



OpenCPI - Open Component Portability Infrastructure

**An Open Source Framework for
Heterogeneous/Embedded Component-
Based Systems and Applications**

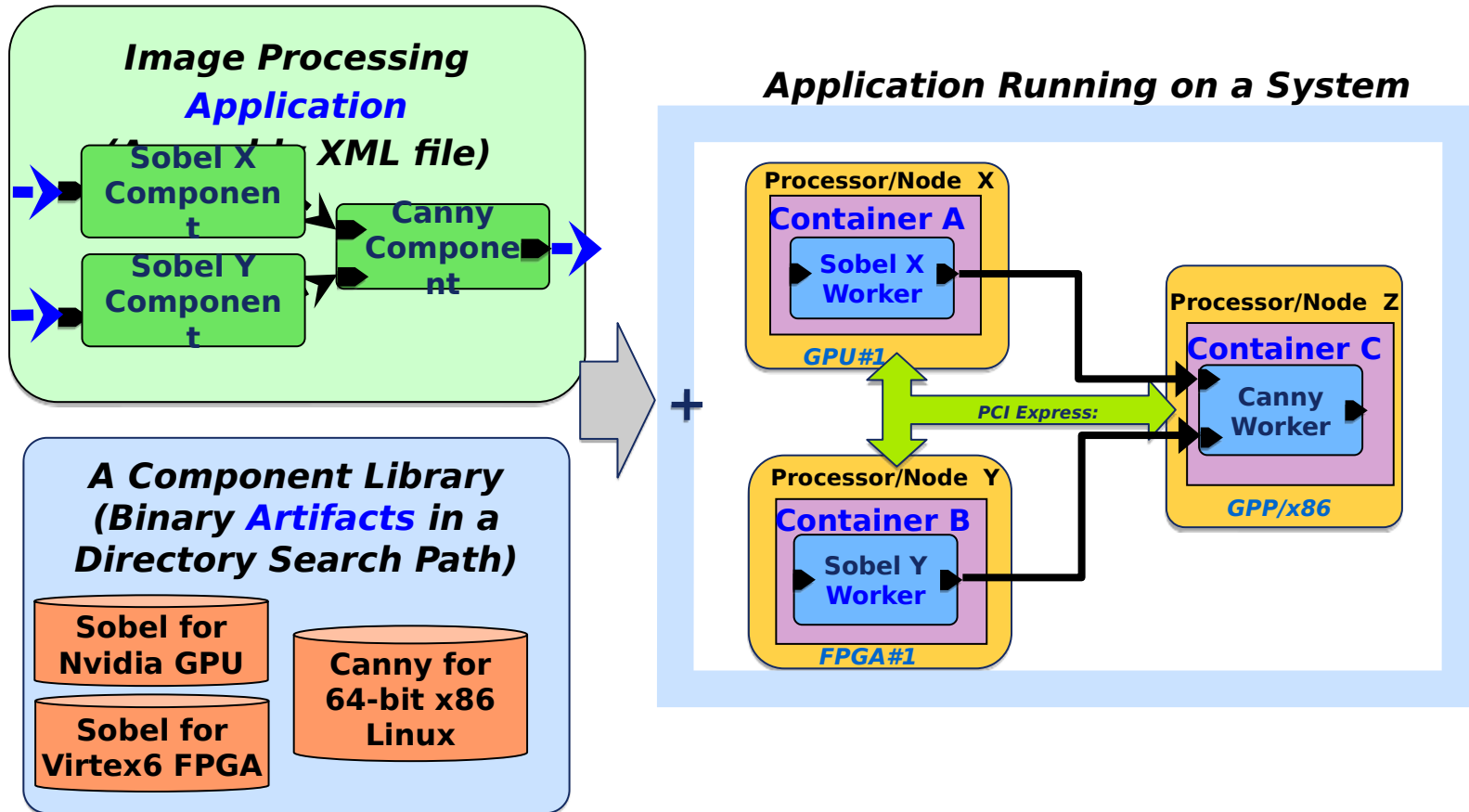
**Introduction to OpenCPI Development
for
Application and Component
Developers**

- This briefing describes OpenCPI from the *development perspective*. It:
 - Describes how the elements needed for execution of an OpenCPI application are created.
 - Defines OpenCPI terminology and concepts that apply to application and component development.
 - Introduces tools used for development.
- This briefing is introductory, not comprehensive.
- A prerequisite for this briefing is the previous “Introduction to OpenCPI” briefing.

