OpenCPI
User Guide

OpenCPI Release: v2.3.4

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

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***) 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.

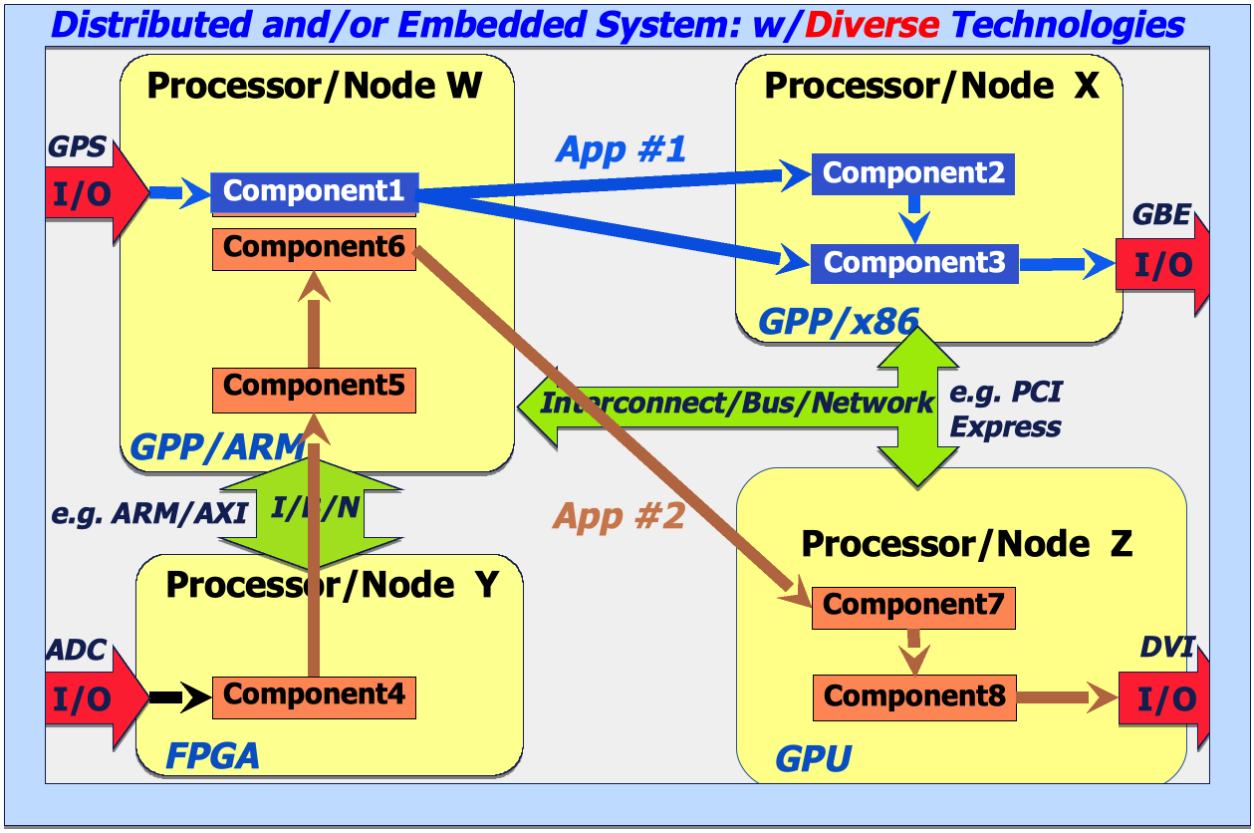


Figure 1: Embedded System with Diverse Processing and Interconnect Technologies

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*** (available with the OpenCPI GUI source code) 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)

Each hardware platform has a ***Getting Started Guide*** which provides specific details of installation and usage for a platform beyond the generic descriptions in the ***OpenCPI Installation Guide***.

The assets developed in each OpenCPI project can have their own documents, including documents (sometimes called data sheets for component specs and workers) for:

- Component specifications
- Workers
- Platforms
- Applications

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).

