OpenCPI Platform Development Guide

DRAFT/Incomplete for review

Revision History

Revision	Description of Change	Date
0.01	Initial	2014-12-15
0.5	Release partial draft	2015-02-27

Table of Contents

1Introduction....4
1.1References....5
2OpenCPI Platforms....6
2.1Development and Execution....7
3Enabling the Development Host....8
3.1Operating System Installation....9
3.2Development Tools Installation for the Development Host....10
4Enabling GPP Platforms....11
4.1Enabling Development for new GPP platforms....12
4.1.1Developing a tool-chain installation script....13
4.1.2Developing an environment setup script....13
4.1.3Developing a cross-development component build script....14
4.1.4Summary for Enabling Development for a new GPP platform....14
4.2Enabling Execution for GPP platforms....15
5Enabling FPGA Platforms....16
5.1Enabling Development for FPGA platforms....17
5.1.1Installing the tool chain....17
5.1.2Integrating the tool chain into the OpenCPI build process....17
5.1.3Building all the existing portable HDL code....17
5.2Enabling Execution for FPGA platforms....18
5.2.1Creating the metadata definition of the platform....18
5.2.2Creating the platform worker....18
5.2.3Testing the basic platform without devices....18
5.3I/O Device Support for FPGA platforms....19
5.3.1Creating device worker metadata....19
5.3.2Subdevices – sharable modules that support multiple devices....19
5.3.3Cards....19
6Enabling GPU Platforms....20
6.1Enabling Development for GPU platforms....21
6.2Enabling Execution for GPU platforms....22
6.3I/O Device Support for GPU platforms....23
7Glossary....24
8From PPT:....26

1 Introduction

This document describes how to enable OpenCPI components and applications to run on new platforms. 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), ARM (Xilinx/Altera/TI)	Compiler tool-chains	Operating System, Libraries, Drivers	
FPGAs	Xilinx (virtex, zynq) Altera (stratix, cyclone) Mentor/Modelsim	Synthesis Place&route Simulation	Bitstream loading Control Plane Drivers Data Plane Drivers	
Graphics Processors	Nvidia Tesla AMD FirePro	Compilers, Profilers	Execution Management Drivers Data Plane Drivers	

1.1 References

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

Table 1 - Table of Reference Documents

Title	**Published By**	**Link**
Introduction	W3C	Public URL: http://www.w3.org/TR/xml
Installation		
Component Development		
Application Development		

2 OpenCPI Platforms

OpenCPI provides a consistent model and framework for component-based application development and execution on various (“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 simply has processors of various types that are “places for components to run”, each processor is itself part of a hardware subsystem and these subsystems are wired together using some interconnect technologies (networks, busses, 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 the 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.

The scope of a system can be a small embedded system that fits in your pocket, or racks full of network-connected hardware that acts 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 are in the system.

Our most common and simple example “system” is the ZedBoard from Digilent (zedboard.org), which is based on 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 in to slots on the PC's motherboard, and those cards have processors (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.

2.1 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 “bag of bits” that execute 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. 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 many 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 execution platform for the GPP/processor of that system.

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 installation guide describes these 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, which includes special tool executables, as well as runtime libraries that support both tool execution as well as application/component execution on the development host.
- Building and testing the OpenCPI component libraries and example applications

There are several layers of configuration and installation in this process, and for a new development platform, these 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 are generally “manual”, requiring written instructions. The latter two can be scripted. Both the instructions and scripts should then be placed in the OpenCPI source distribution for the benefit of other users of the same development host installation.

Documentation and scripting created for development hosts and GPP runtime platforms should be placed in a new directory under the “platforms” subdirectory of the OpenCPI source tree. See the README file in that platforms directory. For a new development host type or GPP runtime platform, a directory for the host type should be created, with its own README file. Several other files should be created in this new directory, as described below.

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 this 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 pushed to the OpenCPI repository.

