Getting Started Guide

Version 1.5

Revision History

Revision	Description of Change	Date
v1.0	Initial creation	2/2016
v1.1	Updated for OpenCPI Release 1.1	3/2017
v1.2	Updated for OpenCPI Release 1.2	8/2017
v1.3	Updated for OpenCPI Release 1.3	2/2018
v1.4	Updated for OpenCPI Release 1.4	10/2018
v1.5	Updated for OpenCPI Release 1.5	4/2019

Table of Contents

List of Figures

List of Tables

1 References

This document assumes a basic understanding of the Linux command line environment. It does not require a working knowledge of OpenCPI.

Table 1: References

Title	Link
OpenCPI Overview	`Overview.pdf`
Acronyms and Definitions	`Acronyms_and_Definitions.pdf`
Installation Guide	`RPM_Installation_Guide.pdf`
Component Development Guide	`OpenCPI_Component_Development.pdf`
RCC Development Guide	`OpenCPI_RCC_Development.pdf`
HDL Development Guide	`OpenCPI_HDL_Development.pdf`

2 Overview of OpenCPI

Open Component Portability Infrastructure (OpenCPI) is a series of tools and runtime platform for developing and deploying heterogeneous applications. It includes:

- A runtime environment to manage and deploy assets in both software and HDL.
- A set of tools for development of applications and components for software and HDL.
- A framework and methodology for targeting mixed processor architecture types.
- A set of HDL and software building blocks for developers to use and expand.
- A series of reference applications running on base platforms.

Assets and applications developed using OpenCPI are meant to simplify complex integration and improve code portability of heterogeneous solutions. OpenCPI extends component-based architectures into GPPs and FPGAs to decrease development costs and time to market through code portability, reuse, and ease of integration.

For more information, consult the *Overview* found in Table 1.

2.1 Projects Overview

Historically, all OpenCPI development, for both the core team and the end user, has been within a single directory structure. As the team size and user base expand, this quickly becomes untenable, especially when taking version control systems into account. The previous working area has been broken into several locations.

The CDK and the source for the Core and Assets Projects are provided by the ANGRYVIPER Team (*cf.* Section 2.2) in the form of various RPMs that can be installed by a System Administrator. In order to exercise any of the supported platforms, a user must have a writable copy of the Core project in order to build HDL using his/her own specific FPGA tool version(s).

Table 2: Project Types

Name	Contents	Expectation/Usage
CDK	Framework, Utilities	End-user will use, not modified
Core Project	Minimal Components, Primitives, RCC Platforms, HDL Simulator Platforms	End-user will build, not modify
Assets Project	Applications, Components, Primitives, Platforms, Cards/Slots, Devices, Assemblies	End-user will build, run, maybe modify
BSP Project	Platforms, Cards/Slots, Devices, etc.	End-user will build, not modify
User Project	User-provided Components, Applications, etc.	End-user will build, run, modify

2.2 What is the ANGRYVIPER Team?

The ANGRYVIPER Team is a group of engineers contracted to support the OpenCPI framework with additional features, reference assets (known as `ocpi.assets`), additional API, RPM-based modular installation, and an integrated development environment (IDE).

3 A Brief overview OpenCPI's Architecture

In this section, the basics of the framework are discussed including the communications, control, and data exchange. At the highest level, the architecture of the framework is connectivity between 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.
- **Authoring Model:** How components are written in order to be effective on various processing technologies and execution environments.
- **Data Transport:** How messages are moved between one component and another.

The core software 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.

Figure 1 represents the conceptual plug and play functionality OpenCPI brings through the use of extensions. These extensions can take the form of management models, new authoring models and new data technologies.

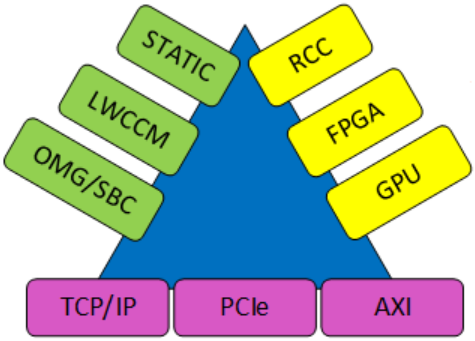

Figure 1: Conceptual extensibility of the OpenCPI framework for new and expanding target bases. Reflecting the three key concepts: Application Management, Authoring Model, and Data Transport.

3.1 Management Models

The basis for the management model is to specify how applications are managed and deployed, including how a control application would statically or dynamically decide to execute one or more component-based applications on a given system. The framework provides two different modes: The first is an application called `ocpirun`, which takes an application XML and control commands to execute. The second is through the use of native C++ API for controlling and launching applications which gives the application developer a greater level of control, including reading and writing properties during execution.

3.2 Authoring Models

The framework uses the concept of "containers" which can host and execute workers. These workers can be hardware-oriented (VHDL within an FPGA) or software-oriented (C/C++ on a GPP[1]).

3.3 Data Transport

The communication model is conceptually based on a protocol model where a component is defined to be able to send or receive messages with defined payloads. The protocol can be as simple as "frames of 200 16-bit unsigned integers". It can also be a variety of messages in a variety of formats based on variable length data types. The transport mechanism can be:

- Passing buffer pointers with no data copying between co-located workers in software.

[1]Standalone CPU or embedded within an FPGA as an SoC

- Directly connecting wires from one FPGA worker to another.
- Moving messages over network sockets (TCP/IP).
- Moving messages between processors using standard buses (PCIe or AXI).

The benefit here is a flexible and transparent methodology for moving data of differing types from worker to worker. This enables a simplified and abstracted development process that greatly removes the hardware implementation from the application.

4 Getting Started

This section will walk through building the Core and Assets Project and all the fundamental steps of creating, building, running, and testing a simple application containing HDL workers. In the *OpenCPI Component Development Guide*, the `ocpidev` command line tool is discussed in detail, but this example will serve as an introduction to using the `ocpidev` command line tool.

4.1 Installation of OpenCPI

The most direct method for installation of the OpenCPI framework and IDE is through the use of the OpenCPI RPMs[2]. The RPMs allows for a normalized installation process with the inclusion of all dependencies required. A detailed installation process can be found in the *RPM Installation Guide*.