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

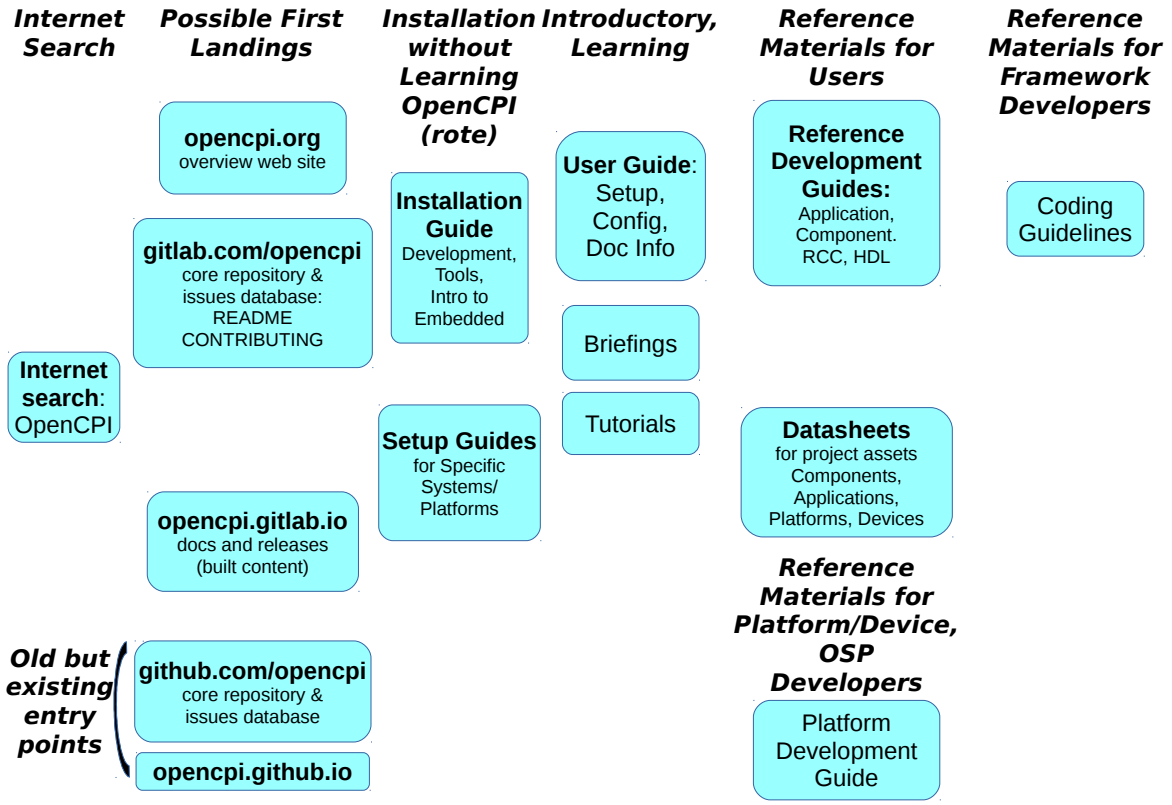


Figure 2: Documentation for OpenCPI

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: **$OCPI_ROOT_DIR/system.xml** **$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 CentOS7 download link
Ubuntu 18.04 Linux on x86-64 bit	**`ubuntu18_04`**		Ubuntu 18.04 LTS general information link Ubuntu 18.04 LTS download link
Ubuntu 16.04 Linux on x86-64 bit	**`ubuntu16_04`**		Ubuntu 16.04 LTS general information link Ubuntu 16.04 LTS download link
MacOS Catalina 10.15	**`macos10_15`**	No FPGA tools or drivers	MacOS Catalina general information link 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 2017.1-2019.2	**`xsim`**	Vivado® Simulator (XSIM) is part of all Vivado HL Editions, including the free WebPack edition. Vivado Simulator general information link Vivado Simulator download link
Xilinx ISE ISim 14.7	**`isim`**	ISE® Simulator (ISim) is part of the older Xilinx ISE tool set. ISE/ISim general information link ISE/ISim download link
Siemens/Mentor Modelsim 10.4c-10.6c	**`modelsim`**	Modelsim® is powerful, fast and expensive. Modelsim general information link No download link available before purchase.

Below 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 OSs).

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 USRP E310	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

The table on the next page 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 any x86-64bit CPU	**ocpi.core** built-in	A CentOS7 installation
ubuntu18_04	Ubuntu 18.04 on x86_64 CPUs	**ocpi.core** built-in	An Ubuntu 18.04 installation
ubuntu16_04	Ubuntu 16.04 on x86_64 CPUs	**ocpi.core** built-in	An Ubuntu 16.04 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 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 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 Xilinx Binary Zynq Release 2019.2 Xilinx Linux git clone 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 Xilinx Binary Release 2019.2 Xilinx Linux git clone Xilinx Linux tag: **xilinx_v2019.2**
adi_plutosdr0_32	Analog Devices Pluto Linux 0.32	**ocpi.osp.plutosdr**	Vivado SDK 2019.2
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
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+ gx230: User Guide, Ref Manual gx530: User Guide, Ref Manual
plutosdr	Analog Devices ADALM-PLUTO	**ocpi.osp.plutosdr**	Vivado WebPACK
zcu104	Xilinx Zynq-Ultrascale Dev Bd	**ocpi.assets** built-in	Vivado WebPACK

OpenCPI Platform Name	Description	OpenCPI Project/Repo	Dependencies required *before* building/installing the platform
`picoevb`	RHS Research PicoEVB	**`ocpi.platform`** built-in	Vivado SDK 2019.2
`adrv9361`	Analog ADRV9361 Zynq FPGA/PL	**`ocpi.osp.analog`**	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
`alst4, alst4x`	***Alst4 Getting Started Guide***
`e31x`	***Ettus E31x Getting Started Guide***
`matchstiq_z1`	***Matchstiq Z1 Getting Started Guide***
`ml605`	***ML605 Getting Started Guide***
`plutosdr`	***PlutoSDR Getting Started Guide***
`zed, zed_ise`	***ZedBoard Getting Started Guide***
`picoevb`	***RHS Research PicoEVB Getting Started Guide***
`adrv9361`	***Analog Devices ADRV9361-Z7035 Getting Started Guide***

7 About the OpenCPI System Configuration File

The OpenCPI system configuration file is an XML file that allows parameters 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 runtime modules (plugins) that support different types of runtime environments
- ***transfer:*** for the runtime modules (plugins) that support different data transport mechanisms.

Each driver has a boolean **load** attribute that specifies whether the module should be loaded (and thus enabled) in the runtime environment. Otherwise all attributes are module-specific.

The **device** child element can appear under any of the **container** driver elements and specifies attributes specific to individual devices 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 FPGA devices.

The **smbsize** attribute to the **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 module below it, which overrides the default.

At installation time, this file is normally only modified for specific device attributes.

8 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's guide for the OpenCPI GUI is provided with the source code installation.

Note: In a future release, this tool will be invokable from the OpenCPI command-line environment after source code installation and the ***OpenCPI GUI User Guide*** will be available as PDF on the OpenCPI documentation website opencpi.gitlab.io.

9 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***.

10 Documenting OpenCPI Assets (Preliminary Version)

This chapter describes how to document different OpenCPI asset types using the Sphinx-based asset documentation system provided with OpenCPI.

Note: the information in this chapter is preliminary and subject to change.

10.1 General Information

This section provides information that applies to documenting all OpenCPI asset types.

10.1.1 Documentation Development Workflow

The general procedure is:

1. Using the OpenCPI documentation development tool (currently **`ocpidoc`**), create a template document in the same directory as the asset you're documenting. For example, if you're documenting an HDL worker, create the template in the same directory where the HDL worker's OpenCPI Worker Description (OWD) and source files are.
2. Using a text editor, fill out the template document following the skeleton instructions provided there and the hints provided in this chapter. The templates are formatted in reStructuredText markup and use a combination of standard Sphinx directives and OpenCPI documentation-specific Sphinx directives.
3. Using the OpenCPI documentation development tool, build the completed document. This step generates an HTML file that is viewable with a browser.
4. Using a text editor, fix any errors in the source file that are causing problems in the HTML output and then build and check the HTML output. Do this step until the HTML output is error-free.

*10.1.2 Hints on Using **`ocpidoc`***

The **`ocpidoc`** tool is part of the OpenCPI development tool set and is executable as a shell command. To get help on **`ocpidoc`** command usage, use the command:

```
ocpidoc –-help
```

To get help on a specific verb (currently, create, build, and clean), use the command:

```
ocpidoc <verb> --help
```

To create template documents for an asset, use the command:

```
ocpidoc create <asset_type> <asset_name>
```

For example:

```
ocpidoc create component complex_mixer
```

To generate HTML output, use the command:

```
ocpidoc build -b
```

The **-b** option suppresses the spellchecker. We recommend using this option during documentation development for faster HTML generation.

To remove generated HTML output, use the command:

```
ocpidoc clean
```

Note: **`ocpidoc`** may be integrated with **`ocpidev`** in a future release so that asset documentation generation functions will be performed using **`ocpidev`** command-line options.

10.1.3 Standard Sphinx Directives used in OpenCPI Asset Documentation

The following table lists the Sphinx directives most frequently used in the asset documentation templates provided by the OpenCPI asset documentation system.

Table to be supplied.

See the Directives section of the Sphinx documentation for descriptions of these directives.

