There may be presynthesized files for some targets and source files for other targets. Any targets that do not have prebuilt cores will use the source files.

When the core supports instantiation from Verilog, there must be a "black box" empty module definition file **<corename>_bb.v**. When the core supports instantiation from VHDL, it must have a package file **<corename>_pkg.vhd** containing a component definition in a package named the same as the core name.

There are two special additional optional XML attributes that apply to HDL primitive cores: **Top** and **PrebuiltCore**.

Top is the attribute that specifies the top module name of a primitive core when it is different from the core name used when it was created with **ocpidev**. Normally the name of the primitive core is the same as the top level module name and this attribute is unnecessary. In some cases the core name is more descriptive and useful, while the top module name is predetermined for some other reason. An example is a core name of **ddc_4ch_v5**, which might be a core for a 4 channel DDC generated specifically for virtex5. The actual generated core from Xilinx CoreGen has the file and module name **duc_ddc_compiler_v1_0**. Thus the **ddc_4ch_v5.xml** file, in the **ddc_4ch_v5** directory would contain:

```
<HdlCore Top='duc_ddc_compiler_v1_0'/>
```

The **PrebuiltCore** attribute is used to specify a file that is a core that is not in source code, but is generated by some other tool and copied into the directory for the HDL primitive core. If this attribute is not set, source files are expected. Here is an example:

```
<HdlCore PreBuiltCore='mycore.ngc'
         OnlyTargets='xcv6lx240t'/>
```

In cases where there are different prebuilt core files for different targets, the files would be in target-specific directories, similar to the shadowing of primitive libraries described above. [this feature not available yet].

The directory for the primitive core might contain these files:

```
fft4k.xml
fft4k_bb.v
fft4k_pkg.vhd
zynq/fft4k.ngc
stratix4/fft4k.qxp
fft4k.v # usable for simulation
```

If the source files were specific to the Xilinx Isim simulator, then the core should be restricted to building only for the zynq, stratix4, and Isim targets using, in the **`fft4k.xml`** file:

```
<HdlCore OnlyTargets='isim stratix4 zynq'/>
```

Primitive cores can depend on other primitive cores or libraries, and this must be indicated by setting the **`Libraries`** or **`Cores`** attribute in the XML file, in the same way as it may be set in a worker's OWD XML file. Circular dependencies are not supported.

6 HDL Assemblies for Creating Bitstreams/Executables

An HDL assembly is a fixed composition of HDL application workers that can act as a whole or part of a heterogeneous OpenCPI application. It is built to create different FPGA bitstreams for different FPGA platforms. It will execute as part of some OpenCPI application with its workers being a subset of the workers in the application.

This section describes how to define and build (synthesize) these assemblies and to ultimately turn them into bitstreams. When the target HDL platform is a simulator, we use the term ***executable*** while when the target is an actual physical FPGA, we use the term ***bitstream***. In both cases there is a single resulting standalone OpenCPI artifact file that is ready for loading and execution.

Creating an artifact from an assemblies is basically: *implementing the assembly for a specific platform with a specific configuration of devices.*. More specifically, it is combining the defined assembly of HDL workers with a ***platform configuration*** to create an ***HDL container*** that is then transformed into a bitstream/ executable. This was shown above in the build hierarchy in the section ***HDL Build Hierarchy***.

The container is the outer module that contains the HDL assembly as well as the platform support modules in the platform configuration.

Assuming 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, specifying application workers and connections between them.
2. Select the platform configuration that the assembly will be implemented 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 "defining the container".
4. Run **`ocpidev build`** to generate the bitstream.

In most cases, steps #2 and #3 above are automatic and use defaults. A quick example, typically used for unit testing would be an assembly file containing a single worker:

```
<HdlAssembly>
  <Instance Worker="bias_vhdl" externals='true'/>
</HdlAssembly>
```

This above example specifies that the assembly consists of a single worker whose ports become the external ports of the assembly. HDL assemblies are created using the **`ocpidev`** tool, e.g. for creating the **`myassy`** assembly:

```
ocpidev create hdl assembly myassy
```

This creates a directory in the project's **`hdl/assemblies`** directory, with the name of the assembly (**`myassy`**), containing an initial HDL assembly XML file **`myassy.xml`**. The name of the XML file is simply the name of the assembly, with the ".xml" file extension. This command can be issued in the project directory or the **`hdl/assemblies`** directory

of a project. When the command is issued for the first time in a project, the **hdl/assemblies** directory itself will be created, with its own XML file: **assemblies.xml**. Thus after creating the first HDL assembly (called **first**) in a project, and building it for the **zed** platform, the directory structure would be:

```
hdl/assemblies/                    # Directory for all assemblies
  assemblies.xml                   # XML for all assemblies
  first/                           # Directory for first assembly
    first.xml                      # XML file for the first assembly
    -- from here down is results from building for zed --
    gen/xyz—assy.v                 # generated assy structural hdl
        -- other generated files --
    target-zynq/                   # synthesized assembly build dir
    container-first_zed_base/      # container dir for first on zed
      gen/first_zed_base-assy.vhd  # generated top level vhdl
          -- other generated files --
      target-zynq/                 # final bitstream build directory
        first_zed_base.bitz        # final bitstream/executable file
```

The actual steps taken by the OpenCPI scripts and tools, to create a bitstream or executable from an assembly, 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 platform configuration, as well as any necessary additional device workers (that are not already in the platform configuration).
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.).