The remainder of this document assumes the RPMs have been installed.
The user's environmental variables must also be configured to build with the Xilinx Vivado software. For more information on Vivado installation/licensing and OpenCPI environment configuration, see the *FPGA Vendor Tools Installation Guide*.

4.2 Environmental Variables

Various environmental variables are used to control OpenCPI. When installed, the RPMs provide "sane" defaults for all, with `OCPI_PROJECT_REGISTRY_DIR` and `OCPI_LIBRARY_PATH` often the only ones an end user will need to modify. However, the ones listed in Table 3 are often useful for debugging purposes.

Table 3: Commonly Used Variables

Variable	Description
`OCPI_CDK_DIR`	The location of the CDK's installation. If unset, many scripts and programs will fail to operate. With RPM installation, it is *always* `/opt/opencpi/cdk`.
`OCPI_LIBRARY_PATH`	The set of locations (or projects) used to find runtime artifacts. *Every file within every path* in this colon-separated list is opened and examined for deployment metadata. For this reason, it is best to point to each projects' `exports` subdirectory and not their root location.
`OCPI_LOG_LEVEL`	The amount of logging output by the runtime system. The default is zero (**0**), indicating no logging output. The maximum logging is **20**. Commonly useful startup and diagnostic information (*e.g.* artifact discovery feedback) is provided at log level **8**. Unusual events are logged at level **4**.
`OCPI_PROJECT_PATH`	The set of projects used to find various artifacts and support infrastructure *during build time*. This colon-separated list is legacy (but still supported) and largely replaced by the project registry (detailed below).
`OCPI_PROJECT_REGISTRY_DIR`	Override the default location of the project registry. If this is not set, the default project registry is OCPI_CDK_DIR/../project-registry.
`OCPI_SYSTEM_CONFIG`	The runtime system XML configuration file. Default is `/opt/opencpi/system.xml` with a fallback to `/opt/opencpi/cdk/default-system.xml`.

4.3 Project Registry

A project registry is a directory that contains references to projects in a development environment. By registering a project, a user is publishing his project so it can be referenced/searched by any other project using that same project registry. The default project registry is explained in the OCPI_PROJECT_REGISTRY_DIR row of Table 3. To add a project to the default project registry in an RPM environment, a user needs to be in the `opencpi` user group. This is described in detail the *RPM Installation Guide*.

[2] An alternative "source build" is also available for advanced usage and is documented in the *OpenCPI Installation Guide* – that style of installation is not supported by this document.

The sharing of projects across users with a common registry has been known to be fragile for various reasons (*e.g.* incorrect permission settings, default "umask" values, etc.) and is *not* recommended for new users.

Note: This section is informational / reference; you will create your personal registry below (Section 4.4.1).

The `ocpidev register project` command is used to register a project.

> *To perform this operation within the IDE: (The project "`my_proj`" must be imported into the IDE and then refresh the OpenCPI Projects view so the project is shown.)*
>
> - In the OpenCPI Projects view, select the project, right-click, select "register" from the menu. (Depending on state of the project, this option may not be available.)

This is done automatically when the first copies of the core and assets projects are created. For user-provided projects (or new copies/overrides of the core/assets projects), the `register` command can be used. `unregister` can be used to remove an existing project from the registry.[3]

> *To perform this operation within the IDE: (The project "`my_proj`" must be imported into the IDE and then refresh the OpenCPI Projects view so the project is shown.)*
>
> - In the OpenCPI Projects view, select the project, right-click, select "unregister" from the menu. (Depending on state of the project, this option may not be available.)

The project registry must also be set on a per-project basis using `ocpidev set registry <registry-location>` to "point back" to the registry. See `ocpidev --help set` for more information.

4.4 Set Up Work Environment (Projects and Registry)

4.4.1 Create Registry and Projects

With the `opencpi-devel` rpm installed, a `core` and `assets` project can be created. It is recommended that each user has his/her own copy of the `core` and `assets` projects. Included with the CDK is an installation script, `ocpi-copy-projects`. This script will copy all of the read-only projects out of `/opt/opencpi/projects/` and into another location. The script takes two parameters: destination location for copied projects and the registry location where copied projects are registered (defaulting to `/opt/opencpi/project-registry` or `OCPI_PROJECT_REGISTRY_DIR`). The recommended destination location is `~/ocpi_projects`.

If no arguments are given, the script is interactive and the user is required to input required information. *It is recommended that the user uses interactive mode.* In interactive mode, the user is able to do the following:

1. Unregister any projects from the destination registry if they choose
2. Create a new registry (recommended)
3. Update `.bashrc` to set the environment variable `OCPI_PROJECT_REGISTRY_DIR` to the destination registry

```
% ocpi-copy-projects
```

This command may produce a handful of warnings. It is likely that these warnings can be ignored. If the command actually outputs an Error, it is likely because the user is not a member of the `opencpi` group in an RPM environment with a shared registry.

After execution, you will be reminded that `OCPI_PROJECT_REGISTRY_DIR` *must be set* to use this new registry. You can open a new terminal and ensure it is set if the script edited `.bashrc` for you. If you don't want to at this time, you must at least execute the `export` command given *before continuing or opening the IDE.*

[3]Note that `unregister` does not remove your project, it just delists it from the registry

4.4.2 Display Installed Projects

To confirm which projects are installed and where they live, the command `ocpidev show projects` can be used.

4.4.3 Building Projects

In this step, the Project's contents will be built. Before completing the remaining steps, the environment must be set up as described in the *RPM Installation Guide*.

To build a project, first pick the desired *Platform* to build for. The available options are listed in Table 4.