10.1.4 OpenCPI-specific Directives

The OpenCPI asset documentation system provides a set of OpenCPI-specific directives used in the asset documentation templates that automate some of the asset documentation tasks. The next sections describe these directives.

10.1.4.1 Properties Directive

This directive lists the properties of a component, as defined in the component specification, in the “Properties” section of the component's documentation. Its usage is:

```
.. ocpi_documentation_properties::
```

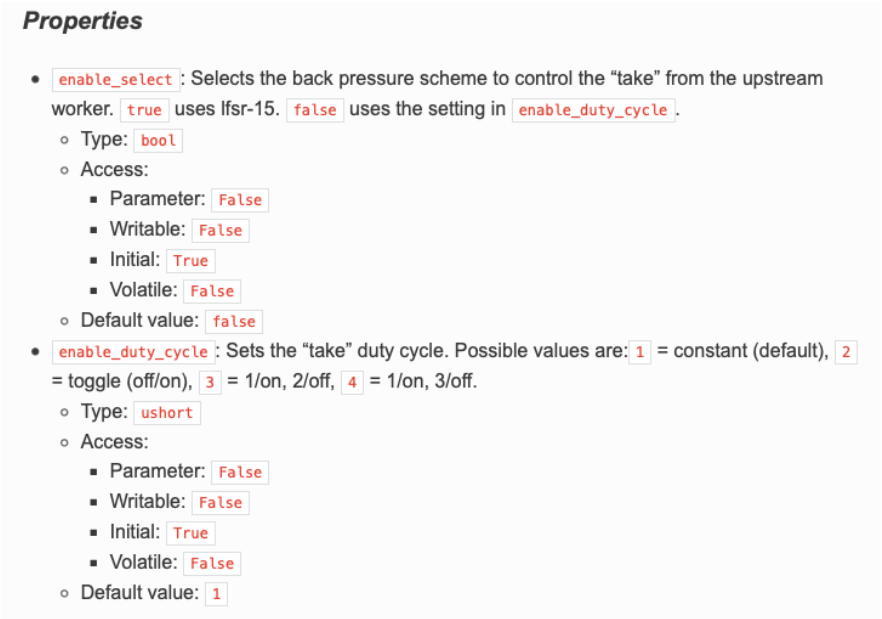
No arguments are needed. The options that can be used are:

- **`parameters`** - By default, the directive lists both “parameter” properties and “non-parameter” properties in the “Properties” section. This option sets that “parameter” properties, rather than properties that are not parameters, are to be listed.
- **`component_spec`** - allows overriding the automatically-determined component specification path. When used, the file path of the component specification relative to the current documentation file must be provided. When this option is not used, the directory above the directory the file this directive is in is searched for a component specification based on the current directory's name.
- Additional text to describe each property, added in the body of the directive. This text must be the name of the property followed by a colon symbol (:), with each property text on a new line. (Additional property text is not passed as an option of the directive, as directive option names must be known at the time of writing the directive, which for property names is not possible.)

Here is an example from the core project's **`backpressure`** component's document, where the property descriptions are added by hand:

```
.. ocpi_documentation_properties::

   enable_select: Selects the back pressure scheme to control the
"take" from the upstream worker.  ``true`` uses lfsr-15. ``false``
uses the setting in ``enable_duty_cycle``.

   enable_duty_cycle: Sets the "take" duty cycle. Possible values
are:``1`` = constant (default), ``2`` = toggle (off/on), ``3`` =
1/on, 2/off, ``4`` = 1/on, 3/off.
```

The HTML output looks like this:



Properties

- `enable_select` : Selects the back pressure scheme to control the “take” from the upstream worker. `true` uses lfsr-15. `false` uses the setting in `enable_duty_cycle` .
  - Type: `bool`
  - Access:
    - Parameter: `False`
    - Writable: `False`
    - Initial: `True`
    - Volatile: `False`
  - Default value: `false`
- `enable_duty_cycle` : Sets the “take” duty cycle. Possible values are: `1` = constant (default), `2` = toggle (off/on), `3` = 1/on, 2/off, `4` = 1/on, 3/off.
  - Type: `ushort`
  - Access:
    - Parameter: `False`
    - Writable: `False`
    - Initial: `True`
    - Volatile: `False`
  - Default value: `1`

10.1.4.2 Ports Directive

This directive lists the ports of a component, as defined in the component specification (OCS), in the “Ports” section of the component's documentation. Its usage is:

```
.. ocpi_documentation_ports::
```

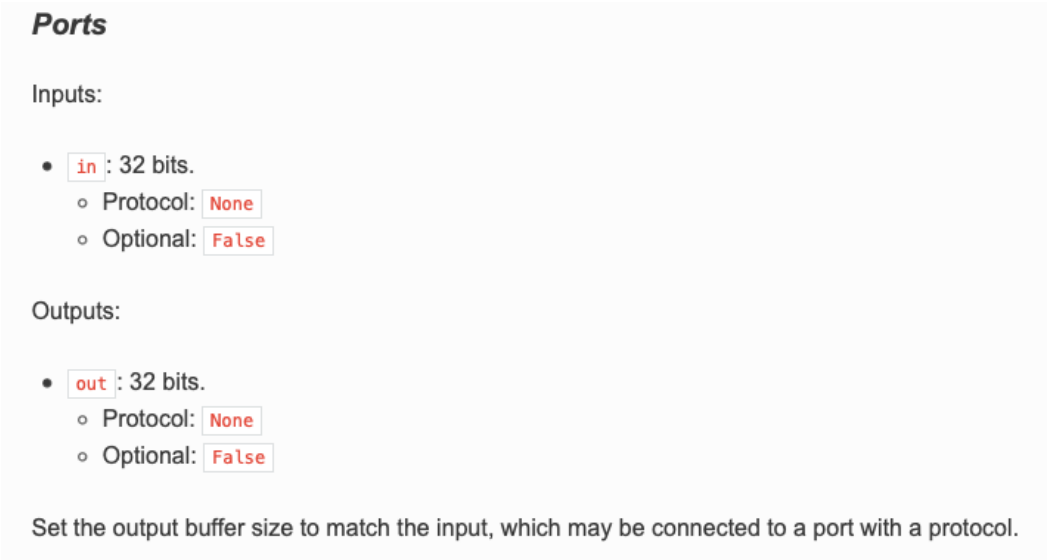
No arguments are needed. The options that can be used with the directive are:

- **`component_spec`** - allows overriding the automatically-determined component specification path. When used, the file path of the component specification relative to the current documentation file must be provided. When this option is not used, the directory above the directory the file this directive is in is searched for a component specification based on the current directory's name.
- Additional text to describe each port, added in the body of the directive. This text must be the name of the port followed by a colon symbol (:), with each port text on a new line. (Additional port text is not passed as an option of the directive, as directive option names must be known at the time of writing the directive, which for port names is not possible.)

