OpenCPI Platform Development Guide

OpenCPI Release: v2.4.7

Revision History

Revision	Description of Change	Date
0.01	Initial	2014-12-15
0.5	Release partial draft	2015-02-27
0.6	Add more general introduction to enabling systems for OpenCPI	2015-04-23
0.7	Add content about device workers/subdevices/device proxies and endpoint proxies.	2015-05-06
0.8	Add content about platforms and platform workers, apply review comments for HDL devices	2015-06-22
0.9	Add content about developing HDL platform support outside the core directory tree	2015-08-28
1.0	Update for 2016Q2 release	2016-05-23
1.1	Update for 2017Q1 release, still requires diagrams and cpmaster and sdp protocols	2017-03-03
1.2	Update for 2017Q2, more detail and clarity on cards/slots/subdevices and many edits. No major new content.	2017-06-15
1.3	Update software platform enablement	2018-05-02
1.4	Update software platform descriptions to final 1.4 capabilities, including HDL platform SD cards etc.	2018-09-25
1.5	Update for release 1.5	2019-04-30
1.6	Updates for release 1.6	2020-01-02
1.7	Updates for 1.7, update for SW platform exports, and ADC/DAC worker model	2020-05-28
1.8	Updates for 2.1: remove CentOS6 references, replace installation, deployment scripts with ocpiadmin, DRC	2021-01-08
1.9	Add shared glossary of terms chapter	2021-03-15
2.0	General re-edit, and describe “supports mappings” for devices in platforms, update DRC	2021-03-21
2.1	Edit for makeless, adding new attributes to the HDL platform XML	2021-07-08

Table of Contents

1 Introduction

This document describes how to enable new platforms for OpenCPI, which will support the execution of component-based applications. It assumes a basic knowledge of OpenCPI as described in the ***OpenCPI Application Development Guide*** and ***OpenCPI Component Development Guide***. Platform development is the third class of OpenCPI development, beyond *application development* and *component development*. It involves configuring, adapting and wrapping various aspects of hardware platforms, operating systems, system libraries, and development tools. It applies to general-purpose computing platforms, FPGA platforms and GPU platforms. It applies to self-hosted development as well as cross development.

The questions this document tries to answer are:

- What are suitable platforms?
- What is involved in making a platform ready?
- What are the actual steps and processes for making a platform ready?

These questions are answered separately for enabling development *targeting the platform* (creating and building assets) vs. enabling *execution on the platform* (running applications).

These questions are answered separately for GPP, FPGA, and GPU platforms.

After introducing all these topics, this document has sections for each aspect in the following table:

Table 1: Categories of Platform Development

	Examples	Development Tools	Execution Environment	I/O and Interconnect Device Support
General-Purpose Processors (GPPs)	X86 (Intel/AMD), ARM (Xilinx/Altera/TI)	Compiler tool chains, make python	Operating System, Libraries, Drivers	
FPGAs	Xilinx (Virtex, Zynq-7000) Intel/Altera (Stratix, Cyclone) Mentor/Modelsim	Synthesis Place&route Simulation	Bitstream loading Control Plane Drivers Data Plane Drivers	Directly attached I/O devices
Graphics Processors	Nvidia Tesla AMD FirePro	Compilers, Profilers	Execution Management Drivers Data Plane Drivers	

1.1 References

This document requires information from several others. To actually perform platform development it is generally useful to understand OpenCPI component and application development, as well as the installation process for previously enabled platforms. To simply get a flavor for what is involved in platform development, only the OpenCPI Introduction Briefing (and Glossary) as well as the OpenCPI Platform Development briefings are required.

Table 1 - Table of Reference Documents

Title	Published By	Link
OpenCPI User Guide	OpenCPI	https://opencpi.gitlab.io/releases/v2.4.7/docs/OpenCPI_User_Guide.pdf
OpenCPI Installation Guide	OpenCPI	https://opencpi.gitlab.io/releases/v2.4.7/docs/OpenCPI_Installation_Guide.pdf
OpenCPI Component Development Guide	OpenCPI	https://opencpi.gitlab.io/releases/v2.4.7/docs/OpenCPI_Component_Development_Guide.pdf
OpenCPI Application Development Guide	OpenCPI	https://opencpi.gitlab.io/releases/v2.4.7/docs/OpenCPI_Application_Development_Guide.pdf

2 OpenCPI Systems and Platforms

OpenCPI provides a consistent model and framework for component-based application development and execution on various combinations of ("heterogeneous") processing technologies, focusing mostly on embedded systems.

An ***OpenCPI system*** is a collection of processing elements that can be used together as resources for running component-based applications. OpenCPI considers each processor part of some hardware *sub*system. These subsystems are wired together using some interconnect technologies (e.g. networks, buses, fabrics, cables).

We call each available processor and its surrounding directly-connected hardware a ***platform***. Most commonly, a platform is a "card" or "motherboard" housing a processor and associated memory and I/O devices. The data paths that allow platforms to communicate with *each other* are called ***interconnects***. The most common interconnects for OpenCPI systems are PCI Express or Ethernet, although others are also supported and used for some platforms (e.g. the AXI system interconnects on Xilinx Zynq-based SoC-based systems).

The scope of a system can be a small embedded system that fits in a pocket, or racks full of network-connected hardware that act as a "system of systems". Since this "system" definition is somewhat broad, the focus is on efforts to enable running OpenCPI at each ***platform*** and ***interconnect*** within a system. Hence ***platform development*** is enabling a platform, and enabling a system is primarily *enabling whatever platforms and interconnects are in the system*.

Our most common and simple example system is the ZedBoard from Digilent (zedboard.org), which is based on the Xilinx Zynq chip. This chip is called a "system on chip" or SoC, and indeed has two processing elements connected with an interconnect, all on one chip: 1) a dual-core ARM general-purpose processor, and 2) an FPGA. They are connected via an on-chip fabric based on the AXI standard, and each is connected to some I/O devices. Thus the ZedBoard ***system*** consists of two ***platforms*** that happen to reside in the same chip, with an AXI ***interconnect*** between them.

Another common example system is a typical PC, which has a multicore (1-12) Intel or AMD x86 processor on a motherboard. If cards are plugged into slots on the PC's motherboard, and those cards have processors (e.g. GPP, FPGA, GPU) on them, then those cards can act as additional platforms in that system.

We consider multi-core GPPs as single "processors" since they generally run a single operating system and act as a single resource that can run multiple threads concurrently.

The final defined element of an OpenCPI system is ***devices***, which are locally attached to some platform to act as source or sink of data to enter or exit the system. Thus devices are distinct from interconnects.

A ***system*** consists of ***platforms*** connected by ***interconnects***, and platforms can have local ***devices***, either permanently attached (e.g. on the motherboard) or on optional ***cards*** in the platform's ***slots***.

2.1 Inside an OpenCPI Platform

As mentioned above, a ***platform*** consists of a processor (GPP, FPGA, GPU, etc.) attached to ***interconnects*** allowing it to communicate with other platforms. The processor may have multiple cores, and may even consist of multiple processor chips (such as a dual-socket motherboard where two Intel X86 processors act together). There are two other elements that make up a platform: ***devices*** and ***slots***.

Devices are hardware elements that are locally attached to the processor of the platform. They are controlled by special workers called ***device workers*** (analogous to "device drivers"), and usually act as sources or sinks for data into or out of the system, and thus can be used for inputs and outputs for a component-based application running on that system.

When a device is hard-wired to the platform, it is defined as part of the platform. It is also common for platforms to be optionally configured with add-on ***cards*** that provide additional devices for the platform. To allow for this, platforms can have ***slots***, which are an intrinsic part of the platform, and enable cards to be plugged in that make additional devices accessible to the platform. Such cards may be plugged in to any platforms that have compatible slots.

The OpenCPI concepts of ***devices*** and ***slots*** currently only applies to HDL/FPGA platforms. On software (GPP) platforms, the (usually Linux) operating system is used for device support and thus no OpenCPI device support is included.

An example ***system*** is the ZedBoard, which has a Xilinx Zynq SoC part on its motherboard that has a dual-core ARM processor as well as an FPGA inside. This board thus has two platforms, 1) an ARM-CPU-based ***platform*** attached to a variety of external peripherals (Ethernet, USB, etc.) as well as 2) an FPGA-based ***platform*** attached to some external peripherals (devices) as well as an attached external FMC ***slot*** into which different ***cards*** may be plugged.

Thus the devices available on a platform are either a permanent part of the platform, or are available on defined cards when they are plugged into one of the platform's slots.

Platforms can have any number of devices, and may have multiple devices of the same type. When cards are plugged into a platform's slots, they make additional devices available to the platform.

- Systems have platforms and interconnects.
- Platforms have slots and devices, and are attached to interconnects.
- Cards plug into slots and have devices.

2.2 Enabling Development for, and Execution on, an OpenCPI Platform

To make a platform usable for OpenCPI one must enable:

- ***development*** – installing and configuring tools to create executables targeting it, on some development host.
- ***execution*** – providing the runtime infrastructure to execute applications (including component unit tests) on it.

Development is the process of producing "executable binaries", and *execution* is the process of running those binaries on the platform, as *part* of the execution of a component-based application on a system. Enabling development targeting an embedded platform typically involves installing a cross-compilation tool chain on some development host that can produce binaries for the targeted platform.

Development activities are typically said to be done at *build-time*; execution activities are typically said to be done at *run-time*.

OpenCPI uses the term "binary artifact" as a technology-neutral term for the binary file that executes on various processing technologies. On GPPs, they are typically "shared object files" or "dynamic libraries". On FPGAs, they are sometimes called "bitstreams". On GPUs, they are sometimes called "graphics kernels".

Enabling *development* targeting a platform includes procuring, installing, configuring and integrating the various tools necessary to enable the developer to design and create binary artifacts from source code, for the platform. Some adaptations, scripts or wrappers are typically required to enable such tools to operate in the OpenCPI development context. OpenCPI does not preclude or require GUI-based IDEs in the component or application development process.

For most embedded systems, the development tools do not run on the system itself, but instead run on a separate "development host", typically a (possibly virtual) PC. The unusual, but still possible, case where the tools run on the targeted embedded system itself, is termed "self-hosted development". When the target platform is embedded and development tools run on a separate "development host", that is termed "cross development", using "cross-tools" (e.g. cross-compilers). OpenCPI focuses on "cross development", except when targeting the development host itself.

All development hosts also act as execution platforms since any host capable of running development tools can act as an OpenCPI execution platform for the GPP/processor of that system. OpenCPI contains tools that must be compiled on the development host and will always run on the development host. Thus a development host is established to execute tools used to create binaries for itself and for other target platforms (cross development). These tools include both OpenCPI's own tools as well as target-specific tools from third parties.

2.3 Enabling New Systems for OpenCPI

Enabling systems for OpenCPI implies enabling the platforms and interconnects in the system for OpenCPI, and thus enabling the processors and devices on the platforms and the devices on any cards used in the system. The first step in the enabling process is to establish the inventory of these elements in the system.

Since specific interconnects, processors, and devices are frequently found in many different systems, it is likely that support is already in OpenCPI. So the enabling process is really for the elements without OpenCPI support. Furthermore, there may be devices in the system that do not require OpenCPI support, either because they are not going to be used, or there is no benefit given the scope of what OpenCPI does.

The system inventory for OpenCPI consists of:

- Processors (attached to interconnects and devices)
- Interconnects (among processors)
- Devices (attached to FPGAs or on cards)

For each, if there is no existing support in OpenCPI, support modules must be developed. For some, there may be existing partial support that must be extended for the intended usage. For new classes of processor (beyond GPP, FPGA, GPU) or new interconnects, the core framework of OpenCPI must be enhanced. Otherwise individual support modules can be developed without modifying the core infrastructure of OpenCPI. Later sections of this document describe the requirements and process of enabling each type of element in the system. Processors must be supported for both development (e.g. compilation tool-chain) and runtime (e.g. device driver changes). Devices and interconnects are normally enabled for runtime only (i.e. their support does not involve changing the build process or tools).

Since OpenCPI is open source software, it is very desirable to contribute new or extended support modules back to the community since most such modules will likely be used in other systems. OpenCPI has a built-in project, in the **`projects/platform`** directory, which contains such sharable support assets.

A key aspect of supporting new system elements is obtaining the necessary technical information and tools, and in some cases performing a variety of experiments to assess feasibility and derive otherwise unavailable information. When the information is not available from public sources (such as data sheets for device ICs), NDAs or other confidentiality agreements may be required. Vendors may not supply the necessary information, leaving reverse engineering the only option, subject to legal constraints.

OpenCPI, especially on FPGAs, operates directly on the FPGAs and usually interacts directly with attached devices. Thus either the vendor must supply the required device workers or supply the information required to develop them. Preexisting FPGA modules for attached devices that were not written with OpenCPI in mind are usually unsuitable as is, and must be modified or replaced. In some cases they can be “wrapped”, although this may result in the addition of undesired overhead. Wrapping involves using existing code as a primitive module inside an OpenCPI device worker.

2.4 Process Template for Enabling OpenCPI on a New System

Here is a list of steps normally required to enable a new system for OpenCPI. The first four steps are used to establish a clear base of information to estimate and plan the effort. This section may be useful for managers planning the effort, and includes non-technical aspects of the process.

2.4.1 System Inventory

Collect information to determine the rough **scope of the effort.**

This is the basic inventory of the relevant parts of the system, each of which needs to be considered in planning to enable the system.

Develop the list of processors, interconnects, and devices in the system that are relevant to OpenCPI applications, establishing the system breakdown. This is usually a "block diagram" and "data sheet" exercise.

The time required for this activity depends to a large extent on whether complete information about the target system is available and not restricted.

2.4.2 Processor, Interconnect and Device Assessment

For each, evaluate the state of support currently within OpenCPI and identify additional technical efforts likely to be required. These assessments establish a rough level of effort, without necessarily establishing feasibility. A ROM (rough order or magnitude) LOE (level of effort) can be established.

This effort requires matching technical support requirements with the current state of support in OpenCPI, and thus would require either gaining some familiarity with the OpenCPI supported hardware universe, or engaging with a group that is already familiar with it.

2.4.2.1 For each processor, determine the level of support within OpenCPI:

- Currently supported (e.g. Zynq ARM processor, Intel AMD x86 processor)
- Variant supported (e.g. Virtex6 vs. Virtex 7 FPGA)
- New processor of existing type (e.g. PPC CPU vs. ARM CPU)
- New class of processor (e.g. Adapteva Multicore). This requires new work usually outside the scope of "enabling a new system"

2.4.2.2 For each processor, determine the level of tool chain support within OpenCPI:

- Currently supported by OpenCPI
- Requires a version upgrade/downgrade/variant of what is currently supported
- Not currently supported by OpenCPI

2.4.2.3 For each interconnect, determine level of support required for each processor type that is attached to it:

- Currently supported
- Requires updating or enhancements (e.g. PCI-E gen3x8, vs. PCI-E gen2x4)
- Not currently supported for processor type (e.g. Ethernet L2 on FPGAs)

2.4.2.4 For each device attached to an FPGA or on a required card, determine the level of support within OpenCPI and identify the additional technical efforts likely to be required the each device.

- Currently supported
- Requires updating or enhancement
- Not currently supported, but similar devices are supported as a model
- Not supported and different from any existing supported devices

2.4.3 Assemble the Technical Data Package

For all elements requiring updated code or new support, collect information necessary to perform the enabling technical development, and to establish more detailed work estimates. The information required here is more comprehensive than the basic information required above: it must be sufficient to perform the needed development. This effort will establish the availability of appropriate information to perform the technical development. It will also find any roadblocks to obtaining the information (vendor unwillingness, legacy unavailability).

In the particular case of device support, some vendors may not expose sufficient information to support their devices in the absence of their own “drivers” that embed their ICD (interface control document), information they consider a trade secret. Such positioning by vendors may make optimal support for OpenCPI challenging or impossible.

Required information for processors/FPGAs include:

- Tool requirements (which tools and versions, which settings, cost)
- Connectivity technical details (e.g. how interconnects, devices and slots are connected, including pin-outs etc.)

Required information for devices include:

- Device data sheets or equivalent functional and interface documentation
- Programming/application guides
- Appropriate/relevant existing OpenCPI support modules
- How the devices are attached to the processors. (e.g. ICD)

As part of this effort, any additional required feasibility experiments, reverse engineering or other design/analysis tasks, are identified. These are tasks to fill in the information gaps in order to have high confidence work estimates and plans.

2.4.4 Experiments to Establish Feasibility and Missing Information

There are typically uncertainties and gaps in technical documentation, and in some cases the information is unavailable due to proprietary restrictions. In any case, as a final step to enable accurate work breakdowns and time/cost estimates, there are usually a set of tasks involving hands-on experience with the target system prior to the actual technical development. Such experiments are derived from the process of the above tasks (i.e. discovering knowledge gaps), and may include:

- Identify, install, and configure vendor-provided tools and hardware.
- Verify and/or establish functional or performance capabilities of the system that are missing or questionable from the information obtained earlier.
- Reverse-engineer missing ICD aspects (assuming no legal impediments).

These hands-on tasks establish the final information to plan and budget for the level of effort and scope of enabling a new system for OpenCPI. This activity depends on access to a real system. When the vendor is performing the work for their own hardware, these tasks may be unnecessary.

2.4.5 Planning and Specification

This phase of enabling a system is specifying the technical capabilities to be achieved for OpenCPI on the target system, and planning the tasks to develop and verify the required functionality. The specifications are mostly based on achieving functionality that already exists on other systems, so they are mostly references to other existing documents, with particular options, exceptions, or limitations highlighted.

Every system element not currently supported requires a development task, while all system elements, including those that are already supported, should still have a verification task planned.

In some cases, a system may have desirable functionality that is not yet well defined or standardized in OpenCPI. In this case some proposed changes to the OpenCPI framework itself may be warranted along with a discussion with OpenCPI maintainers on gitlab (via gitlab issues) or email.

The tasks defined here will be of types described more fully in other sections of this document, where the various aspects of platform development are described in detail.

2.4.6 Technical Development

The various technical development tasks are of the types corresponding to specific sections of this document. Each type in the following list may or may not be required to enable a given system. Small updates or enhancements to existing modules are common.

- Enabling a new development system (including native development and execution)
- New GPP development tools
- New FPGA tools

- New FPGA platform
- New FPGA devices
- New FPGA cards
- New GPP platform
- New interconnect for GPP platforms
- New interconnect for FPGA platforms

2.4.7 Verification

Verification requirements may be very project-specific, but for each system element, a baseline verification should be defined, including specific characterization, both manual and automated.

For processors, running all OpenCPI tests and the unit tests in built-in projects is a baseline. Similarly, for FPGAs, building all built-in projects and running unit tests is a baseline. All such tests can run in the OpenCPI Continuous Integration (CI) system, on gitlab. if a dedicated test system is made available.

2.4.8 Contribution

To reduce all efforts at system enablement for OpenCPI, it is strongly encouraged (or in some cases required by licensing), that enhancements to existing support modules, and new support modules for widely available processors, interconnects and devices be contributed back to the appropriate repositories. Usually this consists of merge requests to the built-in **`platform`** project.

While assembling the data package as described in Assemble the Technical Data Package section above, the developer should maintain a complete list of licensed tools, items and associated repositories in order to ensure changes are contributed appropriately back to the sources.

3 Enabling the (new) Development Host

Before any required tools are installed on the development host, the basic development configuration for the host must be established for OpenCPI. This process starts with the same initial steps as outlined in the ***OpenCPI Installation Guide***: installing the **`git`** package, enabling your user for **`sudo`**, and downloading the OpenCPI source tree using the **`git clone`** command.

A binary/packaged distribution of OpenCPI such as RPM is not appropriate for enabling a new development host: a source installation is required.

The steps required for any software platform (development host or embedded GPP), is described in the next section, Enabling GPP Platforms. The primary aspects that are different with the *development host* are:

- The OpenCPI source tree lives on the development host (or is accessible to it).
- The C++ compilation tool chain is self hosted, rather than cross-compiled.
- A number of development-related software packages must be specified and installed in the Platform Package Installation Script, such as **`make`**, and **`python3`**. Such tools are neither required nor expected on embedded platforms.
- When the development host platform is defined for OpenCPI, the top-level installation script, **`scripts/install-opencpi.sh`**, can both install as well as test the system.

For a new development host, the operating system installation/configuration should be defined and documented, usually requiring at least some manual steps. This should, at a minimum, be described in the README file in the platform's directory. Development hosts that are likely to be used by others are usually established in the **`core`** built-in project.

If a new development host is a variant or new version of an existing supported one, the task is usually modest. However, a development host actually “hosts” other development tools for other platforms (e.g. FPGA tools), and OpenCPI requires certain other tools to be installed (e.g. python3). Thus testing and specifying how all the required packages are installed, and whether various FPGA tools can be installed and run, can be time consuming. OpenCPI already has support for various versions of CentOS, Ubuntu, and MacOS (MacOS without FPGA tools).

4 Enabling GPP Platforms

To execute on GPP platforms, new additions or modifications to the OpenCPI framework software may be required. The framework software is highly portable and is supported on a number of platforms and environments. However, it is possible for a new compiler, toolchain, or system libraries to require adjustments to the OpenCPI framework software. Here are a few reasons that may require modifications to framework software:

- New compilers have new correct warnings that should be addressed.
- System headers conform better to standards, requiring new correct header inclusion.
- Some compilers are “dumber” that others, requiring code to be “dumbed down” to avoid language or library features that are not universally supported.
- Some compilers are stricter than others, requiring code that was previously accepted to be “tightened up” to be strictly compliant and accepted.

These reasons may result in modifications to the OpenCPI framework software, but they should not and cannot make that software stop working on existing supported platforms. Any such modifications should be done with care, and whenever possible, the modifications should be rebuilt and retested on a number of existing supported platforms. Such updates should be submitted in a merge request to OpenCPI.

The process of enabling a GPP platform is based on a source distribution of OpenCPI. A binary/packaged distribution of OpenCPI such as binary RPMs, is not appropriate for enabling a new development host.

Enable a new software platform for OpenCPI requires some experience with **`make`** and **`bash`**. If you are not familiar with make files and bash scripts, it will be difficult. OpenCPI assumes the availability of **`bash`**, **`make`**, and **`python3`** on all GPP development hosts.

4.1 Enabling OpenCPI Development Targeting New GPP Platforms

GPP (software) platforms are established by creating a new directory under the **`rcc/platforms/`** directory in any OpenCPI project.. In a *future* release this will be done using the command:

```
ocpidev create rcc platform <new-platform>
```

The name of the directory (the platform's name) should be a lower case name that usually includes a OS major version after the name of the OS. In particular, different versions of Linux should be named by their distribution name or organization providing the OS. Examples for software platforms are: **`rhel5`**, **`centos7`**, **`ubuntu16`**, **`xilinx13_4`**, **`macos10_13`**. The naming concept is that each software platform has binary compatibility within its minor versioning, but not *necessarily* with major versioning. The currently supported software platforms can be found in the **`rcc/platforms/`** directory of the projects that are part of the OpenCPI source tree. Anyone can add support for new software platforms in their own OpenCPI projects.

The required and optional files in the platform's directory are defined in the Software Platform Files section below, but a discussion of why and how they are used occurs first.

Whether a software platform is the development host itself, or an embedded GPP using cross-development, the tool chain must be established on a development host. For enabling development hosts, there is typically a default tool chain that is installed globally in the system for any development task on that system. For embedded cross-developed platforms, a specific cross-development package must typically be installed using one of several methods described below.

Satisfying the OpenCPI software dependencies for any platform uses two classes of underlying software installations:

1. Prebuilt, globally-installed software packages that are a (usually optional) standard part of an operating system installation.
2. Externally sourced software packages that require a separate combination of downloading, building and installation.

We use the term ***standard packaged software*** to indicate the first category. E.g, a package of development tools is commonly a standard installation option on many systems. Since they are standard packages for a given operating system installation, they are prebuilt and installed globally on the (usually a development) system. Installation of such software is usually accomplished using commands like **`yum`** or **`apt`**.

The second category is essentially ad hoc extra required software that must be installed using specific scripts unique to that software. We use the term ***prerequisites*** for such software, and expect individual scripts to be written to download/build/install them. Software packages required by OpenCPI at runtime are normally prerequisites since they must be cross-compiled for all embedded platforms, whereas those required only for development may be supplied in either of the two categories.

So for any given platform, OpenCPI requires a combination of standard packaged software and prerequisite software.

OpenCPI uses the following process steps when building itself for a software platform, based on the open source tree from **`gitlab.com`**. Enabling all these steps is described below.

1. Install the standard software packages required for OpenCPI from the appropriate network software package repository (usually associated with the Linux distribution), e.g. development tools for the development host.[1]
2. Download and build any *platform-specific* ***prerequisite*** software packages, that are not available as standard packaged software.[2]
3. Download and build the standard (always or frequently used) OpenCPI-specified ***prerequisite*** software packages.
4. Build the OpenCPI framework libraries and executables from source code.
5. Build all the software assets in the built-in projects that are part of OpenCPI, in the same gitlab repository (RCC workers and ACI applications).
6. Run a number of tests to verify OpenCPI operation (software only).