- **Development** is producing things needed for execution.
- The previous briefing defined things *needed* for execution of an **application**. These are:
 1. The application itself, defined as an assembly XML file that describes an assembly of **components**.
 2. Artifact libraries that consist of (binary/executable) **artifacts** compiled from **workers** that implement requested components.
 3. Runtime environments called **containers** that can load and run the artifacts (and thus workers) needed by the application.

Needed: Application XML+Artifacts+Containers

- Execution requires: application assembly + artifact libraries + containers



Who creates these things and how?

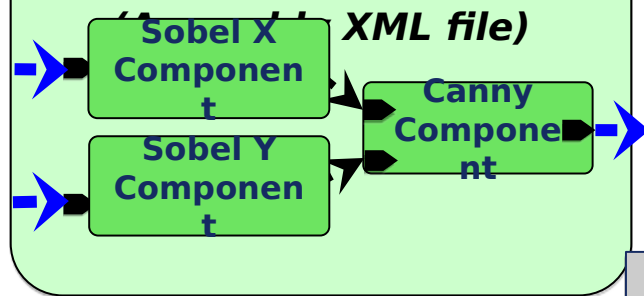
- Different developer types/roles contribute to what's needed for application execution:
 - **Application** developers create **applications** that specify **components** and use **artifacts** to execute on **containers**.
 - **Component** developers create **components**, **workers** and **artifacts** for use by application developers.
 - **Platform** developers create the lower level drivers and configurations that support the **containers** (runtime environments on processors) in a system.
- OpenCPI provides a reference manual for each developer type.

Developer Roles in Providing What's Needed

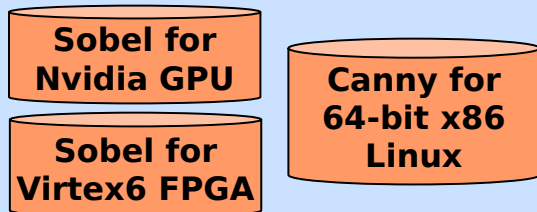
Application Developers

provide application XML

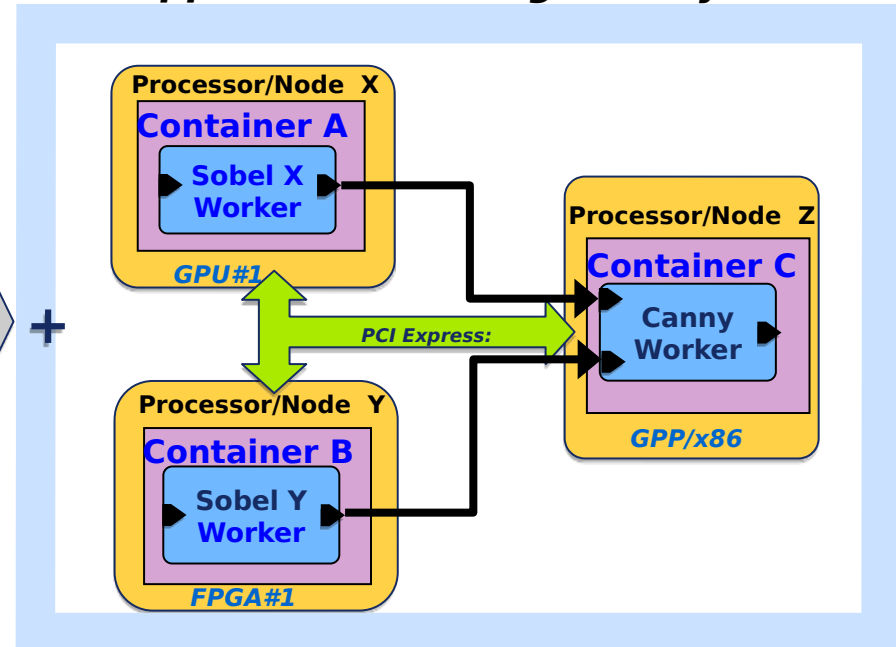
Image Processing Application



A Component Library (Binary Artifacts in a Directory Search Path)



