# OpenCPI
# User Guide

*OpenCPI Release: v2.4.7*

*Revision History*

| Revision | Description of Change | Date |
|---|---|---|
| 1.0 | Creation | 2020-01-02 |
| 1.1 | Add glossary snippet as the last chapter | 2021-03-15 |
| 1.2 | Add chapter on how to document OpenCPI assets | 2021-05-23 |
| 1.3 | Remove AV GUI references and add OpenCPI GUI references, add tutorial project, update build chapter to use ocpiadmin, add hyperlinks to man pages and glossary, update chapter on how to document OpenCPI assets to current asset doc system functionality | 2021-10-26 |
| 1.4 | Add content to system.xml for worker attributes and plugin discovery in chapter 7 | 2022-07-18 |
| 1.5 | Add new chapter on OpenCPI Linux kernel device driver | 2022-08-31 |
| 1.6 | Update “Documentation for OpenCPI” figure, include latest tables of supported platforms, update GUI chapter to reference OpenCPI User Guide on doc website | 2022-10-19 |
| 1.7 | Remove chapter on how to document OpenCPI assets and move it to Writer Guide, update “Documentation for OpenCPI figure to include Writer Guide, update text to refer to Writer Guide | 2022-11-11 |

## Table of Contents

# 1 Overview

This document provides basic post-installation information intended to help an OpenCPI user get started on OpenCPI development. It also contains general configuration information that needs to be referenced over time that does not fall into the other more specific reference documents.

This document first reviews the concepts and terminology needed to understand the OpenCPI environment and documentation. Later sections address specific topics for using OpenCPI over time, such as configuration files and environment variables, and guidance for using various tools external to OpenCPI.

This document assumes that you have access to one of the OpenCPI-supported development systems (listed in the "Table of Supported Development Systems (Hosts)" in the ***OpenCPI Installation Guide*** and in this document) that has been installed and initially configured as an OpenCPI development host and on which at least an OpenCPI-supported FPGA simulator platform has been set up. This prerequisite installation is based on the installation document ***OpenCPI Installation Guide***.

Optionally, this system can also have installed the OpenCPI GUI development tool.

# 2 Overview of OpenCPI

***Open-Source Component Portability Infrastructure*** (OpenCPI) is a development and runtime framework that embraces:

- Open source software and gateware
- Component-based development (CBD)
- Heterogeneous embedded computing

The following diagram shows what OpenCPI considers to be a heterogeneous embedded system, with multiple different processor types (yellow) are connected by multiple different interconnect technologies (green).

In the following figure, two applications (blue and brown) are deployed with their constituent components distributed to the available processors.

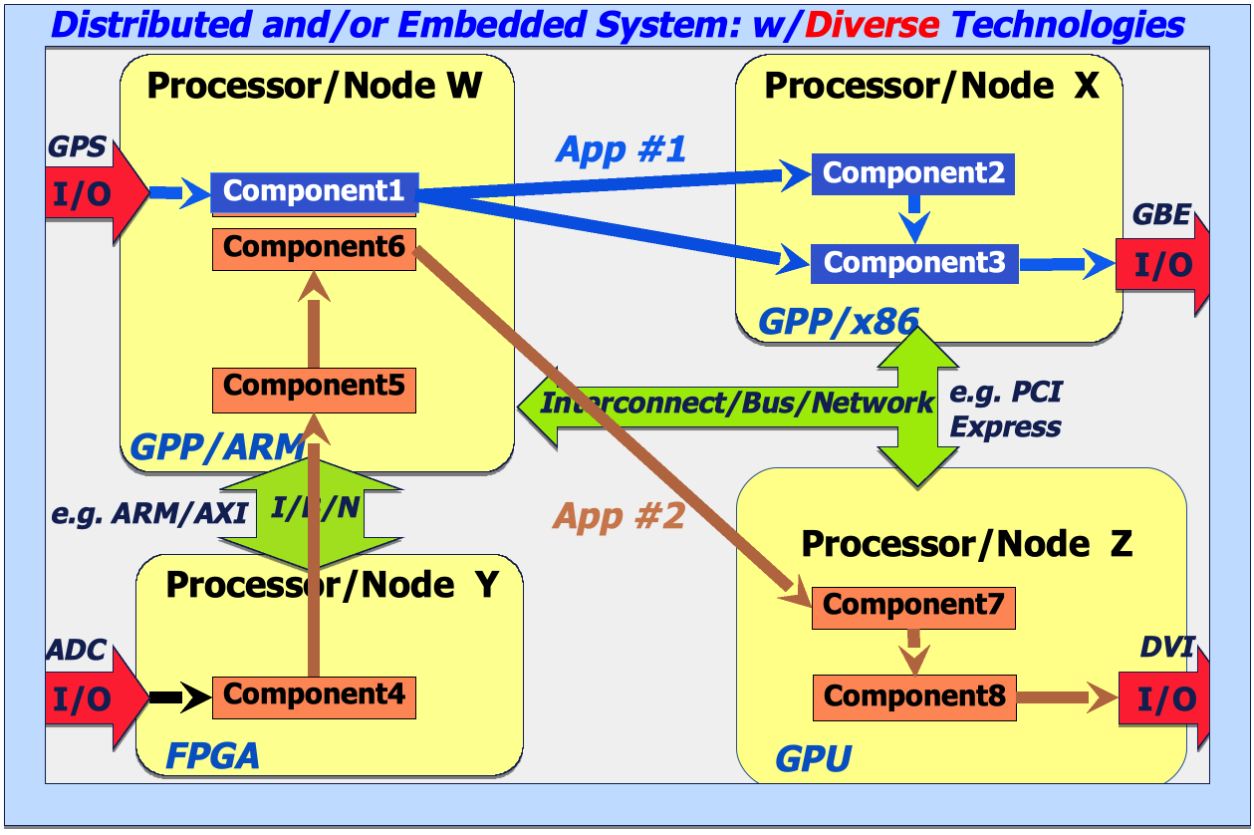


*Figure 1: Embedded System with Diverse Processing and Interconnect Technologies*

The most common type of embedded system that OpenCPI is used with is Software Defined Radio (SDR), but any system with diverse technologies is appropriate. SDRs commonly contain a mix of general purpose processors (GPPs) and FPGAs, both of which perform computing tasks for SDR applications.

OpenCPI has also been used with small clusters of processors in a rack packaged for an embedded environment.

The OpenCPI framework includes:

- A runtime environment for executing components and moving data, including a set of runtime libraries (software and FPGA IP code), command line utilities and “drivers” for different platforms, devices and interconnects
- A set of tools, specifications, documentation, examples and tests for developing components and applications that use them
- A framework and methodology for component-based development mixing processor types and diverse interconnect technologies and targeting embedded applications

### *2.1 Framework Architecture*

At the highest level, the architecture of the framework combines three key concepts:

- Application management model: How component-based applications are managed and controlled, including loading, launching, starting, stopping, configuring, querying, etc. This is sometimes described as how component-based applications are *deployed*. The OpenCPI framework provides a command line tool and a native C++ API for application deployment.
- Authoring model: How components are written in order to be effective on various processing technologies and execution environments. The OpenCPI framework defines authoring model APIs that allow development in languages that target a particular processing technology.
- Data transport model: How messages are moved between one component and another. The OpenCPI framework provides a flexible and transparent methodology for moving data of differing types, enabling a simplified and abstracted development process that greatly removes hardware considerations from the application.

The core software and gateware (FPGA code) provided by OpenCPI is structured to allow extensions in any of these three dimensions: enable new management models, add new authoring models, and adding new data transport technologies.

## *2.2 OpenCPI Concepts and Terminology*

This section provides brief descriptions of the concepts and terms that relate to using OpenCPI that you will encounter in OpenCPI tutorials, guides and references. OpenCPI briefings provide more introductory context for these definitions, while reference documents provide the details.

### *2.2.1 Design Concepts*

A ***component*** is a defined function that has properties and ports. ***Properties*** configure and control the component. ***Ports*** are where data messages are sent and received. Ports support ***protocols***, which define the allowed/expected messages. A component is specified in a small XML document called a ***component spec*** (acronym ***OCS***). A component spec defines its properties and ports and may refer to ***protocol specs*** (acronym ***OPS***) that are common to different components. Protocol specs are also defined in XML. The ***OpenCPI Component Development Guide*** provides details.

An ***application*** is an ***assembly*** of configured and connected components. Each component in the assembly indicates that, at runtime, some implementation – a worker binary – must be found for this component and an executing runtime instance must be created from this binary. Each component in the assembly can be assigned initial (startup) property values. Some component ports are identified as external ports of the assembly and can be bound to files or other devices when the application is run. An application is specified is an XML document (possibly program-generated) called an ***application spec*** (acronym ***OAS***). The application spec is used to launch the application. The ***OpenCPI Application Development Guide*** provides details.