For a fully enabled/supported *development hos*t, the above steps are all done in the single master installation script called:

```
./scripts/install-opencpi.sh
```

For enabled/supported embedded software platforms (and all other types of platforms), the above steps (except testing, step #6) are done with an **`ocpiadmin(1)`** tool operation for platforms that are *not* development hosts:

```
ocpiadmin install platform <platform>
```

Testing embedded platforms is not part of the installation since it usually requires additional hardware setup and configuration.

These above steps, and how to enable them for a software platform, are described in detail in the following sections. A summary of the files required in a platform's directory to enable the above steps is:

1. A software platform definition file, named ***`<platform>`*****`.mk`**, which sets variables that describe the platform. Between 3 and 20 variables might be required.
2. *For development hosts only*, a script to check that the currently running system is in fact the given software platform, e.g. answering the question: is this system running ***`<platform>`***? The script is typically 5-10 lines of shell script code, and is called ***`<platform>`*****`-check.sh`**.

1 Software package repositories are sometimes captured locally when internet access is not always available. Some standard software packages are individually captured/downloaded so they can be (re)installed offline.

2 Similarly, prerequisite software is sometimes downloaded and stored locally/offline and the “download” step simply uses the locally available downloads.

3. If needed, a script to install various standard packaged software from software package repositories to support OpenCPI for this platform, called **`<platform>-packages.sh`**.
4. If platform-specific prerequisite packages are indicated in the software platform definition file, then there should be a *prerequisite* script for each one, named **`install-<prerequisite>.sh`**.

4.1.1 The OpenCPI Build Process for Software Platforms

The OpenCPI build process is designed to support a variety of platforms all built in the same source tree. I.e. all the built results for all platforms coexist in a directory structure, in per-platform directories. This allows:

- Multiple development hosts to share the same file system and same copy of the source tree
- Multiple cross-compiled/embedded platforms all built in the same source tree, even when cross-compiled using different development hosts.
- Multiple disparate runtime systems to share the same OpenCPI installation via network mounts.

This structure allows for rapid and productive development, debugging and testing across a range of platforms simultaneously. The process of building for a given platform thus can coexist with others and can mostly proceed in parallel, independently, from one or more development hosts.

4.1.1.1 Self-identify the Development Platforms

When building for development platforms, the first step in the build process is to decide which platform we are actually running on and where its directory of platform-specific files is. After this, the various scripts and tools associated with that platform are used.

Typing **`./scripts/install-opencpi.sh`** (or individual scripts called by that script) at the top level of the OpenCPI source tree does this self-identification step almost immediately. The algorithm for this is:

- For each project indicated in the project registry and each project indicated in the **`OCPI_PROJECT_PATH`** environment variable, look for software platforms defined in the **`rcc/platforms`** directory, and invoke the **`<platform>-check.sh`** script found there. If that script succeeds, that platform and its directory are now used as *the platform we are running on*.

The exact definition and operation of this script, and examples, are described in the Softare Platform Check Scripts section. Even when targeting cross-development platforms, the *development host platform* must still be identifiable this way. When building OpenCPI for cross-developed platforms, the target platform name is specified explicitly, but the same project search algorithm is used to find a **`rcc/platform/<platform>`** directory for the specified platform. No **`<platform>-check.sh`** script is needed for cross-developed platforms.

4.1.1.2 Install Standard Packaged Software from the OS's Package Repository

The tools required to support OpenCPI development on a development host are usually obtained using standard package management software (e.g. **`yum`** or **`dnf`** on CentOS) and the OS's repositories accessible on the internet. Different Linux distributions use different repositories, and different commands, to retrieve and install software packages from those repositories. If a platform requires any such packages to be installed, there must be a script file ***`<platform>`*`-packages.sh`** in the platform's directory.

> Note that when installing a prepackaged OpenCPI binary distribution (e.g. from RPMs using the **`yum`** command on CentOS platforms), the required packages are usually installed automatically as part of installing that OpenCPI package. Such an OpenCPI installation is not used when enabling new software platforms.

This ***`<platform>`*`-packages.sh`** script is run after self-identification to install any standard packaged software needed to build the OpenCPI software:

- prerequisite packages
- the OpenCPI framework
- the built-in projects in the OpenCPI gitlab repo.

The standard software package repository used may in fact be local on the system or on the local network rather than being internet-based.

These standard packages are generally *globally installed* on the system, and not in any "sand box" only for use by OpenCPI. Thus they are considered default installations of standard software for the platform. If a software package should *not* be globally installed (not be seen or used by other users or other software), we consider that a *prerequisite* package, which requires its own installation script, described next.

Thus if a package requires non-standard/customized patches in order to be used for OpenCPI, it must be installed as a *prerequisite*, *not* as standard packaged software.

Embedded and cross-developed platforms do not usually install software from a software package repository into a global location on the system, but if that is desired or required, such platforms may also have a ***`<platform>`*`-packages.sh`** script.

4.1.1.3 Install Prerequisite Software Packages in the OpenCPI Sand Box

After the installation of pre-built standard packaged software from OS repositories is done (if there are any), the next step is to build and install prerequisite software packages in a directory solely used by OpenCPI (and thus not seen or used by other users or software). Each prerequisite software package has its own installation script, as defined in the Prerequisite Installation Scripts section.

The term ***prerequisite*** here implies a software package that is independently downloaded, usually built-from-source, and installed in the OpenCPI sand box (in the source tree). This is in contrast to the previously discussed packages, installed prebuilt from software package repositories and installed globally, visible and usable by all users and unrelated software.

There are two types of prerequisites, each installed using the same type of script:

1. Platform-specific scripts that support the development requirements of OpenCPI (e.g. compilers, or other required utilities) *for a particular platform*.
2. OpenCPI-standard prerequisites required by some or all platforms, both development and cross-compiled.

The first type requires installation scripts be supplied in the platform's directory, while the second type has installation scripts that are part of OpenCPI. A platform indicates, in its platform definition file (**<*platform*>.mk**) whether it has any such platform-specific prerequisites, and if so, its scripts are run before building any of the second type of prerequisites. This allows the first type (perhaps cross compilers) to enable building the second type (the standard required prerequisites for OpenCPI). All prerequisite installation scripts have the name:

install-<*prerequisite*>.sh

Each such script can build its package in whatever way is needed, which can vary widely, but can rely on previously installed packages.

> When installing a prepackaged OpenCPI binary distribution (e.g. from RPMs using the **yum** command on CentOS platforms), the built prerequisite packages are also installed as part of that RPM installation.

The results of this step appear in the **prerequisites** directory of the OpenCPI source installation.

4.1.1.4 Build the OpenCPI Framework Libraries and Executables

After the standard packaged software is installed, and the platform-specific and OpenCPI common prerequisite packages are built and installed, the OpenCPI framework can be built. This is done as part of the **install-opencpi.sh** script (for development host platforms) or the **ocpiadmin install platform <*platform*>** operation (for embedded platforms).

Building the framework depends on a tool chain declared in the platform definition file, as well as any platform-specific (if any) and OpenCPI common prerequisites.

The results of this step are made available in the **exports/** directory of the source tree, in a subdirectory whose name is the platform. Executables are in **exports/<*platform*>/bin/**, and libraries are in **exports/<*platform*>/lib**.

4.1.1.5 Build the Assets in the Built-in Projects in the OpenCPI git Repository

The final step of building OpenCPI is building all the software assets in the projects that are present in the OpenCPI source tree (from its gitlab repository). The built-in projects are currently **core**, **platform**, **assets**, **tutorial** and **inactive**. This occurs as the last part of the i**nstall-opencpi.sh** script or **ocpiadmin install platform <*platform*>** operation.

This builds all software workers and ACI applications in these projects. This includes performing the equivalent **ocpidev build** command in each project.

When a new software platform is defined correctly, this step represents a successful software platform definition, for building.

4.1.1.6 Test the OpenCPI Installation for a Software Platform

The last step in **`install-opencpi.sh`** is running some tests of the development software platform. This also tests aspects of OpenCPI that are not all related to software platforms, but if it succeeds completely, it is a good indication of success. This script includes some asset unit tests in the built-in projects.

For embedded platforms, the analogous test function is executed on the embedded platform after it has been prepared for OpenCPI (using **`ocpitest`**), which normally occurs after preparing a bootable media (e.g. SD card) for the platform. The **`ocpiadmin install platform <platform>`** operation skips the testing phase.

4.1.2 Software Platform Files in the Platform's Directory

Defining a software platform for OpenCPI involves 4 types of files in its directory, **`rcc/platforms/<platform>`**, in some project:

- The platform definition file ***`<platform>`*`.mk`**
- The platform self identification file for development systems only: ***`<platform>`*`-check.sh`**
- The optional package installation file: ***`<platform>`*`-packages.sh`**
- The optional platform exports file: ***`<platform>.exports`***
- Optional platform-specific prerequisite installation scripts: **`install-`*`<prerequisite>`*`.sh`**

Each type of file is described in the following sections.

4.1.2.1 Platform Definition File: `<platform>.mk`

This required file is processed by **`make`**, and thus uses **`make`** syntax. It contains a set of variable assignments to override default values as required by the platform. The list of valid variables, their default values, and the descriptions, are in the file:

`tools/include/platform-defaults.mk`

Three variables are required, and the rest are only used to override default values. All variables are in “camel case”, with the prefix **`Ocpi`**. The required variables are:

`OcpiPlatformOs` — the operating system name in lower case, e.g. **`linux`** or **`macos`**.

`OcpiPlatformOsVersion` — the major version, usually a short name with a numeric major version. For Linux, it is a prefix before the major version, indicating the distribution, e.g. **`c`** for CentOS, **`u`** for Ubuntu, **`m`** for Mint etc. For macos, the major/minor version, e.g. **`10_13`**.

OcpiPlatformArch — the CPU architecture, usually as returned by the **`uname -m`** command, e.g. **`x86_64`**.

Two other important variables are:

OcpiPlatformPrerequisites — a list of required software prerequisite names for platform-specific prerequisite packages for which the platform will supply installation scripts.

OcpiCrossCompile — an absolute pathname and prefix of the compilation toolchain tools for the platform, assuming packages and prerequisites are installed.

The presence of the **`OcpiCrossCompile`** variable setting indicates a cross-compiled platform. If the cross compiler used is in fact a declared prerequisite for the platform, this make variable definition indicates where its executables are:

```
$(OCPI_PREREQUISITES_DIR)/<prerequisite>/$(OCPI_TOOL_PLATFORM)/bin
```

When this ***`<platform>`*****`.mk`** file is processed, it is checked for variable assignments that are not in the list of valid variables, which results in a warning if variables assigned are not in the list of valid variables. All the default values are as needed for CentOS7 Linux, and thus all assignments (except for the three required ones) are only needed to override those defaults. All variables are initially defined with a default value as "simply expanded" or "immediately expanded" **`make`** variables using the **`:=`** assignment syntax.

A simple example of this file is the current definition for CentOS7. Since the defaults defined in **`platform-defaults.mk`** are basically for this platform, there are not many variables:

```
OcpiPlatformOs:=linux
OcpiPlatformOsVersion:=c7
OcpiPlatformArch:=x86_64
```

For the cross-compiled Xilinx Zynq-7000 Linux platform (**`xilinx13_3`**), from the version **`13_3`** (2013, the 3rd quarter) the platform has a different compiler, and relies on the compiler that is embedded in the Xilinx tools package already installed separately. Its definition file looks like this (with some abbreviations):

```
include $(OCPI_CDK_DIR)/include/hdl/xilinx.mk
OcpiCrossCompile=\
$(OcpiXilinxEdkDir)/gnu/arm/lin/bin/arm-xilinx-linux-gnueabi-
OcpiCFlags+=-mfpu=neon-fp16 -mfloat-abi=softfp -march=armv7-a
OcpiCxxFlags+=-mfpu=neon-fp16 -mfloat-abi=softfp -march=armv7-a
OcpiStaticProgramFlags=-rdynamic
OcpiKernelDir=kernel-headers
OcpiPlatformOs=linux
OcpiPlatformOsVersion=x13_3
OcpiPlatformArch=arm
```

There is no package or prerequisite installation script for this platform since its tool chain is available as a side effect of a separate, global, installation of Xilinx FPGA tools.

This file should not introduce or set any **make** variables not defined in **platform-defaults.mk**. If temporary variables are required, they should use the make **foreach** function if possible (see the **centos7.mk** file for an example).

4.1.2.2 Platform Self-Identification/Check Script for Development Platforms

This script is named ***<platform>*-check.sh** and is required only for development platforms, and is executed with the **bash** shell. It returns success (an exit status of zero) if the running system is indeed this platform. If it is does not determine that the running system is this platform, it should return a non-zero exit status.

Normally each Linux distribution or other operating system has some files that indicate its native distribution or release type, as well as which major version is running. So the task of this script is to check for those files as well as the major version number.

These scripts are usually quite simple, and several are listed here verbatim as examples:

For CentOS Linux platforms, where the **/etc/centos-release** file contains something like:

```
CentOS release 7.8 (Final)
```

the entire script file can be:

```
f=/etc/centos-release
[ -r $f ] && read c r v x < $f &&
  [[ "$c"  == CentOS && "$v" == 7.* ]]
```

For Mint Linux (e.g. version 18), the file is:

```
grep -sq "RELEASE=18" /etc/linuxmint/info
```

For MacOS (any recent version), the file is:

```
[ "$(uname -s)" = Darwin ] && which -s sw_vers &&
  vers=`sw_vers -productVersion |
        sed 's/^\([0-9][0-9]*\.[0-9][0-9]*\).*/\1/' | tr . _` &&
        [ macos$vers = $(basename $(dirname $0)) ]
```

For Red Hat 5 Linux (not currently supported due to old C++ compilers) the following script works, which is careful to avoid the symbolic links to **/etc/redhat-release** that exist on CentOS systems:

```
f=/etc/redhat-release
[ -r $f -a ! -L $f ] && read r x v y < $f &&
  [[ "$c" == R* && "$v" == 5.* ]]
```

Since all scripts are invoked explicitly using the **bash** shell, no execute permission or initial **#!/bin/bash** line is necessary. OpenCPI has an explicit requirement for the **bash** shell on all development platforms.

4.1.2.3 Platform Package Installation Script

The ***<platform>*-packages.sh** script, invoked using **bash**, is required for platforms that use or require standard packaged software, prebuilt and globally installed from a (usually network-based) software package repository. It has two functions:

- install the required software packages from the repository
  —or—
- list the required software packages on standard output

When this script is called with no arguments, it simply uses the appropriate commands to install the required software. Since these packages are installed globally, the installation commands usually require administrative permissions (e.g. using **sudo** in the script).

If the single argument to this script is **list**, then the second, listing function is requested. In the **list** mode, it must list on stdout the names of required packages on 4 lines:

1. The packages necessary for OpenCPI runtime packages/RPMs
2. The packages necessary for OpenCPI development packages/RPMs
3. The packages necessary for an OpenCPI source environment (e.g. for building the framework) beyond what is needed for #1 and #2.
4. The packages that are for development (#2), but which must be installed in a second phase after those on the second line.

In **list** mode, packages are package names suitable for RPM creation. Note that for RPMs, if a required package is *not* the same architecture as the platform (e.g. when a 32-bit package is required on a 64-bit platform), a pathname of a file supplied by the package must be listed rather than the name of the package.

If the first argument to this script is **yumlist**, the same output is required except that it is not limited by the RPM constraint above, and only lists packages to be installed.

The existing scripts for CentOS, Ubuntu, Mint or MacOS platforms are good examples. In general the script is roughly:

```
RPKGS="a b c d" # required packages for runtime
DPKGS="a1 b2 c3 d4" # required packages for development
OPKGS="a5 b6 c7 d8" # required packages for framework development
EPKGS="a9 b10 c11 d12" # Second phase development packages
[ "$1" = list ] && echo $RPKGS && echo $DPKGS && echo $OPKGS &&
                   echo $EPKGS&& exit 0
<install-command> $RPKGS $DPKGS $OPKGS
<install-command> $EPKGS
```

For CentOS and RedHat systems, the ***<install-command>*** is:

```
sudo yum -y install
```

On Debian or Ubuntu or Mint Linux systems, the ***<install-command>*** is:

```
sudo apt install -y
```

On MacOS using the macports package management system, the command would be:

```
sudo port install
```

Any platform-specific software required for OpenCPI must either be installed by this script (for globally installed prebuilt packages from a repository), or using what OpenCPI calls ***prerequisite installations***, which are described next.

Since these scripts are invoked explicitly using the **bash** shell, no execute permission or initial **#!/bin/bash** line is necessary (but not precluded). OpenCPI has an explicit requirement for the **bash** shell on all development platforms.

4.1.2.4 Platform Exports File

This optional file, ***<platform>*.exports**, specifies what files in the platform's directory should be exported and made visible to "users" of the platform. Its syntax is a series of lines specifying files or directories to be exported under three different conditions, indicated by a prefix character at the start of the line, with **+** for development files, **=** for runtime files, and **@** for deployment files (e.g. bootable media, SD card). Thus each (non-comment - using **#**, non blank) line contains three things: a prefix, a source-tree location (LHS), and an exported location (RHS):

```
[+=@] <source-location>  <exported-location>
```

Each line is similar to a "cp" command.

When a file to be exported is in the platform's own directory, the literal string **<platform-dir>** in the LHS is replaced with the actual platform's directory pathname. The LHS string can use wildcards (e.g. ***<platform_dir>*/sub.***).

The second string (RHS) on each line is optional. If it is not present, the file or directory indicated in the first string (LHS) is exported at the top level of the platform's exports.

The RHS string is the location where the LHS file or directory will be exported. A trailing slash on the RHS indicates a directory where the LHS will be placed, which will automatically be created as required.

As a special case, when the prefix is **@** (for deployment), the RHS indicates a location on the bootable SD card (or equivalent). The default location is the root of the SD card, which is equivalent to a slash. On an SD card, the entire OpenCPI runtime package is installed under the **opencpi/** directory.

Much like HDL platforms, and component libraries, the files that are indicated for export are all automatically exported using symbolic links in a **lib/** subdirectory of the platform's directory.

This local **lib/** subdirectory is exported from the project, in the **exports/rcc/platforms** directory of the project.

Finally, depending on the prefix character in the file (**+**, **=**, or **@**), these files are exported by the framework as a whole for the different ways that OpenCPI is packaged (development package, runtime package, deployment/SD-card packages).

Note that the platform definition file, the check file, and the exports file itself are automatically exported and do not need to be mentioned in the exports file.

4.1.2.5 Prerequisite Installation Script

If the **OcpiPlatformPrerequisites** variable is set (not empty) in the platform definition file, an installation script for each listed prerequisite is required to be present in the platform's directory, with the name **install-*<prerequisite>*.sh**. E.g., if the variable assignment is:

```
OcpiPlatformPrerequisites=preqx preqy preqz
```

then scripts named:

```
install-preqx.sh install-preqy.sh install-preqz.sh
```

must all be present.

Prerequisites are installed in an OpenCPI installation directory so that they do not interfere with other global software installations and are thus considered part of the OpenCPI installation (or “sand box”). Both runtime library prerequisites (for any platform) and tool prerequisites (for executing on development platforms) are installed there.

- This directory's default location in the **prerequisites/** subdirectory of the source tree. This may be overridden by setting the **OCPI_PREREQUISITES_INSTALL_DIR** if the default is unacceptable.
- In an RPM OpenCPI distribution (prebuilt, installed globally) it is /**opt/opencpi/prerequisites**.

In a source code installation, prerequisites are built and installed in the source tree itself, in the respective **prerequisites-build/** and **prerequisites/** subdirectories. Thus these packages do not interfere with other global software installations or other OpenCPI installations/versions. Both runtime library prerequisites and tool prerequisites are typically installed here.

When a prerequisite has libraries, the libraries are automatically installed in the directory for framework libraries, e.g. **exports/*<platform>*/lib** in a source build or **/opt/opencpi/cdk/*<platform>*/lib** in an RPM installation.

This script should follow the pattern of other install scripts in the **build/prerequisites/** directory of the OpenCPI framework (e.g. **install-gmp.sh**), in particular:

- Ensure that the **OCPI_CDK_DIR** environment variable is set.
- Source the **$OCPI_CDK_DIR/scripts/setup-prerequisite.sh** script with appropriate arguments.
- Install any resulting platform-independent header files in **$OcpiInstallDir**.
- Install platform-specific files in the **include**, **lib** and **bin** subdirectories of: **$OcpiInstallExecDir**.

- Build both dynamic and static versions of runtime libraries, and build the static libraries with PIC options enabled.
- If cross-building, use the **$OcpiCrossHost** variable as the prefix for tool executables (e.g. as the --host option to ../configure)

The **$OCPI_CDK_DIR/scripts/setup-prerequisite.sh** script, supplied by OpenCPI, which is *sourced,* takes these arguments:

1. The target platform (the platform on which execution takes place). This is passed in from the first argument of the install script, e.g. **"$1"** (quoted to allow empty).
2. The name of the prerequisite, e.g. **gmp**.
3. A short "pretty" description string, e.g. **"Extended Precision Library"**.
4. The URL (or absolute pathname) to download the file from, without the filename, e.g.: **https://ftp.gnu.org/gnu/gmp**
   If the URL ends in **.git**, it will be cloned rather than downloaded and unpacked.
5. The downloaded file name (if a downloaded file), e.g. **gmp-6.1.2.tar.xz**, or, if cloning a git repository, the tag or branch to check out.
6. The top-level directory implicitly created when the download file is unpacked, e.g.: **gmp-6.1.2** or the git repository top level directory. Use a single period (**.**) if the download is a single file, with no implied directory.
7. An indication as to whether the prerequisite should be cross-compiled, or only used on development hosts. A value of **1** indicates runtime and cross compiled for non-development platforms. **0** means only build for development platforms since it is not needed at runtime.

This setup script performs downloading (or git cloning), caching downloads, creating the necessary directories, accessing the tool chain for the platform, and defining convenience functions and variables for use later in the script.

After the setup script is sourced, the environment is setup as follows:

- the current working directory is a newly created platform-specific build subdirectory created under the directory created by the download/unpack, or git clone
- the **OcpiInstallDir** and **OcpiInstallExecDir** variables are set to where the results of the build should be installed
- the **relative_link** function is defined to create appropriate relative symbolic links from the installation directory to this build directory.
- the shell option to terminate on any error (**set -e**) is set.
- variables for explicit (non-autotools) compilation are set compatibly for cross compilation: **CC**, **CXX**, **LD**, **AR**.
- the **OcpiCrossHost** variable is set appropriately for the autotools **--host** option.
- the cross-compilation tools are in the execution **PATH** environment variable.

Note that the current working directory is a new directory created for the platform for which the prerequisite is being built. This enables different builds for different platforms to avoid conflict with each other. However it assumes that the build commands that are executed after sourcing the **`setup-prerequisite.sh`** script *do not* modify files in other parent directories. If the prerequisite's compilation scheme does not allow for build results to be put in a separate "build" directory, then a copy of the sources must be made into this newly created platform-specific directory. An example of this is in the **`install-gpsd.sh`** script in the source tree.

At this point, the prerequisite installation script can perform an appropriate build in this directory. When the build is complete, the results must be installed (usually using **`relative_link`**) in the directories: **`$OcpiInstallDir`** and **`$OcpiInstallExecDir`**. Depending on how the prerequisite package is natively built, some knowledge of its own build system may be required, such as **`autotools`** or **`scons`** or **`cmake`**. Normally the last few lines of the script are similar to the recommended source installation for the package.

This installation happens one of two ways. If the build uses the typical ***autotools*** paradigm of:

```
../configure; make; make install
```

then the **`--prefix`** and **`--exec-prefix`** arguments are provided to the **`configure`** script as, e.g.

```
../configure
  --prefix=$OcpiInstallDir
  --exec-prefix=$OcpiInstallExecDir
```

This will cause the resulting files to be installed in the correct locations (e.g. **`$OcpiInstallDir/include`** for portable headers, and **`$OcpiInstallExecDir/(lib|bin|include)`** for platform-specific files).