Here is an example from the core project's **`backpressure`** component's document, where the port descriptions and additional text are added by hand:

```
.. ocpi_documentation_ports::

   in: 32 bits.
   out: 32 bits.

Set the output buffer size to match the input, which may be
connected to a port with a protocol.
```

The HTML output looks like this:



Ports

Inputs:

- `in`: 32 bits.
  - Protocol: `None`
  - Optional: `False`

Outputs:

- `out`: 32 bits.
  - Protocol: `None`
  - Optional: `False`

Set the output buffer size to match the input, which may be connected to a port with a protocol.

10.1.4.3 Implementation Directive

This directive adds worker implementation detail to a component's documentation page. Its usage is:

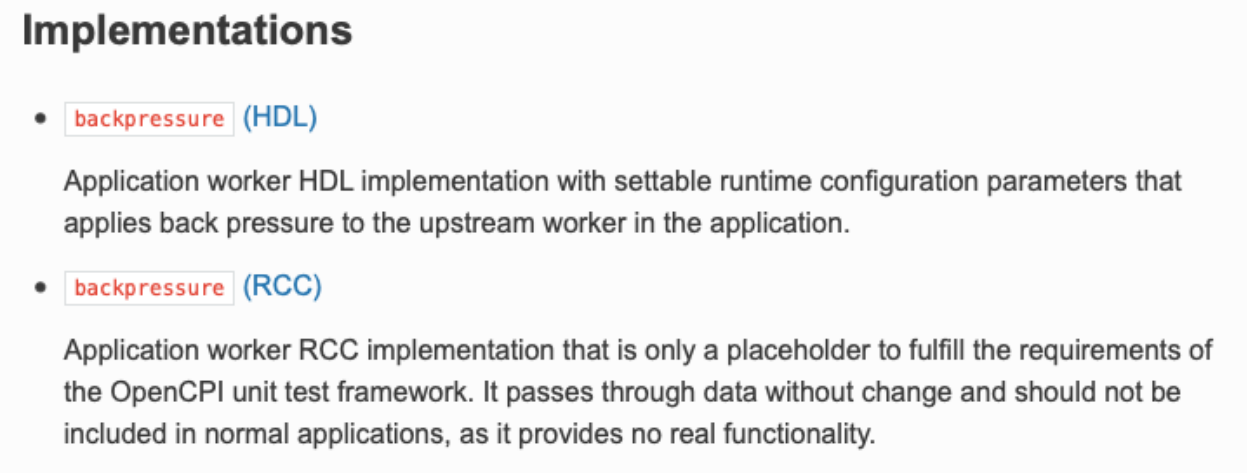
```
.. ocpi_documentation_implementation:: <worker-path> [<worker-path> ...]
```

Each **`<worker-path>`** argument specifies the path to the worker directory that contains the worker to be listed, up to a maximum of 10 worker paths.

Here is an example from the core project's **`backpressure`** component:

```
.. ocpi_documentation_implementations:: ../backpressure.hdl ../backpressure.rcc
```

The HTML output looks like this:



Implementations

- `backpressure` (HDL)

  Application worker HDL implementation with settable runtime configuration parameters that applies back pressure to the upstream worker in the application.

- `backpressure` (RCC)

  Application worker RCC implementation that is only a placeholder to fulfill the requirements of the OpenCPI unit test framework. It passes through data without change and should not be included in normal applications, as it provides no real functionality.

Note: this directive is used in a component's documentation. To automatically generate worker documentation sections in worker documentation, use the worker directive.

10.1.4.4 Worker Directive

This directive lists the worker-specific properties and build configurations in the "Properties" section of a worker's documentation and lists the worker-specific ports in the "Worker Ports" section. (The component properties that can be applied to the worker are listed in the component's documentation). Its usage is:

```
.. ocpi_documentation_worker::
```

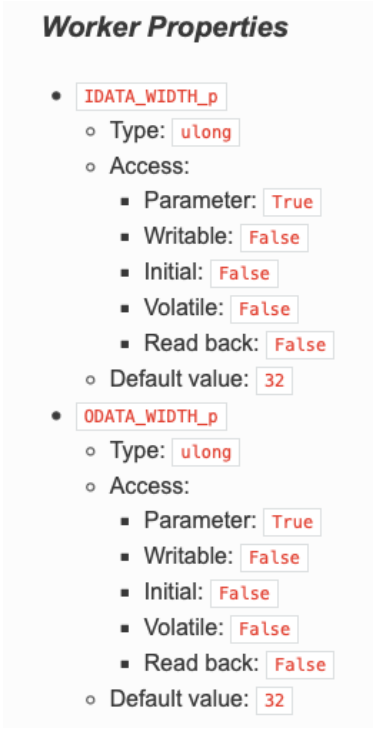
By default, No arguments are needed. The options that can be used are:

- **`worker_description`** - allows overriding the automatically-determined worker description XML file path (OWD). When used, the file path of the worker description relative to the current documentation file must be provided. When not set, the directory the current documentation file is in will be searched for a worker description.
- **`build_file`** - allows overriding the automatically-determined worker build file path. When used, the file path of the worker build file relative to the current documentation file must be provided. When not set, the directory the current documentation file is in will be searched for a worker description.
- Additional text to describe any worker properties (including parameters), added in the body of the directive. This text must be the name of the property followed by a colon symbol (:), with each property text on a new line. (Additional property text is not passed as an option of the directive, as directive option names must be known at the time of writing the directive, which for property and parameter names is not possible.)

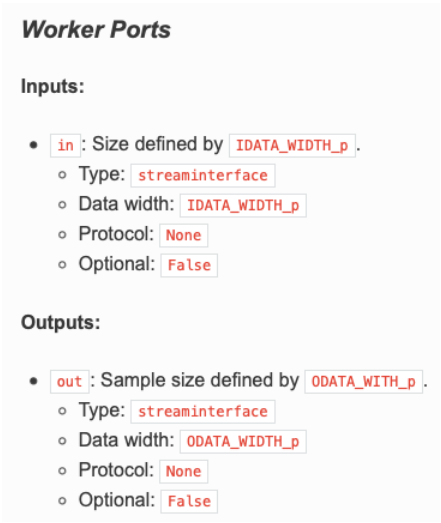
Here is an example of using this directive from the core component's **`backpressure`** HDL worker, where the worker's description and build file paths are automatically determined and descriptions for the worker ports are added by hand:

```
.. ocpi_documentation_worker::

   in: Size defined by ``IDATA_WIDTH_p``.

   out: Sample size defined by ``ODATA_WITH_p``.
```

Here is the HTML output for the "Worker Properties" section:



Here is the HTML output for the “Worker Ports” section:



Note that this directive should be used to document *workers*; to add worker details to a *component's* documentation pages, use the “implementation” directive.