A ***worker*** is a specific implementation of a component with its source code written according to an ***authoring model***. A component can have many implementations/workers. Workers are written in different languages to OpenCPI-defined APIs and are compiled/synthesized for different targets (CPUs, FPGAs etc.). All workers based on the same component spec perform the same function, perhaps using different algorithms to do the same job. A worker is developed in its own directory, which is usually a subdirectory of a ***component library***, which is a collection of component specifications and workers that can be built, exported and installed to support applications. A worker has an associated ***worker description XML*** (acronym OWD) and source code. The ***OpenCPI Component Development Guide*** provides details.

An ***authoring model*** is one of several ways to write a worker, usually for languages well-suited to some processing technology. An authoring model defines how to create component implementations in a specific language using a specific API between the component and its execution environment. Existing authoring models are:

- ***Resource-Constrained C Language*** (RCC): the authoring model used by C or C++ language workers that execute on general-purpose processors (GPPs). The term "resource constrained" indicates the limited set of library calls a worker should use. The ***OpenCPI RCC Development Guide*** provides details.

- ***Hardware Definition Language*** (HDL): the authoring model used by workers that execute on FPGAs, usually written in VHDL. The ***OpenCPI HDL Development Guide*** provides details.
- OpenCL (OCL): the authoring model used by OpenCL language workers targeting graphics processors. The ***OpenCPI OCL Development Guide*** provides details.

### *2.2.2 Runtime Concepts and Tools*

***Artifacts*** are binary executables compiled from one or more workers found in heterogenous ***artifact libraries***. An artifact is the result of building/compiling worker source code and includes embedded metadata needed for runtime discovery. An artifact is a physical unit that can be patched, updated, emailed, like a .dll or .so file for software and a "bitstream" file for FPGAs. An ***artifact library*** is a collection of artifacts for runtime. Artifact libraries follow the typical "library" concept: the runtime has a typical "library path" to find artifacts from libraries. The ***OpenCPI Application Development Guide*** provides details.

***Containers*** are runtime environments for executing workers. A container manages the execution of workers on a processor and arranges control and data communications with other containers, usually on other processors. At application runtime, for each component in the assembly, the system finds a binary artifact containing an implementation (compiled worker) and finds a runtime environment (container) where the worker can run. Artifacts are loaded on demand into the containers to run the XML assembly. Application execution requires an assembly, artifact libraries and containers.

Different developer types contribute to what's needed for application execution:

- ***Application developers*** create applications that specify components and use artifacts to execute on containers. The ***OpenCPI Application Development Guide*** describes these tasks.
- ***Component developers*** create components, workers and artifacts for use by application developers. The ***OpenCPI Component Development Guide*** describes these tasks.
- ***Platform developers*** create the lower level drivers and configurations that support the containers (runtime environments on processors) in a system. The ***OpenCPI Platform Development Guide*** describes these tasks.

The OpenCPI framework provides two methods for deploying an application:

- The OpenCPI tool **`ocpirun`**, which takes an application XML and various control options to execute it.
- A native C++ API for controlling and launching applications called the OpenCPI Application Control Interface (ACI), which allows for greater control, including reading and writing properties during execution.

### *2.2.3 Development Environment Concepts and Tools*

An ***asset*** is a general term for anything developed for OpenCPI by any developer type. Asset types include applications, components, workers, artifacts, protocols and so on. Artifacts, however, are not assets, but are the result of building some assets.

Assets are developed in ***projects*** and are defined by an XML file. For those who are unfamiliar with XML, it stands for eXtensible Markup Language and is a structured type of text file that contains a hierarchy of elements. The file is an element, and that element can have child elements; each element has a type identifier and named attributes with values. For more about XML, see: https://en.wikipedia.org/wiki/XML.

Some asset types have their own directory, so the XML file lives there. Other files for the asset also live there; for example, source code.

Developer operations on assets include: create, destroy, edit, build, clean, and run (for unit tests and applications).

A ***project*** is a workspace in a directory where assets are developed. One project can contain all types of assets and one project can contain multiple assets of any type. Projects exist in the development environment; they do not exist in a runtime-only (deployed) environment. The work product of developers is projects containing assets.

Projects have dependencies on assets in other projects: you can create an application in your project that depends on components and workers in someone else's project. Projects can be declared to depend on other projects.

A ***project registry*** is a directory that contains references to projects in an OpenCPI development environment. OpenCPI provides a default project registry existing at a default location. When you register a project, it can be referenced and searched by other projects that are using the same project registry.

All developer types use the **`ocpidev`** command-line tool for most development tasks. The syntax is:

```
ocpidev [<options>...] <verb> [<noun> [<arg> ...]]
```

For example:

```
ocpidev create application myapp
```

Verbs are: create, delete, build, clean, run and show. Nouns are asset types like protocol, worker, component, application, etc. The command **`ocpidev --help`** displays the complete list of verbs and nouns; man pages describing **`ocpidev`** verbs and nouns are also available.

Developers using the command-line method are expected to edit XML files and source code with a text editor.

OpenCPI also provides the OpenCPI GUI plugin for OpenCPI development. The plugin provides a user interface on top of **`ocpidev`** and uses it to accomplish most actions. The ***OpenCPI GUI User Guide*** provides details. Developers must still use a text editor to edit XML files and source code.

# 3 OpenCPI Documentation

The ***OpenCPI Installation Guide*** is intended to be used to establish an installation ready for users. It is designed to allow installation without much learning of OpenCPI concepts, and thus this guide is not a prerequisite for that document. A user may need to refer to that document when installation requirements change, but it should not be needed for day-to-day use of OpenCPI. After using the installation guide, the entire installation maybe copied to other machines without redoing the installation process.

All documents are referenced at opencpi.gitlab.io.

There are reference manuals for complete descriptions of all supported features and concepts in the different development areas of OpenCPI, namely:

- Application development
- Component development (in general, regardless of authoring model)
- RCC development (C++ workers in libraries)
- HDL development (FPGA development)
- Platform development (OSP and device support development)

There is also an ***OpenCPI Glossary*** that provides definitions for OpenCPI terms and industry terms used in OpenCPI. The glossary is available both as a stand-alone document and as the last chapter of each reference document (including this one).

If a platform has installation issues beyond the generic installation for its platform type described in the ***OpenCPI Installation Guide***, it has its own ***Getting Started Guide*** that describes these platform-specific details.

All assets that can be created in an OpenCPI project with the `ocpidev` command-line tool can have their own documents (sometimes called ***datasheets*** for component specs and workers). The ***OpenCPI Documentation Writer Guide*** provides content and format guidelines for creating these documents using the Sphinx-based asset documentation system that is integrated with `ocpidev`.

On the next page is a graphical diagram showing OpenCPI documentation and how users are expected to use it:

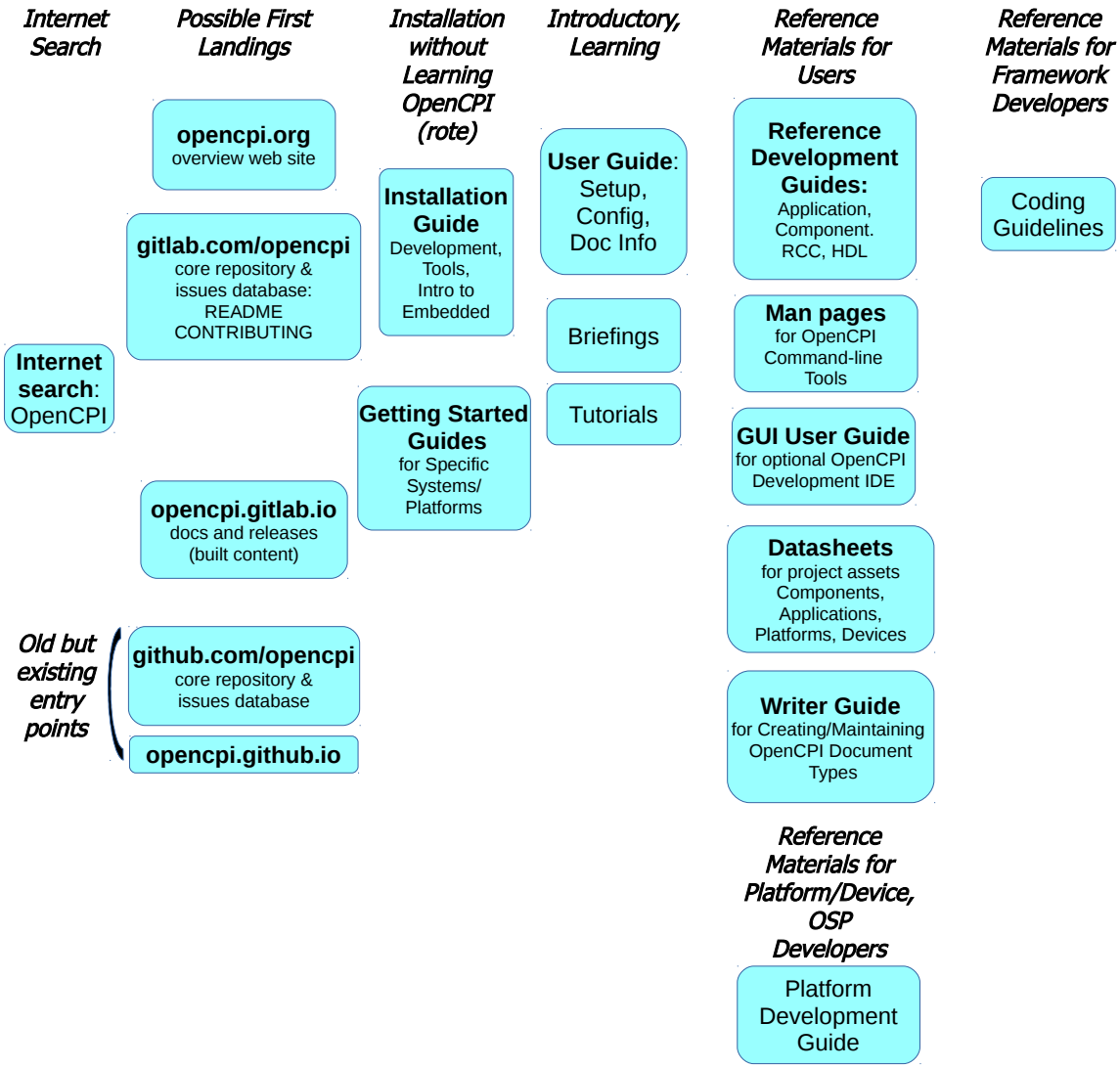


*Figure 2: Documentation for OpenCPI*

# 4 Using OpenCPI's Built-in Projects

OpenCPI comes with a set of built-in projects that contain the assets and artifacts provided by the OpenCPI framework for asset development, as shown in the following table.

*Table 1: OpenCPI Built-in Projects*

| Name in projects/ directory | Project Package-ID | Contents | How Used |
|---|---|---|---|
| `core` | `ocpi.core` | Basic components, primitives, RCC platforms, FPGA simulator platforms. Minimum needed to execute some unit tests. | User builds but does not modify this project. |
| `platform` | `ocpi.platform` | Platform support assets (cards, devices, primitives, specs), as well as reference platforms such as `zed` and `zcu104`. | Platform developers use these assets to build OSPs (see below). |
| `assets` | `ocpi.assets` | Example applications, components, primitives, some legacy platforms and devices, assemblies. | User builds, runs, maybe modifies. |
| `assets_ts` | `ocpi.assets_ts` | Some experimental assets to demonstrate time-stamping | Used as example components. |
| `tutorial` | `ocpi.tutorial` | Example applications, components, worker source code, and test files | Used in OpenCPI tutorials. |

The project package-ID shown in the table is a globally unique identifier for the project asset. (All OpenCPI assets have package-IDs, but not all assets are currently used explicitly.) The **ocpi** prefix in the package-ID indicates that the package is managed by the OpenCPI maintainers and exists in the OpenCPI Gitlab repository. The project package-ID is used when the project is depended on by other projects. All projects depend on the core built-in project. Its package-ID is ***`ocpi.core`***. You can find out more about package-IDs in the ***OpenCPI Component Development Guide***.

When users create projects, they can be in any directory, but generally use the default registry provided by OpenCPI and are easily able to depend on the built-in projects.

When a project contains code to enable a particular platform to be used by OpenCPI, it is called an OSP (OpenCPI System support Project). Some platforms are directly supported by the built-in projects. OpenCPI also provides other OSPs that are available from the OpenCPI gitlab.com site at https://gitlab.com/opencpi/osp. To install a platform based on one of these OSPs (or any other), use the procedure described in the ***OpenCPI Installation Guide***.

# 5 Using OpenCPI Environment Variables

Various environment variables are used to control OpenCPI. All start with the prefix **OCPI_**. Nearly all are optional. During installation, some are set when external tools are installed in non-default locations. Most are only used during development, and a few are used when applications run. Some of these variables are useful for debugging purposes. The OpenCPI environment is established when the user sources the **opencpi-setup.sh** script, in its installed location, which is the **cdk** subdirectory of the source installation (git repo).

When the setup script is sourced, the **OCPI_CDK_DIR** environment variable (which points to the OpenCPI installation) is set to the directory where the script lives; for example, **~/opencpi/cdk**.

The setup command is:

```
source <location-of-opencpi>/cdk/opencpi-setup.sh -s
```

So, if OpenCPI is installed at **~/opencpi**, the command is:

```
source ~/opencpi/cdk/opencpi-setup.sh -s
```

Sourcing this script with no arguments (or with the **-h** or **--help** option), will display more options for special cases. If you prefer the OpenCPI environment to be set up automatically at login, place this command in the shell (bash) login startup file (usually **~/.profile**) after the export PATH line. Note that this will only take effect when you log in or when you start a new "login shell" using the **-l** (ell) option to bash, like:

```
bash -l
```

Any persistent site-specific environment variable settings are placed in the **user-env.sh** file in the installation directory (**$OCPI_ROOT_DIR**). The file is created automatically during installation.

OpenCPI currently only supports the **bash** shell.

The first table below lists variables only relevant to the execution of applications, whether in a development environment or a deployed runtime environment. The second table is for variables only used in a development environment.

*Table 2: OpenCPI Environment Variables for Runtime Configuration*

| Name | Description |
|---|---|
| **OCPI_CDK_DIR** | This variable set by OpenCPI automatically and indicates the location of the OpenCPI development or runtime installation. *This variable should not be set directly. Even though it has CDK in its name, it is indeed used at runtime to locate the OpenCPI runtime installation.* |
| **OCPI_LIBRARY_PATH** | A colon-separated set of directories to be searched for runtime artifacts. When referencing the artifacts exported by a project with sources, be sure to reference the project's **exports** subdirectory, not its source location. When running unit tests or applications in projects, this variable is set automatically if not set already. In pure runtime environments, it may need to be set when artifacts are placed in directories of the user's choice. |
| **OCPI_LOG_LEVEL** | The log level (amount of logging) output by the runtime system. The default is zero (**0**), indicating no logging output. The maximum is **20**. Commonly useful startup and diagnostic information (for example, artifact discovery feedback) is provided at log level **8**. Unusual events are logged at level **4**. Logging is to **stderr**. |
| **OCPI_SYSTEM_CONFIG** | The runtime system XML system configuration file. If not specified, the file is located trying these locations in order:<br>**$OCPI_ROOT_DIR/system.xml**<br>**$OCPI_CDK_DIR/default-system.xml** |
| **OCPI_DMA_MEMORY** | Allows super-user privileged processes to specify physical DMA memory to use when the OpenCPI kernel driver is not used. The format is ***<mbytes>*M$0x*<address>***, with ***mbytes*** in decimal, and ***address*** in hexadecimal. Rarely used. |
| **OCPI_SMB_SIZE** | The size (in bytes, decimal, with optional K or M suffix) of the memory pool to be used for buffers in software containers. Must support buffers for all ports used to communicate into or out of the container. |

*Table 3: OpenCPI Environment Variables for Development Configuration*

| Name | Description |
|---|---|
| **OCPI_PROJECT_PATH** | A colon-separated set of project directories to be considered *in addition* to those that are registered. This variable can be used to temporarily augment the project registry when testing a new project. Rarely used. |
| **OCPI_PROJECT_REGISTRY_DIR** | The location of the current project registry when dealing with unregistered projects or **ocpidev show** commands. By default, this location is $**OCPI_ROOT_DIR/project-registry**, but setting this variable overrides this. |

All of these variables are optional. If you want permanent settings made when using a given OpenCPI installation, settings (**export** commands) may be placed in the **user-env.sh** file in the installation.

