OpenCPI
GUI User Guide

Revision History

Revision	Description of Change	Date
1.0	Creation	2021-04-19
1.1	Beta release updates	2021-07-06
1.1.1	Update Unit Test section	2021-07-23
1.1.2	Multi-select is now available; default target is host OS	2021-07-29
1.1.3	Add examples of setting a default editor to installation section	2022-09-07
1.2	Add sections for Node Graph and update for 2.4.3 changes	2022-10-14

Table of Contents

1 Introduction

The OpenCPI GUI is a PyQt5 application that allows access to most of the **ocpidev** commands. This allows the user to supplement command-line interface (CLI) use of **ocpidev** by using the features of the GUI. This reduces errors and can streamline OpenCPI project development.

While feature-parity with CLI tools has not been completed, a large number of CLI commands have been implemented within the GUI. For those commands that are not available within the GUI, the CLI can still be used.

Where **ocpidev** commands are shown, please refer to the **ocpidev** man pages for a more thorough discussion, as more advanced features are available via the command line.

2 Installation

Source installation is the primary way of using OpenCPI GUI. Prior to running, will need to install PyQt5 by running the command **`sudo apt install python3-pyqt5`** for Debian-based distros and **`sudo yum install python36-qt5.x86_64`** for Red Hat-based distros.

To obtain the GUI source code, it needs to be cloned from the Gitlab repository using **`git`**. If **`git`** is not already installed, use the command **`sudo apt install git`** (Debian-based) or **`sudo yum install git`** (Red Hat-based).

Change to the directory where want the GUI code to reside. The command

```
git clone https://gitlab.com/opencpi/ie-gui.git
```

will clone the GUI source code to the local system.

To launch OpenCPI GUI, use the command **`./install-gui.sh`**. Once this script has run successfully, can run the GUI by executing the command **`ocpigui`**.

After the OpenCPI GUI is installed, a default editor needs to be assigned for **`.xml`**, **`.vhd`**, and **`.cc`** file types. Two examples of editors are given below. The OpenCPI GUI is not limited to use of these two editors, but will use whatever editor is configured within the OS for these file types.

`gedit` is a native Linux text editor that is easily assigned as the default editor by running three commands:

```
xdg-mime default gedit.desktop application/xml
xdg-mime default gedit.desktop text/x-vhdl
xdg-mime default gedit.desktop text/x-c++src
```

Similarly, if **`gvim`** is the desired editor, then this application can be installed by running the command **`sudo apt install gvim`** for Debian-based distros and **`sudo yum install gvim`** for Red Hat-based distros, followed by running three commands to configure **`gvim`** as the default editor:

```
xdg-mime default gvim.desktop application/xml
xdg-mime default gvim.desktop text/x-vhdl
xdg-mime default gvim.desktop text/x-c++src
```

On first launch, the GUI displays two dialogs: the first one asks for the installation path of OpenCPI, while the second one asks for the project path to display. It is recommended to use the path **`~/User_OpenCPI_Projects`** for new projects.

3 Interface Details

The Project Explorer panel shows a view of the currently set project. By default, this is set to show the contents of the **`~/User_OpenCPI_Projects`** directory, which is assumed to be the location for all OpenCPI project assets. However, can change the displayed project via the *OpenCPI Configuration* menu.

The following figure shows the main window. Note that the **`ocpidev`** commands for creating OpenCPI components are in the menu bar. More commands are available via the Project Explorer's right-click context menu.

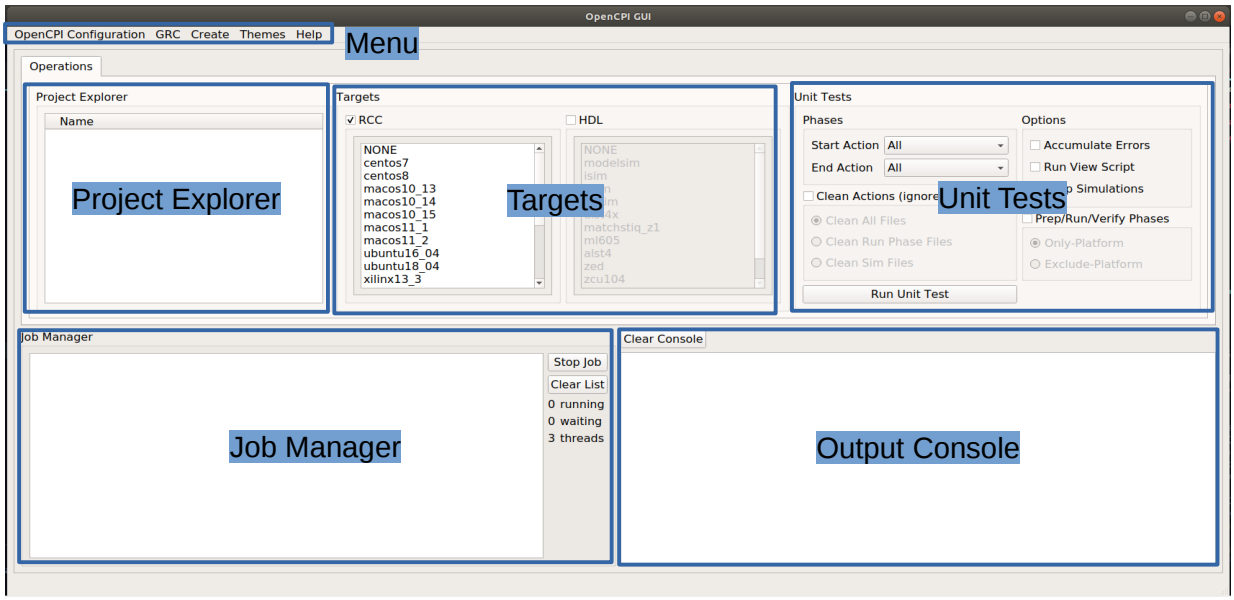

Figure 1: Main Window

The main window is divided into six main areas, which are discussed in detail in the following sections:

- Menu bar
- Project Explorer
- Targets
- Unit Tests
- Job Manager
- Output Console

Note that the commands that are executed within the GUI are displayed in the Output Console for clarity. It should also be noted that a lot of the GUI is generated

dynamically, resulting in instances where the GUI, or individual windows, will be black for a brief period while the necessary information is generated.

4 Menu Bar

While a lot of functionality for OpenCPI projects and related components is provided via Project Explorer's context menu, some features are available through the menu bar.

The Menu Bar contains the following menu options:

- OpenCPI Configuration
- GRC
- Create
- Show
- Node Graph
- Themes
- Help

These categories will be discussed in the following subsections.

4.1 OpenCPI Configuration Menu

The OpenCPI Configuration menu option handles both setting and viewing certain aspects of the OpenCPI GUI environment.

The following figure shows the OpenCPI Configuration menu options.

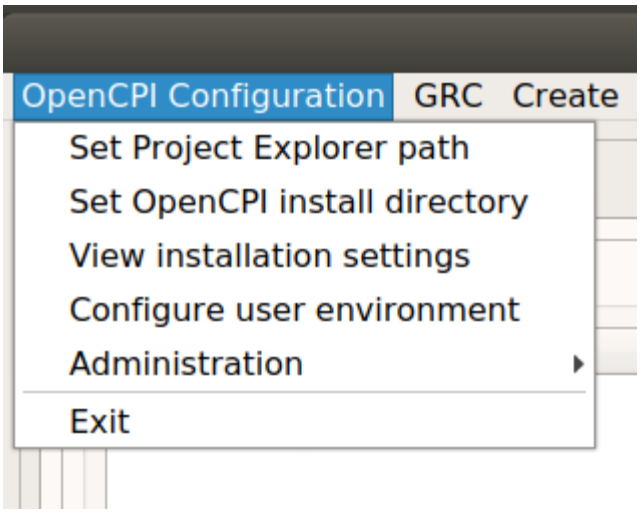

Figure 2: OpenCPI Directory Location Dialog

The sections below describe the individual options for the menu.

4.1.1 Set Project Explorer path

This option allows to set the path that is displayed within the Project Explorer panel. Normally the displayed path is set to **`~/User_OpenCPI_Projects`** but projects created in different locations can be accessed.

On first use of the GUI, this dialog is called to set the correct path prior to setting up the GUI environment. The dialog associated with this menu option shows the Project Explorer's currently displayed path, as shown below.

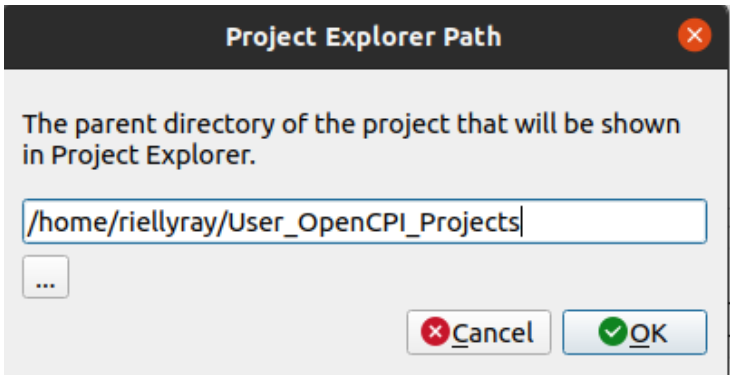

Figure 3: Current Path Displayed in Project Explorer

4.1.2 Set OpenCPI Install Directory

This option allows to set the location where OpenCPI is installed. This location is necessary to ensure that the GUI will correctly execute OpenCPI commands.

On first use of the GUI, this dialog is displayed to set the correct path prior to setting up the GUI environment.

The dialog associated with this menu option shows the path currently set to the **`/opencpi`** directory, as shown below.

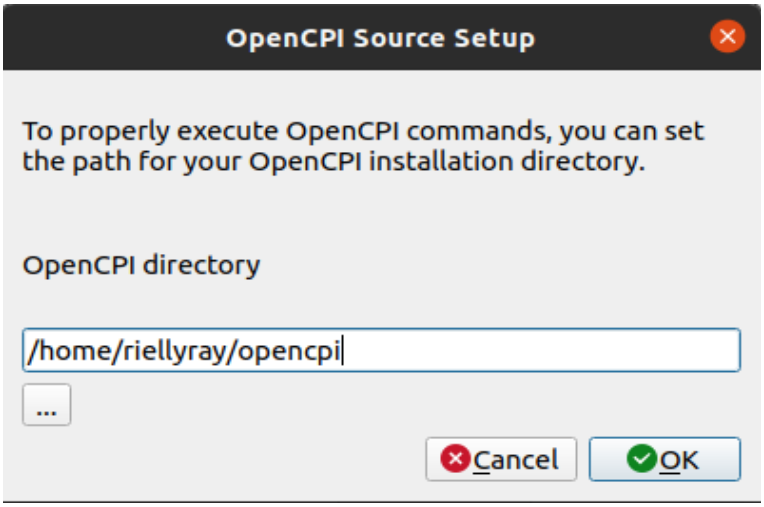

Figure 4: OpenCPI Installation Path

4.1.3 View Installation Settings

This option displays the current environment settings for OpenCPI, including the CDK path and user-set environment settings. As shown in the figure below, if have not made any environment settings, the window indicates that default settings are used. However, if specific settings are made, such as for XSIM, those settings are displayed. As this window is read-only, environment settings cannot be modified using the GUI. An external application is required to make changes.

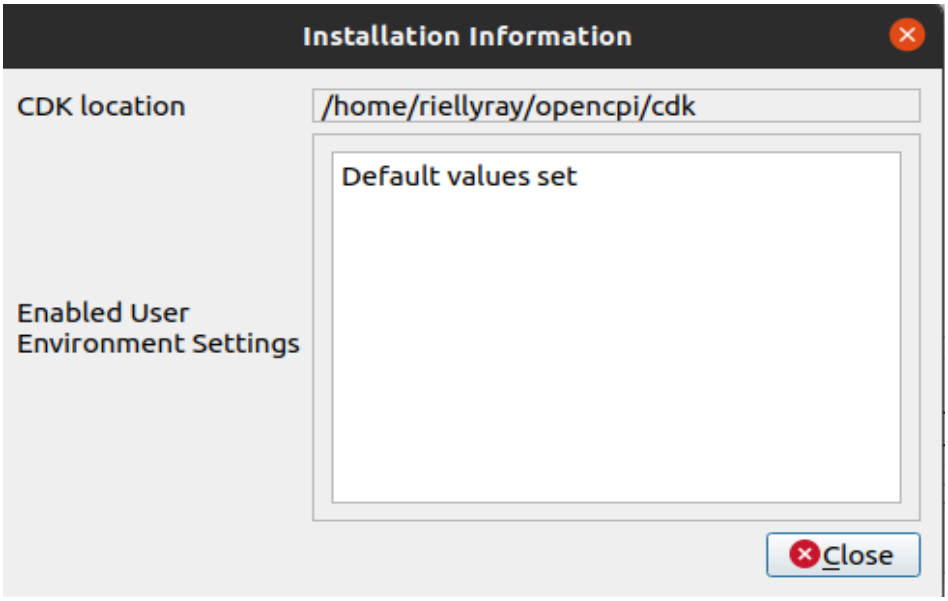

Figure 5: Installation Environment Settings

4.1.4 Configure User Environment

This option launches an external application that allows to update the **`user-env.sh`** file. The application used will be the default application set by the user or the OS for text files, as shown below.