10.1.4.5 Test Result Summary Directive

Note: this directive does not currently function.

This directive adds a testing summary table to the component documentation. This directive should be used with a single occurrence of the test detail directive. Its usage is:

```
.. ocpi_documentation_test_result_summary::
```

No arguments are needed. The options that can be used are:

- **`test_log`** - allows overriding the automatically determined component test log file path. When this option is used, the file path of the test log relative to the current documentation file must be provided. When this option is not used, the directory the current documentation file is in is searched for a test log.

Here is an example of using this directive, where the component's test log path is automatically determined:

```
.. ocpi_documentation_test_result_summary::
```

Note: If this directive is not used with the test detail directive, the links this directive creates are not resolved.

10.1.4.6 Test Detail Directive

Note: this directive does not currently function.

This directive lists, as sections, the different test cases and subcases. This directive provides the destinations of he links generated by the test result summary directive. Its usage is:

```
.. ocpi_documentation_test_detail::
```

No arguments are needed. The options that can be used are:

- **`test_log`** - allows overriding the automatically determined component test log file path. When this option is used, the file path of the test log relative to the current documentation file must be provided. When this option is not used, the directory the current documentation file is in is searched for a test log.

Here is an example of using this directive, where the component's test log path is automatically determined:

```
.. ocpi_documentation_test_detail::
```

10.1.5 Hints that Apply to Documenting All OpenCPI Asset Types

Here are some general hints for working with the template files:

- When editing a section heading in a template, be sure to adjust the line underneath the heading text to be the same length as the text, or you'll get a build error.
- The preferred graphics format for figures and illustrations is SVG. Use an external tool to create your images that can save files in SVG. For example, LibreOffice Draw is a free drawing tool that supports SVG format and is included with the LibreOffice package.
- Make sure the properties and ports defined in your component specifications (OCSes) and worker definitions (OWDs) include **`<Description>`** XML elements. The asset documentation system automatically generates the properties and ports documentation from these elements. If they're not there, the system won't generate any documentation for them, and you'll have to add the descriptions by hand.

- Use title case in section headings and in figure and equation captions. Examples of title case are: “Running the Application”, “Worker Ports”, “MyComponent Block Diagram”.

10.2 Documenting an OpenCPI Application

The document that describes an OpenCPI application exists in the **`applications/`** subdirectory of an OpenCPI project in either in the same **`<application-name>`** directory as the application it describes or directly in the **`applications/`** directory if the application XML does not have its own directory.

An application document should provide:

- One or two introductory paragraphs that describe the application's purpose and what it does
- Known limitations as to which OpenCPI platforms it can use, either inclusively or exclusively
- Things that must be in place before the application can be run, like hardware and build prerequisites
- How to build and run the application or whether it can simply be built and run with **`ocpidev`**
- How to verify its success or failure
- Troubleshooting information and/or a description of any known issues

To document an OpenCPI application:

- Create a template file in the application's subdirectory (or in the **`applications/`** directory of the project if the application does not have its own subdirectory), specifying the application's name. The resulting template is named **`<application-name>-application.rst`**.
- Fill in the sections in the suggested outline in the template with the information about your application, adding new sections as necessary and/or deleting any section headings you aren't going to use.

10.3 Documenting an OpenCPI Component

The documentation that describes an OpenCPI component exists in a separate directory within a component library, and provides a couple of templates you need to fill out. The general workflow is to:

- Create the component directory and its templates
- Add your component information to the main template
- Customize the example application template to your component
- Create worker documents for all the HDL and/or RCC workers that use the component specification. The component document will contain links to these documents.

10.3.1 Create the Component Documentation Directory

Create the component documentation directory in the component library, specifying the component name. The resulting directory is named <***component-name***>**.comp** and contains three template files:

- A template document file named **<*component-name*>-index-rst**. This file is the main template for documenting the component. Follow the skeleton outline and hints in this section to fill it out.
- A template OpenCPI Application Specification (OAS) file named **example_app.xml**. This file is automatically included verbatim into the main component document. You'll need to customize it to your component's features.
- A template document file named **<*component-name*>-test.rst**. This file contains an OpenCPI documentation directive that automatically generates test details for the component. You can ignore this file for now, because the functionality is currently incomplete.

10.3.2 Edit the Component Document Template

This section provides some hints for filling out the main documentation template file **<*component-name*>-index.rst** in addition to what's given in the template.

Note: be sure to adjust the heading line under the component title to be the same length as the component title string. If the line is too long or too short, you'll get a syntax error when you try to build the source file.

10.3.2.1 Add a Top-Level Description

The template provides a placeholder for adding a one-line description that summarizes the component's purpose. It should be an incomplete sentence, like a "man" page description. Here's the one-sentence description for the c**omplex_mixer** component:

```
Multiplies complex IQ data input with a digital sine wave generated
by a numerically-controlled oscillator (NCO).
```

See the **`complex_mixer`** component in **`projects/assets/components/dsp_comps/complex_mixer.comp/complex_mixer-index.rst`** for an example.

Note: the top-level description for the **`complex_mixer`** component document contains additional information to the one-line summary. You do not need to add this information to your component description. In a future release, it will be automatically generated.

10.3.2.2 Create a Component Block Diagram Figure

Note: the "Design" section will be renamed in a future release.

The "Design" section in the template should describe the component's purpose and operation. It's recommended to provide a block diagram of the component's data flow, ports, and properties as part of the component's description in the "Design" section. Use different colors for text and arrows to distinguish between the property interface and the data interface. The block diagram for the component should only show the component properties and ports, not the worker-specific ones.

To create the block diagram and any other figures you want to include that describe the component, use an external tool that supports saving to SVG format, like LibreOffice Draw. Save your figure source and SVG output in the **`<component-name>.comp`** directory.

(**Note**: the OpenCPI components currently store figures in the component unit test suite subdirectory in the path **`<component-name>.test/doc/figures/`**. In a future release of OpenCPI, they will be moved to the **`<component-name>.comp`** subdirectory.

The template provides a sample block diagram reST markup with the name of your component inserted. To this template, add:

- The path to your block diagram SVG file
- The title for your caption, in title case. For example:

```
MyComponent Block Diagram
```

- (optional) An alternate caption title. For example:

```
   :alt: Block Diagram of MyComponent Properties and Ports
```

Follow the same conventions for any other figures you include. See the component **`*-index.rst`** files in **`projects/core/components`** for examples.

10.3.2.3 Describe Any Component Equations

Some components descriptions may benefit from documenting the equations used by the component. You use the standard Sphinx "math" directive to create equations. The template provides a sample equation markup that includes a bulleted list for describing its arguments, with the name of your component inserted. To this template:

- Add the markup for your equation

- (optional) Edit the bulleted list with the markup for your arguments and a description of what they are, or delete the list if you don't need it.