# 6 Building Projects for OpenCPI Platforms

The OpenCPI command:

**`ocpiadmin install platform <`*`platform`*`> [<`*`options`*`>]`**

when run in the top level directory of OpenCPI, will perform the correct build steps for a new platform. It will even download the required OSP if necessary. It will build the built-in projects as well as the OSP for the platform, which is appropriate if you are using this platform but not actually developing the support for it in an OSP; in this case, it is "installing a platform that someone else developed" and thus is more of an installation step than a development task. The command is described in the ***OpenCPI Installation Guide*** and in the **`ocpiadmin(1)`** man page.

Below is a list of supported development system types, with the corresponding OpenCPI ***software platform*** and URLs for related information and download. All supported development platforms are in the **`ocpi.core`** project built in to OpenCPI. Other unsupported or not-yet-supported development system types can be in external projects. The default supported development platform is **`centos7`**.

*Table of Supported Development Systems (Hosts)*

| Description | OpenCPI Software Platform | Limitations | Vendor Links |
|---|---|---|---|
| CentOS7 Linux on x86-64 bit | **`centos7`** | | CentOS7 general information link<br>CentOS7 download link |
| Rocky 8 Linux on x86-64 bit | **`rocky8`** | | Rocky 8 general information link<br>Rocky 8 download link |
| Ubuntu 18.04 Linux on x86-64 bit | **`ubuntu18_04`** | | Ubuntu 18.04 LTS general information link<br>Ubuntu 18.04 LTS download link |
| Ubuntu 20.04 Linux on x86-64 bit | **`ubuntu20_04`** | | Ubuntu 20.04 LTS general information link<br>Ubuntu 20.04 LTS download link |
| MacOS Catalina 10.15 | **`macos10_15`** | No FPGA tools or drivers | MacOS Catalina general information link<br>MacOS Catalina download link |

Below is the list of supported FPGA simulators and the corresponding OpenCPI ***"hardware" (HDL) platforms***. When the associated tools are installed on a development system, the development system then includes these platforms in addition to the software platform. These simulators enable FPGA development without actual FPGA hardware. The default and recommended FPGA simulator is **`xsim`**, which is part of the free (WebPACK) version of the Xilinx Vivado tools package.

*Table of Supported FPGA Simulators*

| Description | OpenCPI Platform | Vendor Links |
|---|---|---|
| Xilinx Vivado XSIM 2017.1-2021.2 | **`xsim`** | Vivado® Simulator (XSIM) is part of all Vivado HL Editions, including the free WebPack edition.<br>Vivado Simulator general information link<br>Vivado Simulator download link |
| Xilinx ISE ISim 14.7 | **`isim`** | ISE® Simulator (ISim) is part of the older Xilinx ISE tool set.<br>ISE/ISim general information link<br>ISE/ISim download link |
| Siemens/Mentor Modelsim 10.4c-10.6c | **`modelsim`** | Modelsim® is powerful, fast and expensive.<br>Modelsim general information link<br>No download link available before purchase. |

On the next page is a table of supported embedded systems, which includes the platforms each system consists of. Embedded systems generally have a software processor (CPU) using the "primary" OpenCPI software platform, along with other, usually hardware/FPGA, platforms. A physical processor (CPU) or processing device (FPGA) may be supported by a variety of OpenCPI platforms (versions or OSes).

*Table of Supported Embedded Systems*

| Description | OSP Repository | Primary Software Platform(s) | Other Platforms in the system | Vendor Links |
|---|---|---|---|---|
| Epiq Solutions Matchstiq-Z1 radio | Part of OpenCPI | `xilinx13_3` | `matchstiq_z1` | epiqsolutions.com/matchstiq |
| Avnet® ZedBoard | Part of OpenCPI | `xilinx19_2_aarch32` | Using Vivado: `zed` Using ISE: `zed_ise` | zedboard.org |
| Ettus E310 USRP™ | gitlab.com/opencpi/osp/ocpi.osp.e3xx | `xilinx19_2_aarch32` | `e31x` | www.ettus.com/all-products/e310 |
| Analog Devices™ ADALM-PLUTO | gitlab.com/opencpi/osp/ocpi.osp.pluto | `adi_plutosdr0_32` | `plutosdr` | wiki.analog.com/university/tools/pluto |
| Analog Devices ADRV9361-Z7035 | gitlab.com/opencpi/osp/ocpi.osp.analog | `xilinx19_2_aarch32` | `adrv9361` | www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/adrv9361-z7035 |
| Avnet MicroZed™ | gitlab.com/opencpi/osp/ocpi.osp.avnet | `xilinx19_2_aarch32` | `microzed_10_cc` `microzed_20_cc` | microzed.org |
| Avnet PicoZed™ | gitlab.com/opencpi/osp/ocpi.osp.avnet | `xilinx19_2_aarch32` | `picozed_10_cc` `picozed_15_cc` `picozed_20_cc` `picozed_30_cc` | picozed.org |
| Xilinx® Zynq UltraScale+™ ZCU102 | gitlab.com/opencpi/osp/ocpi.osp.xilinx | `xilinx19_2_aarch64` | `zcu102` | www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html |
| Xilinx Zynq UltraScale+ ZCU111 | gitlab.com/opencpi/osp/ocpi.osp.xilinx | `xilinx19_2_aarch64` | `zcu111` | www.xilinx.com/products/boards-and-kits/ek-u1-zcu111-g.html |

The following table lists all supported platforms (software and hardware) in the current release. Every system running OpenCPI contains one or more platforms that support

different components of an application running on different platforms in the system. Some platforms may be optionally attached/inserted/plugged into a system, such as PCI-Express plug-in cards with FPGA or CPUs on them.

Each entry in the table specifies the platform name, the OpenCPI project that supports it, and any vendor tool installations that the platform requires. The chapter “Installing Third-party/Vendor Tools” in the ***OpenCPI Installation Guide*** provides download and installation information for most of these vendor tools.

*Table of Supported Platforms*

| **OpenCPI Platform Name** | **Description** | **OpenCPI Project/Repo** | **Dependencies required *before* building/installing the platform** |
|---|---|---|---|
| ***Development Host Platforms*** | | | |
| **`centos7`** | CentOS7 on x86_64 CPUs | **`ocpi.core`** built-in | A CentOS7 installation |
| **`rocky8`** | Rocky 8 on x86_64 CPUs | **`ocpi.core`** built-in | A Rocky 8 installation |
| **`ubuntu18_04`** | Ubuntu 18.04 on x86_64 CPUs | **`ocpi.core`** built-in | An Ubuntu 18.04 installation |
| **`ubuntu20_04`** | Ubuntu 20.04 on x86_64 CPUs | **`ocpi.core`** built-in | An Ubuntu 20.04 installation |
| **`macos10_15`** | MacOS Catalina | **`ocpi.core`** built-in | A MacOS Catalina installation |
| ***Embedded Software Platforms*** | | | |
| **`xilinx13_3`** | Xilinx Linux 14.7 from 2013 Q3 | **`ocpi.core`** built-in | ISE® EDK 14.7 or Vivado® SDK 2013.4<br>Xilinx Linux tag: **`xilinx-v14.7`** |
| **`xilinx13_4`** | Xilinx Linux 14.7 from 2013 Q4 | **`ocpi.core`** built-in | ISE® EDK 14.7 or Vivado® SDK 2013.4<br>Xilinx Linux tag: **`xilinx-v2013.4`** |
| **`xilinx19_2_aarch32`** | Xilinx Linux from 2019Q2 (2019.2) For Zynq-7000 | **`ocpi.core`** built-in | Xilinx Vitis™ SDK 2019.2<br>Xilinx Binary Zynq Release 2019.2<br>Xilinx Linux git clone<br>Xilinx Linux tag: **`xilinx_v2019.2`** |
| **`xilinx19_2_aarch64`** | Xilinx Linux from 2019Q2 (2019.2) For Zynq-Ultra | **`ocpi.core`** built-in | Xilinx Vitis SDK 2019.2<br>Xilinx Binary Release 2019.2<br>Xilinx Linux git clone<br>Xilinx Linux tag: **`xilinx_v2019.2`** |
| **`xilinx21_2_aarch32`** | Xilinx Linux from 2021Q2 (2021.2) For Zynq-7000 | **`ocpi.core`** built-in | Xilinx Vitis SDK 2021.2<br>Xilinx Binary Zynq Release 2021.2<br>Xilinx Linux git clone<br>Xilinx Linux tag: **`xilinx_v2021.2`** |
| **`xilinx21_2_aarch64`** | Xilinx Linux from 2021Q2 (2021.2) For Zynq-Ultra | **`ocpi.core`** built-in | Xilinx Vitis SDK 2021.2<br>Xilinx Binary Release 2021.2<br>Xilinx Linux git clone<br>Xilinx Linux tag: **`xilinx_v2021.2`** |
| **`adi_plutosdr0_32`** | Analog Devices Pluto Linux 0.32 | **`ocpi.osp.plutosdr`** | Vivado SDK 2019.2 |