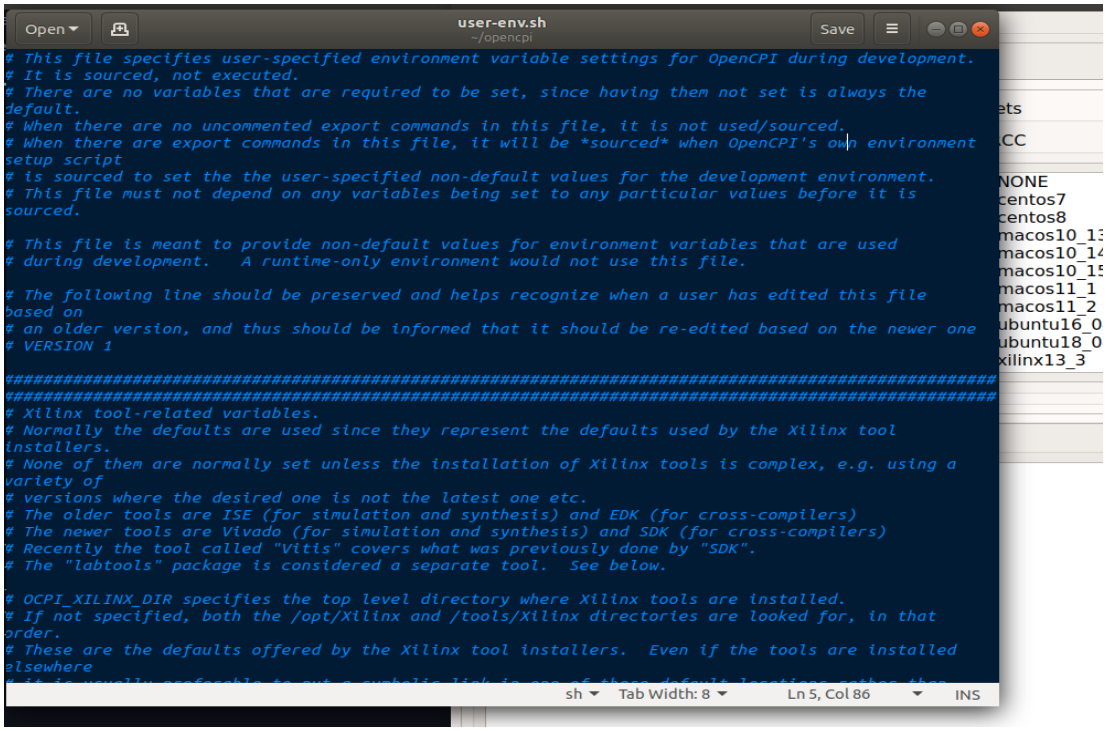

*Figure 6: Modifying the **`user-env.sh`** File*

4.1.5 Administration

This menu option has to sub-options: Install Platform and Deploy Platform. These choices correspond to **`ocpiadmin`** command operations.

Install Platform installs the specified platform using the information provided in the dialog shown below. The only required information is the platform name. Optional information includes the package ID, platform URL, and Git revision. The last two items require the package ID to be provided first. This does mean that the desired platform does not have to be on the local file system. It can be downloaded directly from a remote location, such as a web site or Git repository.

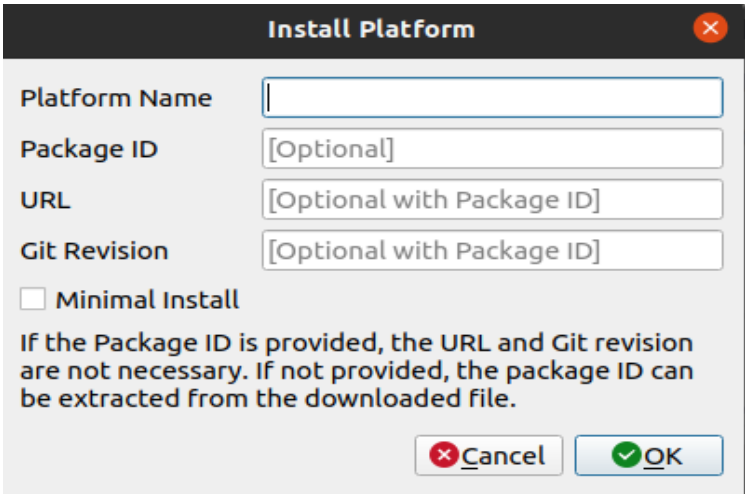

Figure 7: Deploy Platform Dialog

Deploy Platform deploys the platform using the RCC and HDL platforms specified in the dialog, as shown below.

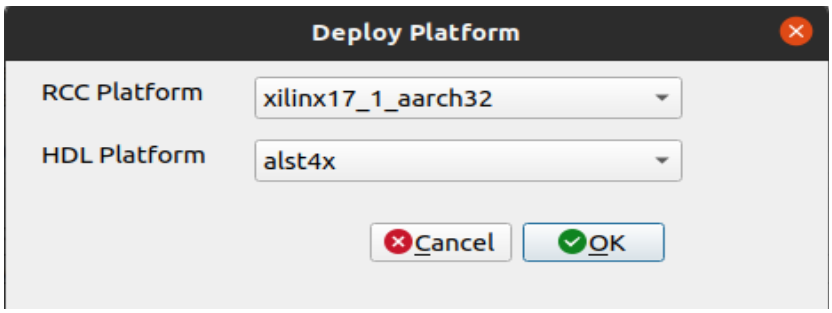

Figure 8: Deploy Platform Dialog

5 GRC Menu

The GRC menu allows to work concurrently with GNU Radio Companion and OpenCPI. Two menu options are available: ***Configure GRC*** and ***Launch GRC***. GNU Radio must be installed to use GRC.

The configuration dialog allows setting of the **`PYTHONPATH`**, which is necessary to launch GRC. The default setting is the normal path for Python packages.

The following figure shows the GRC configuration dialog.

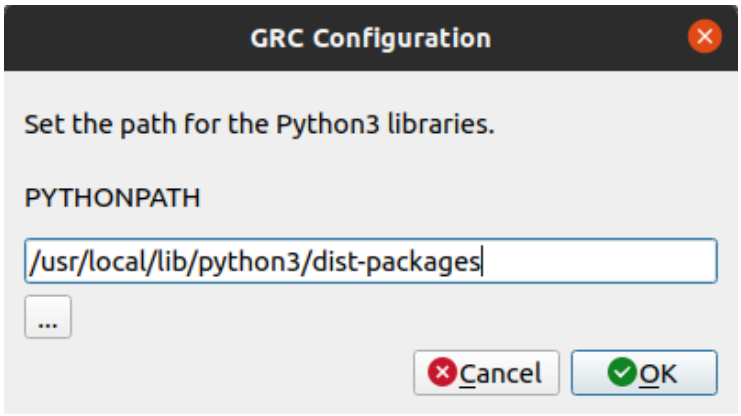

Figure 9: GRC Configuration Dialog

If GNU Radio is installed, executing GRC via the Launch GRC menu opens GRC in a separate window (example below). This setup allows to have both GRC and the OpenCPI GUI open at the same time, if desired.

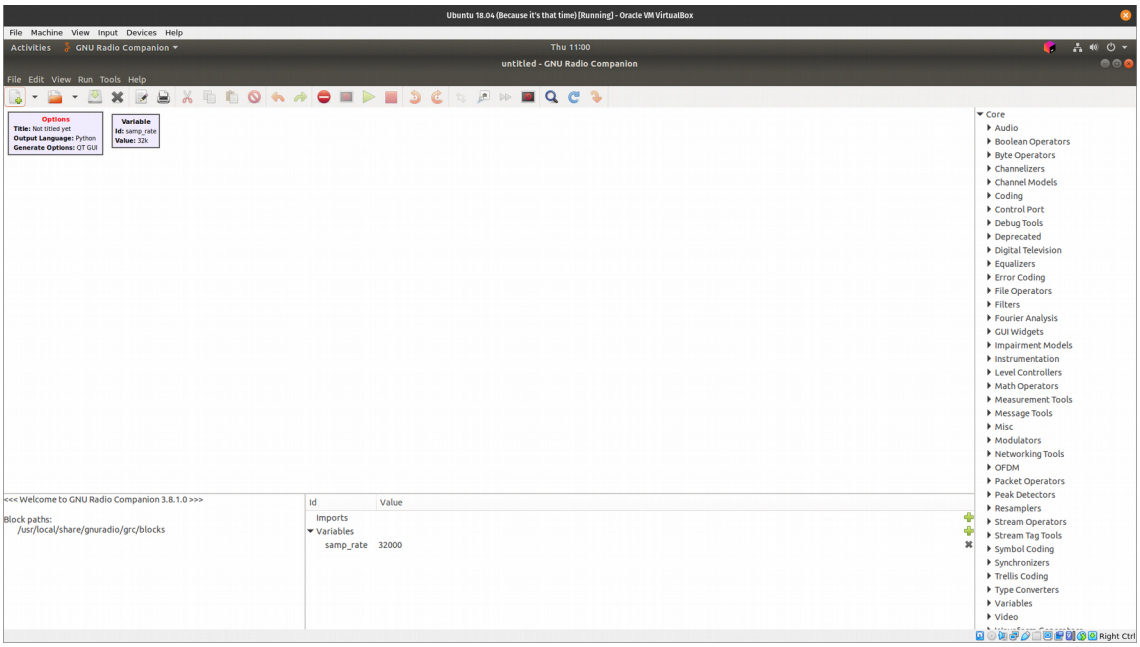

Figure 10: GRC Window

6 Create Menu

This section describes the Create menu options, shown in the figure below. It is important to note that only the basic functionality of **`ocpidev`** create options are provided; the specific commands that are implemented are provided in their respective sections.

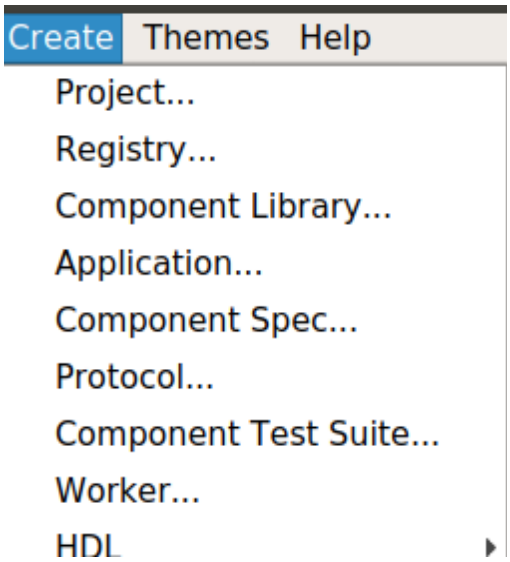

Figure 11: Creation Menu

Command output, if any, is shown in the output console at the bottom of the GUI. However, not every **`ocpidev`** command has output, nor do they have the same format.

6.1 Create Project

The dialog for creating a new project is shown in the figure below. Only Project Name is required; the other fields are optional.

New Project Location allows to specify where the new project will be located, assuming they do not want to use the default path.

Package Prefix defaults to local if not provided, while Package Name has the same name as the project.

Project Dependency allows to specify another project on which the new project depends.

Register Project allows to register the project when it is created rather than as a later step.

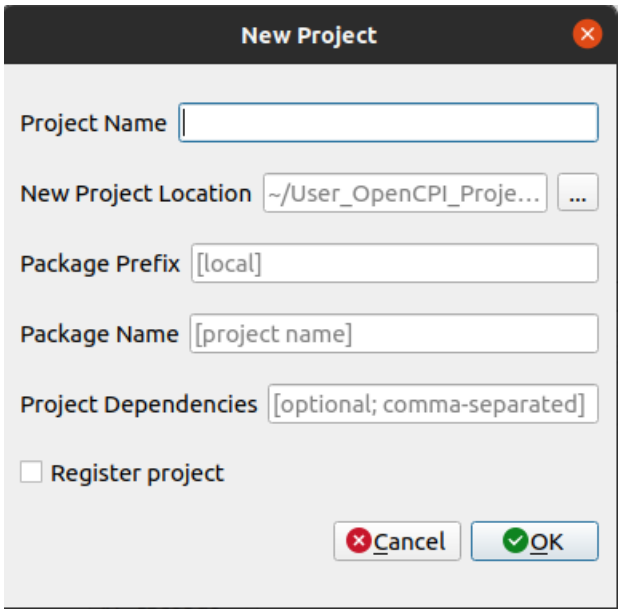

Figure 12: New Project Dialog

If the Project Explorer display path is set to the same location as the new project, the new project appears in the Project Explorer panel, shown below. Otherwise the Project Explorer display path must be changed to see the new project.

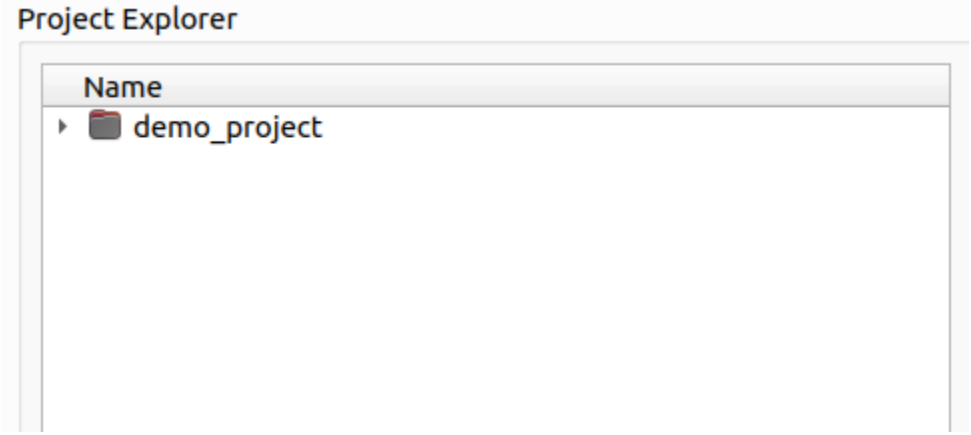

Figure 13: New Project in Project Explorer

6.2 Create Registry

To create new registries, select “Create->Registry...” from the menu bar. A new dialog box appears as shown below.

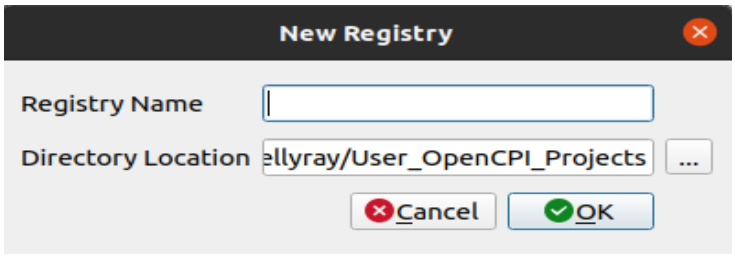

Figure 14: New Registry Dialog

Provide the name of the registry and the desired location. The default location is set to **`~/User_OpenCPI_Projects`**. When the new registry is created, it is added to the designated location.

Just because the registry has been added to the directory doesn’t mean it’s available for use with r OpenCPI projects. will receive a notice in the output console indicating that there is another step to complete, as shown in the following figure.