Application Running on a System



Platform Developers provide containers

Component Developers
provide components, workers,
artifacts

Developers

- **Applications:** are defined as an:
 - **Assembly:** of configured and connected components
- **Components:** are defined functions which have:
 - **Properties:** that configure and control the component, and
 - **Ports:** where data messages are sent and received, and which support:
 - **Protocols:** which define the allowed/expected messages
- **Workers:** are implementations (source code) of components
- **Artifacts:** are binary executables compiled from one or more workers
- **Containers:** are runtime environments for executing workers
- **Asset:** a general term for anything developed for OpenCPI

OpenCPI Project: Workspace for Developers

- The term **asset** is used for things created by all (3) types of developers.
 - applications, components, workers, artifacts, protocols etc.
- The term **project** is used for an OpenCPI workspace (in a directory/folder) where all types of developers create assets.
 - My project can be declared to depend on your project:
 - I can create an application in my project that depends on components and workers in your project.
 - One project can contain all types of assets.
 - One project can contain multiple assets of any type.
- The work product of developers is **projects** containing **assets**.
- **Projects** are in the development environment:
 - They do not exist in a runtime-only (i.e. deployed) environment.

- The general term for things that are developed in projects:
 - components, protocols, workers, applications, and many more (but not artifacts; they are not assets)
- Most assets are defined by an XML file
- Most asset types have their own directory
 - So the XML file lives there
 - Other files for the asset also live there, e.g. source code
- Things a developer does with assets
 - Create and destroy
 - Edit/modify
 - Build and clean (for some types, like workers)
 - Run (for some types, like applications and unit tests)
- Application developers generally
 - Rely on component developers to supply components and artifacts
 - Read (but do not create) component specifications (XML)
 - Create application assembly XML

What is XML?

- For those not familiar
- XML stands for e**X**tensible **M**arkup **L**anguage
- A structured type of text file containing a hierarchy of **elements**
- So the file is an **element**, and that element can have **child elements**
- Each element has a type identifier and named **attributes** w/values
- Elements look like this (e.g. for a protocol, consisting of operations)

```
<protocol>  
  <operation name='op1' />  
  <operation name='op2' />  
</protocol>
```

- So the file has a **protocol** element with **operation** child elements
 - The **operation** elements each have a name attribute
 - An element that has no children can end in: **/>**
 - An element that has children ends in: **</type>**

- The **ocpidev** command-line tool performs operations on assets:
 - Operations (verbs) are: create, delete, build, clean, run, show
 - Except for projects, all operations are on assets in projects
 - Its syntax is:
 - **ocpidev** [*<options>...*] *<verb>* *<asset-type>* *<name>*
 - E.g.: **ocpidev create application myapp**
 - Compilers and FPGA tools are used by **ocpidev** for building
- Asset XML and source code are modified using text editors
- There is also the **OpenCPI GUI** asset development tool:
 - It also performs operations on assets, but with fewer options
 - It calls/uses **ocpidev** to get everything done

OpenCPI Artifacts: OpenCPI's “binaries”



- Artifacts are the result of building workers, by component developers
- They are *not* “assets”, they are the *result* of building assets
- They may or may not be in projects
 - they are initially created in projects
- They are files containing executable code for one or more workers
 - Targeting a specific execution platform/processor, including FPGAs.
 - Loaded into a container to execute those workers for an application.
- They have embedded OpenCPI metadata
 - The metadata describes the workers that were compiled to make them
- For software workers they are roughly like .so or .dll files
- For FPGA workers they are roughly like “bitstream” files
- When an application is run, artifacts are “sought” by looking in directories specified by `OCPI_LIBRARY_PATH`
 - Analogous to `PATH` or `LD_LIBRARY_PATH` on Linux

- Component, Protocol and Application assets are the main building blocks for application development.
- All of these assets are XML.
- Components (a.k.a. “specs” or “component specs”)
 - What an application developer **reads** to know how to use components
 - The building blocks for applications
- Protocols, used by ports in component specs.
 - Describes what messages a component sends or receives on a port
 - Applications connect ports (of components) whose protocols match
- Applications (the “assembly of connected components”)
 - Describes which components should be used
 - How they are configured (via properties)
 - How they are connected (via ports)