An example script for a prerequisite needed only on most development platforms, is the **`patchelf`** tool. The entire install script is:

```
version=0.9
dir=patchelf-$version
[ -z "$OCPI_CDK_DIR" ] &&
  echo Environment variable OCPI_CDK_DIR not set && exit 1
source $OCPI_CDK_DIR/scripts/setup-prerequisite.sh \
       "$1" \
       patchelf \
       "ELF file patching utility" \
       http://nixos.org/releases/patchelf/$dir \
       $dir.tar.gz \
       $dir \
       0
../configure --prefix=$OcpiInstallDir \
             --exec-prefix=$OcpiInstallExecDir \
             CFLAGS=-g CXXFLAGS=-g
make && make install
```

For prerequisite packages that are not set up for autotools building, the results of the build can be simply installed using the **relative_link** function which operates as a smarter symbolic link command analogous to the **ln -s** command. Here is an excerpt from a prerequisite installation where the software package being installed consists of a single source file:

```
[ -z "$OCPI_CDK_DIR" ] && echo Environment variable OCPI_CDK_DIR not
set && exit 1
source $OCPI_CDK_DIR/scripts/setup-prerequisite.sh \
       "$1" \
       inode64 \
       "fix for 32 bit binaries running on 64-bit file systems" \
       https://www.tcm.phy.cam.ac.uk/sw \
       inode64.c \
       . \
       0
...
gcc -c -fPIC -m32 ../inode64.c
ld -shared -melf_i386 -o inode64.so inode64.o
relative_link inode64.so $OcpiInstallExecDir/lib
```

For runtime prerequisite packages that need to be cross compiled, the **OcpiCrossHost** variable is used in the **../configure** command, e.g.:

```
../configure --prefix=$OcpiInstallDir --exec-prefix=$OcpiInstallExecDir\
             ${OcpiCrossHost:+--host=$OcpiCrossHost}
```

With prerequisites that are not set up for autotools, the compilation commands can be used directly, e.g.:

```
$CXX -c *.c; $AR -rs foo.a *.o
```

Note that the variables set or defaulted in the platform definition file (i.e. ***<platform>*.mk**) are available to be used here.

4.1.2.6 The Makefile for a Software Platform

A software platform may have a Makefile in its directory if it needs to perform any processing steps to create the necessary files prior to building OpenCPI. This Makefile is automatically used (with the default make goal/target), prior to building OpenCPI itself with the **build-opencpi.sh** script that is used by the **install-opencpi.sh** or **ocpiadmin install platform *<platform>*.**

This Makefile is used when files need to be brought into the directory from external sources or downloads. An example is when the files needed have license constraints or are based on installing licensed software and can thus not simply be copied into the public repository.

4.1.3 Summary for Enabling Development for a New GPP Platform

- Create the platform's directory in a project's **rcc/platforms/** directory.
- Create the platform definition file.
- For development hosts, create the ***<platform>*-check.sh** script.

- If needed, create the **`<platform>-packages.sh`** script.
- If needed, create the package-specific prerequisite installation scripts.
- If needed, create a local **`Makefile`** for the platform.
- Build the OpenCPI prerequisites to ensure all the scripts are functional.
- Build the OpenCPI framework and make adjustments to the code for issues that arise.
- Successfully run **`./install-opencpi.sh`** on the development host.
- Successfully run **`ocpiadmin install platform <platform>`** for cross-developed platforms.
- Successfully run **`ocpitest`** on embedded hosts.

4.2 Enabling Execution for GPP Platforms

The OpenCPI framework libraries and command-line tools are built using the appropriate compiler as installed above. Once the development host has been enabled, and the OpenCPI core and example components have been built, a few additional steps are required to enable *execution* on the platform.

Several OpenCPI runtime libraries have aspects that use conditional compilation depending on the system or CPU being targeted. The current OpenCPI core libraries have significant Linux dependencies and in several files there is conditional code between Linux and MacOS (such as low level networking details). There are a few areas that have conditional code depending on the CPU being used, such as realtime high resolution timing registers in the CPU. These customizations are not well defined. For new CPU architectures and new operating systems not based on Linux, the code must be examined for these issues, which will typically cause compilation errors.

Setting up a development system for execution is nearly automatic once the build environment is set up and any required code changes in the OpenCPI runtime libraries. For execution on development systems, using the normal installation steps described in the OpenCPI Installation Guide, are sufficient (i.e. using the source **`cdk/opencpi-setup.sh -s`** script).

Loading the OpenCPI kernel driver (via the **`ocpidriver`** command) is necessary when accessing other platforms via the system bus. If the GPP platform has no such bus/fabric, or there are no OpenCPI-capable cards plugged in, the kernel driver is not necessary.

For embedded systems, the setup is more customized.

4.2.1 Creating a Deployment Package (Bootable Image or SD card) for an Embedded Platform

A “runtime” package for an embedded platform is a set of files which can be installed on a network file server (usually the development host), which can be accessed with the embedded system acting as a file client. The next step to achieving runtime is to install some appropriate files on the embedded system itself. We call this set of files a “deployment package”.

A deployment package is created for either an embedded software platform by itself, or a combination of an embedded software platform and an HDL platform. This combination is common on SoCs like Zynq and CycloneV.

The exports file defined above in the platform export file section above specifies the files that will be placed on the SD card for this software platform. When the standard operation to create SD cards is run according to the installation guide (using the **`ocpiadmin deploy platform`** operation), the following steps are taken:

1. Copy the standard OpenCPI runtime files for the platform to the SD Card
2. Copy the files designated with @ in the software platform exports file
3. Copy the files designated with @ in the HDL platform's exports file.

This combination of exports from the two platforms creates the content of the bootable SD card. Notice that the HDL platform is exported last, and thus can specify files that override files copied by the software platform.

Deployment packages are thus created by combining the appropriate exports from the software and HDL platforms to create the necessary files for the “system” that combines that software platform with the HDL platform.

The files on the bootable image can fall into two categories:

- The minimum set of files necessary to run OpenCPI assuming it has access to a runtime package on a network server.
- The additional files necessary to run OpenCPI standalone, i.e. with no network, where the runtime files are on the card.

Thus for platform exports, the **+** lines indicate platform-specific exports for the development package (not used at runtime, only on development host), **=** indicates platform-specific exports for the runtime package (used on development host as server or directly on the SD card), and **@** indicates platform-specific exports for a deployment package (SD card only).

First, in the normal course of building OpenCPI for the targeted platform, the software platform would be already be built by running:

```
$ ocpiadmin install platform xilinx13_3
```

and then the built-in projects, including those containing the HDL platform, would be built with:

```
$ ocpiadmin install platform zed
```

So then the particular command to create the deployment package would simply be:

```
$ ocpiadmin deploy platform zed xilinx13_3
```

When **ocpiadmin deploy platform** is run, the bootable media directory tree is placed in this directory in the source tree:

cdk/*<hdl-platform>*/sdcard-*<sw-platform>*/

This directory can then be copied to an SD card for use on the platform.

5 Enabling FPGA Platforms

Whereas GPP platforms have operating systems and drivers that OpenCPI uses to interface with the hardware surrounding the processor, FPGA platforms do not. OpenCPI provides a runtime infrastructure on FPGAs, which is *roughly* analogous to an embedded operating system, and which requires FPGA "driver" logic that is specific to the platform and its attached devices. Thus the OpenCPI infrastructure combined with the required platform-specific logic defined below, is analogous to the combination of an embedded OS and a "board support package and drivers" for a typical embedded GPP.

Consistent with the definition of ***platform*** given earlier, here we more narrowly define an FPGA ***platform*** as a particular, single FPGA on some hardware board. If a board has multiple OpenCPI-usable FPGAs, each is a platform and each may host a container in which components (actually: *worker instances*) execute. An OpenCPI-usable FPGA is one that can host user-written workers executing in an OpenCPI HDL container.

An FPGA simulator may also be an FPGA platform, which is also a place where OpenCPI HDL workers may execute, with mostly the same infrastructure as physical FPGA platforms.

When an SoC, like the Xilinx Zynq-7000 chip, contains a section of FPGA logic as well as processor cores, the FPGA part is an FPGA platform and the GPP processor core(s) are a GPP software platform. Thus the SoC is indeed a "system on chip": a system with two platforms and an interconnect between them.

Analogous to preparing the support for a GPP platform, enabling an FPGA platform involves steps to enable the development environment, and steps to enable the run-time/execution environment.

Enabling development targeting a platform involves:

- Installing and integrating a development tool chain that can target the FPGA device on the platform (when one is not already installed that also supports the new platform).
- Verifying that the integrated tool chain can process and build all the core OpenCPI FPGA code and portable HDL workers when targeting the FPGA platform's part and part family.

Enabling runtime execution on a non-simulation platform involves:

- Writing specific new VHDL code that supports the particulars of the hardware attached to the FPGA on the platform.
- Updating software drivers to load/unload configuration bitstreams.
- Verifying that the various platform-independent FPGA test applications execute on the platform.

5.1 Physical FPGA Platforms

Platforms are specific FPGAs on a board connected locally to:

- Local I/O devices: ADC, DAC, DRAM, Flash, GbE, etc.
- Interconnects: PCIe, AXI, Ethernet, etc. used to talk to other OpenCPI platforms
- Slots: FMC, HSMC, mezzanine card slots.

For example, a Xilinx ML605 board has PCI Express interconnect, DRAM, and 2 FMC slots (and other minor devices).

For each device on a platform, specific development may be required (described in the Device Development section below). In many cases existing device support may be reused, since OpenCPI FPGA device support is typically done in a way that is sharable across platforms.

Ideally, new device support is developed such that it can be reused across platforms.

Physical FPGA platforms are based on a particular type of FPGA chip: e.g., a Xilinx ML605 development board has a Virtex6 FPGA (xc6vlx240t), with a speed grade and a package.

Examples of physical FPGA platforms are:

Table 2: Example FPGA/HDL Platforms

Board	Part	Interconnect(s)	Description
`ML605`	Virtex6	PCIe	Xilinx PCIe-based Virtex6 development board, with 2 FMC slots
`ZedBoard`	Zynq-7000/PL	AXI internal	Digilent Zynq-7000 Development Board HDL platform is the *"PL" side of the Zynq SoC*, with the PL-attached devices and one FMC slot
`ALST4`	Stratix4	PCIe	Intel/Altera PCIe-based Stratix4 development board with 2 HSMC slots.
`ZC706`	Zynq-7000/PL	AXI Internal PCIe external	Xilinx PCIe-based Zynq development board with 2 FMC slots. HDL platform is the *"PL" side of the Zynq SoC*, with the PL-attached devices, two FMC slots, and an attachment to the PCIe interconnect.

5.2 Simulator FPGA Platforms

OpenCPI provides a software runtime infrastructure to make execution on simulators as similar as possible to execution on physical FPGAs, without simulating any external device-related logic. The simulation execution environment makes execution on different simulators also similar to each other. Multiple simulator instances may execute simultaneously subject to any license restrictions that would allow only a certain number of simulator instances to run at the same time.

Only mixed-language simulators (VHDL and Verilog) may be enabled and used with OpenCPI. These are simulators that support mixing VHDL and Verilog modules in the same design.

Examples of supported FPGA simulators include:

- Xilinx Isim from ISE 14.7
- Mentor Modelsim DE 10.2
- Xilinx Vivado Xsim 2017.1 and 2019.2

Other simulators that could be supported in the future include:

- Aldec
- Cadence

Under OpenCPI, simulators perform co-simulation: the HDL workers inside the simulator (in the HDL container being simulated), can communicate with RCC workers executing outside the simulator. This communication path (from inside to outside of the simulator) is supported on all these HDL simulators.

5.3 Enabling Development for FPGA Platforms

5.3.1 Installing the Tool Chain

For tools not currently supported by OpenCPI, document the basic process of obtaining and installing the tools, highlighting any options or configurations that must be specialized, customized, or simply required for using OpenCPI. Licensing is also an issue for many FPGA tools. See the ***OpenCPI Installation Guide*** sections on installing FPGA tools for examples.

Note that OpenCPI executes FPGA tools in “wrapper scripts” that perform any necessary initialization or setup, including license setup. Thus, for OpenCPI, there is no need for login-time startup scripts. In fact, such scripts can actually cause problems in many cases since OpenCPI frequently invokes multiple alternative tool sets under a single build command. Polluting your environment with settings from multiple tools and vendors is frequently a source of problems.

For OpenCPI development, it is recommended to remove any such “automatic setup at login” items that the tools installation process inserts into your login script(s) and put them in an a separate script that is used as needed.

Some FPGA tool chain installations include a software tool chain for embedded GPP cores on SoCs. This means that one tool installation supports both the FPGA platform and the GPP platform. E.g. a Xilinx ISE installation may include the EDK (Embedded Development Kit) sub-package that supports cross-compilation for the ARM cores on the Zynq SoC.

5.3.2 Integrating the Tool Chain into the OpenCPI HDL Build Process

For a completely new FPGA tool chain, the integration is somewhat difficult and not well documented. It relies on internal knowledge of the OpenCPI build system.

This process includes enabling the OpenCPI FPGA build process to use the right tools to target the “part family” of the FPGA device on the platform. For example, on the “zed” platform, the family of the FPGA part is “zynq”. On the “ml605” platform, the family of the FPGA part is “virtex6”. The required tools for a platform's FPGA may already be installed and integrated, and may already support the particular part family of the platform's FPGA. If not, that support must be added to the integration of those tools.

So the integration process is:

1. If the required tools are not currently integrated with OpenCPI, they must be integrated (not a small job).
2. If the required tools are currently integrated with OpenCPI, but do not yet support the part family, that support must be added.

The collection of OpenCPI-supported part families, and their relationships to tools and vendors, is in the file:

`tools/include/hdl/hdl-targets.mk`

The scripts that wrap and execute FPGA tools are found in the directory:

tools/include/hdl

5.3.3 Building All the Existing Vendor-independent HDL Code

Nearly all HDL code (mostly VHDL) in OpenCPI is portable and can be built (compiled and synthesized) for all part families and vendors and simulators. This first step avoids building platform-specific code. Supplying the targets (part families) is required.

Here is a command (run from the top level of OpenCPI) that builds all the portable code in OpenCPI for currently supported part families (known as "**HdlTargets**" in the OpenCPI FPGA build process):

```
make hdlportable \
  HdlTargets='isim modelsim xsim virtex6 virtex5 \
              stratix4 stratix5 zynq zynq_ise spartan3adsp'
```

It of course assumes *all* the required tools are installed.

This build command builds all primitive libraries and cores as well as all HDL workers in the OpenCPI core tree. It stops short of building anything specific to an HDL platform.

Some of the code built using the above command is explicitly labeled to ***only build*** for certain targets or to ***not build*** for some targets, but most is truly portable and will build for all targets. Once this build command succeeds for the target (part family) of the new platform, you can proceed with the steps below to write the HDL code necessary to enable execution on the platform.

5.3.4 Scripts for HDL Platforms

For physical platforms (not simulation) which require bitstream loading via JTAG, or power-up flash, there are two scripts that must be written and placed in the platform's directory, *if they apply to the platform*:

- A JTAG support script to enable JTAG-based bitstream loading, whose name is: **jtagSupport_*<platform>*** (only for platforms with JTAG bitstream loading)
- A boot-flash loading script to enable scripted loading of bitstreams to the boot flash, whose name is: **loadFlash_*<platform>*** (for platforms with boot-flash loading capabilities)

These scripts typically wrap and call vendor-specific tools for these purposes.

In some cases the appropriate script might already be written for a different platform, in which case one platform's script can be a symbolic link to the other. In other cases there might be a vendor script (e.g. for all Xilinx or all Intel/Altera) which may live in the **runtime/hdl-support/*<vendor>*** directory.

For simulation platforms there must be a script to invoke the simulator from the OpenCPI runtime framework. This script must be called: **runSimExec.*<platform>*** and live in the simulation platform's **hdl/platforms/*<platform>*** directory.

5.4 Enabling Execution for FPGA Platforms

This section assumes the reader is familiar with component and application development with OpenCPI, including developing HDL application workers and assemblies as described in the ***OpenCPI HDL Development Guide***.

An OpenCPI HDL hardware platform is an FPGA with associated devices and slots attached to its pins. Supporting a platform includes determining whether the types of devices and slots attached to the FPGA are already supported by OpenCPI.

If the devices attached to the FPGA are not yet supported in OpenCPI, that support must also be added. Device support in OpenCPI is generally portable (i.e. the device support code can be used to support the same device on different platforms and cards). This type of device support is done separately from a platform or card so it is easily reused on other platforms or cards. Some device support is very platform-specific and is associated with a particular platform. The device support process is described in the Device Support for FPGA Platforms section below.

If a platform has slot types that are not yet supported, that support must be added. Slot types are defined by specific physical connectors, electrical signaling and direction, and pin and signal name assignments. See the section Slots — How Cards Plug into Platforms below.

The term ***card*** is used in OpenCPI to mean a card with additional devices that may be plugged into a compatible ***slot*** on various ***platforms***. Thus devices may be directly attached to the pins of the platform FPGA, or they may exist on a plug-in card that, when plugged into a slot, become attached to the platform FPGA. In this latter case, such devices are not considered part of the platform, but part of the ***card***, which might be plugged into a certain type of slot on any platform. See the section Defining Cards Containing Devices that Plug into Slots of Platforms

The asset types in a project that supports one or more HDL platforms are:

- ***HDL Device Worker*** — a specific type of HDL worker that supports external devices attached to FPGAs
- ***HDL Platform Worker*** — a specific type of HDL device worker providing infrastructure for implementing control/data interfaces to devices and interconnects external to the FPGA or simulator (e.g. PCI Express, Clocks)
- ***HDL Slot Type Definition*** — a specification of the pins and signals that are present in a type of slot that may be present on platforms, into which compatible cards may be plugged.
- ***HDL Card Definition*** — a specification that includes the slot type of a card, the devices present, and how they are wired to the slot.
- ***HDL Platform Configuration*** — a prebuilt (pre-synthesized) assembly of device-level HDL workers that represent a particular configuration (subset) of device support modules for a given HDL platform. The HDL code is *automatically generated* from a brief description in XML.

All these asset types are created (the required directories and files created), and then compiled/built using the **`ocpidev(1)`** tool.

The directories (in any project) where platform support files are placed (by **`ocpidev`**) are:

`hdl/devices` — a component library for portable device workers and proxies

`hdl/cards` — a component library for card-specific device workers

`hdl/cards/specs` — a directory where cards and slot types are defined

`hdl/platforms` — a directory where platform workers and associated platform configurations are placed

`hdl/platforms/<platform>` — a platform worker's directory that also contains its platform configurations

h**`dl/platforms/<platform>/devices`** — a component library for platform-specific device workers and proxies for ***`<platform>`***, which are generally not visible outside the project or from other libraries (including specs)

HDL platform support starts with deciding on a name for the platform (usually a lower-cased version of the name used by the vendor of the platform), and using **`ocpidev create hdl platform`** to create a directory and associated files using the **`ocpidev`** tool. In that directory there will be an initial platform description XML file and an initial source code skeleton for the platform worker.

In summary, the steps to enabling an HDL platform for execution are, (assuming the development tools are enabled):

- Take inventory of the platform by identifying its interconnects, method of reprogramming, devices, and slots
- Define the platform worker, with associated hardware-related metadata and files.
  - Address/configure interconnect issues of the platform worker
  - Build the platform worker (perhaps a skeleton) and a simple complete bitstream
- Implement reprogramming (a.k.a. bitstream loading) of the FPGA
- Implement the platform worker and perform basic tests, with *no* devices enabled.
- Define any new slot types and add them to OpenCPI.
- Define and establish “skeletons” for all new devices on the platform.
- Define the platform with all devices and slots, build and do basic tests.
- Implement any new devices for the platform, and test the platform *with* devices (both pre-existing and new).
- Test any slots using supported cards.

The assets created for new platforms are usually done in a separate OpenCPI project, called an ***OpenCPI System support Project*** (OSP). But assets that are shareable by multiple platforms should ultimately result in gitlab merge requests for the built-in

projects that are part of OpenCPI. There is an OpenCPI built-in project called **platform**, which contains assets used for supporting different platforms (common devices, cards, etc.). Assets shared by multiple platforms should live there.

The next two sections (signals and slots), define XML elements required by platforms. Following that, the specifics of platform worker XML files are detailed.

5.4.1 Signal Declaration XML Elements for Devices, Platforms, Slots and Cards

Signal declaration XML elements are used in a number of contexts in supporting HDL platforms, cards and devices. They declare name, direction, width and other characteristics. Any specific rules or constraints for each context are specified in the respective sections, but the common aspects of signal declarations are described here.

An example of a signal declaration element is:

```
<signal name='data' direction='out' width='16'/>
```

which defines an output signal array 16 wide named **data**. The attributes to a **signal** element are: **direction**, **differential**, **width** and **pin**. Depending on the direction of the signal, there are other attributes that determine the actual signal names.

5.4.1.1 Name Attribute of Signal Element

The **name** string attribute provides the signal name and should comply with typical identifier syntax (leading alphabetic, then alphanumeric or underscore). Signal names are case insensitive.

5.4.1.2 Direction Attribute of Signal Element

The signal **direction** attribute is an enumerated type with the following choices:

in — identifies a signal as an input

out — identifies a signal as an output

inout — identifies a signal as a tristate signal

bidirectional — identifies a signal as usable in either direction

unused — identifies a signal as unused in the current context

The direction is relative to the asset being defined in the XML file (device, platform or card). When defining slot types, the direction is relative to the platform FPGA. In a platform definition, **in** means input to the platform worker. For a device, it is input to the device.

The **direction** attribute for device and platform signals may be an expression based on the worker's parameter properties. This allows a signal to take on different directions based on other configuration information. In this context, the expression may determine that the signal is **unused** for some configuration parameter values. For example:

```
<signal name='data' width='16' direction='mode==2 ? out : unused'/>
```

This indicates that when the worker's **mode** property has the value 2, the **data** signal is an output, otherwise it is **unused**.

For a *slot* signal, declaring **bidirectional** means that its direction is determined by the direction of the card's device worker signal which uses it. Consequently, different cards may implement unique directionality for a bidirectional slot signal. Note that slot signal directions **in**, **out**, or **inout** impose a requirement for all possible cards/device workers which use that signal.

For a *device* signal, declaring the direction to be **bidirectional** merely defines an HDL *inout* port on the device worker, and it is expected that the device worker will instance an I/O buffer itself, from which the tools will determine the direction.

For a *device* or *platform* signal, the direction **unused** indicates that it need not be connected since it is nonfunctional in a particular configuration.

Declaring the direction of a device signal to be **inout** defines the 3 tristate signal ports on the device worker. An **inout** signal inside an FPGA implies a bundle of three signals (in, out, output-enable) which, when attached to a pin/pad of the FPGA may result in a single tristate signal external to the FPGA. The names of the three associated signals is determined by adding the suffixes: **_i**, **_o**, **_oe** respectively. These default suffixes can be overridden for a signal using the **in**, **out**, **oe** attributes, where **%s** in these attributes represents the name in the **inout** attribute. For example, this declaration:

```
<signal name='mySig' direction='inout' oe='The%sEnable' in='%s_in'
        out='%sDriven'/>
```

would imply the three signal names: **mySig_in**, **mySigDriven**, and **ThemySigEnable**, rather than the default: **mySig_i**, **mySig_o**, and **mySig_oe**. These suffixes will be converted to upper case when the signal name is entirely upper case.

Whether this bundle of three signals is implied for **inout** signals depends on the context of the **signal** element. When HDL containers are generated by OpenCPI, a tristate I/O pin is generated with the single external (pin) signal and the three internal signals.

These direction and signal type attributes allow OpenCPI to perform error checking for connections and implement the correct tie-offs and I/O primitives for top-level signals in a design.

5.4.1.3 Width Attribute of Signal Elements

This attribute specifies that the signal is an array of signals with the width specified in the value of the attribute. Each element of the array is an individual signal with a zero-origin index enclosed in parentheses. When VHDL or Verilog code is being generated, the array is defined and used appropriate for the language. Thus if the signal is defined as:

```
<signal input='data' width='3'/>
```

The signals are **data(0)**, **data(1)**, **data(2)**. The **width** attribute may be an expression, which may use other parameter properties. For example:

```
<signal name='data' direction='in' width='mode==2 ? 4 : 2'/>
```

This indicates that when the worker's **mode** property has the value 2, the **data** signal has a width of 4, otherwise its width is 2.

5.4.1.4 Differential Attribute of Signal Elements

This boolean attribute specifies, when true, that the signal is differential and represents a pair of signals with the suffixes **p** and **n** representing the positive and negative of the pair respectively. These default suffixes can be overridden using the **pos** and **neg** attributes, where **%s** in the value of those attributes represent the name defined in the direction attribute.

For example, this declaration:

```
<signal name='mySig' direction='in' differential='1' pos='P_%s'
        neg='%sN'/>
```

would imply the two signal names: **P_mySig** and **mySigN** rather than the default: **mySigp**, and **mySign**.

These suffixes will be converted to upper case when the signal name is entirely upper case. It is invalid to specify an **inout** signal as **differential**.

5.4.1.5 Pin Attribute of Signal Elements

This boolean attribute specifies, when true, that the signal is a pin signal, which implies that it is truly a signal on the pin of an FPGA. This distinction has to do with IO buffers on FPGAs. Most device worker signals are not pin signals since the IO buffers built into the FPGA are only instanced when the final FPGA synthesis is performed, and thus such signals are on the inside of the IO buffers rather than outside. If a device worker specifically instanced IO buffers in the device worker code, then the device worker's signals would be declared as pin signals since they were on the outside of the IO buffers (and thus directly attached to the pins of the FPGA).