See the **`complex_mixer`** component in **`projects/assets/components/dsp_comps/complex_mixer.comp/complex_mixer-index.rst`** for examples.

10.3.2.4 Add Subsections to "Design" As Necessary

Some component descriptions benefit from having topics organized into subsections within the "Design" section. See the **`projects/core/components/file_read.comp/file_read-index.rst`** file for an example of how to organize a "Design" section into different subsections in reST markup.

10.3.2.5 Describe the Component's Properties

The "Properties" section of the template contain the OpenCPI-specific "properties" directive that automatically populates it with bulleted lists of property information.

If the property definitions in the component spec contain descriptions (using the **`<Description>`** element), there is nothing you need to do here. The directive uses the description supplied in the element. If not, you'll need to add descriptions of your component's properties by hand. The template provides placeholders below the "properties" directive for adding additional text to property descriptions. Use these placeholders to add your property values and descriptions. The example reST markup shown in the OpenCPI-specific "properties" directive description adds a description for two properties under the directive. Be sure to keep each description on a single line (no line breaks).

You can also use the placeholder to provide property property information that's additional to what's provided in the component spec. Change the **`property_name`** or string in the placeholder to the name of your property and then provide the additional information as the text value.

10.3.2.6 Describe the Component's Ports

The template's "Ports" section contains an OpenCPI-specific "ports" directive that automatically populates the section with bulleted lists of port information taken from the component spec under the headings "Input" and "Output" when the reST file is built.

If the port definitions in your component spec do not contain description attributes or elements, you'll need to add them by hand. The template provides placeholder lines for one input port named **`input`** and one output port named **`output`**:

```
.. ocpi_documentation_ports::

   input: Primary input samples port.
   output: Primary output samples port.
```

Edit these lines with the name your component specification uses for the input and output ports and a brief description. Make sure you keep these lines aligned with the directive.

If you have additional descriptive information that's not in your description attributes and elements, you can add it below the "ports" directive, separated by a space and not indented. See the example of the **`backpressure`** core component in the OpenCPI-specific "ports" directive description.

10.3.2.7 Describe the Component Unit Test Suite

You can use the "Testing" section in the template to provide a description of your component test suite's purpose, function, test cases and expected results. Add this text under the section heading and above the OpenCPI-specific directive for generating a test summary. Note that the OpenCPI-specific "test result summary" directive is currently incomplete. You can comment out the directive if desired.

10.3.3 Customize the Example Application Template

The purpose of the "Example Application" section is to provide a meaningful example of how your component is used. The **`example_app.xml`** template is a simple OpenCPI Application Specification (OAS) for illustrating how your component can be used. The template application uses the built-in OpenCPI core components **`file_read`** and **`file_write`** with your component connected between the two. To this template, you add the properties and ports defined for your component and any additional XML that's required to demonstrate how it works. If the application template is too simple for your component, you can provide your own example OAS.

10.3.4 Create the Worker Documentation

The "Implementation" section in the main component document template contains the OpenCPI-specific "implementation" directive that references the HDL and/or RCC subdirectories that should contain the worker documents. The worker section describes how create these documents.

10.4 Documenting an OpenCPI Project

The document that describes an OpenCPI project exists at the top level of the project. It serves as the "welcome" page for the project and provides:

- A brief description of the purpose of the project and the assets it contains, similar to a README file
- A description of the criteria or rules in place for adding assets to the project or for excluding them from the project
- The root **`toctree`** directive, which specifies a list of paths to the subdirectories in the project that each contain a directory "index" file that provide the paths to the asset documents in that subdirectory. For example, the notation **`/specs/specs`** tells the asset documentation system to look in the project's top-level **`specs/`** directory for an index file named **`specs.rst`** that provides the paths to the asset documents for specifications (usually protocol specs) that are contained in the **`/specs`** directory.

To document an OpenCPI project:

- Create the template file in the top level of the project. It is named **`index.rst`** and this name should not be changed.
- Replace the notation **`%%NAME_CODE%%`** at the very top of the template file with whatever code name for the project you want to use. (How this feature is used is TBD.)
- Replace the notation **`%%NAME_PROPER%%`** in the template file with the "official" name of your project.
- Delete the "Skeleton outline" and "Description of project" lines in the template file and replace them with a brief description of your project and any criteria for placing content there.
- Modify the default paths provided in the template's root **`toctree`** directive to the subdirectory paths that contain your project's asset documentation.

10.5 Documenting an OpenCPI Protocol

The documentation that describes an OpenCPI protocol exists in a **`specs/`** directory located at the top level of the project or sometimes in an HDL subdirectory (for example, **`hdl/devices/specs/`** or **`hdl/platforms/specs/`**) along with the protocol's specification (OPS).

To document an OpenCPI protocol, create a template file in the **`specs/`** directory, specifying the name of the protocol. The resulting template is named **`<protocol-name>-protocol.rst`**.

The template has only one section: "Protocol", which includes the protocol specification (OPS) XML file. To this template, add:

- A one-sentence description (incomplete sentence) of the protocol's purpose underneath the title. For example:

```
Protocol for streamed complex short data samples, with time and
metadata opcodes which are used to provide additional information
about the sample data while maintaining alignment with samples in
the streamed data.
```

- An introductory sentence and a "literalinclude" link to the metadata XML file that the OPS references. For example, if the OPS references the metadata file **`timed_samples_metadata.xml`**, add the following lines:

```
The included ``timed_samples_metadata.xml`` protocol structure is:

.. literalinclude:: timed_samples_metadata.xml
   :language: xml
```

Note: be sure to adjust the heading line under the protocol title to be the same length as the protocol title string. If the line is too long or too short, you'll get a syntax error when you try to build the source file.

10.6 Documenting an OpenCPI Component Test Suite

The documentation for a component's unit test suite is part of the component documentation.

10.7 Documenting an OpenCPI Worker

To document an OpenCPI worker (HDL or RCC):

- Create the worker template file in the same directory as the worker source code and worker description (OWD), specifying the worker name. The asset documentation system knows which authoring model (RCC or HDL) to use for the template based on the directory you're in when you create it. The resulting file is named **`<worker-name>-worker.rst`**.
- Follow the skeleton outline and instructions in this section to describe your worker. The next sections provide more information about filling out the worker template.

Note: be sure to adjust the heading line under the worker title to be the same length as the worker title string. If the line is too long or too short, you'll get a syntax error when you try to build the source file.

10.7.1 Add a Worker Summary (optional)

By default, the "Implementation" section in a component document doesn't provide any additional information about the worker implementations and just links to the worker documents. If you want a short description of the worker summarizing its function and purpose to appear in "Implementation" section along with the link, you can add this text under the worker title in the worker document. The asset documentation system will automatically include this text in the "Implementations" section in the HTML output for the component document.