Asset Type: Component

Term: *Component*

Definition: A specific function used when composing applications and a “contract” for workers (implementations)

Described by: an OpenCPI Component Specification (**OCS**) XML file
a.k.a. “component spec”

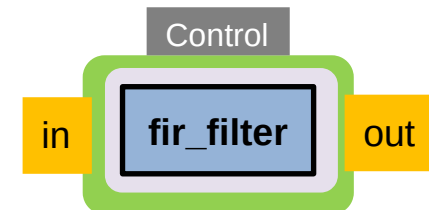
Example: FIR Filter

Features of Components

- **Property** – configurable value used to control the component's operation
 - Controlled at runtime, via:
variable in C/C++, register in VHDL
 - or--
 - Controlled at build time, via:
static const in C/C++
generic in VHDL
- **Port** – an interface used to communicate with other components
 - **protocol** attribute indicates protocol
 - **producer** attribute indicates direction

Example OCS

```
<ComponentSpec>
  <Property Name='numberOfTaps'
    Parameter='true' />
  <Property Name='taps'
    ArrayLength='numberOfTaps' />
  <Port Name='in' Producer='false'
    Protocol='iq_with_time' />
  <Port Name='out' Producer='true'
    Protocol='iq_with_time' />
</ComponentSpec>
```



Asset Type: Protocol

Term: *Protocol*

Definition: Description of a set of messages that may flow between the ports of components

Described by: an **O**penCPI **P**rotocol **S**pecification (**OPS**) XML file
a.k.a. “protocol spec”

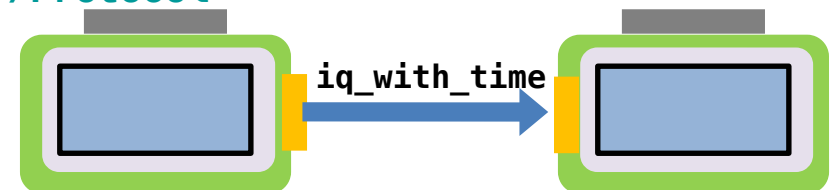
Example: iq_with_time

Features of Protocols

- **Operation** elements describe the messages that may be used
- **Argument** elements describe the individual fields of a message
- The protocol is specified in the Port element of a component spec (OCS)

Example Protocol Specification XML

```
<Protocol>
  <Operation name='iq'>
    <Argument name='data' type='struct'>
      <member name='i' type='short'>/>
      <member Name='q' type='short'>/>
    </Argument>
  </Operation>
  <Operation name='time'>
    <Argument name='time'
      type='ulonglong'>/>
  </Operation>
</Protocol>
```



Asset Type: Application

Term: *Application*

Definition: Connected/configured assembly of components

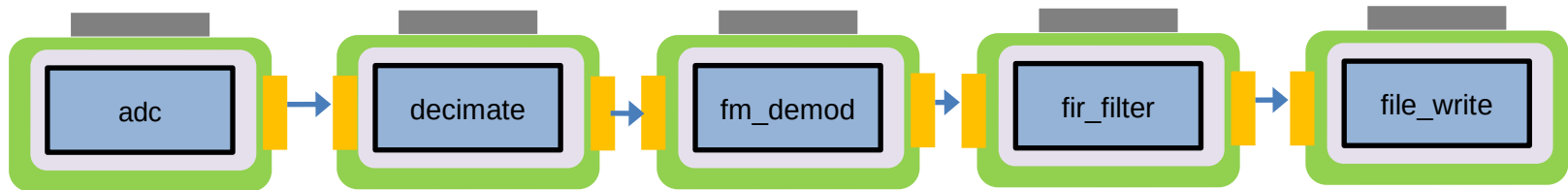
Described by: an OpenCPI Application Specification (OAS) XML file,

a.k.a. “App XML”

Features of Applications:

- Instances of components are indicated by **instance** elements
- Instances have **property** child elements to “configure the instance”
- Instances are connected in various ways, usually with the **connect** attribute

