

Getting Started Guide

Version 1.3

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 |

Table of Contents

| | | |
|----------|--|-----------|
| 1 | References | 6 |
| 2 | What is the ANGRYVIPER Team? | 7 |
| 2.1 | Overview of OpenCPI | 7 |
| 2.2 | Projects Overview | 7 |
| 3 | A Brief overview OpenCPI's Architecture | 8 |
| 3.1 | Management Models | 8 |
| 3.2 | Authoring Models | 8 |
| 3.3 | Data Transport | 8 |
| 4 | Getting Started | 10 |
| 4.1 | Installation of OpenCPI | 10 |
| 4.2 | Environmental Variables | 10 |
| 4.3 | Project Registry | 10 |
| 4.4 | Create and Build the Core and Assets Project | 11 |
| 4.4.1 | Create Core | 11 |
| 4.4.2 | Create Assets | 11 |
| 4.4.3 | Display Installed Projects | 11 |
| 4.4.4 | Building Projects | 11 |
| 4.4.5 | Build Core Project | 12 |
| 4.4.6 | Build Assets Project | 13 |
| 5 | Basic Example Application | 14 |
| 5.1 | Create a Project | 14 |
| 5.2 | Create a Library | 15 |
| 5.3 | Create Components | 15 |
| 5.4 | Create Workers | 16 |
| 5.5 | Create an HDL Assembly | 23 |
| 5.6 | Create an Application | 24 |
| 5.7 | Generate Input Data | 25 |
| 5.8 | Run Simulation | 26 |
| 5.9 | Examine the Output | 26 |

List of Figures

| | | |
|---|---|----|
| 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. | 8 |
| 2 | Block diagram of simple HDL application with three user-defined workers, a <code>FileRead</code> , and a <code>FileWrite</code> | 14 |
| 3 | “RAMP” Output (Passed through by “ANDER”) | 27 |
| 4 | “ANDER” Output | 27 |
| 5 | “SQUARE” Output = Input 2 to “ANDER” | 28 |

List of Tables

| | | |
|---|-----------------------------------|----|
| 1 | References | 6 |
| 2 | Project Types | 7 |
| 3 | Commonly Used Variables | 10 |
| 4 | Supported Platforms | 12 |

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 | Published By | Link |
|-----------------------------|---------------------|---|
| Installation Guide | ANGRYVIPER Team | RPM_Installation_Guide.pdf |
| Acronyms and Definitions | ANGRYVIPER Team | Acronyms_and_Definitions.pdf |
| Overview | OpenCPI | https://goo.gl/RskxiV |
| Component Development Guide | OpenCPI | https://goo.gl/zBwIe0 |
| RCC Development Guide | OpenCPI | https://goo.gl/0ix1E0 |
| HDL Development Guide | OpenCPI | https://goo.gl/OVmRhI |

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).

2.1 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.

2.2 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 in the form of various RPMs that can be installed by a System Administrator. A version of the Core Project is provided containing prebuilt software necessary for all OpenCPI execution (and optionally development). In order to exercise any of the supported FPGA 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 |

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 AV brings through the use of extensions. These extension can take the form of management models, new authoring models and new data technologies.

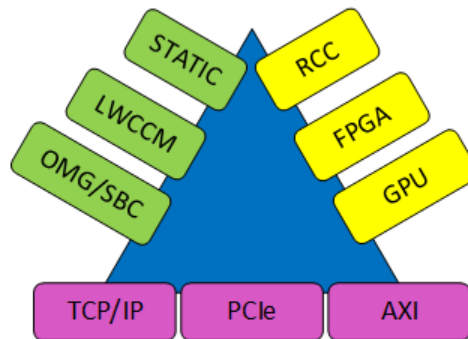


Figure 1: Conceptual extensibility of the OpenCPI framework for new and expanding target bases. Reflecting the three key concepts: Application Management, Authoring Model, and Data Transport.

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 GPP¹).

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 collocated workers in software.