The assembly, platform configuration and container are all in the same namespace and thus must have distinct names. This constraint will be removed in a future release.

HDL adapters will automatically inserted as needed in both steps 1 and 3.

This process is run for all platforms specified with the **--hdl-platform** option to **ocpidev build**.

This results in an artifact file, with the suffix **.bitz**, which can be used at runtime when executing OpenCPI applications. This file is based on the vendor-tool-specific output files like **.bit** for Xilinx and **.sof** for Altera, which are also in the container/bitstream build directory. Those files are not used by OpenCPI after the **.bitz** file is created.

6.1 The HDL Assembly XML file

The assembly is described in an XML file containing an **`HdlAssembly`** top-level XML element, which contains ***worker instances***, ***property/parameter settings***, ***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 Development Guide***).

The XML file is generated initially when the **`ocpidev create hdl assembly`** command is issued.

The worker instances (**`instance`** subelements of the **`HdlAssembly`**) reference HDL workers in some component library, and optionally assign names to each instance. The **`worker`** attribute is the worker's OWD name (without directory or model suffix), and the optional **`name`** attribute is the instance name. Worker names can include package prefixes to select workers in different libraries or projects. When not specified, instance names are either the same as the worker name, without any package prefix (when there is only one instance of that worker 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 can use **`connection`** XML elements or a more compact shorthand described next. The **`connection`** elements define connections among worker data ports. 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. The example above can be written as:

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

Furthermore, when this shortcut is used, you can specify the "from" port using the **`from`** attribute, and the "to" port using the **`to`** attribute. If these instances 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, where the data is flowing into or out of the assembly itself, 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>
```

To specify that *all* unconnected ports of a worker should be made external ports of the assembly, you can use the **`externals`** boolean attribute. E.g., if the assembly is in fact a single worker where both **`in`** and **`out`** ports should be external, you would only need:

```
<HdlAssembly>
  <Instance Worker="process" externals='true'/>
</HdlAssembly>
```

This is a common use case for unit test assemblies used to test single HDL workers.

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 memory. The **`split`** worker takes data from its **`in`** interface and replicates it to both its **`out0`** interface as well as its **`out1`** interface. The HDL assembly has 4 external ports (ADC, SWIN, SWOUT, DAC).

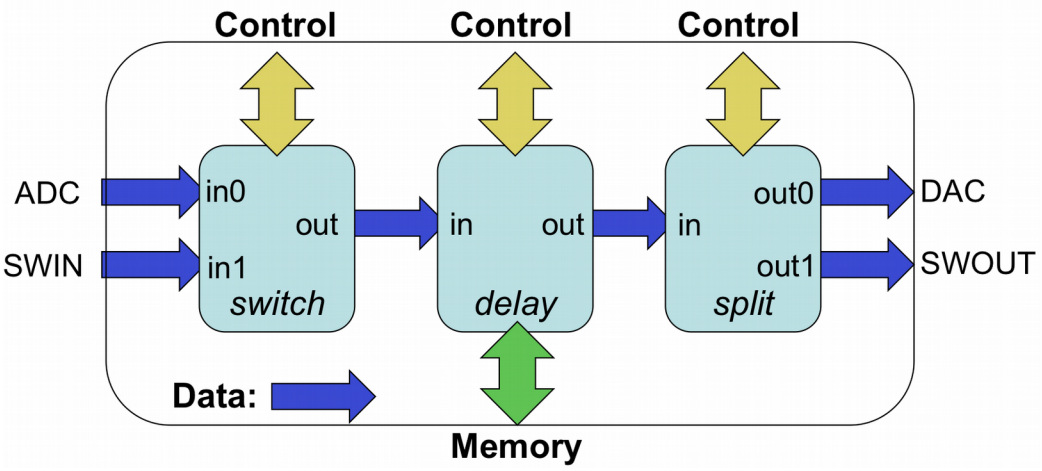

Figure 4: Example HDL Assembly

Given that these three workers are already in a component library, the XML description of the example is below. It uses the **`external`** elements rather than simply adding the **`externals='true'`** to the switch and split workers because the external ports of the assembly have different names than the corresponding worker ports.

```
<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: Example HDL Assembly XML

6.2 Assembly XML Attributes for Building the Assembly

The assembly's XML file can specify in top-level attributes various options when the assembly is built. A common attribute, **ComponentLibraries**, is used to specify where the workers should be found to make the assembly. An assembly XML file for the above application might be:

```
<HdlAssembly ComponentLibraries='testcomps'>
  --- instances, connections, externals ---
</HdlAssembly>
```

In this case the assembly's workers will be found in the indicated component library, or other libraries in the search path, described earlier in the ***HDL Search Paths when Building*** section. This attribute can also be set in the **hdl/assemblies** directory XML file **assemblies.xml** to apply to all the assemblies in that directory (see just below), or in the project's **Project.xml** file.

