OpenCPI
Platform Development
Guide

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

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 Overview*** 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 development vs. execution.

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	Operating System, Libraries, Drivers	
FPGAs	Xilinx (Virtex, Zynq) 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 is required.

Table 1 - Table of Reference Documents

Title	Published By	Link
OpenCPI Overview	OpenCPI	https://github.com/opencpi/opencpi/raw/2018.Q3/doc/pdf/OpenCPI_Overview.pdf
OpenCPI Installation Guide	OpenCPI	https://github.com/opencpi/opencpi/raw/2018.Q3/doc/pdf/OpenCPI_Installation.pdf
OpenCPI Component Development Guide	OpenCPI	https://opencpi.github.io/releases/1.5.0.test/doc/OpenCPI_Component_Development.pdf
OpenCPI Application Development Guide	OpenCPI	https://github.com/opencpi/opencpi/raw/2018.Q3/doc/pdf/OpenCPI_Application_Development.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 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, we target the 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 *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 allow the source or sink of data flowing between components 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 devices accessible to the platform. Such cards may be plugged in to any platforms that have compatible slots.

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, an ARM-based ***platform*** attached to a variety of external peripherals (Ethernet, USB, etc.) as well as 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 optionally configured as being 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 Development and Execution

Every platform must be enabled for:

- development – to create executables for it on a some development system
- execution – to provide the runtime infrastructure to execute applications 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 for development involves enabling compilation/build on some development system that may be different than the platform being enabled. I.e. if an embedded platform is being enabled for development, that typically means installing a cross-compilation tool chain on some other development host that can produce binaries for the targeted embedded 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* is procuring, installing, configuring and integrating the various tools necessary to enable the developer to design and create binary artifacts from source code, for a given target platform. Some adaptations, scripts or wrappers are typically required to enable such tools to operate in the OpenCPI development context. OpenCPI does not contain, 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 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 "development host", that is termed "cross development", using "cross-tools" (e.g. cross-compilers).

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 both to create binaries for itself and other target platforms (cross development). These tools include both OpenCPI's own tools as well as target-specific tools from third parties.

Enabling systems may involve creating or modifying infrastructure code in OpenCPI, but "enabling for development" is defined in this document as enabling developers to *build* workers and applications for subsequent execution on a platform. I.e. the development of infrastructure code enables execution, but does not "enable development".

2.3 Enabling New Systems for OpenCPI

Enabling systems for OpenCPI implies enabling the platforms and interconnects in the system for OpenCPI, as well as 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 possible that support for some of them is already available in OpenCPI. So the enabling process is reduced to only dealing with the elements that do not already have 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 processors)

For each, if there is no existing support in OpenCPI, support modules must be developed. For some, there may be 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 layers 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) and runtime. Devices and interconnects are enabled for runtime only.

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.

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 refuse to supply the necessary information, leaving reverse engineering the only option.

Since OpenCPI, especially on FPGAs, operates directly on the FPGA devices and usually interacts directly with attached devices, either the vendor must supply the required device workers or supply the information required to development them. Preexisting FPGA drivers for attached devices that were not written with OpenCPI in mind are usually unsuitable as is, and must be either modified or replaced. In some cases then can be “wrapped”, although this may result in the addition of undesired overhead.

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 unavailable or unduly restricted.

2.4.2 Processor, interconnect and device assessment

Evaluate the state of support currently within OpenCPI and identify additional technical efforts are 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 a processor 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 Technical Data Package

For all elements requiring updated 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 developments. 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, which settings, cost)
- Connectivity technical details (e.g. how interconnects, devices and slots are connected, including pin-outs etc.)

Require information for devices include:

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

As part of this effort, any additional required feasibility experiments or reverse engineering tasks are identified. These are tasks to fill in the information gaps in order to have a 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 developments. Such experiments are derived from the process of the above tasks (i.e. discovering knowledge gaps), and may include:

- 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 efforts establish the final information to plan and budget for the effort of enabling a new system for OpenCPI. This activity depends on access to a real system. When the vendor is performing the work for there own hardware, this task 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 each functionality. The specifications are mostly based on achieving functionality that already exists on other systems, so the specifications 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 supposedly already supported, should still have a verification task planned.

In some cases, supporting a new system may in fact introduce new classes of support for OpenCPI, in which case the specifications will need to define functionality more fully. If such specifications are made available to the OpenCPI community, they can end up being a common template for such support on other systems. Otherwise the functionality would be described as the existing baseline for similar devices, with any core enhancements clearly specified.

The tasks defined here will be of types described more fully in other sections of this document, where the various types 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 GPP platform
- New interconnect for GPP platforms
- New interconnect for FPGA platforms
- New FPGA platform
- New FPGA device
- New FPGA cards

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.

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.

3 Enabling the Development Host

Before any platform-specific tools are installed on the development host, the basic development configuration for the host must be established for OpenCPI. This process is based on a source distribution of OpenCPI. A binary/packaged/RPM-based distribution of OpenCPI is not appropriate for enabling a new development host.

The ***OpenCPI Installation Guide*** describes the installation process for source distributions on *supported* development hosts, but for new hosts the basic steps are:

- Installation of the OS with suitable options and packages.
- Installation of native development tools to drive OpenCPI's build system on the development host.
- Retrieving current source code for OpenCPI
- Establish a build environment with appropriate options
- Building the core OpenCPI software, targeting the development host
- Building and testing the OpenCPI installation.

Building the framework software for the development host includes building OpenCPI tool executables, as well as runtime libraries that support both tool execution as well as application/component execution on the development host.

This section addresses issues specific to development hosts, while following sections address issues for any target software platform for execution, which also includes development hosts.

For a new development host, the operating system installation/configuration should be defined and documented, frequently using manual steps. Sometimes it is as simple as "make sure you have **`git`**, and enable some users for **`sudo`**".

3.1 Operating System Installation

While an existing development system with various software installed can certainly be used as a development host, it is valuable to make a clean installation from scratch to avoid configuration conflicts and problems that simply arise from a mis-match or mis-configuration of the environment. While most OS installations have a number of manual steps, they should still be written down so they can be reliably repeated on new systems. Scripts should be written when a number of steps can be automated, being careful to avoid dependencies on other software this early in the installation process.