¹Standalone or within an FPGA

- 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 Base 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. 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.

4.2 Environmental Variables

Various environmental variables are used to control OpenCPI. When installed, the RPMs provide “sane” defaults for all, and `OCPI_LIBRARY_PATH` is often the only one 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

| Main RPMs | Description |
|--|--|
| <code>OCPI_CDK_DIR</code> | The location of the CDK's installation. If unset, many scripts and programs will fail to operate. With RPM installation, it is <i>always</i> <code>/opt/opencpi/cdk</code> . |
| <code>OCPI_LIBRARY_PATH</code> | The set of locations (or projects) used to find runtime artifacts. <i>Every file within every path</i> in this colon-separated list is opened and examined for deployment metadata. For this reason, it is best to point to each projects' <code>exports</code> subdirectory and not their root location. |
| <code>OCPI_LOG_LEVEL</code> | 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 (<i>e.g.</i> artifact discovery feedback) is provided at log level 8 . Unusual events are logged at level 4 . |
| <code>OCPI_PROJECT_PATH</code> | The set of projects used to find various artifacts and support infrastructure <i>during build time</i> . This colon-separated list is legacy (but still supported) and largely replaced by the project registry (detailed below). |
| <code>OCPI_PROJECT_REGISTRY_DIR</code> | Override the default location of the project registry. If this is not set, the default project registry is <code>OCPI_CDK_DIR/./project-registry</code> . |
| <code>OCPI_SYSTEM_CONFIG</code> | The runtime system XML configuration file. Default is <code>/opt/opencpi/system.xml</code> with a fallback to <code>/opt/opencpi/cdk/default-system.xml</code> . |

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 user or project using that same project registry. The default project registry is explained in the `OCPI_PROJECT_REGISTRY_DIR` row of in table 3.

To add a project to the default project registry, a user needs to be in the `opencpi` user group. This is described in detail the *RPM Installation Guide*. The `ocpidev register project` command is used to register a project. 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.²

²Note that `unregister` does not remove your project, it just delists it from the registry

The project registry can also be set on a per-project basis using `ocpidev set registry <registry-location>`. See `ocpidev --help set` for more information.

4.4 Create and Build the Core and Assets Project

4.4.1 Create Core

With the `opencpi-devel` rpm installed, a `core` project can be created. It is recommended that each user has his/her own copy of the `core` project. Included with the CDK is an installation script, `/opt/opencpi/projects/new_project_source`. This script will copy the read-only `core` project out of `/opt/opencpi/projects/core` and into another location. The script takes two parameters: the project being copied and the destination folder, `~/ocpi_coreproject` is the recommended location. Before running this script, make sure the user is a member of the `opencpi` group as explained in the *RPM Installation Guide*.

```
% /opt/opencpi/projects/new_project_source core <path>/<name>
```

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.

4.4.2 Create Assets

With the `opencpi-project-assets` RPM present, a copy of the `assets` project can be installed. It is recommended that each user has his/her own copy of `assets`. Included with the CDK is an installation script, `/opt/opencpi/projects/new_project_source`. This script will copy the template `assets` project out of `/opt/opencpi/projects/assets` and into another location. The script takes two parameters the project being copied and the destination folder, `~/ocpi_assetsproject` is the recommended location.

```
% /opt/opencpi/projects/new_project_source assets <path>/<name>
```

4.4.3 Display Installed Projects

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

4.4.4 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.