While HDL primitives and workers are built for FPGA part families, indicated using **--hdl-target** options to **ocpidev build**, HDL assemblies are built for specific HDL *platforms*, using **--hdl-platform** options. The platform includes specific devices attached to the FPGA as well as other specific FPGA attributes. In general, anywhere that **--hdl-target** is used, **--hdl-platform** can be used, since a platform implies a target. The attributes suitable for HDL assembly XML files are:

Table 20: HDL Assembly XML File Attributes

Attribute Name in HDL Assembly XML file	Usable as default in hdl/assemblies assemblies.xml file?	Description
ComponentLibraries	Y	A list of component libraries to search for the workers in the HDL assembly (in order)
OnlyPlatforms	Y	An exclusive list of platforms for which this assembly (and default containers) should be built.
ExcludePlatforms	Y	A list of platforms for this the HDL assembly should *not* be built.
Containers	N	A list of containers to build for this assembly (name of local XML files for containers, without suffix)

The **hdl/assemblies** directory of a project acts as a sort of library of assemblies. This XML file **assemblies.xml** in the **hdl/assemblies** directory is automatically created when the first HDL assembly is created, as:

```
<assemblies/>
```

If the assemblies in this directory should not all be built, or should be built in a particular order, the **Assemblies** attribute can be set to a list of assemblies to be built in the specified order.

6.3 Specifying the Containers that Implement the Assembly on Platforms

The container is the top-level module and implements the assembly on the platform. Separating the assembly from the container in this way keeps the assembly portable and hardware-independent (assuming the workers are). The assembly's external input and output connections are unspecified until it is implemented in a container on the platform. By itself, the HDL assembly is usable in a simulation testbench, when it is built for a simulation platform.

In an assembly's XML file, 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. PCI Express or SoC AXI buses). Thus the implicit default container specification is to connect every external port of the assembly such that it connects external to the HDL platform, to connect to workers running on containers on other platforms. Using the previous assembly example, the default container would implement the assembly with all 4 ports (ADC, SWIN, DAC, SWOUT) connected to the platform's interconnect.

The optional **`DefaultContainers`** attribute 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 attribute is defined as empty:

```
DefaultContainers=''
```

no default containers are built for this assembly. If the **`DefaultContainers`** attribute is not set at all (not mentioned in the assembly's XML file, the default situation), 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).

Based on these defaults, if nothing is said at all about containers in the XML file, default container bitstreams will be built for whatever platforms are mentioned in the **`--hdl-platform`** options to **`ocpidev build.`** In many cases this default case is all that is required (no container attributes at all).

Default containers are used to connect the external ports of the assembly to 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 its own XML file. The **`Containers`** attribute specifies a list of containers (with container XML files) that should be built in addition to those indicated by the **`DefaultContainers`** attribute (or absence thereof). The **`Containers`** attribute does not suppress the building of the default containers.

This example assembly XML file relies only on the component libraries specified in search paths, or in the **`ComponentLibraries`** attribute setting in the **`hdl/assemblies/assemblies.xml`** file, and builds default containers for whatever platform is specified using the **`--hdl-platform`** option. It is essentially what is generated automatically by the **`ocpidev`** command.

```
<HdlAssembly>
  -- instances, connections, externals --
</HdlAssembly>
```

An XML file that builds a default container on the **ml605** base platform configuration 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:

```
<HdlAssembly DefaultContainers='ml605 alst4/lime_adc'
             Containers='in_2_adc'>
  -- instances, connections, externals --
</HdlAssembly>
```

6.4 HDL Container XML files

A container XML file is required to connect to multiple interconnects (or not the first one) or to make connections to local devices. The top-level element of the container XML file is the **HdlContainer** element. It may have these attributes: **platform**, **config**, **constraints**, **onlyPlatforms**, **excludePlatforms**.

The first three provide specific information as to which platform, platform configuration, and FPGA tools constraints file should be used for this container. These are common when a container is indeed specifically designed for a particular platform and configuration.

The **onlyPlatforms** and **excludePlatforms**, allow a container to be (re)used for multiple platforms by specifying which platforms are suitable for the container. Wildcards can be used in these platforms, which normally specify all simulator platforms using the value: ***sim**.

6.4.1 HDL Container XML Top Level Attributes

The top level of the HDL Container XML can have the XML attributes in the table below. All are optional, and all are of string type.

Table 21: HdlContainer Attributes

HdlContainer Attribute Name	Description
platform	The platform to use for this container. Used when the container is specific to one platform.
config	The platform configuration to use, possibly for more than one platform if multiple platforms have the same platform configuration name (e.g. the default, or **base** configuration).
constraints	The constraints file to use, possibly for more than one platform if multiple platforms have the same constraints file name (even with different suffixes). Suffixes are optional. Typically placed in the assembly's directory.
onlyPlatforms	A comma-separated list of platforms or wildcard strings to specify all the platforms this container could be built for. E.g. ***sim** for all simulator platforms.
excludePlatforms	A comma-separated list of platforms or wildcard strings to specify all the platforms this container *should not* be built for. E.g. ***sim** for no simulator platforms.