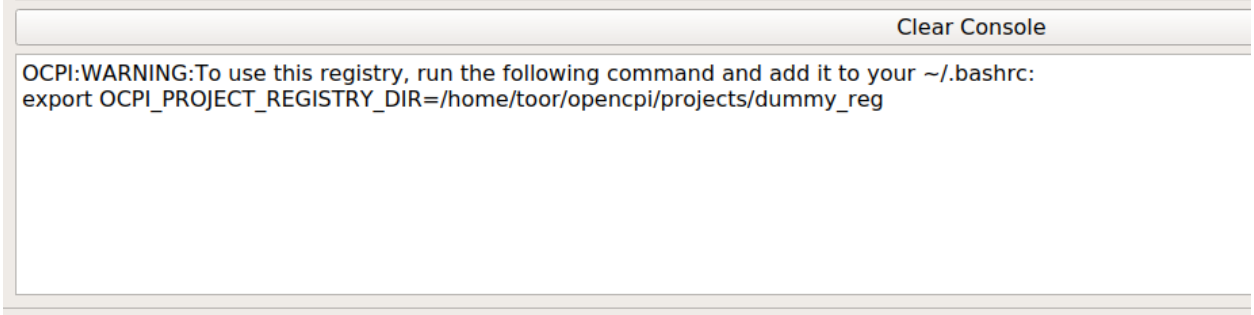

Figure 15: New Registry Warning

6.3 Create Library

Creating a new library is similar to creating a new project. The primary difference is that have to associate the new library with an existing project, as shown below. The Associated Project drop-down menu lists all the projects listed in Project Explorer' display path. The new library is located at **`/<project>/components/<library>`**.

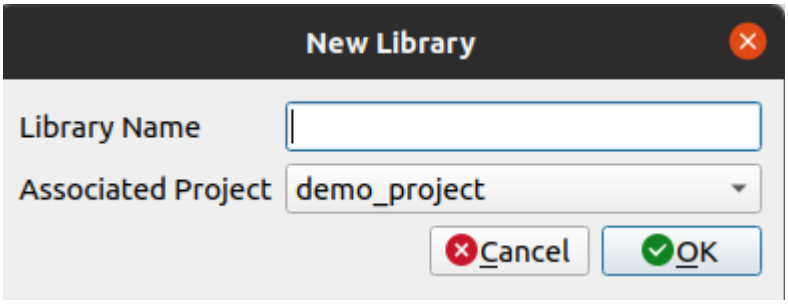

Figure 16: New Library Dialog

6.4 Create Application

Adding applications to a project is much like adding a library: provide the application name and associated project, as shown in the figure below. also have a selection of what type of application wish to create. can create an ACI application, an XML-only application, or XML in a subdirectory. These options are mutually exclusive so can only choose one. The new application will be found in **`/<project>/applications`**.

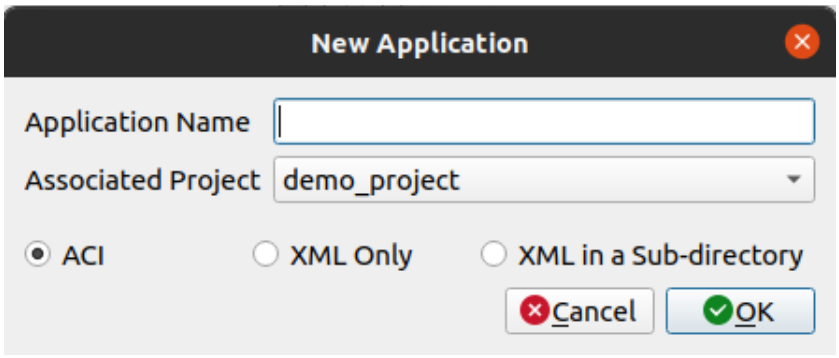

Figure 17: New Application Dialog

6.5 Create Component Spec

Unlike the previous dialogs, the new component spec dialog has an additional selection: Associated Library, shown below. As a particular project can have multiple libraries, not only have to select the desired project, but also the library with which the spec should be associated. The new spec will be found in ***/<project>/components/<library>***.

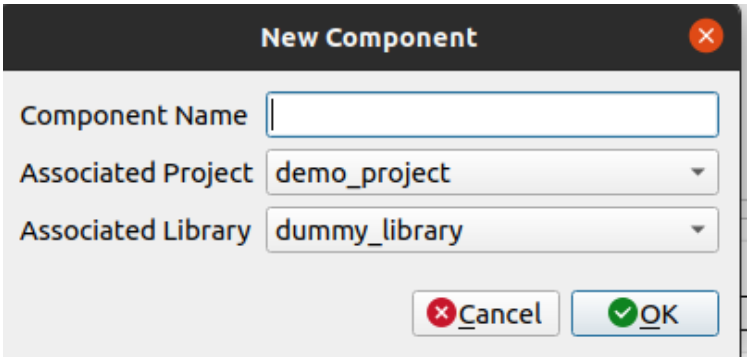

Figure 18: New Component Dialog

6.6 Create Protocol

Like component specs, the New Protocol dialog asks for the project and library associations for the protocol, shown below. The new protocol can be found in the ***/<project>/<library>/specs*** directory.

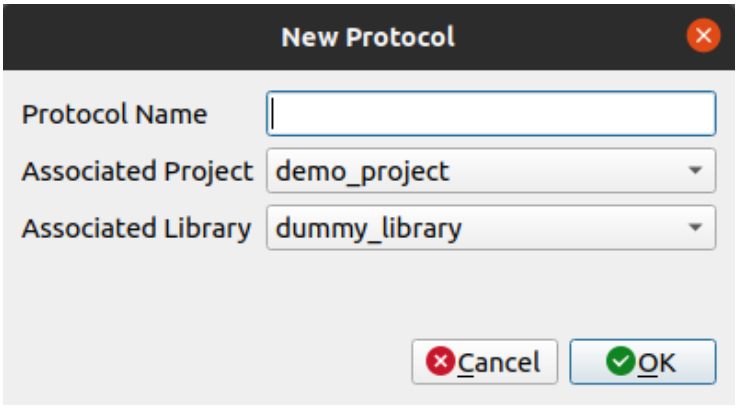

Figure 19: New Protocol Dialog

6.7 Create Unit Test Suite

When creating a new test suite, no name is required, but the project, library, and component spec with which the test will be associated are required. Keep in mind that it can take up to 30 seconds for this information to display because the GUI needs to discover all the available components.

The new test is added as `/<project>/components/<library>/<component_spec>.test`.

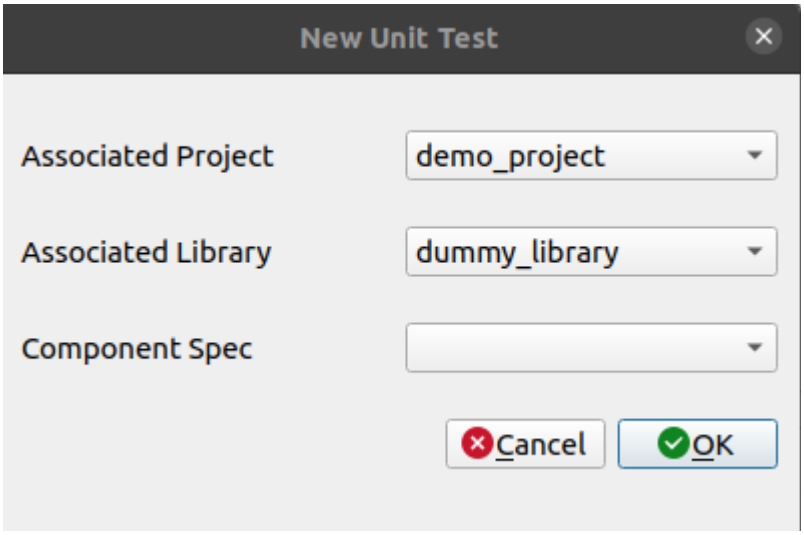

Figure 20: New Test Suite Dialog

6.8 Create Worker

New workers have quite a few options compared to the other creation dialogs, as shown in the following image. After entering the worker's name, select the project it is associated to.

Next, pick the model it will be using: RCC or HDL. Based on the model, the languages available will change. RCC allows for either C or C++ while HDL only allows for VHDL.

Finally, select the project's library that will be used by the worker and any component spec; this is the only item in the dialog that is optional to change.

Keep in mind that it can take up to 30 seconds for this information to display because the GUI needs to discover all the available components.

The new worker is found under its associated library at ***/<project>/components/<library>/<worker_name>***.

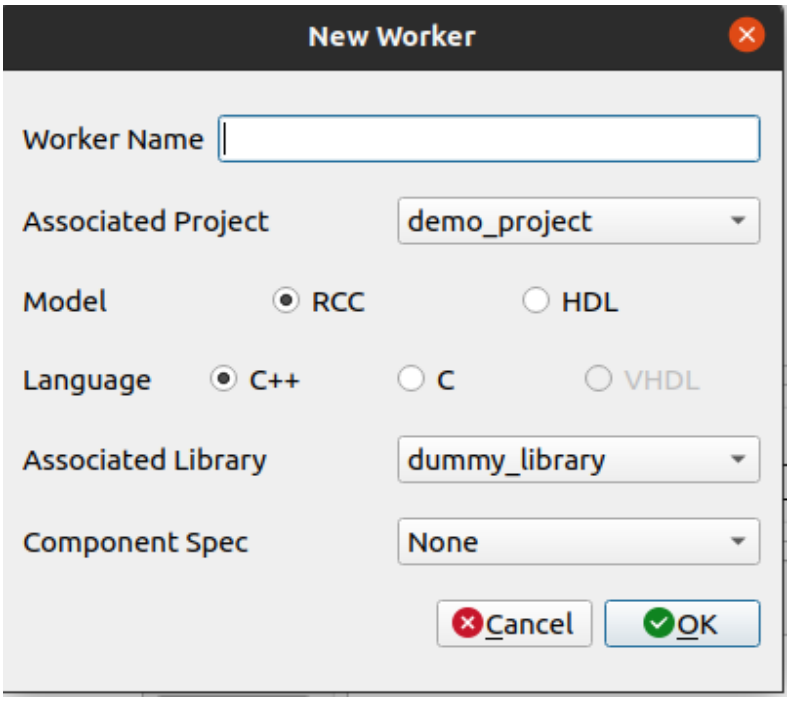

Figure 21: New Worker Dialog

6.9 Create HDL Assembly

Creating an HDL assembly only requires the assembly name and associated project, displayed below. The new assembly will be found in the project directory at **`/<project>/hdl/assemblies/<assembly_name>`**.

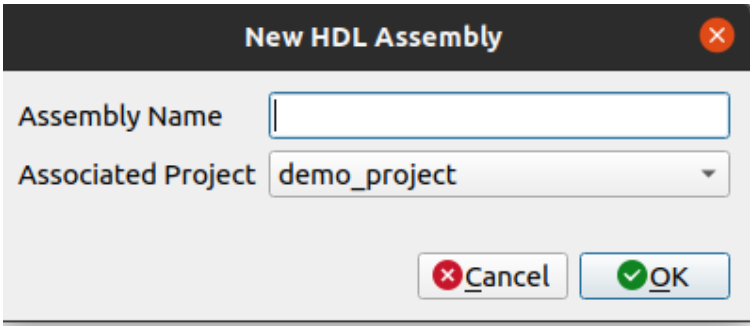

Figure 22: New HDL Assembly Dialog

6.10 Create HDL Platform

Making a new HDL platform has some additional fields (below) beyond the normal name and project: part number and time server frequency. Once the new platform information is input, the HDL platform is located under the project's HDL directory at **`/<project>/hdl/platforms/<platform_name>`**.

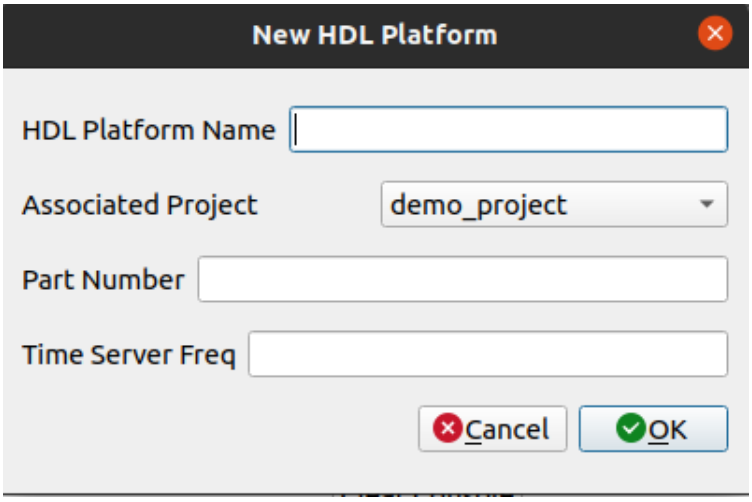

Figure 23: New HDL Platform Dialog

6.11 Create HDL Primitive Library

The dialog for primitive libraries is the same as for HDL assemblies: library name and associated project. When completed, the new primitive library is added to the HDL directory at **`/<project>/hdl/primitives/<lib_name>`**.

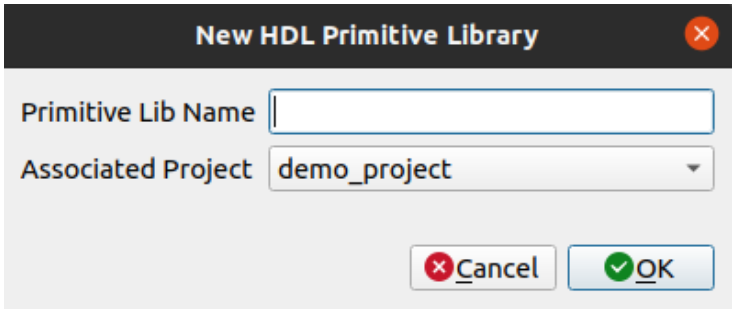

Figure 24: New HDL Primitive Library Dialog

6.12 Create HDL Primitive Core

This dialog is the same as HDL primitive library: provide the name and associated project. The new core will be added to the **`/<project>/hdl/primitives/<core_name>`**.

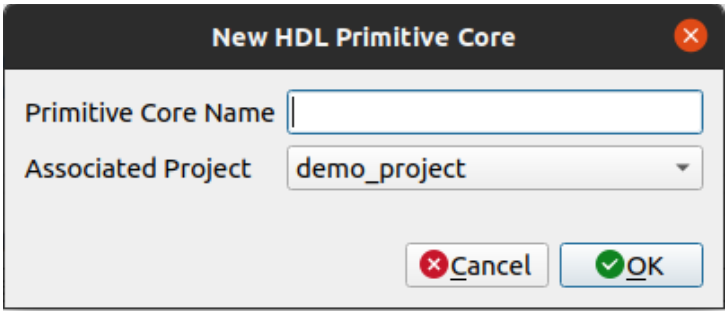

Figure 25: New HDL Primitive Core Dialog

7 Show Menu

The Show menu item is not context-sensitive. That is, don't have to select a particular item within Project Explorer for it to function. Two options are available: show registry and show workers.