[diagram needed]

The OpenCPI code generator for containers and cards and devices needs to know this distinction to do its job correctly.

5.4.2 Slots — How Cards Plug into Platforms

As mentioned earlier, platforms can have ***slots***, which are an intrinsic part of the platform, and enable cards to be plugged in that add devices to the platform. Such cards may be plugged in to any platform that has compatible slots. Slot types are defined independently and then used when describing platforms and cards. A platform has slots of defined types, and cards which are designed for the same slot type may be plugged into the defined slots on that platform. A common slot type is the FMC (FPGA Mezzanine Card), which is defined by the VITA standards organization as VITA-57.1. In fact this standard defines two slot types: FMC-LPC (Low Pin Count using a connector

with 160 pins), and FMC-HPC (High Pin Count using a connector with 400 pins). These slot types are already defined in OpenCPI.

So before platforms or cards are defined, slot types must be defined. Then a platform or card definition refers to slot types of known predefined types. Defining new slot types generally does not involve writing HDL code; just writing descriptive metadata in XML.

A slot type is defined in an XML file placed in the **hdl/cards/specs** directory of an OpenCPI project by **ocpidev**. The name of the file is ***<slot-type>-slot*.xml**, where the ***<slot-type>*** must be a name that can be used is programming languages (i.e. use underscores rather than hyphens), but is otherwise case insensitive. The name should normally be the exact name used in whatever standard document defines the slot type.

The slot type definition file has a top-level XML element **SlotType**, with an optional **name** attribute that must match the name of the file without the **.xml** suffix. This top-level element contains **signal** child elements as defined above in Signal Declaration XML Elements for Devices, Platforms, Slots and Cards. When a pin in a slot type is defined to be used in either direction, it should be declared **bidirectional**.

A slot type is associated with one or more connectors, and the pins of the connectors are numbered. When a slot type is defined, each pin of each connector is given both a physical pin identifier as well as a signal name. OpenCPI uses only the signal names, although it is useful to put the pin identifiers in a comment next to each signal definition.

The direction of signals in a slot are specified relative to the platform side of the slot (sometimes called the *carrier* side or the *motherboard* side). Thus if the signal is sourced on the card (*output* from the card), it is *input* to the platform. Thus such a slot signal is defined as an *input* signal for the slot type.

Signals defined as **inout** in a slot type do not imply the three separate signals that are implied for such signals when *inside* an FPGA. The slot type signal is singular and associated with a single pin.

An example small slot type XML file would contain:

```
<SlotType name='myslottype'>
  <signal name='present' direction='in'/>          <!-- Pin K1 -->
  <signal name='util0' direction='bidirectional'>  <!-- Pin K4 -->
  <signal name='data' direction='out' width='4'/>  <!-- Pin K5 -->
</SlotType>
```

The name of a slot type is used in the XML description files of platforms and cards.

5.4.3 Creating the HDL Platform

An HDL platform is created using the **ocpidev create hdl platform** command, which establishes the platform's directory as **hdl/platforms/*<platform>*** in a project and creates these files:

- the ***<platform>*.xml** describing the platform, called the OpenCPI HDL Platform Description (OHPD) XML file

- an initial VHDL source file ***<platform>*.vhd** for this platform worker.

There are several other types of files that may be manually added in this directory, which include:

- the **<platform>.exports** file, which identifies the files that are needed to *use* the platform.
- constraints files (e.g. .xdc files if using the Vivado tool set), to support building final bitstreams for the platform.
- platform configuration XML files, which define various device configurations for the platform.

These files are defined in more detail in the following sections.

An HDL platform is implemented by a HDL platform worker. The platform worker is a special type of "device worker", with extra requirements for the platform. Developing HDL device workers in general is the subject of the next major section Device Support for FPGA Platforms below, but information specific to platform workers is described here.

5.4.4 The OpenCPI HDL Platform Description (OHPD) XML File

The OHPD file that defines an HDL platform is named the same as the platform name with an "**.xml**" suffix. It does double duty as both a normal HDL device worker's OpenCPI Worker Description (OWD) file, but also containing additional information that is unique to its role in describing the actual HDL platform. The HDL device worker aspects are fully described below in XML Metadata for Device Workers. This section describes the extra XML elements for HDL platform workers.

Thus the OHPD is an OWD for the platform worker, *with additional information.*

The OHPD has these special aspects, beyond an HDL device worker's OWD, all described in detail below:

- The top level XML element is "**HdlPlatform**"
- The spec being implemented is "**platform-spec**".
- The slots present on the platform are indicated by **slot** elements
- The devices physically present on the platform are indicated by **device** elements.
- Special platform port types for platform workers are: **metadata**, **timebase**, **cpmaster**, **sdp**, and **unoc**.

So as a minimum, an example of the OHPD for **myplat** would be:

```
<HdlPlatform Language='vhdl' spec='platform-spec'>
  <specproperty name='platform' value='myplat'/>
</HdlPlatform>
```

5.4.4.1 *Top-level Attributes of Platform Workers*

HDL platform workers can have top-level attributes defined for all workers, as described in the ***OpenCPI Component Development Guide***, such as **`spec`**, **`language`**, **`version, libraries`** and **`controlOperations`**. They can also have top-level attributes defined for HDL workers as described in the ***OpenCPI HDL Development Guide***, such as **`dataWidth`** (not applicable here), and **`cores`**.

Platform workers have several additional top-level attributes: **`part`**, and **`configurations`**. The are described in the following table:

Table 3: Top-level Attributes of HDL Platforms

Attribute	Description
`part`	Required. The FPGA part used in this platform, using the vendor-neutral format: ***`<part>-<speed>-<package>`***. This part must already be defined in the **`hdl-targets.mk`** file in the OpenCPI framework under **`tools/include/hdl/`**. An example for the **`zed`** platform (ZedBoard) that uses a Xilinx Zynq-7000 part: is **`part='xc7z020-1-clg484'`**. Another example is the **`alst4`** platform based on an Altera Stratix4 part: **`part='ep4sgx230k-c2-f40'`**.
`configurations`	The list of platform configurations that should be built for this platform. Each mentioned name indicates the presence of a corresponding XML file for the platform configuration in the platform's directory. If not specified, the **`base`** platform configuration (with no devices) will be built. See the Platform Configurations section below.

5.4.4.2 *Properties of Platform Workers*

Several platform properties are parameters (constants) that must be set in the OHPD using the **`specproperty`** element with a **`value`** attribute:

`platform` — required string property must be set to the name of the platform

`sdp_width` — **`uchar`** property, must be set in platforms using the SDP (see SDP Interface for Interconnect Support below)

`sdp_length` — **`ushort`** property, must be set in platforms using the SDP

`nSwitches` — attribute is set the number of general purpose switches available, default is zero

`nLEDs` — set to the number of general purpose LEDs available, default is zero

`nSlots` — set to the number of slots on the platform, default is zero

An example setting these constants is:

```
<HdlPlatform Language='vhdl' spec='platform-spec'>
  <specproperty name='platform' value='myplat'/>
  <specproperty name='nLEDS' value='4'/>
</HdlPlatform>
```

Some predefined properties must be supported by the HDL code in the platform worker:

`switches` — volatile **`ulong`** property returning the state of the switches; switch 0 is the LSB, driven by the platform worker as **`props_out.switches`**.

`LEDs` — a writable **`ulong`** property to control the LEDs of the platform; LED 0 is the LSB, driven into the platform worker's **`props_in.LEDs`**.

`slotCardIsPresent` — a volatile **`bool`** array indicating whether a card is present is each slot.

Since the platform worker is like any other worker, it can define any of its own properties using the **`property`** element in its OWD.

There are several properties defined in **`platform-spec.xml`** that must be connected in the platform worker's VHDL code, as described below in Writing the Platform Worker Source Code.

5.4.4.3 Platform Ports in a Platform XML File

Platform workers have ports that are different than ports of application and normal device workers. They allow the platform worker to provide required services to the rest of the HDL infrastructure. How they are used or referenced in the platform worker's source code is described in Writing the Platform Worker Source Code.

A ***metadata access port*** must be present in all platform workers. The following line must be present and enables access to bitstream metadata via the platform worker's properties.

```
<metadata master='true'/>
```

A ***timebase output port*** must be present in all platform workers. The following line must be present and enables the platform worker to provide timekeeping signals to the rest of the infrastructure.

```
<timebase master='true'/>
```

The platform worker must arrange for control plane access which provides off-chip access to the on-chip control plane. This can be accomplished in two ways, direct and indirect.

To *directly* support a ***control plane master port***, the platform XML declares:

```
<cpmaster master='true'/>
```

This indicates that the platform worker will provide an addressable path from the controlling processor's software into the FPGA, using the **`cpmaster`** port signal protocol. This usually involves adapting an address window on a software addressable bus that the FPGA is connected to, to the protocol and signals of the **`cpmaster`** port. This normally implies that the external pathways (interconnects) for control and data are separate. See the Control Plane Master Interface for more details on this interface.

A platform can *indirectly* support a control plane by providing an *interconnect* port that serves both as a control and data plane access path. This is common when there is a single connection between the system bus and this FPGA and the FPGA is both a slave

on this bus (for control and possibly data) as well as a master (for DMA data). The absence of a **cpmaster** element in the platform XML implies that the interconnect port will support both control and data and no XML elements are required in the platform XML and no source code in the platform worker is required for the control plane.

The platform worker must declare a ***system interconnect port*** to support data flow between workers in this HDL platform and workers in other platforms connected to this platform via the system's interconnect, such as PCI Express. As just mentioned, if this interconnect port will *also* serve as the path for the control plane, then no **cpmaster** port need be declared.

[diagram showing alternative control plane configurations]

There is a legacy interconnect port type which is found in some existing HDL platforms, called **unoc**. It is no longer recommended and not described further here. For new platforms, the interconnect port type is **sdp** (for Scalable Data Plane). When this line is included in the platform XML:

```
<sdp name="mybus" master='true'/>
```

it indicates that the platform worker will adapt and connect the system's bus to the protocol defined for on-chip **sdp** ports. This **sdp** element has an optional **count** attribute which can specify that this port supplies multiple concurrent channels between the system's bus/memory and the on-chip infrastructure. Data flow connections between the platform's on-chip HDL application workers and off-chip workers (on other platforms) will be allocated to the channels in round robin fashion.

When the **sdp** is declared, the **sdp_width** parameter of the platform worker indicates the width of the **sdp** port, in DWORDS (32 bit words), and the **sdp_length** parameter indicates the maximum burst length (in **sdp_width*32** bit words) that is tolerated on the system interconnect. See the SDP Data Plane Interface section for details of this interface.

5.4.4.4 Device Elements in a Platform XML File

Device elements (**<device>**) in the OHPD indicate devices that are part of the platform and are directly attached to the platform FPGA's pins. These device elements *declare* which devices are *physically present* and *may* be used (and thus may be instanced) in platform configurations and containers. These declarations here, by themselves, *do not* cause the device workers to be instanced in any containers or bitstreams. Devices are used and instanced by being referenced in platform configurations and containers.

Device elements indicate:

- The HDL device worker managing this device (like a device driver) *(required* **worker** *attribute)*
- The name of the device *(optional* **name** *attribute)*. If no **name** is provided, the worker name is used, and if there are multiple devices using the same worker, a zero-based ordinal is automatically appended to the worker name.

- Which parameter settings should be used for the device worker for this device *(optional* **property** *child elements)*
- Any mapping between the device worker's external signals and the names that the platform uses for those same device signals (e.g. in its constraints file) *(optional* **signal** *child elements)*
- A mapping which can specify which actual other devices in the platform are ***supported*** by the indicated device worker, using **<supported>** child elements.

In most cases a device element simply indicates that a device exists and which device worker should be used to manage it. E.g.: if a platform had a flash device that was managed by the device worker named **flash.hdl**, this might be the device element:

```
<device worker='flash'/>
```

Like normal application workers, device workers can have parameter or initial property settings. To make device workers reusable on multiple platforms, different parameter values may be needed for different platforms. E.g., if a device worker has several clocking modes depending on how its hardware is configured, these property values indicate to the device worker how it should operate on this particular platform. Such property values are supplied using property elements (with **name=** and **value=** and **valueFile=** attributes), much like the property values for application components in an OpenCPI application.

Specifying a parameter value here means that the specified value will serve as the default value whenever this device is used (instanced) on this platform. Thus when a device is instanced elsewhere in a platform configuration, container, or slave assembly or application, no platform-specific value need be, or should be specified. If such a value *is* specified where the device is instanced, it much match the value declared for the platform, otherwise it is an error. However, in the device declaration, if the **default=** or **defaultFile=** attributes are used rather than the **value=** or **valueFile=** attributes, then the value can indeed be overridden where it is instanced.

In the following example, the device element says there is a **lime_adc** device present, and on this platform, its **use_ctl_clk** property should be set to **true**.

```
<device worker='lime_adc'>
  <property name='use_ctl_clk' value='true'/>
</device>
```

There is one *required* device on every platform: the ***time server***. It must be specified including the lines:

```
<device worker='time_server'>
   <property name='frequency' value='100e6'/>
</device>
```

The frequency of the clock supplied on the required **timebase** port must be specified as the **frequency** parameter to this device worker.

Device elements can have **signal** elements to indicate that the standard signal naming should be overridden to match up with the constraints file of the platform (so

that the constraints file can remain untouched). These **signal** elements use the **name** attribute to indicate the signal name as declared by the device worker, and the **platform** attribute to indicate the name used for the platform. If not mapped this way, the platform signal for this device's signal is the device's name, followed by underscore, followed by the device worker's declared external signal name.

If the **platform** attribute is the empty string, it indicates that this device signal is not connected to the device on this platform. The following example indicates the presence of a **lime_dac** device, but that on this platform the **tx_clk** signal is not connected:

```
<device worker='lime_dac'>
  <signal name='tx_clk' platform=''/>
</device>
```

These **signal** elements for signal name mapping under the **device** elements here do *not* use the same syntax as the **signal** elements used to *declare* signals under the top-level elements of platform workers, device workers or slot type definition XML files.

When the specified device worker is a ***subdevice*** which has declared that it supports other devices (using the **<supports>** elements in the subdevice worker's OWD), the **<device>** element here in the OHPD can provide a mapping to indicate exactly *which* devices should be supported by this subdevice on *this* platform. If no mapping is supplied, then the default mapping is used. The section Subdevice Workers describes subdevices in more detail.

The default mapping is where the ordinal of this subdevice is used to find the supported device of the same ordinal. E.g., if this subdevice is the 2nd of 3 subdevices of this type present on the platform, then any supported devices are assumed to be the 2nd of 3 of that type of device. If a supported device is a singleton on the platform, then it is assumed to be the supported device.

When the default mapping cannot be used, a mapping is indicated by including one **<supported>** element here for each **<supports>** element in the subdevice worker's OWD. These **<supported>** elements have one attribute, **device**, indicating the name of the particular device on this platform that should be supported.

```
[example of device worker with <supports> and <device> with
<supported> as pointers in the two directions of association]
```

So the subdevice worker is saying that it ***supports*** other devices of certain types, and this mapping is saying: on this particular platform, these are the devices that are actually ***supported*** (which must be of the worker type indicated in the underlying **<supports>** element).

5.4.4.5 Signal Elements in a Platform XML File

In order to provide the required services at its declared **cpmaster**, **sdp**, or **timebase** platform ports, the platform worker normally requires direct access to some external signals that are not associated with any other device. These are signals such as clocks, interconnects, LEDs, and switches. These ad-hoc signals are declared using the signal elements defined in Signal Declaration XML Elements. The signal names should generally match those in the platform's constraints file, including case. However, the

platform worker can define its own (perhaps more convenient) signal names and use the **platform** attribute to specify an associated platform signal that corresponds to the constraints file.

Signals implied by the presence of devices do not need to be specified.

If a platform worker itself wants direct access to slot signals, it must declare those signals using the **signal** element, even though the signal's existence is already implied by the existence of the **slot** element. An example of this is the “presence” signal in FMC slots. They are not related to any device on a card, but are used by the platform worker to know when a card is plugged in (is present). So, in the **ml605** platform worker these signal elements are present:

```
<!-- These "card-is-present-in-slot" signals are from each slot-->
<signal name='fmc_lpc_prsnt_m2c_l' direction='in'/>
<signal name='fmc_hpc_prsnt_m2c_l' direction='in'/>
```

The slot names are **fmc_lpc** and **fmc_hpc**, and the standard name for this signal is **prsnt_m2c_l**. When the signal is an output (from the FPGA to the slot), the signal must not be used as an output by any device worker for a device on a plugged-in card.

5.4.4.6 Slot Elements in a Platform XML File

When a platform has slots, it includes a **slot** element to declare the existence of a slot. The required **type** attribute must match the name of a defined slot type. The slot type definition files are normally in the **hdl/cards/specs** directory in the **ocpi.core** project, which is automatically searched whenever a platform XML file is processed. Examples of slot types are:

- **fmc_lpc**: “low pin count” variant of the “FPGA Mezzanine Card” (FMC) from VITA57
- **fmc_hpc**: “high pin count” variant of the FMC cards from VITA57
- **hsmc**: High Speed Mezzanine Card from Intel/Altera

The optional **name** attribute of the slot element may assign a name to the slot. If it is not present, the slot's type becomes the slot name. If **name** is *unspecified* and if more than one slot of the same type is present, a zero-origin ordinal is appended to the slot-type as the name. E.g. if there were two slots of type **hsmc** and they were not given names, their names would be **hsmc0** and **hsmc1**. Slot names are needed for two purposes:

1. When a card is plugged into a slot, that slot is identified by its name. Slot names are case *insensitive* for this purpose (when mentioning slot names in HDL container XML).
2. The default name for signals between the platform FPGA and a slot is the slot name as a prefix, followed by underscore, followed by the signal name as defined in the slot type definition file (usually based on a standards document).

These fully prefixed slot signal names do not appear in source code or XML, but they do usually correspond to the names found in a platform's constraints file, which is typically

supplied separately by the board vendor and only modified for OpenCPI for other reasons (e.g. not for signal name changes). I.e., the signal names associated with pins of the FPGA attached to slot pins are predetermined by the vendor or board designer and not changed or redefined by the platform worker.

E.g. for a slot signal named **`PRSNT_M2C_L`**, the name of the signal from the platform FPGA to the slot, for the second of two **`fmc_lpc`** slots that did not have assigned names, would be "**`fmc_lpc1_PRSNT_M2C_L`**". Slot names (and slot type signals) *are case sensitive* for this purpose (prefixing global net names) since there are some tools and systems where the case of such signals matters in the constraints file. For a given slot, there is also a **`prefix`** attribute which can override the default **`<slot_name>_`** prefix. This is useful when there is only one slot of that type, and the platform uses the slot signal names directly without any prefix.

There is one other aspect to slot elements in the platform XML file: slot signal mapping. Most slot signal names are based on the specification of the slot types. E.g., the FMC slot signal names are defined by VITA57. If a platform's constraints file (and documentation) do not use the standard names from the slot type specification, then extra child elements are added to the platform worker's **`slot`** XML element, to map the standard (e.g. VITA57) signal names to the signal names used by this platform's constraints file and documentation.

As an example, the ZedBoard platform has a single FMC LPC slot. It uses the VITA57 signal names for *almost* all these signals. However, it changes signal names when they are differential and use the `CC` suffix. Whereas VITA57 always puts the `CC` suffix last, the ZedBoard puts the differential suffix last (i.e. **`_N`** and **`_P`**). Below is the **`slot`** element for the ZedBoard platform XML that forces the slot name to be upper case **`FMC`**, and remaps the offending signals so OpenCPI knows how to route signals to the slot's connector on this platform that *does not* follow the VITA57 conventions:

```
<slot name='FMC' type='fmc_lpc'>
  <!-- These signals don't use VITA57 signal names-->→
  <signal slot='LA00_P_CC' platform='LA00_CC_P'/>
  <signal slot='LA00_N_CC' platform='LA00_CC_N'/>
  <signal slot='LA01_P_CC' platform='LA01_CC_P'/>
  <signal slot='LA01_N_CC' platform='LA01_CC_N'/>
  <signal slot='LA17_P_CC' platform='LA17_CC_P'/>
  <signal slot='LA17_N_CC' platform='LA17_CC_N'/>
  <signal slot='LA18_P_CC' platform='LA18_CC_P'/>
  <signal slot='LA18_N_CC' platform='LA18_CC_N'/>
  <signal slot='CLK0_M2C_N' platform='CLK0_N'/>
  <signal slot='CLK0_M2C_P' platform='CLK0_P'/>
  <signal slot='CLK1_M2C_N' platform='CLK1_N'/>
  <signal slot='CLK1_M2C_P' platform='CLK1_P'/>
  <!-- These signals do not have connections to the platform -->
  <signal slot='DP0_C2M_P' platform=''/>
  <signal slot='DP0_C2M_N' platform=''/>
</slot>
```

Some platforms do not connect all possible slot signals to the FPGA. In this case the slot element maps them to an empty signal name on the platform, indicating that these

slot signals cannot be used on this platform. In the above example, the last two slot signals mentioned (**DP0_C2M_P/N**) are not connected to the (Zynq) FPGA on the ZedBoard platform.

Finally, when a platform does not use the slot name prefix in its signal names, a leading slash can be given in the **platform** attribute to indicate that no such prefix should be applied. E.g. a signal mapping of:

```
<signal slot='DP0_C2M_P' platform='/DP0_SLOT2_P'/>
```

would imply that the signal name in the constraints file would be **DP0_SLOT2_P** rather than **FMC_DP0_C2M_P**.

5.4.4.7 Examples of Platform XML Files

An example of a complete XML file for a Zynq-based HDL platform is below. The platform worker directly supports a control plane (using **cpmaster**), and sets the time server's clock frequency to **100e6**, and declares an SDP data plane (via **sdp**) with 2 channels. It has one **fmc_lpc** slot and 4 LEDs. No switches are declared. One signal (to drive external LEDs) is declared. No clock signals are declared since this platform worker uses on-chip clock-generator resources in the Zynq chip.

```
<HdlPlatform Language="VHDL" spec='platform-spec'>
  <specproperty name='platform' value='myzynq'/>
  <specproperty name='nLEDs' value='4'/>
  <metadata master='true'/>
  <timebase master='true'/>
  <cpmaster master='true'/>
  <device worker='time_server'>
    <property name='frequency' value='100e6'/>
  </device>
  <sdp name="zynq" master='true' count='2'/>
  <slot name='FMC' type-'fmc_lpc'/>
  <signal name='led' width=4' direction='out'/>
</HdlPlatform>
```

A second example uses an interconnect that indirectly supports a control plane so no **cpmaster** is necessary, but signals to get clocks and raw interconnect signals are declared. Notice that three different clocks are taken from external inputs, and the PPS inputs and outputs are used to support the time server.:

```
<HdlPlatform Language="VHDL" spec='platform-spec'>
  <specproperty name='platform' value='mypci'/>
  <metadata master='true'/>
  <timebase master='true'/>
  <device worker='time_server'>
    <property name='frequency' value='200e6'/>
  </device>
  <sdp name="pcie" master='true'/>
  <signal name="sys0_clk" direction='in' differential='true'/>
  <signal name='sys1_clk' direction='in' differential='true'/>
  <signal name='pci0_clk' direction='in' differential='true'/>
  <signal name='pci0_reset_n' direction='in'/>
  <signal name='pcie_rx' direction='in' differential='true'
                         width='4'/>
  <signal name='pcie_tx' direction='out' differential='true'
                         width='4'/>
  <signal name='led' direction='out' width='13'/>
  <signal name='ppsExtIn' direction='in'/>
  <signal name='ppsOut' direction='out'/>
</HdlPlatform>
```

5.4.5 Writing the Platform Worker Source Code

While a platform worker's XML (OWD) has extra elements to describe the platform's hardware (devices, slots), it is still an OWD, and thus it describes any implementation-specific properties and ports of the platform worker. Being a device worker, it can also define external signals that are connected externally to pins.

Thus to complete the OHPD, any such ports and properties must be defined. The OpenCPI Component Specification (OCS) for all platform workers defines certain ports and properties that all platform workers must support and implement, but platform workers can and typically do have other ports and properties that are platform-specific.

5.4.5.1 Platform Worker Properties

Some required platform worker properties are dealt with entirely in the platform XML file since they just require parameter values which can be specified in the OWD.

Several OCS properties are readable characteristics of the platform that may be defined as parameters with a constant value in the OWD, *or* be declared as volatile with a runtime-determined value. These include the **`nSwitches`**, **`nLEDs`** properties. If they are volatile, the worker must drive them by assigning values, e.g.:

```
props_out.nSwitches <= to_ulong(3);
props_out.nLEDs <= to_ulong(7);
```

Otherwise, the OWD can simply specify the value, e.g.:

```
<specproperty name='nSwitches' parameter='true' value='3'/>
```

Some platform properties are always volatile such as **`switches`** and **`slotCardIsPresent`**, so the platform worker drives them with:

```
props_out.switches <= switch_input_pins;
props_out.slotCardIsPresent <= (others => '0');
```

Some are writable such as **`LEDs`**, and are input to the worker:

```
my_led_pins <= props_in.leds(3 downto 0);
```

Finally, some properties are associated with platform ports and are described below in the sections about each platform port type.