*6.4.2 HDL Container XML **config** attribute*

The **config** attribute specifies which platform configuration should be used when building this container. The platform configuration usually specifies which devices should be present and which constraints files (and parameters for them) to use. A platform configuration is needed when the default one (called **base**, which has no devices) is not appropriate. Containers should use (and share) platform configurations

whenever possible (and not indicate constraints files directly) since this reduces duplications and maintenance when the devices and constraints are replicated unnecessarily in different container XML files.

*6.4.3 HDL Container XML **constraints** attribute*

The **constraints** attribute indicates a constraints file to be used when building the container (bitstream). Constraints files are typically associated with platform configurations and *not* containers, but this attribute allows overriding whatever constraints file is already associated with the indicated platform configuration (via the **config** attribute). When no constraints file is indicated here or in the platform configuration, the default one is used, whose name is the platform name (followed by the appropriate suffix. e.g. **.xdc**).

The constraints attribute can apply to multiple platforms, and since it does not need a suffix (the suffix is implied by the tool needed for the platform), it can apply to different platforms that use different tools. An example, showing that the same constraints file being used for different platforms with different tools and different suffixes is:

```
<HdlContainer onlyPlatforms='zed,zed_ise' constraints='myconstraints'/>
```

For zed (using the Vivado tool set) the file would be **myconstraints.xdc**, while for **zed_ise** (using the ISE tool set) the file would be **myconstraints.ucf**.

As with the **constraints** attribute of platform configurations, the file name can be followed by parameters supplied to the constraints file using the “URL query” syntax where a **?** character separates the file name from ***<name>=<value>*** parameters. The **&** or ***;*** characters can be used between ***<name>=<value>*** parameters, although the **&** is less convenient since it must be escaped as **&** in XML. An example is:

```
<HdlContainer constraints='my-constraints?rx=1;mode=5'>
```

The parameters are set as global variables that can be accessed in the constraints file itself.

*6.4.4 HDL Container XML **connection** Child Element*

These **connection** elements specify connections among external ports of the assembly, devices on the platform, and interconnects of the platform. Connection elements have these attributes:

- **external**: specify an external port of the assembly
- **device**: specify a device on the platform or on a card
- **interconnect**: specify an interconnect on the platform, or ***** for the only one
- **port**: specify the device's port (required when device has more than one data port)
- **otherdevice**: specify a second device for device-to-device connections
- **otherport**: specify the otherdevice's port for device-to-device connections

- **card**: specify a type of card that a device is on (required if device is not part of the platform, i.e. not defined in the platform's XML)
- **slot**: specify the slot that a card is plugged into for a device (required when the card is supported in more than one of the platform's slots)

Using these attributes you can specify connections between:

- external and interconnect
- external and device
- interconnect and device
- device and otherdevice

Device-to-device connections are currently only supported when neither is on a card, or both are on the same card.

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 FPGAs or software containers, you would have:

```
<HdlContainer platform='alst4' config='alst4_conf1'>
   <connection external='in' device='adc'/>
   <connection external='out' interconnect='pcie'/>
</HdlContainer>
```

The connection to the **adc** device will be resolved in one of two ways:

- if the **adc** device was already included in the **alst4_conf1** configuration of the **alst4** platform, then that **adc** device instance will be used
- If not, then the **adc** device logic would be instantiated in the container itself, and used there

6.4.5 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 for a given worker and thus not found in OCS files. I.e., the worker declares what services it needs for its particular implementation. 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, but not for data. When generating a container, all services required by the assembly, as well as any services required by device workers instanced in the container, must be satisfied by instancing the appropriate service modules in the generated container code.

Time service requirements (based on **timeinterface** elements in the worker's OWD) are satisfied by:

- instancing a ***time client*** module for each worker that needs “time of day”, and
- 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, as specified by the **`timeinterface`** element in the worker's OWD.

Memory service requirements [which are currently not supported 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. Note that such memory services are unrelated to data message buffering used to connect workers together or connect them to devices and interconnects.