| OpenCPI Platform Name | Description | OpenCPI Project/Repo | Dependencies required *before* building/installing the platform |
|---|---|---|---|
| | | ***FPGA/HDL Platforms*** | |
| **zed** | ZedBoard Zynq FPGA w/Vivado | **ocpi.platform** built-in | Vivado WebPACK |
| **zed_ise** | ZedBoard Zynq FPGA w/ISE | **ocpi.platform** built-in | ISE 14.7 WebPACK |
| **zed_ether** | ZedBoard Zynq FPGA/PL w/dual 1 GbE | **ocpi.platform** built-in | Vivado WebPACK |
| **matchstiq_z1** | Epic Solutions Zynq FPGA/PL | **ocpi.assets** built-in | Vivado WebPACK |
| **e31x** | Ettus E31x Zynq FPGA/PL | **ocpi.osp.e3xx** | Vivado WebPACK |
| **ml605** | Xilinx PCI-Express card w/Virtex6 | **ocpi.assets** built-in | ISE 14.7 |
| **alst4, alst4x** | Altera PCI-Express card w/Stratix4 gx230, gx530 | **ocpi.assets** built-in | Quartus12.1+<br>gx230: User Guide, Ref Manual<br>gx530: User Guide, Ref Manual |
| **plutosdr** | Analog Devices ADALM-PLUTO | **ocpi.osp. plutosdr** | Vivado WebPACK |
| **zcu104** | Xilinx Zynq-UltraScale+ Dev Bd | **ocpi.platform** built-in | Vivado WebPACK |
| **zcu106** | Xilinx Zynq-UltraScale+ Dev Bd | **ocpi.platform** built-in | Vivado WebPACK |
| **picoevb** | RHS Research PicoEVB | **ocpi.platform** built-in | Vivado WebPACK |
| **adrv9361** | Analog ADRV9361 Zynq FPGA/PL | **ocpi.osp. analog** | Vivado licensed |
| **microzed_10_cc**<br>**microzed_20_cc** | Avnet MicroZed Zynq FPGA/PL | **ocpi.osp.avnet** | Vivado WebPACK |
| **picozed_10_cc**<br>**picozed_15_cc**<br>**picozed_20_cc**<br>**picozed_30_cc** | Avnet PicoZed Zynq FPGA/PL | **ocpi.osp.avnet** | Vivado WebPACK |
| **zcu102** | Xilinx Zynq UltraScale+ MPSoC | **ocpi.osp.xilinx** | Vivado licensed |
| **zcu111** | Xilinx Zynq UltraScale+ RFSoC | **ocpi.osp.xilinx** | Vivado licensed |

| OpenCPI Platform Name | Description | OpenCPI Project/Repo | Dependencies required *before* building/installing the platform |
|---|---|---|---|
| `x310` | Ettus X310 Xilinx Kintex-7 FPGA w/dual 1/10 GbE | `ocpi.osp.ettus` | Vivado licensed |

Some of the platforms listed in the table above require system-specific setup procedures to be performed in addition to the general procedures described in the ***OpenCPI Installation Guide***. The following table lists these systems and provides links to the corresponding setup documents for these systems.

*Table of Platforms with System-Specific Setup Requirements*

| OpenCPI Platform Name | OpenCPI System-Specific Setup Guide |
|---|---|
| `adrv9361` | ***Analog Devices ADRV9361-Z7035 Getting Started Guide*** |
| `alst4, alst4x` | ***Alst4 Getting Started Guide*** |
| `e31x` | ***Ettus E31x Getting Started Guide*** |
| `matchstiq_z1` | ***Matchstiq Z1 Getting Started Guide*** |
| `microzed_10_cc`<br>`microzed_20_cc` | ***Avnet MicroZed 7Z010 Getting Started Guide***<br>***Avnet MicroZed 7Z020 Getting Started Guide*** |
| `ml605` | ***ML605 Getting Started Guide*** |
| `picoevb` | ***RHS Research PicoEVB Getting Started Guide*** |
| `picozed_10_cc`<br>`picozed_15_cc`<br>`picozed_20_cc`<br>`picozed_30_cc` | ***Avnet PicoZed 7Z010 Getting Started Guide***<br>***Avnet PicoZed 7Z015 Getting Started Guide***<br>***Avnet PicoZed 7Z020 Getting Started Guide***<br>***Avnet PicoZed 7Z030 Getting Started Guide*** |
| `plutosdr` | ***PlutoSDR Getting Started Guide*** |
| `x310` | ***Ettus X310 Getting Started Guide*** |
| `zcu102` | ***Xilinx ZCU102 Getting Started Guide*** |
| `zcu111` | ***Xilinx ZCU111 Getting Started Guide*** |
| `zed, zed_ise` | ***ZedBoard Getting Started Guide*** |
| `zed_ether` | ***ZedBoard Ethernet Getting Started Guide*** |

# 7 The OpenCPI System Configuration File

The OpenCPI system configuration file is an XML file that allows settings to be specified that affect the runtime system. Its normal location is:

**$OCPI_ROOT_DIR/system.xml**

but if it is not present (that is, not customized for the system), the default file is used, which is:

**$OCPI_CDK_DIR/default-system.xml**

Its location may also be overridden by the **OCPI_SYSTEM_CONFIG** environment variable. An example file is:

```
<opencpi>
  <container>
    <ocl load='0'/>
    <rcc load='1'/>
    <hdl load='1'>
      <device name="PCI:0000:05:00.0" esn="000013C1E1F401"/>
      <device name="PCI:0000:04:00.0" esn="91d28408"/>
    </hdl>
    <remote load='1'/>
  </container>
  <transfer smbsize='128K'>
    <pio load='1'/>
    <!-- <dma load='1'/> -->
    <socket load='1'/>r
  </transfer>
</opencpi>
```

The top-level elements are categories of modules (plugins) and the next level is individual plugins. The two top-level categories are:

- ***container:*** for the plugins that support different types of runtime environments
- ***transfer:*** for the plugins that support different data transport mechanisms.

Each plugin has a boolean **load** attribute that specifies whether the module should be loaded (and thus enabled) in the runtime environment. Otherwise all attributes are plugin-specific. These OpenCPI plugins are like device drivers which know how to discover and manage certain devices.

At installation time, this file is normally only modified for specific processing device attributes. Note that in the container plugin category, the term (and XML element) **<device>** is in fact the processor that supports the OpenCPI runtime environment, which is an instance of an OpenCPI ***platform***.

### *7.1 Configuration Elements for Container Plugins*

The **`<device>`** child element can appear under any of the **`container`** plugin elements and specifies attributes specific to individual processing devices (of some platform type) supporting that container type. In the example above, an **`esn`** attribute is applied to two HDL devices and indicates the electronic serial numbers of the JTAG cables attached to specific PCIe-based FPGAs used as OpenCPI processors (when they are present in the system).

These **`<device>`** elements can contain **`<instance>`** child elements that can provide device-specific default property values when certain workers are used on that device in applications or loaded artifacts (e.g. FPGA bitstreams). To understand how and when these property values get applied to the underlying workers, two concepts must be introduced: 1) how artifacts are used for execution, and 2) the particular case of ***load-time*** workers. These two concepts are introduced in the next sections.

#### *7.1.1 How Artifacts are Used when Applications are Executed*

When an application is run, for each ***instance*** in the application the following three *deployment decisions* are made:

1. Which ***container*** (and underlying processing device) the instance will run on.
2. Which ***artifact*** will be used/loaded on that container to support the instance.
3. Which built ***worker*** in that artifact will be used (when there is more than one built worker in the artifact).

To prepare for application execution, the artifact chosen (in decision #2) is loaded onto the container (like a shared object/library is loaded onto a CPU, or an FPGA bitstream is loaded onto an FPGA). An artifact may contain multiple compiled/built workers, so choice #3 is needed when there are multiple compiled/built workers in the chosen artifact that all implement the component spec indicated in the application **`<instance>`** (using the **`component`** attribute).

We call these compiled/built workers ***artifact workers***. Note that there may be artifact workers in the artifact that are not used by the application at all. They may exist in the artifact, but are not used or mentioned by the application (XML).

The artifact workers in an artifact may be ***reentrant***, which means that the artifact worker can support multiple application instances at the same time. This is normal and common in software artifacts (shared objects/libraries). I.e. if the application has two instances using the component **`file_read`**, which are both running in a software/RCC container, that read two different files at the same time, both of those instances will use the same artifact worker in the **`file_read`** artifact.