In summary, the task to enable this aspect of the installation is to create the README file explaining the initial OS installation, and the retrieval of the OpenCPI distribution. Once the distribution is retrieved, 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.

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 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
- tcl

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
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 are required for various tools for other 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.

In addition to the above scripts, two more items must be created for the development 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 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 5 values: OS, OSVersion, Machine, Target, Platform.

4 Enabling GPP Platforms

Enabling a GPP platform requires two different aspects: enabling development/cross development, and enabling execution. If the platform is a development host, then it implicitly has been enabled for self-hosted development rather than cross-development. In both cases, enabling for development allows building component and application binaries for the platform. This also enables building the OpenCPI core runtime libraries and runtime command line tools for execution in 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, required 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. For most Linux systems, this amounts to a single command to install the default tool chain, as described above. For the most popular OpenCPI development system, running the CentOS6 Linux distribution, this is accomplished by the single command:

```
% sudo yum -y groupinstall "development tools"
```

When the tool chain desired for OpenCPI execution on the development host is *not* the default tool-chain for that system, 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 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 Developing an environment setup script

For each GPP development 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 identify the platform and the tool-chain necessary to build for it. Some typical environment variables that must be setup in this script include:

- OCPI_KERNEL_DIR: location of linux kernel for building device drivers
- OCPI_TARGET_PLATFORM: the name of the target platform
- OCPI_BUILD_SHARED_LIBRARIES: 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_ARCH: the architecture parameter used when building Linux device drivers
- OCPI_CFLAGS, OCPI_CCFLAGS, OCPI_LDFLAGS: typical usage

4.1.3 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 "**`linux-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.

4.1.4 Summary for Enabling Development for a new GPP platform

An installation script to install the tool-chain on the development host.

An environment script to set up environment variables for building 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 any 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. [more clarity here].

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.

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 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).

Another class of FPGA platform is simulators. They are also a place where OpenCPI components may execute, supported by appropriate infrastructure logic.

Analogous to preparing the support for a GPP platform, enabling an FPGA platform involves these steps:

- Installing and integrating a development tool-chain that can target the FPGA 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”.
- Writing specific new VHDL code that supports the particulars of the hardware attached to the FPGA on the platform.
- Verifying that the various portable FPGA test applications execute on the platform.

The first two steps are considered enabling the development environment, while the last two are enabling the runtime environment.

5.1 Enabling Development for FPGA platforms

5.1.1 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 execute 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. Thus polluting your environment with settings from multiple tools and vendors is frequently a source of problems.

5.1.2 Integrating the tool chain into the OpenCPI build process.

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

The scripts that wrap and execute FPGA tools are found in tools/cdk/include/hdl.

5.1.3 Building all the existing portable HDL code

Primitive libraries, components, adapters, devices, subject to specific exclusions.

Essentially go through the build flow, up to, but not including, platforms and assemblies.

5.2 Enabling Execution for FPGA platforms

Introduce device workers, cards and slots. Details of slots, but defer cards to IO section.

5.2.1 Creating the metadata definition of the platform

Platform worker xml, plus slots.

5.2.2 Creating the platform worker

Filling in the details in the xml

Writing/copying code

Potentially adding cores/primitives, but maybe just having subsidiary files.

Building the platform worker, and the default platform configuration.

Interconnect support for control and data.

5.2.3 Testing the basic platform without devices

Without supporting any of the devices on the platform, you can build and run various tests. This may be the first chance to test the full bitstream build flow.

Q: can we have a null platform to test the full build flow?

5.3 I/O Device Support for FPGA platforms

Device workers, cards and slots. Describe cards in detail..

How the devices are included in the platform metadata

How signal prefixes work.

5.3.1 Creating device worker metadata

The general case is reusable and not specific to any platform.

Adding device declarations to platform (worker) metadata

5.3.2 Subdevices – sharable modules that support multiple devices.

5.3.3 Cards

Like platforms in that they reference various devices.

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 platforms

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)