Here is a typical example of VHDL code in a platform worker for its properties:

```
props_out.switches          <= (others => '0');
props_out.slotCardIsPresent <= (0 => not fmc_prsnt,
                                others => '0');
props_out.UUID              <= metadata_in.UUID;
props_out.romData           <= metadata_in.romData;
metadata_out.clk            <= ctl_in.clk;
metadata_out.romAddr        <= props_in.romAddr;
metadata_out.romEn          <= props_in.romData_read;
led(0)                      <= props_in.leds(0);
```

Below is a summary table with the OCS properties and their accessibility.

Table 4: Platform Worker OCS Properties

Name	Type	Access	Description
platform	String	Parameter	Platform name Set in OWD <specproperty>
sdp_width	UChar	Parameter	Width in DWORDS of SDP. Set in OWD for **sdp** platform port
UUID	ULong *16	Readback	Unique ID of bitstream file. Connected in source code for **metadata** platform port.
romAddr	UShort	Writable	Address for reading bitstream metadata. Connected in source code for **metadata** platform port.
romData	ULong	Volatile	Data when reading bitstream metadata Connected in source code for **metadata** platform port.
nLEDs	Ulong	Parameter/ Readback	How many LED indicators are present? Can be parameter *or* readable.
LEDs	ULong	Readable+ Writable	Actual LED settings, up to 32, LSB is LED 0
nSwitches	Ulong	Parameter/ Readback	How many switches are present? Can be parameter *or* readable.
switches	ULong	Volatile	Actual switch settings, up to 32, LSB is switch 0
nSlots	Ulong	Parameter	How many slots are present? Set in OWD <specproperty> Must match the number of **slot** elements in OWD
slotCardIsPresent	Bool array	Volatile	Indicate whether a card is plugged into each slot
slotNames	String	Parameter	Comma-separated list of slot names Set in OWD <specproperty>

5.4.5.2 Ports of Platform Workers

Platform workers have ports that are different than ports of application workers. They are not used for moving data to and from other workers. They allow the platform worker to provide required services to the rest of the HDL infrastructure. The platform ports were introduced in Platform Ports in a Platform XML File. Each has implications in the platform worker source code.

The simplest platform port is the ***metadata port***, which simply requires that the platform worker connect the signals at the metadata port to the properties associated with the port using this exact VHDL code:

```
props_out.UUID              <= metadata_in.UUID;
props_out.romData           <= metadata_in.romData;
metadata_out.clk            <= ctl_in.clk;
metadata_out.romAddr        <= props_in.romAddr;
metadata_out.romEn          <= props_in.romData_read;
```

For the ***timebase port***, the worker is required to provide three output signals to that port, and take one output signal from that port, e.g.:

```
timebase_out.clk      <= clk;
timebase_out.pps      <= '0';
timebase_out.usingPPS <= '0';
my_pps_out_pin        <= timebase_in.pps
```

These signals are the basis for timekeeping on the platform. They are a clock and a PPS input signal and an optionally connected PPS output signal. The platform worker should provide the timebase clock that is best suited for timekeeping, which usually means the one with the least jitter and drift over the short term. On some platforms this is simply the same as the control clock, but on others there may be a clock with better performance for this purpose and the platform worker should use it. The **`timebase_out.usingPPS`** signal informs the timekeeping system whether the PPS signal is actually working and valid. The **`timebase_out.pps`** signal should be driven by a real PPS (pulse per second) if the platform has that capability. The **`timebase_in.pps`** signal is an output from the OpenCPI timekeeping service that is driven whether or not there is a PPS input signal on **`timebase_out.pps`**.

If a platform worker is *directly* supporting a ***control plane master port*** (indicated by the **`cpmaster`** element in the OWD), the platform worker must provide an addressable path from the controlling processors's software into the FPGA. This usually involves adapting an address window (64MB in the current release) on an addressable bus that the FPGA is connected to, to the protocol and signals of the **`cpmaster`** port. The adaptation is normally put into its own module and then instanced in the platform worker. The platform worker must choose an appropriate clock and (asserted high) reset signal to serve as the platform's control clock and reset.

Remember that the platform worker may provide for a control plane *indirectly* by providing an ***interconnect port*** that can serve the same purpose. In that case no platform worker support of a **`cpmaster`** port is necessary.

A **`cpmaster`** example from the ZedBoard platform is where an addressable bus port in the Zynq chip, which connects the processor to the FPGA, is called the **`M_AXI_GP0`**. The Zynq *CPU* is the master, and generates read and write accesses to the FPGA acting as a slave. In this case an adapter module was written (called **`axi2cp`**) to convert the protocol used by the **`M_AXI_GP0`** port in the SoC hardware, to the OpenCPI control plane protocol. This module is instanced in the Zed platform worker and connected to the Zynq CPU on one side, and the platform worker's **`cpmaster`** port on the other. In the Zed platform worker this adapter and its connection is shown by this code:

```
cp : axi2cp port map(clk     => clk,
                     reset   => reset,
                     axi_in  => ps_m_axi_gp_out(0),
                     axi_out => ps_m_axi_gp_in(0),
                     cp_in   => cp_in,
                     cp_out  => cp_out);
```

The signaling protocol of a **cpmaster** port is described in the Control Plane Master Port Protocol section.

The last platform port type, ***system interconnect,*** is specified using the **sdp** element in the OWD. The platform worker must provide a path from the platform's bus/interconnect to the **sdp** port. When the interconnect used for data flow can also be used to provide control plane access, then the **cpmaster** port described above is unnecessary. The key attribute needed for the system interconnect to indirectly support control plane access is that the processor on the other side of the interconnect can read and write into the FPGA with the FPGA acting as an addressable slave. This is common when there is only one external bus connected to the FPGA, such as PCI Express.

In the case of the Xilinx Zynq platform, there are multiple hardware interfaces between the on-chip system interconnect and the FPGA (called PL on Zynq). In this case, the platform worker reserves one such interface exclusively for control access (the **M_AXI_GP0** connected to a **cpmaster** port), and uses different interfaces for the data plane.

The **sdp** ports act as bus masters, generating addressed DMA requests *to* the platform worker which should adapt and present those requests to the system interconnect. The **sdp** ports can *also* act as bus slaves, allowing their use for control plane purposes. The platform worker decides whether to support the **sdp** port as only a bus master (e.g. as it does on Zynq), or as both master or slave (as it does on PCI Express systems).

The **sdp** ports can be multichannel, meaning that the platform worker can provide multiple simultaneous paths between the system interconnect and the FPGA data plane infrastructure, to more efficiently support simultaneous data flows between workers in the FPGA and workers outside the FPGA.

The signaling protocol of an **sdp** port is described in the SDP Port Protocol section.

5.4.5.3 Platform Worker Clocks

As seen above in the discussion of the control plane adaptation (the **cpmaster** port), the platform worker sources the clock (and reset) used for the OpenCPI control plane on the platform. It also sources any clocks associated with the “data plane” (sdp) where data/messages are flowing to other workers in other FPGAs or software platforms. In both these cases the clock and reset signals are part of the interface at these ports.

A platform worker usually also sources another clock for timekeeping on the platform (in the **timebase** port).

In general these clocks already exist on the platform and are simply assigned to these additional purposes. I.e. the control clock may be the clock already associated with the

bus between the CPU and FPGA. In some cases the platform worker may synthesize/generate new clocks for these (3) purposes: control, data, and timekeeping.

5.4.6 Building the Platform Worker

The platform worker is built in its directory using **`ocpidev build`**, with results in one or more target subdirectories. Platform configurations are also built in the platform worker directory when **`ocpidev build`** is invoked. In fact, when nothing else is specified, a "base" platform configuration (with no device workers) is built whenever the platform worker is built. To specify more platform configurations to build, specify them in the **`Configurations`** attribute in the XML file, and for each one, write a platform configuration XML file describing which devices (and any parameters for them) that should be included in the platform configuration. See the Platform Configurations section below for a complete description of platform configurations.

Running **`ocpidev build`** in a platform worker directory builds the platform worker, the **`base`** configuration, and any other platform configurations as specified in the **`Configurations`** XML attribute. No **`--hdl-platform`** option is needed since it is implied by being in the platform worker's directory.

5.4.7 The HDL Platform Exports File

HDL platforms have exports very similar to RCC platforms. They declare which local files in the platform's directory (whether static or built/generated) are required by users of the platform. A typical example of an exported file for an HDL platform is the default constraints file. See the Platform Exports File section for GPP/RCC platforms.

The ***`<platform>`*`.exports`** file itself is automatically exported.

5.4.8 Specifying Platform Configurations in XML Files.

HDL platform configurations are described in the ***HDL Build Hierarchy*** section of the OpenCPI HDL Development Guide.

For each platform configuration mentioned in the **`Configurations`** attribute in the platform worker's XML file, there must be a corresponding XML file. This XML file has these aspects:

- The top level element is **`<HdlConfig>`**.
- A **`device`** child element is present for each device in the configuration.

For devices previously defined as being part of the platform (and thus mentioned in the platform worker's OHPD), the device element simply has a "**`name=`**" attribute indicating which of the platform's devices should be included in the platform configuration.

Platform configurations can also specify devices that are on cards plugged into one of the platform's slots. Specifying the **`card`** attribute indicates which card the device is on, and implies that the card is plugged into one of the platform's slots. If there are multiple slots of the type that the indicated card is defined for, then a **`slot`** attribute must be used to unambiguously indicate which slot the card is plugged into.

Thus platform configurations can indicate a mix of devices: those that are part of the platform, and others that should be made available assuming a certain type of card is plugged into one of the platform's slots. The following example, (for the ZedBoard platform), indicates that a configuration should be built that assumes a **lime-zipper-fmc** card is plugged into the slot on the platform, the **lime_adc** device should be instanced, and any device worker that *supports* the **lime_adc** device on that card should also be included. There is no **slot** attribute included or required since the platform has only one slot.

```
<HdlConfig>
  <device name='lime_adc' card='lime-zipper-fmc'/>
</HdlConfig>
```

Device elements in this file can also set values for parameter properties of the device worker for the device, but *only those that are not already specified with the **value** attribute in the board definition* (either the platform worker XML file (the OHPD) or the card definition XML). I.e. the board definition file specifies fixed aspects of the device as it exists on that board, but any other parameter properties not mentioned for the board can be configured as required in the platform configuration (or in the container). If the value in the OHPD file uses the **default** attribute (attribute whose name is *default*), then it may be overridden in the platform configuration. E.g.:

```
<HdlConfig>
  <device name='lime_adc' card='lime-zipper-fmc'>
    <property name='use_control_clock' value='true'/>
  </device>
</HdlConfig>
```

The same capability exists for the platform worker itself. Parameter property values for the platform worker can be specified by top level property elements in this file, e.g.:

```
<HdlConfig>
  <property name='ocpi_debug' value='true'/>
  <device name='lime_adc' slot='lime-zipper-fmc'/>
</HdlConfig>
```

The **HdlConfig** element has an optional attribute, **constraints**, which can specify the name of a constraints file to be used with this platform configuration (rather than the default one for the platform). The appropriate suffix is applied automatically.

The **constraints** attribute can be a file name optionally 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:

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

The parameters are set as global variables that can be accessed in the constraints file itself. So the above example sets the **rx** and **mode** variables for use when the constraints file is processed by the underlying tools. When a constraints file is parameterized this way, it must take care to accept the situation where the variable is

not set at all, presumably setting it to a default value. E.g., in the tcl language one could have:

```
if { ! [ info exists rx ] } { set rx 0 }
```

5.4.9 Control Plane Master Interface

This interface is defined in the HDL primitive library whose name is **`platform`**, found in the **`ocpi.core`** built-in project. The input and output signal records for the interface are thus: **`platform.platform_pkg.occp_in_t`** for signals going *from* the master (i.e. the platform worker), *into* the control plane, and **`platform.platform_pkg.occp_out_t`** for signals going from (out of) the control plane *to* the platform worker. I.e. in this case the "in" and "out" are relative to the control plane slave, which is part of the generic OpenCPI HDL infrastructure.

This control plane interface is a simple 32 bit data bus with these features:

- There are byte enable bits allowing the master to read or write individual bytes (or aligned shorts) within a 32 bit location.
- When the master system bus or interconnect is driving this interface, and that bus supports 64 bit accesses, those access must be correctly converted into 2 sequential 32 bit accesses on this interface.
- The master drives clock and (asserted high) reset.
- The address is a DWORD address.
- A request is asserted using the **`in.valid`** signal, and acknowledged (accepted) using the **`out.take`** signal, and thus a request is consumed when both are asserted (like AXI).
- A read response (and data) is asserted using the **`out.valid`** signal, and must be accepted by asserting the **`in.take`** signal. The response must be accepted even if the request is not. I.e. a read request is only accepted by the slave when the associated read response data is accepted by the master.

One example master module is the **`axi2cp`** module that converts an AXI 4 slave to a cpmaster. It is found in the **`hdl/primitives/axi`** library in the **`ocpi.core`** built-in project.

[timing diagrams needed here].

5.4.10 Scalable/Simulatable Data Plane (SDP) Interface

When FPGA platforms have off-chip interconnects to connect with other FPGAs or GPPs, they need to be adapted to the on-chip infrastructure that is common to all OpenCPI FPGA platforms. Examples of off-chip interconnects are PCI Express, Ethernet, and the Zynq AXI interconnect between the FPGA side of the SoC (called PL) and the multi-core GPP side (called PS).

The main interface in the OpenCPI data-plane infrastructure is the SDP.

The S in SDP is used both for the Scalable aspect (where the width of the data path is parameterized) as well as the Simulatable aspect (where the entire on-chip data plane infrastructure runs in simulation).

The SDP interface is an on-chip (inside of FPGA) interface which is used for transferring packets to/from OpenCPI infrastructure modules. The purpose of this chapter is to provide the information required to adapt other interfaces/interconnects to the SDP. Each interconnect is adapted to the SDP so that the rest of the on-chip data-plane infrastructure remains common across all HDL platforms supported by OpenCPI.

Within the OpenCPI data plane stack, the data plane is organized in two layers, the "message transport" and the "data transfer" layers. [diagram from briefings]

The ***transport*** layer is interconnect agnostic and provides buffer and message management. The data ***transfer*** layer is defined by OpenCPI with a protocol specification for Remote DMA (RDMA) which provides data transfer interoperability between heterogeneous components executing within FPGAs, DSPs, and GPPs that all have access to each other's addressable space. The RDMA software module allows new software drivers to be plugged into OpenCPI, enabling data exchange over a variety of interconnects. OpenCPI provides several intrinsic drivers with the standard distribution including Host Memory, network sockets and DMA bus/fabric.

SDP was designed to process the OpenCPI RDMA protocol inside of FPGA containers. Other requirements and/or design goals of SDP were:

- Minimize differences between SDP and common FPGA interconnects (AXI, PCIe)
- Minimize complexity of FPGA infrastructure which must interface to SDP
- Provide portability and commonality of DMA infrastructure.

The SDP interface has the following attributes:

- Bidirectional: either side of a connection can be master
- Multi-master: multiple FPGAs can be masters on an interconnect
- Packet-based: header and (optional) data
- Split transaction: Read request packets elicit read response packets (with data), Write request packets (with data) are posted, with no response
- Full-duplex

The side of the interface closer to the off-chip interconnect is called the SDP interface master, and thus must interact with a slave SDP interface. At the VHDL port level, there are two record structures (one per direction) with VHDL types **`m2s_t`**, and **`s2m_t`**. Data is bidirectional and of type **`dword_array_t`** with parameterized width using the generic parameter **`sdp_width`** to define the width in DWORDS (32 bit words), hence the "Scalable" in SDP.

The VHDL ports (e.g. for an SDP master) are:

```
sdp_out      : out m2s_t
sdp_out_data : out dword_array_t(0 to sdp_width-1)
sdp_in       : in  s2m_t
sdp_in_data  : in  dword_array_t(0 to sdp_width-1)
```

Within both `m2s_t` and `s2m_t` records is a common messaging record of VHDL type `sdp_t`. The figure below illustrates the port structure using VHDL types, and the tables following it describe the signals.

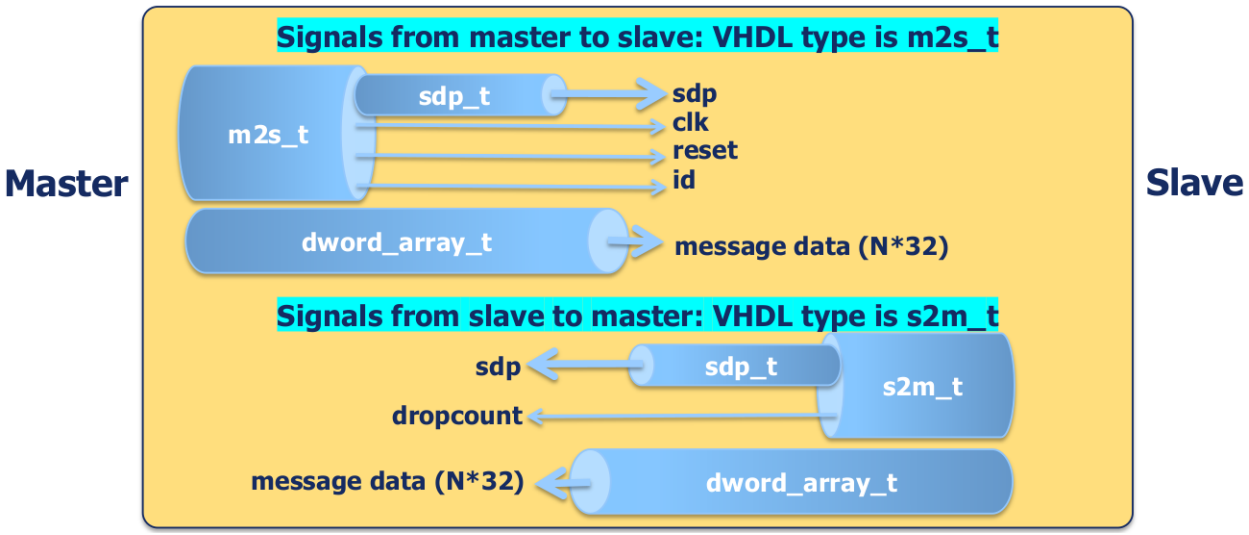

Figure 1: SDP VHDL Record Structure

Table 5: SDP `m2s_t` Signals

Signal	Type	Description
`sdp`	`sdp_t`	SDP header and handshake signaling common to masters and slaves
`clk`	`std_logic`	The clock for the SDP instance, usually from the HDL platform worker.
`reset`	`bool_t`	Associated synchronous reset (asserted high for minimum of 16 cycles)
`id`	`id_t`	The SDP node/position/ordinal assigned to the attached slave

Table 6: SDP `s2m_t` Signals

Signal	Type	Description
`sdp`	`sdp_t`	SDP header and handshake signaling common to both SDP masters and slaves
`dropCount`	`uchar_t`	Count of non-decoded/dropped packets from slave (only when slave is terminator)

The `sdp_t` VHDL record contains header and handshake signals. The figure below shows the record structure and following tables describe the record signals.

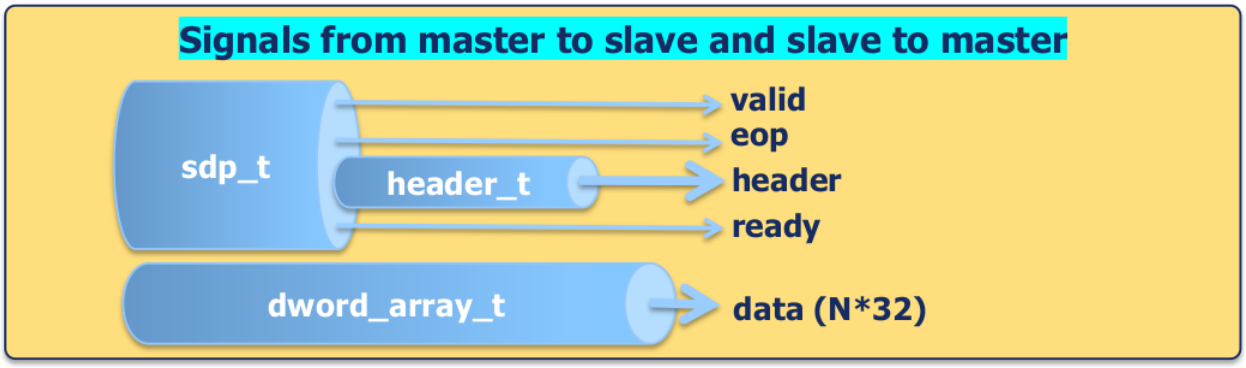

*Figure 2: **sdp_t** Record Structure*

Table 7: SDP **sdp_t** Signals

Signal	Type	Description
`valid`	`bool_t`	Data/header is valid (like AXI). Analogous to "FIFO not empty".
`eop`	`bool_t`	End of packet. Qualified by valid.
`header`	`header_t`	Header defines contents of packet. Will be stable from sop to eop
`ready`	`bool_t`	Can accept/is accepting data *from other side* - like AXI; analogous to "FIFO dequeue".

*Table 8: SDP **header_t** Signals*

Signal	Type	Description
`count`	`count_t`	Number of dwords of data minus 1 (like AXI). For read requests, requested count. For writes and read-responses, count of data in packet.
`op`	`op_t`	Operation: read or write or read response.
`xid`	`xid_t`	Transaction ID for matching read responses to read requests
`lead`	`unsigned`	Number of invalid bytes at start of first dword of packet.
`trail`	`unsigned`	Number of invalid bytes at end of last dword of packet
`node`	`id_t`	*For packets to slaves*: The on-chip node that should receive this packet. *For packets to masters:* The on-chip node that should receive the read response (if any).
`addr`	`addr_t`	LSBs of dword address. *For requests to slave*, address within SDP node. *For requests to master*, simply word address LSBs.
`extaddr`	`extaddr_t`	*For requests to master*, address MSBs to achieve 36 bit addressing.

Packets are transferred using a sequence of transfers controlled by **`valid`** and **`ready`**. In FIFO-speak, **`valid`** is analogous to "FIFO not empty", and **`ready`** is analogous to "dequeue". The **`valid`** signal in one direction is acknowledged by the **`ready`** signal in the other direction. When they are both asserted (from opposite directions) at a rising edge, the transfer is complete. The **`valid`** signal can lead **`ready`** or vice versa. This terminology is similar to AXI.

The following diagram shows three SDP transfers:

- ready trailing valid
- ready leading valid
- ready simultaneous with valid

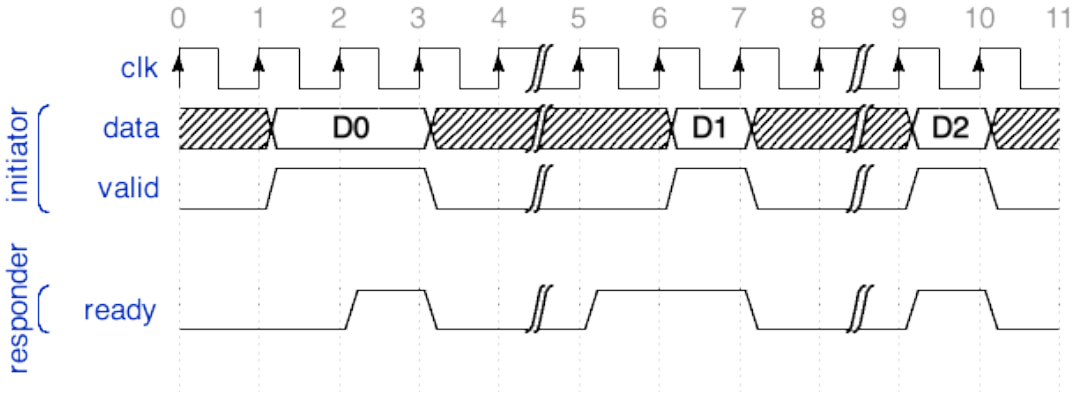

Figure 3: Three SDP Transfers with Different Ready and Valid Assertions

The **eop** signal indicates the last transfer in a packet, and is qualified by the **valid** signal. When **valid** is not asserted, **eop** is meaningless/undefined. The last transfer in a packet is indicated when both **valid** and **eop** are asserted. The start of a packet is inferred (after previous **eop** or **reset**). The following diagram shows an example of a packet transfer completion.