The relationships between vendor tools and targets (i.e. platforms) is discussed in the `FPGA_Vendor_Tools_Installation_Guide.pdf`

Table 4: Supported Platforms

Simulator Vendor	Platform Name	Project Package-ID [1]	Link
ModelSim DE 10.6a/10.4c	`modelsim`	`ocpi.core`	`https://www.mentor.com/products/fv/modelsim/`
Xilinx Vivado 2017.1	`xsim`	`ocpi.core`	`https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools.html`
Xilinx ISE 14.7	`isim`	`ocpi.core`	`https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/design-tools.html`
Hardware Platform	**Platform Name**	**Project Package-ID**	**Link**
Epiq Solutions Matchstiq-Z1	`matchstiq_z1`[2]	`ocpi.assets`	`https://epiqsolutions.com/matchstiq/`
Avnet ZedBoard (Vivado build)	`zed`[3]	`ocpi.assets`	`http://zedboard.org/`
Avnet ZedBoard (ISE build)	`zed_ise`[4]	`ocpi.assets`	`http://zedboard.org/`
Altera Stratix IV GX	`alst4`	`ocpi.assets`	`https://www.intel.com/content/www/us/en/products/programmable/fpga/stratix-iv.html`
Xilinx ML605 (Virtex-6)	`ml605`	`ocpi.assets`	`https://www.xilinx.com/products/boards-and-kits/ek-v6-ml605-g.html`
Ettus USRP E310	`e3xx`	`ocpi.bsp.e310`	`https://www.ettus.com/product/details/E310-KIT`
Software Platform	**Platform Name**	**Project Package-ID**	**Link**
Centos6	`centos6`	`ocpi.core`	`https://www.centos.org/`
Centos7	`centos7`	`ocpi.core`	`https://www.centos.org/`
Xilinx Linux (xilinx-v2013.4 release)	`xilinx13_3`[5]	`ocpi.core`	`https://github.com/Xilinx/linux-xlnx/releases/tag/xilinx-v2013.4`
Xilinx Linux (xilinx-v14.7 release)	`xilinx13_4`[6]	`ocpi.core`	`https://github.com/Xilinx/linux-xlnx/releases/tag/xilinx-v14.7`

[1]The Project Package-ID is the unique identifier for a project. Please reference the *Component Development Guide* for more details
[2]Reference the Matchstiq-Z1 Getting Started Guide to build for `matchstiq_z1`
[3]Reference the Zedboard Getting Started Guide to build for `zedboard`
[4]Use Xilinx ISE tools to build for this platform
[5]RCC platform associated with the zedboard and matchstiq_z1 platforms
[6]RCC platform associated with the e310 platform

4.4.4 Build Core Project

This section presents a handful of commands that can be used to build for various platforms. To complete this Guide using the `xsim` simulator, use the build command in the box at the end of this section.

> The steps in this section are specific to the `xsim` simulator, which is installed as part as the Xilinx Vivado software. For more information on Vivado installation/licensing and OpenCPI environment configuration, see the *FPGA Vendor Tools Installation Guide*.

Note that all build commands in this Guide can optionally be performed using the ANGRYVIPER IDE instead of "`ocpidev build`".

To build the RCC workers, go to the `core` project and run:

```
# If user will later target the Zedboard or Matchstiq-Z1:
% ocpidev build --rcc --rcc-platform xilinx13_3
```

> *To perform this operation within the IDE:*
>
> 1. Open the ANGRYVIPER Perspective
> 2. Select the asset from OpenCPI Project View
> 3. Import to ANGRYVIPER Operations Panel using ">" button
> 4. Select the RCC and/or HDL platforms for the build (use Ctrl for multiple selection)
> 5. Click "Build"

```
# If user will later target the Zedboard and/or Centos7:
% ocpidev build --rcc --rcc-platform xilinx13_3 --rcc-platform centos7
```

The `--rcc-platform` option specifies the RCC platform to the build for.

Use the following `ocpidev` command to build the project for the desired platform using the information from Table 4.

```
% ocpidev build --rcc-platform <platform name> --hdl-platform <platform name>
```

Building for multiple platforms can be combined into a single `ocpidev` call, *e.g.*:

```
% ocpidev build --hdl-platform modelsim --hdl-platform xsim --hdl-platform isim --rcc-platform centos7
--rcc-platform xilinx13_3
```

In the `core` project, the last operation this command performs is to build the project's `hdl platforms`. You can be sure it succeeded if the last few sections of output are "`=======Building platform xsim`".

> For the example in Section 5, you *must* build for `xsim` at a minimum. This takes approximately 10 minutes:
>
> ```
> ocpidev build --hdl-platform xsim
> ```
>
> *Note:* If you have ISE installed instead of Vivado, you can follow this guide using `isim` instead of `xsim`

4.4.5 Build Assets Project

Refer to table 4 to determine whether the platform being built for lives in the `ocpi.assets` project. If so, follow the same building procedures documented in Section 4.4.3 for the same platforms. The last assets built in the `assets` project are `applications`. You can be sure it succeeded if the last few sections of output include "`========Building apps cic_int_dc_offset_iq_imbalance_mixer_cic_dec`".

If the `xsim` platform is being used, the `ocpi.assets` project does *not* need to be built at this time.

5 Basic Example Application

This section provides a *basic* example application that the user can create. There are more detailed examples and presentation slides available as "Standalone Training" on `github.io`.

This simple application will contain three HDL Workers: `ramp`, `square`, and `ander`, which can be seen in Figure 2.

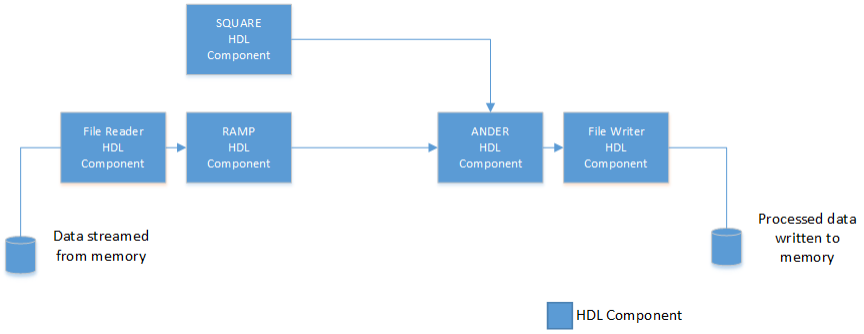

Figure 2: Block diagram of simple HDL application with three user-defined workers, a `FileRead`, and a `FileWrite`.