When the application is started, based on the deployment decisions above and after the chosen artifacts are loaded, ***runtime workers*** are created in the container based on the chosen artifact worker in the loaded chosen artifact. At that time, the runtime worker is initialized and the property settings indicated in the application are applied to that

runtime worker, all prior to the application being started. So the artifact worker is used to create a runtime worker to implement the instance in the application.

Remember the distinction between the artifact worker in the artifact (a chunk of binary code), and the runtime worker that is a runtime object that runs the code in/from the artifact worker, in a container. The term ***worker*** is used three ways: 1) the source code, 2) the compiled/built worker code in an artifact, and 3) the runtime object that "does work" executing in a container.

Any property settings are made on the runtime worker and do not affect the binary code of the artifact worker in the artifact. Hence the artifact worker can be used for more than one runtime worker (application instance) at the same time (if it is reentrant), and can also be serially reused in later applications for new workers in those new applications (whether it is reentrant or not), without the artifact being reloaded.

At this time, artifact workers in HDL/FPGA artifacts are never reentrant.

### *7.1.2 Load-time Workers*

Normally, runtime workers created for applications have a lifecycle that is the same as the application they are part of. They are created (from artifact workers), initialized, configured, and started when the application starts and are released and destroyed when the application ends.

Artifact workers may be labeled as having a different lifecycle, where runtime workers are created when the artifact is *loaded*, independent of applications, and destroyed when (just before) the artifact is *unloaded*. Such workers are called ***load-time*** workers since they are created/initialized/configured/started when the artifact is *loaded*, which is usually when the artifact is first needed by some application. They have a lifecycle that persists across multiple applications being executed that re-use the same loaded artifact. They are created whether or not they are mentioned in any application. Workers that are slaves to proxies mentioned in the application are considered “mentioned in the application”.

Artifact workers are labeled as being load-time by setting the **`loadtime`** attribute to true in either the worker's OWD or when a device is declared as part of a platform (or card) in the XML file that defines the platform or card (in the **`<device>`** elements there). So a platform developer may designate a device worker to be load-time for that platform (or card) even if it is not declared as being load-time in its OWD.

In HDL artifacts, the ***platform worker*** and the worker that keeps time are always considered to be load-time workers.

If a load-time worker is specified in the application, only ***writable*** properties are allowed to be specified in the application, since the load-time worker is started before the application is started.

### *7.1.3 Providing Default Values for Worker Properties in **`system.xml`***

One of the things that can be specified in the **`system.xml`** file for a container are default property settings for workers that run in that container. These property settings are applied to runtime workers when they are created, before any property settings

specified by the application are applied. If the indicated worker is a load-time worker, these properties are applied when the worker is created at the time the artifact is loaded. Otherwise they are applied when the runtime worker is created for the application.

These property settings can thus serve as device-specific, default values. This is accomplished by inserting **<instance>** elements in the **<device>** elements under the **<container>** element in the **system.xml** file, e.g.:

```
<opencpi>
  <container>
    <hdl>
      <device name='myfpga'>
        <instance worker='foo'>
          <property name='pname' value='pvalue'/>
        </instance>
      </device>
      .. other HDL devices/containers
    </hdl>
    ... other models
  </container>
</opencpi>
```

The above example indicates: *whenever you encounter a **foo.hdl** worker in an application where its runtime worker will run on this device, apply this property setting before applying any settings in the application*. Thus the attributes of the **system.xml <instance>** elements (e.g. **worker**) are used to match against any instances in applications, after the above deployment decisions have been made, to see whether these settings in the **system.xml** apply to any of the instances (and thus runtime workers) in the application.

The **<instance>** elements in **system.xml** use a subset of the valid attributes for **<instance>** elements in application XML (OAS). Note the authoring model suffix for worker names here is implicit and thus is not needed (although it is accepted).

The **worker** attribute thus may have a fully qualified worker name (including package prefix and model) or just the worker name, the worker name and model suffix and may or may not use a build configuration ID (after a hyphen, before the model suffix).

At this time, property values for parameter property values are not allowed in **system.xml** and are thus not used in this matching process. The property values in **system.xml** are thus only used for initial or writable properties.

The **system.xml <instance>** elements can unambiguously specify ***device workers*** for particular devices in the device's platform, using the **device** attribute (which is also valid in **<instance>** elements in applications). Its value is matched against the artifact worker.

Properties set this way must either be ***initial*** properties or ***writable*** properties.

Specifying load-time worker properties in **system.xml** for a processing device is typically used to specify static processing-device-specific properties for device workers

on the platform. Such properties are typically *not* specified in portable applications. Here is an example where a load-time worker, which corresponds to a particular device worker in the artifact, is configured:

```
<opencpi>
  <container>
    <hdl>
      <device name='myfpga'>
        <instance device='ethermac1'>
          <property name='macaddr' value='1122334455566'/>
        </instance>
      </processor>
    </hdl>
  </container>
</opencpi>
```

The above example indicates that the **macaddr** property of the device worker that supports the **ethermac1** device should be set to **1122334455566**, when that device is present and used in an artifact (bitstream). The use of the word “device” is overloaded here. Plugins manage devices whose meaning is specific to the plugin, but container plugins support setting properties on workers that represent specific lower-level devices attached to the underlying platform.

### *7.2 Configuration Elements for Transfer Plugins*

The **`smbsize`** attribute to the top-level **`transfer`** element specifies a default value for the size in bytes of the buffer memory pool for each data transfer endpoint. It can also be applied to each transfer plugin module below it, which overrides the default.

### *7.3 Discovery of Devices Managed by Plugins*

Most plugins act as drivers for a certain type of device. They manage and discover the devices they support. Container plugins manage *processing* devices that can support containers for running workers. Transfer plugins manage *interconnect* devices that can support connections *between* containers (to support connections between workers running on different containers). Other plugin types are used internal to the system.

When plugins are loaded and used, they find their devices to be managed by a *discovery* process. This involves scanning the system environment for candidate devices. The **`<device>`** elements under a plugin's XML element are used to provide extra attributes for the devices once they are discovered. These are attributes that are not automatically determined during the discovery process. I.e. they are only necessary when the plugin cannot determine them when a device is discovered. Thus no **`<device>`** element is necessary for discoverable devices that require no extra attributes.

Some devices are in fact *not* discoverable and must be explicitly described using the **`<device>`** element with attributes. These are termed ***static device***s, which are not ***discoverable devices***. When a plugin is asked to discover its devices, it first looks for **`<device>`** elements with a **`static`** attribute set to **`true`**, and tries to use them explicitly. The plugin then looks for discoverable devices (that are not already indicated as ***static***).

This discovery process (use static devices, then find discoverable ones), can be controlled by the **`discovery`** attribute of the plugin, which is an enumeration of these values:

- all – the default value indicating use of both static and discoverable devices
- none – do not do discovery for this plugin
- static – only use statically defined devices
- dynamic – only use dynamically discovered devices (avoid static ones)

# 8 The OpenCPI Linux Kernel Device Driver

OpenCPI uses a Linux kernel device driver for OpenCPI container discovery and for communication between CPUs and FPGAs on certain types of OpenCPI systems. On Linux, this device driver is a loadable kernel module. This kernel module is built against the Linux kernel headers that correspond to the running kernel it will be used with. Arranging for this build is part of platform development for the Linux RCC platform, and the build happens via the `ocpiadmin install platform` command for the platform.

The next sections describe when the driver is required, how to load it, driver configuration options, and how to allocate additional DMA memory for it when required by OpenCPI applications.

### *8.1 When the Linux Kernel Device Driver is Required*

Whether or not an OpenCPI system requires the Linux kernel device driver depends on two things:

- The platforms on the system that are enabled for use by OpenCPI during application execution
- The interconnect technology used in the system that wires the platforms together

Systems with platforms connected to a system bus, such as PCI Express or the Xilinx Zynq AXI interfaces between the CPU (PS) and the FPGA (PL) parts of the SoC, require the device driver when the bus or fabric is used for communication between the host CPU platform and another platform. Systems that do not require this type of communication do not need the driver.

### *8.2 Loading the Linux Kernel Device Driver*

When the Linux kernel device driver is required, it must be loaded. For embedded systems, including PCI Express-based systems that are set up as runtime-only systems, loading is done automatically by the scripts that establish the modes of operation for OpenCPI (server, network, or standalone) during OpenCPI installation and setup (see the ***OpenCPI Installation Guide*** for details).

If the system has not been set up with one of the modes of operation for OpenCPI (or the mode-of-operation script used to set up the system was changed to disable automatic driver loading) *and* the system is a development host, driver loading must be done explicitly using the **`ocpidriver`** tool. See the **`ocpidriver(1)`** man page for detailed usage information about the tool.