Show registry displays the projects currently associated with the default registry, as shown below.

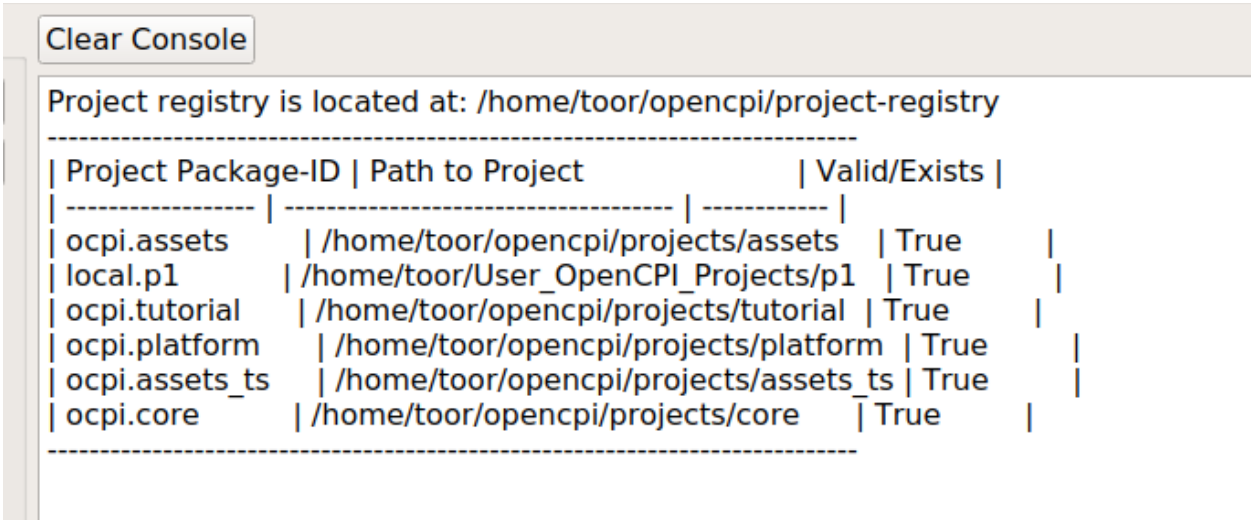

Figure 26: Show Registry Output

Show workers shows the workers currently created, as shown below.

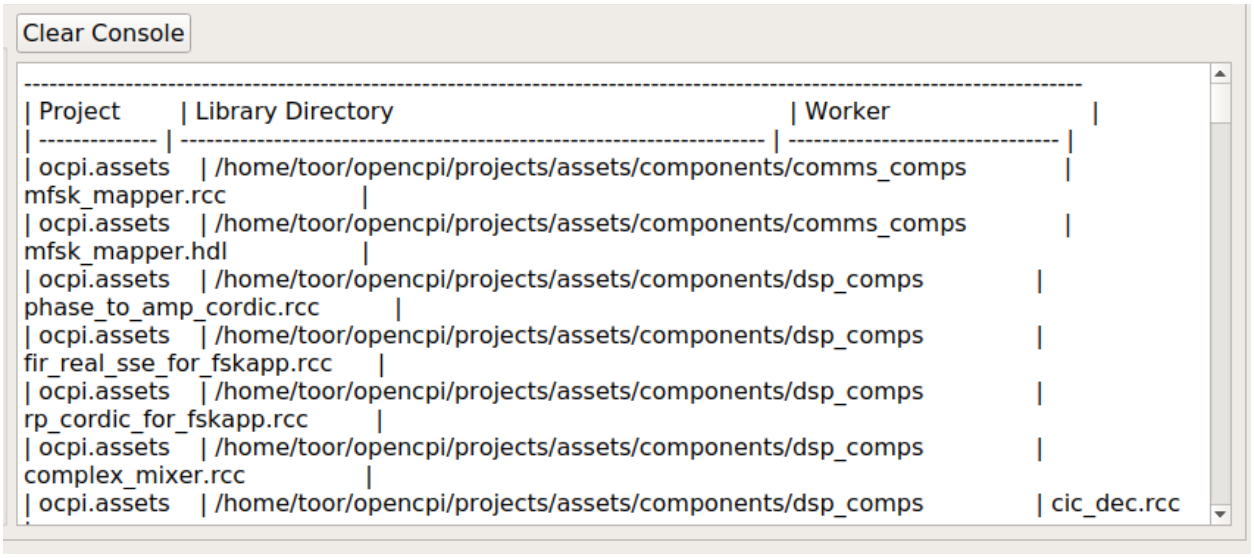

Figure 27: Show Workers output

8 Node Graph Menu

The Node Graph is designed for users who want to make application XML files in a more visual way instead of coding in XML. The Node Graph allows to string OpenCPI components together and then edit different values for properties, buffer counts, etc. and then export these applications into an OpenCPI-compliant application XML file. This next section takes on a tour of how the Node Graph works.

Shown below is what the Node Graph looks like when launch it. On the right is a list of r available components. will be able to manually drag and drop these components into the flow graph area, but not yet.

Figure 28: Node Graph on Launch

To start creating a new application in the Node Graph, select **File → New → New Application**. The other options have are **File → Open** and **File → Import**. We will go deeper into the differences between these actions in the next sections. Once click **File → New → New Application**, the Node Graph appears as shown below and can start dragging in components.

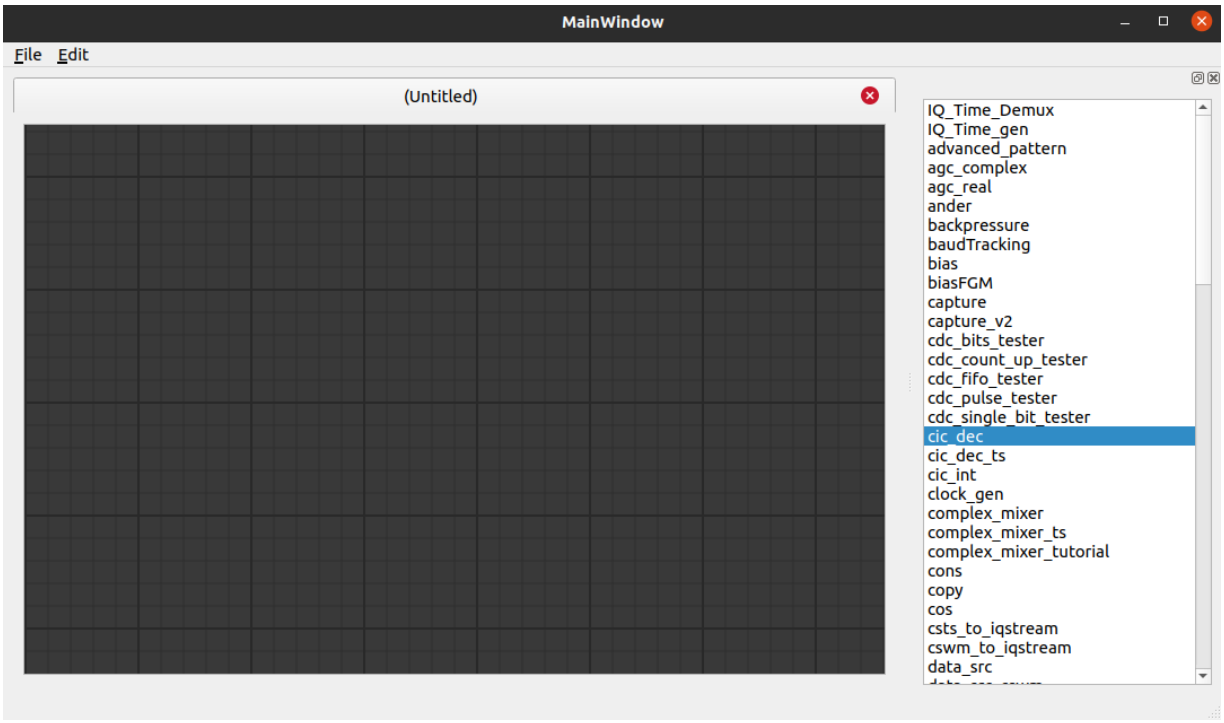

Figure 29: Node Graph After File → New → New Application

8.1 Opening and Saving a Node Graph

In the Node Graph, can save and export. Exporting converts an application that the user have created into an XML file. We will go into more depth about this action in a different section. Saving allows to save both the component information have changed and edited *and* the locations of the components, which allows the Node Graph to populate the components in the locations left them when saved. This will allow users to easily come back and work on applications at later times as it will look exactly the same as left it. Keep in mind OpenCPI does not care about this json created from saving will have to export if the user wants OpenCPI to recognize it.

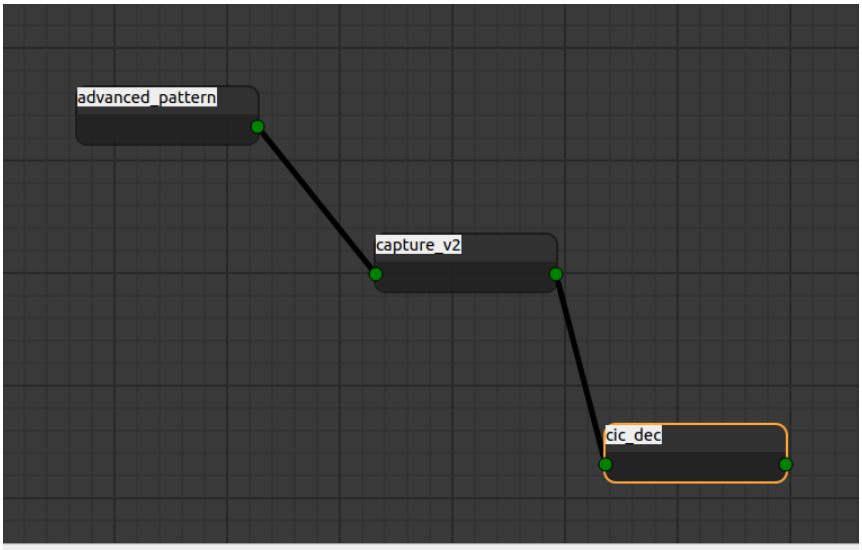

Figure 30: Example Application to be Opened

This is an example application created using the GUI. Here, we selected **File → Save As** and called it **`test_app`**. Use **File → Save As** the first time save the application. Subsequently, can just use **File → Save**. This is what it looks like when select **File → Save As**:

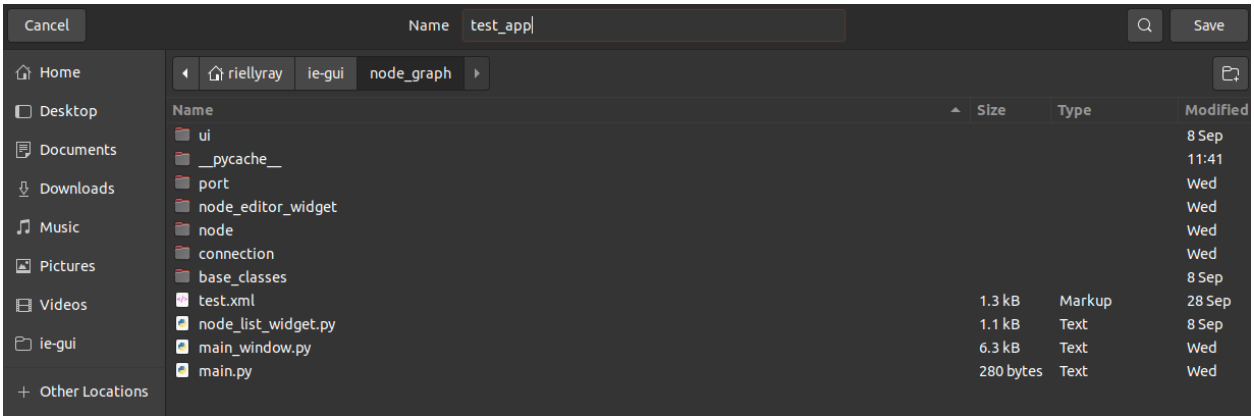

Figure 31: Saving the Example Application

Once this is saved, can then select **File** → **Open** and open the saved `.json` file. The result looks like this:

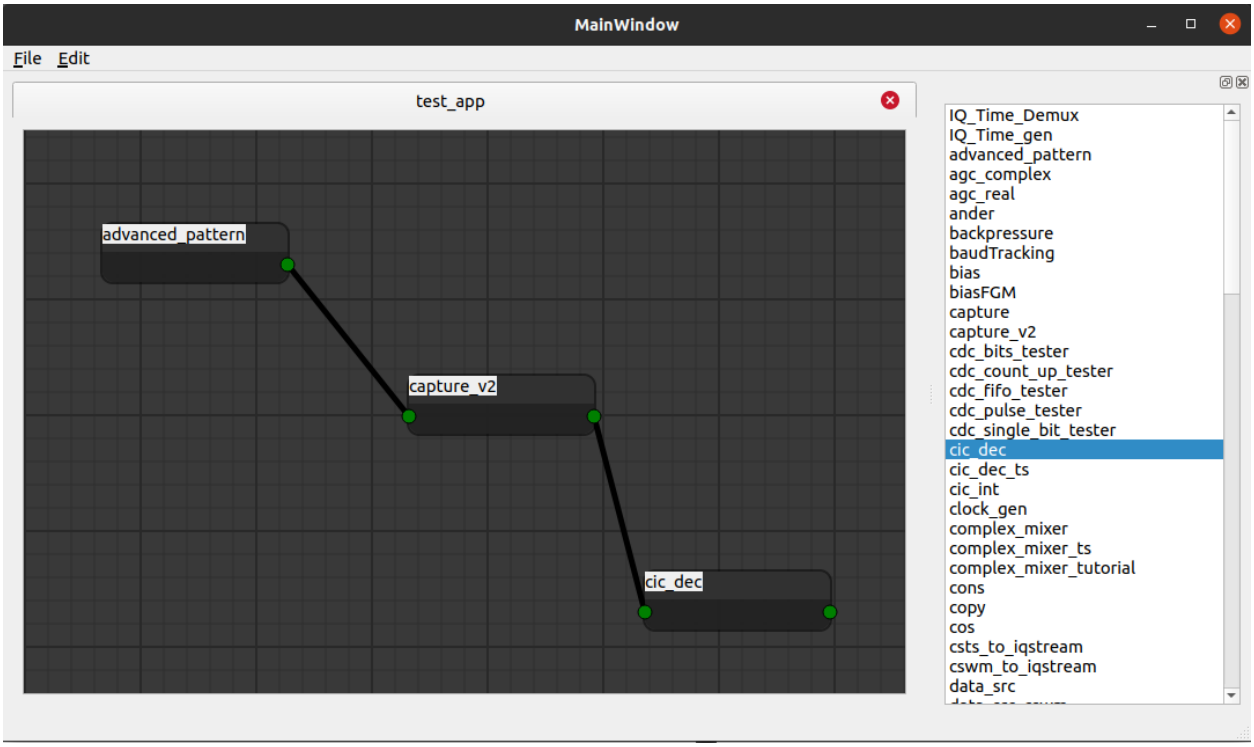

Figure 32: Node Graph After Opening JSON File

Once the json file is open, can start to drag in new components or edit the ones that are in there already, etc.