See the installation guide for instructions on OS installation for the typical development hosts. Ideally, a new development host should have a new section written about it in the installation guide document so others can benefit, including:

- How to get the OS (DVD, download, etc.).
- How to perform the installation, with options at least appropriate for OpenCPI development.
- How to update the OS to the most recent patch level.How to obtain/download the OpenCPI code base.

This (normally manual) new procedure should be included in the README file in the platform's directory (see below for software platform directories in projects). The last step, retrieving the OpenCPI source distribution, could consist of network download or downloading to another system and burning CDs/DVDs, or even installing additional tools in order to accomplish the download. Typically, for network attached systems, it is simply accomplished by using:

```
% git clone https://github.com/opencpi/opencpi.git
```

Thus the configuration of the OS would need to include “install git”. Once the OpenCPI source distribution is present, and a tag is explicitly checked out, scripts to finish the installation can be run out of that tree. The remaining instructions assume you are in the directory created by the **`git clone`** command, i.e. you must then do:

```
% cd opencpi
```

In summary, the task to enable this first aspect of the development host installation is to make some basic installation and configuration choices for the development OS, and then follow the instructions below for enabling any software platform. The OS installation and configuration issues should be in the README file in the new platform's directory.

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.

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

4.1 Enabling Development for 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`**, **`centos6`**, **`centos7`**, **`ubuntu14`**, **`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:

- Prebuilt, globally-installed software packages that are a (usually optional) standard part of an operating system installation.
- 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 category.

So for any given platform, OpenCPI is enabled by 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 **`github.com`**, and 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 ***prerequisite*** software packages that are *specific to the platform*, but are not available as standard packaged software.[2]
3. Build the standard (always or frequently used) OpenCPI ***prerequisite*** software packages.

Build the OpenCPI framework libraries and executables

4. Build all the software assets in the built-in projects that are part of OpenCPI, in the same github repository (RCC workers and ACI applications).
5. Run a number of tests to verify OpenCPI operation (software only).

These 6 steps are all done in a master installation script called:

`./scripts/install-opencpi.sh`

These steps, and how to enable them for a software platform, will be described in detail in the following sections, but a summary of the files required in a platform's directory to enable these steps for each software platform is here:

A software platform definition file, named ***`<platform>`*`.mk`**, which sets variables that describe the platform. Between 3 and 20 variables might be required.

6. *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`**.
7. If needed, a script to install various standard packaged software from software package repositories to support OpenCPI for this platform, called ***`<platform>`*`-packages.sh`**.
8. 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`**.

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.

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-identification of 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) 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 Installation of Pre-packaged Software from a Package Repository

The tools required to support OpenCPI development on a development host are usually obtained from standard packed software in 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 installation is not appropriate when enabling new software platforms.

This script is run first (after self-identification), to install the software that enables building other software: prerequisites, the OpenCPI framework, and the built-in projects in the OpenCPI github repo. The software package repository used may in fact be local on the system or on the local network rather than being internet-based.

These 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 packages 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 Installation of Prerequisite Software Packages in the OpenCPI Sand Box.

After the installation of pre-built standard packaged software from repositories is done (if there are any), the next step is to build and install prerequisite software packages in a directory soley 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 software package that is independently downloaded, possibly built-from-source, and installed in the OpenCPI sand box. 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-<package>.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 prerequisite packages are also installed as part of that RPM installation.

4.1.1.4 Building the OpenCPI Framework Libraries and Executables

After the standard packaged software is installed, and the platform-specific and OpenCPI standard prerequisite packages are built and installed, the OpenCPI framework can built. This is done either as part of the **./scripts/install-opencpi.sh** script or simply by issuing the **make** command in the top level of the OpenCPI source tree.

Building the framework depends on a tool chain declared in the platform definition file, as well as any platform-specific (if any) and OpenCPI standard 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 Building 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 in its github repository (currently **core**, **assets**, and **inactive**). This occurs as the last part of the **./scripts/install-opencpi.sh** script or issuing the **make projects** command in the top level of the OpenCPI source tree.

This builds all software workers and ACI applications in these projects. It is equivalent to using the **ocpidev build** command in each project.

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

4.1.1.6 Testing the OpenCPI Installation for a Software Platform

The last step in **./scripts/install-opencpi.sh** is running the final tests of a 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, this 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. For these platforms, **./scripts/install-opencpi.sh** 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 use **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/[cdk/]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 a variables assigned are not in the list of valid variables. All the default values are as needed for CentOS6 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 the **platform-defaults.mk** are basically for CentOS6, there are not many variables:

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

For the cross-compiled Xilinx Zynq 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 $(OCPI_CDK_DIR)/include/hdl/xilinx.mk
tooldir:=$(OcpiXilinxEdkDir)/gnu/arm/lin/bin
OcpiCrossCompile=$(tooldir)/arm-xilinx-linux-gnueabi-
OcpiCFlags+=-mfpu=neon-fp16 -mfloat-abi=softfp -march=armv7-a
cpiCxxFlags+=-mfpu=neon-fp16 -mfloat-abi=softfp -march=armv7-a
OcpiStaticProgramFlags=-rdynamic
OcpiKernelDir=release/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.

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 so several are listed here verbatim as examples:

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

```
CentOS release 6.9 (Final)
```

the entire script file can be:

```
f=/etc/centos-release
[ -r $f ] && read c r v x < $f &&
  [[ "$c"  == CentOS && "$v" == 6.* ]]
```

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 runtime packages/RPMs
2. The packages necessary for 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.