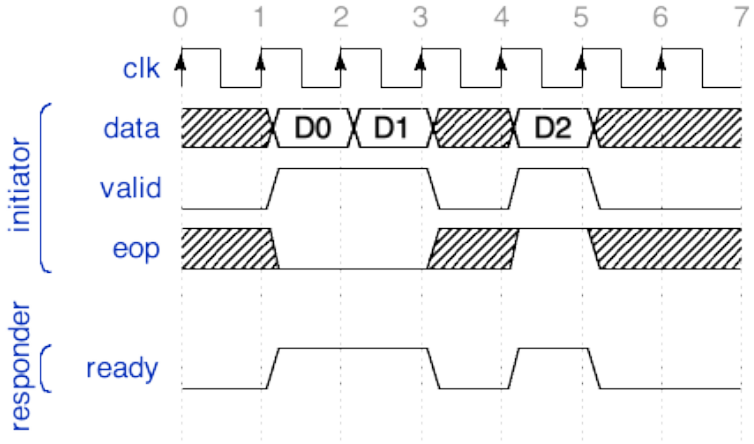

Figure 4: SDP Packet Transfer Showing EOP

The initiator must hold the header constant for the duration of a packet. The header signal structure is continuously valid at the first transfer in the packet through the last one; i.e. it does not need to be captured on the first transaction in the packet. It becomes valid when the valid signal is asserted at the beginning of a packet. The data signal may provide new data on each transaction in the packet. For some packets (e.g. read requests), only the header is used, and the data is unused. The following diagram shows a packet transfer including the header.

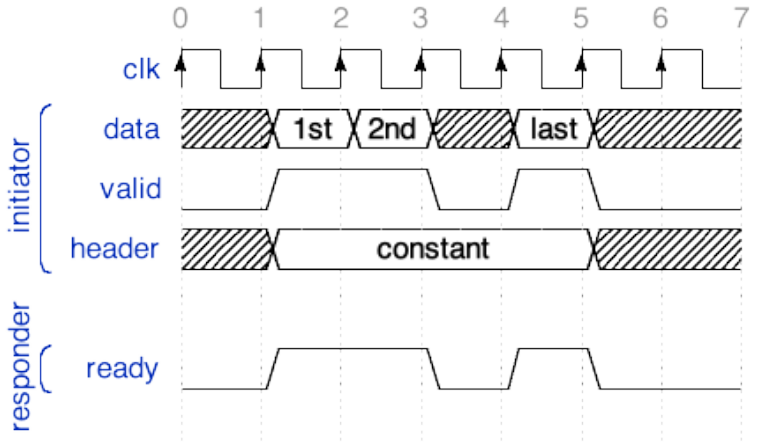

Figure 5: SDP Packet Transfer Showing Header

SDP is split transaction, like PCIe and AXI. Read request packets elicit read response packets (with data). Write request packets (with data) are posted, with no response. The first transfer of a packet transfers data (unless it is a read request). The header defines what the packet is for (its operation or op):

- A read request (with address and no data)
- A read response (data responding to a read request)
- A write request (with address and data)

The following diagram shows an example of a write request with basic header information and data.

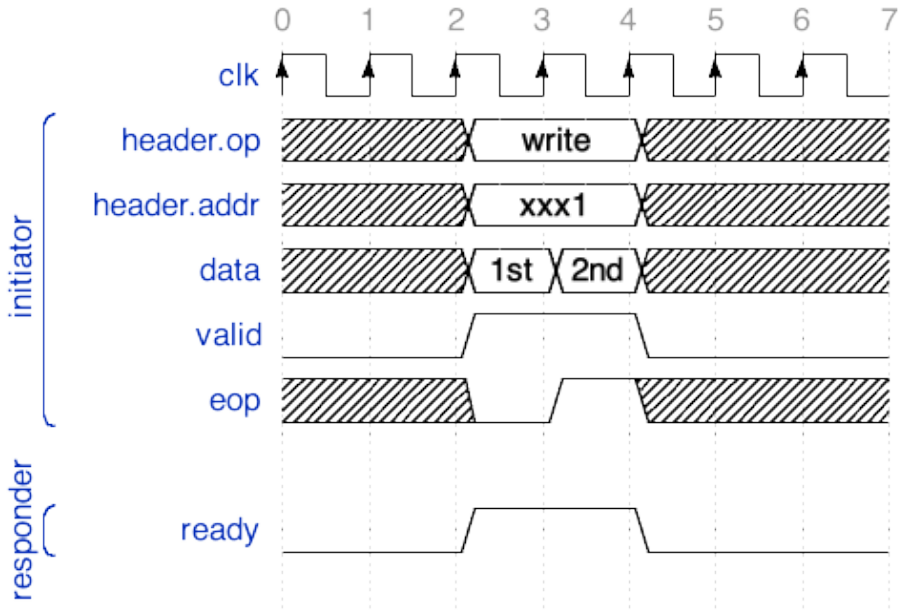

Figure 6: Basic Write Request with Address and Data

The header defines **`xid`** (transaction ID) for matching responses to read requests. The following diagram shows an example of a read request with corresponding response.

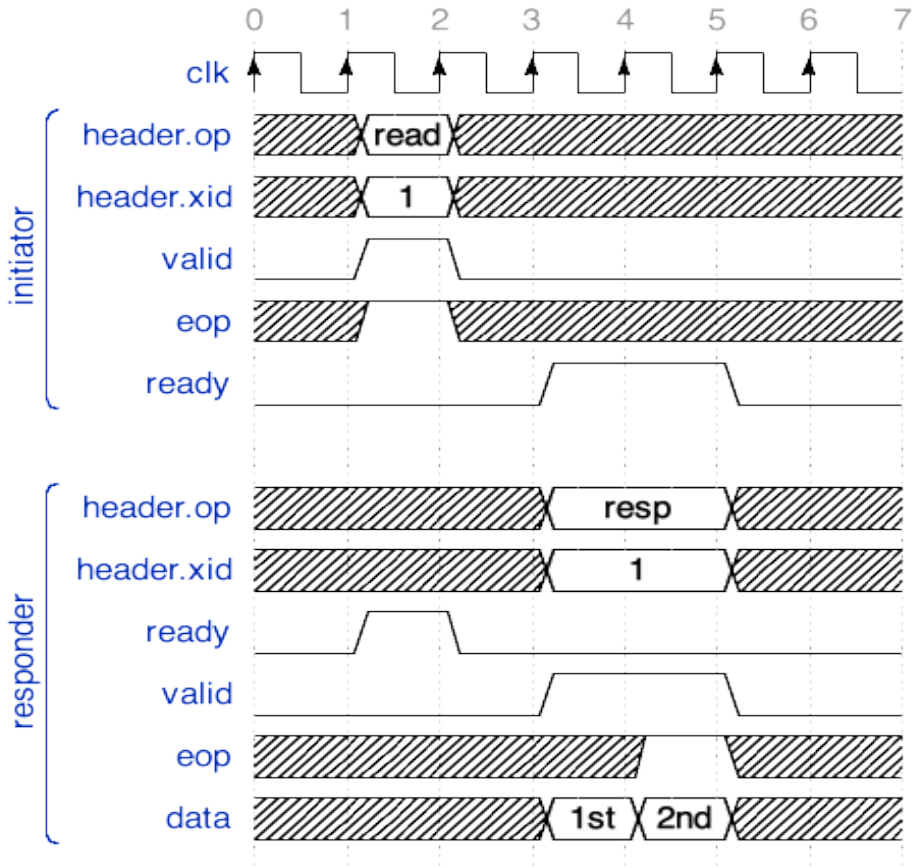

Figure 7: Read Request and Corresponding Response

The amount of data in a packet may be any number of 8-bit bytes. The header specifies **`count`** (number of dwords of data minus 1). For read requests, **`count`** is the requested number of dwords. For write and read responses, **`count`** is the number of dwords in the packet. The maximum count allows for 16KB (e.g. enough for jumbo Ethernet frames).

In addition to **`count`**, the header specifies leading invalid bytes in first dword and trailing invalid bytes in last dword. The following diagram shows how 16 bit writes are conveyed on a 32 bit interface using **`lead`** and **`trail`**.

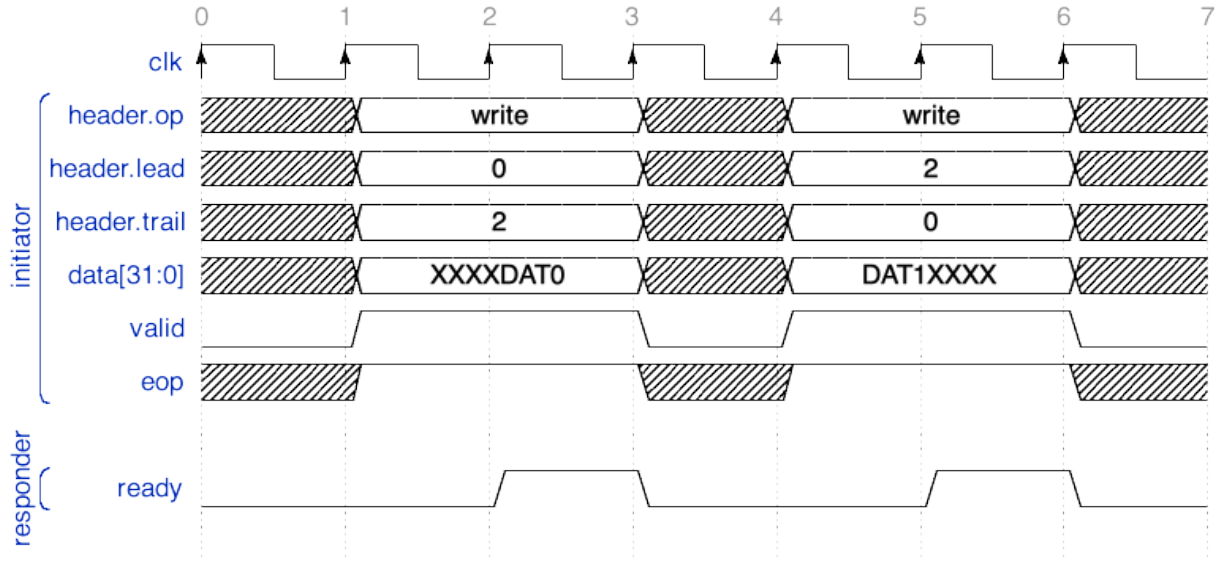

Figure 8: Two 16-bit Writes on a 32-bit Interface with Leading/Trailing Invalid Bytes

The data in a packet (if not a read request) is aligned, little endian. This is relevant if the SDP interface is configured with a dword-width of > 1. For example, if width is 2 dwords, and a single dword transfer has dword address of 1, the LSB dword will be padding, and MSB dword is the single dword of valid data. The following diagram illustrates this scenario.

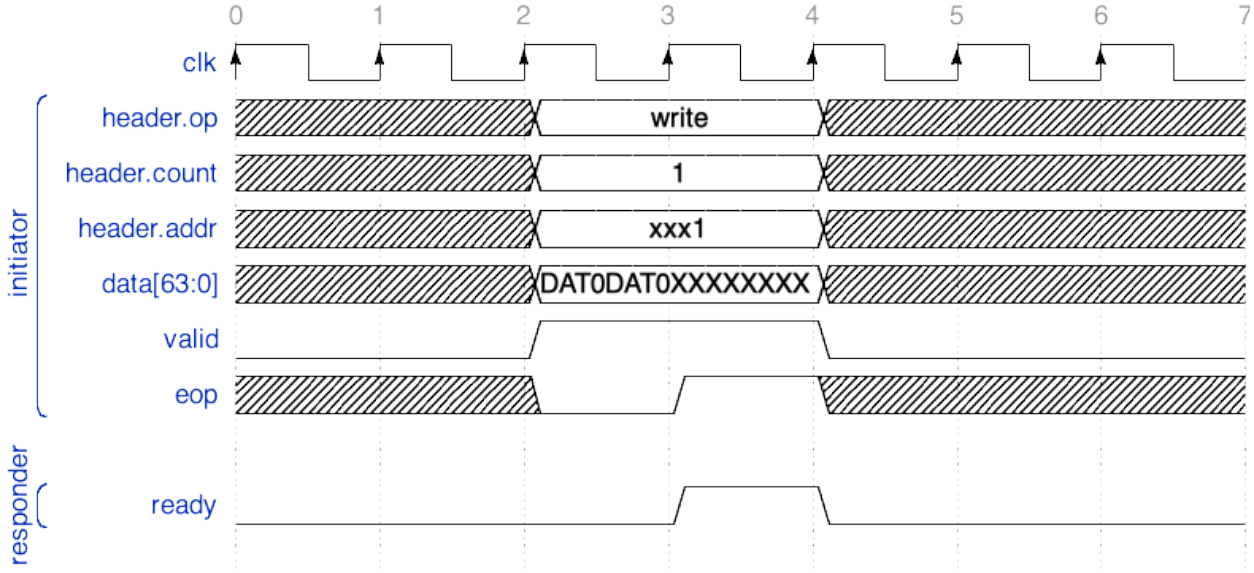

Figure 9: 32-bit Write on 64-bit Interface Showing Little-endian Alignment

5.4.11 The UNOC Data Plane Interface

The precursor/legacy interface for the on-chip data plane for OpenCPI was called UNOC (micro-network-on-chip). This interface is used in some PCI Express interconnect designs and is deprecated. Those designs are being converted to use the SDP.

5.4.12 Testing the Basic Platform without Devices

You can build and run various tests without supporting any of the devices on the platform. This can be the first chance to test the full bitstream build flow. This mode is sometimes called the “accelerator” mode, where the FPGA is simply being used to perform data processing functions, with data input from and output to software workers.

In particular, to test the control plane support, you can test the platform with no interconnect/dataplane support by using applications like “tb_bias_v2” which do not require interconnect support for the data plane.

Data plane support can then be tested one direction at a time, using simple applications like “patternbias_v2” or “biascapture_v2” that only flow data in one direction between the FPGA platform and the processor.

The biascapture_v2 assembly (in **`assets/hdl/assemblies/biascapture_v2/biascapture_v2.xml`**) contains the following XML, which takes external input and captures it:

```
<HdlAssembly>
  <Instance Worker="bias_vhdl" connect='capture_v2' external='in'/>
  <Instance Worker='capture_v2'/>
</HdlAssembly>
```

The file-bias-capture_v2 application (in **`assets/applications/file-bias-capture_v2/file-bias-capture_v2.xml`**) reads from a file and then uses the above assembly. The XML is:

```
<application package='ocpi.core' done='capture_v2'>
  <instance component="file_read" connect="bias">
    <property name='filename' value='test.input' />
    <property name='messagesInFile' value='false' />
    <property name='opcode' value='0' />
    <property name='messageSize' value='2048' />
    <property name='granularity' value='4' />
  </instance>
  <instance component='bias' connect="capture_v2">
    <property name='biasValue' value='0x01020304'/>
  </instance>
  <instance component="ocpi.assets.util_comps.capture_v2">
  <property name='stopOnFull' value='true'/>
  <property name='numRecords' value='256'/>
  <property name='numDataWords' value='1024'/>
  </instance>
</application>
```

Testing in one direction greatly simplifies initial debugging of platform workers that are using new interconnect adapters for SDP.

5.5 Device Support for FPGA Platforms

Device support involves the creation of workers that enable OpenCPI to use the devices attached to an HDL platform or card. *Device workers* are like "device drivers" for specific FPGA-attached hardware devices — workers that interface with devices using device-specific signals and I/O pins.

In addition to a control interface and data interface(s) that an application worker would have, device workers also have signal connections with hardware attached to pins of the FPGA. For example, a device worker for an output device (like a DAC or a parallel output port) would have some signals attached to FPGA pins that are connected to the output device or connector, and would also have a normal worker input data port that would be connected to some application worker producing the data.

[Diagram showing app worker → device worker → device.]

When the device is attached to the FPGA via dedicated pins, the device is considered part of the platform. When the device is on an optional card plugged into a slot (which has pins connected to the FPGA), the device is considered part of the card, not the platform. In both cases, the same device worker is used to access and control the device. An OpenCPI HDL device worker can be reused across platforms and cards.

A simple DAC device worker might have an XML descriptor (OWD) like this:

```
<HdlDevice Language='vhdl' spec='dac-spec'>
  <streaminterface name='in' datawidth='8'/>
  <signal name='valid' direction='out'/>
  <signal name='data' direction='out' width='8'/>
  <signal name='dataclk' direction='out'/>
  <signal name='ready' direction='in'/>
</HdlDevice>
```

The **`HdlDevice`** element is similar to the **`HdlWorker`** element except that it allows some extra features like the **`signal`** child elements.

The **`spec='dac-spec'`** attribute indicates that this device worker implements the component spec common to all DAC devices (with properties and ports). It can add its own properties as required in its OWD. The **`streaminterface`** element simply sets the physical data width of the input port to 8. The **`signal`** elements specify the names of device signals (HDL language “ports”) connected to the FPGA pins that are connected to a device managed by this device worker.

All the details of an HdlDevice XML are mentioned below. An example of the device worker source code (using **`version='2'`**) for this (simplistic, single clock domain) device worker might be:

```
dataclk     <= ctl_in.clk;  -- use the control clock as output clk
data        <= in_in.data;  -- drive data output from input
valid       <= in_in.valid; -- drive output valid from input valid
in_out.take <= ready;       -- accept input when output is ready
```

5.5.1 A Device Worker Implements the Data Sheet

It is recommended to use the device manufacturer's data sheet's signal and register names in the device worker code (and its OWD) so they are easily correlated to the names in the data sheet. This provides easier code maintenance and system-level debugging. E.g., a system engineer may find it valuable to probe and examine device-level settings when referring to the data sheet, even though they might not be HDL coders. When the naming conventions in the data sheet are at odds with "better" naming conventions, it may still be preferable to use them for the above reasons.

- *Implementing all the functionality in a device may not be desirable when the device worker initially written for a platform, since that platform may not use all the device's capabilities. When a device worker is insufficient in this way for a new platform or project, it should be enhanced in such a way that any existing uses of it will still work (backward compatibility) so that the community in fact gets the benefit of a newer more complete device worker implementation.*

As mentioned above, multi-function chips should generally be supported by multiple device workers. But *within* any device worker for a function, if there are features or modes that carry significant overhead, they should be controlled by parameter properties (thus generics in VHDL) so that the resources are not wasted when the feature/mode is not being used. This promotes re-use of the device worker.

When devices have internal registers, the most common approach is to define each internal register as a property in the OWD (not in the OCS), and use the "raw property" feature of HDL workers to access those registers. This enables direct control and display of the internal configuration of the device with no software programming since the **`ocpirun`** and **`ocpihdl`** commands can easily access this information. It also allows a clean "hardware-to-software" handoff when the device worker implements what the datasheet describes.

5.5.2 Device Component Specs and Device Worker Modularity

In order to ensure that similar devices have common application-visible behavior, OpenCPI has *device class specs* that represent commonly used classes of devices. This is a special case of the component specification (OCS) being the basis of multiple implementations (workers) of the same functionality. In this device case, the "multiple implementations" are for different devices of the same class. Device workers for devices in the same class implement the same OCS; a device class is represented by this device class OCS. Current classes include **`clock_gen`**, **`qdac`**, and **`qadc`**.

As the integration of functionality on single chips increases, it is common for a "device chip" to implement multiple functions of different classes of device (e.g. an ADC and a DAC). Generally, this should *not* result in a single device worker for the multi-function chip. Creating such a single device worker would result in the functions not being represented to the system or to users the same as a similar function from a non-integrated single-purpose chip. Thus, a device worker should be developed for each class that is present on the multifunction chip.

This avoids exposing the multi-function integration of a single chip to applications and thus enhances the re-use of applications. Furthermore, when only one function is

needed, a multi-function device worker would not use FPGA resources for functions that are not being used. While it is possible to develop "swiss-army-knife" device workers for multifunction chips that intend to avoid using resources for parts that are unused, OpenCPI recommends designs that are more modular, namely:

- Chips that contain multiple functions should generally be treated as multiple devices using multiple device workers of the appropriate classes.

When a multi-function device has pins or hardware that must be shared among the functional device workers (e.g. a common reset or SPI or clocks), a "subdevice" module can be used for such shared logic. This is discussed below in Subdevice Workers.

The first step in developing support for a new device is to identify the device class specs that are relevant, and determine the list of device workers that must be created. Each device class spec defines common properties, data ports, and implied functionality that all device workers of the class should try to implement.

Some examples of classes of devices currently defined are:

- ADC and DACs which convert between isochronous data and flow-controlled data and may have scheduling and time-stamping functionality.
- Upconverters and downconverters between IF/Baseband and RF.
- Clock generators
- DRAM

5.5.3 Device Proxies — Software Workers that Control HDL Device Workers

A device proxy is a software worker (RCC/C++) that is specifically paired with one or more 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. While it is possible that the HDL device worker itself could support the required generic interface, for many device classes, it is more productive to split the supporting code for the device into a (software) proxy and a HDL device worker. When a device worker has a proxy, it is termed the "slave" of that proxy. Using a proxy is not always required since the underlying (slave) device worker is always controllable directly, using the "data sheet" level of interface.

The requirements for a class of devices may in fact be split into a low level part that HDL device workers typically implement, and a high level part that would usually be implemented in a proxy. An example might be where an ADC device worker had a low-latency gain adjustment input port that could probably not be implemented in software, as well as a high level sample rate setting that would be better implemented in a proxy.

Device proxies are simple C++ (*not C*) RCC workers where the XML (OWD) specifies a **`slave`** attribute, indicating which worker is the slave for that proxy. That attribute enables convenient access to all the slave worker's properties from the proxy's code. A more powerful mode is where the proxy specifies a ***slave assembly*** – a group of slaves with interconnections, all controlled by the proxy. [more explanation of slave assemblies, typically used by DRCs, should be included here].

Thus there are two patterns for implementing HDL device support in OpenCPI:

- *Device-worker-only*, where the device worker implements both the device class component spec as well as any required higher level properties.
- *Device-worker-and-proxy,* where the device worker implements only the device component spec, and the device proxy implements the higher level properties for the class.

A common example of higher level properties for a device class is the center/tuning frequency of an RF/IF up/down converter. The high level property is a convenient floating point number, which typically requires setting a variety of device registers to accomplish. The proxy code would make the necessary translations and computations before setting the correct register values in the HDL device worker.

5.5.4 Subdevice Workers

[Diagrams here, based on existing briefings etc.]

Writing a device worker to be reusable across many embedded systems is made more difficult by two facts:

- Multifunction devices have aspects that are shared between functions (e.g. common reset)
- Controlling different devices sometimes involves sharing control/configuration buses (e.g. SPI and I2C) or other hardware among unrelated devices.

In order to preserve the modularity of distinct classes of devices, as well as the reusability of device workers, OpenCPI supports a further specialization of device workers called subdevice workers.

A subdevice worker implements the required sharing of low level hardware between device workers. It is defined to *support* some number of device workers, and is thus instantiated *automatically* whenever any of its *supported* device workers are instantiated in a platform configuration or container. Subdevices are commonly *platform-specific* or *card-specific*.

New subdevice workers may be required when a device with an existing device worker is used on a new platform or card, since the sharing of hardware (e.g. an I2C bus) may require different logic. That may result in the existing device worker being refactored to share functions that it did not share before, by defining an interface to a subdevice.

There are cases where a subdevice is required to support a single reusable device worker when some low level logic must be different for different platforms. This allows the device worker itself to remain portable, letting alternative subdevices do platform/card-specific dirty work.

A subdevice is specified as existing on a platform or card, and is instantiated automatically whenever any of its supported devices are needed. Thus the right subdevice is chosen when the card or platform is already known, in the container. It can also be explicitly instantiated in a platform configuration or container.

So a given device worker may have a number of different subdevices that support it, but the choice of which subdevice to use is always unambiguous since for a given platform or card, only one subdevice is appropriate.

Subdevice workers typically have no control interface. Like device workers, they have signals that are attached to FPGA pins. These pins are usually what is “shared” between the device workers that the subdevice supports. It is possible that subdevices have no external signals and only exist to coordinate between several optionally present device workers.

Subdevice workers usually have connections to the device workers they support. The XML (a minimal OWD) for a subdevice defines:

- The hardware FPGA pins it is attached to (via the “signals” element like all device workers)
- The device workers it supports (using **`<supports>`** elements).
- How it is connected to each of the device workers it supports (using **`<connect>`** elements under **`<supports>`** elements).

Since platforms and cards declare which devices they have, including subdevices, they specify which subdevices are present. This allows different subdevices to be used for different platforms while leaving the device workers untouched and reusable.

Finally, when platforms have multiple devices of the same type, declared in the HDL platform's OWD, the platform (not the subdevice) can specify the unambiguous association between subdevices and devices, using <supported> elements in the platform's OHPD/OWD. See the Device Elements in a Platform XML File section for how these elements work.

[a repeated diagram with the two associations]

For help in understanding these aspects of OpenCPI, look closely at the **`lime_zipper_fmc_lpc`** card (**`hdl/cards/specs/lime_zipper_fmc_lpc.xml`**), and the devices that it includes. This card is used in the **`zed`** platform for certain configurations based on the **`hdl/platforms/zed/zed_zipper_fmc*`** files. To learn more about raw properties, take a look at the **`si5351`** device at **`hdl/devices/si5351.hdl`**. The many raw properties declared in the xml file correspond to hardware registers on the si5351 chip. See chapter 7/page 23 (“Register Map Summary") at **`https://cdn-shop.adafruit.com/datasheets/Si5351.pdf`**. Each raw property in the OpenCPI **`si5351.xml`** corresponds to a line in the datasheet table there.