Table 4: Supported Platforms

| Simulator Vendor | Platform Name | Project |
|---------------------------------|---------------------------|-------------|
| ModelSim DE 10.6a/10.4c | modelsim | ocpi.core |
| Xilinx Vivado 2017.1 | xsim | ocpi.core |
| Xilinx ISE 14.7 | isim | ocpi.core |
| Hardware Platform | Platform Name | Project |
| Epiq Solutions Matchstiq-Z1 | matchstiq_z1 ¹ | ocpi.assets |
| Avnet ZedBoard | zed ² | ocpi.assets |
| Avnet ZedBoard (ISE mode) | zed_ise ³ | ocpi.assets |
| Altera Stratix IV GX | alst4 | ocpi.assets |
| Xilinx ML605 (Virtex-6) | ml605 | ocpi.assets |
| Software Platform | Platform Name | Project |
| Centos6 | centos6 | ocpi.core |
| Centos7 | centos7 | ocpi.core |
| Xilinx Linux (2013 3rd Quarter) | xilinx13.3 ⁴ | ocpi.core |

¹Reference the Matchstiq-Z1 Getting Started Guide to build for `matchstiq_z1`

²Reference the Zedboard Getting Started Guide to build for `zedboard`

³Use Xilinx ISE tools to build for this platform

⁴RCC platform asociated with the zedboard and matchstiq_z1 platforms

4.4.5 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.

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

# 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`

4.4.6 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 4.4.4 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 simple application will contain three HDL Workers: `ramp`, `square`, and `ander`, which can be seen in Figure 2.

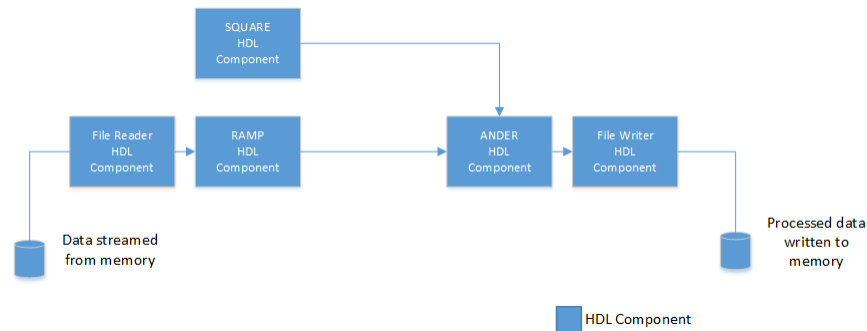


Figure 2: Block diagram of simple HDL application with three user-defined workers, a `FileRead`, and a `FileWrite`.

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`.

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
```

The project that was just created should have also been registered. This assumes the default registry is being used *and* the user is a member of the `opencpi` group (as explained in the *RPM Installation Guide*). The project registration can be done separately using the following `ocpidev` command from within the project:

```
% ocpidev register project
```

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
```

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 spec ramp
% ocpidev create spec square
% ocpidev create spec ander
```

```

DemoProject
|-- components
|   |-- lib
|   |   |-- package-id
|   |   '-- workers
|   |-- Library.mk
|   |-- Makefile
|   '-- specs
|       |-- ander-spec.xml
|       |-- ramp-spec.xml
|       '-- square-spec.xml
|-- exports
|   |-- imports -> ../imports
|   '-- project-package-id
|-- imports -> /opt/openmpi/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` postfix³. 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"/>

```

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

```

<Property Name="messagesize" Type="ushort" Default="2048" Readable="true" Writable="true"/>
<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 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++.

³Some older spec files use `_spec` as well.

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
```

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
|   |-- hdl
|   |   |-- ander-build.xml -> ../../ander.hdl/gen/ander-build.xml
|   |   |-- ander.xml -> ../../ander.hdl/ander.xml
|   |   |-- ramp-build.xml -> ../../ramp.hdl/gen/ramp-build.xml
|   |   |-- ramp.xml -> ../../ramp.hdl/ramp.xml
|   |   |-- square-build.xml -> ../../square.hdl/gen/square-build.xml
|   |   '-- square.xml -> ../../square.hdl/square.xml
|   |-- package-id
|   '-- 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, you can determine the key files:

First, navigate to the directory above DemoProject