It 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`***, is meant to provide a platform-specific extension to the **`Project.exports`** file in the root of the OpenCPI source tree. Its syntax is the same (with **+** for development files and **=** for runtime files), with an additional prefix, **`@`**, for exports that are intended for the deployment package (e.g. bootable media, SD card). When a file to be exported is in the platform's own directory, the literal string **`<platform_dir>`** is replaced with the actual platform's directory pathname. As with the top-level **`Project.exports`** file, the literal string **`<target>`** is replaced with the platform name itself. One example line in this file:

```
=platforms/zynq/zynq_system.xml <target>/system.xml
```

means that the default **`zynq_system.xml`** file shared by all Zynq platforms should be placed in this platform's exported directory in runtime packages, as **`system.xml`**.

This example:

```
=<platform_dir>/README* <target>/
```

means that all **README** files in the platform's directory should be exported in the platform's exported directory in runtime packages.

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 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 overriden 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 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/prerequisties/** 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 executabled (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 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:

- the current working directory is a 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 shoulld be installed
- the **relative_link** function is defined to create appropriate relative symbolic links from this build directory and the installation 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.

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 directory: **$OcpiInstallDir** and **$OcpiInstallExecDir**.

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-install.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
```

[Here we can open up the platform-definition variables universe for more advanced cases.]

4.1.3 Summary for Enabling Development for a New GPP Platform

- Create the platform's directory in a project.
- 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.
- 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 **`./install-opencpi.sh`** ***`<platform>`*** for cross-developed hosts.
- 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, loading the kernel driver (using the **`ocpidriver`** command), and setting the PATH environment variable correctly, and setting the OCPI_CDK_DIR environment variable is usually sufficient. These are normal installation steps.

The kernel driver is in fact only necessary when accessing other platforms via the system bus. If the GPP platform has no such bus/fabric, 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 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”.

Currently OpenCPI only supports deployment packages on systems with FPGAs, and the deployment package is named according to the HDL platform in that system, even though it is mostly defined by the software platform that is running the system.

Deployment packages are created by asking the HDL platform to use the runtime package for its associated software platform and add the necessary files for the “system” that combines that software platform with the HDL platform.

The files for a deployment package are of two types:

- Files that are useful to install on an associated development host or file server
- Files that are installed on the (usually bootable) media installed in the embedded system.

A deployment package usually has a directory containing the second type of file that can be directly copied to a (e.g. SD card) medium that will enable the embedded system to boot and run OpenCPI applications.

The files on the bootable image also 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.

A deployment package, based on an HDL platform and an associated software platform, is defined by an exports file in the HDL platform's directory. It has the same syntax as the other exports files, with a new leading character @, indicating that the export is only for the deployment package.

Thus for HDL platform exports, the **+** lines indicate platform-specific exports for the development package, **=** indicates platform-specific exports for the runtime package, and **@** indicates platform-specific exports for a deployment package.

In the top-level OpenCPI Makefile, there is a “deploy” goal which will populate the exports tree with the deployment package for the platforms specified in the Platforms variable. The platforms in this list are either HDL platforms, or of the form ***<hdl-platform>:<sw-platform>***. In the former case the associated software platform is inferred from what has been declared by the HDL platform's ***<platform>*.mk** file in its directory. An example of everything needed to make a deployment package for zedboard with xilxinx13_3 as it’s software platform is listed below. First, in the normal course of building OpenCPI for the targeted platform, the software platform would be already be built by doing:

```
$ ./scripts/install-opencpi.sh xilinx13_3
```

and then the project containing the HDL platform would be built with:

```
$ ocpidev build hdl platform zed
```

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

```
$ make deploy Platform=zed:xilinx13_3
```

When **make deploy** is run, the bootable media directory tree is placed in this directory:

```
exports/<hdl-platform>/<hdl-platform>-deploy/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 that OpenCPI uses to interface with the hardware surrounding the processor, FPGA platforms do not. OpenCPI provides an infrastructure on FPGAs, which requires FPGA logic that is specific to the platform, analogous to a “board support package” that adapts an embedded operating system to a given hardware “board” and associated devices.

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 the same infrastructure as physical FPGA platforms.

When an SoC, like the Xilinx Zynq 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 for 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 applies to the new platform).
- Verifying that the integrated tool chain can process and build all the core OpenCPI FPGA code and portable components when targeting the FPGA platform's part and part family.

Enabling runtime execution for 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, Ethernet, etc. used to talk to other OpenCPI platforms
- Slots: FMC, HSMC, mezzanine card slots.

For example, a Xilinx ML605 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 6: Example FPGA/HDL Platforms

Board	Part	Interconnect(s)	Description
`ML605`	Virtex6	PCIe	Xilinx PCIe-based Virtex6 development board, with 2 FMC slots
`ZedBoard`	Zynq/PL	AXI internal	Digilent Zynq 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/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 allowing only a certain number of simulator instances to run at the same time.

At the time of this writing, only mixed-language simulators (VHDL and Verilog) may be enabled and used with OpenCPI.

Examples of supported FPGA simulators include:

- Xilinx Isim from ISE 14.7
- Mentor Modelsim DE 10.2
- Xilinx Vivado Xsim 2016.4

Other simulators that may be supported include:

- Aldec

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.

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 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 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

[This is a large topic that is not fully documented].

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:

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

The database of part families, and their relationships to tools and vendors, is in the file:

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

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

`tools/cdk/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 portable HDL code can be built using the "make hdlportable" command in the top level directory of OpenCPI, supplying the targets (part families) is required.

Here is a command 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 virtex6 virtex5 \
              stratix4 stratix5 zynq spartan3adsp'
```

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 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), there are two scripts that must be also 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)

In some cases the appropriate script might be 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 **hdl/vendors/*<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 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 requires 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 separate from a platform or card so it is naturally 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 support 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 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 (presynthesized) assembly of device-level HDL workers that represent a particular configuration of device support modules for a given HDL platform. The HDL code is *automatically generated* from a brief description in XML

All these asset types are initially created using the **`ocpidev`** 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 configuration

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 **Makefile,** 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 slots using supported cards.

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 enumeraterd 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, **input** 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

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 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. This if the signal is defined as:

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

The signals are **data(0)**, **data(1)**, **data(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 in to 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).

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>*.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 definition file contains 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 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 standardized, 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 signals that are implied for such signals when *inside* an FPGA. The 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='data' width='4'/> <!-- Pin K5 -->
</SlotType>
```

The name of a slot type is used in the XML description files of platforms (that have slots) and cards.

