# OpenCPI
# Platform Development
# Guide

***Preliminary***

*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 |

## Table of Contents

# 1 Introduction

This document describes how to enable OpenCPI components and applications to run on new platforms. 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 component development and application 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 execution vs. development.

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),<br>ARM (Xilinx/Altera/TI) | Compiler tool-chains | Operating System,<br>Libraries,<br>Drivers | |
| **FPGAs** | Xilinx (virtex, zynq)<br>Altera (stratix, cyclone)<br>Mentor/Modelsim | Synthesis<br>Place&route<br>Simulation | Bitstream loading<br>Control Plane Drivers<br>Data Plane Drivers | Directly attached I/O devices |
| **Graphics Processors** | Nvidia Tesla<br>AMD FirePro | Compilers, Profilers | Execution Management<br>Drivers<br>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 Introduction | OpenCPI | |
| OpenCPI Installation Guide | OpenCPI | |
| OpenCPI Component Development Guide | OpenCPI | |
| OpenCPI Application Development Guide | OpenCPI | |

# 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. While, at a high level, the system has processors of various types that are "places for components to run", each processor is itself part of a hardware subsystem. 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. We call the data paths that allow platforms to communicate with each other ***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 component to enter or exit the system. Thus devices are distinct from interconnects.

A ***system*** consists of ***platforms*** connected by ***interconnects***, and platforms can have locally attached ***devices***.

### *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 add devices to the platform. Such cards may be plugged in to any platforms that have compatible slots.

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 on the platform.

- Systems have platforms and interconnects.
- Platforms have slots and devices.
- Cards plug into slots and have devices.

### *2.2 Development and Execution*

Every platform must be enabled for development as well as execution. *Development* is the process of producing “executable binaries”, and *execution* is the process of running those binaries on the available platforms in a system, as part of the execution of a component-based application. 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.

### *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 interconnects, processors, and common devices are frequently found in many different systems, it is likely 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 in a particular situation, either because they are not going to be used, or their use is outside the scope of what OpenCPI does.

The system inventory for OpenCPI is:

- Processors
- Interconnects
- Devices

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 require development support as well as runtime support. Devices and interconnects require only runtime support.

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 usually results in unnecessary extra overhead.

## *2.4 A 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.

### *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.

Collect sufficient information to 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 “data sheet” exercise.

The result is a list of system elements that need to be assessed, specifically identified as to the devices involved. This activity should not take more than a week unless information about the target system is unavailable or unduly restricted.

### *2.4.2 Processor, interconnect and device assessment*

Evaluate the state of support already in OpenCPI and what 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 whether it is already supported in OpenCPI, needs some incremental/enhanced support, or needs entirely new support. The categories are:*

- Already 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 whether new tools are required that have not been integrated with OpenCPI before. Categories are:*

- Tools are already supported by OpenCPI
- Tools require a version upgrade/downgrade/variant of what is already supported.
- Tools are completely new and not yet supported by OpenCPI.

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

- Interconnect is already supported adequately in OpenCPI
- Interconnect support requires enhancements (e.g. PCI-E gen3x8, vs. PCI-E gen2x4).
- Interconnect support missing for processor type (e.g. Ethernet L2 on FPGAs).

*2.4.2.4 For each device, evaluate the state of support already in OpenCPI and what additional technical efforts are likely to be required the each device, as it will be attached to its processor..*

Determine whether it is already supported in OpenCPI, needs some incremental/enhanced support, or entirely new support. The categories are:

For each device attached to a processor, or on a required card, determine whether existing OpenCPI support is adequate. Categories are:

- Device is already supported
- Device support requires updating or enhancement.
- Device support not present, but similar devices are supported as a model.
- Device is 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. 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 do 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 of course makes them less desirable for OpenCPI-based systems.

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 always 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 tasks 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.

### *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, which will hopefully end up being a 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 cards
- New FPGA device

### *2.4.7 Verification*

Verification requirements are 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), 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 must be established for OpenCPI. The ***OpenCPI Installation Guide*** describes the installation process for supported development hosts, but 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 component libraries and example applications

Building the core software for the development host includes 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 platform for execution, which also includes development hosts.

For a new development platform, the following aspects must be defined, documented, and in some cases, new scripts and software will be required:

- Operating system installation/configuration
- Retrieval of OpenCPI source distribution.
- Installation of required packages that are available as options with the OS distribution.
- Installation of other required packages that require specific download/installations.

The first two steps are generally manual, requiring written instructions. The latter two are scripted. Both the instructions and scripts should be submitted back to the OpenCPI source distribution for the benefit of other users of the same development host installation.

### *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. Note that the last step is the first point where any OpenCPI support scripts become available. 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
```

Once the OpenCPI source distribution is present, scripts to finish the installation can be run out of that tree (after they are created and added to the OpenCPI repository). 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 installation is to create the README file explaining the initial OS installation, and the retrieval of the OpenCPI distribution. Create the new directory under **`platforms`**, named after the development platform, and put the README file in that directory.

### *3.2 Development Tools Installation for the Development Host.*

Most of the remaining steps for enabling the development host are the same as those for supporting any runtime platform, and are explained in the ***Enable GPP Platforms*** section below. Prior to the installation of any target-platform-specific tools, a basic set of tools must be installed on the development host that will support development for all platforms.

OpenCPI development requires a number of tools to be available. These required tools fall into two categories:

- Tools typically part of a standard OS and development environment installation.
- Tools that OpenCPI has specific scripts for downloading and installing.

The first category should be installed by a new script written for the development host platform, and placed in the platform's subdirectory under the name ***<platform>-packages.sh***. This script does what is necessary to perform a standard installation (presumably globally on the system for all users) of at least the following software tools:

- make
- git
- C/C++ compiler tool chain
- python

This script should be written assuming a minimal OS installation has been performed. I.e. it should not assume the OS is already a typical development system installation.

For OpenCPI binary distributions using the RPM or similar package management schemes, this script would be unnecessary since the installation would automatically deal with such dependencies.

For reference, the contents of this script for CentOS6, with the name:

```
platforms/centos6/centos6-packages.sh
```

is currently:

```
#!/bin/sh
# Install prerequisite packages for CentOS6
echo Installing standard extra packages using "yum"
sudo yum -y groupinstall "development tools"
echo Installing packages required: tcl pax python-devel
sudo yum -y install tcl pax python-devel fakeroot redhat-lsb-core
echo Installing 32 bit libraries '(really only required for
modelsim)'
sudo yum -y install glibc.i686 libXft.i686 libXext.i686 ncurses-
libs.i686
```

Sometimes extra tools here are required for various tools for other target platforms. In this example above, some 32 bit libraries are included because they were discovered to be needed by the Modelsim FPGA simulation tool. It was convenient, for users, to put them here for all installations rather than install them as needed for some tools. The

**tcl**, **fakeroot**, and **redhat-lsb-core** packages are not needed by OpenCPI itself, but are required to install some other packages used with OpenCPI.

This script is run by the higher level script **install-prerequisites.sh**.

After creating this script, the remaining steps for enabling the development host are the same as for any GPP platform.

# 4 Enabling GPP Platforms

Enabling a GPP (General Purpose Processor) platform requires two different aspects: enabling development/cross development, and enabling execution. Enabling for development allows building component and application binaries for the platform and also enables building the OpenCPI core runtime libraries and runtime command line tools for execution on the target platform.

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

- New compilers sometimes have new correct warnings that should be addressed.
- System headers are sometimes more standards conformant, which is good.
- Some compilers are “dumber” that others, requiring code 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 core OpenCPI 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 retested on existing supported platforms.

### *4.1 Enabling Development for New GPP Platforms*

Whether a GPP platform is the development host itself, or an embedded GPP using cross-development, the tool chain must be established on a development host. For development hosts, there is typically a default tool chain that is installed globally in the system for any development task on that system. This is the purpose of the **`platforms/<platform>-packages.sh`** script mentioned above.

When the tool chain desired for OpenCPI execution on the development host is *not* the default tool-chain for the development host, then it is installed separately much like other cross-compiler tool chains for embedded hosts.

Thus development tools for GPP platforms are established in three ways:

1. The development host's globally installed default tools.
2. A specific prerequisite installation in OpenCPI's own installation area
3. A side effect of another tool installation that it is part of.

The first case is the default for development hosts. The second is used either:

- on a development host when OpenCPI should not use the default global tool-chain installation, or
- for normal cross-tool installation

The third is used when such an installation by itself is not desired since the tool-chain is part of a separate package installation. An example of this third case is the Xilinx FPGA tool set which also includes the compiler tool chain for the ARM CPUs that are inside the Xilinx Zynq SoC (FPGA + dual ARM core).

OpenCPI has a “prerequisites area” (normally at /**`opt/opencpi/prerequisites`**), where various installations are placed so that they do not interfere with other global software installations. Both runtime library prerequisites and tool prerequisites are typically installed here. It is also acceptable to install tools in a global location, but such an installation may not be acceptable for all users. Hence all the built-in prerequisite installations that are standard for OpenCPI are installed in this sandbox area.

The primary tasks for enabling GPP cross development are to:

- Develop a script for installing the tool-chain package in the prerequisites area (unless the platform is the development host, and the standard tool-chain is used).
- Develop an environment setup script for OpenCPI cross development of the OpenCPI core software
- Develop a cross-development script for component development in the tools/include/rcc directory.

#### *4.1.1 Developing a Tool-chain Installation Script*

If the tool chain is truly specific to the platform, this script should live in the **`platforms`** directory in the file:

**platforms/<platform>/install-<tool>.sh**

Otherwise it should be in:

**scripts/install-<tool>.sh**

This script should follow the pattern of other install scripts in the scripts directory, in particular:

- Placing the package in the prerequisites area.
- Downloading from a known internet location if at all possible.
- If not possible, then telling the user to get the installation file somehow and where to put it (see the **scripts/install-opensplice.sh** for an example of this).
- Making the script clean up a previous installation when performing a new one.
- This script should source the installation setup script called **scripts/setup-install.sh**, which should be *sourced* early in the script.
- This script should be expected to be called from the root of the OpenCPI tree.

A simple example of these scripts is in:

**scripts/install-gtest.sh**

This script can then be called manually or become part of a platform installation script.

### *4.1.2 The File to Define the Full Target Specification for the Platform.*

A target specification file should be placed in the platform's directory with the name **target**. This file is a single line with the target specification consisting of three fields separated by hyphens. It is used during all compilations in OpenCPI to identify the platform. The CentOS6 target (in the file: **platforms/centos6/target**), is:

**linux-c6-x86_64**

The first field is the operating system name (another example is **macos**). The second field is the “OS version”, and the third is the “machine” or “processor” or “architecture” being compiled for, commonly returned from the linux utility **uname -m**.

The OS version field is normally a single letter assigned to the OS distribution (**c** for CentOS, **u** for Ubuntu, **r** for RedHat etc.), followed by a major version number.

The third field is the machine architecture that should be shared across operating systems. E.g. **x86_64** is for the 64 bit version of the Intel x86 architecture, and is used for all Linux and MacOS targets for that architecture.

### *4.1.3 Developing the Environment Setup Script*

For each GPP platform, there must be a script that sets up the environment to build for that target platform. Any time development is done in the OpenCPI tree, such a script must be sourced before any building takes place. It is common to run such a script in any new command window for the target platform.

The script to enable development for a given GPP platform should be in the file:

**platforms/<platform>/<platform>-env.sh**

This script should set up the environment variables that specify the tool-chain necessary to build for it. Some typical environment variables that must be setup in this script include:

- OCPI_DYNAMIC: whether libraries should default to static (0) or dynamic(1)
- OCPI_TARGET_OS: the OS: first part of the “target triple”
- OCPI_TARGET_OS_VERSION: the OS Version: second part of “target triple”
- OCPI_TARGET_ARCH: the third/last part of the “target triple”
- OCPI_TARGET_HOST: the “target triple”
- OCPI_CFLAGS, OCPI_CXXFLAGS, OCPI_LDFLAGS: typical usage

### *4.1.4 Files to Provide Settings for Environment Variables for Framework Development*

The file **<platform>-env.sh** must be created and should define default values for environment variables needed during framework development. The variables specified in this file may be overridden by users when they customize their own environments. This file is used in environment setup scripts to establish defaults based on the platform.

These settings are not those explicitly *required* for the platform, but just defaults. If they are hard requirements, they should be in the ***<platform>*-target.sh** and ***<platform>*-target.mk** files. They only should apply to building the OpenCPI framework, and not for component development.

The most common variable setting in these files is for compiler warning options, e.g.:

```
export OCPI_CXXFLAGS+=-Wno-sign-conversion
```

or

```
export OCPI_CXXFLAGS+=-std=c++0x
```

The two files, **<platform>-target.sh** and **<platform>-target.mk** serve the same purpose for different environments. The first sets variables used in shell scripts and the second sets variables for make files. The variable settings should be the same in each. The settings only apply to building the OpenCPI framework and not to component development.

For all three of these files, the range of settings is best understood by examining all the existing platforms.

### *4.1.5 The Script to Recognize the Currently Running Platform*

The final step is to modify the file **platforms/getPlatform.sh** to recognize when it is running on this new platform and return the new target consistent with the others, with five values: OS, OSVersion, Machine, Target, Platform.

On a CentOS6 platform, this happens:

```
% platforms/getPlatform.sh
linux c6 x86_64 linux-c6-x86_64 centos6
```

On a MacOS platform this happens:

```
% platforms/getPlatform.sh
macos 10_11 x86_64 macos-10_11-x86_64 macos10_11
```

### *4.1.6 Developing a Cross-development Component Build Script*

While the above environment setup script is used when building the core code of OpenCPI (libraries and executables), a different script is used when building software components. This allows for building components for multiple targets in a single pass.

This script is in the file:

**`tools/cdk/include/rcc/<dev-host>=<target-host>.mk`**

Thus the script is specifically describing the tools used when building on a particular development host targeting a particular target GPP host. An example is the file:

**`tools/cdk/include/rcc/linux-c6-x86_64=linux-zynq-arm.mk`**

which contains the script to compile for the zynq version of linux with an ARM processor, when the development host is CentOS6 linux on an X86_64 (64-bit x86) processor.

Thus the required “target” file in a platform's directory contains the target triple used here. E.g. since **`platforms/zed/target`** file contains “l**`inux-zynq-arm`**”, cross-building for the **`zed`** platform on a CentOS6 development host looks for the file named “**`linux-c6-x86_64=linux-zynq-arm.mk`**”

These scripts are responsible for setting the following environment variables whose names include the target name (in this case **`linux-zynq-arm`**):

- Gc_linux-zynq-arm
- Gc_LINK_linux-zynq-arm
- Gc++_linux-zynq-arm
- Gc++_LINK_linux-zynq-arm

These variables are the compile and link commands for C and C++ components respectively.

### *4.1.7 Summary for Enabling Development for a New GPP Platform*

- An installation script to install the tool-chain on the development host
- Environment scripts (3) to set environment variables for the OpenCPI core.
- An environment script to set up environment variables for building C/C++ components.
- Test the installation and build of standard OpenCPI prerequisite software packages.
- Build the OpenCPI core and make adjustments to the code for issues that arise.

### *4.2 Enabling Execution for GPP Platforms*

Execution on GPP platforms requires some command line tools, libraries, and device drivers. The core OpenCPI 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.

# 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 some 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.

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

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 runtime/execution environment.

Enabling development for a platform involves:

- Installing and integrating a development tool-chain that can target the FPGA device on the 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 platform involves:

- Writing specific new VHDL code that supports the particulars of the hardware attached to the FPGA on the platform (except for simulators)
- Verifying that the various portable FPGA test applications execute on the platform.

In the sections below, these two aspects are described in detail.

### *5.1 Physical FPGA 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. Platforms are specific FPGAs connected locally to devices such as:

- Local I/O devices: ADC, DAC, DRAM, Flash, GbE, etc.
- Interconnects: PCIe, Ethernet, etc. 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 attached device on a platform, specific development may be required (described in the Device Development section below). However, in many cases existing device support from other platforms may be reused, since OpenCPI FPGA device support is typically done in a way that is sharable across platforms.

Examples of physical FPGA platforms are:

- Xilinx ML605 development board with a Virtex6 FPGA-attached
- ZedBoard platform with a Zynq SoC (with internal ARM and FPGA).
- Altera Stratix4 development board.
- Xilinx ZC702 development board with Zynq SoC

### *5.2 Simulator FPGA Platforms*

OpenCPI provides a software server infrastructure to make execution on simulators as similar as possible to execution on physical FPGAs, without simulating any external device-related logic. The simulation server environment makes execution on different simulators also similar to each other. Multiple simulator instances may execute simultaneously (subject to any license restrictions).

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
- Mentor Modelsim

Other simulators that may be supported include:

- Aldec
- Xilinx Vivado xsim

## *5.3 Enabling Development for FPGA Platforms*

### *5.3.1 Installing the Tool Chain*

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.

### *5.3.2 Integrating the Tool Chain into the OpenCPI HDL Build Process.*

[[This is a large topic that is currently 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:

1. If the required tools are not available integrated with OpenCPI, they must be integrated (not a small job).
2. If the required tools are available 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 platforms, 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 Portable 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) 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'
```

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) for 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:

- A JTAG support script to enable JTAG-based bitstream loading, whose name is: **jtagSupport_*<platform>***
- A boot-flash loading script to enable scripted loading ot bitstreams to the boot flash, whose name is: **loadFlash_*<platform>***

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 Altera) which may live in the **hdl/vendors/*<vendor>*** directory.

For simulation platforms there must be a script to invoke the simulator underneath the OpenCPI simulator server as invoked by the "**ocpihdl simulate**" command. This script must be called: **runSimExec.*<platform>***

### *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.

An OpenCPI HDL hardware platform is an FPGA with associated devices and slots attached to its pins. The first step to supporting the platform is to determine whether the types of devices and slots attached to the FPGA are already supported by OpenCPI.