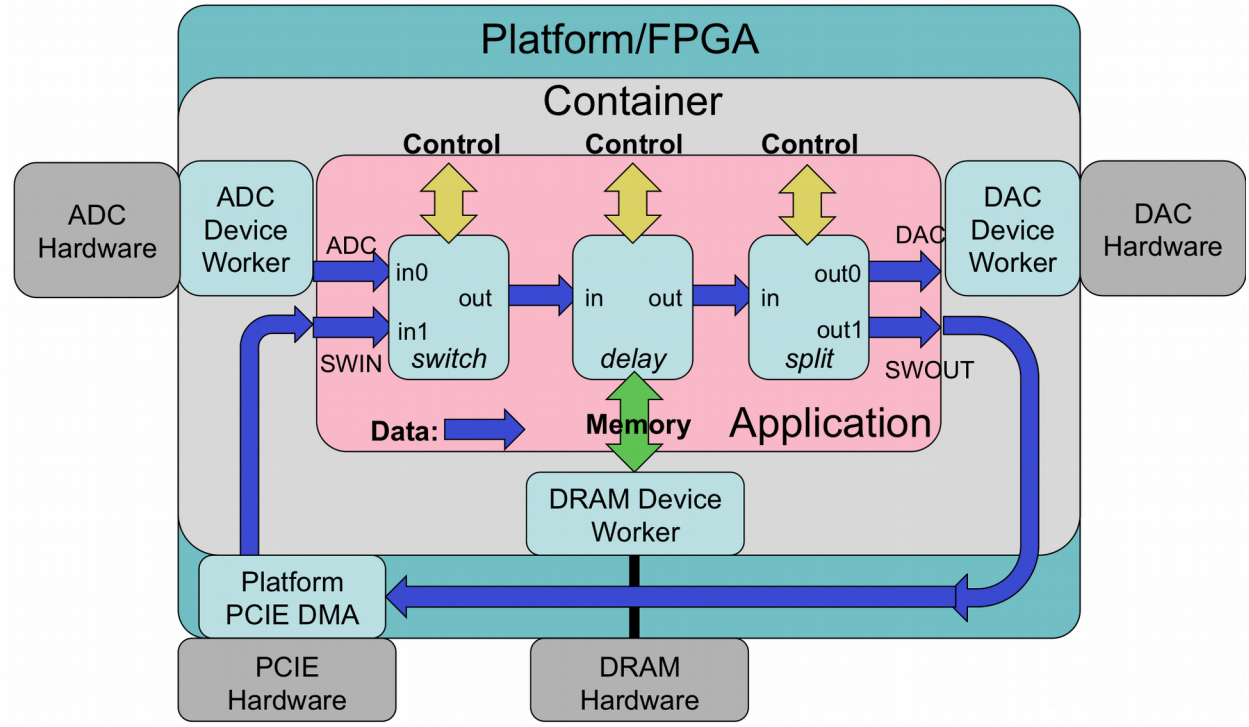

Figure 6: OpenCPI HDL Assembly on Container and Platform

6.4.6 Preparing the Bitstream/Executable Artifact File

This section describes how the container build process injects metadata into the resulting artifact file. There is no user control over this process, but it is useful to understand for troubleshooting purposes. Containers are built into bitstream/executable files by the FPGA back-end place-and-route or simulation tools. These tool-specific files are then post-processed into a generic **`.bitz`** file that acts as an OpenCPI artifact for application execution. This post-processing always compresses the tool-specific output files using **`gzip`**.

This post processing also uses an ***artifact XML*** file that is generated and describes what is in the bitstream/executable. This information includes the contents of the platform configuration, the assembly, and the container. The post-processing attaches *artifact XML* to the file in two ways.

The first is that the XML is compressed and embedded into the logic of the bitstream in a block memory. This allows OpenCPI software to extract it when the bitstream is loaded in an FPGA. It enables software to know what is in the bitstream without knowing or having access to the file it was originally loaded from. E.g., if the FPGA was booted from a flash memory attached directly to the FPGA and not accessible to software, software can still retrieve this information and know how to use the bitstream. The **`ocpihdl`** tool, described in the ***ocpihdl command-line utility*** section, is used to manually query this embedded information for troubleshooting purposes.

The second way the XML is attached to the file is outside the FPGA configuration logic, but attached to the bitstream (**`.bitz`**) file so that it can be retrieved from the file generically, regardless of the FPGA or simulation tools used to create the file. This allows software to know what is in the file, without it being loaded into a device, and without knowing any other files or locations that the artifact file came from. It makes the file self-describing. This attached XML is the same for all artifact files for all authoring models. The **`ocpixml`** command-line tool is used to extract this XML data into a separate file for examination.

In summary:

- target-specific FPGA or simulation tools create the raw container output file
- this file has embedded artifact XML in an initialized block memory
- this file is compressed
- the artifact XML is then attached to the compressed file like all other artifact files

Artifact files are installed in an OpenCPI runtime directory as referenced by the **`OCPI_LIBRARY_PATH`** environment variable. They are in the format expected by the internal OpenCPI mechanisms to load bitstreams onto platforms at runtime.

All artifacts built in a project are available in the **`artifacts/`** subdirectory of the project, and when exported from the project (if requested in the project's **`Project.exports`** file), in the **`exports/artifacts/`** directory.

These directories can be placed in the **`OCPI_LIBRARY_PATH`** environment variable that is used to find artifacts when executing OpenCPI applications. The **`ocpidev run`** command and the unit testing framework do this automatically when running applications or unit tests. When *not* using these, setting **`OCPI_LIBRARY_PATH`** is required.

6.5 How Clocks are Connected in an HDL Assembly or Container

Every connection in an assembly or container is in some clock domain. When workers are written using the control clock for data interfaces, the connections are in that clock domain. If one worker (A) specifies that there is a clock *input* signal on a data interface, implying that it can accept any clock, then the clock domain of the connection is determined by the clock at the other end of the connection (B), and the appropriate clock signal is fed into (A). Note clock handling is independent of data flow direction.

There are three cases to determine the clock domain of the connection when one end takes a clock as input (interface A):

1. If the interface (B) at the other end of the connection has no declared clock, then the control clock is used for that connection and fed to (A).
2. If the other end (B) has a clock *output* signal (i.e. **`clockDirection='out'`**), that clock signal will be fed into worker A's interface and will thus be the clock for the connection.
3. If that other interface (B) uses a clock from another one of its worker's ports (i.e. it has a **`clock`** attribute set to that other interface's name), then the clock for the connection is in fact the clock used at the interface at (B)'s worker, indicated by that **`clock`** attribute.