For testing and simulation purposes, the OpenCPI `file_read` and `file_write` workers will be used to read input data from, and write output data to, files on the system.

The basic format of an `ocpidev` command is "`ocpidev [options] <verb> <noun> <name>`". For this demonstration, the following `nouns` will be used: `project`, `library`, `spec`, `worker`, `hdl assembly`, and `application`. `ocpidev` supports "tab-completion" for *most* options, including platform names. It is recommended that the user take advantage of this capability when performing these examples to help get a feel for the options available from the tool.

The full list of `ocpidev` verbs, nouns and options can be explored via `ocpidev --help`, `ocpidev --help <verb>`, or the *OpenCPI Component Development Guide*.

5.1 Create a Project

Choose the name "DemoProject" for this project. To create the "DemoProject" project use the following `ocpidev` command:

```
% ocpidev create project --register DemoProject
```

To perform this operation within the IDE:

- Place the cursor in the OpenCPI Projects panel, right click, select asset wizard.
- Select the asset type ("Project") in the drop-down, fill in the required inputs, click finish.
- When the process finishes, the new asset is displayed in both project views. (If the asset has an XML editor, then the editor opens.)

The project that was just created should have also been registered. The project registration can be done separately using the following `ocpidev` command from within the project:

```
% ocpidev register project
```

To perform this operation within the IDE: (The project "`DemoProject`" must be imported into the IDE and then refresh the OpenCPI Projects view so the project is shown.)

- In the OpenCPI Projects view, select the project, right-click, select "register" from the menu. (Depending on state of the project, this option may not be available.)

Observe the directory structure created, as well as the files.

```
$ tree --charset ascii DemoProject
|-- exports
|   |-- imports -> ../imports
|   `-- project-package-id
|-- imports -> /opt/opencpi/project-registry
|-- Makefile
|-- Project.exports
|-- Project.mk
`-- project.xml
3 directories, 5 files
```

Change directories into "DemoProject".

5.2 Create a Library

A library is a convenient way of grouping workers. For a simple case, the framework defaults to a `components` directory as the library. To create the default "components" library, use the following `ocpidev` command:

```
% ocpidev create library components
```

To perform this operation within the IDE:

- Place the cursor in the OpenCPI Projects panel, right click, select asset wizard.
- Select the asset type in the drop-down, fill in the required inputs, click finish.
- When the process finishes, the new asset is displayed in both project views. (If the asset has an XML editor, then the editor opens.)

Observe the directory structure created, as well as the files:

```
DemoProject
|-- components
|   |-- lib
|   |   |-- package-id
|   |   `-- workers
|   |-- Library.mk
|   `-- Makefile
|-- exports
|   |-- imports -> ../imports
|   `-- project-package-id
|-- imports -> /opt/opencpi/project-registry
|-- Makefile
|-- Project.exports
|-- Project.mk
`-- project.xml
```

For more advanced projects, one or more libraries would be placed within the `components` directory. This usage is highly recommended and explained in the *OpenCPI Component Development Guide*.

5.3 Create Components

Components are black boxes that dictate what properties and interfaces are implemented. A single implementation of a *component* is referred to as a *worker*, which is discussed in the next section. *Components* are defined by an OpenCPI Component Specification (OCS). The OCS is commonly referred to as the *spec file* or *spec*. Each *spec* includes the details that are to be **consistent among different authoring models**; meaning that if there are various implementations of a single *component*, *e.g.* a software (RCC) worker and an HDL worker, both *workers* expect the same input and both should have the same output regardless of which architecture the worker runs on.

This application requires three *components*, so three *specs* will be generated. To create the three template *specs* use the following `ocpidev` commands:

```
% ocpidev create component ramp
% ocpidev create component square
% ocpidev create component ander
```

> *To perform this operation within the IDE:*
>
> - Place the cursor in the OpenCPI Projects panel, right click, select asset wizard.
> - Select the asset type in the drop-down, fill in the required inputs, click finish.
> - When the process finishes, the new asset is displayed in both project views. (If the asset has an XML editor, then the editor opens.)

```
DemoProject
|-- components
|   |-- lib
|   |   |-- ander-spec.xml -> ../specs/ander-spec.xml
|   |   |-- package-id
|   |   |-- ramp-spec.xml -> ../specs/ramp-spec.xml
|   |   |-- square-spec.xml -> ../specs/square-spec.xml
|   |   `-- workers
|   |-- Library.mk
|   |-- Makefile
|   `-- specs
|       |-- ander-spec.xml
|       |-- ramp-spec.xml
|       `-- square-spec.xml
|-- exports
|   |-- imports -> ../imports
|   `-- project-package-id
|-- imports -> /opt/opencpi/project-registry
|-- Makefile
|-- Project.exports
|-- Project.mk
`-- project
```

Notice that three *spec* templates were generated. Also note that *spec files* can be easily identified by their `-spec` postfix[4]. In between the `ComponentSpec` XML tags, the *component* properties and interfaces need to be defined. **Every component interface (or *port*) must define the direction and format of the messages they will send or receive.** This formatting is known as the *protocol*, and for this example, we will use `rstream`, which is a stream of up to 4096 16-bit samples per message.

For the `ramp` *component*, insert the following XML snippet in between the `ComponentSpec` XML tags:

```
<Port Name="in" Producer="false" Protocol="rstream_protocol.xml"/>
<Port Name="out" Producer="true" Protocol="rstream_protocol.xml"/>
```

[4]Some older spec files use `_spec` as well.

For the `square` *component*, insert the following XML snippet in between the `ComponentSpec` XML tags:

```
<Port Name="out" Producer="true" Protocol="rstream_protocol.xml"/>
```

For the `ander` *component*, insert the following XML snippet in between the `ComponentSpec` XML tags:

```
<Port Name="in1" Producer="false" Protocol="rstream_protocol.xml"/>
<Port Name="in2" Producer="false" Protocol="rstream_protocol.xml"/>
<Port Name="out" Producer="true" Protocol="iqstream_protocol.xml"/>
```

For more details, see the *OpenCPI Component Development Guide*. Now that the *specs* are defined, the next step is to create the *workers*.