5.4.3 Creating the XML Metadata Definition for the Platform

A platform is created using the **`ocpidev create platform`** command, which establishes the platform's directory under **`hdl/platforms`** and creates an initial **`Makefile`**, the XML file ***`<platform>`*`.xml`** describing the platform, and an initial VHDL source file ***`<platform>`*`.vhd`** for the platform worker.

The XML file that defines an HDL platform is named the same as the platform name with an "**`.xml`**" suffix. It describes both the hardware aspects of the platform as well as the OWD for the HDL worker that controls the platform, which is called the "platform worker". Thus the platform XML is an OWD for the platform worker, with additional information. The platform worker is a special type of "device worker", with extra requirements for the platform. Developing device workers in general is the subject of the next major section below, but information specific to platform workers is described here.

The platform XML is an OWD with these special aspects, 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.
- The external signals (to pins) required by the platform worker (as for any device worker) are indicated by **`signal`** elements.

- Special platform port types for platform workers: **metadata**, **timebase**, **cpmaster**, **sdp**, and **unoc**.

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

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

Other requirements are described below.

5.4.3.1 Properties of Platform Workers

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

platform — required string attribute must be set to the name of the platform

sdp_width — **uchar** attribute, must be set in platforms using the SDP (see Interconnect Support below)

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 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 to specific platform ports in the platform worker code as described next.

5.4.3.2 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 processors'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 implies that the external pathways (interconnects) for control and data are separate.

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.

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).

5.4.3.3 Device Elements in a Platform XML File

Device elements here 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 instanced) in platform configurations and containers. These declarations here, by themselves, *do not* cause the device workers to be instanced in any bitstreams. Devices are used and instanced by being referenced in platform configurations, containers and assemblies.

Device elements indicate:

- The HDL device worker supporting 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 appended.
- Which parameter settings should be used for the device worker for this device *(optional* **property** *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)*

In most cases a device element simply indicates that a device exists and which device worker should be used to support it. E.g.: if a platform had a flash device that was supported 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 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=” attributes), much like the property values for application components in an OpenCPI application.

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 **timebase** port must be specified as the **frequency** parameter to this device worker.

Devices 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, 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.

5.4.3.4 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.

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

If a platform worker wants 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.

5.4.3.5 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" 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 name becomes the slot type. 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.

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**). Here 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**.

5.4.3.6 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>
```

5.4.4 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 OWD, any such ports and properties must be defined. The spec (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.

5.4.4.1 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 2: 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	Readable	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/ Readable	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/ Readable	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 number **slot** elements in OWD
slotCardIsPresent	Bool array	Volatile	Indicate whether card is plugged into slot
slotNames	String	Parameter	Comma-separated list of slot names Set in OWD <specproperty>

5.4.4.2 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.reset <= reset;
timebase_out.ppsIn <= '0';
my_pps_out_pin     <= timebase_in.pps
```

These signals are the basis for timekeeping on the platform. They are a clock and associated reset, 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.

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 a address window 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/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 of 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 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 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 Protoco section.

5.4.4.3 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” 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), unless there will never be any need for timekeeping on the platform.

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 purposes.

5.4.5 The Makefile for the Platform Worker

Like any other worker, a platform worker has a worker **`Makefile`** that is used for various purposes, normally created by the **`ocpidev`** command. Since the **`HdlTarget`**(**`s`**) and **`HdlPlatform`**(**`s`**) are implicit for a platform worker (its name), there is no need to specify either (in the file or on the command line).

As with other HDL workers, the **`Makefile`** may reference other local source files or primitive cores and libraries from elsewhere. The basic contents for a platform worker's Makefile, as created by **`ocpidev`**, is simply:

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

The platform worker is built in its directory (in one or more target subdirectories). Platform configurations are also built in the platform worker directory. 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`** make variable in this **`Makefile`**, and for each one, write an XML file describing which devices (and any parameters for them) that should be included in the platform configuration. See the next section for a complete description of platform configurations.

Running **`make`** in a platform worker directory builds the platform worker, the **`base`** configuration, and any other platform configurations as specified in the **`Configurations`** make variable.

Each platform worker must have a make file fragment called ***`<platform>`*`.mk`** which defines **`make`** variables required for users of the platform. There is one variable required to be set in this file: **`HdlPart_`*`<platform>`***. This variable must be set to the part name of this platform. Also, if there is an association between this HDL platform and an associated software platform (typically for FPGA SoCs with embedded processors), the HdlRccPlatform_<platform> variable can be set to name the RCC software platform associated with this HDL platform. An example file is for the Xilinx Zynq part on the ZedBoard platform:

```
HdlPart_zed=xc7z020-1-clg484
HdlRccPlatform_zed=xilinx13_4
```

This indicates the actual part for the platform, and also indicates that the associated software platform is the 2013.4 release of Xilinx Linux.

The **`ExportFiles`** make variable is specific to platform workers and specifies which local files in this directory must be made available to users of the platform. Files in this list include the jtag and flash support scripts for the platform, as well as constraints files required to build container bitstreams for the platform.