8.2 Importing an Application

In the Node Graph, the import action allows to import external application XML files as long as they fit the following criteria:

- The connection must always be its own instance. The Node Graph does not support reading in XMLs that have the connection as a property of the component. Here is an example (**Connect=** is not supported for the Node Graph to input):

```
<Instance Component="ocpi.assets.util_comps.zero_pad"
Connect="tx_fir_real"> <Property Name="num_zeros" Value="38"/>
```

- The ports need to be attributes of the connection instance. **To** and **From** are not supported by the GUI as attributes of the component. Here is an example that is not supported by the GUI:

```
<Instance
Component="ocpi.assets.dsp_comps.phase_to_amp_cordic"
Connect="drc" from='out' to='tx'>
<Property Name="magnitude" Value="20000"/> <Property
Name="DATA_WIDTH" Value="16"/>
<Property Name="DATA_EXT" Value="6"/>
<Property Name="STAGES" Value="16"/>
</Instance>
```

- Buffer size must be included in the instance of the connection itself, not in the port of the component.. Here is an example that is not supported by the GUI:

```
<Connection>
<Port Instance="iq_imbalance_fixer" Name="out" buffercount='2'/>
<Port Instance="file_write" Name="in" buffersize="16352"
buffercount='7'/>
</Connection>
```

  Here is an example that is supported by the GUI:

```
<connection buffersize='4000'>
<port from='out' instance='pattern'/>
<port to='in' instance='bias'/>
</connection>
```

- The **slave** attribute for components is not currently supported (support may be added in a future release). Here is an example:

```
<instance component="ocpi.assets.devices.ad9361_config"/>  <instance
component="ocpi.assets.devices.ad9361_config_proxy"
slave="ad9361_config">
<!-- when ad9361_config_proxy is the highest-level proxy in an app
xml, this property must be specified here in the app xml OR set
using the ACI to reflect a valid RF port input for the hardware
being used, if a higher level proxy is included, typically that
proxy should set this value -->
<!--property name="rx_rf_port_input" value=""/--> <!-- relying on
ACI -->
</instance>
```

- The **model** attribute for components is not currently supported (support may be added in a future release). Here is an example:

```
<Instance Component="ocpi.assets.dsp_comps.fir_real_sse"
Name="rx_fir_real" Model="hdl" Connect="baudTracking"> <Property
Name="taps" ValueFile="idata/rx_rrcos_taps.dat"/> </Instance>
```

The following figure shows an example application that the Node Graph can import:

```
<?xml version="1.0" ?>
<Application>
    <Instance Component="ocpi.core.file_read" Name="file_read">
        <Property Name="fileName" Value="test.input"/>
        <Property Name="messagesInFile" Value="false"/>
        <Property Name="opcode" Value="0"/>
        <Property Name="messageSize" Value="4"/>
        <Property Name="granularity" Value="1"/>
    </Instance>
    <Instance Component="ocpi.core.file_write" Name="file_write">
        <Property Name="fileName" Value="test.output"/>
        <Property Name="messagesInFile" Value="false"/>
        <Property Name="stopOnEOF" Value="true"/>
    </Instance>
    <connection>
        <port Instance="file_read" Name="out"/>
        <port Instance="file_write" Name="in"/>
    </connection>
</Application>
```

Figure 33: Valid XML for the Node Graph

copy.xml
~/opencpi/projects/assets/applications

```
<application package='ocpi.core'>
  <instance component='file_read'>
    <property name='filename' value='test.input'/>
    <property name='messageSize' value='4'/>
  </instance>
  <instance component='file_write'>
    <property name='filename' value='test.output'/>
  </instance>
  <connection>
  <port instance='file_read' name='out'/>
    <port instance='file_write' name='in'/>
  </connection>
</application>
```

Figure 34: XML to Import

To import, select **File** → **Import**. Below is the result of the Node Graph importing the above example application:

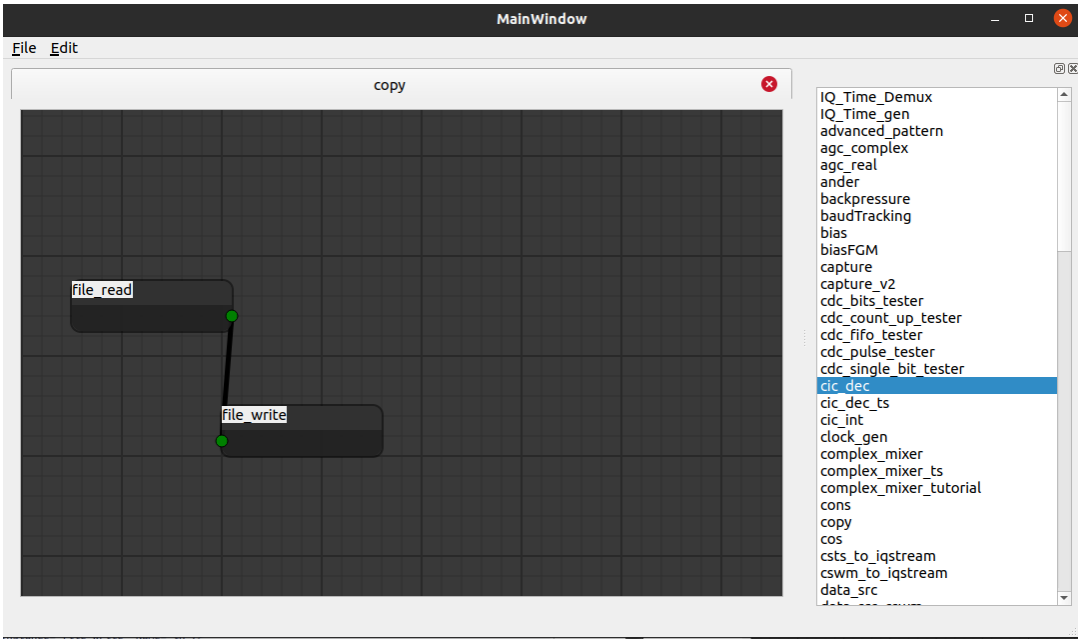

Figure 35: Node Graph After Importing Application

Because the Node Graph is importing an XML, it does not have access to the locations of the components, so it tries its best to space everything out properly. It could get a little messy if it's a huge application with multiple connections and components.

8.3 Exporting an Application

The Node Graph can also export applications that created in the Node Graph into OpenCPI-compliant application XML. To export, select **File** → **Export** then save it. The Node Graph converts it into OpenCPI- compliant XML. This will allow the user to then be able to build, run, etc. the application just made in OpenCPI. Also remember if the user also wants to come back to this and it look the same the user will also need to save this. If don't and just import the nodes will not be in the same place.

Below is an example application created with the Node Graph:

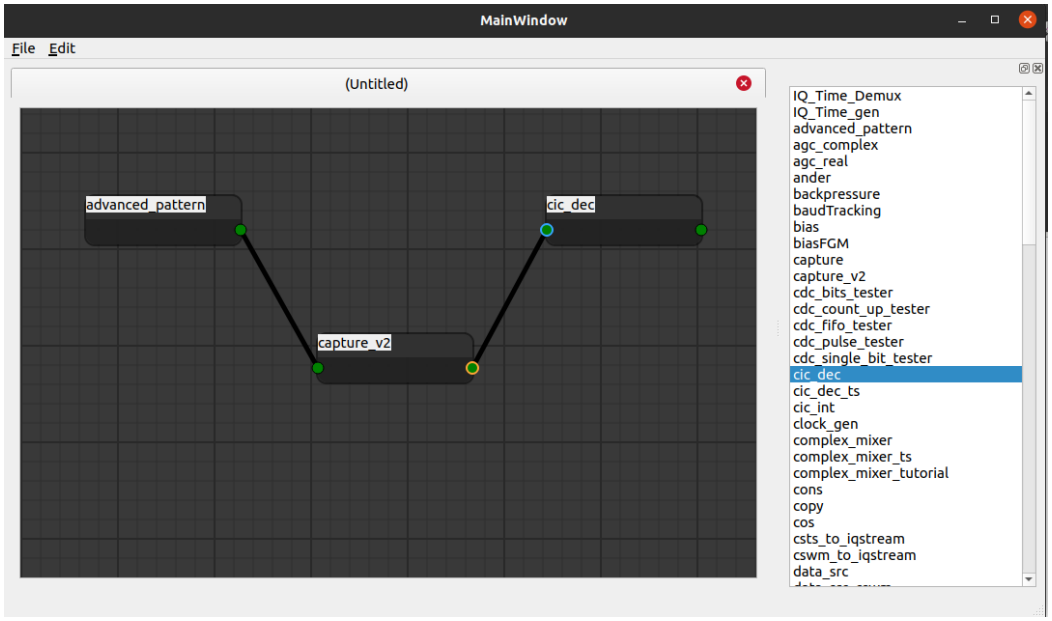

Figure 36: Node Graph After Importing Application

Below is the result after exporting it:

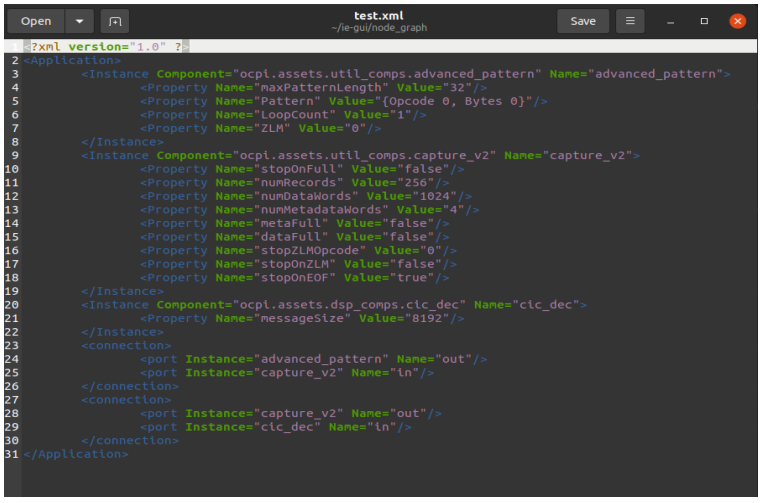

Figure 37: Node Graph After Importing Application

8.4 Components and Component Side Panel

This is what a component and its side panel looks like in the Node Graph:

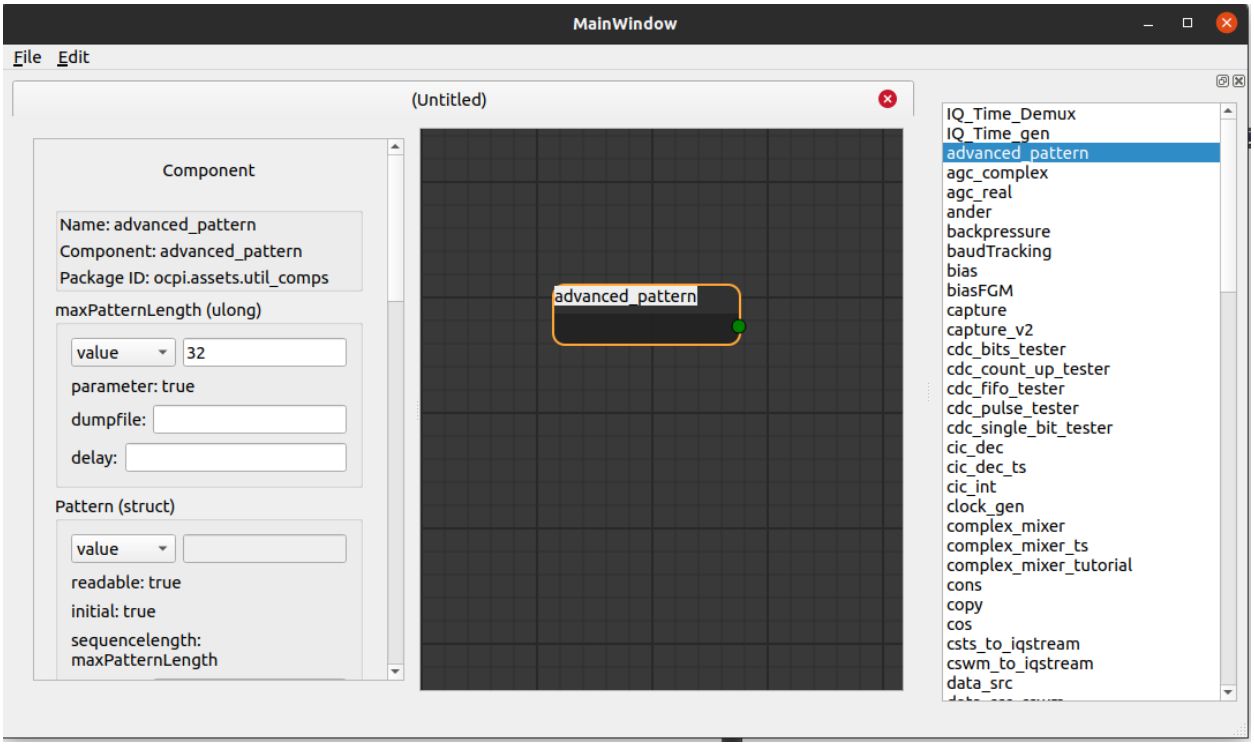

Figure 38: Component with its Side Panel

The side panel allows to change the values of all the properties that a given component has. Keep in mind that not all values need to be filled in. If one is left empty, the Node Graph accounts for that so it allows to change only the properties that want without having to worry about the others.