5.4 Create Workers

As mentioned in the previous section, *workers* are specific implementations of *components*. More than one *worker* can implement a *component*. For this example, only one implementation per *component* will be used, and each of these *workers* are *HDL Workers*. More details about *HDL Workers* can be found in the *OpenCPI HDL Development Guide*. This example does not use any *RCC (Software) Workers*, but more details about them can be found in the *OpenCPI RCC Development Guide*.

When creating *workers*, two options need to be defined, one implicitly and one explicitly. The type of *worker* is defined implicitly by appending either `.rcc` or `.hdl` to the name of the *worker*. *HDL Workers* use the general format for the *worker* names: `worker_name.hdl`. The language of the *worker* can be explicitly defined; for *RCC Workers* there are currently two choices: C++ or C.

To create the *HDL Workers*, use the following `ocpidev` commands:

```
% ocpidev create worker ramp.hdl
% ocpidev create worker square.hdl
% ocpidev create worker ander.hdl
```

> *To perform this operation within the IDE:*
>
> - Place the cursor in the OpenCPI Projects panel, right click, select asset wizard.
> - Select the asset type in the drop-down, fill in the required inputs, click finish.
> - When the process finishes, the new asset is displayed in both project views. (If the asset has an XML editor, then the editor opens.)

Notice that the creation of the `ramp`, `square`, and `ander` *HDL Workers* generated a `.hdl` directory for each. Each of the `.hdl` directories contain that *worker*'s *OpenCPI Worker Description* (OWD), which can be identified by the following format: `worker_name.xml`. The OWD is where that *worker's* specific properties and port protocols are defined. The definitions in the OWD are specific to each individual worker and may override a set of attributes defined in the OCS. Later the OWD will be edited for each *worker*.

Observe that in each of the `.hdl` directories there is also the *skeleton file* or *wrapper* in the language that was specified. In this case, the `worker_name.vhd` is the *skeleton file* for the language option VHDL. Later, the *skeleton file* will be edited for each *worker*.

The `gen` directory contains other important code that is generated from the framework and will not need to be edited. For more details see the *OpenCPI Component Guide*.

The project should now look similar to the following:

```
DemoProject/
|-- components
|   |-- ander.hdl
|   |   |-- ander.vhd
|   |   |-- ander.xml
|   |   |-- gen
|   |   |   |-- ander-build.xml
|   |   |   |-- ander-defs.vhd
|   |   |   |-- ander-defs.vhd.deps
|   |   |   |-- ander-impl.vhd
|   |   |   |-- ander-impl.vhd.deps
|   |   |   |-- ander.mk
|   |   |   |-- ander-skel.vhd
|   |   |   `-- ander-skel.vhd.deps
|   |   `-- Makefile
|   |-- lib
|   |   |-- ...
|   |   `-- workers
...
|   |-- ramp.hdl
|   |   |-- gen
|   |   |   |-- ramp-build.xml
|   |   |   |-- ramp-defs.vhd
|   |   |   |-- ramp-defs.vhd.deps
|   |   |   |-- ramp-impl.vhd
|   |   |   |-- ramp-impl.vhd.deps
|   |   |   |-- ramp.mk
|   |   |   |-- ramp-skel.vhd
|   |   |   `-- ramp-skel.vhd.deps
|   |   |-- Makefile
|   |   |-- ramp.vhd
|   |   `-- ramp.xml
|   |-- specs
|   |   |-- ander-spec.xml
|   |   |-- ramp-spec.xml
|   |   `-- square-spec.xml
|   `-- square.hdl
|       |-- gen
|       |   |-- square-build.xml
|       |   |-- square-defs.vhd
|       |   |-- square-defs.vhd.deps
|       |   |-- square-impl.vhd
|       |   |-- square-impl.vhd.deps
|       |   |-- square.mk
|       |   |-- square-skel.vhd
|       |   `-- square-skel.vhd.deps
|       |-- Makefile
|       |-- square.vhd
|       `-- square.xml
...
```

However, if you are using version control[a], you can determine the key files:

First, navigate to the directory above DemoProject

```
% ocpidev clean project DemoProject
```

> *To perform this operation within the IDE:* In the OpenCPI Projects view, select the project, right-click, select clean from the menu.

```
% tree --charset ascii DemoProject/
DemoProject/
|-- components
|   |-- ander.hdl
|   |   |-- ander.xml
|   |   `-- Makefile
|   |-- lib
|   |   |-- ander-spec.xml -> ../specs/ander-spec.xml
|   |   |-- package-id
|   |   |-- ramp-spec.xml -> ../specs/ramp-spec.xml
|   |   |-- square-spec.xml -> ../specs/square-spec.xml
|   |   `-- workers
|   |-- Library.mk
|   |-- Makefile
|   |-- ramp.hdl
|   |   |-- Makefile
|   |   `-- ramp.xml
|   |-- specs
|   |   |-- ander-spec.xml
|   |   |-- ramp-spec.xml
|   |   `-- square-spec.xml
|   `-- square.hdl
|       |-- Makefile
|       `-- square.xml
|-- exports
|   |-- imports -> ../imports
|   |-- lib
|   |   `-- components -> ../../components/lib
|   `-- project-package-id
|-- imports -> /opt/opencpi/project-registry
|-- Makefile
|-- Project.exports
|-- Project.mk
`-- project.xml


11 directories, 21 files
```

[a] The `lib`, `imports`, and `exports` should not be in SCM.

Now the OWD files for each *worker* will be updated. This will provide implementation-specific information for the interfaces that were abstractly defined at the *component* level in the OCS.

The version 2 HDL worker API was introduced in OpenCPI v1.5. It is the recommended interface for future designs. All of the workers in this guide will use this API. For more information on the differences between version 1 and 2 HDL worker APIs, see the OpenCPI HDL Development Guide and the release notes for OpenCPI v1.5.