5.4.6 Specifying Platform Configurations in XML Files.

For each platform configuration mentioned in the Configurations variable in the platform worker **`Makefile`**, 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 OWD), 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 should be plugged into a slot on the platform, and thus device workers to support the **`lime_adc`** device on that card should 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 in the board definition* (either platform worker XML file or 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). 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>
```

5.4.7 Control Plane Master Interface

This interface is undocumented at this time.

5.4.8 Scalable Data Plane (SDP) Interface

A key interface in the OpenCPI data-plane infrastructure in the SDP. When FPGA plaforms 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. Prime 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). Each interconnect is adapted to the SDP so that all the rest of the on-chip data-plane infrastructure remains common across all HDL platforms supported by OpenCPI.

The SDP port 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 SDP.This protocol is undocumented at this time.

Before describing the details of the interface, it is important to understand where SDP exists in the OpenCPI data plane stack. The data plane is organized in two layers, the transport and the data transfer layers. Furthermore, OpenCPI defines 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. REF OpenCPI RDMA Protocol Specification?

The transport layer is fabric agnostic and provides buffer and protocol management. The data transfer layer is a RDMA driver level module that allows new drivers to be plugged into OpenCPI enabling data exchange over a variety of fabrics. OpenCPI provides several intrinsic drivers with the standard distribution including Host Memory, network sockets and PCI bus.

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

- Minimize unnecessary differences between SDP and common FPGA interconnects (AXI, PCIe)
- Minimize complexity of FPGA infrastructure which must interface to SDP

The SDP interface has the following attributes:

- Bidirectional: either side of a connection can be master
- Multi-master
- Packet-based: header and data (optional)
- 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 interface closer to the off-chip interconnect is called the SDP interface master. At the VHDL port level, there are two structures (one per direction) with VHDL types m2s_t, and s2m_t. Data is also bidirectional and of type dword_array_t with parameterized width with generic sdp_width to define the width.

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** structures is a common messaging structure of VHDL type **sdp_t**. The figure below illustrates the port structure using VHDL types, and the tables following tables describe the signals.

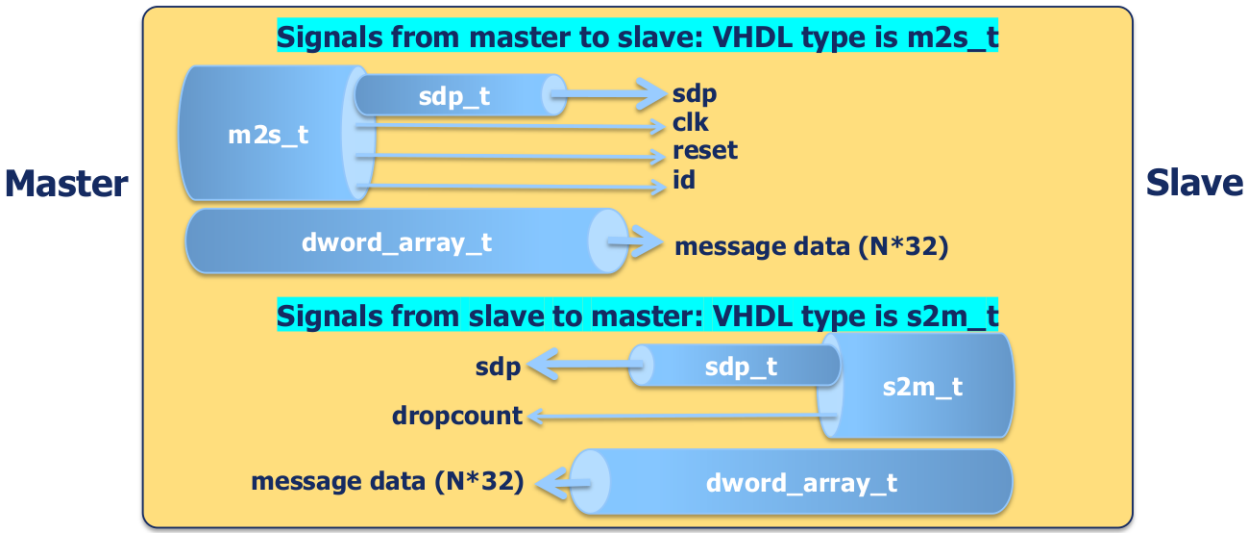

Figure 1: SDP VHDL Signal Records

*Table 3: 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 4: 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 signaling for SDP. The figure below illustrates the record structure and following tables describe the record signals.

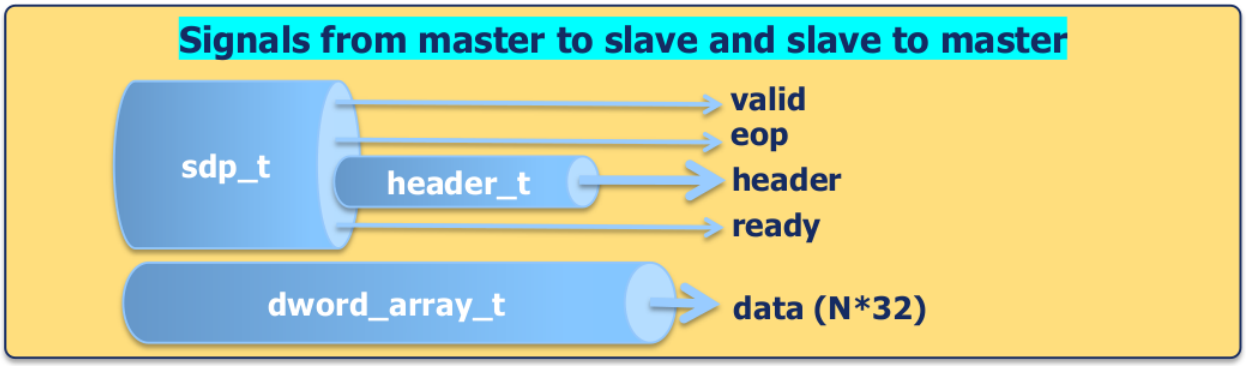

*Figure 2: **sdp_t** Record Structure*

Table 5: 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 6: 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/write/response.
xid	xid_t	Transaction ID for matching 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 response (if any)