10.7.2 Include a Block Diagram of the Worker Implementation

If there's a need to illustrate your worker with a block diagram, follow the instructions given in the section on creating component figures, with the following differences:

- Describe the worker-specific properties and ports, not the component-specific ones.
- You can't use the figure numbering reference markup in a worker document. Remove the figure template markup and instead introduce the figure with "the following figure". Here is an example markup:

```
The following figure shows a block diagram representation of the HDL
implementation:

.. figure:: ../<component-name>.comp/figures/myworker_diagram.svg
   :alt: HDL Implementation for MyWorker
   :align: center

   MyWorker Block Diagram
```

It's recommended that you keep you worker document figures in your **`<component-name>.comp`** subdirectory.

10.7.3 Describe the Worker Properties

The “Details” section of the template contains the OpenCPI-specific “worker” directive that automatically creates a “Worker Properties” subsection underneath any information about the worker that you add manually, and fills them with bulleted lists of property information taken from the OWD. Note that this section only describes worker-specific properties; the component properties that can be used in the worker are described in the component document's “Properties” section. For an example, see the **`metadata_stressor`** component document.

10.7.4 Add Worker Ports Information

The asset documentation system does not automatically generate HDL worker ports information from the OWD. If your worker OWD provides worker port descriptions, you’ll need to add this information by hand. Create a new subsection “Worker Ports” under “Details”, and then add the input and output port details. Refer to **`projects/core/components/metadata_stressor.hdl/metadata_stressor-worker.rst`** file and the other **`*-worker-rst`** files in the OpenCPI **`core`** project for examples of how to format this information.

10.7.5 Add Subsections to “Details” As Necessary

Some workers have additional information that needs to be described, like finite state machines or control timing and signals information. Create new subsections under “Detail” as necessary to cover these topics. See the **`projects/core/components/metadata_stressor.hdl/metadata_stressor-worker.rst`** file for an example of how to format this information.

10.7.6 Ignore the Utilization Section

This section in the template contains an OpenCPI-specific directive that is not currently implemented. You can ignore this section for now and/or delete it from your document.

10.8 Documenting HDL-specific OpenCPI Assets

This section describes how to create documentation for OpenCPI HDL assets.

Note: The OpenCPI asset documentation system does not currently provide templates for HDL assets (with the exception of HDL primitive core and library assets) and does not contain support for automatically generating properties and ports information from the XML descriptions of these assets. Support for other HDL-specific assets will be added in a future release.

10.8.1 Documenting an HDL Assembly

The document that describes an OpenCPI HDL assembly exists in the **`hdl/assemblies/`** directory of an OpenCPI project in the same **`<assembly-name>`** directory as the HDL assembly it describes. An HDL assembly document should provide information about:

- Why the assembly was created
- Any specific applications it is intended to support
- Which containers are specified for the HDL assembly, and for each one, how it maps the inputs and outputs of the assembly to other devices or CPUs in the system.

The following points about the function of HDL assemblies and containers may be helpful in the HDL assembly documentation development process:

- The XML for an HDL assembly basically specifies: which workers, how they are (statically) parameterized, how they are connected, and what the "external ports" of the assembly are as a whole.
- A container specifies, for each external port of the assembly, which devices they are connected to or whether they are "connected to the system's interconnect", which usually means they are connectable to software workers.
- The "default container" specifies to connect all external ports to the system interconnect so they are connectable to software workers somewhere else.

As of this writing, the asset documentation system does not provide a template for documenting an OpenCPI HDL assembly. Use the file system to create a file named **`index.rst`** in the same directory as the HDL assembly, and then use standard reStructuredText markup and Sphinx directives to create headings and content within **`index.rst`**.

10.8.2 Documenting an HDL Card

To be supplied.

10.8.3 Documenting an HDL Device Worker

(General information on what an HDL device worker document should contain to be supplied.)

To document an HDL device worker, use the component and worker templates as follows:

- Create a component reST file and fill in the template with general information about the HDL device worker
- Create an HDL worker reST file and fill in the template with the details about the HDL device worker

The OpenCPI **`platform`** project's **`hdl/devices/`** directory (**`<location-of-opencpi>/projects/platform/hdl/devices/`**) contains examples of HDL device worker reST documents. The data sink QDAC CSTS HDL device worker and the data source QADC HDL device worker are two examples. The reST source files are:

`hdl/devices/data_sink_qdac_csts.comp/data_sink_qdac_csts-index.rst`
`hdl/devices/data_sink_qdac_csts.hdl/data_sink_qdac_csts-worker.rst`
`hdl/devices/data_src_qadc_csts.comp/data_src_qadc_csts-index.rst`
`hdl/devices/data_src_qadc_csts.hdl/data_src_qadc_csts-worker.rst`

10.8.4 Documenting an HDL Platform Worker

(General information on what an HDL platform worker document should contain to be supplied.)

To document an HDL platform worker, you currently use the component and worker templates as you would for HDL device workers or other workers:

- Create a component reST file and fill in the template with general information about the HDL platform worker
- Create an HDL worker reST file and fill in the template with the details about the HDL platform worker

10.8.5 Documenting a Device Proxy Worker

(General information on what a device proxy worker document should contain to be supplied.)

Use the component and worker templates to document a device proxy worker. See the OpenCPI platform project's AD9361 Config Proxy CSTS document for an example. The source locations for the reST source files are:

`hdl/devices/platform_ad9361_config_proxy_csts.comp/`
`hdl/devices/platform_ad9361_config_proxy_csts.rcc/`

10.8.6 Documenting an HDL Primitive

The documentation that describes an OpenCPI HDL primitive exists in each primitive's directory in the project's **`hdl/primitives/`** directory, along with the HDL primitive's source code. To document an OpenCPI HDL primitive core or a module in a primitive library, create a template file in the same directory as the target HDL primitive, specifying the primitive's name. The resulting template is named ***`<primitive-name>`*`-primitive.rst`**. (The asset documentation system does not currently

distinguish between a primitive core and a module in a primitive library. You can use the same template for either one.)

To this template, add:

- A one-sentence description (an incomplete sentence, like the description on a "man" page) of the primitive's function, underneath the title.
- Information about the primitive's function and purpose in the "Design" section, along with any equations or diagrams as necessary.
- Details on how the primitive is implemented in the "Implementation" section.
- A bulleted list of the primitive's input and output ports, in the "Ports" subsection of the "Interface" section. Here is a sample markup for the ports information; note that each list item should not have any line breaks:

```
Ports
~~~~~

 * ``clk`` (``std_logic``), in: Clock. Inputs and outputs registered
on rising edge.

 * ``rst`` (``std_logic``), in: Reset. Active high, synchronous with
rising edge of clock.