For the `ramp` *worker*, replace the worker OWD (in `components/ramp.hdl/ramp.xml`) with this XML:

```
<HdlWorker language="vhdl" spec="ramp-spec" version="2">
  <StreamInterface Name="in" DataWidth="16"/>
  <StreamInterface Name="out" DataWidth="16" InsertEOM="true"/>
</HdlWorker>
```

For the `square` *worker*, replace the worker OWD (in `components/square.hdl/square.xml`) with this XML:

```
<HdlWorker language="vhdl" spec="square-spec" version="2">
  <StreamInterface Name="out" DataWidth="16" InsertEOM="true"/>
</HdlWorker>
```

For the `ander` *worker*, replace the worker OWD (in `components/ander.hdl/ander.xml`) with this XML:

```
<HdlWorker language="vhdl" spec="ander-spec" version="2">
  <StreamInterface Name="in1" DataWidth="16"/>
  <StreamInterface Name="in2" DataWidth="16"/>
  <StreamInterface Name="out" DataWidth="32" InsertEOM="true"/>
</HdlWorker>
```

For more details, see the *OpenCPI Component Development Guide*. Now that the OWDs are defined, the next step is to edit the VHDL *skeleton files*.

At this point, if you did the "`ocpidev clean`" example above, you need to regenerate the example code:

- `ocpidev build worker ramp.hdl`
- `ocpidev build worker square.hdl`
- `ocpidev build worker ander.hdl`

Replace the `ramp.vhd` *skeleton file* (`components/ramp.hdl/ramp.vhd`) with the following VHDL:

```
library IEEE; use IEEE.std_logic_1164.all; use ieee.numeric_std.all;
library ocpi; use ocpi.types.all; -- remove this to avoid all ocpi name collisions
architecture rtl of worker is
  signal do_work : bool_t;
  signal out_data_i, buff_data : std_logic_vector(15 downto 0);
begin
  -- When we are allowed to process data:
  do_work <= out_in.ready and in_in.valid;
  -- Outputs:
  in_out.take <= do_work;
  out_out.valid <= do_work;
  out_data_i <= std_logic_vector(signed(in_in.data) + signed(buff_data));
  out_out.data <= out_data_i;

  -- Initialize or save off previous value when valid:
  ramp : process(ctl_in.clk)
  begin
    if rising_edge(ctl_in.clk) then
      if ctl_in.reset = '1' then
        buff_data <= (others => '0');
      elsif its(do_work) then
        buff_data <= out_data_i;
      end if;
    end if;
  end process ramp;
end rtl;
```

Replace the `square.vhd` *skeleton file* (`components/square.hdl/square.vhd`) with the following VHDL:

```
library IEEE; use IEEE.std_logic_1164.all; use ieee.numeric_std.all;
library ocpi; use ocpi.types.all; -- remove this to avoid all ocpi name collisions
architecture rtl of worker is
  signal do_work : bool_t;
  signal cnt : unsigned(7 downto 0);
begin
  -- When we are allowed to process data:
  do_work <= out_in.ready;
  -- Outputs:
  out_out.data <= (others => '1') when cnt < 32 else
                  (others => '0');
  out_out.valid <= do_work;

  -- Generate the square pulse's counter
  square : process(ctl_in.clk)
  begin
    if rising_edge(ctl_in.clk) then
      if ctl_in.reset = '1' then
        cnt <= (others => '0');
      elsif its(do_work) then -- advance when we are pushing
        cnt <= cnt + 1;
        if cnt = 63 then
          cnt <= (others => '0');
        end if;
      end if;
    end if;
  end process square;
end rtl;
```

Replace the `ander.vhd` *skeleton file* (`components/ander.hdl/ander.vhd`) with the following VHDL:

```
library IEEE; use IEEE.std_logic_1164.all; use ieee.numeric_std.all;
library ocpi; use ocpi.types.all; -- remove this to avoid all ocpi name collisions
architecture rtl of worker is
  signal do_work : bool_t;
begin
  -- When we are allowed to process:
  do_work <= out_in.ready and in1_in.valid and in2_in.valid;
  -- Outputs:
  in1_out.take <= do_work;
  in2_out.take <= do_work;
  out_out.valid <= do_work;
  out_out.data(15 downto  0) <= in1_in.data and in2_in.data;
  out_out.data(31 downto 16) <= in1_in.data;
end rtl;
```

The *workers* must be built at this time using the following `ocpidev` command from the `DemoProject` directory:

```
% ocpidev build --hdl-platform xsim
```

To perform this operation within the IDE:

1. Open the ANGRYVIPER Perspective
2. Select the asset from OpenCPI Project View
3. Import to ANGRYVIPER Operations Panel using ">" button
4. Select the RCC and/or HDL platforms for the build (use Ctrl for multiple selection)
5. Click "Build"

5.5 Create an HDL Assembly

Before an application can be made, an *assembly* for the *HDL workers* needs to be created. An *HDL assembly* is a synthesized netlist of connected application workers. For this example, use `demo_assembly` as the name of the assemblies directory for the application. From the `DemoProject` directory, run the following `ocpidev` command to create the assembly:

```
% ocpidev create hdl assembly demo_assembly
```

To perform this operation within the IDE:

- Place the cursor in the OpenCPI Projects panel, right click, select asset wizard.
- Select the asset type in the drop-down, fill in the required inputs, click finish.
- When the process finishes, the new asset is displayed in both project views. (If the asset has an XML editor, then the editor opens.)

Notice that this command produces the `hdl` directory, `assemblies` directory within it, `demo_assembly` directory within that, and the *OpenCPI HDL Assembly* (OHAD) (`demo_assembly.xml`).

```
DemoProject/hdl/
|-- assemblies
|   |-- demo_assembly
|   |   |-- demo_assembly.xml
|   |   `-- Makefile
|   `-- Makefile
`-- Makefile
```

Navigate to the `demo_assembly` directory and replace `demo_assembly.xml` with this XML:

```
<HdlAssembly>
  <Instance Worker="file_read" Connect="ramp"/>
  <Instance Worker="ramp"/>
  <Instance Worker="square"/>
  <Instance Worker="ander" Connect="file_write"/>
  <Instance Worker="file_write"/>
  <Connection>
    <Port Instance="ramp" Name="out"/>
    <Port Instance="ander" Name="in1"/>
  </Connection>
  <Connection>
    <Port Instance="square" Name="out"/>
    <Port Instance="ander" Name="in2"/>
  </Connection>
</HdlAssembly>
```

Now the assembly for this application is complete. To build the HDL assembly, run the following command:

```
% ocpidev build --hdl-platform xsim
```

To perform this operation within the IDE:

1. Open the ANGRYVIPER Perspective
2. Select the asset from OpenCPI Project View
3. Import to ANGRYVIPER Operations Panel using ">" button
4. Select the RCC and/or HDL platforms for the build (use Ctrl for multiple selection)
5. Click "Build"

This will take a couple of minutes to run. You can confirm that it succeeded by locating the `*.bitz` file in `hdl/assemblies/demo_assembly/container-demo_assembly_xsim_base/target-xsim/`.