Clock associations are transitive: when the clock is set for (A), and another interface on the same worker specifies that it uses (A)'s clock (using the **`clock`** attribute), then that automatically determines the clock domain for any connection on that other port.

For worker ports in an assembly that are promoted to be external ports of the assembly, the clocking of the external port is inherited from the clocking of the worker port. Finally, if the clocks at both ends of a connection are determined to be different clocks, a clock-domain adapter is inserted into the connection.

These capabilities allow parts or all of assemblies to have a “data clock” independent of the control clock. Using the example where all ports are in a data clock domain with the clock supplied on one of its ports, all connected workers in the assembly that have independent data clocks will all be running their data ports and processing in that separate clock domain. By using the convention that input ports accept input clocks, all connected workers will simply propagate their clock domains.

The OpenCPI infrastructure for streaming data into and out of the the platform/FPGA (the SDP) is in a platform-defined clock domain separate from the control clock. Thus if all workers in the assembly are split-clock workers, then the entire data flow is in that SDP clock domain. An example in the core project (in **`projects/core/components/bias.test`**, when built for unit testing), is the unit test assembly for the **`bias_clock.hdl`** worker found in:

`gen/assemblies/bias_clock_0`

The assembly uses the **`bias_clock.hdl`** worker as well as two other unit test infrastructure workers, **`metadata_stressor.hdl`** and **`backpressure.hdl`**. Since

all three workers are split-clock, the input to the assembly carries the clock for the whole assembly, as shown in:

`gen/assemblies/bias_clock_0/gen/bias_clock_0-assy.v`

When this assembly is deployed in the default container, the data clock for the assembly is connected to the SDP's clock, separate from the control clock.

[Use a better example, and use a diagram]

7 HDL Simulation Platforms

The OpenCPI concept of an HDL platform encompasses both physical FPGA-based platforms as well as HDL simulators such as Mentor's modelsim, and Xilinx isim or xsim.

At build time, simulators are “just another target” when building primitives and workers, and “just another platform” when building assemblies and containers. The names for simulators in both target-related and platform-related XML attribute are **`modelsim`** , **`xsim`** and **`isim`**. E.g.:

```
ocpidev build --hdl-platform=zed \
              --hdl-platform alst4 --hdl-platform=modelsim”
```

would build for the ZedBoard Zynq-based platform, the Altera Stratix4-based platform, and the modelsim platform. If a worker was intended only for simulators, its **`OWD XML file`** would typically contain the attribute setting:

```
OnlyTargets='isim modelsim xsim'
```

Similarly, if an *assembly* was intended only for simulators, its XML file would typically contain the same attribute.

Even though simulators do not perform synthesis, building for simulators tries to elaborate the design at each build level to catch errors as early as possible. To suppress this incremental elaboration for faster build times, the XML attribute **`NoSimElaboration`** can be set to **`1`**. Since simulation builds are normally fast anyway, this is rarely worth it, especially for modelsim.

At runtime, an installed simulator is an available HDL platform just like any installed hardware HDL platform. Simulators act as much like a hardware FPGA platform as possible:

- It is discoverable (using **`ocpihdl search`**, or **`ocpirun -C`**)
- It appears as available without a bitstream being loaded

Previous versions of OpenCPI used the **`ocpihdl simulate`** command to start a simulator container as a server process. This is no longer necessary or supported. For backward compatibility for some test benches, this command does nothing but sleep indefinitely.

The following simulation server behavior is currently *disabled*, but will be enabled in a future release.

- A bitstream can be “loaded”, which in fact starts simulation (usually automatically as needed by **`ocpirun`**, or explicitly using **`ocpihdl load`**)
- It is persistent, so an application can be executed multiple times, which in fact will occur in the same simulation run
- It can be queried (e.g. to find the current value of a worker's properties) using various **`ocpihdl`** commands

7.1 Execution of Simulation Bitstreams and Containers.

Simulators that are installed for use by OpenCPI are automatically available as containers, similar to HDL hardware platforms or even software containers. When an application is run, these containers are available to be used if artifacts are built and accessible via the **OCPI_LIBRARY_PATH** environment variable (as described in Preparing the Bitstream/Executable Artifact File). For example, assuming that modelsim and isim simulators are installed for OpenCPI, on a CentOS7 system, the command **ocpirun -C** would output:

```
Available containers:
#  Model Platform       OS     OS-Version  Arch     Name
0  hdl   isim                                      lsim:isim
1  hdl   modelsim                                   lsim:modelsim
2  rcc   centos7        linux  c7          x86_64   rcc0
```

If an application is run, and bitstream files are available to use these simulators, they will be used automatically. To force components in the application to use a particular simulator, the -P option to ocpirun can be used, e.g.:

```
% ocpirun -P=modelsim myapp
```

would run **myapp**, forcing all components to be executed with modelsim. To force one component in the application to use modelsim, you could say:

```
% ocpirun -Pmycomp=modelsim myapp
```

The **ocpirun** command normally used to run OpenCPI applications has several options that apply only to simulators:

*Table 22: Simulations Options to **ocpirun***

Name	Letter	Description
sim_dir	*none*	The name of a directory where simulation outputs will be placed. The default is **simulations**, relative to where **ocpirun** itself is running.
sim-ticks	*none*	The number of simulation clock cycles to execute or until the application is done.