 * ``clk_en`` (``std_logic``), in: Enable. Enables / disables data
input when high.

 * ``data_in`` (``std_logic``), in: Data input. Binary input data."

 * ``seed`` (``std_logic_vector``, ``polynomial_g'length`` bits),
in: Seed. Initial CRC value.

 * ``final_xor``  (``std_logic_vector``, ``polynomial_g'length``
bits), in: Final XOR. Value to XOR CRC result with.

 * ``crc_out`` (``std_logic_vector``, ``polynomial_g'length`` bits),
out: CRC. Available after one clock cycle delay.
```

- Any dependencies or limitations for the primitive in the "Dependencies" and "Limitations" sections.

The HDL primitives assets provided with OpenCPI are not currently documented in the asset documentation system's format. However, you can look at the README files in the **`projects/core/hdl/primitives/`** subdirectories for examples of the information that should be supplied for this asset type.

You can also look at the HDL primitives documents provided in the OpenCPI SDR components library.

Everything you add to the template must be entered manually. There are currently no directives for automatically generating the information. If a section in the template does not apply to your primitive, delete it.

10.8.7 Documenting an HDL Signals Specification

To be supplied.

10.8.8 Documenting an HDL Slot

To be supplied.

10.8.9 Documenting an HDL Subdevice Worker

(General information on what an HDL subdevice worker document should contain to be supplied.)

To document an HDL subdevice worker, use the component and worker templates as follows:

- Create a component reST file and fill in the template with general information about the HDL subdevice worker
- Create an HDL worker reST file and fill in the template with the details about the HDL subdevice worker

The OpenCPI **`platform`** project's **`hdl/devices/`** directory (**`<location-of-opencpi>/projects/platform/hdl/devices/`**) contains examples of HDL subdevice worker reST documents. The AD9361 CSTS Data Sub HDL subdevice worker and the AD9361 SPI CSTS HDL subdevice worker are two examples. The reST source files are located in:

`hdl/devices/platform_ad9361_csts_data_sub.comp/`
`hdl/devices/platform_ad9361_csts_data_sub.hdl/`
`hdl/devices/platform_ad9361_spi_csts.comp/`
`hdl/devices/platform_ad9361_spi_csts.hdl/`

10.9 Creating the OpenCPI Asset Documentation Structure

OpenCPI asset documentation structure is a nested series of reST document files that follows the OpenCPI project directory layout. The project document is the top-level document. It gives the paths to reST directory index files in the next level of subdirectories (**`specs/`**, **`components/`**, **`hdl/`**, **`applications/`**). These files either list the asset documents in the subdirectory or give the paths to directory index files in the next level of subdirectories. For example, the index file in the **`specs/`** directory usually lists (as a wildcard expression) the protocol documents contained in the subdirectory. The index file in the **`components/`** directory usually gives the paths to component library index files, because this directory often contains multiple component libraries, each with their own index file that points to the components in their library.

If you're documenting a library of assets, you may need to create some of these directory index files. The next section explains how.

In a future release, this structure will be automatically constructed as part of OpenCPI asset creation.

10.9.1 Documenting a Library of OpenCPI Applications

If you're creating asset documentation for multiple applications in the **`applications/`** directory, you'll need to create a directory index reST file to point to these documents. To create this directory index file:

- Create the template file in the top-level **`applications/`** directory. The resulting template is named **`applications.rst`** and needs no further editing.

Here is an example:

```
.. Application directory index pages


Applications
============
This directory contains applications contained in the OpenCPI
MyProject project.
.. toctree::
   :maxdepth: 2
   :glob:

   */*-application
```

10.9.2 Documenting an OpenCPI Component Library

Each component library needs a directory index file that points to the documentation for the components contained in the library. To create this file:

- Create the template file in the component library, specifying its name as an argument. The resulting template is named **`<library-name>-library.rst`**.
- Replace the "Skeleton outline:" line in the template file with a description of your component library and its scope. The template sets up everything else for you.

Here is an example of an OpenCPI component library directory index file:

```
.. dsp_comps library index page


 DSP components library
 ======================
The DSP component library (``dsp_comps``) provides components for
digital signal processing (DSP).

.. toctree::
   :maxdepth: 1
   :glob:
   :caption: DSP components

   *.comp/*-index
```

10.9.3 Documenting a Directory with Multiple Component Libraries

When your project has multiple component libraries, a component directory index file must exist that allows the asset documentation system to find the different component libraries. To create this directory index file:

- Create the template file in the component directory that contains the multiple libraries. The resulting template is named **`components.rst`** and needs no further editing.

Here is an example:

```
.. Component directory index page


Components
==========
Available components and component libraries.
.. toctree::
   :maxdepth: 2
   :glob:

   */*-library
```

10.9.4 Documenting a Directory of Protocol Specifications

If you're creating asset documentation for multiple protocol specs in the **`specs/`** directory, you'll need to create a directory index reST file to point to these documents. To create the directory index:

- Create the template index file in the top-level **`specs/`** directory that contains the multiple protocol specs. The resulting template is named **`specs.rst`** and needs no further editing.

Here is an example of the reST markup for a specs directory, taken from the directory index file for protocol specs in the OpenCPI built-in platform project:

```
Protocols
=========
This directory contains protocols used throughout the OpenCPI
platform project.
 .. toctree::
  :maxdepth: 1
  :glob:
  :caption: Protocols
  *-protocol
```

10.9.5 Documenting an HDL Primitive Library

Each HDL primitive library needs a directory index reST file that points to the documentation for the HDL primitives contained in the library. To create this file:

- Create the template file in the HDL primitive library, specifying its name as an argument. The resulting template is named **`<library-name>-library.rst`**.
- Replace the “Skeleton outline:” line in the template file with a description of your HDL primitive library and its scope. (Use an incomplete sentence, like the description lines in man pages.) The template sets up everything else for you.

Here is an example:

```
Communication primitive library
===============================
Primitive library for dealing with communication protocols.

.. toctree::
   :maxdepth: 1
   :glob:
   :caption: Primitives

   */*-primitive
```

Note: be sure to adjust the heading line under the primitive library title to be the same length as the protocol title string. If the line is too long or too short, you’ll get a syntax error when you try to build the source file.

10.9.6 Documenting a Directory of HDL Primitive Libraries

If you’re creating asset documentation for multiple HDL primitive libraries in the **`hdl/primitives/`** directory, you’ll need to create a directory index reST file to point to these documents. To create this directory index file:

- Create the template index file in the top-level **`hdl/primitives`** directory. The resulting template is named **`primitives.rst`** and needs no further editing.

Here is an example:

```
Primitives
==========
Available primitives and primitive libraries.

.. toctree::
   :maxdepth: 2
   :glob:

   */*-library
```

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***.