Finally, for a further understanding of subdevices, take a look at **`lime_spi`** (in **`hdl/devices/lime_spi.hdl`**), which is a subdevice that handles raw property accesses and low level SPI functionality. It supports the lime tx/rx devices, which means that the lime tx/rx devices can delegate their raw property accesses to the **`lime_spi`** subdevice.

5.5.4.1 Using RawProp Ports with SubDevices

The example below defines a subdevice with no control interface (no properties or control operations), driving two signals that are an I2C interface, and supporting a si5351 clock generator device worker via a **rawprop** worker port (defined in <rawprop> XML Elements below). It is declaring that if that supported device is present, it should also be present and be connected to that device worker via the **rawprop** port. By including this subdevice in a board description file (in the platform's OHPD/OWD or a card's spec file), it will be associated *on this board* with the **si5351** device worker.

```
<HdlDevice language="vhdl">
  <componentspec nocontrol='true'>
  <rawprop/>
  <supports worker='si5351'>
    <connect port="rawprops" to="rawprops"/>
  </supports>
  <Signal name='sda' direction='inout'/>
  <Signal name='scl' direction='inout'/>
</HdlDevice>
```

The most common connection between a device worker and a subdevice that supports it is the **rawprop** connection that enables a device worker to delegate some or all of its raw property accesses to the subdevice. The device worker declares a **rawprop** master port for this delegation, and the subdevice declares an array of one or more a **rawprop** slave ports to support device workers this way. This type of connection is common when there is a shared control path like SPI or I2C to access the registers of several devices.

The **rawprop** port has a bundle (VHDL record) of signals that is identical to the raw signals defined in the Raw Access to Properties section of the ***OpenCPI HDL Development Guide***. This record is called **raw** in both the control interface signals (**props_in.raw**, **props_out.raw**) as well as the **rawprop** port signals (**rawprops_in.raw** and **raw_props_out.raw**).

A **rawprop** worker port consists of a VHDL record of signals that allow a device worker to easily delegate all of its raw property accesses to the subdevice, using this VHDL:

```
rawprops_out.present     <= '1';
rawprops_out.reset       <= ctl_in.reset;
rawprops_out.raw         <= props_in.raw;
props_out.raw            <= rawprops_in.raw;
```

The **present** signal tells the subdevice that there is a connection to a device worker it supports. The **reset** signal tells the subdevice that this device worker is being reset, and the **raw** subrecord conveys the raw property signals between the device worker and the subdevice. In the subdevice, the **rawprop** port may be an array port (with **count** attribute > 0) when the subdevice worker is supporting multiple device workers.

5.5.4.1 Using DevSignal Ports with SubDevices

When the **rawprop** connection is not sufficient for all of the shared functionality in the subdevice (raw properties, presence and shared control reset), another type of connection is used, which is a customized set of signals. This is called a **devsignal** port. The port is declared for both the device worker (or platform worker in some cases) and the subdevice worker using **devsignal** element.

The **devsignal** element declares the port and has four attributes: **name**, **master**, **signals** and **count**. The optional **name** attribute provides a port name, with the default being **dev**. The boolean **master** attribute defines a master/slave role for the port, which is used relative to signal directions. The **signals** *attribute* is the name of a file containing a top-level **signals** *element*, containing signal definitions for this port. The direction of the signals declared in the file are relative to the master port. Here is an example signals file (named **mydevsignals.xml**):

```
<signals>
  <signal name='DATA_CLK_P' direction='in'/>
  <signal name='DATA_CLK_N' direction='in'/>
  <signal name='SYNC_IN' direction='out'/>
  <signal name='ENABLE' direction='out'/>
</signals>
```

The master port (usually the device worker) would drive the **SYNC_IN** and **ENABLE** signals as outputs, and the slave port (usually the subdevice) would receive them as inputs. The port declaration in the OWD of the device workers would be:

```
<devsignal master='true' signals='mydevsignals'/>
```

Since the signals are common to the ports of both workers, they are in their own file, usually in the **specs** directory of the component library containing both device and subdevice workers (usually **hdl/devices/specs**). Since the purpose of the subdevice is normally to share/multiplex output signals among more than one device worker, the subdevice's port declaration normally includes a count to indicate that the port is actually an array of identical ports. So the subdevice port declaration would be something like:

```
<devsignal signals='mydevsignals' count='2'/>
```

As with the **rawprops** examples above, the connection of these ports is indicated by the **supports** element and its **connect** child element. See the <devsignal> XML Element section below.

5.5.5 Testing Device Workers with Emulators

The OpenCPI unit test framework described in the OpenCPI Component Development Guide can be used to test device workers, using a special type of device worker called an ***emulator***.

An emulator device worker is actually a device *emulator* that emulates a device for test purposes. Thus while a normal device worker manages and controls a device by driving and receiving signals from the device (via FPGA pins), the emulator *acts like the*

device. So if the device has a reset input pin, the normal device worker will drive that reset signal as an output of the device worker, into the actual device. The emulator for the device will have that reset signal as an *input* signal.

We discuss this relationship such that:

- We use “emulator” to mean “emulator worker”, which is a special type of device worker (much like a platform worker is a special type of device worker).
- The device worker manages a device, and may have an associated emulator, which is ***its*** emulator.
- The emulator emulates a device which has a device worker, which is ***its*** device worker.

There is nothing about emulators that restricts them to running only in simulators, so when it is useful, they can be written to be synthesizable and executed in hardware.

When used in the unit test framework, the emulator is automatically instanced to test its device worker.

5.5.5.1 Emulator Signals, Ports and Properties

An emulator establishes the relationship with its device worker using the top level **`emulate`** attribute whose value is the name of the device worker for the device it emulates. An emulator automatically inherits the signals from its device worker, with the directions reversed. Thus no **`signal`** elements need be defined in an emulator's OWD.

Similarly, for non-data, non-control ports (usually **`rawprop`** or **`devsignal`** ports), the emulator has the same ports defined, with the same name, with the opposite master attribute value. I.e. when the device worker is a master of such a port, the emulator is a slave. The emulator has its own control port, and may have its own data ports (e.g. for recording test data into a file).

Finally, the emulator also inherits all the parameter and writable/initial properties of its device worker so that when used, it sees the same parameter and property values that its device worker sees. It can then emulate accordingly. It can have its own additional properties of course, but it *cannot* have its own parameters with multiple values or mentioned in test cases. Its build configurations are the same as its device worker's build configurations.

When testing a device worker with its emulator, both the emulator and the device worker are instantiated, their signals are connected, and the non-control, non-data ports are connected between them. Thus they form a test unit (UUT) where each has its own control port, and each may have its own data ports.

An example is where a device worker for an ADC, might have an emulator that takes test data from a file, and feeds the data into the ADC device worker as if it was the ADC hardware device.

5.5.5.2 Emulator Worker OWD XML Files

An emulator's OWD references the OCS for all emulators, **emulator-spec**, using its **spec** attribute. It identifies its device worker using the **emulate** attribute, whose value must include the **.hdl** model suffix. An emulator must have a control interface so it cannot set the **nocontrol** attribute to **true**.

An example emulator OWD is:

```
<HdlDevice emulate='mydevice.hdl' spec='emulator-spec'>
  <property name='errorcount' volatile='true'/>
  <streaminterface name='tracedata' producer='1'/>
</HdlDevice>
```

This OWD says that the emulator will have a volatile property **errorcount** to report the number of errors encountered while observing its device worker's behavior. This property is in addition to all its device worker's parameters and writable properties that are automatically inherited.

The **streaminterface** element is directly introducing a data output port even though there is no such port in the OCS, since emulators are not required to have such ports.

The “results” of running the emulator would be the **errorcount** property's value and the data produced at its **tracedata** output port.

An emulator OWD has these restrictions when compared to a normal device worker:

- It must implement/reference the **emulator-spec** (via **spec** attribute)
- It must reference its device worker via the **emulate** attribute
- It cannot have any property whose name conflicts with any parameter or writable property of its device worker.
- It cannot have any data port whose name conflicts with any of its device worker's ports.
- It cannot have any signals (it will inherit its device worker's signals)

5.5.5.3 Using an Emulator for Device Worker Unit Testing

When a **.test** directory is defined for a device worker (using the **ocpidev create test** command), OpenCPI expects to find an emulator worker in the same component library and will instantiate and connect it next to the device worker in all test assemblies.

All the testing then proceeds the same as testing application workers with these additions:

- The final values for the emulator's own properties is also available to verification scripts.
- Data input ports of the emulator must be supplied with data with input files or generator scripts.
- Data output ports on the emulator will captured and available to verification scripts.

Essentially the UUT becomes the combination of a device worker and its emulator.

If the emulator is written to be synthesizable, test execution can include hardware platforms as well as simulator platforms.

Note that emulator workers are always started before the device worker they are associated with.

5.5.5.4 Using an Emulator in Containers without the Unit Test Framework

When a more complex testing configuration is needed beyond the “one device worker with its emulator” scenario describe above, it can be done by instantiating device workers and emulators directly in a container using the “floating device” feature.

Floating devices are simply those that are not really part of the platform being targeted (usually simulators in this case), but are devices instantiated and connected directly to their emulators. This allows for test configurations that combine device workers, subdevice workers, and emulators in various ways.

An example container XML file that uses emulators is:

```
<HdlContainer platform='isim'>
   <device worker='lime_spi_em' floating='1'/>
   <device worker='lime_spi' floating='1'/>
   <device worker='lime_tx' floating='1'/>
   <device worker='lime_tx_em' floating='1'/>
</HdlContainer>
```

This example instantiates two device workers (**`lime_spi`** and **`lime_tx`**) as well as their emulators. The emulators are automatically wired up with the device workers they are emulating. They all use the **`floating`** attribute since they are not defined as existing on the **`isim`** platform.

[Preliminary feature with limited support]

5.5.6 Higher-level Proxies Suitable for Applications

As discussed above, we use ***device proxies*** to normalize (or make more user-friendly) the behavior of a class of devices. The granularity of such classes is sometimes below the level appropriate for applications, but is optimal for sharing, reuse, and rapid enablement of new platforms.

An example of fine granularity is a clock generator chip. There are many such chips, and device workers are written for them. They should all act the same, in terms of how they are set up and programmed, usually using a device proxy. When users or applications want access to a clock generator device, they should be able to use it the same way as any other clock generator device.

However, clock generator chips are also frequently used to drive other devices to a specific clock (e.g. sampling) frequency, and there may be some specific relationship on a given platform or card between specific clock generator chips and the devices they provide clocking too.

So, as an alternative to (or in addition to) device proxies that are defined for the granularity of individual devices, that have device workers, OpenCPI provides for higher level proxies which have *multiple* slaves***,*** for presenting a collection of devices and

proxies to applications as something higher level to connect and configure within the application. A good example is the Digital Radio Controller (DRC) proxies as described in Digital Radio Controller Proxies.

The purpose of these multi-slave proxies is to remove all device specifics from applications, rather than simply normalize the behavior of a device to its class. Applications and users want to see standard, portable interfaces for endpoints (e.g. sources and sinks of radio data). Multi-slave proxies make that possible.

As with subdevices at the bottom of the OpenCPI "device support" stack, these multi-slave proxies are at the top of the "device support" stack, appropriate for use by applications. But both may *internally* be platform specific. Both exist to "leave the device workers alone" so that they are reusable across platforms and cards.

[Insert diagram for "stack"]

Applications use these higher-level proxies by instantiating a component of a higher level proxy spec. These proxies can represent any application-level subsystem or source or sink of data, and they can delegate connections to their IO ports to connect to the ports of their slaves.

5.5.7 XML Metadata for Device Workers/Subdevices/DeviceProxies

The four types of workers that relate to device support in OpenCPI are:

- Device Workers
- Device Proxies
- Subdevices
- Device Emulators

All these are workers and share the XML structure of workers via the OWD for ports and properties.

Device workers use the normal top-level **`spec`** attribute to identify the class of the device. Device workers and subdevice workers use the **`HdlDevice`** top level XML tag, have **`signal`** elements for hardware signals, and may have **`rawprop`** and **`devsignal`** ports to connect device workers to subdevice workers. If a device worker uses a subdevice worker, it may in fact have no **`signal`** elements.

Subdevice workers have **`supports`** child elements describing which device workers (i.e. device types) they support and how they should be connected to them.

Device proxies have a top-level **`slave`** attribute identifying which worker they are a proxy for. The name should include the authoring model suffix, such as:

```
slave='adc-chip123.hdl'
```

Multi-slave proxies for controlling a group of devices specify multiple slaves, as described in the proxy section of the OpenCPI RCC Development Guide.

The **`signal`** elements in the OWD for devices and subdevices are as described above for platform workers in Signal Declaration XML Elements.

5.5.7.1 RawProp XML Elements for Device Workers and Subdevice Workers

The **`rawprop`** child element identifies a port of the worker that forwards/delegates the raw property signals from a device worker's control interface to a subdevice worker. It may be an array port (typically at the subdevice). Its attributes (all optional) are:

Name — The name of the port (default is **`rawprops`**)

Optional — Indicates if a connection to this port is not required (default **`false`**). Normally true on subdevices supporting multiple optional devices.

Count — Indicates if specified > 0 that this is an array of raw property ports (default is `0`, meaning *not* an array port). Normally only specified on a subdevice supporting multiple devices.

Master — Boolean Indicating who generates addresses for raw accesses (usually the device worker sets it **`true`** and the subdevice leaves it **`false`**).

The port is an array port if **`count`** is specified and greater than 0 or if the **`count`** attribute value is an expression based on parameter values.

5.5.7.2 DevSignal XML Elements for Device Workers and Subdevice Workers

This XML element represents a custom signal bundle that is connected between device workers and subdevices. It has these attributes:

Name — the name of the port (the default is **`dev`**).

Optional — whether this port must be connected or not.

Count — indicates an array of similar ports with the same signals (when > 0 or an expression)

Signals — indicates a file containing a top level **`signals`** element consisting of **`signal`** child elements. The **`.xml`** suffix is not required in the attribute.

Master — Boolean Indicating who is the master, which simply is the side where the direction of the underlying signals is referenced to.

The file indicated by the **`signals`** attribute enumerates the signals in the bundle, similar to the **`signal`** elements in a device worker's OWD. The direction of the signals is relative to the master. If the signal is declared as “input”, that means it is an input signal to the master, and an output from the slave.

5.5.7.3 The <Supports> XML Element for Subdevices

Subdevices indicate which device workers (i.e. device types) they *support* by using **<supports>** XML elements: the subdevice is declaring that it ***supports*** the specified device worker when that worker is present (in platform configuration XML or in container XML). To indicate how the subdevice is connected to a device worker it supports, it specifies **<connect>** child elements within the **<supports>** elements, e.g.:

```
<supports worker='lime_dac'>
  <connect port='rawprops' to='rawprops' index='1'/>
  <connect port='dev' to='dev' index='1'/>
</supports>
<supports worker='lime_adc'>
  <connect port='rawprops' to='rawprops' index='0'/>
  <connect port='dev' to='dev' index='0'/>
</supports>
```

The attributes of the **<supports>** element in the subdevice's OWD are:

worker — the name of the device worker this subdevice supports

The attributes of the **<connect>** child element of the **<supports>** element are:

port — the port on the *supported* device that should be connected to this subdevice

to — the port on this supporting subdevice to be connected

index — index into this subdevice's port array (when its count attribute is > 0).

It is possible that different instances of the same device on a platform or card are supported by entirely different subdevices or by a single subdevice.

5.5.8 Associating Device Workers and Subdevice Workers with Platforms and Cards

Both device workers and subdevice workers are enumerated in the XML description of a platform or card using the **device** child element and the **worker** attribute. This declares the existence of the device on the platform or card as well as the device worker that is used. The order of these elements defines the ordinals of the devices when multiple instances of the same device type (device worker) are present.

The presence of subdevice workers in this declared list makes them available to support any devices that are used in a platform configuration or container. When a device is used (instantiated based on the platform configuration XML or the container XML), any subdevices that exist on the platform that *support* the device will also be instanced and connected.

5.5.9 Design Pattern for ADC and DAC Device Workers

When supporting ADC and DAC devices with device workers in OpenCPI (for use on platforms or cards that have them), OpenCPI specifies interfaces to be used for the digital data input and output. These interfaces are then connected to OpenCPI-provided "helper" workers that perform a number of related functions generically. This simplifies the device-specific ADC and DAC device workers.

For Quadrature ADC devices:

- the generic helper worker is called **`data_src_qadc`**
- the **`devsignal`** interface is based on the file **`qadc-16-signals`**

For Quadrature DAC devices

- the generic helper worker is called **`data_sink_qdac`**
- the **`devsignal`** interface is based on the file **`qdac-16-signals`**

This section describes the ADC/DAC interfaces and the provided helper workers.

5.5.9.1 ADC Device Workers

The following are requirements for an ADC device worker:

1. Supply the sample clock of the ADC device as an output.
2. Produce data in the ADC's sample clock domain (no clock domain crossing), one clock per sample.
3. Indicate if possible, when the sample data for the clock period is not valid (i.e. if/when the ADC knows it is non producing usable samples).

The following diagram shows how the device-specific ADC device worker fits into the platform, with the ADC helper worker (**`data_src_qadc`**) and optional timestamper worker providing the appropriate messages to the rest of the system.

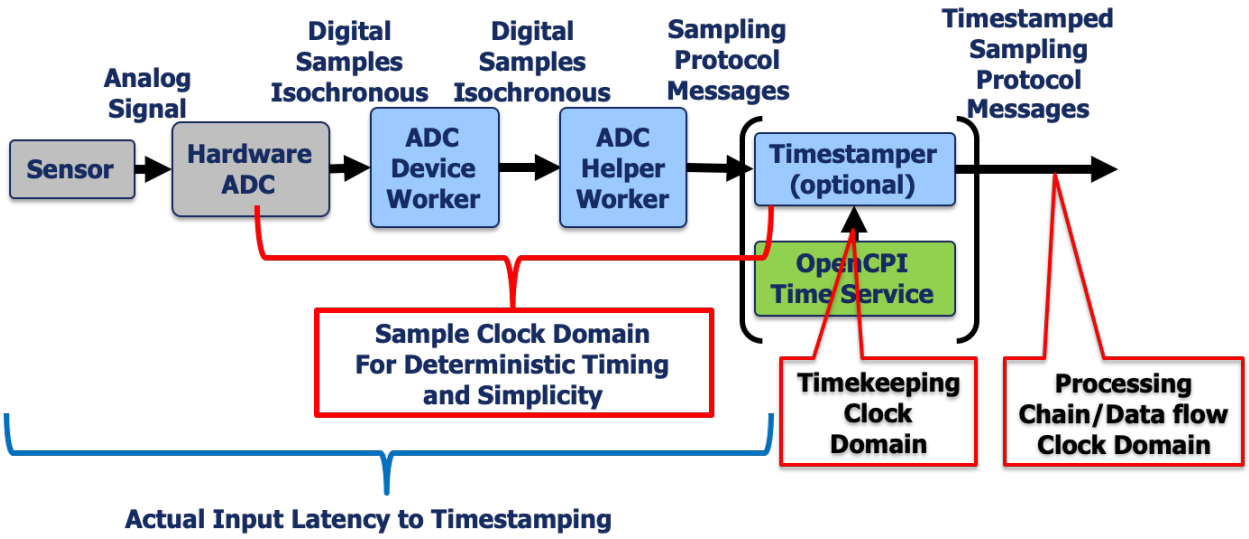

Figure 10: ADC Device Support Model

The **data_src_qadc** helper performs the following functions, so the hardware device worker should not:

1. Detect overrun and record it in a volatile property
2. Produce protocol messages indicating sample discontinuity (for overruns or invalid data).
3. Normalize bit width of output data, possibly sign-extending if requested.

This helper worker has a producer streaming data port which implements the *ComplexShortWithMetadata* protocol. This worker accepts input data using the **qadc** interface. The following table describes the signals in that interface. The **data_src_qadc** worker serves as the master of that interface, and the direction described is with respect to **data_src_qadc**.

The **data_src_qadc** worker represents a single channel of I/Q sample data. If the actual ADC device is multichannel, it will have separate interfaces connected to separate **data_src_qadc** workers for each channel.

Table 9: qadc-signals Signals

Signal	Direction	Description
clk	**input**	Sample clock associated with **data_i** and **data_q**, rising edge.
data_i	**input**	Sample data corresponding to **ComplexShortWithMetadata** protocol's **samples** argument's **I** member
data_q	**input**	Sample data corresponding to **ComplexShortWithMetadata** protocol's **samples** argument's **Q** member
valid	**input**	The **data_i** and **data_q** data for this sampling interval is valid.
present	**output**	Indicator to the connected ADC device worker that there is a connection

When support is added for a new type of ADC, a device worker must be created with a slave **qadc** **devsignal** interface to feed the **data_src_qadc**. Its function is to adapt between this interface and the hardware interface of ADC device. For example, some ADC devices, such as RF digitizers, may interleave I and Q or interleave data from multiple ADCs when streaming data over the device's bus. It is the job of an ADC device worker to de-interleave the data from this bus and present each ADC's samples on a distinct parallel signal bus as defined by the **qadc** interface.

To use this ADC, both the **data_src_qadc** and the device-specific worker must be declared in any platform OWDs or card specifications where the device exists. The device-specific worker must have a **supports** element in its OWD, declaring that it supports the **data_src_qadc**, and is connected to its **qadc** interface. The inclusion of the **data_src_qadc** worker in the platform or card XML should set any of its parameter properties appropriately, especially the **ADC_WIDTH_BITS**, and **ADC_INPUT_IS_LSB_OF_OUTPUT_PORT** parameter properties.

When a single ADC device has multiple channels, the ADC device worker may have multiple ports that are supported by different helper workers, where the ports are declared as optional. This allows a subset of the ports/channels to be used in a given platform configuration or container. In this case, the **present** signal indicates to the device worker which ports/channels are actually being used.

Platform configurations or containers can specify a **data_src_qadc**, causing the ADC worker to be included automatically (due to the **supports** relationship). Such assemblies can be used by applications by simply instantiating the **data_src_qadc** component in the application.

5.5.9.2 DAC Device Workers

The following are requirements for DAC device workers:

1. Supply the sample clock of the DAC device as an output
2. Process data in the DAC device clock domain (no clock domain crossing)
3. Utilize, as appropriate for the hardware, an indication that the data for the current sampling interval is valid or not

The **data_sink_qdac** helper performs the following functions, so the hardware device worker should not:

1. Interpret protocol messages indicating EOF or lack of valid data.
2. Normalize bit width of output data
3. Provide underrun indicator when rate of input data is less than DAC sample rate
4. Indicate to the DAC whether the current output sample has valid data.

This worker has a consumer WSI data port which accepts the *ComplexShortWithMetadata* protocol. This worker produces data using the **qdac** interface. The following table describes the signals in that interface. The **data_sink_qdac** worker serves as the master of that interface, and the direction described is with respect to **data_sink_qdac**.

Table 10: qdac-signals Signals

Signal	Direction	Description
clk	**input**	Sample clock associated with **data_i** and **data_q**, rising edge.
data_i	**output**	Sample data corresponding to **ComplexShortWithMetadata** protocol's **samples** argument's **I** member
data_q	**output**	Sample data corresponding to **ComplexShortWithMetadata** protocol's **samples** argument's **Q** member
valid	**output**	**data_i** and **data_q** signals hold valid sample data for this period
present	**output**	Indicator to the connected device worker that there is a connection

When support is added for a new type of DAC, a device worker must be created with a slave **qdac** interface to interface with **data_sink_qdac**. Its function is to adapt between this interface and the interface of a DAC device. For example, some DAC devices will require interleaved I and Q data to multiple DACs when streaming data over the device's bus. In this case, it is the job of a DAC device worker to interleave the data to this bus and accept each DAC's datastream as a distinct parallel signal bus as defined by the **qdac** interface.

Both the **data_sink_qdac** and the device-specific worker must be declared in any platform OWDs or card specifications where the device exists. The device-specific worker must have in its OWD, a **supports** element declaring that it supports the **data_sink_qdac**, and is connected to its **qdac** interface.

Any platform configuration or container can specify a **data_sink_qdac** worker, and that will cause the device-specific worker to be included automatically (due to the **supports** relationship). Such assemblies can be used by applications that simply ask for the **data_sink_qdac** component.

5.5.10 Summary of Worker Types for Supporting HDL Devices

Device Workers directly control and attach to physical devices, as "device drivers", and generally implement the data sheet for the device, providing access and visibility to the device's native registers and capabilities.

Subdevice Workers enable multiple device workers to share some underlying hardware, like shared resets, shared SPI or I2C busses. They also allow device workers to stay portable when low level modules differ by platform or card.

Proxy Workers (for device workers) provide a higher level and more generic interface to make the device look more like others in its class, providing more user and software-friendly access and visibility to the device's capabilities.