To open the side panel of a given component, double-click on the component. Double-click off the component to close it. Keep in mind if double-click away from the side panel, the Node Graph remembers any changes made and they will be there the next time double-click to reopen it.

8.5 Connections and Connections Side Panel

This is what a connection and its side panel looks like in the Node Graph:

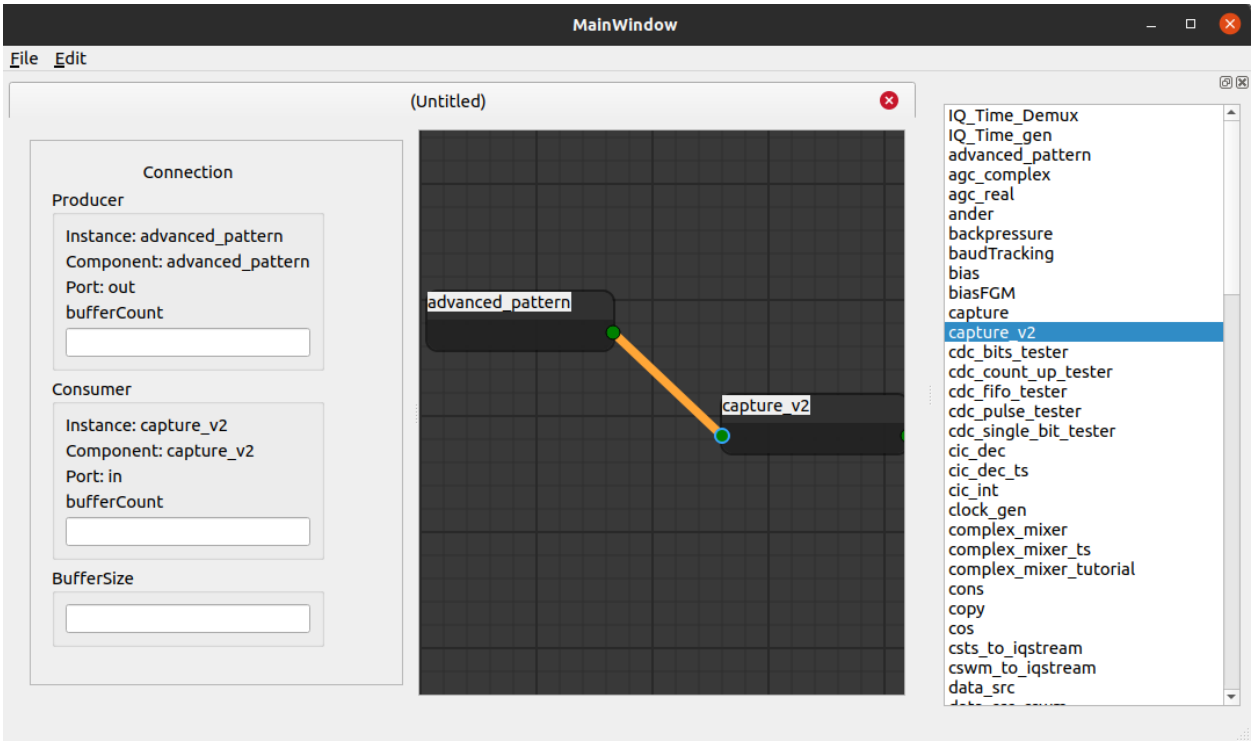

Figure 39: Connection with its Side Panel

The side panel allows to change the values of the ports buffer count on each end as well as the buffer size for the connection. Keep in mind that not all values need to be filled in. If one is left empty, the Node Graph accounts for that so it allows to change only the properties that want to without having to worry about the others.

With connections, an output port (port on the right of the component) can connect to multiple input ports (port on the left of the component), but an import port can only have one connection.

To open the side panel of a given connection, double-click on the connection. Double-click off the connection to close it. Keep in mind that if double-click away from the side panel, the Node Graph remembers any changes made and they will be there the next time double-click to reopen it.

8.6 Edit Menu/Right-click Menu

The Node Graph also has an edit menu with cut, copy, paste, and delete actions that function very similarly to equivalent desktop commands. can cut, copy, paste, and delete connections and nodes with just a few exceptions. The exception is can copy, cut and paste a connection as long as include at least the input port/component in r selection.

To cut, copy paste and delete, right-click on a component to display the pop-up menu or highlight multiple connections and/or components and use the Edit menu.

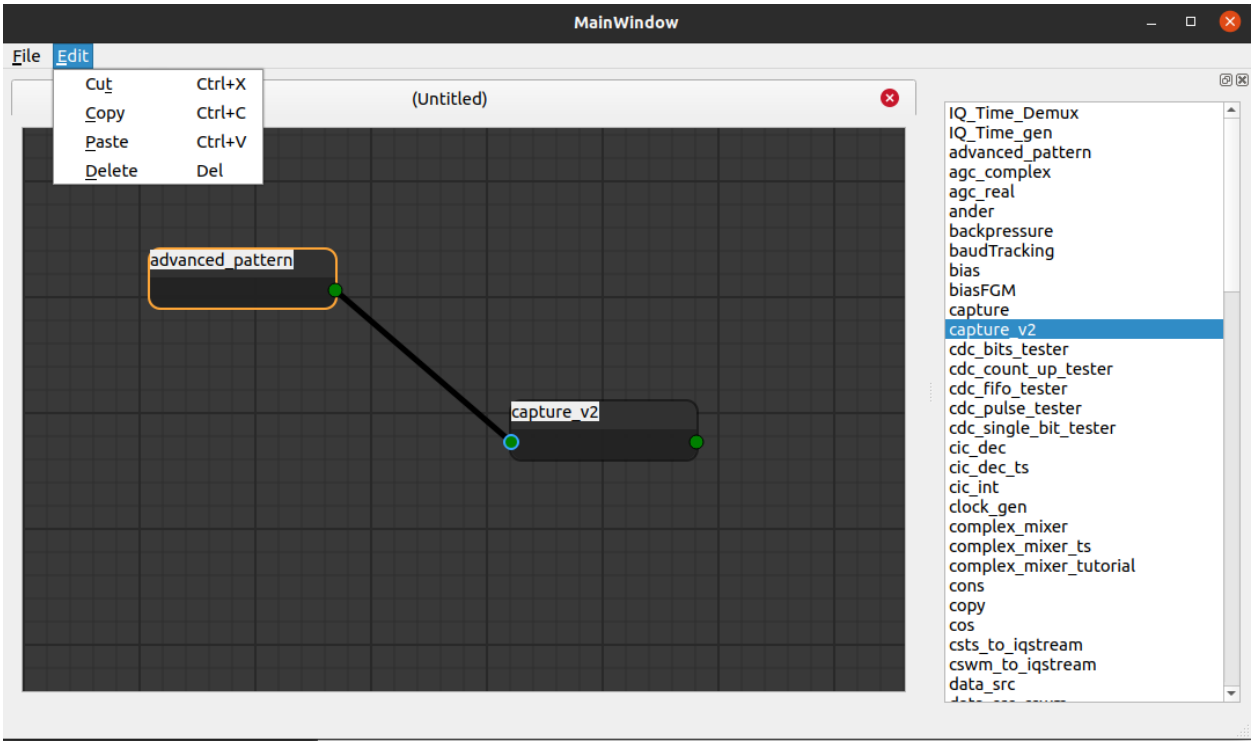

Figure 40: Cut, Copy, Paste, Delete Example

9 Themes Menu

Several color themes have been provided for OpenCPI GUI.

- None: restores the color theme to the default OS theme.
- Ubuntu: provides the default theme for the Ubuntu desktop environment.
- Dark Orange: a dark theme, with splashes of orange highlights.
- Irritable Eel: a blue-green theme for an underwater feel.
- Color Blind: a color palette that should work for most types of color blindness.
- Toxic Sunset: for those with a tolerance for clashing colors.

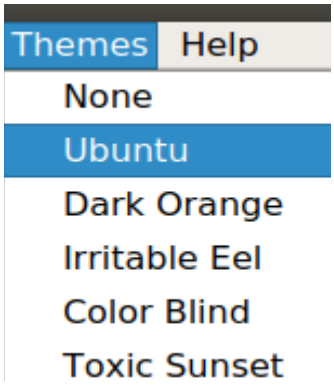

Figure 41: Theme Options

10 Help Menu

The Help menu provides the following options:

- Help Contents: a link to the OpenCPI documentation website where the most recent OpenCPI documentation is located.
- Report Bug/Enhancement: a link to the OpenCPI Gitlab repository, to allow ticket creation of bug reports or feature requests.
- About: the standard license and copyright information about the application.

11 Project Explorer

To access most **`ocpidev`** commands other than those related to creation and testing, select the target in the Project Explorer panel and right-click. A context menu appears with a number of different options (below).

The context-sensitive menu only gives access to commands that the asset right-click on has available. If right click an XML-only application, for example, it only gives the options delete, edit, run, and refresh. Right-clicking a project gives its own options and so on. Below is an example:

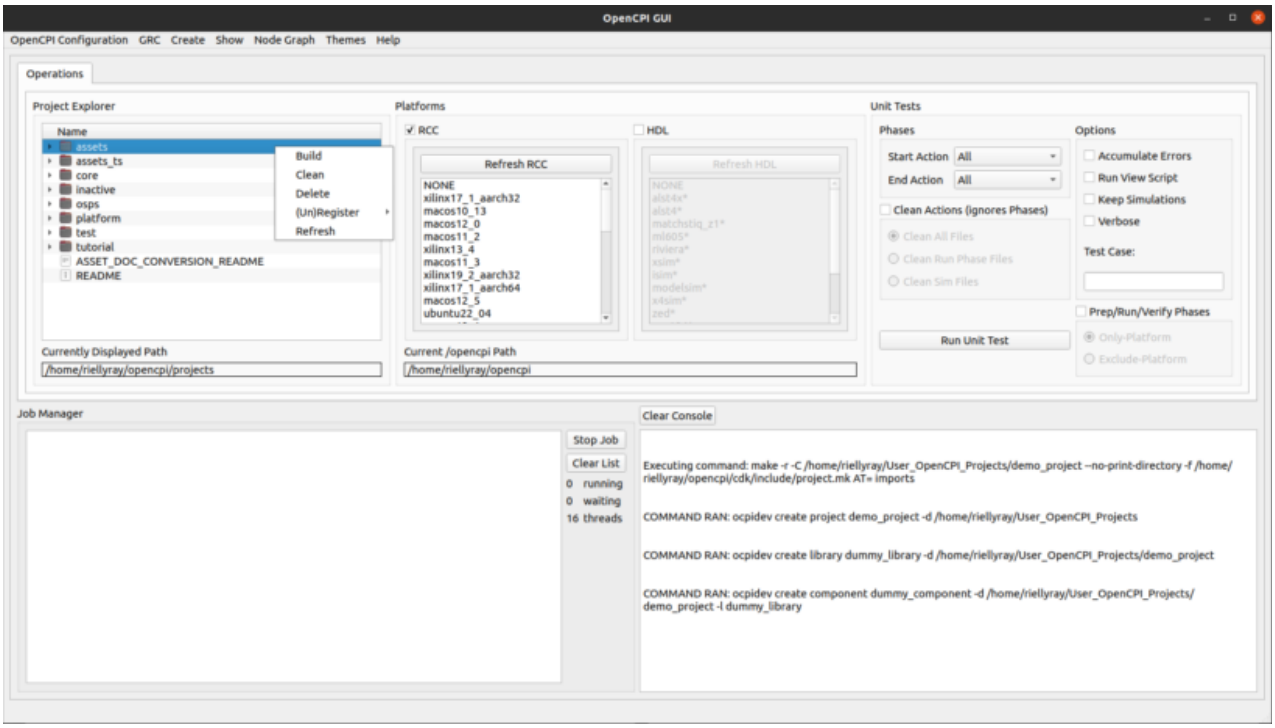

Figure 42: Project Explorer Context Menu

11.1 Build Menu

The Build menu can be used for any asset that can be built: projects, libraries, applications, unit tests, workers, and HDL assets. After clicking on the build action, the appropriate **`ocpidev`** command is called and the results printed to the output console at the bottom of the screen.

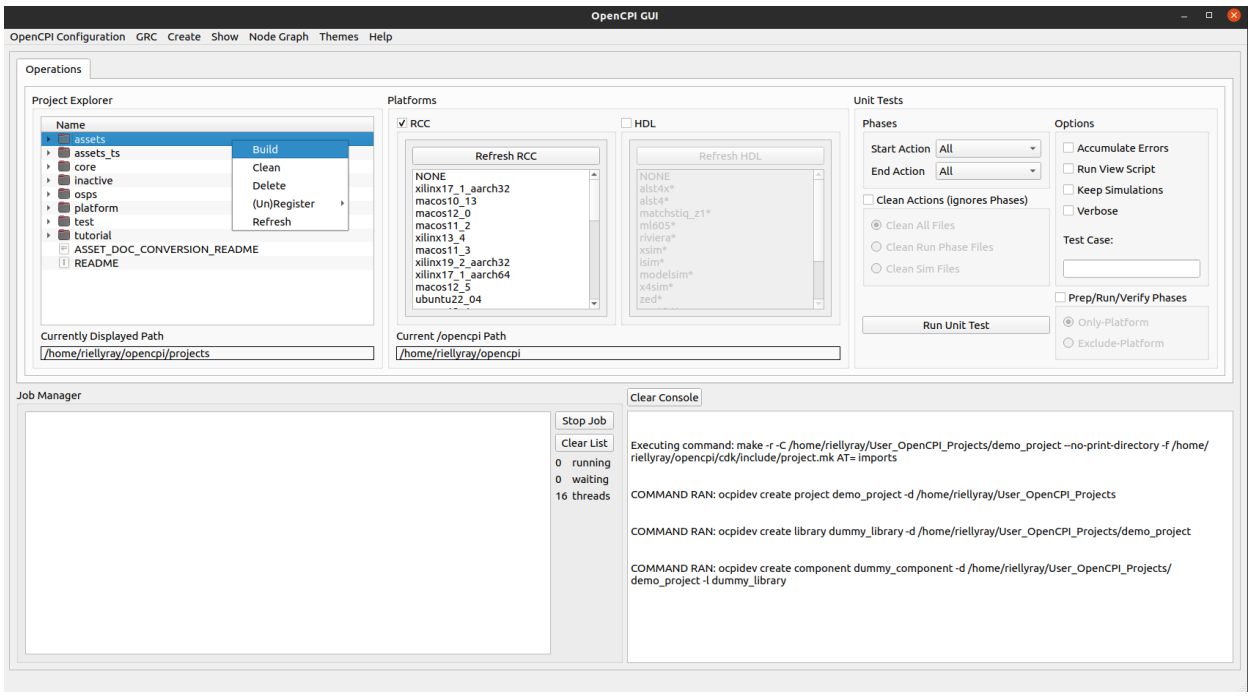

Figure 43: Build Menu

By default, clicking on an asset and running the build command will build the asset for the host OS. To use something other than the OS, can select an RCC or HDL target first (Section 10), then run the build command on the desired Project Explorer asset.