Loading the device driver with **`ocpidriver`** can be done either:

- When needed by OpenCPI applications that are to run on the target platform(s), where a user manually runs the **`ocpidriver load`** command
- When the system is booted, where the appropriate operating system startup script runs the **`ocpidriver load`** command

Loading the device driver manually when needed loads it in a generic way that is common to all Linux systems, whereas loading the device driver at boot time means that it is specific to the Linux operating system in use on the development host (CentOS7, Ubuntu, etc.).

### *8.3 Configuring the Linux Kernel Device Driver*

The setting for the OCPI_DMA_CACHE_MODE environment variable, which controls the Linux cacheing mechanism on platforms that support cacheable DMA buffers, applies when using the Linux kernel device driver.

The default setting for OCPI_DMA_CACHE_MODE is 1. In this mode, DMA buffers use normal cacheing when accessed by the CPU and the caching of a buffer's contents is explicitly invalidated when new buffers are received by the CPU and explicitly flushed after new buffers are filled (written) by the CPU.

A setting to mode 0 disables the CPU caching for the DMA buffers, while a setting to mode 2 enables cache-coherent DMA between the CPU and other (OpenCPI platform) DMA masters on the bus/fabric.

All platforms support mode 0, most platforms support mode 1, and few platforms support mode 2. Using the default mode is usually the best policy.

### *8.4 Reserving Additional Memory for the Linux Kernel Device Driver*

When OpenCPI communicates over a system bus or fabric to other platforms on the bus/fabric, it uses the OpenCPI Linux kernel device driver for OpenCPI container discovery and DMA-based communication, which requires local (reserved) DMA memory resources. DMA memory resources must be allocated or reserved on the CPU-side memory that is accessible to both the CPU (via the local **mmap** system call) and OpenCPI's DMA engines, which are used on all DMA-capable system busses and which allow the FPGA to directly access (read and write) system memory.

On Linux, OpenCPI uses a default of 128 KB of memory for the OpenCPI Linux kernel device driver. However, OpenCPI applications may have buffering requirements that necessitate additional DMA memory resources.

OpenCPI requires a physically contiguous DMA memory area. Some Linux systems make this easy and dynamic (that is, possible when applications are running), and some do not. For those that do not, DMA memory reservation is required such that the DMA memory required by OpenCPI is reserved at boot time. Consequently, the requirement for DMA memory reservation is derived from 1) how much DMA memory is required by the applications that run, and 2) whether the Linux kernel requires DMA memory allocations above a certain size to be reserved at boot time.

This section describes how to use the Linux kernel parameter **memmap** to reserve 128 MB of additional DMA block memory from the Linux kernel at boot time for DMA-based communication on a PCI Express-based system. While this parameter supports many formats, the following usage has proven to be sufficient:

**memmap=*size*$*start***

Where *size* is the number of bytes to reserve in either hexadecimal or decimal, and *start* is the physical address in hexadecimal bytes. All addresses and sizes must be on even boundaries (**0x1000** or **4096** bytes).

The OpenCPI PCI DMA engine currently requires that the user accessible-mode DMA memory pool be in a 32 or 64-bit memory range; due to the way that Linux manages memory, it is recommended that the address be higher than the first 24 bits.

With these requirements, the first step to reserve additional memory for the example here is to find a usable contiguous memory range by examining the BIOS physical RAM map as reported by **dmesg**.

Run **dmesg** and filter on BIOS to review the physical RAM map:

```
dmesg | grep BIOS
```

The output should look similar to the following:

```
BIOS-provided physical RAM map:
  BIOS-e820: 0000000000000000 - 000000000009f800 (usable)
  BIOS-e820: 000000000009f800 - 00000000000a0000 (reserved)
  BIOS-e820: 00000000000ca000 - 00000000000cc000 (reserved)
  BIOS-e820: 00000000000dc000 - 00000000000e4000 (reserved)
```

```
BIOS-e820: 00000000000e8000 - 0000000000100000 (reserved)
BIOS-e820: 0000000000100000 - 000000005fef0000 (usable)
BIOS-e820: 000000005fef0000 - 000000005feff000 (ACPI data)
BIOS-e820: 000000005feff000 - 000000005ff00000 (ACPI NVS)
BIOS-e820: 000000005ff00000 - 0000000060000000 (usable)
BIOS-e820: 00000000e0000000 - 00000000f0000000 (reserved)
BIOS-e820: 00000000fec00000 - 00000000fec10000 (reserved)
BIOS-e820: 00000000fee00000 - 00000000fee01000 (reserved)
BIOS-e820: 00000000fffe0000 - 0000000100000000 (reserved)
```

Select a "(usable)" section of memory and reserve a subsection of that memory. Once the memory is reserved, the Linux kernel will ignore it. In this case, there are three usable sections to consider:

```
BIOS-e820: 0000000000000000 - 000000000009f800 (usable)
BIOS-e820: 0000000000100000 - 000000005fef0000 (usable)
BIOS-e820: 000000005ff00000 - 0000000060000000 (usable)
```

On close inspection of the usable regions, the first range is too small and is below the first 24 bits, while the third range is simply too small. Fortunately, the second address space meets the address range requirement (between 24 and 32 bits) and it is large enough to reserve several hundred megabytes of memory.

The starting memory address for the user-mode DMA region is calculated by subtracting **0x08000000** (128 MB) from the largest memory region available, as long as it is greater than **0x08000000** (128MB) and inside the 32-bit address range (address is less than 4GB). In this example, the second region is the largest: **0x5FEF0000** - **0x100000** = **0x5FDF0000** = **1,608,450,048** (1.6GB) and it is inside of the 32-bit address space. The starting memory address (**0x5FEF0000** - **0x08000000**) is **0x57EF0000**. This is the value used to construct the **memmap** parameter, as shown below:

```
memmap=128M$0x57EF0000
```

When calculating the starting address, ensure that it occurs on an even page boundary of 4KB. This may necessitate an additional adjustment to the starting address. In some cases, the **$dmesg | grep BIOS** command returns a value like **0x5FEFFFFF**. It is recommended to simply change this address so that its low word is all zeros, for example, **0x5FEF0000**, prior to calculating the starting address.

Once the **memmap** parameter has been calculated, it will need to be added to the kernel command line in the boot loader. For CentOS, the **grubby** utility can be used to add the parameter to all kernels in the startup menu. The single quotes are *required* or the shell will interpret the **$0**:

CentOS7 uses **grub2**, which requires a *double* backslash:

```
sudo grubby --update-kernel=ALL --args=memmap='128M\\$0x57EF0000'
```

To verify that the current kernel has the argument set:

```
sudo -v
sudo grubby --info $(sudo grubby --default-kernel)
```

CentOS7 displays a *single* backslash before the **$**, for example:

```
args="ro rdblacklist=nouveau crashkernel=auto rd.lvm.lv=vg.0/root
quiet audit=1 boot=UUID=96933\cb5-f478-4933-a0d4-16953cf47f5c
memmap=128M\$0x57EF0000 LANG=en_US.UTF-8"
```

If no longer desired, the parameter can also be removed with the following command:

```
sudo grubby --update-kernel=ALL --remove-args=memmap
```

Reboot the system to apply the new configuration for memory reservation. Once the system has finished booting, examine the state of the physical RAM map to confirm that the desired memory has been reserved:

```
dmesg | more
Linux version 3.10.0-1160.31.1.el7.x86_64
(mockbuild@kbuilder.bsys.centos.org) (gcc version 4.8.5 20150623
(Red Hat 4.8.5-44) (GCC) ) #1 SMP Thu Jun 10 13:32:12 UTC 2021
Command line: BOOT_IMAGE=/vmlinuz-3.10.0-1160.31.1.el7.x86_64
root=/dev/mapper/centos-root ro crashkernel=auto
rd.lvm.lv=centos/root rd.lvm.lv=centos/swap rhgb quiet
LANG=en_US.UTF-8 memmap=128M$0x57EF0000
BIOS-provided physical RAM map:
  BIOS-e820: 0000000000000000 - 000000000009f800 (usable)
  BIOS-e820: 000000000009f800 - 00000000000a0000 (reserved)
  BIOS-e820: 00000000000ca000 - 00000000000cc000 (reserved)
  BIOS-e820: 00000000000dc000 - 00000000000e4000 (reserved)
  BIOS-e820: 00000000000e8000 - 0000000000100000 (reserved)
  BIOS-e820: 0000000000100000 - 000000005fef0000 (usable)
  BIOS-e820: 000000005fef0000 - 000000005feff000 (ACPI data)
  BIOS-e820: 000000005feff000 - 000000005ff00000 (ACPI NVS)
  BIOS-e820: 000000005ff00000 - 0000000060000000 (usable)
  BIOS-e820: 00000000e0000000 - 00000000f0000000 (reserved)
  BIOS-e820: 00000000fec00000 - 00000000fec10000 (reserved)
  BIOS-e820: 00000000fee00000 - 00000000fee01000 (reserved)
  BIOS-e820: 00000000fffe0000 - 0000000100000000 (reserved)
user-defined physical RAM map:
 user: 0000000000000000 - 000000000009f800 (usable)
 user: 000000000009f800 - 00000000000a0000 (reserved)
 user: 00000000000ca000 - 00000000000cc000 (reserved)
 user: 00000000000dc000 - 00000000000e4000 (reserved)
 user: 00000000000e8000 - 0000000000100000 (reserved)
 user: 0000000000100000 - 0000000057ef0000 (usable)
 user: 0000000057ef0000 - 000000005fef0000 (reserved) <== New
 user: 000000005fef0000 - 000000005feff000 (ACPI data)
 user: 000000005feff000 - 000000005ff00000 (ACPI NVS)
 user: 000000005ff00000 - 0000000060000000 (usable)
 user: 00000000e0000000 - 00000000f0000000 (reserved)
 user: 00000000fec00000 - 00000000fec10000 (reserved)
 user: 00000000fee00000 - 00000000fee01000 (reserved)
 user: 00000000fffe0000 - 0000000100000000 (reserved)
DMI present.
```