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. Common slot types are FMC (FPGA Mezzanine Cards) as defined in the VITA 57 standard, and HSMC (High Speed Mezzanine Card) as defined by Altera. This slot-type support process is described in the "Defining New HDL Platform Slot Types" section below. It generally does not involve writing HDL code, just writing descriptive metadata in XML.

Similarly, 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), although there are certainly exceptions to this. The device support process is described in the "Device Support for FPGA Platforms" section below.

The term "card" is used in OpenCPI to mean a card with additional devices that may be plugged into a slot on the platform. 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.

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 creating a directory with that name wherever the platform support code and metadata should live. In that directory in all cases there should be a "Makefile" containing the following line, usually the last line in the file:

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

Directories for platforms supported in the OpenCPI core repository are in the **`hdl/platforms`** directory, but the directory for any other platform can be anywhere else. If several platforms are being supported together, it is convenient, but not required, to place all their directories in a single **`platforms`** directory since that name is specially supported in the **`OCPI_HDL_PLATFORM_PATH`** environment variable. Use of this environment variable is explained in the HDL section of the OpenCPI Component Development Guide.

Then the "platform worker" (source code and OWD/XML) and other platform-specific files are placed in this directory. In summary, the steps to enabling an HDL platform for execution are, (assuming the development tools are enabled):

- Determine and identify slot types and devices for the platform.
- Define any new slot types and add them to OpenCPI.
- Define and establish “skeletons” for all new devices on the platform.
- Define the platform worker, with associated hardware-related metadata and files.
- Implement the platform worker and perform basic tests, with *no* devices enabled.
- Implement any new devices for the platform, and test the platform *with* devices (both pre-existing and new).

### *5.4.1 Creating the XML Metadata Definition of the Platform*

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 runs 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” (a worker that touches pins). 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:

- The top level XML element is “**`HdlPlatform`**”
- The spec being implemented is “**`platform-spec.xml`**”.
- The slots present on the platform are indicated by **`slot`** elements
- The devices 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. Signal elements are described in the device support section below.

#### *5.4.1.1 Slot Elements in a Platform XML File*

Each slot element of the platform XML file declares 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 CDK, which is automatically searched whenever a platform XML file is processed. Examples of slot types are:

- **`fmc_lpc`**: the “low pin count” variant of the “FPGA Mezzanine Card” from VITA57.
- **`fmc_hpc`**: the “high pin count” variant of the FMC cards from VITA57
- **`hsmc`**: the High Speed Mezzanine Card from 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.
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 correspond to the names found in a platform's constraint file, which is normally supplied separately and is not defined nor modified by OpenCPI.

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 since there are some tools and systems where the case of such signals matters in the constraints file.

There is one other aspect to slot elements in the platform XML file: slot signal mapping. Normally, the 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 slot element, indicating the correspondence between what the standard (e.g. VITA57) says and what this platform's constraints file and documentation use as a signal name for that slot signal.

As an example, the ZedBoard platform has an 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). So 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 through the slot's connector to cards that *do 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'/>
  <signal slot='DP0_C2M_P' platform=''/>
  <signal slot='DP0_C2M_N' platform=''/>
</slot>
```

Finally, 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.

Card-based devices are not considered part of the platform, and thus constraints for card-based devices are handled differently and are not in the platform's constraints file.

#### *5.4.1.2 Device Elements in a Platform XML File*

Device elements indicate devices that are part of the platform and are directly attached to the platform FPGA's pins. Device elements indicate these things:

- *Required:* the HDL device worker supporting this device (like a device driver).
- The (optional) name of the device, if not based on the device worker's name.
- Which parameter settings should be used for the device worker for this device.
- 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).

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'/>
```

Similar to slot names, if there is no “**`name=`**” attribute provided for a device element, the worker name is used, and if there are multiple devices using the same worker, a zero-based ordinal is appended.

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 can 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 **false**.:

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

Like slots, 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.

Also like slots, 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>
```

### *5.4.2 Creating the Platform Worker*

While a platform worker's XML (OWD) has extra elements to describe the platform's hardware (devices and 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.

We first describe the required properties and ports required by the OCS for all platform workers, followed by a number of optional worker port types that are in fact unique to platform workers that no other type of worker can use.

#### *5.4.2.1 Properties Required for all Platform Workers.*

The spec (OCS) for platform workers contains properties that should be supported by all platform workers. These are listed in **hdl/platforms/specs/platform_spec**.xml.

Supporting the properties falls into these categories:

1. Readable constants that may be specified as parameters in the OWD itself.
2. Readable constants whose values may be specified in the worker code.

3. Properties directly supported by instantiating existing modules in the worker.
4. Properties directly supported by special ports required of the worker
5. Properties that require specific code support in the platform worker.

Nearly all of them are supported by existing primitive modules that all platform workers must instantiate, and some are simply read-only values that the platform worker code (or OWD) can specify values for.

It is best to look at some of the existing platform workers to see how these properties are supported (e.g. **`zed`**, **`ml605`** or **`alst4`**).

Here is a table with the properties and their category listed.

*Table 2: Platform Worker Required Properties*

| **Name** | **Type** | **Access** | **Cate-gory** | **Description** |
|---|---|---|---|---|
| UUID | ULong*16 | Readable | 4 | Unique ID of bitstream file |
| timeNow | ULongLong | Readable | 3 | Current time-of-day (32.32 GPS time) |
| romAddr | Ushort | Writable | 3 | Address for reading bitstream metadata |
| romData | ULong | Readable | 3 | Data when reading bitstream metadata |
| nMemories | Ulong | Readable | 1 or 2 | How many memories the platform has |
| memories | Ulong*4 | Readable | 2 | Information about individual memories |
| nLEDs | Ulong | Readable | 1 or 2 | How many LED indicators are present |
| LEDs | ULong | Readable +Writable | 5 | Actual LED settings, up to 32, LSB is LED 0 |
| timeStatus | Ulong | volatile | 3 | Status of the the time service |
| timeControl | Ulong | writable | 3 | Control register for the time serv.ce |
| timeDelta | ULongLong | writable | 3 | Perform hardware subtraction via timeNow |
| ticksPerSecond | Ulong | volatile | 3 | Timekeeping clock rate |

#### *5.4.2.2 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 worker has three port types:

- Control plane access ports, providing off-chip access to the on-chip control plane.
- Metadata access ports to provide access to built-in bitstream metadata via properties
- Timebase ports to provide the basis for timekeeping on the platform.
- Interconnect ports to allow external interconnect hardware interfaces to support data flow (e.g. PCIe, AXI etc.).

To support the (single) control plane access port (called a "**cpmaster**" port), the platform worker must provide an addressable path from the controlling processors's software into the FPGA. This usually involves adapting a 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't control clock/reset.

An 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 hardware, to the OpenCPI control plane protocol. This module is instanced in the Zed platform worker and connected to the 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:

```
-- Adapt the axi master from the PS to be a CP Master
cp : axi2cp
  port map(
    clk     => clk,
    reset   => reset,
    axi_in  => ps_axi_gp_out,
    axi_out => ps_axi_gp_in,
    cp_in   => cp_in,
    cp_out  => cp_out
    );
```

The **metadata** port is supported by connecting properties to elements of the port record, by inserting these lines into the platform worker:

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

The "clk" signal is the platform's control clock (see above).

The timebase port is to provide signals that are the basis for timekeeping on the platform. They are a clock, a PPS input signal and an optionally connected PPS output

signal. The platform worker should use a clock that is best suited for timekeeping, which usually means the one with the least jitter and drift over the short term. On some platforms this is simply the same as the control clock, but on others there may be a clock with better performance for this purpose and the platform worker should use it.

The fourth type of port on the platform worker is the one that enables off-chip data streaming (messaging) to and from other FPGA or software workers. It is called the "**unoc**" port (for "micro-network-on-chip"). [Note this port is becoming obsolete and replaced by the scalable data plane, and thus will not be documented here]. On some platforms (e.g. PCIe-based), both the control plane and data plane use the same off-chip interconnect, and the platform worker simply provides separate access paths to that interconnect for control and data purposes. The control plane acts as bus slave, whereas the data plane may act as both master and slave.

#### *5.4.2.3 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 are ones that 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.

#### *5.4.2.4 Building the Platform Worker Using its Makefile.*

Like any other worker, a platform worker has a worker Makefile that is used for various purposes. Since the HdlTarget and HdlPlatform are implicit for a platform worker, 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 is simply:

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

In addition to building the platform worker, you can also build platform configurations here 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.

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.

### *5.4.3 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 aspecs:

- 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 file* (either platform configuration 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' card='lime-zipper-fmc'/>
</HdlConfig>
```

### *5.4.4 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” 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” or “biascapture” that only flow data in one direction between the FPGA platform and the processor.

### *5.5 Device Support for FPGA Platforms*

Device support development involves the creation of specialized workers that enable OpenCPI to use the devices attached to an HDL platform. *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.

Device workers are like normal application workers with one important difference: they have signal connections with hardware attached to pins of the FPGA. The control interface and data interfaces to device workers are the same as an application worker. 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.

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

```
<HdlDevice Language='vhdl' spec='dac-spec'>
  <port name='in' datawidth='8'/>
  <signal output='valid'/>
  <signal output='data' width='8'/>
  <signal input='ready'/>
</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 "spec" common to all DAC devices (properties and ports). It can add its own properties as required.

The "**`port`**" 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 for this (simplistic, single clock domain) device worker would be:

```
in_out.take <= in_in.ready and (ready or not in_in.valid);
valid       <= in_in.ready and in_in.valid;
data        <= in_in.data;
```

### *5.5.1 Device Component Specs and Device Worker Modularity*

In order to ensure that similar devices have common application-visible behavior, OpenCPI has *device class specs* (DCS) that represent commonly used classes of devices. This is a specialization of the component specifications (OCS ) that are the basis of multiple implementations (application 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 DCS; a device class is represented by its DCS. There is no difference in syntax between an OCS and a DCS.

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. It possible to develop "swiss-army-knife" device workers for multifunction chips that intend to avoid using resources for parts that are unused. However, the recommended practice of OpenCPI is to be 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 timestamping functionality.
- Upconverters and downconverters between IF/Baseband and RF.
- Clock generators
- DRAM

### *5.5.2 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 the actual 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.

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.

### *5.5.3 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 will promote 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.

### *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 may be 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 be different. That may result in the existing device worker being refactored to share functions that it did not share before.

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 worker that the subdevice supporting it. 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.

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 platform (in the platform OWD), it will be associated on this platform with the si5351 device worker.

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

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 worker. The “reset” signal tells the subdevice that this worker is being reset, and the “raw” subrecord is conveying the raw property signals into the device worker down to the subdevice. Similarly, the raw output signals from the subdevice are conveyed back as the device worker's raw property output signals. The raw property signals are defined in the component developer's guide.[ref]

When the “rawprop” connection is not sufficient for all of the shared functionality in the subdevice, another type of connection is used, which is a customized set of signals. This is called a “devsignals” connection.

Since platforms and cards declare which devices the 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.

### *5.5.5 Higher-level Endpoint Proxies Suitable for Applications*

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 software radio system as such, an endpoint proxy can represent any application-level source or sink of data.

### *5.5.6 XML Metadata for Device Workers/Subdevices/DeviceProxies/EndpointProxies*

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

- Device Workers
- Device Proxies
- Subdevices
- Endpoint Proxies

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 devices to subdevices. If a device worker uses a subdevice, 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'

Endpoint proxies have multiple “slave” child elements indicating which other workers they are using to implement their higher level function for applications.

#### *5.5.6.1 Signal XML Elements for Device Workers and Subdevice Workers*

Signal elements are children of the top level HdlDevice element, and contain the following attributes. They identify the signals that are attached to the pins of the FPGA.

- ***Input='name':*** identifies a signal as an input and provides its name.
- ***Output='name':*** identifies a signal as an output from and provides its name.
- ***Inout='name':*** identifies a signal as a tristate signal and provides its name.
- ***Bidirectional:='name':*** identifies a signal as usable in either direction and provides its name
- ***Width='nn':*** specifies that the signal is an array of signals with the width specified.
- ***Differential='true|false':*** specifies that the signal is differential

All signal names are restricted to legal identifies in VHDL and Verilog. Inout signals are only really tristate at the top level of the design (in the container). Inside device workers a signal identified as “inout”, represents three signals, with the suffixes “**`_i`**” for input, “**`_o`**” for output, and “**`_oe`**” for the output enable. Containers ensure that tristate signals result in a signal tristate IO pin. Similarly, differential signals represent two signals, one with a “**`_p`**” suffix, and one with a “**`_n`**” suffix.

#### *5.5.6.2 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. Its attributes are:

- ***Name='name':*** the name of the port (defaults to “rawprops”)
- ***Optional='true|false':*** indicates whether a connection to this port is required.
- ***Count='nn':*** indicates this is an array of raw property ports.
- **Master='true|false':** indicates who generates addresses for raw accesses

Raw property signals are listed in the following two tables.

*Table 3: Master-to-Slave Raw Property Signals*

| Name | Type | Purpose |
|---|---|---|
| present | bool_t | Indicate to slave whether port is connected |
| reset | bool_t | Indicate to slave whether master is being reset |
| renable | bool_t | Indicate to slave that a raw read is in progress |
| wenable | bool_t | Indicate to slave that a raw write is in progress. |
| addr | ushort_t | Indicate to slave the address of the current read/write access. |
| benable | Slv(3 downto 0) | Indicates to slave which bytes are being read/written |
| data | word32_t | Write data 4 bytes wide. |

*Table 4: Slave-to-Master Raw Property Signals*

| Name | Type | Purpose |
|---|---|---|
| done | bool_t | Indicate to master that the access is done |
| data | word32_t | Read data 4 bytes wide. |
| present | bool_array_t | Indicate to master which other masters (device workers) are present. Range is 0 to raw_max_devices-1 |

#### *5.5.6.3 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 two attributes:

- ***Name='name':*** the name of the port (the default is “dev”).
- ***Optional='true|false':*** whether this port must be connected or not.
- ***Count='nn':*** indicates an array of similar ports with the same signals.
- ***Signals='sig-file':*** indicates a file containing a top level Signals element consisting of “signal” child elements.

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

#### *5.5.6.4 Slave XML Element for Endpoint Proxies.*

When an EndpointProxy requires more than one slave to implement its application-relevant endpoint type, it uses the “slave” child elements to describe them (rather than simply using the “slave” attribute at the top level). The attributes of the “slave” child elements are:

- ***name='device-worker-name':*** The name, including authoring model suffix, of the required slave device worker, e.g. “**`lime_adc.hdl`**”.

#### *5.5.6.5 The Supports XML Element for Subdevices.*

Subdevices indicate which device workers they support by using “supports” XML elements, and 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="rprops" index='1'/>
  <connect port="dev" to='dev' index='1'/>
</supports>
<supports worker='lime_adc'>
  <connect port="rawprops" to="rprops" index='0'/>
  <connect port="dev" to='dev' index='0'/>
</supports>
```

The attributes of the “supports” element are:

- ***worker='worker-name':*** which names the device worker it supports
- ***index='which instance':*** which identifies which device of that type is being supported by the subdevice.

The attributes of the “connect” child element of the “supports” element are:

- ***port='subdevport':*** the port on the subdevice that should be connected
- ***to='devworkerport':*** the port on the supported device worker that should be connected
- ***index='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.

### *5.5.7 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.

# 6 Enabling GPU Platforms

### *6.1 Enabling Development for GPU Platforms*

Installation script/procedure for tool kit/driver package.

Describe ocpiocl.

Do we ever need to do anything else for real compilation?

Installation of libraries, defining where they are.

Test build of existing opencl components.

### *6.2 Enabling Execution for GPU Platforms*

OpenCL should in general do the right thing.

Testing the existing OCL components and test applications.

### *6.3 I/O Device Support for GPU Pßlatforms*

Perhaps actually video output, openGL integration?

None at this time.

# 7 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 worker.

Tool-chain

container

artifact

build flow

cross-build, cross-compile

development host

platform

card

slot

platform worker

device worker

platform configuration

HDL container

default container
interconnect
control plane
data plane

# 8 From PPT:

OpenCPI Platform Development Briefing
FPGA Platform Support in OpenCPI
OpenCPI applications are composed from components.

Components are written and compiled for various platforms:
GPPs running C/C++-based software components
GPUs running OpenCL-based components
FPGAs (or FPGA simulators) running VHDL-based components

When applications are executed, each component executes:
On a platform (e.g. Linux on x86_64, or Xilinx or Altera FPGA)
In a container (a runtime environment established on that platform)

Three things must exist for an OpenCPI application to run:
Components must be written and built/compiled for platforms.
Applications must be specified as consisting of components
Platforms must be available and enabled to run containers

This briefing describes how FPGA platforms are enabled and used
FPGA Platform Development in OpenCPI
We 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.
each may host a container in which to execute components.

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

Platforms are specific FPGAs connected locally to:

Local I/O devices: ADC, DAC, DRAM, Flash, GbE, etc.
Interconnects: PCIe, Ethernet, etc. to talk to other containers
Slots: FMC, HSMC, mezzanine card slots.
E.g., a Xilinx ML605 has PCI Express, DRAM, and 2 FMC slots.

Platform FPGAs host containers in which components execute.
FPGA Platforms in OpenCPI
An FPGA with attached Interconnects, Devices, and Slots
Platform logic, Device/Interconnect Logic, Application Logic
ML605 Platform Example
The FPGA: Vendor: Xilinx, Family: Virtex6, Part: xc6vlx240t
Interconnect(s): PCI Express (and GbE - not supported yet)
Devices: DVI video output (not supported yet)
Slots: ISO/VITA 57.1 Slots: FMC-LPC and FMC-HPC
ZedBoard Platform Example
The FPGA: Vendor: Xilinx, Family: Zynq-7000, Part: xc7z020
Interconnect(s): AXI (to ARM) (and GbE - not supported yet)
Devices: HDMI video output (not supported yet)
Slots: ISO/VITA 57.1 Slot: FMC-LPC
ALST4 Platform Example
The FPGA: Vendor: Altera, Family: Stratix4, Part: ep4sgx230
Interconnect(s): PCI Express (and GbE - not supported yet)
Devices: HDMI video output (not supported yet)
Slots: Altera HSMC card slots (2)
Slots and Cards
Many FPGA platforms have slots/connectors for add-in cards.
E.g. FPGA Mezzanine Cards (ANSI/VITA FMC Standard 57.1)
E.g. HSMC (Altera's High Speed Mezzanine Card)

OpenCPI defines Slot Types.
A slot type has pins, direction, and default signal names
Typically documented in the standard specification (e.g. ANSI/VITA 57.1)
Platforms specify what slots are present, of what types

Using the <slot> XML element in the HdlPlatform XML file.