```
% ocpidev clean project DemoProject
% tree --charset ascii DemoProject/
DemoProject/
|-- components
|   |-- ander.hdl
|   |   |-- ander.xml
|   |   '-- Makefile
|   |-- Library.mk
|   |-- Makefile
|   |-- ramp.hdl
|   |   |-- Makefile
|   |   '-- ramp.xml
|   |-- specs
|   |   |-- ander-spec.xml
|   |   |-- ramp-spec.xml
|   |   '-- square-spec.xml
|   '-- square.hdl
|       |-- Makefile
|       '-- square.xml
|-- Makefile
|-- Project.exports
|-- Project.mk
'-- project.xml
```

5 directories, 15 files

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.

For the *ramp worker*, insert the following in between the Hd1Worker XML tags (in components/ramp.hdl/ramp.xml):

```
<StreamInterface Name="in" DataWidth="16"/>
<StreamInterface Name="out" DataWidth="16"/>
```

For the *square worker*, insert the following in between the Hd1Worker XML tags (in components/square.hdl/square.xml):

```
<StreamInterface Name="out" DataWidth="16"/>
```

For the *ander worker*, insert the following in between the Hd1Worker XML tags (in components/ander.hdl/ander.xml):

```
<StreamInterface Name="in1" DataWidth="16"/>
<StreamInterface Name="in2" DataWidth="16"/>
<StreamInterface Name="out" DataWidth="32"/>
```

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

For the `ramp.vhd` *skeleton file* (components/ramp.hdl/ramp.vhd), insert the following *before* the “begin”:

```
signal idata_vld : bool_t;
signal odata_vld : bool_t;
signal buff_data : std_logic_vector(15 downto 0);
```

Replace the content between `begin` and `end rtl;` with the following:

```
-----
-- Valid Input (when up/downstream Workers ready and upstream is valid)
-----
idata_vld <= ctl_in.is_operating and in_in.ready and out_in.ready and in_in.valid;
-----
-- Take (when upstream Workers ready and valid)
-----
in_out.take <= ctl_in.is_operating and in_in.ready and in_in.valid and out_in.ready;
-----
-- Valid Output (when "Give")
-----
out_out.valid <= odata_vld;

ramp : process(ctl_in.clk)
begin
  if rising_edge(ctl_in.clk) then
    if (ctl_in.reset = '1') then
      buff_data <= (others => '0');
      odata_vld <= '0';
    elsif (idata_vld = '1') then
      buff_data <= std_logic_vector(signed(in_in.data) + signed(buff_data));
      odata_vld <= '1';
    else
      odata_vld <= '0';
    end if;
  end if;
end process ramp;

out_out.data <= buff_data;
-----
-- Give (when downstream Workers ready & valid)
-----
out_out.give <= out_in.ready and odata_vld and ctl_in.is_operating;
-----
-- Pass the SOM/EOM out
-----
out_out.som <= in_in.som;
out_out.eom <= in_in.eom;
```

For the `square.vhd` *skeleton file* (`components/square.hdl/square.vhd`), insert the following *before* the “begin”:

```
signal msg_cnt      : unsigned(15 downto 0);
signal cnt          : unsigned(15 downto 0);
signal odata_vld    : bool_t;
signal missed_odata_vld : bool_t := '0';
signal max_sample_cnt : unsigned(15 downto 0);
```

Replace the content between `begin` and `end rtl`; with the following:

```
-----
-- Simple counter to trigger the square pulse ON and OFF
-----
```

```
counter : process (ctl_in.clk)
begin
  if rising_edge(ctl_in.clk) then
    if (ctl_in.reset = '1' or cnt = 63) then
      cnt <= (others => '0');
    elsif (out_in.ready and ctl_in.is_operating) then
      cnt <= cnt + 1;
    end if;
  end if;
end process counter;
```

```
-----
-- Generate the square pulse
-----
```

```
square : process(ctl_in.clk)
begin
  if rising_edge(ctl_in.clk) then
    if (ctl_in.reset = '1') then
      out_out.data <= (others => '0');
      odata_vld <= '0';
    elsif (out_in.ready and ctl_in.is_operating) then
      if (cnt < 32) then
        out_out.data <= (others => '1');
      else
        out_out.data <= (others => '0');
      end if;
      odata_vld <= '1';
    else
      odata_vld <= '0';
    end if;
  end if;
end process square;
```