5.6 Create an Application

One of the simplest ways to make an application is to use the `-X` option of `ocpidev`. This flag will create a "simple" *OpenCPI Application Specification* (OAS) in the `applications` directory. In this case, it will also create the `applications` directory, since it does not yet exist. For the example, choose the name `DemoApp` for the name of the application. Run the following `ocpidev` command from the `DemoProject` directory to generate the application:

```
% ocpidev -X create application DemoApp
```

To perform this operation within the IDE:

- Place the cursor in the OpenCPI Projects panel, right click, select asset wizard.
- Select the asset type in the drop-down, fill in the required inputs, click finish.
- When the process finishes, the new asset is displayed in both project views. (If the asset has an XML editor, then the editor opens.)

Notice that this command generated the applications directory as well as the `DemoApp.xml`.

```
DemoProject
|-- applications
|   |-- DemoApp.xml
|   `-- Makefile
...
```

There are two things to keep in mind while using this demo application. One is that the property `filename` in the components `file_read` and `file_write`. The `fileName Value` defines where the `file_read` component will look for input data into the application and where the `file_write` component will write the output data out of the application.

Navigate to the applications directory and create the two directories mentioned in the `file_read` and `file_write` component instance:

```
% mkdir idata odata
```

In order to complete the OAS, replace `applications/DemoApp.xml` with this XML:

```
<Application>
  <Instance Component="ocpi.core.file_read" Connect="ramp">
    <Property Name="fileName" Value="idata/input_file.bin"/>
  </Instance>
  <Instance Component="local.DemoProject.ramp"/>
  <Instance Component="local.DemoProject.square"/>
  <Instance Component="local.DemoProject.ander" Connect="file_write"/>
  <Instance Component="ocpi.core.file_write">
    <Property Name="fileName" Value="odata/output_file.bin"/>
  </Instance>
  <Connection>
    <Port Instance="ramp" Name="out"/>
    <Port Instance="ander" Name="in1"/>
  </Connection>
  <Connection>
    <Port Instance="square" Name="out"/>
    <Port Instance="ander" Name="in2"/>
  </Connection>
</Application>
```

In order to simulate the application, input data needs to be generated to drive the application. The next section will focus on generating input data for this application.

5.7 Generate Input Data

The `file_read` component will search the `idata` directory for `input_file.bin` to drive the application. For this example, a simple `Python` script is provided to generate the expected input. This application is very simple and the `ramp` component could generate data internally, but in order to have a more complete example, the `ramp` *worker* was designed to depend on externally generated data.

Create a file `generate_input.py` in the `applications/idata` directory and insert the following `Python` code into the file.

```
#!/usr/bin/env python2
import numpy as np
import sys

# verify input arg count
if len(sys.argv) < 4:
    print("Usage expected:\n\t"+sys.argv[0]+" filename val len\n")
    sys.exit(1)

# create array of length "len" filled with "val"
data = np.empty(int(sys.argv[3]), dtype=np.int16)
data.fill(sys.argv[2])

# write data to output file
data.tofile(sys.argv[1])
```

To generate the expected input, run the following command from the `idata` directory:

```
% python generate_input.py input_file.bin 128 2000
```

The first argument into the script is the output file. The second argument is the value at which the `ramp` will accumulate by. The last argument is simply the number of values written to the file. This means the `ramp` will end up accumulating 128 (or 0x80) 2,000 times.

5.8 Run Simulation

To run the simulation, navigate to the `applications` directory and run the following commands:

```
# OCPI_LIBRARY_PATH provides a list of locations for the framework to search for built
# RCC and HDL artifacts. For this example, make's default OCPI_LIBRARY_PATH is
# sufficient, so the variable can be unset.
% unset OCPI_LIBRARY_PATH
% make run OcpiRunArgs="-d -t 1"
```

The simulation will run for one second of runtime ("`-t 1`") and write the output to a file in `odata/output_file.bin`. This `make` command uses `ocpirun` to run the application. For more details on `ocpirun` and running applications, see the *OpenCPI Application Development Guide*.

5.9 Examine the Output

To observe the output, another `Python` script is provided which will demultiplex and plot the data.

Create a file `plot_output.py` in the `odata` directory and insert the following `Python` code into the file:

```
#!/usr/bin/env python2
import numpy as np
import matplotlib.pyplot as plt
import sys

# verify input args
if len(sys.argv) < 2:
    print("Need data file name, ex:\n\t"+sys.argv[0]+" filename\n")
    sys.exit(1)

# read data from input file
data=np.fromfile(sys.argv[1],dtype=np.int16)

# demultiplex data, odds to the upper 16-bits, evens to the lower 16-bits
upper16=data[1::2]
lower16=data[0::2]

# plot the upper 16-bits
plt.figure(1)
plt.plot(upper16)
plt.title("Output Data - Upper 16")
plt.grid()

# plot the lower 16-bits
plt.figure(2)
plt.plot(lower16)
plt.title("Output Data - Lower 16")
plt.grid()

plt.show()
```

To plot the generated output, run the following command from the `odata` directory:

```
% python plot_output.py output_file.bin
```

Figures 3 and 4 show the upper and lower 16 bits of the `ander` output. The upper 16 bits are the output of `ramp`, passed through for display. The lower 16 bits are the result of "anding" this input with a square wave from `square`.