A new "**(reserved)**" area is shown between the second "**(useable)**" section and the (ACPI data) section. Now, when the **ocpidriver load** command is run, it will detect the new reserved area and pass that data to the Linux kernel device driver.

### *8.5 Erroneous CMA Region Memory Allocation Warning*

The Contiguous Memory Allocator (CMA) is a feature available in the most recent Linux kernels and in some older kernel versions. When the Linux kernel supports CMA, the OpenCPI Linux kernel device driver will attempt to make use of the CMA region for direct memory access. In use cases where many memory allocations are made, you may receive the following kernel message:

```
alloc_contig_range test_pages_isolated([memory start], [memory end])
failed
```

This is a kernel warning, but does not indicate that a memory allocation failure occurred, only that the CMA engine could not allocate memory in the first pass. Its default behavior is to make a second pass, and if that succeeds, you should not see any more error messages. This message cannot be suppressed, but can be safely ignored. An actual allocation failure will generate unambiguous error messages.

# 9 Using the OpenCPI GUI

OpenCPI has an optional “proto-IDE” called the OpenCPI GUI for performing most development tasks. It is optional in that it is a front-end to the **`ocpidev`** command-line tool and everything it does can be done using **`ocpidev`** without an OpenCPI GUI installation.

The OpenCPI GUI is available as source code that can be downloaded with the **`git`** utility; the ***OpenCPI Installation Guide*** provides instructions on how to download, start, and configure it. A user guide for the OpenCPI GUI is provided on the OpenCPI documentation website.

# 10 Working with FPGA Vendor Tools with OpenCPI

FPGA vendor tools typically have environment setup scripts that are recommended to be run (sourced) by the user before using their tools. In some cases it is even recommended to do this in shell startup files that are run at login or when *any* new shell/terminal windows are started. ***This is not recommended nor necessary when using OpenCPI.*** OpenCPI runs vendor tools in sand-boxed environments so they can easily coexist with other vendor tools without interference and can never "pollute" the general environment for other installed software.

If you need to run vendor tools separate from OpenCPI, do so in a separate terminal/shell window where you explicitly run (source) the vendor setup scripts in that window without affecting any other windows.

Information about installing the tools is in the ***OpenCPI Installation Guide***.

# 11 Glossary of Terms

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

### *11.1 OpenCPI Terminology*

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

***ACI***

See ***Application Control Interface***.

***adapter worker***

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

***application***

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

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

***Application Control Interface (ACI)***

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

***application worker***

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

***argument***

See ***operation argument***.

***artifact***

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

***asset***

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

***authoring model***

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

***back pressure***

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

***build configuration***

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

***CDK***

See ***Component Development Kit***

***component***

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

***Component Development Kit (CDK)***

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

***component library***

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

***component specification***

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

***component unit test suite***

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

***configuration property***

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

***configuration property accessibility***

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

***container***

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

***control-application***

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

***control interface***

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

***control operations***

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

***control plane***

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

***control software***

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

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

***core project***

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

***data interface***

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

***data plane***

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

***device proxy worker***

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

***device worker***

See ***HDL device worker***.

***Digital Radio Controller (DRC)***

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

***HDL assembly***

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

***HDL authoring model***

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

***HDL build hierarchy***

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

***HDL build process***

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

***HDL card***

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

***HDL data interface***

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

***HDL device emulator worker***

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

***HDL device worker***

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

***HDL interface***

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

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

***HDL platform***

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

***HDL platform configuration***

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

***HDL platform worker***

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

***HDL primitive***

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

***HDL primitive core***

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

***HDL primitive library***

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

***HDL slot***

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

***HDL subdevice worker***

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

***HDL worker***

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

***lifecycle state model***

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

***OAS***

See ***OpenCPI Application Specification***.

***OCS***

See ***OpenCPI Component Specification***.

***OHAD***

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

***OHPD***

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

***opcode***

See ***operation code***.

***OpenCL (OCL) authoring model***

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

***OpenCPI Application Specification (OAS)***

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

***OpenCPI Component Specification (OCS)***

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

***OpenCPI HDL Assembly Description (OHAD)***

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

***OpenCPI HDL Platform Description (OHPD)***

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

***OpenCPI Protocol Specification (OPS)***

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

***OpenCPI System support Project (OSP)***

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

***OpenCPI Worker Description (OWD)***

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

***operation argument***

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

***operation (within a protocol)***

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

***operation code (opcode)***

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

***OPS***

See ***OpenCPI Protocol Specification***.

***OSP***

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

***OWD***

See ***OpenCPI Worker Description***.

***package ID***

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

***parameter***

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

***platform***

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

***platform worker***

See ***HDL platform worker***.

***port***

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

***port readiness***

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

***project***

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

***project registry***

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

***property***

See ***configuration property***.

***protocol specification***

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

***protocol summary***

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

***RCC, RCC authoring model***

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

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

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

***registry***

See ***project registry***.

***RCC worker***

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

***run condition***

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

***run method***

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

***spec file***

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

***SpecProperty***

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

***system***

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

***target***

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

***worker***

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

***worker property***

A ***worker property*** is a configuration property related to a particular implementation (design) of a worker; that is, one that is not necessarily common across a set of implementations of the same high-level component specification (OCS). A worker property is additional to the properties defined by the component specification being implemented. See the section on how a worker access its properties in the ***OpenCPI RCC Development Guide*** and the sections on property access and property data types in the ***OpenCPI HDL Development Guide*** for more information.

***unit test***

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

***Zero-Length Message (ZLM)***

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

### *11.2 Industry Terminology*

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

***Advanced eXtensible Interface (AXI)***

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

***Advanced RISC Machine (ARM)***

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

***ARM***

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

***AXI***

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

***Board Support Package (BSP)***

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

***Central Processing Unit (CPU)***

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

***CPU***

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

***Digital Signal Processor (DSP)***

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

***digital signal processing***

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

***DSP***

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

***eXtensible Markup Language (XML)***

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

***Field-Programmable Gate Array (FPGA)***

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

***FPGA***

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

***FPGA bitstream***

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

***framework***

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

***gateware***

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

***General-Purpose Processor (GPP)***

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

***GPP***

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

***GPU***

See ***Graphics Processing Unit***.

***Graphics Processing Unit (GPU)***

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

***Hardware Description Language (HDL)***

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

***HDL***

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

***Integrated Synthesis Environment (ISE®) Design Suite***

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

***ISE® Simulator (Isim)***

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

***isim***

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

***OCL, OpenCL***

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

***Open Computing Language (OCL, OpenCL)***

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

***OSS***

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

***Open Source Software (OSS)***

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

***PCI***

See ***Peripheral Component Interconnect***.

***PCIe***

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

***Peripheral Component Interconnect (PCI)***

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

***Peripheral Component Interconnect Express (PCIe)***

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

***RPM***

See ***RPM Package Manager***.

***RPM Package Manager (RPM)***

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

***System on a Chip (SoC)***

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

***Verilog***

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

***VHSIC Hardware Description Language (VHDL)***

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

***Vivado® Design Suite (Vivado, Xilinx Vivado)***

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

***Vivado® Simulator***

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

***XML***

See ***eXtensible Markup Language***.

***Xsim***

See ***Vivado® Simulator***.