Separate from platforms, OpenCPI defines Cards

Cards have a slot type, and on-board devices

Cards specify wiring from devices to slot pins/signals.

Cards are like platforms: they have devices

But no FPGAs, just devices connected to slot pins

FPGA Platform Development on OpenCPI

As consistent as possible with FPGA "application" development (OpenCPI sub-assemblies on FPGAs).

Platform development involves the creation of specialized types of workers that enable the HDL platform.

Device Workers:  like "device drivers" for specific FPGA-attached hardware devices — workers that talk to devices using ad hoc, device-specific signals and I/O pins.

Platform Workers:  device workers with special responsibilities for enabling the whole platform.

Two other modules are automatically generated based on XML specifications:

Platform Configurations

a pre-linked core combining a platform worker with some device workers

Containers: top-level bitstream designs

Deploying an application subassembly on a platform configuration.

Device Workers

Device Workers:  like "device drivers" for specific FPGA-attached hardware.

Device Workers are special workers that have ad-hoc signals to some device attached to the FPGA

Still are controlled by a normal "WCI" control interface.

Have data-plane interfaces to provide data to/from application workers.

Have ad-hoc signal connections to hardware

Designed to be portable

e.g. same DAC on different FPGAs, or on different cards

Designed to be replicated

E.g. multiple ADCs of the same type connected to one FPGA

Analogous to "device drivers" in software.

Platform Workers

Platform Workers are specialized Device Workers

Have ad-hoc signals to the platform hardware

Not for any individual device (that's what other device workers do)

Perform standardized "platform enabling functions"

Provide control path from outside the FPGA (e.g. "register access")

Provide timekeeping services and clocks

Provide external access to bitstream metadata

Standard control interface and properties

Platform name, fpga device, chip serial number

Switch settings (input), LEDs (output)

Bitstream UUID and metadata

Time-of-day, timekeeping/PPS status

Can have platform-specific properties/controls

E.g. PCI device ID assigned by BIOS for PCI-based platforms

Platform Worker Ports

Control Interface (WCI), like any worker

Control operations and properties

But platform workers also bootstrap everything, including WCIs…

Data ports, when data producing/consuming devices cannot be optional (i.e. when they can't have their own device workers)

Rare, theoretical case

Special "infrastructure" ports to provide key services

Time Service (provide platform timekeeping)

This port is required of all platform workers
Metadata Master (provide access to metadata via WCI properties)
This port is required of all platform workers
Control Plane Master (provide a control path into the device)
This port is optional, but must be provided by some device if not here
Data Plane NOC Master (for data connections)
Optional port, when platform supplies an interconnect path to the chip

Platform (Worker) Description XML
The <HdlPlatform> XML file defines a platform and its worker

This file is used to define the platform worker
Uses the "platform_spec" OCS as its "spec" (in hdl/platforms/specs)
Defines ad-hoc platform signals
Defines any platform-specific properties
Defines any optional infrastructure ports (CP Master, NOC Master)

This file also lists the available devices and slots on the platform
Each device is defined by its device worker
A device (and device worker) can be mentioned more than once
I.e. there might be multiple instances of a device on a platform
Each slot is named and has a type
Platform Configurations
Specifies one particular "configuration" of a platform

Builds on the minimum base of support for the platform
Base support XML is the <HdlPlatform> XML file
Base support code is the platform worker code.

The "configuration" of the platform consists of:
Which devices are supported and instanced in this configuration
Which device(s) provide control plane (if not the platform worker)

A configuration may be constrained or optimized

Constraints to physically place/constrain device workers

Creates a platform configuration "core" to use for apps

Analogous to "an OS kernel with built-in device drivers".

Does not preclude more devices added in container for the app

Platform Configuration Description XML

Is a <HdlConfig> XML document

Platform is implied when it is in a platform's directory

Otherwise use the "platform" attribute

Name is implied by the filename of the XML file

Otherwise use the "name" attribute

Child elements under <HdlConfig> are <device>

Specifies which of the platform's devices are included (<device …/>)

Device "Name" attribute matches some <device> in platform XML

"control" boolean attribute says whether a device provides a control path to the platform (when the platform worker doesn't do it).

Default is no devices at all: the base configuration

No source code associated with Platform Configurations

All is generated

Future: Specific constraints, Clock specs, Multiple control paths

Containers

A container deploys an application subassembly on a specific (FPGA) platform configuration.

To specify a container you specify:

The application assembly to be deployed

If container is specified in an assembly directory, that assembly is implied.

The platform configuration on which the assembly should run

If just the platform is specified, its base configuration is implied.