11.2 (Un)Register Menu

The (Un)Register menu option allows the user to register or deregister the selected project. Registering sets the project to the default registry if it wasn't set upon creation while UnRegister removes that association.

11.3 Clean Menu

Much like the Build menu, the Clean menu applies to the same OpenCPI assets, but runs the **`ocpidev clean <asset_type> <asset_name>`** command.

11.4 Delete Menu

To delete OpenCPI assets, select the item and right-click to bring up the Delete menu. A confirmation dialog appears, allowing a chance to decline deletion. This is important because the deletion is irrevocable; the asset is not saved prior to deletion.

11.5 Edit File

This command opens the selected file in r default text editing program. The following file types are available for editing:

- **.xml**
- **.cpp**
- **.txt**
- **.cc**
- **.c**
- **.vhd**
- **.v**
- **.py**
- **.sh**
- **.deps**
- **.mk**
- **.h**
- **.hh**
- **.rst**
- **.exports**
- **Makefile**
- **package-id**
- **workers**

11.6 Refresh

While the Project Explorer is configured to auto-update when the display path is changed or there are changes within the project directory, a manual refresh option is available in case want to use it.

11.7 Run Application

This option allows to execute **`ocpidev run`** on the selected application. The dialog allows to provide optional arguments prior to executing the command, as shown below.

Figure 44: Run Application Dialog

Multiple arguments can be added to a field, separated with a space, as shown below.

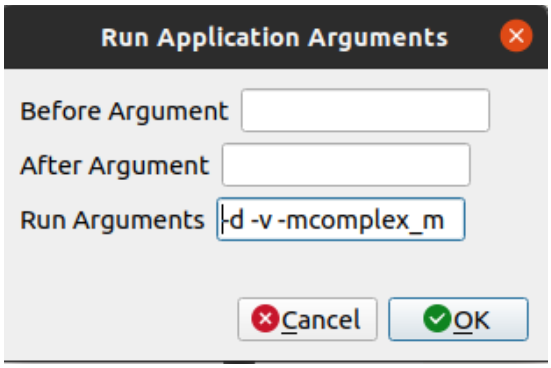

Figure 45: Using Multiple Arguments in Run Application

12 RCC and HDL Platforms

To access RCC or HDL platforms, simply click the checkbox next to the group name (as shown in the figure below). By activating the HDL panel, any selections in the RCC panel are ignored; in other words, the HDL platforms take priority over the RCC platforms, if that is an option for the particular command.

Multiple selection of RCC and HDL platforms is now available. Thus, a single OpenCPI command can target multiple platforms at the same time, rather than having to run a separate command for each target.

If NONE is selected as a platform, regardless of operation, then the default platform is the host OS. This also applies when multi-selecting platforms: including NONE in the item selection will nullify the multiple selection and result in the command targeting the host OS.

Items listed in the RCC and HDL panels are read-only. Attempting to rename the item by double-clicking will not work; the item will show the new name but the change is not saved. This is because the list is dynamically created via an **`ocpidev`** command.

Note: If the RCC/HDL panels are not populated correctly or show an error message, restart the GUI. The problem occurs because of incorrect data about the location of OpenCPI. This problem will affect other parts of the GUI, so a reload should ensure that all parts of the GUI work correctly as well.

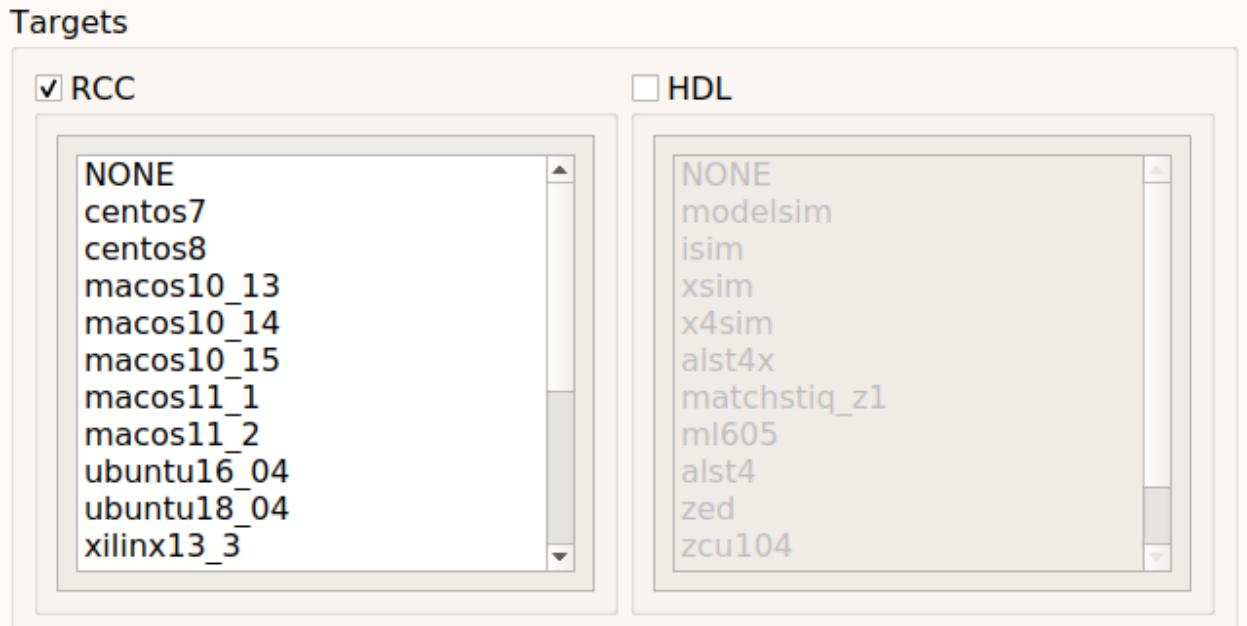

Figure 46: RCC and HDL Platforms

13 Unit Tests

The Unit Tests panel is separated into four sections: Phases, Clean Actions, Options, and Prep/Run/Verify Phases, as shown in the following screenshot.

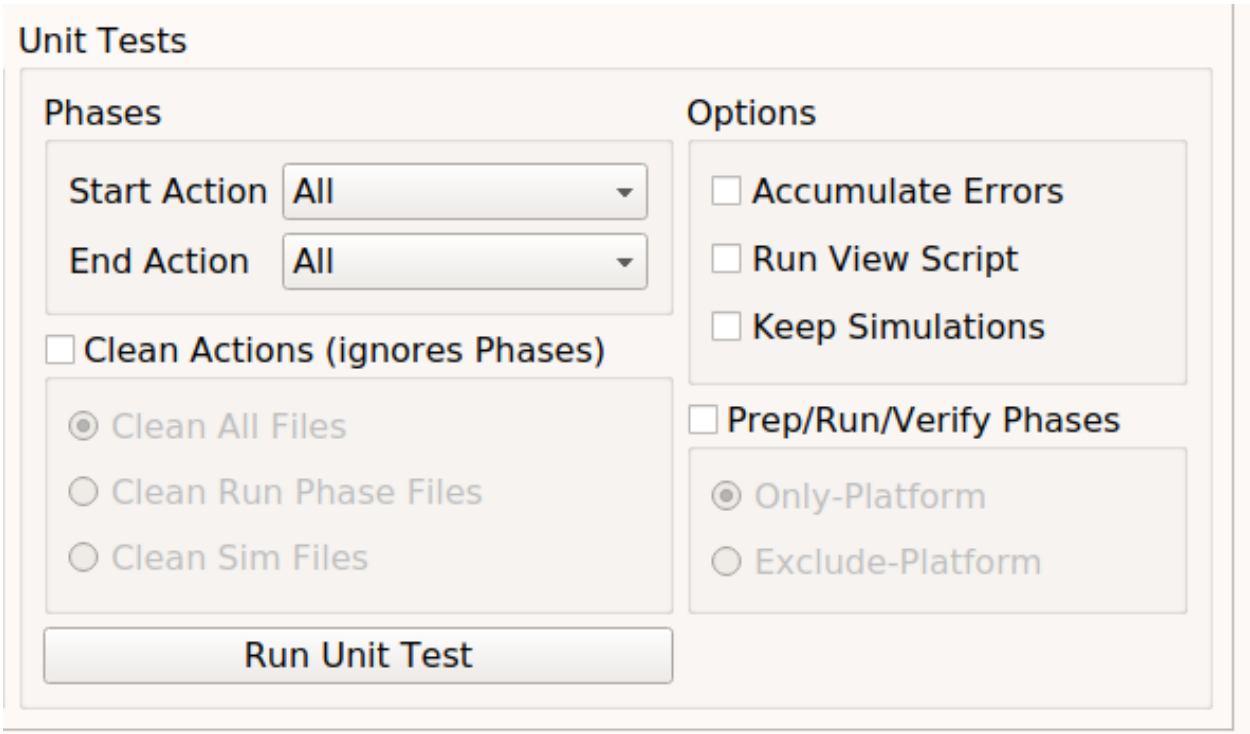

Figure 47: Unit Test Panel

Unit tests can have both RCC and HDL platforms selected as targets. Multiple selection is available for these targets, as shown in the following screenshot.

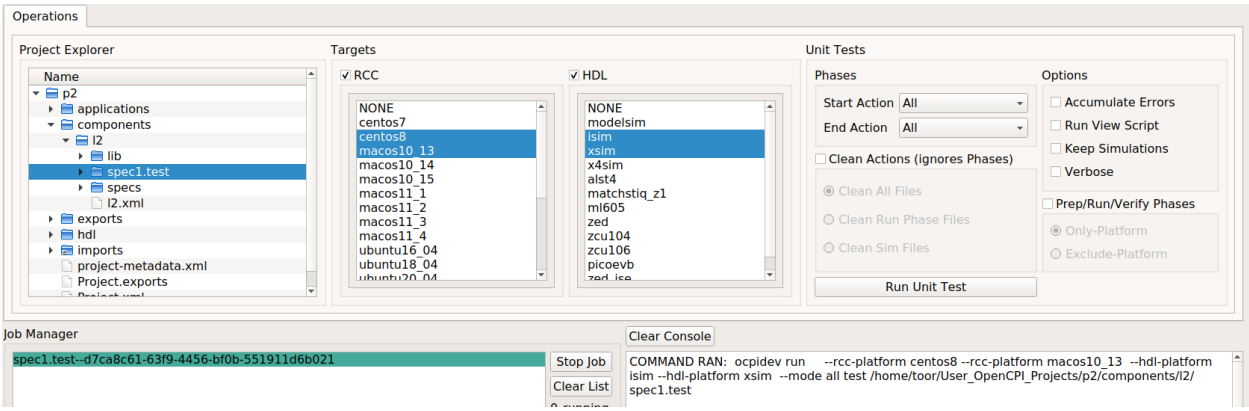

Figure 48: Unit Test Multi-selection

For normal unit tests, if “NONE” is selected as a target, whether individually or in a multiple selection, the unit test defaults to running tests with no RCC or HDL targets.

However, if “Only-Platform” or “Exclude-Platform” options are used, a notification appears, indicating that “NONE” cannot be selected when either of those options are selected.

13.1 Phases

The Phases area consists of the Start and End Actions, which correspond to the five phases available when testing. The Start Action is the initial phase for the unit test while the End Action is the final phase for the test.

While all five phases are available to each action (shown below), only the accepted phase sequences are allowed (shown in a tool tip). Invalid sequences will result in an error. While only two actions are shown, the actual sequences include all additional phases between the start and end actions.

For clarity, the allowed start/end actions are:

- all/all
- generate/generate
- generate/build
- prepare/verify
- prepare/prepare
- prepare/run
- run/run
- verify/verify

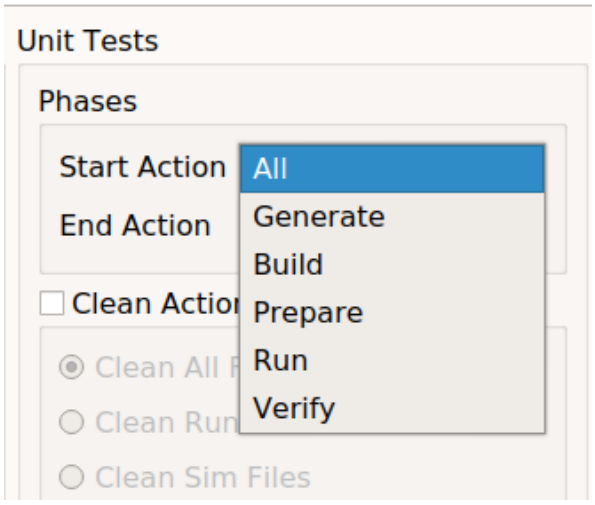

Figure 49: Unit Test Phases

13.2 Clean Actions

Clean actions supersede the phases; activating the clean actions panel will ignore any phases that are selected and only run one of the clean actions. By default, Clean All Files is set to run when the panel is active, but any of the actions can be selected.

The following figure shows the Clean Actions panel.

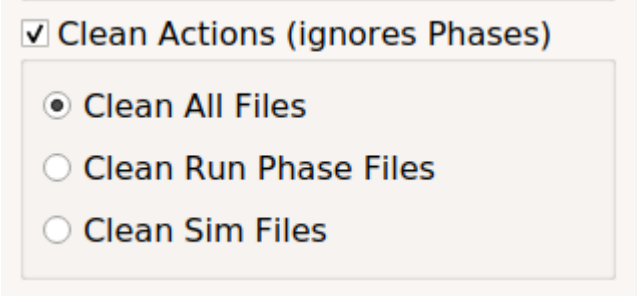

Figure 50: OpenCPI Clean Actions Panel

13.3 Options

The items in the Options panel (below) can be applied to any unit test. Multiple options can be stacked in the same test run.