```
-----
-- Give (when downstream Worker ready & valid)
-----
```

```
out_out.give <= out_in.ready and (odata_vld or missed_odata_vld) and ctl_in.is_operating;
```

```
-----
-- Valid Output (when "Give")
-----
```

```
out_out.valid <= out_in.ready and (odata_vld or missed_odata_vld) and ctl_in.is_operating;
```

```
-----
-- SOM/EOM - counter set to message size, increment while giving
-----
```

```
max_sample_cnt <= props_in.messageSize srl 2;
```

```
messageSize_count : process (ctl_in.clk)
```

```

begin
  if rising_edge(ctl_in.clk) then
    if(ctl_in.reset = '1') then
      msg_cnt <= (others => '0');
    elsif (odata_vld = '1') then
      if(msg_cnt = unsigned(std_logic_vector(max_sample_cnt-1))) then
        msg_cnt <= (others => '0');
      else
        msg_cnt <= msg_cnt + 1;
      end if;
    end if;
  end if;
end process messageSize_count;

backPressure : process (ctl_in.clk)
begin
  if rising_edge(ctl_in.clk) then
    if(ctl_in.reset = '1' or out_in.ready = '1') then
      missed_odata_vld <= '0';
    elsif (out_in.ready = '0' and odata_vld = '1') then
      missed_odata_vld <= '1';
    end if;
    out_out.byte_enable <= (others => '1');
  end if;
end process backPressure;

-----
-- SOM Output (when downstream Worker ready, valid and message count is zero)
-----
out_out.som <= '1' when (out_in.ready = '1' and odata_vld = '1' and
                        msg_cnt = 0) else '0';

-----
-- EOM Output (when downstream Worker is ready, valid and
-- message count is equal to message length - 1)
-----
out_out.eom <= '1' when (out_in.ready = '1' and odata_vld = '1' and
                        msg_cnt = max_sample_cnt-1) else '0';

```

For the `ander.vhd` *skeleton* file (components/ander.hdl/ander.vhd), insert the following *before* the “begin”:

```

signal idata_vld : bool_t;
signal odata_vld : bool_t;

```

Replace the content between `begin` and `end rtl`; with the following:

```

-----
-- Take (when both upstream Workers and downstream Worker ready)
-----
in1_out.take <= in1_in.ready and in2_in.ready and ctl_in.is_operating and out_in.ready;
in2_out.take <= in1_in.ready and in2_in.ready and ctl_in.is_operating and out_in.ready;

idata_vld <= ctl_in.is_operating and out_in.ready and
             in1_in.ready and in2_in.ready and
             in1_in.valid and in2_in.valid;

-----
-- Enable all bytes on the stream
-----
out_out.byte_enable <= (others => '1');

```

```

-----
-- Data
-- AND the RAMP and SQUARE pulse together on the lower 16-bits
-- Let the RAMP data pass through on the upper 16-bits
-----

and_low_pass_high : process (ctl_in.clk)
begin
  if rising_edge(ctl_in.clk) then
    if (ctl_in.reset = '1') then
      out_out.data <= (others => '0');
    elsif (idata_vld = '1') then
      out_out.data(15 downto 0) <= in1_in.data and in2_in.data;
      out_out.data(31 downto 16) <= in1_in.data;
    end if;
    odata_vld <= idata_vld;
  -----
  -- Pass the RAMP's SOM/EOM out
  -----

  out_out.som <= in1_in.som;
  out_out.eom <= in1_in.eom;
end if;
end process and_low_pass_high;

-----
-- Valid Output (when output data is valid (which includes upstream==ready))
-----

out_out.valid <= odata_vld;

-----
-- Give (when downstream Workers ready and our output data is valid)
-----

out_out.give <= odata_vld and ctl_in.is_operating and out_in.ready;

```

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

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

5.5 Create an HDL Assembly

Before an application can be made, an *assembly* for the *HDL workers* need 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
```

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
```

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
```

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 application 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.

```
#!/bin/env python
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, \code{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:

```
#!/bin/env python
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`.