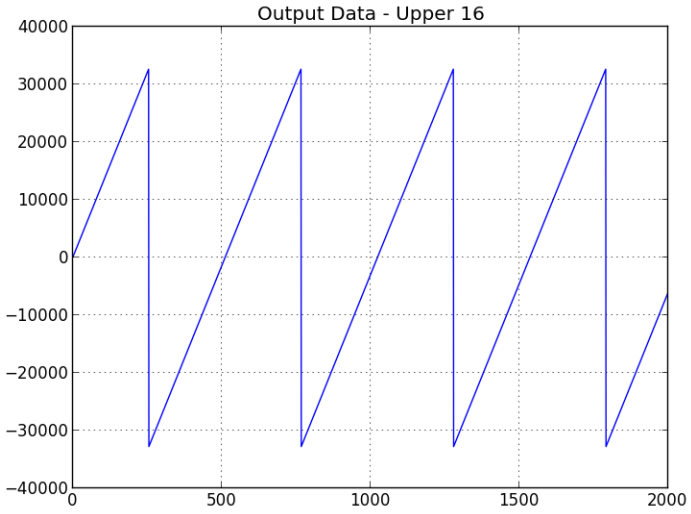

Figure 3: "RAMP" Output (Passed through by "ANDER")

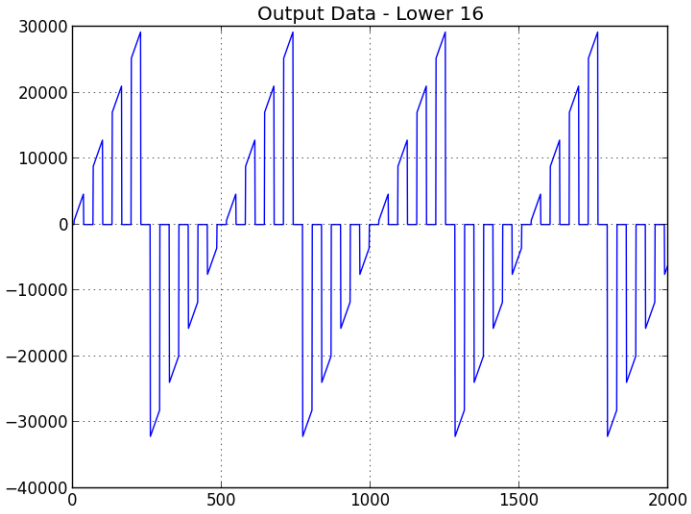

Figure 4: "ANDER" Output

For completeness, the output plot of the `square` *component* is provided in Figure 5. The steps to generate a unit test for the `square` *component* are outside the scope of this document.

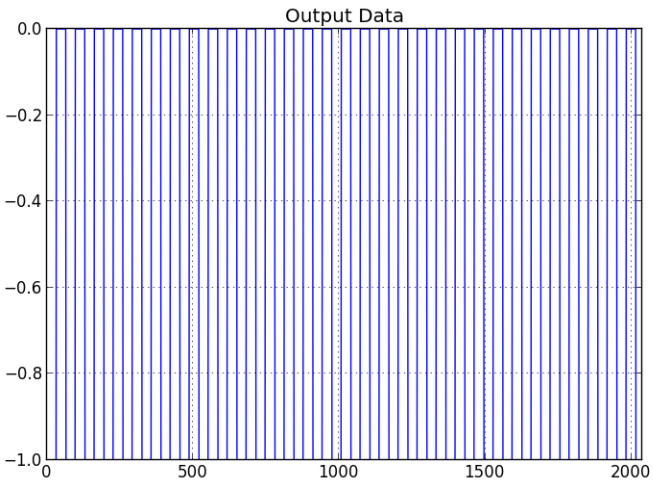

Figure 5: "SQUARE" Output = Input 2 to "ANDER"

5.10 Adding Backpressure

For more realistic testing, the "`backpressure`" component can simulate downstream components being "too busy" to receive data. By default, all unit tests of a *single* component includes backpressure to assist development (see the *Component Development Guide*). This section adds the `backpressure` component between the `ander` and `file_write`. If `ander` properly implements backpressure, it will indicate to `ramp` and `square` that they should not push more data to it. If they properly handle backpressure, they will halt their generation routines and avoid a discontinuity in their output.

New Assembly XML

Replace the existing assembly XML (from Section 5.5) with the following code:

```
<HdlAssembly>
<Instance Worker="file_read" Connect="ramp"/>
  <Instance Worker="ramp" />
  <Instance Worker="square"/>
  <Instance Worker="ander" Connect="backpressure"/>
  <Instance Worker="backpressure" Connect="file_write"/>
  <Instance Worker="file_write"/>
  <Connection>
    <Port Instance="ramp" Name="out"/>
    <Port Instance="ander" Name="in1"/>
  </Connection>
  <Connection>
    <Port Instance="square" Name="out"/>
    <Port Instance="ander" Name="in2"/>
  </Connection>
</HdlAssembly>
```

New Application XML

Replace the existing application XML (from Section 5.6) with the following code:

```
<Application>
  <Instance Component="ocpi.core.file_read" Connect="ramp">
    <Property Name="fileName" Value="idata/input_file.bin"/>
  </Instance>
  <Instance Component="local.DemoProject.ramp"/>
  <Instance Component="local.DemoProject.square"/>
  <Instance Component="local.DemoProject.ander" Connect="backpressure"/>
  <Instance Component="ocpi.core.backpressure" Connect="file_write">
    <property name='enable_select' value='true'/>
  </Instance>
  <Instance Component="ocpi.core.file_write">
    <Property Name="fileName" Value="odata/output_file.bin"/>
  </Instance>
  <Connection>
    <Port Instance="ramp" Name="out"/>
    <Port Instance="ander" Name="in1"/>
  </Connection>
  <Connection>
    <Port Instance="square" Name="out"/>
    <Port Instance="ander" Name="in2"/>
  </Connection>
</Application>
```

Results

Once the workers (Section 5.4) and assemblies (Section 5.5) are rebuilt, the application's results (Sections 5.8 and 5.9) should be exactly as the output *without* backpressure. If not, the HDL workers will most likely drop data when deployed to a non-simulation platform.