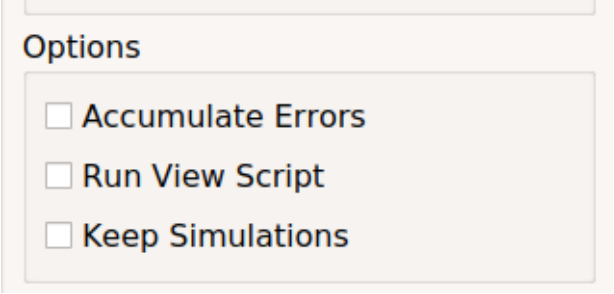

Figure 51: Unit Test Options

13.4 Prepare/Run/Verify Phases

When enabled, this dialog allows the user to select **`only-platform`** or **`exclude-platform`**, which are used as arguments to the unit test execution command. These options only apply to prepare, run, verify, or any combination of those phases.

Both options allow for selection of an RCC, an HDL, or both to be included in the test.

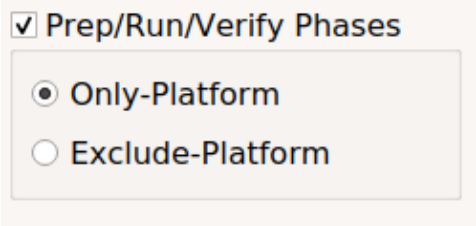

Figure 52: Prepare, Run, Verify-specific Options

14 Job Manager

The Job Manager is a key part of the GUI. It handles multithreading for long-running commands, such as build, unit tests, and install platform. Each process is shown in the manager window and is color-coded based on it's status (see below). When a job is started, its displayed name constitutes the name of the OpenCPI asset plus a unique hash value. This ensures that the Job Manager can identify separate actions on the same asset.

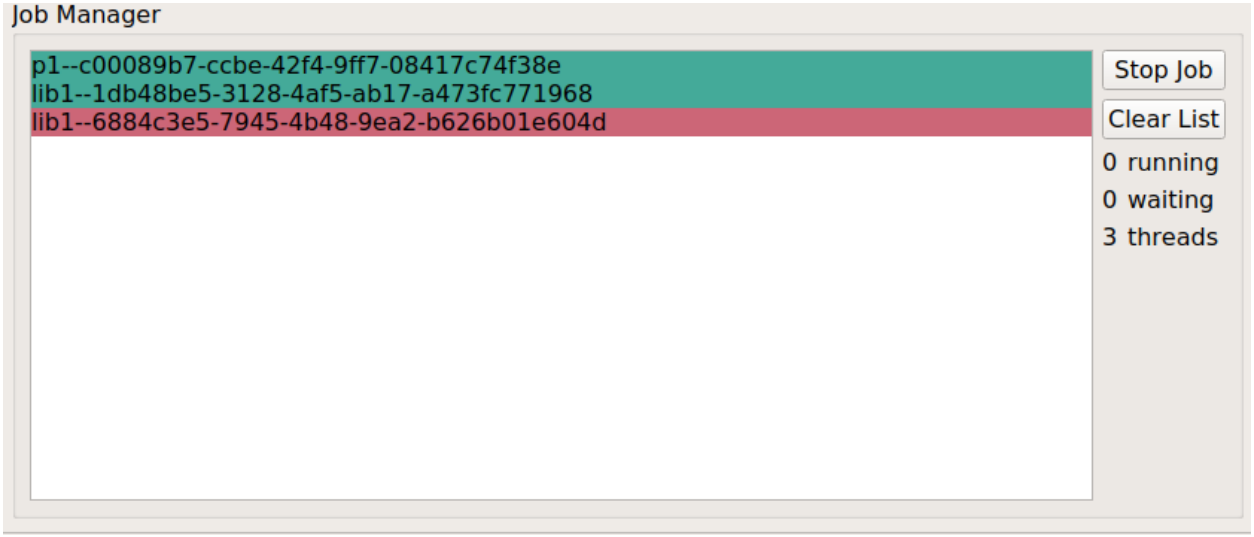

Figure 53: Job Manager Panel

The colors have the following meanings:

- Green: job successfully completed
- Red: job failed with an error message
- Yellow: job is currently running
- Grey: job has been manually stopped

Two buttons are provided to the right of the Job Manager window itself: Stop Job and Clear List. The Stop Job button allows manual stopping of the selected job; the job is completely stopped and not just paused. To re-run the job, it will have to be recreated and launched.

Clear List will clear all finished jobs from the manager window. This includes successful, unsuccessful, and manually stopped jobs but does not affect currently running jobs.

Below the two buttons is a group of thread notifications. The "running" count shows how many multithreaded operations are currently being executed. The "waiting" count

shows how many jobs are enqueued for processing. The “threads” count shows how many threads are available for job processing.

The total number of threads is based on the thread pool available for the local system. The pool is equal to the number of CPUs on the system, thus a higher number of processors means more threads are available.

The Job Manager has a context menu available, shown below.

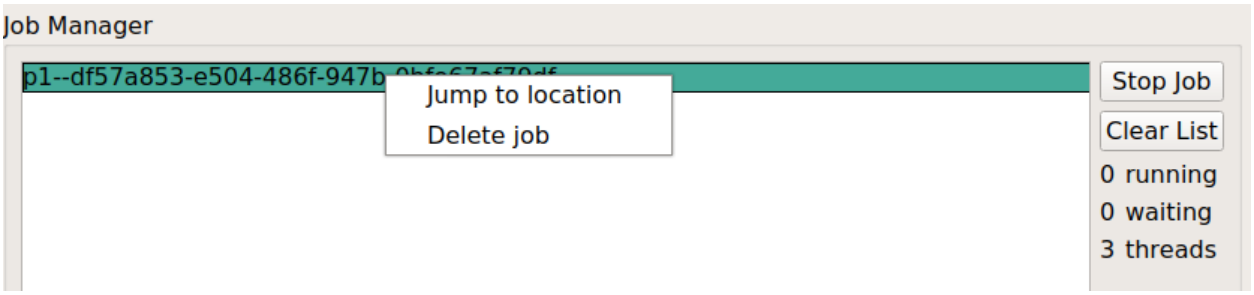

Figure 54: Job Manager Context Menu

This menu provides two capabilities: jump to job output and delete selected job. Jumping to job output scrolls the Output Console to the start of the selected job’s output, highlighting the specific job name for the worker, as shown below.

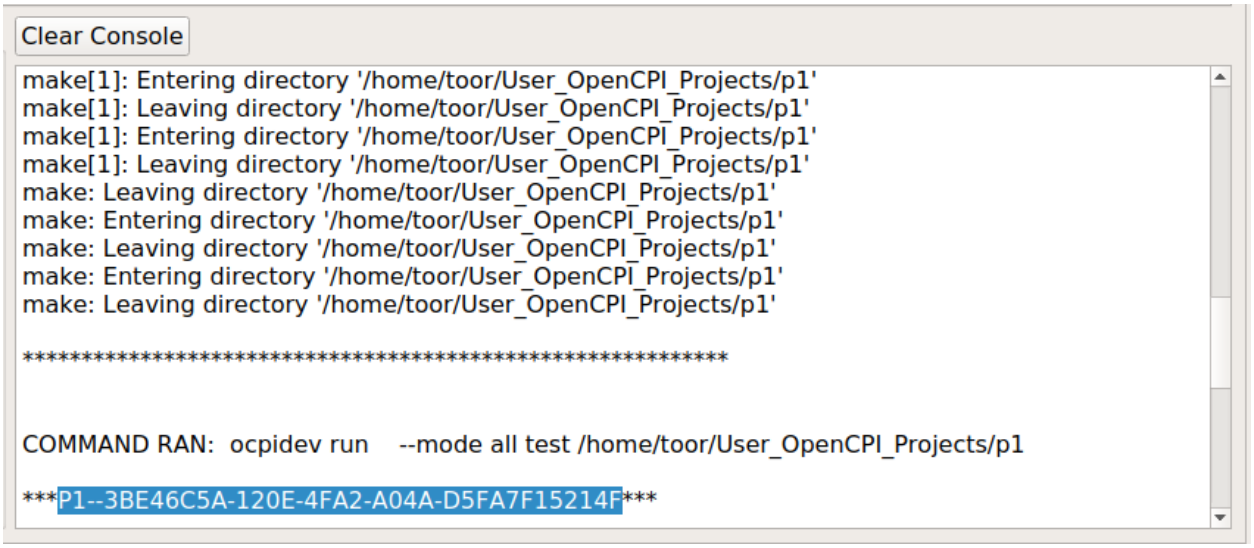

Figure 55: Job Manager "Jump to location" Command

Deleting the selected job removes that particular job from the Job Manager window. This is different from the Clear List button, which removes all jobs from the manager window.

15 Output Console

The output console (shown below) is a redirection of the OS terminal. In other words, what would normally see when running command line **`ocpidev`** will be shown in the GUI console. However, it is output only; it does not accept input commands.

The Clear Console button allows to clear the information in the console as desired. This is most commonly used between operations so that each command's output is all that is displayed. However, if multiple jobs have been executed, using the "Jump to location" right-click option in the Job Manager will scroll the console's content to the start of the selected job's output.

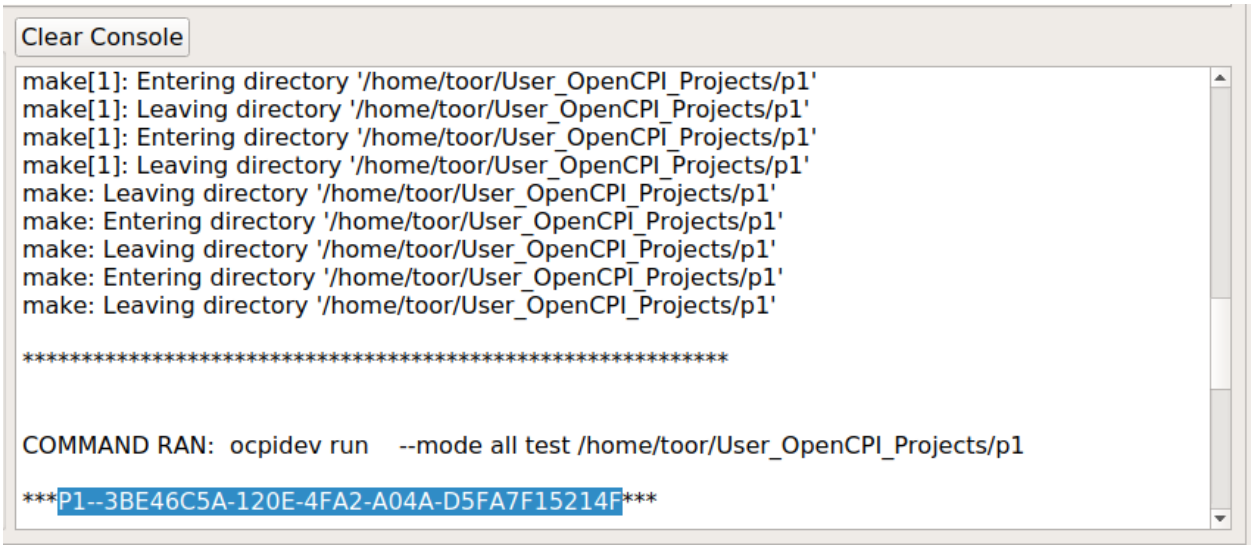

Figure 56: Output Console Example

16 Glossary of Terms

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

16.1 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.

16.2 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***.

17 Functionality Notes

The following sections provide information about particular functionality of the OpenCPI GUI.

17.1 Reusing Project Names

If a project is deleted and a new project with the same name is created, an error appears in the Output Console, as shown below.

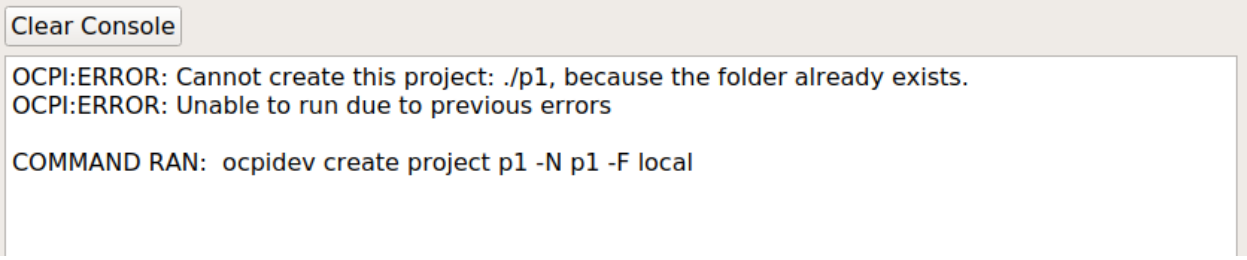

Figure 57: Project Name Reuse Error

This error is erroneous, as the project is still created. Certain metadata is maintained in the file system when projects are created and deleted, leading to some confusion within the GUI about the current status of a project.

While project name reuse can be performed, to avoid confusion with the error message, it is recommended to avoid reusing project names.

17.2 Editing Files

As noted previously, certain files can be edited via the GUI. Editing is provided via the Linux **`xdg-open`** command, meaning whichever editing program has been set in the OS for that particular file type will be opened. Most commonly, this will be the default text editor for the OS, but if the user has changed the application associated with a particular file type, it may be different.

17.3 Unit Tests and HDL Targets

Unit tests will fail if the HDL panel under Targets is checked but the HDL target is set to NONE, as this will cause an empty **`--hdl-platform`** argument to be passed to the **`ocpidev run`** command. Therefore, if an HDL target needs to be used, it is best to uncheck the HDL panel under Targets.

17.4 RCC Workers

There is a known issue with OpenCPI regarding RCC workers and C++ files. When using the command

```
ocpidev create worker <name>.rcc -L c++ -S <spec>
```

the following error may occur

```
Error: Invalid language "c++" for model "rcc". Available: "c"
"c++"
```

Depending on system configuration, this has been found to occur in both the GUI and CLI execution. This will be addressed in a future OpenCPI update.

A workaround for this is to remove the **`-L c++`** portion of the command, as the default functionality is to create C++ files:

```
ocpidev create worker <name>.rcc -S <spec>
```