addr	**addr_t**	LSBs of dword address. For requests to slave, address within specific 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

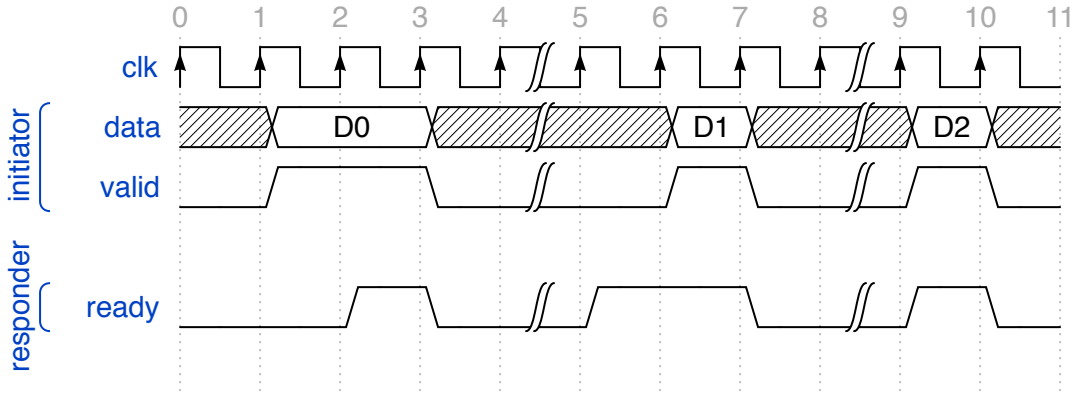

Figure 3: 3 SDP Transfers with Different Ready and Valid Assertions

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.

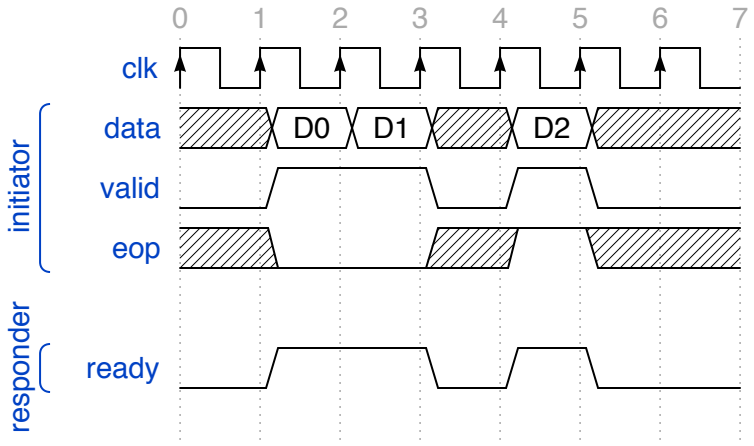

Figure 4: SDP Packet Transfer Showing EOP

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.

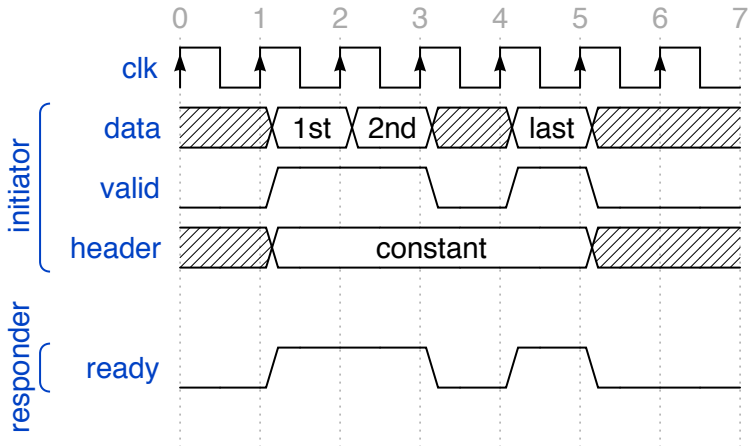

Figure 5: SDP Packet Transfer Showing Header

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.

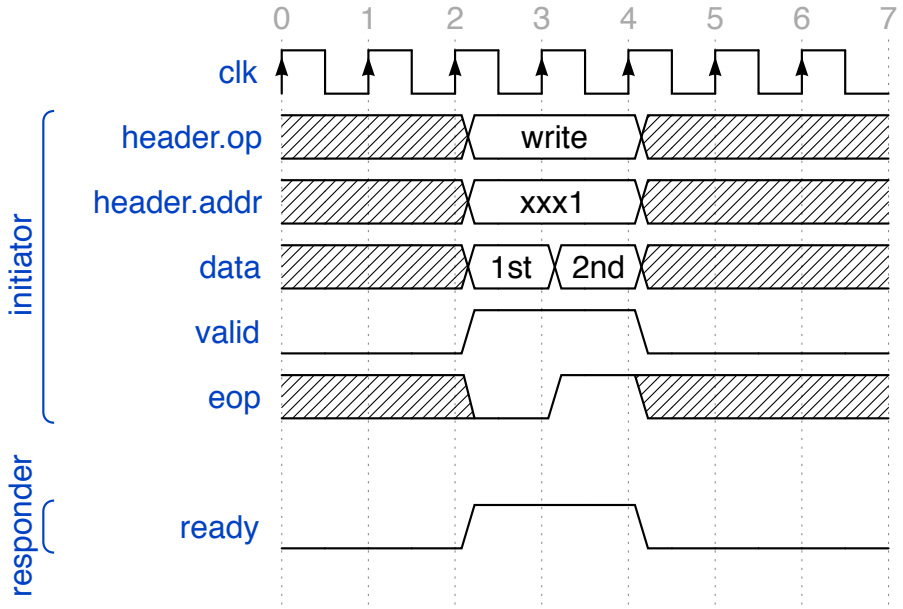

Figure 6: Basic Write Request with Address and Data

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.

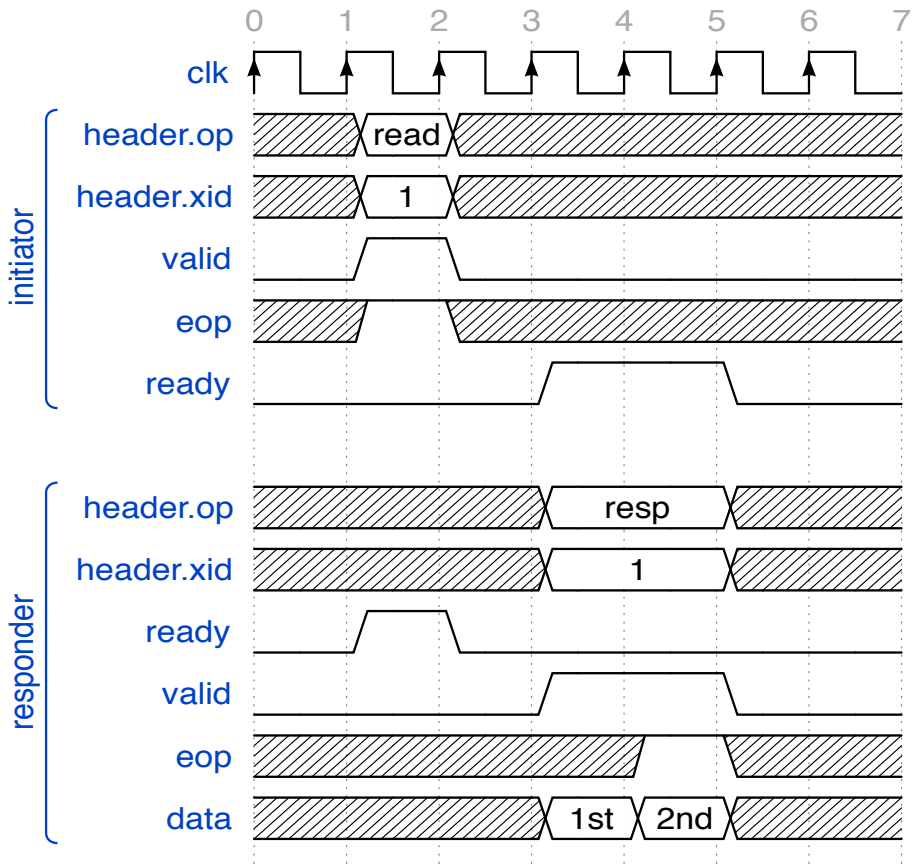

Figure 7: Read Request and Corresponding Response

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`**.

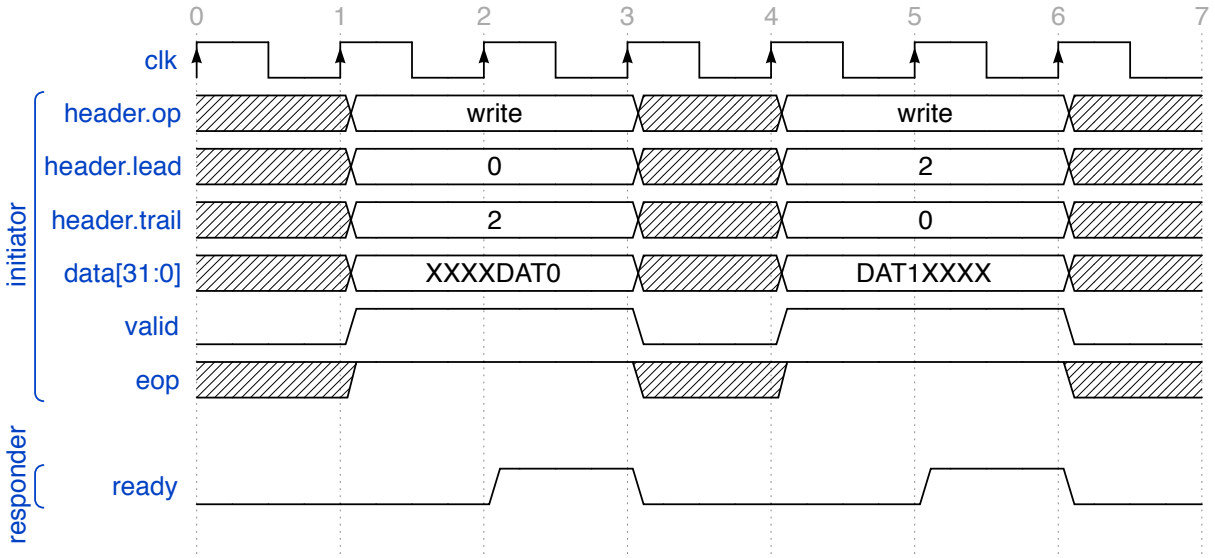

Figure 8: Two 16-bit Writes on a 32-bit Interface with Leading/Trailing Invalid Bytes

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.

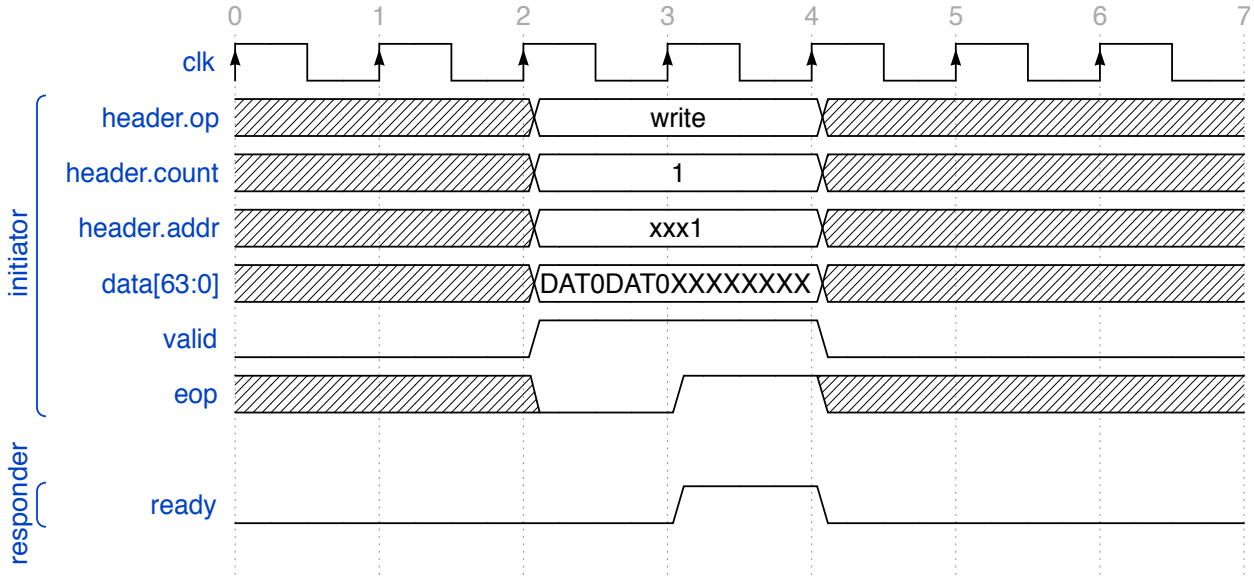

Figure 9: 32-bit Write on 64-bit Interface Showing Little-endian Alignment

5.4.9 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.

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 development 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 printer) would have some signals attached to FPGA pins that are connected to the output device, 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 (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 supported 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;
data        <= in_in.data;
valid       <= in_in.valid
in_out.take <= ready;
```