A complete description of the **ocpirun** command is in the ***OpenCPI Application Development Guide*** and in the ocpirun manual page.

As with execution on any hardware HDL device (FPGA), some of the components in the application may be in software containers running on the host processor. The data connections between workers inside the HDL device (or simulator) and outside the HDL device in software containers work normally. Since HDL execution is much slower in simulators than in real FPGA hardware, the software workers will see data consumed or produced by the HDL simulator device much slower.

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.

Each time a simulator is actually run under **ocpirun**, it will execute in a new subdirectory created for that simulation run, with the name:

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

Thus running applications that use simulators will result in one or more subdirectories holding simulation results for each simulation run. No subdirectories are created when the simulator is simply discovered using **ocpirun -C** or **ocpihdl search**.

Each simulation execution that is launched by **ocpirun** when it runs an application 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 **assert** in VHDL.
- The simulation run exceeds the control-plane clock cycle count provided by the **--sim-ticks** option to **ocpirun.**
- This ocpirun command receives a control-C.

The results of any simulation run can be viewed using the **ocpiview** command. This command with no arguments opens the most recent simulation run found in the **simulations** directory, using the simulation viewer associated with the simulator used in that run. If given an argument, it is the directory containing a particular simulation run. The normal pattern of development is to run **ocpiview** after execution if examining the simulation run in detail is needed.

For Xilinx “isim”, the actual underlying viewing command (in the subdirectory for the simulation run) would be:

```
isimgui –view sim.wdb
```

For modelsim, it would be:

```
vsim -view vsim.wlf
```

In all cases the log of the simulator's output for the run is in the **sim.out** file.

8 HDL Device Naming

The term ***HDL Device*** is used here to refer to an instance of an HDL platform in an OpenCPI system (and not a device attached to an FPGA *inside* the platform). HDL devices have unique names within the *system*. HDL devices host HDL containers for execution. Each name starts with a prefix indicating how the device is discovered and controlled by OpenCPI software.

The control schemes currently supported are:

PCI
: FPGA devices/boards accessible by PCI Express

Ether
: FPGA devices accessible via link-layer Ethernet

LSim
: FPGA devices that are in fact simulators (see the ***HDL Simulation Platforms*** section)

UDP
: FPGA devices that are accessible via IP/UDP

PL
: FPGA device in a Zynq SoC accessible via the on-chip AXI interconnect

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.

8.1 PCI-based HDL devices

HDL platform 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 PCIe device is the Xilinx ML605 development board. Another is the Altera Stratix4 development board (called **`alst4`** in OpenCPI).

8.2 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/TCP. 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 bridges 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`**. Newer Linux systems use more complex (but predictable) names that relate to busses and slots, e.g. **`enp14s0`** for PCI-based network interfaces. 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 must 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 sub-command of the **`ocpihdl`** utility that lists only the network interfaces available and usable for OpenCPI (the **`ethers`** command to **`ocpihdl`**).

8.3 Simulator Device Naming

OpenCPI runs HDL simulators in a way that makes them look like any other device to software. When they are available (installed for OpenCPI), they are discoverable like any other device. They are accessed by name according to this syntax:

`LSim:<simulator>`

Current simulators are **`modelsim`**, **`xsim`** and **`isim`**. Such simulators are discovered automatically, and can be listed with the **`ocpihdl search`** command, along with hardware FPGA platform devices.