Emulator workers are used to test device workers by providing the mirror image of the device worker's external signals so they can emulate the device in simulation.

5.6 Defining Cards Containing Devices that Plug into Slots of Platforms

A card is specified in a card definition XML file initially created using the **`ocpidev create hdl card <cardname>`** command, which creates the file ***`<cardname>-card`*`.xml`** file in the **`hdl/cards/specs`** directory. This file has a top-level **`card`** XML element with a required **`type`** attribute, and contains **`device`** elements.

The **`type`** attribute is the slot type and must match the name of a defined slot type.

The **`device`** elements declare device instances on the card, and act the same as **`device`** elements in an HDL platform XML file with two additional aspects:

First, when property values are specified in the device declaration, a **`platform`** attribute may be used to indicate the platform that the value should be applied to. I.e. if the platform attribute is specified, the value will only be applied when the card is used with (plugged into) the platform. If no platform attribute is present for a property value, it will apply to all platforms. This allows values to be platform-specific. This is useful when the value of a device's parameter depends on the *combination* of card and platform. The example below indicates that for this device on this card, the **`Data_CLK_Delay`** parameter should be set to 5, unless the card is being used on the **`zcu104`** platform, in which case the value should be 7.

```
<device worker='lime_adc'>
  <property name='Data_CLK_Delay' value='5'/>
  <property name='Data_CLK_Delay' value='7' platform='zcu104'/>
</device>
```

In both platform XML and card XML files the **`signal`** child elements of **`device`** elements indicate a non-default mapping between device worker signals and platform or card-level signals. Whereas the signal mapping on platforms use the **`platform`** attribute for the platform/card-level signal name, in card definition files, the **`card`** attribute is used. This makes it clear that in the case of cards, you are mapping a device worker's signal to a card signal.

Card signal names in this case indicate the slot type's signals. Thus each device instance's signals are essentially wired to slot pins.

Here is example of a simple card definition file with one device. It has one required parameter property setting (**`use_ctl_clk`**), one device signal that is not available on the card (**`tx_clk`**) and some other signals mapped to card signals that are derived from the slot type:

```
<card type='fmc_lpc'/>
  <device worker='lime_dac'>
    <property name='use_ctl_clk' value='true'/>
    <Signal name="tx_clk"    slot=''/>
    <Signal name='tx_clk_in' slot='LA04_N'/>
    <Signal name="tx_iq_sel" slot='LA29_N'/>
    <Signal name="txd(0)"    slot='LA25_N'/>
    <Signal name="txd(1)"    slot='LA25_P'/>
    <Signal name="txd(2)"    slot='LA22_N'/>
    <Signal name="txd(3)"    slot='LA22_P'/>
    <Signal name="txd(4)"    slot='LA20_N'/>
    <Signal name="txd(5)"    slot='LA20_P'/>
    <Signal name="txd(6)"    slot='LA16_N'/>
    <Signal name="txd(7)"    slot='LA16_P'/>
    <Signal name="txd(8)"    slot='LA12_N'/>
    <Signal name="txd(9)"    slot='LA12_P'/>
    <Signal name="txd(10)"   slot='LA08_N'/>
    <Signal name="txd(11)"   slot='LA08_P'/>
  </device>
</card>
```

5.7 Reporting FPGA Platform Timekeeping Characteristics

[Preliminary feature documentation]

When enabling an FPGA platform, characteristics of the timekeeping clock of the system should be reported as part of platform documentation.

The following characteristics of the timekeeping clock must be reported:

- Frequency Error (Hz)
- Phase accuracy (s)
- Jitter (s)
- Short term stability (Allan Variance)

In addition to these characteristics, the ability to accurately provide time to HDL workers must be measured and reported.

The following characteristics regarding timestamping accuracy must be reported:

- Timestamp resolution (s)
- Timestamp accuracy (s)

OpenCPI provides applications for collecting these measurements given a particular hardware setup. The following section describe the required hardware test setups for collecting these measurements.

5.7.1 Timekeeping Clock Characteristics

The test setup for measuring timekeeping clock characteristics can be seen in Figure 11: Timekeeping Clock Measurement Setup. The setup requires one dedicated input to the FPGA and one dedicated output from the FPGA. The dedicated input is a PPS signal sourced from a high precision frequency standard and connected to the PPS input of the Hardware Time Server (HTS) inside the FPGA design. The dedicated output is connected to the PPS output signal generated by the HTS, which is derived from the HTS clock and PPS inputs. The output signal is then connected to a high precision frequency counter which measures the required performance characteristics.

Given this test setup, running the **`timekeeping_clock_characterization`** application in the **`platform`** project will report the required characterization values for the platform under test. The following sections describe how the application collects those characteristics from the test setup.

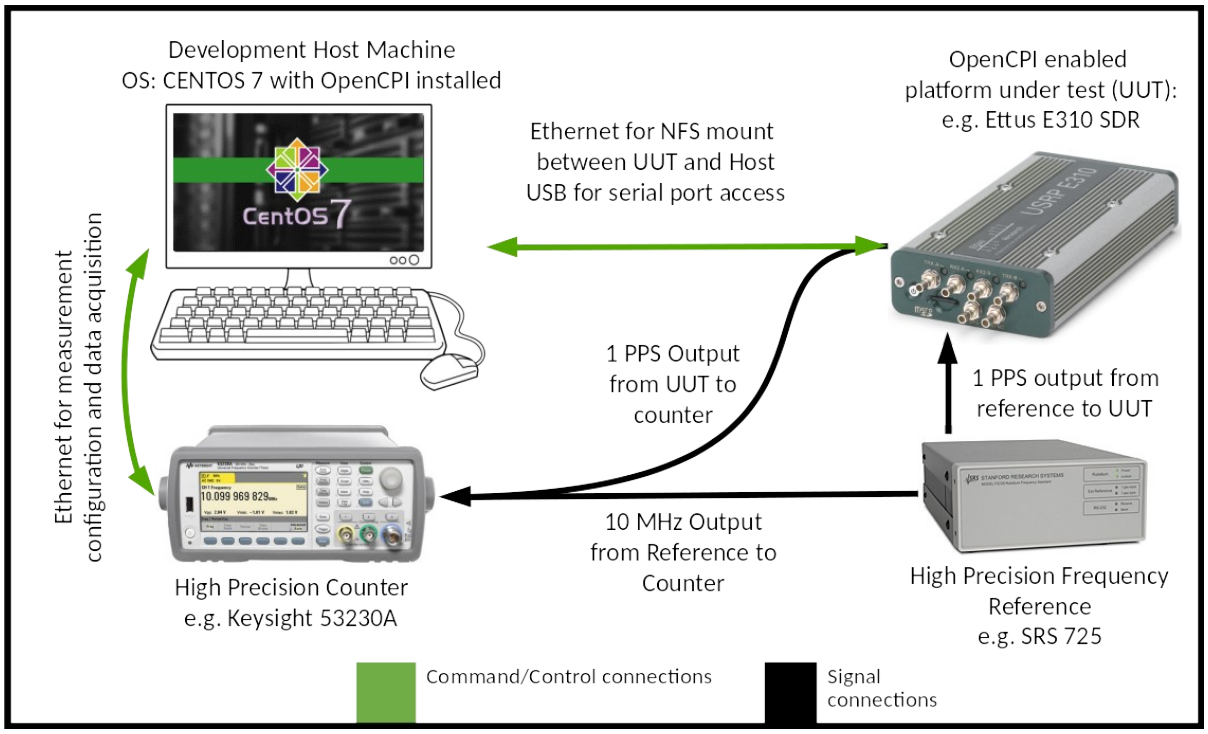

Figure 11: Timekeeping Clock Measurement Setup

5.7.1.1 Frequency Error (Hz)

The Frequency Error is measured with the high precision frequency counter. 100 frequency measurements are made and the Frequency Error is reported as the maximum and minimum frequency of the PPS generated by the HTS.

5.7.1.2 Phase Accuracy (s)

The Phase Accuracy is measured with the high precision frequency counter. 100 time interval measurements are made between the rising edge of the PPS from the precision frequency standard and the rising edge of the PPS generated by the HTS, and the Phase Accuracy is reported as the average time interval.

5.7.1.3 Jitter (s)

The Jitter is measured with the high precision frequency counter. 100 period measurements are made on the PPS generated by the HTS, and the Jitter is reported as the standard deviation of the the period measurements.

5.7.1.4 Short Term Stability (Allan Variance)

The Allan Variance is measured with the high precision frequency counter. 100 frequency measurements are made of the PPS generated by the HTS and the Allan Variance is reported for both 10 and 100 frequency measurements.

5.7.2 Timestamping Accuracy

The test setup for measuring accuracy can be seen in Figure 12: Timestamping Accuracy Measurement. A GPS signal is sourced to both the on-platform GPS device and a GPS frequency reference. This requires a dedicated FPGA input connected to the GPS time phase-aligned PPS output from the GPS frequency reference.

This dedicated FPGA input must be connected to the signal input of the **`signal_time_tagger.hdl`** device worker in the **`platform`** project's **`hdl/devices`** library. This worker collects timestamps at the rising edge of an FPGA input signal.

In this test setup, running the **`timestamping_accuracy`** application in the **`platform`** project reports the characterization values for the platform under test. The following sections describe how the application collects those characteristics from the test setup.

5.7.2.1 Timestamp Resolution (s)

The Timestamp Resolution is the period of the clock provided to the HTS.

*5.7.2.2 Timestamp Accuracy (s)**

The Timestamp Accuracy is measured by collecting timestamps from 100 consecutive PPS inputs sourced from the GPS frequency reference and computing the difference from the nearest integer second. The accuracy is reported as the minimum, maximum, and standard deviation of these differences.

**Note that the fractional second PLL in the HTS settles over time, so the highest amount of inaccuracy will occur out of reset. Because of this settling, accuracy measurements must be reported out of reset or immediately following the load of a new FPGA bitstream.*

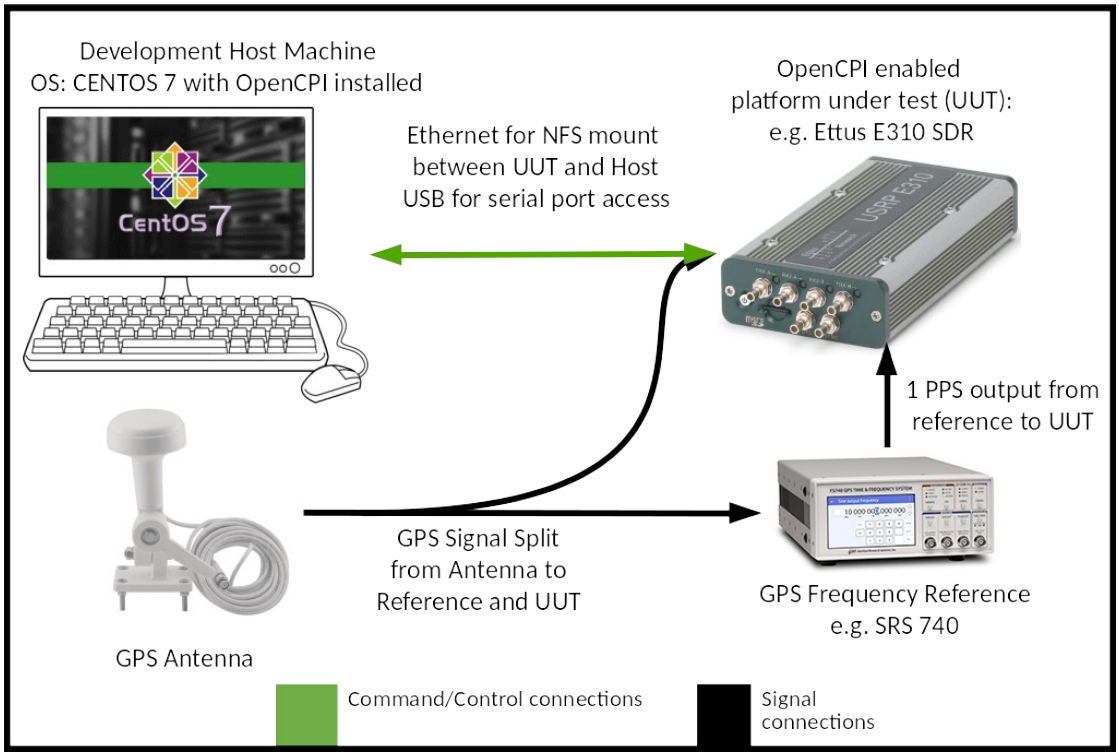

Figure 12: Timestamping Accuracy Measurement

5.8 Supporting the Digital Radio Control (DRC) Component Specification

The DRC component is used when an application needs to use radio hardware in the system, for the purpose of controlling it and streaming sample data to and from it. The "digital radio" function in a system usually has antennas for transmitting and receiving RF signals, and ***channels*** which convert the RF signals to and from base-band digital samples that are produced and consumed by the application.

Usage of the DRC component by application developers is described in the ***OpenCPI Application Development Guide***. The DRC section of that document should be read as a prerequisite for this section. This section describes how a platform developer can implement support for the DRC component specification on a given system, in order to support SDR applications (applications that need to use radio hardware) on the system.

This support involves writing a system-specific RCC proxy worker that implements the DRC specification, as defined in the file **`drc-spec.xml`**. This proxy worker in turn controls the various radio-related devices in the system which are usually supported by FPGA/HDL device workers for that hardware. Radio-related hardware (and thus device workers) can include ADCs, DACs, integrated RF transceivers, clock generators, GPIOs, antenna switching, RF filter banks, power amplifier controls — any hardware that is needed to manage hardware resources that deliver the radio input/output functions as defined by the DRC.

As with all workers, the DRC proxy worker implements its spec (OCS), has an OWD XML file describing it, and has one or more source code files. If the system being supported lives in its own supporting project (OSP), then this worker would normally be in the **`hdl/devices`** or **`hdl/cards`** subdirectory in that project.

DRC proxy workers usually have the name:

`drc_<hdl_platform>.rcc`

when the DRC worker is specific to an HDL platform that is attached to radio hardware. If the DRC is specific to a plug-in card with radio hardware, then the DRC worker is specific to that card, but not specific to an HDL platform, and thus would be named:

`drc_<hdl_card>.rcc`

5.8.1 Preparing the Worker Descriptor XML (OWD) for a DRC Proxy Worker

There are several aspects to preparing a DRC worker's OWD XML file:

- Specifying which properties for controlling DRC configurations notify the worker code when they are written
- Specifying the constant property values of the implementation and the associated radio hardware,
- Specifying the required slave workers and their properties and interconnections, which will be used and controlled by the DRC proxy worker.

After the OWD XML file is prepared, the worker source code skeleton is re-generated based on the prepared XML when the **`ocpigen build`** command is issued.

5.8.1.1 Specifying Notifications for Properties that Control DRC Configurations

As described in the ***Properties for Controlling DRC Configurations*** section in the ***OpenCPI Application Development Guide***, there are several properties which, when written, change the state of a specified DRC configuration. These are the **`prepare`**, **`start`**, **`stop`**, and **`release`** properties. The value of each property is an ordinal indicating which configuration is being controlled, and thus the following lines must be placed in the DRC proxy's OWD:

```
<specProperty name='prepare' writesync='1'/>
<specProperty name='start' writesync='1'/>
<specProperty name='stop' writesync='1'/>
<specProperty name='release' writesync='1'/>
<specProperty name='status' readsync='1'/>
```

These **`writesync`** attributes indicate that the worker will be notified when these properties are written. In the case of the last one, the status property, the **`readsync`** attribute indicates that the proxy should be notified just prior to the status being queried. This allows the proxy to refresh the status, perhaps based on querying the hardware (i.e. the underlying slave device workers).

5.8.1.2 Constant Values in the OWD for a DRC Proxy Worker

The DRC spec defines a number of properties that are considered constant values for a given DRC proxy worker. These properties are defined in the OCS in one of two ways:

1. As an explicitly constant parameter property, defined in the OCS with **`parameter='`**true' and providing a default value that can be overridden in the DRC proxy's OWD using, e.g.:

   ```
   <specproperty name='MAX_CHANNELS' value='2'/>
   ```

2. As a readable property set in the OCS *without* **`parameter='true'`**, which may be resolved in the worker OWD either by:

   - *Having no mention in the OWD, which implies that the DRC proxy will provide a value at runtime during initialization before or during start.*
   - *Specifying in the OWD that the property is in fact a compile time constant by using:*

     `<specproperty name='<prop>' parameter='true' value='12'/>`

The following table describes these constant properties and whether they *must* be constants (upper case) or can be either constants or determined at initialization time.

Table 11: DRC Constant Properties

Name	Default in OCS	Description
MAX_CONFIGURATIONS	**2**	Maximum number of configurations supported. This is not platform-specific.
MAX_CHANNELS	**2**	Maximum number of RX or TX channels in any configuration.
MAX_RX_CHANNELS	**1**	Maximum number of RX channels in any configuration. This sizes the proxy's RX port.
MAX_TX_CHANNELS	**1**	Maximum number of TX channels in any configuration. This sizes the proxy's TX port.
MAX_RX_RF_PORTS	**1**	Number of RX RF ports, as specified in the **rf_ports_rx**
MAX_TX_RF_PORTS	**1**	Number of TX RF Ports, as specified in **rf_ports_tx**
rf_ports_rx		Names of RF ports capable of RX, first being default. Type is sequence of strings
rf_ports_tx		Names of RF ports capable of TX, first being default. Type is sequence of strings
min_rx_tuning_freq_MHz		Type is **double**
max_rx_tuning_freq_MHz		Type is **double**
min_rx_bandwidth_3db_MHz		Type is **double**
max_rx_bandwidth_3db_MHz		Type is **double**
min_rx_sampling_rate_Msps		Type is **double**
max_rx_sampling_rate_Msps		Type is **double**
supports_rx_samples_complex		Type is **bool**
supports_rx_samples_real		Type is **bool**
supports_rx_gain_mode_auto		Type is **bool**
supports_rx_gain_mode_manual		Type is **bool**
min_tx_tuning_freq_MHz		Type is **double**
max_tx_tuning_freq_MHz		Type is **double**
min_tx_bandwidth_3db_MHz		Type is **double**
max_tx_bandwidth_3db_MHz		Type is **double**
min_tx_sampling_rate_Msps		Type is **double**
max_tx_sampling_rate_Msps		Type is **double**
supports_tx_samples_complex		Type is **bool**
supports_tx_samples_real		Type is **bool**

5.8.1.3 The `slaves` element in the DRC Proxy Worker OWD

The second key aspect of a DRC proxy's OWD is to specify which workers should act as slaves for the DRC proxy along with their property values and interconnections. This is done by defining a ***slave assembly*** under the **<slaves>** element in the OWD. This

assembly is quite similar to an HDL assembly and will be described mostly as how it is different from a normal HDL assembly. All the instances in this slave assembly are considered slaves of the proxy and thus are accessible to the proxy worker code as described in the proxy/slave section of the ***OpenCPI RCC Development Guide***.

During the execution of an application, when a DRC proxy worker is chosen as an implementation for a DRC component instance in the application, all the slave instances mentioned in the proxy worker's slave assembly will automatically become part of the application. Any connections among the slave instances also become part of the application.

External ports of the slave assembly are associated with the ports of the DRC proxy itself. This means that all external ports of the slave assembly must have names that correspond to the names of ports of the DRC proxy worker. Thus when the application makes connections to the DRC proxy worker's ports, those connections are actually connected to the external ports of the slave assembly (and thus to ports of slave instances). We say that the proxy's ports are *delegated* to the underlying slave ports.

The slave workers implement “channels”, using some combination of device workers specific to the radio platform and DSP workers doing work that the hardware devices cannot do. Some hardware is more capable than others, so these DSP workers are essentially implementing functionality that might be available in some radio hardware, but is not available in the particular radio hardware present in the DRC's platform. Instances of device workers must have the same property settings as are specified in the HDL platform or card XML.

To support radios that have multiple channels, the DRC proxy worker's **`rx`** and **`tx`** ports are ***array ports***, meaning they act as an indexable set of ports under a single name and direction. Thus when an RX channel is enabled in a DRC configuration, the data will be produced on a slave port that corresponds to (is delegated from) the proxy's **`rx`** port, which has a connection in the application for receiving baseband sample data. On a two-channel radio, the DRC proxy's port would have an array count of two, and the underlying slave assembly would have different slave workers connected to those two elements of the array port. An application would have different connections to the proxy's **`rx`** port with **`index='0'`** for the first channel and **`index='1'`** for the second channel. Here is an example of a two-rx-channel slave assembly where two device workers are used to receive two channels of received data.

```
<slaves>
  .... slave instances ....
  <external name='rx' count='2'/>
  <connection>
    <port name='out' instance='adc0'/>
    <external='rx' index='0'/>
  </connection>
  <connection>
    <port name='out' instance='adc1'/>
    <external='rx' index='1'/>
  </connection>
</slaves>
```

The DRC spec (OCS) already defines the **rx** port as:

```
<port name='rx' count='MAX_RX_CHANNELS' producer='1' optional='1'/>
```

An example of a two-channel DRC proxy is the one for the FMCOMMS2/3 card, found in **projects/assets/hdl/cards/drc_fmcomms_2_3.rcc**.

5.8.2 Wiring the DRC Proxy Source Code

DRC proxies are RCC workers written in C++, and like all RCC workers, the code skeleton is initially generated based on the OWD for the proxy. There is a helper class for DRC proxies, **OCPI::DRC::ProxyBase**, that should be used when writing a DRC proxy. The DRC worker class should inherit this helper class. This class is available by including the file **OcpiDrcProxyApi.hh**, *after* the line in the file that "uses" the worker's namespace containing its generated data types. In the example below, for a proxy worker whose name is **drc_example.rcc**, the extra lines to use this helper class are underlined and colored in blue:

```
#include "drc_example-worker.hh"

using namespace OCPI::RCC; // for access to RCC types and constants
using namespace Drc_exampleWorkerTypes;

#include "OcpiDrcProxyApi.hh" // must be after "using namespace"
namespace OD = OCPI::DRC;

class Drc_fmcomms_2_3Worker : public OD::DrcProxyBase {
```

This base class knows about the requirements of the DRC spec, and simplifies the coding for handling the management of the state machine for DRC configurations. It implements all the property notification methods associated with the properties that control DRC configurations, and maintains the state of the configurations according to the defined lifecycle state machine. It defines 5 virtual methods, 3 of which the proxy *must* implement (**start_config**, **stop_config** and **release_config**) and 2 that the proxy *may* implement:

```
  virtual RCCResult prepare_config(unsigned config);
  virtual RCCResult start_config(unsigned config) = 0;
  virtual RCCResult stop_config(unsigned config) = 0;
  virtual RCCResult release_config(unsigned config) = 0;
  virtual RCCResult status_config(unsigned config);
```

All these methods take the configuration ordinal as an argument, return an **RCCResult** type, and can access the configurations using code like this:

```
RCCResult start_config(unsigned config) {
  auto &conf = m_properties.configurations.data[config];
  ...
```

and thus, according to how all C++ RCC workers are written, the parts of the configuration structure can be accessed using code like:

```
auto nChannels = conf.channels.length;
for (unsigned n = 0; n < nChannels; ++n) {
      auto &channel = conf.channels.data[n];
      ..... channel.tuning_freq_MHz...
```

The most important method is **`start_config`**, which takes the configuration information and sets up the hardware and slave workers to implement the requested configuration and make it operational. The **`prepare_config`** method may be used to do any preprocessing on the configuration information to minimize the overhead of subsequent starting and restarting a configuration repeatedly in a multi-configuration (mode switching) application. The **`prepare_config`** method must not change any hardware settings that could impair another configuration that was already running, but it could be called to do preprocessing and error checking on the "next" configuration, while a previous configuration was operational, to reduce switching time.

The **`stop_config`** is intended to allow currently used channel resources to now be used for a different configuration, while possibly preserving the validation and preparation work previously done. The **`release_config`** method indicates that there is no more need for the configuration and that any validation and preparation resources may be relinquished.

This model allows for there to be "preparation resources" used during preparation that do not affect running configurations, and then there are other additional (hardware, channel, etc.) resources that are necessary to make the configuration operational.

Finally, the **`status_config`** method is responsible for filling in status for channels in the requested configuration that are not simply copies of what is in the configuration. I.e., the actual hardware value that is within the requested tolerance, but not the exact value requested. Eg.:

```
RCCResult status_config(unsigned config) {
  auto &conf = m_properties.configurations.data[config];
  auto &status = m_properties.status.data[config];
  for (unsigned n = 0; n < conf.channels.size(); ++n) {
    auto &conf_channel = conf.channels.data[n];
    auto &status_channel = status.channels.data[n];
    status_channel.tuning_freq_MHz = ...
```

Note that the requested values of each channel are copied into the status for each channel prior to this method being called so only when the actual value varies from the requested value is this method necessary.

6 Glossary of Terms

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

6.1 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.

6.2 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***.