Application



Application Specification XML

```
<Application>
  <Instance component='adc'           connect='decimate' />
  <Instance component='decimate'       connect='fm_demod' />
  <Instance component='fm_demod'       connect='fir_filter' />
  <Instance component='fir_filter'     connect='file_write' />
  <Instance component='file_write' />
</Application>
```

Developers

- **Application developers** create the application assembly XML that uses components and artifacts created by **component developers**, which run on containers created by **platform developers**.
- An application is an OpenCPI **asset** developed in an OpenCPI **project**.
- An application is developed with the OpenCPI **ocpidev** command line tool and/or the **OpenCPI GUI**.
- **Applications** use **components** which in turn use **protocols**. All of these assets are described in XML files.
- A component may have **properties** to control its operation and **ports** for communicating with other components. A port identifies its **protocol** and the direction of data flow.
- A protocol asset has **operations** to identify the types of messages that may be used and **arguments** to describe individual message fields.
- An application has **instances** to specify component instances and **property values** for configuring the instance. Instances can be connected.
- An **artifact** is the result of building a **worker**, which is an implementation of a component. An artifact is not an asset.

- **Component development** creates the assets and artifacts needed to define and run **applications**, and encompasses:
 - Specifying the **component** and **protocol** assets (in XML files)
 - Creating **workers** that implement the **component** and **protocol** specs
 - Building **workers** into artifacts (using authored source code)
- **Workers** are assets with their own directory, XML and source code
 - The XML refers to the component spec and describes implementation details
 - The source code, in a language like C++ or VHDL, implements the spec
- Components/protocols/workers are created in **component Libraries**
 - A directory structure holding components, protocols and workers
- **Libraries/components/protocols/workers** created/deleted by **ocpidev**
- **Component libraries** and **workers** are *built* by **ocpidev**
- All these assets are created in OpenCPI **projects**.

Term: *Worker*

Definition: A concrete implementation of a component, with source code

Described by: OpenCPI Worker Description XML (**OWD**), in a worker's directory

Features of Workers

- ♦ Worker name suffix indicates *authoring model* — the API and language for the source code
 - `<worker>.rcc` for C/C++
 - `<worker>.hdl` for VHDL
- ♦ Worker description XML (**OWD**) indicates
 - Which component spec (OCS) is implemented
 - Which language is used for source code
 - May contain additional properties & port information to expand or refine OCS

Example:

One **OCS** $\Rightarrow$ Two Workers

- `fir_filter.hdl/`
OWD file: `./fir_filter.xml`

```
<HdlWorker spec='fir-spec'
           language='vhdl' />
```


Source file:
`./fir_filter.vhd`
- `fir_filter.rcc/`
OWD file: `./fir_filter.xml`

```
<RccWorker spec='fir-spec'
           language='c++' />
```