Top level connections

Application external ports to devices or interconnects

Special case: can connect devices to interconnects

Example XML

<HdlContainer platform='ml606/ml605_flash_dac' assembly='modem'># <connection external='in' interconnect='pcie'/># <connection external='out' device='dac'/>#</HdlContainer>

I.e. deploy the "modem" assembly on the "ml605+flash+dac" configuration, with input from PCIe and output to DAC

No source code associated with Containers: all is generated

Containers in HDL Assembly Makefiles

A Default Container is defined as:

All external ports of the assembly are connected to the interconnect

No device worker connections are made

E.g. PCI Express, or (future) ethernet

If no external ports, a standalone container is built (e.g. tb_bias)

Two "make" variables specify the containers/bitstreams to build

DefaultContainers=

If specified empty, no default containers are built

If not specified at all, default containers are built for "base" platform configurations for platforms requested in HdlPlatforms

If specified not empty, only those default containers are built

Containers= specific XML-based containers

The names of HdlContainer xml files to build

Using specific platform configurations

Specifying external port connections to devices and interconnects

Example HDL Assembly Makefiles

Simplest assembly/bitstream Makefile is:

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

The assembly XML is assumed to be in <cwd>.xml

Implies: build default containers for whatever platforms are requested

No specific containers are indicated

More complex example

DefaultContainers=isim_pf ml605_flash

Containers=ml605_ADC

include $(OCPI_CDK_DIR)/include/hdl/hdl-assembly.mk

Build default container for base config of isim_pf and "flash" config of ml605

Build a specific container as described in ml605_ADC.xml

Another Example

DefaultContainers=

Containers=ml605_ADC

include $(OCPI_CDK_DIR)/include/hdl/hdl-assembly.mk

Build no default containers, only one in ml6065_ADC.xml

Clocks

The platform worker is responsible for producing two clocks, with no particular relationship:

The control plane clock

The timekeeping clock

The control plane clock is used for:

All workers' control interfaces

All control plane infrastructure

Default clock for all other worker ports (data, time, etc).

The timekeeping clock drives the timekeeping system

Should be the "best" (drift, PPM, etc.) clock available in the HW.

Is used to keep the notion of time of day, sync'd to PPS input

Provides "time of day" to any worker that wants it

The platform may provide other clocks

Slots and Cards

Platforms have slots of certain types.

E.g. ML605s have one FMC_LPC slot and one FMC_HPC slot.

E.g. Zedboards have one FMC_LPC slot

A platform's slots are defined by:

type

non-default signal names for slot pins (different than the slot type spec)

Platform configurations may indicate specific cards in specific slots.

Containers may indicate or imply specific cards in specific slots

Cards have devices just like platforms have devices.

When a device is used, its device worker is instantiated and connected.

Some devices on a card can remain unused and not instantiated

Subdevices and sharing I/Os

Device workers support hardware that performs a logical function.

I.e. a function an application may request

Device workers should not control multiple functions

Which would force logic to be present when functions are not used.

Just because a device might have 2 functions (e.g. ADC+DAC), does not mean one device worker should control both functions.

Modular devices workers are good, but raise problems:

Some hardware is shared between functions

E.g. one SPI or I2C to configure multiple functions.

Solution is subdevices:

They implement a shared module used by multiple device workers.

Present and needed when any of the device workers are present.

Instantiated and connected automatically as needed.

Subdevices

A subdevice is generally a device worker with no WCI

It "serves" multiple actual device workers.

Built like a normal device worker.

It has the signals for external pins that must be shared between its "client" device workers.

E.g. it would control SPI pins that its device workers need to share.

It defines worker ports that are connected to its clients.

They must be optional since not all clients may be present

They may be a "devsignals" ports when the signals are custom.

They may be "rawprop" ports when the subdevice supports sharing of a lower level access path (e.g. SPI or I2C) to hardware-based properties.

Client device workers declare their requirements for subdevices

And how they are connected to them

Subdevices: example

The shared module for Lime chip SPI and clocking options:

```
<HdlDevice language="vhdl">
  <ComponentSpec nocontrol='true'/>
  <rawprop name='props' count='2' optional='1'/>
  <devsignals count='2' optional='1' signals='lime-spi-signals'/>
  <Signal Output="reset"/>
  <Signal Input="sdo"/>
  <Signal Output="sclk"/>
  <Signal Output="sen"/>
  <Signal Output="sdio"/>
  <Signal Input="rx_clk_in"/> <!-- could be rx_clk_out from lime -->
  <Signal Input="tx_clk_in"/>
</HdlDevice>
```

Note: no real component spec.  Completely custom w/ no control

Note: rawprop slave is x2 to be ready for lime_rx and lime_tx

Note: devsignals slave is x2 to be ready for lime_rx and lime_tx

Note: both are optional since parts may not be connected

Note: devsignals port has a custom set of signals for this worker

Defined in the file lime-spi-signals in the specs directory

Subdevices: example: the client devices
The OWD for the lime_tx device worker:

```
<HdlDevice language="vhdl" FirstRawProperty="dc_regval" spec='qdac-spec'>
  <property name='Present' type='uchar' volatile='true'/>
  <property name='ClkMode' type='uchar' parameter='true' default='0'/>
  <xi:include href='lime-properties.xml'/>
  <StreamInterface Name="in" DataWidth="32"/>
  <rawprop name='rawprops' master='true'/>
  <requires worker='lime_spi'>
    <connect port="rawprops" to="props" index='1'/>
    <connect port="dev" to='dev' index='1'/>
  </requires>
  <Signal Output="tx_clk"/>
  <Signal Output="txen"/>
  <Signal Output="tx_iq_sel"/>
  <Signal Output="txd" width="12"/>
</HdlDevice>
```

Note: rawprop and devsignals ports are for subdevice connection(s)
Note: explicit declaration that this worker REQUIRES the subdevice
and connects to "index 1" (for tx) of the subdevice.
Note: this device worker still has direct signals to the device
Note: this device worker actually declares the same properties as rx.
Subdevices: custom signals in worker ports
What needs to be shared may have custom signals
<devsignals master='true' signals='lime-spi-signals'/>
The "devsignals" port type is for this purpose.
The "signals" attribute is the file full of signal declarations
Allows different workers and subdevices to use the same list of signals.
The direction of signals is relative to the "master"
The "signals" file for the "devsignals" port for lime_spi:

```
<signals>
  <signal input='rx_clk_in'/> <!-– might be used by either side -->
  <signal input='tx_clk_in'/> <!-– might be used by either side -->
  <!-- To let each side know whether they are both present -->
  <signal input='rx_present'/>
  <signal input='tx_present'/>
</signals>
```

Note: a way to tell each client whether the other is present

Note: this is separate from the "rawprop" port, which is a standard port type for device workers (not application workers).

Note: this is just a bundle of signals that fans out to all clients.

FPGA Architecture

Devices and Subdevices: Example

Build flow (bottom to top)