9 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, simulators, etc.)
- Testing the existence of a specific FPGA device
- Reading and writing specific registers in an FPGA device
- 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>]
```

Full documentation for the **`ocpihdl`** command is at the ocpihdl manual page.

10 HDL Platform and Device Development

HDL Platform development is the activity that makes a FPGA-based hardware platform fully enabled for running OpenCPI applications. This activity is sometimes called making an ***OpenCPI System-support Package (OSP)***. This is a more specific OpenCPI term for what is sometimes called a ***Board Support Package (BSP)***. In OpenCPI the definition of an HDL platform is an FPGA surrounded by and connected to a set of devices, and possible slots that accept plug-in optional cards with devices on them.

Developing HDL platform workers and device workers is described in a separate ***OpenCPI Platform Development Guide*** document. It covers the development of HDL:

- Platform workers: the singleton worker that bootstraps the platform and container
- Device workers: workers that support external devices attached to FPGAs
- Platform configurations: assemblies of platform workers and some device workers
- Slot types: standard definitions of the signals and pins of a slot connector
- Slots on platforms: how to define slots on HDL Platforms
- Cards for slots: how to define cards that contain devices and plug into slots

All of these asset types can be developed in projects along with other assets.

11 Glossary of Terms

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

11.1 OpenCPI Terminology

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

ACI

See ***Application Control Interface***.

adapter worker

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

application

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

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

Application Control Interface (ACI)

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

application worker

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

argument

See ***operation argument***.

artifact

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

asset

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

authoring model

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

back pressure

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

build configuration

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

CDK

See ***Component Development Kit***

component

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

Component Development Kit (CDK)

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

component library

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

component specification

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

component unit test suite

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

configuration property

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

configuration property accessibility

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

container

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

control-application

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

control interface

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

control operations

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

control plane

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

control software

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

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

core project

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

Datagram Remote Direct Memory Access (DG-RDMA)

Datagram Remote Direct Memory Access (DG-RDMA) is a protocol for achieving remote direct memory access (RDMA) that uses a datagram service (DG). In OpenCPI, DG-RDMA is a data plane protocol that achieves RDMA using the Layer 2 (L2) Ethernet type of datagram service. While OpenCPI protocols define how components and workers communicate regardless of authoring model, platform, or container, data plane protocols like DG-RDMA define how containers

communicate and provide the underlying inter-container protocol that supports inter-worker communication. Inter-container protocols can be carrying multiple worker-to-worker conversations between containers, and each of these conversations can be using different inter-component protocols.

data interface

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

data plane

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

device proxy worker

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

device worker

See ***HDL device worker***.

Digital Radio Controller (DRC)

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

HDL assembly

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

HDL authoring model

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

HDL build hierarchy

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

HDL build process

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

HDL card

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

HDL data interface

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

HDL device emulator worker

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

HDL device worker

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

HDL interface

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

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

HDL platform

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

HDL platform configuration

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

HDL platform worker

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

HDL primitive

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

HDL primitive core

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

HDL primitive library

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

HDL slot

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

HDL subdevice worker

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

HDL worker

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

lifecycle state model

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

OAS

See ***OpenCPI Application Specification***.

OCS

See ***OpenCPI Component Specification***.

OHAD

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

OHPD

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

opcode

See ***operation code***.

OpenCL (OCL) authoring model

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

OpenCPI Application Specification (OAS)

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

OpenCPI Component Specification (OCS)

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

OpenCPI HDL Assembly Description (OHAD)

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

OpenCPI HDL Platform Description (OHPD)

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

OpenCPI Protocol Specification (OPS)

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

OpenCPI System support Project (OSP)

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

OpenCPI Worker Description (OWD)

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

operation argument

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

operation (within a protocol)

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

operation code (opcode)

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

OPS

See ***OpenCPI Protocol Specification***.

OSP

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

OWD

See ***OpenCPI Worker Description***.

package ID

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

parameter

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

platform

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

platform worker

See ***HDL platform worker***.

port

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

port readiness

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

project

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

project registry

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

property

See ***configuration property***.

protocol specification

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

protocol summary

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

RCC, RCC authoring model

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

Resource-Constrained C (RCC) authoring model

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

registry

See ***project registry***.

RCC worker

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

run condition

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

run method

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

spec file

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

SpecProperty

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

system

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

target

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

worker

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

worker property

A ***worker property*** is a configuration property related to a particular implementation (design) of a worker; that is, one that is not necessarily common across a set of implementations of the same high-level component specification (OCS). A worker property is additional to the properties defined by the

component specification being implemented. See the section on how a worker access its properties in the ***OpenCPI RCC Development Guide*** and the sections on property access and property data types in the ***OpenCPI HDL Development Guide*** for more information.

unit test

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

Zero-Length Message (ZLM)

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

11.2 Industry Terminology

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

Advanced eXtensible Interface (AXI)

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

Advanced RISC Machine (ARM)

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

ARM

> See ***Advanced RISC Machine***.

AXI

> See ***Advanced eXtensible Interface***.

Board Support Package (BSP)

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

Central Processing Unit (CPU)

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

CPU

> See ***Central Processing Unit***.

Digital Signal Processor (DSP)

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

digital signal processing

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

DSP

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

eXtensible Markup Language (XML)

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

Field-Programmable Gate Array (FPGA)

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

FPGA

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

FPGA bitstream

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

framework

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

gateware

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

General-Purpose Processor (GPP)

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

GPP

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

GPU

See ***Graphics Processing Unit***.

Graphics Processing Unit (GPU)

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

Hardware Description Language (HDL)

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

HDL

> See ***Hardware Description Language***.

Integrated Synthesis Environment (ISE®) Design Suite

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

ISE® Simulator (Isim)

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

isim

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

OCL, OpenCL

> See ***Open Computing Language***.

Open Computing Language (OCL, OpenCL)

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

OSS

> See ***Open Source Software***.

Open Source Software (OSS)

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

PCI

See ***Peripheral Component Interconnect***.

PCIe

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

Peripheral Component Interconnect (PCI)

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

Peripheral Component Interconnect Express (PCIe)

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

RPM

See ***RPM Package Manager***.

RPM Package Manager (RPM)

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

System on a Chip (SoC)

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

Verilog

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

VHSIC Hardware Description Language (VHDL)

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

Vivado® Design Suite (Vivado, Xilinx Vivado)

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

Vivado® Simulator

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

XML

See ***eXtensible Markup Language***.

Xsim

See ***Vivado® Simulator***.