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.

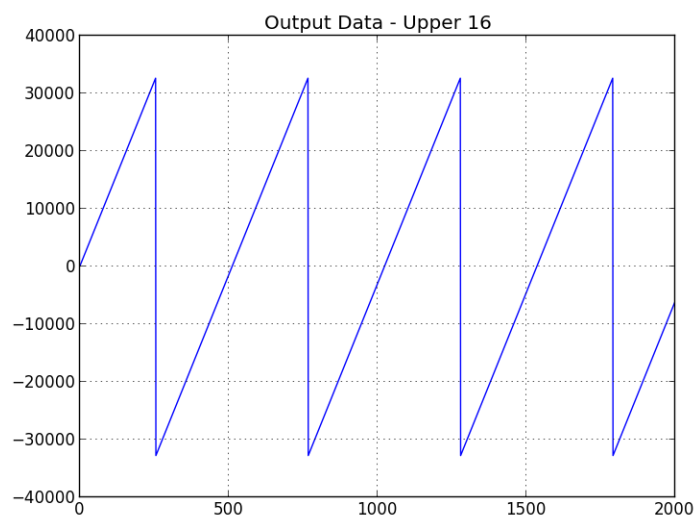


Figure 3: “RAMP” Output (Passed through by “ANDER”)

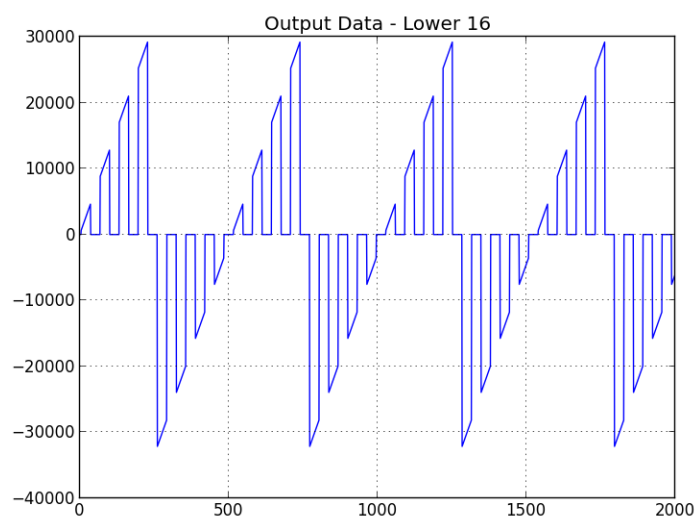


Figure 4: “ANDER” Output

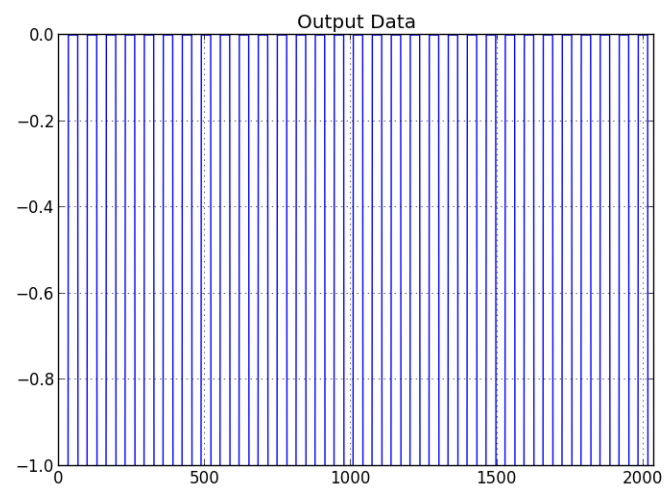


Figure 5: “SQUARE” Output = Input 2 to “ANDER”