Source file: `./fir_filter.cc`

Asset Type: Component Library

Term: *Library*

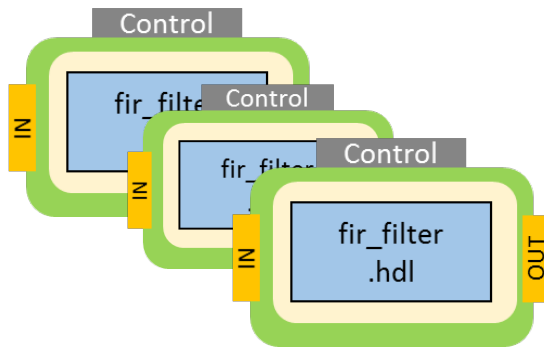
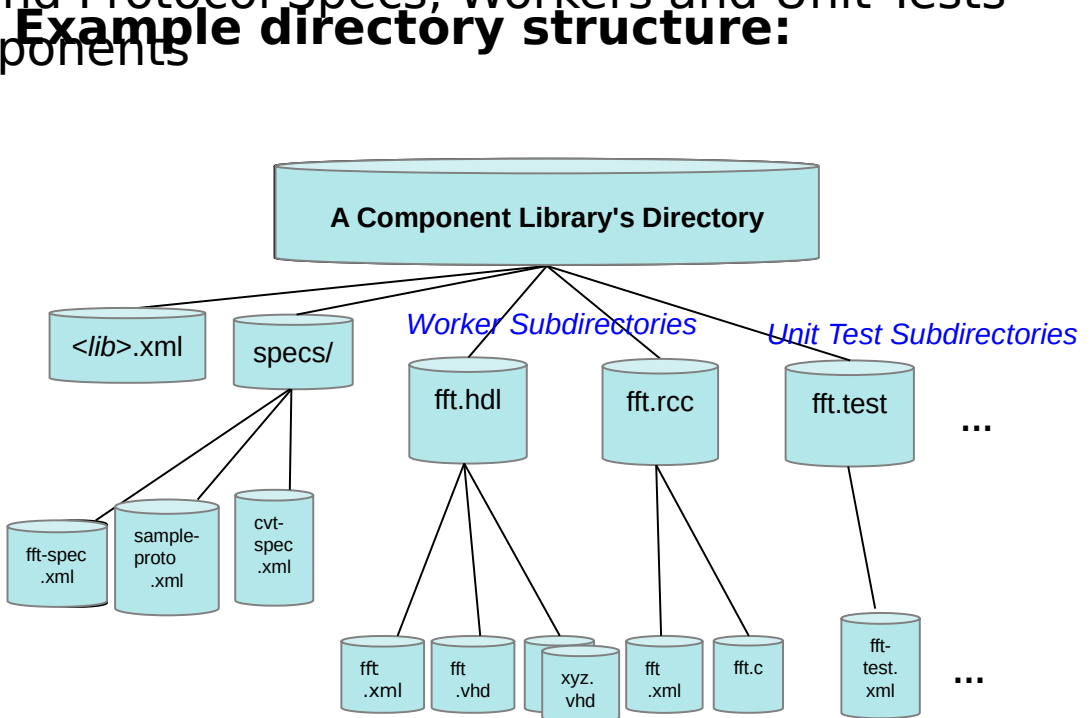
Definition: A related set of components in a single directory in a project

Described by: Directory structure and Makefile, no XML for the library

Contains: Component and Protocol Specs, Workers and Unit Tests

Example of "Utility" Components

- Provide a logical grouping
- Reduces clutter
- Encourages reusability
- Built time search rules



Asset Type: HDL Assembly

Term: *HDL Assembly*

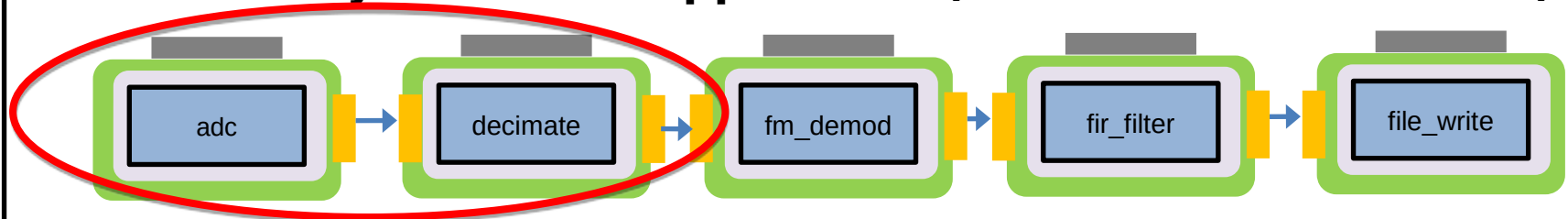
Definition: A group of pre-connected HDL/FPGA workers built as an artifact.

intended to be used as a FPGA-based subset of an application

Features of HDL Assemblies:

- Instances of *workers* are indicated by **instance** elements
- Instances may have **property** child elements for build-time property values
- Instances are connected in various ways, usually with the **connect** attribute

HDL Assembly: subset of application (subset indicated in red)



HDL Assembly Description XML

```
<HdlAssembly>
  <Instance worker='adc'           connect='decimate' />
  <Instance worker='decimate'      externals='true' />
</HdlAssembly>
```

- After reading this briefing, you should be familiar with the terms, concepts, tools and assets related to introductory application and component development.
- You can try out what you've learned here by following Tutorial 1, "Introduction to Component Development".
 - One version is for command-line tool usage only
 - Another is for GUI usage.