5.5.1 A Device Worker Implements the Data Sheet

It is recommended to use the data sheet's signal names and register names in the device worker code (and its OWD) so they are easily correlated to the names in the data sheet. This allows for easier code maintenance and system level debugging. E.g., a system engineer may find it especially 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 these reasons.

- *Implementing all the functionality in a device may not be desirable when it is initially written for a given 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 programming since the **`ocpirun`** and **`ocpihdl`** commands can easily access this information. It 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.

As the integration of functionality on single chips increases, it is common for a "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 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 in a later section.

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 a device worker 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.

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.

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

- *Device-worker-only*, where the device worker implements both the device 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 PPTs 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.

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 whenever any of its *supported* device workers are instantiated in a platform configuration or container. Subdevice workers may simply support the different device workers used on a single multi-function device, or they may support a variety of different device workers that are found on a platform. Thus they are usually very *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.

There are cases where a subdevice is required to support a single 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 to do platform/card-specific dirty work.

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 also 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
- How it is connected to each of the device workers it supports.

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

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 below). It is declaring that if that 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 OWD 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' properties.

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. The **reset** signal tells the subdevice that this device worker is being reset, and the **raw** subrecord is conveying 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 > 1) when the subdevice worker is supporting multiple device workers.

5.5.4.2 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 both ports, 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.

5.5.5 Testing Device Workers with Emulators

A device worker may specify that it is actually a device *emulator* that emulates a device for test purposes. Thus while a normal device worker supports 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 supports 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.

5.5.5.1 Emulator Signals, Ports and Properties

Emulators establish this relationship with a 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 **devsignals** 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.

Finally, the emulator also inherits all the parameter and writable 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 parameters with multiple values. 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.

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 emulator='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

The OpenCPI unit test framework described in the ***OpenCPI Component Development*** document also applies to testing device workers. 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.

n the container as “floating” devices (devices which have not been declared as existing on the platform).

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 then 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 Endpoint Proxies Suitable for Applications

[Preliminary feature with no specific support]

As discussed above, we use ***device proxies*** to normalize 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, in addition to device proxies that are defined for the granularity of individual devices, that have device workers, OpenCPI also defines specifications for some higher level proxies called ***Endpoint Proxies,*** for presenting a collection of devices to applications as something to connect and configure within the application. The purpose of higher level Endpoint 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). Endpoint proxies make that possible.

Endpoint proxies are simply proxies that typically have multiple slaves, which themselves are probably device proxies, or in some cases device workers.

As with subdevices at the bottom of the OpenCPI “device support” stack, endpoint proxies are at the top of the “device support” stack, appropriate for use by applications. But both may internally be somewhat 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 endpoint proxies by instantiating a component of an endpoint proxy spec. A good example of a class of endpoint proxies is a “radio front end” (as defined by the RedHawk system or GNU Radio), or a “transceiver subsystem” as defined in the Wireless Innovation Forum, or the “RF Chain” as defined in the JTRS MHAL specification. Since OpenCPI is not a software radio framework as such, an endpoint proxy can represent any application-level subsystem or source or sink of data.

5.5.7 XML Metadata for Device Workers/Subdevices/DeviceProxies/EndpointProxies

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

- Device Workers
- Device Proxies
- Subdevices

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 they support and how they are 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'
```

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 extends the raw property signals from a device worker to a subdevice worker. It may be an array port. 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 > 1 that this is an array of raw property ports (default is **1**). 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**).

*The port is an array port if **count** is specified greater than 1 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 > 1)

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

The file indicated by the **signals** attribute enumerates the signals in the bundle, similar to the **signal** elements in a device worker's OWD.

5.5.7.3 The Supports XML Element for Subdevices.

Subdevices indicate which device workers they support by using **supports** XML elements. To indicate how they are connected to a device worker they support, they specify **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 are:

worker — the name of the device worker it supports

index — identifies which device of that type (on the platform or card) is being supported by the subdevice via this **supports** element

The attributes of the "connect" child element of the "supports" element are:

port — the port on this subdevice that should be connected

to — the port on the supported device worker that should be connected

index — index into the subdevice's port array (when its count attribute is > 1).

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 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 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 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 devices 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*>.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 platform XML files except for one thing. In both platform XML and card XML files the **signal** child elements of **device** elements indicate a 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 are derived from the slot type's signals. Thus each device instance is 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>
```

6 Glossary

Application – In this context of Component-Based Development (CBD), an application is a composition or assembly of components that as a whole perform some useful function. The term “application” can also be an adjective to distinguish functions or code from “infrastructure” to support the execution of component-based application. I.e. software/gateware is either “application” or “infrastructure”.

Configuration Properties – Named scalar values of a worker that may be read or written by control software. Their values indicate or control aspects of the worker’s operation. Reading and writing these property values may or may not have side effects on the operation of the worker. Configuration properties with side effects can be used for custom worker control. Each worker may have its own, possibly unique, set of configuration properties. They may include hardware resource such registers, memory, and state.

Control Operations – A fixed set of control operations that every worker has. The control aspect is a common control model that allows all workers to be managed without having to customize the management infrastructure software for each worker, while the aforementioned configuration properties are used to specialize components.

Infrastructure – Software/gateware is either application of or infrastructure.

Worker – A concrete implementation (and possibly runtime instance) of a component, written according to an authoring model.

Authoring Model – A set of metadata and language rules and interfaces for writing a

[Incomplete]